EKONOMI OCH SAMHÄLLE

Skrifter utgivna vid Svenska handelshögskolan
Publications of the Swedish School of Economics
and Business Administration

Nr 151

PATRIK PAETAU

ON THE BENEFITS AND PROBLEMS OF THE OBJECT-ORIENTED PARADIGM INCLUDING A FINNISH STUDY

Helsingfors 2005

On the Benefits and Problems of the Object-Oriented Paradigm including a Finnish Study

Key words: Object-oriented paradigm, benefits with object-oriented paradigm, problems with object-oriented paradigm, software components, reuse

Patrik Paetau
Swedish School of Economics and Business Administration
Department of Management and Organization (Information Systems Science)
P.O.Box 287
65101 Vaasa, Finland

Distributor:

Library
Swedish School of Economics and Business Administration
P.O.Box 479
00101 Helsinki, Finland

Telephone: +358-9-431 33 376, +358-9-431 33 265
Fax: +358-9-431 33 425
E-mail: publ@hanken.fi
http://www.hanken.fi

ISBN 951-555-893-X (printed)
ISBN 951-555-894-8 (PDF)
ISSN 0424-7256

Edita Prima Ltd, Helsingfors 2005

For my son Robin

ACKNOWLEDGEMENTS

Although there may still be much to discover and understand as far as benefits and problems with the object-oriented paradigm are concerned, the writing of this dissertation is now, after several years, coming to an end. The opportunity has therefore come to look back and give due credit to those who have supported and encouraged me in different ways.

My first interest in the object-oriented world began in 1989 when my former manager at Tietotehdas Oy (Tietoenator nowadays) Vice President Tiina Kurki asked me if I wanted to get to know what objects actually are. As a young systems analyst working in a project that was developing a large money market information system for the Union bank of Finland (Nordea Oy nowadays) I was eager to start with something new.

After my time at Tietotehdas Oy, which I found to be an interesting and beneficial experience, I moved to the Swedish School of Economics and Business Administration in Helsinki where Associate Professor Pertti Lounamaa and my colleague Dr. Leif Andersson encouraged me to write a Licentiate thesis concerning the object-oriented paradigm and Activity Based Costing. This Licentiate thesis consisted of an Activity-Based Costing application that I programmed using C++ and the object-oriented database ObjectStore.

After completing the Licentiate thesis in 1995, I took my first steps on the pure object-oriented road. I would like to thank Professor Bo Sundgren at the Stockholm School of Economics in Sweden, Professor Sture Hägglund at Linköpings universitet, Sweden and especially Associate Professor Inger Eriksson at the Swedish School of Economics and Business Administration in Finland who gave me initial advice on how to continue on the sometimes rather challenging road of the object-oriented paradigm.

My supervisors Juhani Iivari at the University of Oulu in Finland and Professor Wita Wojtkowski at Boise State University in Idaho, US combined constructive comments and encouragement to persist with the process. At the Kilpisjärvi Information Systems Research Seminar on April 17, 1997 Professor Markku Nurminen from the University of Turku, Finland and Professor Pertti Kerola from the University of Oulu, Finland, also gave me some helpful comments on how to continue with the study.

I would like to thank the Hanken Foundation at the Swedish School of Economics and Business Administration and especially the Rector Professor Marianne Stenius for helping me financially through some difficult times in recent years.

Without the cooperation of all the Finnish software companies that took part in the survey and especially the companies that participated in the case studies, this dissertation would not have been possible. With this in mind, I would like to thank the companies and their employees who kindly agreed to be interviewed.

Professor Bo-Christer Björk has acted as my supervisor during the later stages of this dissertation and I would like to express my paramount appreciation to him for all the support and encouragement that he has shown me and for all his priceless comments.

I would also give my best thanks to Senior Lecturer Dino Cascarino for the high-quality language proof of my manuscript. Further, I would like to thank Professor Anssi Öörni who proof read the Finnish survey instrument.

Furthermore my gratitude extends to everyone who offered comments and help on the object-oriented road: Professor Eero Vaara, Professor Guy Ahonen, Professor Johan Fellman, Professor Gunnar Rosenqvist, Senior Lecturer, Susanna Taimitarha, Researcher Turid Hedlund, Docent Juha-Pekka Tolvanen and Research Professor Tarja Tiainen. I wish to particularly acknowledge Dr. André Sahlström, with whom I shared many ups and downs when working with this dissertation.

Finally my warmest thoughts go to those who are closest to me: my family and friends; my wife Solveig, my sons Robin and David, my mother Gunvor, my grandmother Astrid, my cousin Fredrik Sjöberg, my friends Melker Olenius and Georg Wrede and many more.

> 'I have a cat named Trash … If I were trying to sell him (at least to a computer scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented'.
>
> Roger King (My Cat is Object-Oriented)

Patrik Paetau

Vaasa, August 2005

CONTENTS

APPENDICES

FIGURES

TABLES

1 INTRODUCTION

Today information systems tend to be much larger and more complex than before, due to more powerful computers, the increased expectations from end users and use of the Internet for exchanging all kinds of information (Jacobson et al., 1999). The functionality of information systems today is also more often based on distributed computing, on graphical applications and on multimedia systems (Nerson, 1992). In contrast to the constant decreases in hardware costs, the costs of software development and maintenance have not decreased over time.

This development has lead to an interest in new information system development paradigms. Yourdon & Argila (1996, pp. 4-5) propose that the applications and software systems of tomorrow will be so large and complicated that conventional software development techniques, which depend on programmers and software developers developing all programming code from scratch, will be inherently imperfect. According to Yourdon & Argila there are not enough people, nor money to build the large and complex applications and software systems of the future, with the software development techniques of today. Despite the new methods that are available, most people still use the same information system development methods that have been in use for as long as 25 years (Jacobson et al., 1999).

The interest in new information system development methods has been triggered by the need of system developers for making work in development easier and more productive. Therefore, an information system development method or paradigm to initiate this is needed.

The growth in size and complexity of information systems and the resulting increase in software development costs have led to a greater market demand for new software development paradigms, methods and techniques that would make software development easier, cheaper and more efficient. Already in the 1980's the object-oriented paradigm was considered as a possible answer to this dilemma, and today it is still viewed by many as the best available solution to the "software crisis" (Johnson, 2002). Nowadays one can also point towards possible solutions such as outsourcing part of the work in software development to developing countries like India, the Philippines, Vietnam, Indonesia and China. This is, however, only a partial answer to the costs involved, and does not therefore address the real problems.

According to Johnson (1997a) the object-oriented paradigm is becoming the industry standard for software development. One can therefore argue that using it has been a major means in handling the software crisis (Johnson, 2002).

The idea of object orientation has been a part of our history for more than 2000 years. Already the philosophers Aristotle and Plato wrote about such things as objects, classes, subclasses, associations and object behaviour. Later Augustine, Duns Scotus, Bertrand Russell and Alfred North Whitehead have broadened these ideas. Though these philosophers used different terms for different things, the basic idea was the same. (Martin & Odell, 1995, p. xiv)

The more contemporary idea of the object-oriented paradigm first appeared in Simula in the 1960s (Pidd, 1995). In 1962 two Norwegians, Dr. Kristen Nygaard and Ole-Johan Dahl started to work with a project called Simula at Norsk Regnesentral in Oslo (Norsk Regnesentral is the Norwegian Computing Center).

Simulation is often used for problems where time dependent processes are studied, for example, questions regarding queuing are often studied with the help of simulation. Nygaard and Dahl decided in 1962 to look at the reality as a number of processes, which was a very different method compared with the approaches in earlier procedure based languages like Cobol and Fortran. The result of the work by Nygaard and Dahl was the programming language Simula 1 (1962-1964) that was an ad hoc extension of the Algol60 programming language, Simula 1 was a new programming language applicable for use in complex simulation problems (Andersen, 1996, p. 327; Ralston et al., 2003).

In 1966 Dahl & Nygaard additionally published a work on Simula 1 (Dahl & Nygaard, 1966). The aim of Simula 1 was to obtain a programming language that could be used for computer-based simulations, and for describing a very complex reality that is to be simulated (Andersen, 1996, p. 327). Object-oriented software was thus invented to support the modelling and simulation of real-world systems like car suspensions systems, oil refining processes, or medical systems (Pawson, 2002). As an example of Simula 1 code a definition of a class (Sklenar, 1997) is presented:

```
Class Rectangle (Width, Height); Real Width, Height; ! Class with two parameters;
Begin
  Real Area, Perimeter; ! Attributes;
  Procedure Update; ! Methods (Can be Virtual);
  Begin
     Area := Width * Height;
     Perimeter: = 2*(Width + Height);
  End of Update;
  Boolean Procedure IsSquare;
  IsSquare := Width=Height;
  Update; ! Life of rectangle started at creation;
  OutText("Rectangle created: ");
  OutFix(Width,2,6);
  OutFix(Height,2,6);
  OutImage
End of Rectangle;
```

In addition, Nygaard and Dahl had developed a new idea where complex reality was described by active elements that send and receive messages from other active elements, and they used Simula 1 as a base when they later developed Simula-67 out of Algol60 by taking the block concept from Algol60 a step further and introducing the concept of an object and the concept of a class (Khoshafian & Abnous, 1995, p. 13; Ralston et al., 2003). Simula 67 was ready in 1967 and had already most of the important object-oriented properties (Jacobson et al., 1995, p. 45). In Simula 67 and later versions the concepts of processes used in 1962 were replaced by more common concepts (the active elements). The active elements then become objects, and objects with similar qualities

were made to belong to the same class. (Andersen, 1996, p. 328) For a more comprehensive study of Simula, consider, for example, Dahl & Lindqvist (1993).

Conceivably, it was in 1969 that Nygaard and Dahl developed the object-oriented paradigm. The same year they tried to simulate the movements of ships in a fjord and found that this was extremely difficult. Nygaard and Dahl developed an idea where they were working with objects (ships, waves, the coast line) instead of structural entities such as the movement of the ships and the blowing of the wind. (Harrington, 1995, p. 16) Although Nygaard and Dahl are considered being the developers of the object-oriented paradigm, Nygaard has often stated that he and Dahl were influenced by the work of Börje Langefors who at that time worked with system theory that included subsystems that had internal and external behaviour. Another early pioneer in this area was Stephen Zilles who wrote a paper on "how procedures can be used to represent another class of system components, data objects, which are not normally expressed as programs" (Zilles, 1973; cited by Mikhajlov, 1999, p. 31).

The ideas of Nygaard and Dahl and the Simula programming language affected the development of the next interesting object-oriented programming language Smalltalk that was developed in the research laboratories of Xerox in Palo Alto (US) in the 1970's (Holm, 1998, p. 19; Koskimies, 1997, p. 6). Smalltalk was the first well-known object-oriented programming language and has its origin in the doctoral work of Alan Kay at Utah University (Graham, 2001, p. 3). Smalltalk was originally developed to program Dynabook, and additionally it became the software component of the Dynabook that was a kind of early laptop computer (Eliëns, 2000, p. 12).

Until 1984 the object-oriented paradigm was confined mostly to research laboratories, a few universities, some governmental agencies and the artificial intelligence community (Love, 1993, p. 40). Consequently, although the object-oriented paradigm was "founded" in the 1960's, there was a paradigm shift as late as in the 1980's and early 1990's when object-oriented software development became more common (Fernandes, 1998). One reason for the increased interest in object-oriented software development in the 1980's was the pure object-oriented programming language Smalltalk, which actually proved that the object-oriented paradigm is a complete programming paradigm (Koskimies, 1997, p. 6).

Since 1969 the object-oriented paradigm has evolved and become more mature and it has now been used for developing all kinds of information systems, administrative and business applications as well as technical applications (Eriksson & Penker, 1996, p. 27; Graham, 2001, pp. 64-65; Jacobson, 1993). The object-oriented paradigm is now in fact involved in almost all aspects of computing on a variety of platforms (Eriksson & Penker, 1996, p. 27). Moreover, there are object-oriented operating systems, object-oriented programming languages, object-oriented databases, object-oriented CASE tools, object-oriented 4GL tools, object-oriented software development methodologies, object-oriented knowledge-based systems and object-oriented expert-based systems, etc. (Harmon, 1995; Jacobson, 1993). In fact, some companies have used the object-oriented paradigm successfully for a very long time (Jacobson, 1993). Graham (2001, pp. 64-65) presents a short history of the object-oriented paradigm from 1990 to 2000,

and he concludes that in the early years of the 21st century the object-oriented paradigm will almost certainly see a nearly universal adoption.

The object-oriented paradigm also has a strong theoretical basis and background and enjoys widespread support in the academic community (Fichman & Kemerer, 1993), and therefore academic research will also support its development (Smith & McKeen, 1996).

The core of the object-oriented paradigm is the development of new information systems out of standard, existing components. The standard components could be, for example, class libraries that can be bought off the shelf (Rothering, 1994). The components can also be "lower" components like binary trees, hash tables, buttons, checkboxes and scrollbars (Tyma, 1998). Radin (1996) and Sparling (2000) propose that components can be seen as encapsulated black boxes with specified behaviour. One advantage of black boxes is that the software developer does not inevitably need to understand the internal workings of the black boxes (Martin & Odell, 1992, p. 10). In other words, software is developed like cars for example, where the car developer does not necessarily know how the different parts (like the carburettor) work, but can still build a car.

When working with the object-oriented paradigm the main information systems development issue is to work with objects, and the main development question is 'what' the information system shall do. With the object-oriented paradigm one is looking at things and what services these things offer, what states (data) the things have, and what behaviour (functionality) the things offer. (Eriksson, 1992, p. 16; Henderson-Sellers, 1992, p.35) The main focus is on objects and their behaviour, and although the objects might be complex internally, the software developer does not necessarily need to understand this complexity (Martin & Odell, 1992, p. xi). The object-oriented model is in fact more data oriented than a traditional approach (Henderson-Sellers & Edwards, 1994, p. 19). Khoshafian & Abnous (1995, p. 41) propose that the object-oriented paradigm in fact is based on data, and that the conventional software development paradigm is based on procedures. In traditional functional oriented information systems development one is looking at 'how' something should be done; 'How does this thing work?' 'Which procedure?' 'Which function?' (Eriksson, 1992, p. 16; Henderson-Sellers, 1992, p. 35)

Many authors propose that the development of information systems becomes faster and more efficient if the object-oriented paradigm is used (Henderson-Sellers & Edwards, 1990). This might be due to the fact that these techniques support reuse, which means that a new application does not always have to be developed from scratch (Fichman & Kemerer, 1993; McClure, 1996). Often existing models, class libraries, frameworks (Noack & Schienmann, 1999), architectures, code, documentations (Stevens & Pooley, 2000, p. 212), business plans, cost analyses, project plans, user manuals, requirements, designs (Räisänen, 1997b, p. 33), test suites, templates and of course classes are reused (McClure, 1996). One can also, for example, connect class libraries with CASE repositories so that new classes can be rapidly developed from existing classes (Martin & Odell, 1992, p. 12). It has also been proposed that support, maintenance and service

of the information system become easier and consequently also cheaper (Henderson-Sellers & Edwards, 1990).

In the object-oriented paradigm the integration of analysis, design and implementation within a single framework becomes possible because there is a uniform paradigm throughout development (Kaindl, 1999; Korson & McGregor, 1990). There exists a direct relationship between objects identified during analysis and objects in the implementation (Hopkins, 1992). This fact is important because information systems built according to some older software paradigm are often expensive and cumbersome to support and maintain. In fact, large organisations assign more than 50% of the total programming effort to the maintenance of older systems (Sommerville, 1996, p. 660). Wilkie (1993, p. 2) presents figures showing that 60-80% of overall software development costs are in fact maintenance costs. In light of this, the proposed benefit of easier maintenance of object-oriented systems is significant (Hopkins, 1992).

The object-oriented paradigm is probably not a 'silver bullet' (the term 'silver bullet' was originally coined by Fred Brooks (1987) as a term for oversold software process innovations (Fichman & Kemerer, 1993)) that solves all problems in information systems development today or in the future (Coad & Yourdon, 1991, p. 154; Finch, 1998).

When considering if the object-oriented paradigm can indeed be considered a 'silver-bullet' or not one can look back on different forecasts of its importance in the past.

In the beginning of the 1990's Bill Gates proclaimed: "The object-oriented paradigm is going to be the most important emerging software paradigm of the 1990s" (Martin & Odell, 1992, p. 3). Later on in 1991 the object-oriented paradigm was even predicted to do the same for software as the microchip did for hardware (Verity & Schwartz, 1991; Winblad et al., 1990, p. 23). Additionally the president of Borland International Inc. Philippe Kahn claimed that object-oriented information systems development would be predominant in the future (Verity & Schwartz, 1991).

The results from a study by International Data Corporation in 1991 showed that 70% of large US corporations claimed that they were using the object-oriented paradigm or that they intended to start using it soon (Verity & Schwartz, 1991). Moreover, in 1991 45% of the Fortune 500 companies in the United States were working with the object-oriented paradigm to some extent, and 60% of these companies were developing applications for business use according to the *Survey on Object Technology, 1991* (Taylor, 1992, p. xv).

In a study referred by Henderson-Sellers (1992, p. 12), 9 companies out of a sample of 51 in the state of New South Wales in Australia were using object-oriented software development methodologies.

Kozaczynski & Kuntzmann-Combelles (1993) claimed that the world was full of companies that were using software that was built according to traditional procedural methods, and that the step towards a new paradigm was not always self-evident. Integrating older traditional functional legacy systems with objects-oriented software would actually be difficult. In the same year Fichman & Kemerer (1993) even claimed

that object-oriented software would not be the predominant information systems development paradigm in the future.

In 1994 10% of the corporate IS groups were committed to the object-oriented paradigm according to Harmon (1995), and one year later Jacobson et al. (1995, p. 70) argued that there was a consensus that the object-oriented paradigm would be the most important information systems development platform in the near future. The same year Harmon (1995) estimated that by the end of 1996 more than 40% of the IS groups in the US would use the object-oriented paradigm.

In a study in 1996 it was found that 43,4% of the companies studied used the object-oriented paradigm (Pickering, 1996, p. 3-9), so the forecast in 1995 by Harmon turned out to be rather accurate. Still in 1997, Meyer (1997a) stated that there were some people, such as the chief of IEEE Software Al Davis, who thought the object-oriented paradigm would either fail or die in the future, although Meyer (1997a) himself was of the opposite opinion.

In 1998 Bhattacherjee & Gerlach (1998) argued that despite widespread knowledge of the benefits with object-oriented development and object-oriented tools that had been disseminated via journals, professional associations and vendors, the object-oriented paradigm had not removed entrenched information system development practices.

In 1999 Bansiya et al. (1999) proposed that many software companies had transitioned into the object-oriented paradigm and that the object-oriented tool market had been growing fast (a 42% growth rate in 1995). In the same year, Buchholz (1999) also presented a few estimations from some marketing research institutes that revealed that about 60% of the information systems development projects would probably be object-oriented in 2002.

In 2000 Pressman (2000, p. 525) argued that software experts seemed to share the opinion that future software would be developed according to the object-oriented paradigm. That same year Johnson (2000) made a study on the benefits and problems of object-oriented software development in the United States.

In 2001 Murphy (2001) addressed the question on benefits with the object-oriented paradigm and proposed that although the object-oriented paradigm was claimed to have many benefits there was not much empirical evidence.

As can be seen from the historical presentation above, there have been numerous optimistic opinions on the object-oriented paradigm becoming the predominant software development paradigm. In order to become such a dominant paradigm the benefits of the object-oriented paradigm ought to be realised and the problems ought to be handled in the right way. As a consequence, more empirical evidence concerning the benefits and problems that have been encountered in real use is needed.

On the whole, there seems to be very little comprehensive knowledge on the benefits and problems with the object-oriented paradigm in information systems science and there is also a lack of empirical information and studies on this issue. Therefore, it is hoped that this study will compensate for the deficiency through bringing a greater

awareness of the benefits and problems associated with the object-oriented paradigm that will also help working hands-on with information system development.

1.1 Purpose and boundaries of this study

There are few available studies on the benefits and problems of the object-oriented paradigm (Pomberger & Blaschek, 1996, p. 282), of which the studies by Johnson (2000) and Pickering (1996) are probably the most worthy of note. Furthermore in the book by Cockburn (1998, pp. 23-30) several benefits of the object-oriented paradigm are presented; these benefits are based on comprehensive interviews with project leaders that Cockburn has made, discussions with consultants and experts also made by Cockburn, and on information from project reports that Cockburn has read.

Because knowledge of the benefits and problems of the object-oriented paradigm might be a significant success factor for software companies as well as for other companies, more awareness of this issue is therefore needed. Verity & Schwartz (1991) propose that in an era when hardware is a commodity, software will be the most important competitive factor, and the software companies, the traditional industry and service companies, and the computer manufacturers that exploit object-oriented software development the best are likely to succeed in the computer and software industry itself.

However, the choice of software development paradigms is of course not the only critical issue for the success of companies. There are other issues to consider as well, for example, Szyperski (1999, pp. 4-5) writes about the possibility of buying standard software and information systems instead of developing them.

Purpose

The purpose of this study is to investigate and gain some understanding of what *benefits and problems* there are with the *object-oriented paradigm.*

> The object-oriented paradigm is based on a modelling approach of the real world out of objects, classes, inheritance, etc., which is in contrast to the more traditional functional paradigm that is based on separate functions and separate data (Chidamber & Kemerer, 1994)

A more precise definition of an *object-oriented paradigm* and further definitions are given in the section (1.2) on primary definitions in this study.

Note that a paradigm is more than a type of information systems development or information systems life cycle. A paradigm is also more than an information systems development method or information systems development methodology.

It is important to note that ‘problems’ are not ‘pitfalls’. Pitfalls as considered by, for example, Webster (1995) are something negative that can happen during software engineering.

By ‘benefits’ it is meant benefits in comparison with some other paradigm, usually the traditional functional software development paradigm that uses traditional functional

programming languages like Pascal and C. The same concerns ‘problems’. Problems with the object-oriented paradigm are problems in comparison with some other paradigm, like the traditional functional software development paradigm. It is worth noting here that there are other software development paradigms than the functional paradigm and the object-oriented paradigm; Koskimies (1997, pp. 1-2) mentions the procedural paradigm, the logical paradigm and the limited paradigm, Zhou et al. (1998) presents the mobile computing paradigm, and, for example, Murer (1997), introduces the software component paradigm and Bosch et al. (1997) even proposes that this paradigm is the natural extension of the object-oriented paradigm.

The purpose is also to present *different aspects* of the benefits and problems if such aspects are found in the previous studies (literature study) or in the empirical part of this study. The different aspects found in the empirical part of this study are principally presented in the summaries of the case studies.

First a comprehensive review of previous (the literature) studies is performed and the opinions of different researchers were deliberated and examine. When this review is made the author of this study searches for benefits and problems with the object-oriented paradigm.

Then an empirical study is made on what benefits and problems Finnish software companies have experienced when working with the object-oriented paradigm. In this study a large number of propositions on benefits and problems with the object-oriented paradigm are presented. Then the software developers in the Finnish software companies express their subjective opinion on which benefits or problems they have experienced. Comments are also written down.

No hypotheses are developed. One can say that the purpose is “scan” the Finnish market regarding software companies and the benefits and problems with the object-oriented paradigm. This will give some insight into the issue. The purpose is not to test any specific theory. The purpose is more to investigate specific assertions on proposed benefits and problems with the object-oriented paradigm. The assertions are, however, usually based on some theories. Note here that this study is not concerned with creating any new theory although two models for further research are presented. As a conclusion, one can argue that this study is somewhere between, on the one hand testing a theory, and on the other hand creating a new theory.

Boundaries

The focus of this study is on the areas of the object-oriented paradigm specified above. This study is neither concerned with object-oriented information systems development analysis *methods* nor with design *methods*; if the reader is interested in these issues the work by Wieringa (1998) is recommended. There is a difference between pure analysis and an information systems development analysis method because one can carry out an information systems development analysis without an analysis method as reported by Fitzgerald (1995).

This study does not investigate *detailed* benefits and problems at the programming level as many programming problems are often tied to a specific programming language

(Webster, 1995, p. 191), or database management level (for example, a lack of primary keys in object-oriented databases). For studies on more detailed programming benefits and problems with the object-oriented paradigm in software development the studies by Khoshafian & Abnous (1995), Miah (1997) and Ooil (2002) may be recommended.

As mentioned above, detailed issues of the object-oriented paradigm are not deliberated in this study. Examples of detailed issues are comprehensive programming issues (such as how to use pointers in C++), exhaustive design issues (such as how to draw a relation) and comprehensive database questions (such as how to implement an index). However, in this study a few more detailed benefits with the object-oriented paradigm are presented when *considered appropriate*, of which the benefits connected with the core concepts in the object-oriented paradigm are the best examples (Snyder, 1993).

In other words, this study is concerned with the object-oriented paradigm and not with object-oriented programming issues. This is important since the object-oriented paradigm is often confused with certain object-oriented programming languages such as C++ or Java (Khoshafian & Abnous, 1995, p. viii). It has always to be remembered that the object-oriented paradigm is more than just a programming language, and that the whole object-oriented paradigm and not just an object-oriented programming language has to be utilized, in order to achieve all the benefits of the object-oriented paradigm (Holm, 1998, p. 12).

It must be noted that the selection of questions for the survey could have been made in another manner, and that and other questions than those selected could have been selected as well.

One also has to consider the possibility that individuals who were more favourably disposed to the object-oriented paradigm were more likely to respond to the survey, thereby biasing the results in favour of the object-oriented paradigm. However, great care was taken in the wording of the cover letter, survey instructions and survey items to avoid any bias for or against the object-oriented paradigm.

1.2 Primary definitions for this study

The aim of this section is to present the primary underlying definitions that are used in this study. Major object-oriented concepts like objects, classes and inheritance are, however, presented more thoroughly in sections of their own. The definitions are by necessity brief, incomplete, and a bit oversimplified. The secondary underlying definitions for this study are presented in Appendix 6. The definitions are presented in an order that is based on the structure of the study, and not alphabetically.

Information system. Ives et al. (1980) defines an information system as ‘a collection of subsystems defined by functional or organizational boundaries’.

Martin & Odell (1995, p. 2) define an information system as a system that has information, and an ordinary system as a system without information.

Examples of ordinary systems are patient monitoring systems and plant control systems. However, the issue as to which systems that are information systems and which systems are ordinary systems can be discussed.

Paradigm. In *software engineering,* the term paradigm is used to denote a particular approach or concept that is used to refer to the way a given task is presented to and handled by the user (Webster, 1995, p. 26). A user is defined as the information systems developer or the end user.

Examples of paradigms used in software engineering are the functional paradigm (Wybolt, 1992), the object-oriented paradigm (Pree, 1997), and the component-based software development paradigm (Szyperski, 1999, p. 31). A shift from one paradigm to another can be considered as a revolution (Törnebohm, 1997) but in software engineering the shift from one paradigm to another is probably less radical though, for example, the object-oriented paradigm is very different from the functional paradigm.

Object-oriented paradigm. The object-oriented paradigm is a particular approach to software engineering and represents another paradigm for developing software systems differing from the traditional functional paradigm (Wybolt, 1992). The "old" functional paradigm also has other names, for example, Cackowski et al. (2000) call it "Algorithmic Decomposition". The object-oriented paradigm is based on a modelling approach of the real world out of objects, classes and inheritance, etc., which is in contrast to the more traditional functional paradigm that is based on separate functions and separate data (Chidamber & Kemerer, 1994). For a comparison of the traditional functional software engineering paradigm and the object-oriented paradigm, one can, for example, study the article by Wybolt (1992).

Object-orientation. Object orientation is a synonym for the object-oriented paradigm used by some authors (Meyer, 1995, p. 2).

Object-oriented method. In this study the concept 'object-oriented method' is not used although several authors and researchers, like Graham (2001, p. 1), use this concept. According to several researchers and authors the concept of 'object-oriented method' connotes to a whole philosophy of systems development encompassing programming, knowledge elicitation, requirements analysis, business modelling, system design, database design and several other related issues (Graham, 2001, p. 1). The concept of 'object-oriented method' is thereby very much similar to the concept of 'object-oriented paradigm'. There are, however, authors that see differences between the concepts of 'object-oriented method' and 'object-oriented paradigm', as for example, in the following quotation from Morris et al. (1996, p. 22):

> Typically, a paradigm is a model that breaks the development process into a series of phases that deal with different but closely related aspects of the development. In each phase of the paradigm, methods are needed to accomplish the goals of the phase, and techniques and tools are needed to apply the methods. Thus, a method is defined to be: A systematic way of proceeding with a well-defined phase of development of a computer system product. A method is composed of a series of steps.

1.3 Research problems

According to several researchers there are many benefits in using the object-oriented paradigm in information systems development (Booch, 1994, pp. 3-25; de Champeaux et al., 1993, p. xiv; Henderson-Sellers & Edwards, 1990; Jacobson et al., 1995, pp. 45-48; Smith & McKeen, 1996; Winblad et al., 1990, pp. 43-51, etc.). For example, according to Taylor (1990, pp. 103-107) there are the following potential benefits: faster development, higher quality, easier maintenance, reduced cost, increased scalability, better information structures and increased adaptability.

Eleven years later Graham (2001, pp. 41-42), mentions the same benefits as Taylor, but further he mentions benefits like information hiding through encapsulation helping to build more secure systems, better supported prototyping and evolutionary delivery, and that the object-oriented paradigm is a good tool for managing complexity, etc.

There are also of course problems with the object-oriented paradigm. However, there is still little knowledge on how companies have experienced the benefits and problems when using the object-oriented paradigm (Miah, 1997). Maring (1996) proposes that companies know little about how to use the object-oriented paradigm with predictable results. However, in a study by Villeneuve & Fedorowicz (1996) with 218 practitioners it was found that perceived benefits of the object-oriented paradigm depend on the size of the software development project and the scope of use of the object-oriented paradigm through the systems development life cycle. In the study by Johnson (2000) the question of benefits and problems with object-oriented systems development was also studied and it was found that the benefits are recognized but the problems are virtually nonexistent.

Nevertheless, there is still a lack of comprehensive studies on how to develop object-oriented information systems by utilising the benefits and avoiding the problems. One can actually argue that there is a need for knowledge on this issue in the information systems development community (McGregor, 1996).

The concept or issue of ‘benefit’ can of course also be discussed; for example, Gillach & Deyo (1993) propose that there is no real benefit of the object-oriented paradigm if the ‘benefit’ does not enforce the business impact of the developed application. This is then measured by return on investment etc.

When the review of previous studies was made several benefits and problems with the object-oriented paradigm were found. The research problems have been developed out of these benefits and problems. This reference to literature is recommended by Eisenhardt (1989) and by Yin (1994, p. 9). It is important to recognise that the purpose of the review of previous studies is to make better questions and not to look for answers about what is known of something (Yin, 1994, p. 9).

The research problems consists of several specific questions that in fact are the questions used both in the questionnaire for the survey and in the questionnaire for the case studies.

The research problems for this study are the following:

RP1: What are the *benefits* experienced with the object-oriented paradigm in information systems development?

Have the information systems development projects, for example, been faster or easier? Has the reuse concept been useful?

RP2: What are the *problems* experienced with the object-oriented paradigm in information systems development?

For example, has the object-oriented paradigm been considered immature? Has the object-oriented paradigm been considered difficult or complex?

1.4 Scientific methodology

1.4.1 Literature review

The review of previous studies is as follows; first, some basic object-oriented concepts and the object-oriented paradigm are presented. Then discussions and empirical results about the benefits and problems of the object-oriented paradigm are presented. Object-oriented analysis and object-oriented design are considered important areas of the object-oriented paradigm, and they are also considered more powerful but also more inferior than traditional analysis and design, and therefore these are considered in this study.

The previous studies are examined in such a fashion that the object-oriented paradigm, the object-oriented theory, the research problems, and the research questions can be deliberated. It is interesting to look for *similarities* and *conflicts* between different scientific sources and between concepts, theory, research questions, and research problems (Eisenhardt, 1989).

When the previous studies were read it was found that several of the benefits were connected to each other, as well as several problems were connected to each other. This interesting matter was the base for the identification of possible connections between benefits, and between problems, which gave birth to two theoretical models for further research.

Because there was a lot of interest in the object-oriented paradigm during the 1990's, some of the sources and references are nowadays (in 2005) slightly outdated. It is interesting to note how much harder it is to find articles, books and conference material nowadays than it was in the late 1990's when this study started. Some important journals like the Journal of Object-Oriented Programming are not published anymore.

1.4.2 Measuring productivity and quality in information systems development

When selecting an empirical research method for this study there must be an awareness of the different options that are available. In order to confirm whether an aspect of a software development paradigm can be considered as a benefit or a problem, this aspect must be compared with that of another paradigm. If the aspect in question increases the productivity or quality of the information systems development project or the information system itself, then one can argue that this aspect is a benefit (if it also fulfils the definition of a benefit of course). Equally, if an aspect lowers the productivity or quality of the information systems development project or the information system itself, then one can argue that this aspect is a problem (again if it also fulfils the definition of a problem).

Nevertheless, in order to be able to compare different aspects one must use some kind of approach. Different aspects and concepts of the object-oriented paradigm could be compared with the same aspects and concepts of some other software development paradigm, like the functional paradigm by using software metrics or some other suitable methods like studying the resulting information system itself or studying the project specific accounting figures of software companies, etc. Of these approaches the one using software metrics can be considered probably the most interesting. Software metrics are therefore presented next and finally there is a short analysis of the possibility of using software metrics in this study.

Software Metrics

There are a lot of different metrics. One can mention software metrics, software quality metrics, etc. In this sub section only software metrics will be presented.

Software metrics can be classified in several different ways; Meyer (1998) classifies software metrics into product metrics and process metrics. Henderson-Sellers (1996, pp. 43-56) also deals with product metrics and process metrics. Meyer (1998) further divides product metrics into external product metrics visible to users etc. (like product non-reliability metrics, functionality metrics, performance metrics, usability metrics and cost metrics for products), and internal product metrics visible only to the software development team (like size metrics, complexity metrics and style metrics). Process metrics consists of cost metrics (for projects), effort metrics (concerning the human part), advancement metrics, process non-reliability metrics and reuse metrics (Meyer, 1998).

The question of which software quality metrics to use in which special occasion is challenging because quality can be defined in several ways (Reeves & Bednar, 1994) and each quality definition probably needs a special software quality metrics for measuring the quality. However, Kan (1995, p. 83) classifies software *quality* metrics in three categories:

1. Product metrics, metrics that describe the characteristics of the product such as size, design features, performance, complexity and quality level.

2. Process metrics, metrics that is used for improving the software development and maintenance process. Examples are effectiveness of removing defects during development, the pattern of testing defect arrival and the response time of the fix process.
3. Project metrics, metrics that are concerned with the software development project. Examples include the number of developers, the schedule, costs, productivity and staffing pattern over the life cycle of the software.

However, according to Pancake (1995) there are not many reliable measurement units for predicting progress, assessing productivity and evaluating costs in the *object-oriented* world. The problem is due, among other things, to the lack of experience of object-oriented metrics (Räisänen, 1997a, p. 16). The lack of experiences of object-oriented metrics is a serious problem especially when an organisation is adopting a new technology or paradigm, and one has little experience of the new technology or paradigm (Chidamber & Kemerer, 1994).

How can productivity be measured when most of the code in a system is reused? Good metrics need to be used in order to be able to measure reuse, reusability, developing for reuse and developing with reuse, etc. (Kan, 1995, p. 31; Smith & McKeen, 1996), because traditional product metrics are not sufficient for assessing, characterizing, measuring and predicting the quality of object-oriented software systems (Basili et al., 1996b). Frakes & Terry (1996) surveyed a number of metrics and models of software reuse and reusability and proposed that although many of the metrics lack formal validation they are being used and are found useful in industrial information systems development projects. Industrial practice is important because object-oriented metrics also have to be used, in order to gain experience in how these can be used, and in how these should be used (Räisänen, 1997a, p. 16).

However, according to Berard (1998) and Webster (1995, p. 96-97) there are metrics for object-oriented software development that are used to characterize object-oriented software engineering products, object-oriented software engineering processes and object-oriented software engineering people. The object-oriented metrics is, however, different from traditional metrics because of encapsulation, information hiding, localisation, inheritance and object abstraction techniques (Berard, 1998). Webster (1995, p. 97) presents the following proposed metrics for the object-oriented paradigm (quotation):

> - Time for analysis, design, implementation, testing.
> - Average worker-days per class, average number of classes per developer.
> - Rate of change of class and subsystem interface.
> - Hierarchy metrics, including nesting level, number of abstract classes, “fan-out” (number of derived classes per base class).
> - Class metrics (both average and per class), including number of class variables, number of instance variables, number of class methods, number of instance methods and number of overridden methods.

- Instance metrics, including size (in bytes) per instance, number of instances during execution and number of persistent instances.
- Method metrics, including size (in lines of code), number of parameters.
- Coupling and cohesion metrics, including number of classes referenced by a given class.
- Reuse metrics, including number of classes used in more than one project.

Ambler (1998, pp. 174-177) also presents some metrics that can be used when estimating object-oriented software development projects. Furthermore de Champeaux (1996) presents some proficient object-oriented metrics.

Bansiya et al. (1999) and Webster (1995, p. 96) propose that the object-oriented paradigm has a lack of mature metrics, and that traditional software metrics that evaluate product characteristics like size, complexity, performance and cost do not apply to object-oriented development. This is due to the use of reuse and polymorphism, etc. that are special for object-oriented applications. For example, productivity metrics like 'lines of code produced' is a clear disincentive in an object-oriented environment, new metrics like 'number of reusable classes built' or 'number of classes / objects reused' are better and could be used instead (Gillach & Deyo, 1993; Webster, 1995, p. 98).

However, Martin & Odell (1992, p. 37) propose that object-oriented design and programming give much lower McCabe Cyclomatic Complexity Metrics than traditional functional development. The McCabe Cyclomatic Complexity Metrics is a widely used static software metric. By using this metric, one can measure '*soundness'* and '*confidence'* for a program. The metrics is based on the measurement of the number of linearly independent paths through a program module. The resulting measure is a number that one can compare to the complexity number of other software programs. (VanDoren, 1997) A more comprehensive explanation on the metrics can be found in McCabe and Butler (1989).

This proposal indicates that one could actually measure functional development and object-oriented development with the same metrics (McCabe Cyclomatic Complexity Metrics). The proposal by Martin & Odell (1992, p. 37) is based on software development work at NCR and it may be pondered whether the different measured software development projects could actually be compared. Berard (1998) also used metrics for the estimation of cyclomatic complexity of object-oriented systems. It was found that over 95% of the object-oriented software development methods had a cyclomatic complexity of four or less.

Henderson-Sellers & Edwards (1994, Chapter 10) and Pressman (2000, pp. 657-671) also contribute to the area of object-oriented metrics. Furthermore, Bansiya & Davis (1997) present different metrics for object-oriented development. They introduce different types of metrics from simple system size in classes to averages of depth of inheritance and even further to more complex metrics regarding the number of polymorphic methods. Bansiya & Davis (1997) also propose that it is important to

understand the usability of metrics in evaluating object-oriented systems. Metrics are used to chart and rationalise development of an object-oriented application and should not be used for evaluating the performance or quality of the object-oriented application. (Bansiya & Davis, 1997) Nevertheless, in order to conclude that a metrics is suitable for measuring object-oriented systems one has first of course to empirically validate the metrics. Empirical validation aims at testing the usefulness of a metrics in practice and therefore it is an important activity in order to establish the overall validity of a metrics. (Basili et al., 1996b)

Chidamber & Kemerer (1994) also present a metrics suite for object-oriented design, as they do not consider the previous methods of metrics appropriate for measuring object-oriented systems. As examples of previous methods of metrics, they mention conventional software metrics applied to traditional functional software design, as well as software metrics developed with traditional methods for measuring new object-oriented systems. The new metrics suite for object-oriented design that Chidamber & Kemerer (1994) present is based on measurement theory, and consider viewpoints of experienced object-oriented software developers. The metrics was found to possess a number of desirable properties and suggested a number of ways in which the object-oriented paradigm may differ in terms of wanted and even necessary features from more traditional functional metrics approaches (Chidamber & Kemerer, 1994).

Basili et al. (1996b) validated the metrics suite for object-oriented development designed and implemented by Chidamber & Kemerer (1994) and came to the conclusion that five out of six object-oriented metrics appeared to be useful in predicting class fault-proneness during both the high-level and low-level design phases in the software development life cycle. The object-oriented design metrics developed by Chidamber & Kemerer (1994) also showed better predictions than the tested traditional code metrics (Basili et al., 1996b).

Xenos et al. (2000) present a set of traditional metrics that they claim can be used for object-oriented programming. Furthermore, they present object-oriented metrics that consists of class metrics, method metrics, coupling methods, inheritance metrics and system metrics. All the metrics presented by Xenos et al. (2000) are, however, for object-oriented *programming*. Other aspects like analysis, design or maintenance of the object-oriented software development process are not connected with any metrics. However, the conclusion of the survey that Xenos et al. (2000) present is that nowadays there are several good metrics available for evaluating and measuring object-oriented programming, the difficult thing is more how to find the appropriate metrics for a specific object-oriented implementation and programming project.

There are other new object-oriented metrics now available, Bansiya presents another object-oriented metric in Bansiya et al. (1999), which is based on Entropy, and which is especially useful in predicting the implementation complexity of classes if the design of classes does not change substantially during implementation. Finally, it is worth noting that metrics are an important but not yet fully understood aspect of object-oriented software development; metrics can be used as input into estimating object-oriented projects, improving object-oriented software development efforts, and metrics can be

useful when selecting object-oriented software development tools (Ambler, 1998, p. 194).

Short analysis

It is surprisingly difficult to find suitable metrics for other aspects and concepts than programming and measurement of complexity for comparisons between software paradigms, like the traditional functional paradigm and the object-oriented paradigm. This argument is supported by Nierstrasz & Dami (1995, p. 24) who propose that traditional metrics are as a rule of limited use in the object-oriented world. It is worth noting that according to Henderson-Sellers (1996, p. 66), it is already difficult to compare two object-oriented projects; the issue of size is important, a measure or estimator derived from a small object-oriented project cannot be transferred without further detailed evaluation to large object-oriented projects or to non-object-oriented systems; metrics tuned for C++ are likely to be inappropriate for Smalltalk, Eiffel or CLOS (examples of programming languages with divergent underlying object models, which encourage different idioms in programming than C++).

Because paradigms are so different it might turn out to be a comparison between 'apples' and 'pears', an argument supported by Henderson-Sellers (1996, p. 1) who proposes that object-oriented systems are different in ways that effect their measurements (as examples one can mention the different life cycles, the different system structures and the issue of using classes and objects). Therefore the comparison between paradigms with metrics would not be adequate, but it is of course important to be aware of both the possible similarities and dissimilarities of the products, processes or people being compared, perhaps something that can be compared by measurements.

As a conclusion, the approach of trying to compare a paradigm like the traditional functional paradigm with the object-oriented paradigm would probably turn into a comparison between different objects and therefore very complicated. Another problem is that metrics in this area is rather immature. A third argument for not using metrics as a research method is that it would not be a realistic option within the budget and time constraint of this study.

The other research options mentioned earlier in this sub section i.e. studying the resulting information system itself and studying the project specific accounting figures of software companies are also discarded here for the same reasons.

1.4.3 Empirical research and the selection of a research method

There are several research approaches that have to be considered when choosing the best research method for the empirical part of this study. A few of these approaches are presented below and analysed in terms of their potential usefulness.

Action research. This is applied research where there is an attempt to obtain results of practical value to groups with whom the researcher is allied, while at the same time adding to the theoretical knowledge (Galliers, 1992). Action research is a qualitative research approach in which the researcher associates himself with the practical

outcomes of the research. This approach is interesting; although it has the same weaknesses as the case study approach it is different in the sense that the researcher is actively involved in the organisation studied. However, one question arises: is it realistic for the researcher to consider being active in an organisation? It might be difficult to find a software company that is willing to have a researcher actively involved in the work of the company.

Case studies. This research method is based on an attempt at describing the relationships that exist in reality, usually within a single organisation or organisational grouping (Galliers, 1992). Case studies are a typical qualitative research approach. With case studies more knowledge of the phenomena being studied can usually be found than in surveys, assuming that the interviews, etc. are successful (Galliers, 1992). There is also a likelihood of generating novel theory when cases are used for theory building (Eisenhardt, 1989). According to Benbasat et al. (1987), case studies are good for capturing knowledge from practitioners (for example, system analysts) and for developing theories. Case studies are a good approach if the main questions of the research are 'how' and 'why' questions (Yin, 1994, p. 9). This claim by Yin (1994, p. 9) is supported by Walsham (1995) who proposes that the interpretative school also thinks that case studies are the appropriate research strategy for 'how' and 'why' questions. Case studies are also well suited for research in an area where few previous studies have been carried out, which in fact is the case with the current study (Benbasat et al., 1987). A researcher using the case study approach often has little presumptive knowledge of what the interesting variables are and how they will be dealt with (Gable, 1994). According to Benbasat et al. (1987) this approach is well suited to information system research, because the technology is rather new and organisational questions are interesting.

There are also of course problems with case studies, which are discussed in more detail later in this study. However, in short it has been claimed that case studies lack statistical validity, that they can be used to generate hypotheses but not test them (Gummesson, 1991, p. 77), that they lack rigor, that they result in too much material that is difficult to handle (Yin, 1994, pp. 9-11), that they are time intensive (Covaleski & Dirsmith, 1990; Yin, 1994, pp. 10-11) and that making generalisations based on them is problematic (Eisenhardt, 1989; Gummesson, 1991, p. 77; Yin, 1994, p. 10).

Despite the above-mentioned problems, however, *the case study approach is seen to be a suitable method* for the empirical part of this dissertation.

Evaluation study. In the evaluation of the innovation, the innovation (e.g. the object-oriented paradigm) is compared with a stated goal or criterion (Järvinen, 2004, p. 11) and / or one tries to answer the question "how useful is the particular innovation?" The evaluation can be made by using for example analytical approaches, the case study research method, experimental studies, field studies or simulation (Järvinen, 2004, p. 13). *This research method is appropriate for this study* and the case study method can be used as well as the survey method that is a part of field studies, when working with this research method.

Grounded theory. The research method could also be out of grounded theory that is based on an approach where hypotheses are not used in the beginning of the research work. Grounded theory is concerned with the development of theories. It is an inductive (from data to theory) theory. (Lundahl & Skärvad, 1999, p.105) The theory is developed out of the data from the field (Järvinen & Järvinen, 1995, p. 45). For a more comprehensive study of grounded theory, Glaser & Strauss (1967) may be considered, for example. The grounded theory approach is not considered useful for this study because no theory will be developed out of the data from the field.

Subjective, argumentative. This creative research method is based more on option/speculation than observation, thereby placing greater emphasis on the role/perspective of the researcher (Galliers, 1992). It can be applied to an existing body of knowledge (reviews) as well as actual/past events/situations (Galliers, 1992). This approach could be used here but it does put a lot of responsibility on the researcher who subjectively discovers the results informally. This approach is useful when creating a theory that can be tested. However, the aim of this study is not to create new theory, it is more based on validating earlier proposals of the phenomena being studied (the experienced benefits and problems of the object-oriented paradigm). There are several weaknesses to this approach, mostly because it is the researcher who subjectively interprets the phenomena being studied.

Surveys. Obtaining snap shots of practices, situations or views at a particular point in time (via questionnaires or interviews) from which inferences are made (using quantitative analytical techniques) regarding the relationships that exist in the past, present and future (Galliers, 1992). *Surveys seem to be an appropriate approach*. Some kind of description of real world situations can be found, although the insight of the phenomena being studied might be limited (Yin, 1994, p. 13). Surveys are of course a typical quantitative research approach where one is collecting data on some phenomena from a large number of sources (Nandhakumar & Jones, 1997). In surveys associations that exist in several organisations can be found and generalisable statements can be made about the phenomena being studied (Gable, 1994). There are of course problems with surveys too; problems that are discussed later in this study. In this dissertation, scanning the market in order to get a general picture of the experienced benefits and problems with the object-oriented paradigm in Finnish software companies will be performed and surveys are appropriate for this. Surveys for information systems research are discussed by, for example, Newsted et al. (1998).

Systems development. According to Nunamaker et al. (1991) systems development could be used as a research approach in information systems research when relevant research questions and valid hypotheses can be stated. Systems development as a research approach should conform to the following criteria; that the purpose is to study an important phenomenon in areas of information systems through system building, that the results make a significant contribution to the domain, that the system is testable against all the stated objectives and requirements, that the new system can provide better solutions to information system problems than existing systems and that experience and design expertise gained from building the system can be generalized for future use (Nunamaker et al., 1991, p. 101).

Systems development could also be a complementary research approach to some other research approach like case studies (Nunamaker et al., 1991). In this study the focus is not on the whole system development process, only on benefits and problems with the object-oriented paradigm, therefore making systems development an unsuitable research approach.

The following research approaches were also considered: *analytic induction*, *consultancy*, *descriptive/interpretative research*, *field experiments*, *forecasting*, *laboratory experiments*, *observing*, *participant observation*, *simulation & game/role playing, theorem proving and written materials & documents* but were almost immediately found irrelevant for this study.

Hamilton & Ives (1992) studied how often case studies, field studies, field tests and laboratory studies had been used in published MIS articles 1970-79. Case studies were the most commonly employed empirical strategy. In another study of three IS journals by Nandhakumar & Jones (1997): MIS Quarterly, Information Systems Research and the European Journal of Information Systems between 1993 and 1996, surveys were the most commonly used research approach, experiment was the second and semi structured interviews was the third.

Discussion. After considering the different approaches outlined above, the following conclusion has been reached: the best solution and the overall empirical research method for this dissertation will be the *evaluation research method with a combination of a survey and a case study*.

An important argument for a survey is presented by Verschoor & Low (1994) who argue that "as with any study investigating general 'state-of-practice', a survey is a feasible means of providing data with sufficient external validity". Because the empirical part of this study is concerned with treatment of the 'state-of-practice' of experienced benefits and problems with the object-oriented paradigm among Finnish software companies, a survey seems to be an appropriate research method.

The empirical research methods and the empirical research design are discussed later on in this study.

1.5 Structure of the study

This dissertation begins with an introduction and a short historical review of the object-oriented paradigm. Subsequently the aim and boundaries of the study are presented. Some primary definitions are then presented and afterwards the research problems are defined.

The scientific methodology to be used in this study is then discussed. Followed by a discussion on the possibility to measure productivity and quality in information systems development, a presentation of a number of empirical research methods and a selection of the research approach to be used in this dissertation.

A review of the basic object-oriented concepts, objects, classes, relations, inheritance, dynamic binding and polymorphism is then presented in chapter 2 as a preparation for a major chapter (chapter 3) on the benefits and problems with the object-oriented paradigm where twelve benefits and twelve problems are presented and discussed.

This is followed by chapter 4 with the empirical part that begins with an introduction and is then followed with a presentation of the research method and the research design. Here the survey and case studies methods are presented and discussed. After that the research questions used in the survey and the questionnaire are discussed. The pilot study is then introduced after which the results are presented. Afterwards the survey and the case studies are presented.

The section on the analysis of the theory and the empirical findings then follows. In this section the theory found in the review of previous studies is compared with the findings from the pilot study, the survey and the case studies.

In chapter 5 there is a summary of the findings from the empirical study and an analysis and discussion of the empirical findings.

The last chapter consider four sections: a repetition of the results of the study, some limitations of the study and the empirical part, some recommendations for practitioners and finally some recommendations for researchers. In the last section there is a look in the future with suggestions for future research. In this section two tentative theoretical models are further presented. The first model, the CBB model concerns the connections between the benefits of the object-oriented paradigm. The second model, the CBP model focuses on the problems.

2 ON OBJECT-ORIENTED CONCEPTS

The object-oriented concepts that will be presented in this chapter are closely connected to fundamental concepts of object-oriented programming, but because these concepts reappear in the chapters on benefits and problems with the object-oriented paradigm they are considered as basic object-oriented concepts and not specifically object-oriented programming concepts. Note, however, that the concepts presented can often be found in object-oriented design and object-oriented databases, and to some extent in object-oriented analysis and knowledge databases, with the exception of dynamic binding and polymorphism.

Because the terminology of the object-oriented paradigm is different in different object-oriented programming languages, some differences are described in Table 1 below before the object-oriented concepts are presented.

The table is from Henderson-Sellers (1992, p. 264) and originated in Winblad et al. (1990):

Table 1: Object-oriented concepts in different object-oriented programming languages

Smalltalk	**C++**	**Objective-C**	**Object Pascal**	**Eiffel**	**CLOS**
Object	Object	Object	Object	Object	Instance
Class	Class	Factory	Object type	Class	Class
Method	Member Function	Method	Method	Routine	Method Generic function
Instance variable	Member	Instance variable	Object variable	Attribute	Slots
Message	Function call	Message expression	Message	Applying a routine	Generic function
Subclass	Derived class	Subclass	Descendent type	Descendent	Subclass
Inheritance	Derivation	Inheritance	Inheritance	Inheritance	Inheritance

In this chapter the concepts from Smalltalk will be used because Smalltalk is a pure object-oriented language and one of the original object-oriented languages.

The concepts in the object-oriented paradigm can be interpreted in different ways as can be seen from Table 1. Objective-C is the programming language that came with the NeXT computers (Verity & Schwartz, 1991), C++ is a hybrid programming language, Object Pascal is a modified version of the programming language Pascal (Pascal was developed by Niklaus Wirth in Zurich in 1970), Eiffel is the programming language developed by Bertrand Meyer and CLOS (Common Lisp Language) is an object-oriented version of the programming language Lisp.

There are also of course other object-oriented programming languages than the programming languages presented in Table 1. Eliëns (2000, p. 142) and Graham (2001, p. 71) propose that there are more than 100 object-oriented and object-based

programming languages. Koskimies (1997, p. 7 & p. 15) presents the object-oriented programming languages Oberon-2, Kevo (based on prototypes, developed in Finland, and connected to Antero Taivalsaari) and Ada 95, which is an object-oriented extension to the programming language Ada (Ada is still popular in government). Heller (2003) introduces the object-oriented programming languages D (the D language by Walter Bright), Python, Lazlo, Jscheme, Needle, and Water (the Water language by Clear Methods Company). Love (1993, p. 227) introduces the object-oriented programming languages ProGraph, Actor, Dylan and Pro-Kappa. Wilkie (1993, pp. 213-214) presents the object-oriented programming languages Trellis (developed by the Digital Equipment Corporation) and Object Cobol (in 1993 there were still 70-80 billion lines of Cobol source code in the world, and the hybrid Object Cobol makes it possible to migrate from Cobol to object oriented programming). According to Fogarty (2004) there is still a lot of new Cobol code written every year. The information systems development work and maintenance work that is done by programming in Cobol is, however, nowadays often based on the object-oriented paradigm (Fogarty, 2004).

Object-oriented programming languages are interesting but there are also object-*based* languages like Ellie, Modula-2, PowerBuilder and Visual Basic (Graham, 2001, p. 109). Further, there are scripting languages like JavaScript and TCL (Watson, 1999) and these languages should not be confused with object-oriented programming languages.

Object-oriented programming languages are either pure object-oriented programming languages, like Smalltalk, where everything is an object (Fagerström, 1993, p. 20), or hybrid object-oriented programming languages like Ada 95, C++, Object Pascal, Turbo Pascal with Objects, Modula-3, Object Cobol and modern versions of Simula like Beta (Koskimies, 1997, pp. 6-7).

Cockburn (1998, p. 29) presents further hybrid programming languages such as C@+ and SOM (IBM's System Object Model), of which SOM probably cannot be considered a very pure programming language. Eliëns (2000, pp. 145-147) presents the hybrid object-oriented programming languages Concurrent Smalltalk (an extension of Smalltalk), DLP (an extension of Prolog), FLAVORS (an extension of Lisp and supported by the company Symbolics in the US), LOOPS (an extension of Lisp), Orient-K (a language for parallel knowledge processing), POOL-T (a simplified version of Ada) and Vulcan (an extension of Prolog). Khoshafian & Abnous (1995, p. 18) present the object-oriented programming languages CommonLoops from the company Xerox and Common Objects. Java and the object-oriented programming language C# (also called 'C-sharp') should also not be excluded in this context.

Another way of classifying object-oriented programming languages is presented by Mikhajlov (1999, p. 71) and Weck (1997) who propose that there are object-oriented programming languages that employ classes, and there are those that rely on prototypical objects. The first category that employs classes includes all the most well known object-oriented programming languages like C++, Simula and Java. The second category that rely on prototypical bases, are called prototype-based, and the programming languages Cecil, Self and Kevo are examples of such programming languages. In prototype-based programming languages, objects are created by cloning an existing object, the prototype, and modifying the clone (Weck, 1997).

2.1 The object

The concept of object is central to the object-oriented paradigm. An object is an instance of a class.

An object consists of data and possibly a method, which is a procedure or a function; an object is thus an abstraction (King, 1989; Snyder, 1993). Abstraction is defined by Stevens & Pooley (2000, p. 10) in the following way (quotation):

> Abstraction is when a client of a module doesn't need to know more than what is in the interface.

The object is in other words an entity that is clearly delimited from its environment, although objects of course have contact with the environment (Taylor, 1992, p. 47). One main difference between an object and a module is that an object rarely operates in isolation, and at runtime an object-oriented information system can usually be seen as a network of communicating objects, which cooperate to achieve the overall functionality of the information system (Mikhajlov, 1999, p. 32).

Note that although objects and classes are different things the concept of an 'object' is often, in reality, used to mean the class description itself; as a result there are, for example, 'object models' meaning class descriptions and 'account objects' meaning instances of an "`Account`" class (Cockburn, 1998, p. 5).

An object can also consist of other objects; objects that contain other objects are called *composite objects*. However, in many systems composite objects have reference variables to other objects, so they do not actually 'consist' of other objects. (Taylor, 1992, p. 44) Composite objects can have objects that are also composite, and this type of nesting can go on (Taylor, 1992, p. 47).

An object is defined in several ways in Webster's Encyclopaedic Unabridged Dictionary of the English Language:

One definition (quotation):

> Anything that is visible or tangible and is stable in form.

Another definition (quotation):

> Anything that might be apprehended intellectually.

A third definition (quotation):

> A thing, person, or matter to which thought or action is directed.

Martin & Odell (1992, p. 16) define an object in the following way (quotation):

> An object is any thing, real or abstract, about which we store data and those methods that manipulate the data.

Later, in 1995, Martin & Odell (1995, p. 26) define an object as follows (quotation):

> An object is anything to which a concept applies. It is an instance of a concept.

An object is theoretically an entity that can save state (in the attribute values) and has a number of operations (behaviour) that can either examine or affect the state (Jacobson et al., 1992, p. 44; Kung et al., 1995). The behaviour of an object is the total set of services (operations) of the object (Henderson-Sellers & Edwards, 1994, p. 54). Operations are the only way to change the internal data of an object (Gamma et al., 1995, p. 11). They can be considered as the commands of the object; operations can be seen as answers to the requests for the object to do something (Henderson-Sellers & Edwards, 1994, p. 52). Operations have signatures and the signature of an operation consists of the operation's name, the objects it takes as parameters and the operation's return value (Gamma et al., 1995, p. 13). Requests (messages) are actually the only way to get an object to execute an operation (Gamma, et al., 1995, p. 11). The messages may in fact do either of two things. The messages can ask the object to perform a computation and return a value or the messages can modify the object's content, changing its state or value (Khoshafian & Abnous, 1995, p. 39).

As a summary, one can say that objects have a state, behaviour and an identity (Booch, 1994, p. 83). Below is a figure (Figure 1) of an object (truly a class) with attributes and services (Yourdon & Argila, 1996, p. 10):

Figure 1: Object with attributes and services

SUBSCRIPTION
----------Attributes----------
subscription_id
subscription_status
subscription_details
subscriber_id
recipient_id
service_bureau_id
pricing_id
payment_id
----------Services----------
RECOGNIZE SUBSCRIPTION _REQUEST
REQUEST_SUBSCRIPTION
Enter_Paid Subscription
Enter_Comp_Subscription
Report_Subscriber
Terminate_Subscription
Renew_Subscription

An attribute is defined as an abstraction of a single characteristic, possessed by all the entities that were abstracted as an object (Shlaer & Mellor, 1988, p. 26). The attributes can be descriptive attributes, naming attributes or referential attributes (Shlaer & Mellor, 1992, p. 16). In this area, a domain is defined as the set of values an attribute

can adapt to (Shlaer & Mellor, 1988, p. 37). The objects hold the attribute values (Kung et al., 1995; Rumbaugh, 1997).

In short, in an object-oriented language the objects are entities with functionality inherently tied to the data (Davis, 2000); examples of objects are invoices, organisations, shapes in drawing programs, screens in an application, nodes in CASE tools, mechanisms in robotic devices, engineering drawings, airplanes, airplane flights, airline reservations, icons on screens, order filling processes, customers, products and buildings, etc. (Martin & Odell, 1992, p. 15). There are of course other ways of presenting objects, for example, Kozaczynski & Kuntzmann-Combelles (1993) claim that objects can be icons, strings, subsystems or even servers. Gamma et al. (1995, p. 13) also indicate that complete subsystems can be objects.

Note that objects can be built with traditional functional programming languages like C, Cobol and Fortran as well as object-oriented programming languages (Martin & Odell, 1992, p. 11; Sanguinetti, 2000). However, an object-oriented programming language like Smalltalk is usually more convenient to use when building objects, and gives rise to several advantages (Jenz, 1999c).

2.1.1 Methods

Method is basically a Smalltalk term; in C++ methods are called member functions and in Eiffel methods are called routines (Henderson-Sellers, 1992, p. 234). The methods specify the behaviour of the object. The behaviour of the object is based on the set of messages that the object can respond to, since the methods execute the performance that is requested by the message (Wirfs-Brock et al., 1990, p. 20). As a rule, the methods are stored in classes not in objects and a method provides an implementation of an operation.

However, in the prototype-based object-oriented programming language *Self,* methods are stored in objects, although this is made possible by pointers, and the code of the methods is not copied to every object (Rumbaugh, 1997).

Methods are functions or procedures; functions return an object and procedures do not return anything. Thus, functions give a query facility and procedures change the state of an object. For example, in Smalltalk a method is always a function; the object that is returned can be ignored if necessary. The knowledge of traditional functional and procedural programming can of course be utilised when developing methods in the object-oriented world (Henderson-Sellers, 1992, p. 234).

A change in the object's attribute values by a method might cause changes to the attribute values of other objects (Kung et al., 1995) because objects are connected to each other by the message passing facility, and according to Webster (1995, p. 24), a message is a command to an object. The connection between objects has to be made according to some rules and in 1987 Ian Holland (Lieberherr, 2005) presented the 'Law of Demeter' which stated that objects should not navigate too far from their immediate surroundings in accessing other objects, or else the scope of coupling becomes more

difficult. Lieberherr et al. (1988; cited by Fagerström, 1995, p. 231) presents the following group of rules for the 'Law of Demeter':

> For all classes C and for all methods M in class C, M can call methods (send messages) to the following objects:
>
> - Argument objects.
> - The object itself.
> - Objects that are referred by instance variables.
> - Global objects.
> - Objects that have been developed by C.

The use of the Law of Demeter leads to a disciplined use of classes. There are of course other rules that can also be used when working with connections between objects.

2.1.2 Identity

The identity of an object distinguishes it from all other objects (Fagerström, 1993, p. 17). Object identity enhances the notion of pointers in traditional programming languages (like C), primary keys in databases and file names in operating systems (Wilkie, 1993, p. 18); an identity is in other words implemented through the rules of the implementation environment (Fagerström, 1995, p. 25).

An identity is not the same as an identifier. An identifier is a set of one or more attributes. The values of the attributes make the object unique and the user can then distinguish the objects from each other. (Putkonen, 1994) In other words, an identifier is defined as a set of one or more attributes that uniquely distinguishes each instance of an object (Shlaer & Mellor, 1988, p. 32). If an object has several identifiers one identifier will become the preferred identifier (Shlaer & Mellor, 1992, p. 15). Every object has an identity of its own, which means that if two objects have the same attribute values (the same identifier) they can still be identified by using the identities (Kung et al., 1995). Object identity is a semantic concept associated with objects and the easiest way of implementation is to use the hardware memory address of an object as its identity (Wilkie, 1993, p. 18). The identity cannot be altered, and the object has the same identity as long as the object exists (Fagerström, 1993, p. 17).

2.1.3 Relations

Relations are also considered in another chapter of this study. However, a short presentation of the message-passing relation will be given here. In this relation the information associations with other objects are specified. The associations can be static relations or dynamic relations. Static relations exist for a long period (the objects are 'connected'). In dynamic relations two objects communicate with each other. (Jacobson et al., 1992, p. 45) In a more extensive sense relations can be considered as relationships that are an abstraction of a set of associations that systematically hold between different things, which are, in fact, objects (Shlaer & Mellor, 1992, p. 21). The objects provide services to the clients (the programs or users) that have called the objects. The clients call the objects according to a message-passing technique. Jacobson et al. (1992, p. 47)

use the word ‘stimuli’ instead of the word ‘message’. A message is a signal from one object to another object that requests the receiving object to carry out one of its methods. A message includes a keyword called a selector and one or several arguments, for example, resetTime (4, 12:30) where ‘resetTime is the selector and ‘4’ and ’12:30’ are arguments (Stevens & Pooley, 2000, p. 16). It is worth noting that messages and the message sending syntax differ between various object-oriented programming languages.

In other words, a message sending activity is occurring when an object calls a method in another object. The methods of the objects perform calculations that correspond to the abstraction of the object. A message is ‘technically’ the name of an object and the name of a method that can have parameters. The object that initiates a message (sends a message) is a sender and the object that receives the message (from the sender) is a receiver. The sender might also require a response from the receiver and the response is usually called a return value (Taylor, 1990, p. 43). The message passing technique can roughly be compared with a subroutine call in a procedural language, but is actually a more extensive activity, because the message is located without searching through the entire object (Henderson-Sellers, 1992, p. 24 & p. 240). Rumbaugh (1997, p. 6) explains the message passing-technique in the following concise way (quotation):

> You call an operation and the compiler figures out which subroutine to call by examining the class of the object (stored in a standard place) and then looking up the correct method in a table. This works provided each operation is attached to one class as in C++ or Smalltalk.

The request for a service (a message, or a ‘stimuli’) that the clients send to the objects can contain parameters, and a result can be obtained. The requests can be generic; a client can issue the same request (message) to several different objects.

As an example of how relations and objects could be used in an information system, one can present a system with accounts in a bank. An object could then represent an account. The variables of the account would be called instance variables (as, for example, account number, owner, balance and control number). Some methods could be associated with the account. These methods could be, for example, the methods ‘open’, ‘close’, ‘check’, ‘deposit’ and ‘withdraw’. When an object is calling these methods relations are created between objects.

2.1.4 Encapsulation

Objects are encapsulated, the external aspects of an object are separated from the internal aspects, and other objects can only access the external parts of the object. Using operations (methods) is the only way to change an object’s state and internal data. The objects thus support the concept of *information hiding*. Information hiding is defined by Parnas (1972) and presented by Pree (1997) as follows (quotation):

> A module is characterized by its knowledge of a design decision, which it hides from others. Its interface was chosen to reveal as little as possible about its inner working.

The difference between encapsulation and information hiding is that encapsulation is the act of grouping data, and operations that affect that data, into a single object, however, the content of the object could be visible to other objects. Information hiding is based on the *private* part of the object, which is how the object carries out the operations; this private part of the object is not visible to other objects. In other words, encapsulation is the bundling of data and methods, and information hiding is implemented through private instance variables that only the methods of the object can access. (Wirfs-Brock et al., 1990, p. 6 & p. 18)

Having said that, many researchers like Rumbaugh (Rumbaugh et al., 1991, p. 7) do not distinguish between encapsulation and information hiding. Henderson-Sellers (1992, p. 19) proposes that encapsulation does not guarantee information hiding but information hiding guarantees encapsulation. According to Penker (1994), one can state that encapsulation is to define an object as having both data and functions. Data is encapsulated and can only be accessed through the functions that are defined and permitted. The objects can also have *public* instance variables (as in C++), and these instance variables all clients can access. When an object performs a service that a client requested, the object uses a method.

Encapsulation promotes reuse because data and functionality packed together becomes a feasible module for reuse (Davis, 2000). The design of an information system is also easier due to the encapsulation feature. Encapsulation is also the ground for the benefits of the object-oriented paradigm that concerns robustness and management of complexity (Webster, 1995, p. 182). Encapsulation also helps to build secure information systems (Graham, 2001, p. 67).

2.1.5 Discussion on objects

When an object is examined, what is interesting is that the object consists of functions and data. In a traditional system an object can be seen as a subprogram or a function (a module) with data, or as a combination of data (often in a record) with ordinary functions and procedures that deal with the data.

The encapsulation of objects with functions and data has also some shortcomings. Höydalsvik & Sindre (1993, p. 246) noticed that it might be difficult to implement some *business rules* with strict encapsulation. If encapsulation is studied more carefully, one can see encapsulation as an extension of the record concept of Pascal and the struct concept of C. In Pascal and C the record and struct concept only consist of data, when an object in a class consists of both data and functions, and the functions define all of the ways the data can be manipulated and examined. Thus, data and methods are not normally global in an object-oriented program. The data and methods are local to a particular class. (Pidd, 1995)

The object is a natural entity for many concepts in the real world as the example with the bank shows. This is among other things due to the fact that it is often easier to look at real things that exist, for example, in business, than it is to look at separate programs and files or databases, in order to get a picture of a system in an organisation (Smith & McKeen, 1996). This is especially true in large and complex data systems. However,

different persons or actors in the application area might see the objects in different ways. There is a subjective view of the object. One can talk about subjects that are a collection of state and behaviour specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool. (Harrison & Ossher, 1993)

The subject-oriented programming approach can be considered a supplement or a challenge to the object-oriented paradigm. One main point is to remember, however, that different persons and different applications, etc. can see the same objects in different ways, which makes it difficult to have one object or one class for all the different parts. If the object is shared between several applications then a change in the view of the object of one application, might force the other application to be modified according to the new appearance of the object (Harrison & Ossher, 1993). Because it can be difficult to find objects, and because different persons and different applications, etc. can see the same objects in different ways, it is often useful to classify the objects into different categories. The following categories (quotation) are proposed by (Eriksson & Penker, 1996, pp. 81-87):

- Business objects. Business objects are objects like actors (persons, roles), entities (orders, accounts, storage places), processes (business processes), occurrences (impulses), rules (perform calculations, control) and aims.
- Control objects. The objects perform a course of action by considering the action and then calling other objects and co-ordinating them.
- Entity objects. An entity object describes an object-oriented interface to another physical part of the system. A physical part can, for example, be a printer.
- User interface objects. An object like a window in the user interface.
- Database objects. For persistent storage of objects.
- Product objects. A result like a report that has been generated from the system.
- Communication objects. A communication object administers the communication between different computers in the physical architecture.
- Interface objects. An interface object encapsulates the internal structure of a subsystem, and presents the interface of the subsystem to the environment.

Other categories can of course be found, and all systems will not have objects from all categories (Eriksson & Penker, 1996, p. 86).

The connection between objects in the object-oriented model and objects in the real world has been criticised by Höydalsvik & Sindre (1993) and by McGinnes (1992). Further, Pawson (2002) argues that business objects are behaviourally deficient because they are defined principally in terms of their attributes and associations, and not out of their functionality. Stevens & Pooley (2000, p. 219), however, propose that problem domain objects by their nature frequently recur in different contexts; a company can therefore develop (out of the problem domain objects) a collection of business objects that reflects the common entities in the business of the company.

2.2 The class

The basic component of an object-oriented system is the class. The independence of classes defines the external structure of the object-oriented system (Bansiya & Davis, 1997). Martin & Odell (1992, p. 21) define a class in the following way (quotation):

> A class is an implementation of an object type. It specifies a data structure and the permissible operational methods that apply to each of its objects.

A class is a template (a type) for the objects, a collection of similar objects. The class forms the description of the objects that belong to the class; in the class the methods and variables for the objects are defined. At runtime the objects have a certain structure and behaviour, and the description of that structure and behaviour is a class. The difference between a class and a type is that a type defines the interface of a set of objects and a class defines the implementation, a type can have different implementations (Madsen, 1995). Gamma et al. (1995, p. 16) give a good explanation of the difference between a class and a type (quotation):

> An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface – the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

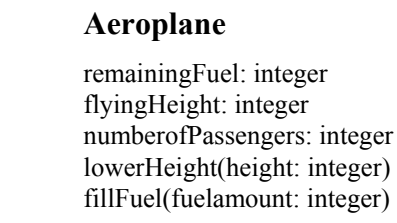
Object-oriented languages support the separation between type and class by the abstract data types (ADT) approach, the export approach or the modularisation approach. The abstract data types approach is used in Smalltalk; the export approach is used in C++ (which actually uses classes to specify both an object's type and its implementation according to Gamma et al. (1995, p. 17)) where mechanisms like public, private and protected are used for this. (Madsen, 1995)

However, according to Khoshafian & Abnous (1995, p. 33) abstract data types are important because abstract data types are represented and implemented in object-oriented systems through classes.

The benefits of the abstract data type are interesting, because as one can recognize when reading this study, many of the benefits of the object-oriented paradigm are connected to the abstract data type and the class. The class is, for example, a very reusable artefact that supports the reuse concept and the abstract data type is the base for encapsulation.

In the following figure (Figure 2) an example of a simple aeroplane class is presented (source: Fagerström, 1995, p. 26):

Figure 2: A rough sketch of a class



Generally, every object has to belong to a class; one exception is, however, the prototype-based object-oriented programming languages.

In the object-oriented paradigm a general class can first be defined, this general class is often known as an abstract data type (Pidd, 1995). In a pure object-oriented solution the class is in fact an abstract data type, and the details of the class are private for the class (Korson & McGregor, 1990). In other words, a class is an implementation of an abstract data type (Henderson-Sellers, 1992, p. 229). Pree (1997) proposes that object-oriented programming languages improve the module concept of functional programming languages by having a straightforward definition of abstract data types (the class), and by providing programming language constructs for the extension and modification of the abstract data types (the classes).

The user interface of the class consists of two types of class methods. The first type consists of accessor functions that return abstractions about the state of an instance (the values of the instance variables). The other type of method consists of transformation procedures that can change the state of the class. This is simply done so that the values of the instance variables are changed. (Korson & McGregor, 1990)

A class that consists of objects (instances) has an interface that specifies the operations of the class, the instance variables of the class, the constants of the class and the exceptions of the class (Nierstrasz, 1989). A class can also have variables and methods that are only in the class and that are not duplicated to the instances (objects) of the class; such variables are called class variables, and such methods are called class methods (Taylor, 1992, p. 58).

According to Korson & McGregor (1990) several researchers suggest a similar point when writing about objects and instances. Actually, the terminology in studies concerning object-orientation differs from one researcher to another. For example, Shlaer & Mellor (1992, p. 163) make a difference between an object and an instance; an object is an abstraction of a real-world thing and an instance is a single specified instance of a class. Wirfs-Brock et al. (1990, p. 22) talk about instances as being objects of a class, objects that behave according to the specification of the class. According to Hopkins (1992) objects are instances of classes and objects that belong to the same class (are instances of the same class) have identical behaviour but private data.

A class can also have other classes as a specification. If the class only has other classes as instances the class is called a *metaclass*. A metaclass is a class whose instances are themselves classes (Booch, 1994, p. 134). In a metaclass, information that relates to the class itself and not to the objects of the class is defined (Koskimies, 1995). The number

of objects is an example of information that relates to the class itself and not to the objects of the class; the number of objects cannot be stored in the objects. For example, in C++ that does not have any metaclasses, such information is stored in a static class member. The static class member belongs to the static storing class in C++ (Prata, 1991/1992, p. 431). In Smalltalk there are metaclasses, every class belongs to a metaclass and all metaclasses belong to a single metaclass (Taylor, 1992, p. 156).

The interface of the class is often divided into three parts, a private part, a public part and a protected part. In all of these three parts the class members (data and methods) can be described. If the class members are in the private part, only the class itself can access the class members. If the class members are in the protected part, only the class itself and the subclasses of the class can access the class members. If the data or the methods are in the public part, then all classes (clients) that are visible to the class can access the data or methods. C++ supports this division of the interface of the class. Other programming languages do not always have all three parts. (Booch, 1994, p. 105)

As an example, one can present an information system with accounts in a bank. The account has an owner, some data and some methods for the handling of the account. An account is an object. All objects that are the same become then an entity that is a class.

When developing classes, one can notice that the descriptions of classes are similar to abstract data types. An abstract data type is the description of a class with no implementation details, when a class is the implementation of an abstract data type (Henderson-Sellers, 1992, p. 24). A class has member functions, and is divided into a private part and a public part. Important is also to remember that a class is a *description* of a collection of objects. An object has the structure of its class, but takes up memory and has data values. (Korson & McGregor, 1990) It must be remembered, however, that an abstract data type is a user-defined type, for example, if integers, reals and chars are defined by the programming language then the abstract data type extends the set of types to user-defined types that can be anything like customers, machines or invoices (Henderson-Sellers, 1992, p. 21). Object classes are in other words types for objects in the same way as an integer, for example, might be the type for a customer's age.

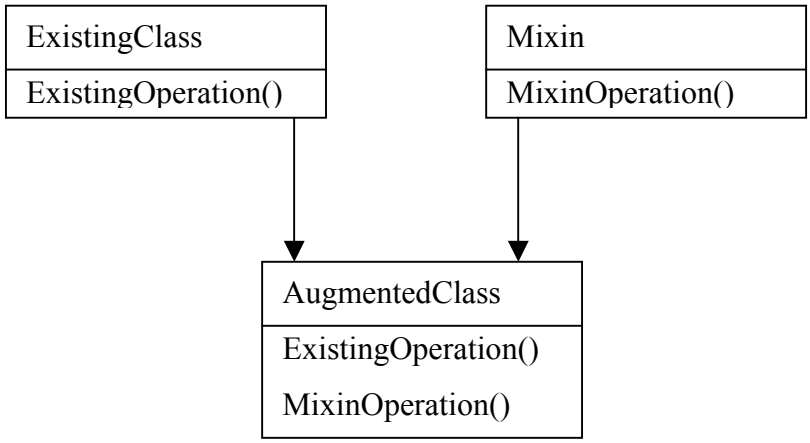
2.2.1 Different types of classes

There are several different types of classes. There are abstract classes that sometimes are called 'virtual' classes (Taylor, 1990, p. 56) or abstract base classes (Webster, 1995, p. 174). These classes are classes from which inheritance can be made and these classes do not have any objects. The concrete classes or concrete base classes (Webster, 1995, p. 174) are principally classes that the inheritance mechanism does not use and these classes do have objects. The abstract classes have such qualities that they can be easily reused (for example, a common structure with methods for a physical thing like a machine), and they constitute therefore a natural base for the inheritance mechanism. (Wirfs-Brock & Johnson, 1990) Concrete classes are designed first so that their instances (objects) are useful, and second so that they can be used for inheritance (Wirfs-Brock et al., 1990, p. 109). All superclasses are abstract classes, but all subclasses are not concrete classes. Abstract classes do not necessarily have subclasses,

for example, when they have been defined for capturing architectural intent. The subclasses are not always concrete classes, because there might be several levels of abstractions (abstract classes) before concrete classes are reached. (Selic et al., 1994, p. 261)

A special type of class is a mixin class. A mixin class is a class that is intended to provide an optional interface or functionality to other classes. Mixin classes can be used only when multiple inheritance is used. (Gamma et al., 1995, p. 16) In Figure 3 the usage of a mixin class is presented (from Gamma et al. (1995, p. 16)).

Figure 3: An example of the usage of a Mixin class



According to Wirfs-Brock & Johnson (1990) there are three different methods for describing the contract that exists between the superclass and the subclass in the implementation of abstract classes. A contract is the list of requests that a client can make to a server (Wirfs-Brock et al., 1990, p. 31). These methods can also make a base for development of derived classes (subclasses) from abstract classes (Wirfs-Brock & Johnson, 1990). The three methods are:

1. Base methods, these methods provide such qualities of a behaviour that is useful to subclasses. The purpose is to implement behaviour in one place that can then be inherited to subclasses.

2. Abstract methods, generate such behaviour that subclasses ought to override. The behaviour does not do anything particularly useful, and the subclasses have to implement the entire method again. The abstract methods are used when specifying the responsibility of the subclasses.

3. Template methods provide step-by-step algorithms. Each step can invoke an abstract method (a method in the class in question) or a base method (a method in the superclass). The purpose of the template method is to create

an abstract definition of an algorithm. The subclass has to implement a specific behaviour, in order to be able to provide the services required by the algorithm.

An abstract class and its methods serve as a *minimal* specification for all the subclasses. An important part of the specification of an abstract class is the specification of every method that will be inherited to the subclasses of the class. From the specification of the methods of the class it has to be clear whether the method is an abstract method that must be overridden, or a base method or a template method that should be directly inherited. (Wirfs-Brock & Johnson, 1990) An abstract class has to be meaningful and capture common patterns, and not only exist as a collection of shared attributes. An abstract class can even be developed with only one subclass if the class consists of something that might be reused in the future. (Selic et al., 1994, p. 258) Below, in Figure 4, is an example of an abstract class programmed in Java (Binder, 1999, p. 535):

Figure 4: An example of an abstract class

```
Abstract class Account {
     abstract Money balance ();
     abstract void credit (Money amount);
     abstract void debit (Money amount);
}
```

Several abstract classes can become a *framework*. Frameworks have to be flexible, so that they can be modified and applicable in as many problem domains as possible (Taivalsaari, 1993, p. 159). Often frameworks implement graphical user interfaces (Tepfenhart & Cusick, 1997).

There are also generic classes. A generic class or a parameterised type is a class that has one or more arguments of unspecified type. In an object-oriented programming language a generic array can, for example, store sometimes reals, sometimes integers and sometimes customers. Generic classes can, for example, be implemented in Eiffel, Ada and C++. For a further discussion on generic classes, see Meyer (1988).

When the interactions between classes and objects are presented, one can also introduce the concepts of coupling and cohesion. In structured design coupling measures the binding between code elements in different modules, and cohesion measures the binding between code elements that are found in the same module (Page-Jones, 1992a). Coupling in object-oriented design is described by Coad & Yourdon (1991, pp. 129-133) as the connections between objects and between classes and coupling can be interaction coupling or inheritance coupling. In object-oriented design we can also talk about the cohesion of a class in terms of the methods defined in the class (Page-Jones, 1992a). Coad & Yourdon (1991, pp. 134-135) present service cohesion, class cohesion and generalization-specialization cohesion. One can further introduce coupling between classes or between methods of the same class or between methods situated in different classes (Page-Jones, 1992a).

It is important to remember when designing abstract classes (superclasses) that the class will be the base for every one of it's subclasses. Everything that is declared has to be

very common, so that it can be inherited to all subclasses. Defining subclasses from superclasses has also to be carried out carefully. Rumbaugh (1996) gives some advice on how to define *useful* subclasses, a subclass has to include all the attributes of its superclasses, a subclass may add more attributes, a subclass should not constrain the values of inherited attributes, an implementation of an inherited operation must be compatible with the behaviour of the implementation in the superclass (because the meaning of an operation should never be overridden by a method in a subclass), and finally, a subclass ought to be made only when it changes the structure of the superclass by adding an attribute, association or operation, etc. (Rumbaugh, 1996).

2.2.2 Discussion about classes

Programmers that work with object-oriented programming languages often work with the client-server model when designing programs. In this model the client is a function or a program that uses the server that is the class. The class and the server are in other words the same thing. The client deals with the server only through the public interface and therefore the only responsibility of the client, and hence also the programmer, is to know this interface. When the class is developed and modified, and when the client is developed and modified, this can be achieved irrespective of each other as long as the interface between them remains the same (Prata, 1991/1992). This is a noteworthy advantage with object-oriented programming compared with corresponding traditional programming. Also Coad et al. (1995, pp. 481-485) and Webster (1995, p. 23) propose that client – server applications and the object-oriented paradigm fit well together.

In traditional programming independent modules can of course be developed, but as long as these modules use common data with other modules, they are not totally independent of the environment in the same manner as the class with its objects, that have both methods and data encapsulated. A problem with the traditional programming approach is that when common data is used among programs or subprograms, a change in the common data can lead to a ripple across all programs that share the data (Fichman & Kemerer, 1993). In the object-oriented paradigm common data is not used (but can however be used) and such ripple effects cannot happen, therefore the object-oriented paradigm is better regarding this issue.

2.3 Relations

There are several relations between objects and classes. Some of these relations have already been presented in this study. Eriksson (1992, pp. 44-49) presents the following:

2.3.1 Class - object relation

The class - object relation exists between objects and classes. An object belongs to a class and has the description of the class. This relation was described earlier in the section on classes.

2.3.2 Uses relation or message-passing relation

The objects send messages to each other and perform each other's operations. The objects use each other. This relation was described in the chapter on objects. In which order the object-oriented program execute, and in which order the objects send messages to each other is managed by a special routine that is called Scheduler (Fagerström, 1993, p. 55). Note that objects can be executed in parallel.

2.3.3 Association relation

An association between two objects implies that two objects are connected to each other. The connection can be one way or two ways. A one-way connection means that one object can refer to the other object but not the other way around. In a two-way connection both objects know of each other and can refer to each other. Usually the association relation is implemented with pointers. (Eriksson, 1992, pp. 44-49) Fagerström (1993, p. 48) explains association as a relation where a class has to "know" of another class.

Association is a relation when an object uses services of another object (Henderson-Sellers, 1992, p. 31). An association is also needed for message passing. If the association is two ways, a pointer is needed in both classes. If the association has another cardinality than one-to-one, one container class is needed for the administration of the association if the association is one way, and if the association is two ways then two container classes are needed. (Eriksson & Penker, 1996, pp. 212-214)

A 'link-attribute' can be attached to an association; in the 'link-attribute' some extra information about the association can be stored. The 'link-attribute' is usually a class and the class has a 'link-object' that administers and represents the 'link-attribute'. In the 'link-object' the date and time of the creation of the 'link-attribute' are usually stored. (Eriksson & Penker, 1996, p. 51) Some object-oriented programming languages do not differentiate between aggregation and association (Henderson-Sellers, 1992, p. 31). In most object-oriented programming languages association is modelled indirectly by a client-server relationship (Henderson-Sellers & Edwards, 1994, p. 56).

2.3.4 Aggregation relation

In the aggregation relation an object consists of several other objects (Eriksson, 1992, pp. 44-49; Fagerström, 1993, p. 32), or a class consists of several other classes (Eriksson & Penker, 1996, pp. 200-201; Fagerström, 1993, p. 46). The aggregation relation is always static (Eriksson, 1992, pp. 44-49). Eriksson & Penker (1996, pp. 200-201) present an example of a car that is an object that consists of a motor that is another object (an aggregation) and four wheels that are also objects (aggregations). Aggregation is in other words a 'has a' or 'consists of' relationship (Henderson-Sellers, 1992, p. 31). In most object-oriented programming languages aggregation is modelled indirectly by a client-server relationship (Henderson-Sellers & Edwards, 1994, p. 56). Aggregation is usually directly implemented in the object-oriented programming language (Fagerström, 1995, p. 99). Aggregation is particularly useful in object-oriented

databases, where groups of objects are held within a container object for management and manipulation (Wilkie, 1993, p. 19).

2.3.5 Inheritance relation

Inheritance represents a 'is a' relationship in a hierarchy of classes (Henderson-Sellers, 1992, p. 31). Both the *generalisation* relation and the *specialisation* relation are inheritance relations. More specialised classes can be developed out of more common classes and more general classes can be developed out of specialised classes. Specialisation and generalisation are types of abstraction (Henderson-Sellers & Edwards, 1994, pp. 44-46). Pant et al. (1996), however, found that class size and complexity might grow because of generalisation, and that it might take 55% extra effort to develop components for reuse compared with developing general components.

2.3.6 Discussion about relations

There are several different types of relations and probably the message-passing relation is the most interesting because in this relation something is moved from one object to another object. This is somewhat like using a procedure or function in a traditional programming language where parameters are passes to the procedure or function.

The class – object relation, the inheritance relation and the aggregation relation are relations used when defining classes or objects and are interesting in a definition sense. The inheritance relation can be very powerful for developing new classes because everything does not need to be developed from scratch.

The association relation is often considered the weakest relation and especially in the analysis phase of information system development it is important to write down which objects need to be aware of each other.

When making modifications in an object-oriented information system and when doing maintenance on an object-oriented information system or application, one has to be careful with the relations. A broken relation in an object-oriented information system or application can be harmful, and before deleting or modifying a class or an object, one has to analyse the relations of the object. In addition, modifying inheritance hierarchies might affect relations and therefore one has to be careful when modifying, adding or deleting classes to inheritance hierarchies.

2.4 Inheritance

Inheritance is often considered the most important object-oriented concept (Al-Ahmad & Steegmans, 2000). This is because inheritance helps in reuse (Radin, 1996), and software developers can avoid coding redundancies by placing new issues in a hierarchy of classes (Fichman & Kemerer, 1993).

The relation between classes where the definition and implementation of one class is based on another class is called inheritance. Through inheritance class hierarchies can be built and in information systems there can be one or several class hierarchies (Eriksson, 1992, p. 49). Inheritance means in practice that the attributes and operations of a superclass are automatically defined for all of its subclasses (Kung et al., 1995).

By using inheritance it is possible to reuse classes and code when developing information systems in such a way where parts of the old system can be used directly by the new parts. Inheritance can be used within a system as well as between different systems. (Korson & McGregor, 1990) When a class inherits from another class, the derived class (the subclass) becomes a precise copy of the base class (the superclass); it is of course possible to bring to the derived class further new parts. The derived class becomes a specialised class of the more general superclass. The derived class has to get a new name and can be modified, which is achieved by giving the derived class new parts. Parts that exist in the superclass can be developed and modified in the derived class as well; it can, for example, be done so that the derived class excludes an inherited part (Selic et al., 1994, p. 261).

Fagerström (1995, pp. 33-34) proposes that when implementing inheritance all attributes are inherited (though it is not sure that all the code in the derived class can use all attributes), all methods are inherited (though it is not sure that all methods can be used by the subclass), methods can be modified, attributes can be added, methods can be added and other relations that the superclass have will be inherited (like associations and aggregations). Note, however, that the inheritance mechanism works a little differently in various object-oriented programming languages (Fagerström, 1995, pp. 33-34). The finding of an object's method in an inheritance hierarchy is attained in the following way; first the class of the object is investigated, if the method is not found there, the investigation goes on in the superclasses one after another (Fagerström, 1995, p. 35).

Development and modification of the derived class have to be performed very carefully otherwise problems will arise. The inheritance mechanism does not copy the code of the superclass to the subclass; the subclass is connected to the superclass by references. When the superclass is modified, the modifications are inherited to the subclasses. This means that a controlled modification of many objects can be achieved easily and simultaneously. In other words, if two classes are linked by an inheritance relation, then the modifications that are made into the superclass will automatically be transferred into the subclass. (Korson & McGregor, 1990)

The class that inherits can be called a derived class, a subclass or a specialised class. The class that the subclass inherits from is called a superclass, a base class, an abstract class or a generalised class (Penker, 1994, p. 17). When starting with a common class and then creating a new class that inherits the common class one can talk about specialisation. The subclass becomes a specialised class and the class specialisation is a powerful, robust and safe way of building modular code (Henderson-Sellers, 1996, p. 19). This is accomplished without any notable risk of damaging the existing and working modules in the system (Henderson-Sellers & Edwards, 1994, p. 23). However, the inheritance hierarchy that is based on conceptual classifications (among classes) is

usually the easiest to understand, maintain, extend and use (Al-Ahmad & Steegmans, 2000).

The modification of a method in a subclass (where the method exists also in a superclass) has to be performed carefully. Inadvertent polymorphism might otherwise occur. An example illustrates the danger. First, a subclass defines a method *check* that does not exist in the superclass. The superclass is then later on modified, so that a new method *check* is added, inadvertently using the same name but with another meaning. Then the method in the subclass overrides the version of the method in the superclass by accident, and the purpose of the program might suffer. (Cockburn, 1993)

An inheritance between classes can be seen as a static activity. New classes inherit qualities, instance variables and methods when the classes are defined and created. The inherited parts exist forever. As an example of how inheritance could be used in an information system, one can present a system with accounts in a bank. First, a class representing a common account with account number, owner, balance and control number, etc. is created. Later a new savings account is created and this is achieved through the savings account inheriting the common account. The savings account can of course be modified and further developed during the inheritance. What is important is that the savings account can be developed from the common account, and one does not have to develop it from scratch.

Inheritance can be classified into extension inheritance and specialisation inheritance. According to Al-Ahmad & Steegmans (2000) inheritance in the object-oriented world should also always belong to one of these inheritance options. Extension inheritance means that the subclass should add new behaviour accompanied with new instance variables. However, the behaviour of the superclass is maintained. In specialisation inheritance everything in the superclass is usually not maintained. The behaviour of the superclass is often modified because of reasons such as correctness or generally speaking specialisation. (Al-Ahmad & Steegmans, 2000)

Finally, one should be careful and remember that there is a difference between class inheritance and interface inheritance (also called subtyping). Class inheritance defines a class (and an object) out of another class. Interface inheritance describes when an object can be used in place of another. Many programming languages (like C++ and Eiffel) use the concept of inheritance for both interface inheritance and class inheritance, the information system developer is therefore forced to carefully study the programming language that is used in order to perform proper inheritance. (Gamma et al., 1995, p. 17).

2.4.1 Multiple inheritance

In multiple inheritance, a class inherits from two or more classes. All object-oriented programming languages do not support multiple inheritance. It can easily generate new qualities that are difficult to anticipate; therefore multiple inheritance has to be used with great care. If a class, for example, inherits from two classes that both have a method with the same name, there will be a conflict, and it might be hard to decide which method to use. The choice can be made by the system or by the programmer.

Often the same problem also arises with instance variables; this problem is, however, best solved by joining a prefix to the instance variables corresponding to the classes (Nierstrasz, 1989).

If a class further inherits two classes that both have the same superclass, then the structure of the superclass will appear two times in the new class (Booch, 1994, p. 64). Wilkie (1993, p. 24) also presents this kind of inheritance and calls it *repeated inheritance*. Repeated inheritance occurs when a subclass acquires the features of a superclass through inheritance several times. For example, we have the original superclass 'Employee'; and then the subclasses 'Manager' and 'Sales Person' both inherit the superclass 'Employee'. If one makes a mistake and makes a multiple inheritance from both the class 'Manager' and the class 'Sales Person' in order to get the subclass 'Sales Manager', the class 'Employee' would then exist twice in the subclass 'Sales Manager'. (Wilkie, 1993, p. 24)

The main advantage with multiple inheritance is the increased power while specifying classes and the increased opportunity for reuse (Putkonen, 1994). According to Koskimies (1995) most researchers, however, are of the opinion that multiple inheritance leads to more problems than benefits and therefore should be avoided. Also Webster (1995, p. 172) warns of the dangers with multiple inheritance. For example, Koskimies (1997, p. 54) presents the following:

- The inheritance structure among classes becomes more complicated, the hierarchical model is not suitable and a network model has to be developed.
- There is a danger of name conflicts. There is a risk that a class inherits several features with the same name.
- There is a danger that a class can be inherited several times.

Some object-oriented programming languages like C++ and Eiffel support multiple inheritance while others such as Oberon-2 and Java do not support it (Koskimies, 1997, p. 51). Java has, however, a support for multiple inheritance of abstract interfaces.

Note also that there are object-oriented programming languages like Smalltalk where everything is an object, including classes and base types like integers and floating point numbers. This means that objects (in fact classes) can be reused by inheritance (Khoshafian & Abnous, 1995, p. 16).

2.4.2 Discussion about inheritance

The inheritance mechanism is often claimed to be the most promising part of the object-oriented concept. Due to the inheritance mechanism many useful things can be done when developing information systems in the business world. The fact that new modules can be developed out of old ones makes the work faster when one does not have to program everything from the very beginning. The inheritance mechanism can be used to an advantage when developing menus, windows and buttons, etc. Madsen (1995) proposes that inheritance is well suited for present classification hierarchies that are tree structured. However, there are researchers like Lauesen (1998) who think that the

object-oriented paradigm is not very useful in the business world. On the other hand there are others like Lam (1997) who are of the opposite opinion and argue the following: “object-orientation models mirror the business systems”.

In many object-oriented software packages there is a superclass or a base class (this superclass is called Object in Smalltalk and Tobject in Object Pascal) that all classes in the package are based upon (Booch, 1994, p. 113). A base for the programming then exists and expectantly programming therefore becomes easier.

A disadvantage with inheritance and a complex class structure is that it can be difficult to gain a proper understanding of how the subclass is constructed. Because objects can send messages to themselves, and methods up and down the hierarchy can execute, all the superclasses to the subclass have to be examined in order to acquire an understanding of the whole inheritance structure. Several levels in the hierarchy have to be examined and this is often referred to as the *yo-yo problem*. (Taenzer et al., 1989) If a superclass gets a new instance variable or method, then all subclasses will also be connected to this new item (in some programming languages this can be stopped and the programmer can also override the methods that the subclass does not need (Winblad et al., 1990)), although the item may have only been designed for one subclass or perhaps some subclasses. By performing appropriate software design this problem can usually be contended with. However, generally the inheritance mechanism is best used when developing an application with a hierarchy. If the application does not have a hierarchy it can be difficult to utilise the inheritance mechanism properly because there are few things that can be reused.

Making modifications to an abstract class (a superclass) is a rather complicated activity because the modifications affect all the subclasses of the abstract class. The changed superclass has to be completely retested, as do all subclasses, and all classes using either the modified class or any of its subclasses (Selic et al., 1994, p. 265). Some authors like Bosch (1997) propose that the main disadvantage of inheritance is that the software engineer usually must have a detailed understanding of the internal functionality of a superclass when overriding superclass methods and when implementing new behaviour to the superclass. However, if one knows the semantics of the method (from the superclass) that will be modified one can rather safely make the necessary modifications. Suitable documentation of the superclass is then needed.

Further the maintenance of an information system might be more difficult due to the inheritance concept. However, according to Selic et al. (119, p. 266) inheritance aids the maintenance process in many ways, it helps us to find the proper abstraction level for a change, it makes it possible to make the change to only one place and thereby avoids error-prone copying, and if supported by tools it automatically makes the change to all desirable places.

2.5 Dynamic binding

Binding means the attaching of a procedure call and the code that has to be executed in response to the call. In static binding the code is known during compilation; in other words there is a permanent linking of a function call to the class type of an object (Pidd,

1995). In dynamic binding the code that will respond to the call of the procedure is not known until the moment of the call at runtime (Korson & McGregor, 1990; Parson & Wand, 1997). When a message is sent to an object, the object and its status variables and class are found with the help of a table of symbols. From the table of methods of the class, the method that corresponds to the message is found. (Korson & McGregor, 1990) Due to dynamic binding (also called late binding) the number of condition cases (for example, IF cases) is reduced, which in turn makes the system less complex (Fagerström, 1995, p. 226).

Gamma et al. (1995, p. 14) present a picture of dynamic binding (quotation):

> Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.

The basic principle of dynamic binding is the possibility to change the realisation of some operations in some subclass in order to get the modified operations performed by the superclass. Operations that are used in dynamic binding are called *virtual*. (Koskimies, 1995) Without dynamic binding the parameters have to be fixed in advance, this should mean that completely common components cannot be built (Korson & McGregor, 1990). Dynamic binding is often used to allow information system developers to subclass and customise existing interfaces (Fayad & Schmidt, 1997). Because dynamic binding is connected with the inheritance mechanism, and because in order to use dynamic binding, one has to understand how programs are compiled and executed, one can argue that dynamic binding is a rather complex concept (Fagerström, 1995, p. 225).

As an example of how dynamic binding could be used in an information system, one can present a system with accounts in a bank. Dynamic binding could be used to find a class with accounting information from a call of the type "check (account)" that will become a message. First the table of symbols is checked from which a reference to the static variables of the object and a reference to the class of the object are found. Then the proper method is found from the table of methods of the class. The methods have different codes that work as a key when finding the method. The method then becomes 'check' and the account is in this way checked. More simply one can say that there is a superclass that has a method "check" (it can be a prototype function). The superclass has two subclasses that both have their own versions of the method 'check'. If a dynamic object then points to a member of the superclass, then it may also point to any member of the subclasses of the superclass. The computer then executes the correct version of the method 'check' depending on the type of the dynamic object.

2.5.1 Comparison of dynamic binding with a conventional solution

It is interesting to compare dynamic binding to a similar solution in a conventional and non object-oriented language. The solution could be an if ... then...else cascade like the following pseudo-code in a conventional language:

```
if (the account is a savings account)
    then CheckSaving(account)
else if (the account is a business account)
    then CheckBusiness(account)
else ...
```

If a new type of account is added, a new extension of the if...then...else cascade must be written, and a new version of the 'check' method must also be written for the new type of account.

If the account object, however, is defined as a dynamic object, the concept of dynamic binding could be used. This makes it easier to add a new type of account. The programmer merely defines a new class, which is a new subclass of the superclass in question, and the subclass has its own version of the method 'check'. This example is based on an illustration by Pidd (1995).

2.5.2 Discussion about dynamic binding

The use of virtual operations and dynamic binding gives the programmer many new possibilities to develop classes and operations that are totally independent. The compiler chooses the code (and class) or operation. Using dynamic binding makes, however, the programs more difficult to understand and more difficult to maintain (Wilde & Matthews, 1993). In order to be able to follow the code that is executed, the superclasses of the objects have to be examined. The hierarchy of the classes has to be examined up and down, and this is probably cumbersome (Koskimies, 1995). According to Harrington (1995) it is important that not only the programmers and system analysts who have developed the program have to understand it, but also other programmers and system analysts as well. Therefore the disadvantage with a program that is difficult to understand because of dynamic binding is a serious one, especially when developing production software, strong typing is therefore recommended (Madsen, 1995).

2.6 Polymorphism

The term polymorphism is Greek and means "many forms" (Taylor, 1990, p. 48). A definition by Booch and Vilot (1990; cited by Henderson-Sellers & Edwards, 1994, p. 71) is as follows (quotation):

> Polymorphism is a concept in type theory in which a name may denote objects of many different classes related by some common base class. Thus, any object denoted by this name is able to respond to some common set of operations in different ways.

Fagerström (1993, p. 25) writes that polymorphism can be explained as an issue where a reference from one object can refer to several other objects from different classes. Often there is, however, a requirement that the objects that are referred to must be in the same hierarchy (Fagerström, 1993, p. 25).

There are many kinds of polymorphism. In general polymorphism is the ability to take more than one form. In an object-oriented programming language, a polymorphic reference is a reference that over time can refer to objects of more than one class, when there is an inheritance hierarchy between the classes (Korson & McGregor, 1990; Parson & Wand, 1997). Different classes can, in other words, have functions that have the same name (Pidd, 1995). When a superclass has several subclasses then the subclasses can have different implementations of the methods of the superclass (Eriksson & Penker, 1996, p. 66). A software developer who is working with the superclass might think that it is the method of the superclass that is used, but in fact it might be a method in one of the subclasses that is actually used (Eriksson & Penker, 1996, p. 66).

Polymorphism can thus be found when there is a connection between inheritance and dynamic binding (Booch, 1994, p. 72). The polymorphic reference can be a static type or a dynamic type (Korson & McGregor, 1990). The static type of a polymorphic reference is determined from the declaration of the object in the program (the static class), but when the program runs, an object in a subclass of the static class can be referred (Koskimies, 1995). The dynamic type of a polymorphic reference can change during the execution of the program. Polymorphism is thus linked with dynamic binding where binding is at run-time (Henderson-Sellers & Edwards, 1994, p. 71).

Deubler & Koestler (1994) found that polymorphism in connection with dynamic binding is a powerful mechanism for avoiding interdependencies among components. More general software can thus be produced. However, dynamic binding cannot be applied if a virtual method has not been developed; the virtual method has furthermore to be exactly the same in all subclasses in question (Deubler & Koestler, 1994).

According to Coleman et al. (1994, p. 218) and Radin (1996) polymorphism also aids reuse. This is so because new components can be used in the same environment as old components without having to modify the calling environment. Polymorphism in connection with dynamic binding also promotes reuse according to Wilkie (1993, p. 2).

However, Parson & Wand (1997) propose that the concept of polymorphism is not always advantageous because the arbitrary overriding and cancelling of inherited features are questionable practices. It is logically incorrect because by definition, everything of a type should apply to its subtypes (Parson & Wand, 1997). Also Binder (1999, p. 26) proposes that polymorphic messages have many advantages, but points out that runtime binding can obscure and fragment control relationships, leading to bugs.

2.6.1 Example of polymorphism

As an example of how polymorphism could be used in a business information system, one can present a system with accounts in a bank. When someone sends a message to a class, then several different objects in different classes can perform the request of the message. If the message concerned some type of control, one object can check the account number, another object checks the balance and a third object checks the owner. Wirfs-Brock et al. (1990, p. 23) present an example with a message `Print`. The message is sent by one object and can be responded to by several printing objects; if a

receiver implements a method with the same signature as the message, it will respond. A signature of an operation is the same as the names and types of information in the object. The combination of signatures that an object can offer is called its protocol (Jacobson et al., 1995, p. 50). In other words, the set of messages to which the object can respond is called the protocol of the object (Henderson-Sellers, 1992, p. 24).

2.6.2 Discussion about polymorphism

Due to polymorphism the application treats the class as a unit with only one type of object (although other classes with objects exist) that can be reached with only one message (Korson & McGregor, 1990). Programs can be developed by using classes without studying all the details in the realisation of the class. Although, for example, Meyer (1988, pp. 484-485) is of the opinion that this is good programming standard, this programming standard can be criticised because the number of classes grows remarkably (Koskimies, 1995). With polymorphism, however, fewer special cases arise and the maintenance of the system often becomes easier and clearer. Money and time can then be saved, which is important for many companies. It is also easier to learn how to use a class than it is to program the corresponding unit (Korson & McGregor, 1990).

As with dynamic binding polymorphism could make it more difficult to understand the program, because the behaviour of the lower level classes might be problematic to understand (Taenzer et al., 1989; Wilde & Matthews, 1993). However, if polymorphism is used in a correct way it supports abstraction and the program might actually be easier to understand.

3 THE OBJECT-ORIENTED PARADIGM

The object-oriented paradigm provides a new way of conceptualising information system development and is considerably different from 'traditional' information system development paradigms including the procedural, logical or functional (Henderson-Sellers, 1992, p. 16). Note that strictly theoretically the object-oriented paradigm has few features of its very own. According to Koskimies (1997, p. 2) inheritance and the aspects of inheritance are the only true exclusive features of the object-oriented paradigm.

The key breakthrough in the object-oriented paradigm according to most researchers is, however, the ability to build large applications from a set of components by reuse (Verity & Schwartz, 1991). However, Räisänen (1997a, p. 16) is of a somewhat different opinion and argues that object-oriented software development methods and object-oriented models are not very suitable for developing large information systems and large software applications. Räisänen (1997a, p. 16) believes this is because object-oriented software development methods and models have a limited support for grouping objects into larger working units. The lack of grouping support might lead to 'ravioli' code where there are a lot of understandable objects, but the communication between the objects is almost impossible to comprehend (Räisänen, 1997a, p. 16).

Since 1997 and the opinions of Räisänen (1997a, p. 16) the object-oriented paradigm has, however, evolved and design patterns, frameworks and packages can be used in order to promote reuse.

3.1 The object-oriented paradigm and the information systems development life cycle

The object-oriented system development life cycle model identifies the three traditional activities of analysis, design and implementation, but the paradigm does not separate the activities as strongly as the procedural approach. Instead the object-oriented system development life cycle model treats the different activities as more compounded and dependent on each other. The primary reason why the different activities can be combined is that they all contain the same elements, the objects. The objects and the relations between the objects are central both in the analysis and in the design. The objects and the relations between the objects are in fact the entire base for the design. (Korson & McGregor, 1990; Nowicki & Kosiak, 1996)

The development process or life cycle of a system convey in which order the phases in a development method ought to be performed; the development process can, according to Eriksson & Penker (1996, p. 99 & pp. 299-300) be:

- Sequential. A traditional approach to describe the development process. For example, the 'waterfall model''.

- Incremental. Every step in the development process is a small evolutionary life cycle with analysis, design, implementation and testing.
- Parallel and incremental. In this development process the development is performed incrementally but several steps are carried out at the same time in parallel. This development process is often called concurrent engineering.
- Iterative. The steps are performed by iteration, for example, in the order analysis-design-analysis.
- Recursive and incremental. In an incremental step iteration can be achieved.
- Fountain like. All increments are carried out without any special order.

Pittman (1993) presents object-oriented system development as iterative or incremental. By 'iterative' Pittman (1993) means a series of solutions to a problem where every single iteration is a part of the solution and satisfies the requirements. Every single iteration is complete, but its accuracy or acceptability gets better with each iteration (Pittman, 1993). By 'incremental' Pittman (1993) means a style where system functionality is built so that every result of each increment is a part of the solution and not an entire solution.

The object-oriented system development process is (there are some exceptions) iterative or incremental, and can follow several different life cycle models. Henderson-Sellers (1996, pp. 5-11) presents the following life cycle models suitable for object-oriented software development:

- The Fountain Model. This life cycle model means that one phase in the process can always fall back on an earlier phase in a fountain like structure (Korson & McGregor, 1990). In this life cycle model the overlap between activities is in focus, though it is accepted that most activities come in a certain order, see Figure 5 (source: Henderson-Sellers & Edwards, 1993; cited by Henderson-Sellers, 1996, p. 7).
- The Pinball Model: In this life cycle model the bumpers and paddles are the various needed activities of an object-oriented life cycle: finding classes, attributes, methods and object relationships; defining collaborations, inheritance, aggregation and subsystems and converting the design to programming code, testing the system, and on-site implementation. The ordering of transition between the different places of activity can be different from one project to another, see Figure 6 (source: Ambler, 1994; cited by Henderson-Sellers, 1996, p. 10; Ambler, 1998, pp. 28-29). This is the case because object-oriented information systems development is iterative (Ambler, 1998, p. 28). Further, the ball represents the current version of the information system under construction. The points scored during the game represent the benefits achieved by the project, when the ball bounces off a bumper number

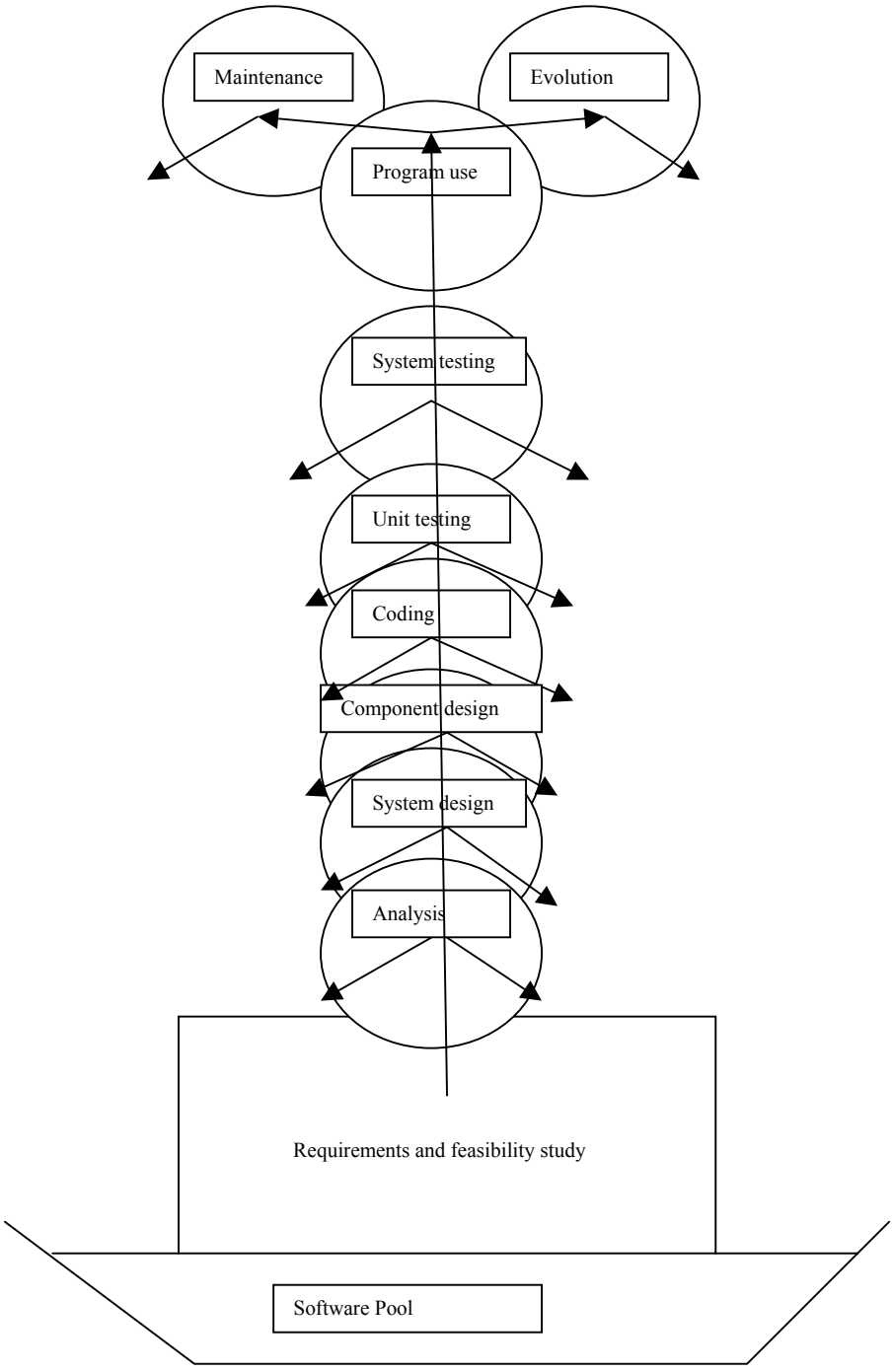
of points are achieved. In the pinball model the player is the project manager who 'guides' the ball through the game. The better the player, the more points can be scored. The paddles represent project resources; the player uses the paddles to keep the ball bouncing between bumpers, whereas a project manager commits resources to keep a project going. The coin (money for playing) and putting the coin in the machine represents the project feasibility study. The plunger represents management approval to begin the project; the project manager (the player) handles the plunger.

- The SOMA OO Model. This model is based on a diagram with boxes. Each box represents an activity, and each activity has an output that is tested. A message-passing technique and some other concepts are vital in the life cycle model. Graham (1995; cited by Henderson-Sellers, 1996, p. 11) presents this life cycle model.
- The Spiral Model. This model that has its origin in the work by Boehm (1986; cited by Henderson-Sellers, 1996, p. 5) and is based on a risk-driven 'spiral' in which iterative and incremental information systems development revolves through four basic activities (assessment of objectives, risk assessment, product development and planning). Though the spiral model is used in several object-oriented information systems development projects there are some researchers that are critical towards the iterative and recursive nature of the spiral life cycle model (as an example of one such critical researcher one can present Berard, 1992, chapter 4; cited by Henderson-Sellers, 1996, p. 5).

The life cycle models presented above are of course not the only models that can be used for object-oriented software development. For example, Pressman (2000, pp. 40-41) presents the Component-based development process model that is based on the spiral model. The iterative structures in the different models do not mean that the object-oriented information systems development process would lack the different phases of analysis, design and implementation. The three phases are of course used in the object-oriented information systems development process too, but as mentioned earlier, the difference is that the phases are more connected to each other (de Champeaux et al., 1993, p. 19; Korson & McGregor, 1990). Usually the real-world objects from the analysis phase are directly translated into objects in the design phase, and further into the implementation phase, so there are objects in all phases according to Taylor (1992, p. 71) and de Champeaux et al. (1993, p. xiv). One can say that there is an integration of analysis, design and implementation within a single framework, using common concepts and notation (Hopkins, 1992).

The life cycle models presented are of course not the only life cycle models. Other life cycle models that are more often used for traditional functional information systems development are, for example, the Waterfall Model (a very popular life cycle model first proposed by W. W. Royce from the US Air Force in a 1970 paper), the V-model, the Sawtooth Model, the Shark Tooth Model and the Issue-based Life Cycle Model. These life cycle models are presented by Bruegge & Dutoit (2000, pp. 471-485) and by several other authors like Sommerville (1996).

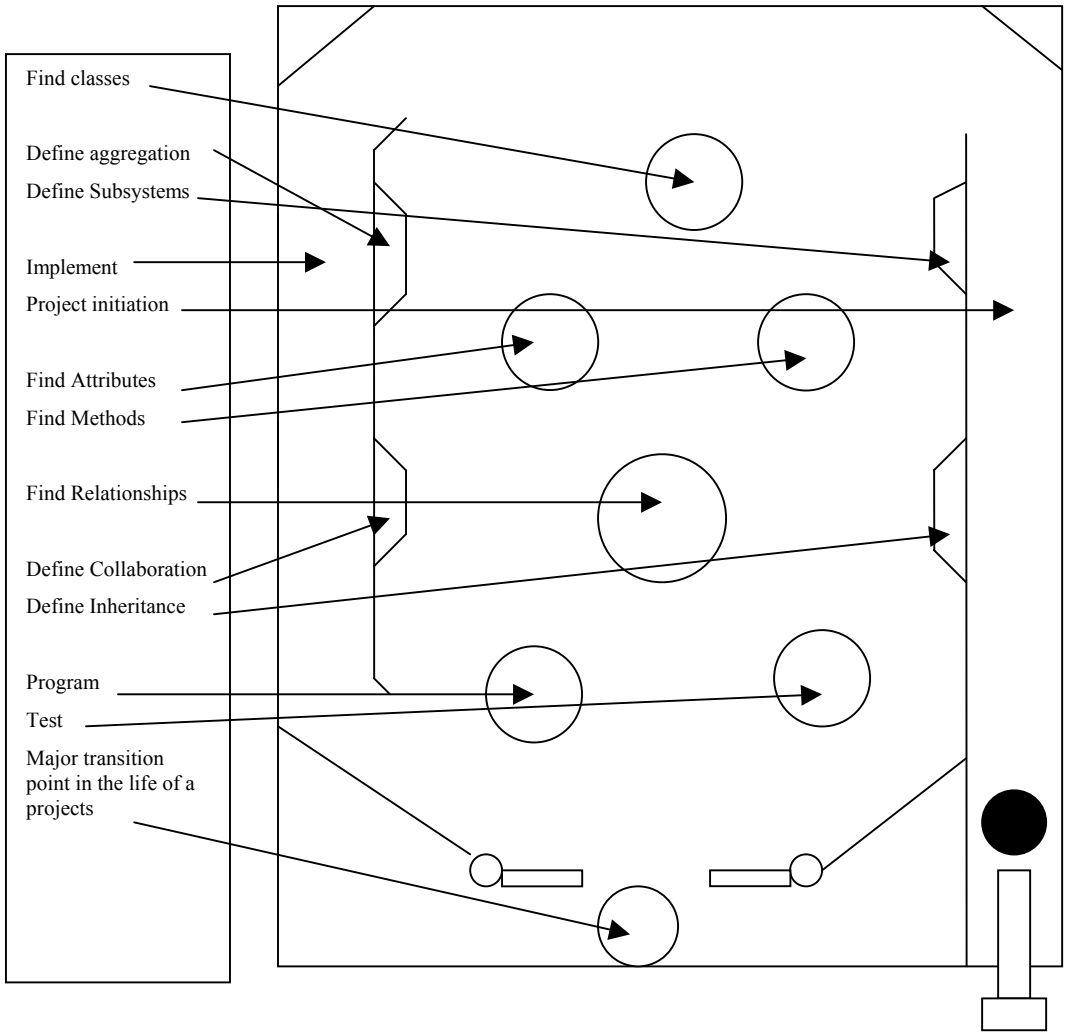
Figure 5: The Fountain model describing the system levels



The iterative or incremental information systems development process models presented above are additionally often suitable for user interaction in the design phase, and therefore user requirements do not have to be 'frozen' as early in the information system development process as in the traditional waterfall life cycle (Henderson-Sellers, 1993; Noack & Schienmann, 1999).

Although there are benefits in using one of the above presented iterative or incremental life cycle models when developing object-oriented information systems there are of course exceptions where another solution has been applied. In a development project reported by de Champeaux et al. (1992) the fountain model was known, but despite that, the waterfall model was followed. One reason for this was the familiarity of the product domain to the system analysts (de Champeaux et al., 1992).

Figure 6. The pinball model



3.2 Discussion about the object-oriented paradigm

If an information system or application is to be built according to the object-oriented paradigm, there will of course be many important questions. Does the software company have knowledge of the object-oriented paradigm? Is the software company presently using the object-oriented paradigm? How shall the software company most profitably use the object-oriented paradigm? What pitfalls are there in using the object-

oriented paradigm? How can pitfalls be avoided? What kind of information systems (and in which fields of applications) can be developed to an advantage according to the object-oriented paradigm? What obvious advantages come with the object-oriented paradigm?

Coad et al. (1995, p. 485) argue that leading firms in nearly every industry use the object-oriented paradigm in some way, including banking, government, insurance, investment, manufacturing and telecommunications, etc. Jacobson et al. (1995, p. 69) and Eriksson (1992, pp. 31-33) are of the same opinion. Interesting, however, is that Smith & McKeen (1996) argue that object-oriented programming is the dominant type of programming for the PC, and that traditional programming usually is used on mainframes. In addition, Henderson-Sellers (1992, p. 261) proposes that object-oriented programming languages are mostly used on PC's and Unix workstations.

Note that according to Petre (2000, p. 3) the object-oriented paradigm in point of fact originated in object-oriented programming, and gained its first notable success with the programming of graphical interfaces. Cockburn (1998, p. 26) and Smith & McKeen (1996) are of the same opinion and argue that the object-oriented paradigm supports graphical user interfaces particularly well, and that modern user interfaces are almost always object-oriented.

Eriksson (1992, pp. 31-33) proposes that the object-oriented paradigm is especially useful when developing simulation applications, CAD/CAM applications, transaction based systems and multimedia applications.

Further Martin & Odell (1995, p. 3) claim that the object-oriented paradigm can be used beneficially as a mechanism that *connects* and *organizes* several different system approaches like information engineering, business, reengineering, client-server development, visual programming, 4GLs, SQL, object-oriented databases, relational databases, knowledge databases, fuzzy logic, generic algorithms and structured techniques, etc.

An interesting question is whether the object-oriented software development is the dominating software development paradigm today (2005). One can look back to gain some understanding on this issue. Yourdon (1994, pp. 17-18; cited by Zhang, 1999, p. 66) proposes that in 1994 only 3.8% of projects in 100 companies had applied the object-oriented paradigm with a 91.7% success rate. According to Helton (1998) the object-oriented paradigm had not been used for developing many major business applications in 1998.

In the beginning of the 1990's, some researchers thought that the object-oriented paradigm would mostly be used for developing technical applications, but in fact, today one can almost certainly say that the object-oriented paradigm has been used in several applications (Love, 1993, p. 82 & p. 96). Martin & Odell (1995, p. 4) argue that the object-oriented paradigm can be used for developing any kind of system, and that object-oriented software development *as a whole* is important, though Zhang (1999, p. 66) argues (out of his statistical investigation of papers published in OOPSLA proceedings and the journal Communications of the ACM) that object-oriented

programming has been studied a lot, and actually much more than object-oriented analysis, for example.

According to Mathiassen et al., (2000, pp. 5-6) traditional paradigms have successfully been used for developing information systems and applications, whose purpose was to automate labour intensive information processing tasks, and because most of these information systems have already been developed, new information systems are now in focus. These new information systems are built *upon* the traditional 'automated' information systems to support individual problem solving, communication and coordination. When developing these 'new' information systems the object-oriented paradigm is useful, because the object-oriented paradigm can successfully be used for developing information systems that are not fundamentally based on handling large amounts of uniform data, but are more focused on managing distributed and specialized data through the organisation. (Mathiassen at al., 2000, p. 6)

3.3 Benefits with the object-oriented paradigm

In an interesting article in Business Week 30th September 1991 the object-oriented paradigm was considered a major contribution to the software community. The object-oriented paradigm was supposed to have an immediate and practical payoff for many reasons, mostly due to easier development, use and reliability of software (Verity & Schwartz, 1991).

According to Love (1993, p. 43) a problem with the traditional functional paradigm is that it requires a conceptual model for an information system that is based on a top down level design, and that this conceptual model is then transformed into an implementation model which is the actual programming code. The transformation is so extreme that mapping from the conceptual model to the programming code and vice versa is difficult to maintain as the information system evolves. A benefit of the object-oriented paradigm is that the conceptual model is retained and becomes more explicit by the virtue of the programming language syntax. The retaining of the conceptual model is important because then the maintainers of the information system can more easily read the programming code and determine what the original software developer had in mind at every single development step.

According to Snyder (1993) the following benefits are connected with the core concepts in the object-oriented paradigm: all objects embody an abstraction, objects provide services, clients issue requests, objects are encapsulated, requests identify operations, requests can identify objects, new objects can be created, operations can be generic, objects can be classified in terms of services, objects can have a common implementation and objects can share partial implementation.

Eriksson (1992, p. 31-33) proposes that the object-oriented paradigm is especially useful when developing graphical user interfaces. Because as much as 50% of the code of a whole information system can be code for the graphical user interface and the functional paradigm is not very suitable for developing graphical user interfaces, the fact that the object-oriented paradigm is useful when developing user interfaces is an important benefit of the object-oriented paradigm (Fagerström, 1995, p. 17).

In graphical object-oriented user interfaces there are icons that represent the objects and the user can manipulate the icons (Capper et al., 1994). When a user selects an icon an event (a method call) occurs that the user interface must manage (Fagerström, 1995, p. 17). This occurrence is not easily implemented with the functional paradigm and therefore most graphical user interfaces are developed with the object-oriented paradigm (Fagerström, 1995, p. 17). For example, a product item can be added to a sales list by selecting the icon that represents the product item and then moving the product item icon onto the sales list icon by "dragging and dropping" (Capper et al., 1994).

Gehringer & Manns (1996) studied the benefits of the object-oriented paradigm by asking managers who direct object-oriented programming projects. The findings were as follows (quotation):

> Rate the next twelve statements on a scale of 1 (=strongly disagree) to 5 (=strongly agree)
>
> The object-oriented paradigm has been of benefit to us by:
>
> | | |
> |---|---|
> | Facilitating software reuse | 3.59 |
> | Making maintenance easier | 3.76 |
> | Decreasing development time (faster time-to-market) | 3.30 |
> | Increasing software quality | 3.61 |
> | Allowing the development of more complex systems | 3.67 |

In previous studies several other benefits with the object-oriented paradigm were found. One difficulty, however, was the fact that the benefits related to many diverse aspects:

- Productivity issues and reuse.
- Life cycle issues (one model, analysis, natural mapping to the problem, domain design, maintenance).
- Complexity issues.
- Quality issues.
- Software components and End user computing (development of applications out of components).
- Portability.

One has to acknowledge that when researching into the object-oriented paradigm some more general benefits might have been overlooked. One difficulty identifying the benefits is that some researchers have different opinions on what benefits there are with the object-oriented paradigm.

3.3.1 Object-oriented analysis

Object-oriented analysis is not in itself a benefit of the object-oriented paradigm but the rather integrated object-oriented analysis and object-oriented design and the one model concept is considered a benefit (in comparison with traditional analysis and design

which is more separated in the software life cycle). There are also some researchers like Mylopoulos et al. (1999) who propose that object-oriented analysis is more powerful than traditional analysis like data flow diagrams and structural analysis.

Analysis is the activity where the task of the information system is described, in other words, what the information system ought to do. Analysis does not tackle problem-solving activities (de Champeaux et al., 1993, p. 7).

Coad & Yourdon (1990, pp. 18-19) propose that analysis is the study of a problem domain, leading to a specification of externally observable behaviour; a complete, consistent, and feasible statement of what is needed; coverage of both functional and quantified operational characteristics (e.g., reliability, availability and performance). During analysis (requirements analysis and elicitation analysis) information systems developers formulate the problem with the end users (the clients) and build the problem domain model (Bruegge & Dutoit, 2000, p. 8).

Larman (2002, p. 6) gives the following definition (quotation):

> Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new computerized library information system is designed, how will it be used?

"Analysis" is a broad term, best qualified as *requirements analysis* (an investigation of the requirements) or *object analysis* (an investigation of the domain objects).

According to Webster (1995, p. 106) analysis consists of identifying and defining the following:

- Problem domain.
- User requirements and needs.
- The methodology to be used, including object model and notation.
- Classes from problem domain.
- Subjects from problem domain.
- Object responsibilities.
- Class hierarchies.
- Object structures (whole and part).
- Class and instance attributes.
- Class and instance methods.
- Object relationships and interactions.
- State transitions.
- Information locations, including persistent information and data dictionaries.
- Event flows and message sequences.
- Dynamic models, including use cases and scenarios.

Object-oriented analysis is concerned with identifying the objects that map into elements of the information system that is to be developed (Lauesen, 1998; Nerson, 1992). In other words, in object-oriented analysis the real world or the domain is modelled into classes and objects and into relations between these (Eriksson, 1992, p.

30). It is important to understand that the objects in analysis should be representations and not implementation driven (Jacobson, 1993; Parson & Wand, 1997). This is important because systems analysis involves modelling a domain, when software design is implementation driven (Parson & Wand, 1997). However, object-oriented analysis is not always easy to carry out; Kaindl (1999) gives the following recommendations for object-oriented analysis (quotation):

- Acquire preliminary requirements.
- Become familiar with the domain as it is.
- Envision the domain after the new system is deployed.
- Focus on either static or dynamic issues first, depending on the domain.
- Combine OO modelling with defining the evolving requirements in natural language.
- Do not yet commit to specific decisions about the object inside the program.

When identifying objects and classes a grammatical method is often used. A description of the system in natural language is written or a requirements specification is used. The nouns become objects, the verbs become operations (Mathiassen et al., 2000, p. 5; Sommerville, 1992, p. 213) and the operations correspond to methods. This way of finding objects is considered as too simple by some researchers (Henderson-Sellers & Edwards, 1994, p. 142), but Pittman (1993) maintains it is useful. Another approach to finding objects is presented by Pittman (1993) who argues that many object-oriented software development *methods* provide good checklists for identifying objects. When identifying operations and class protocol relationships (EER modelling) one can utilise scenarios, class-responsibilities-collaborator cards, finite-state machines, state charts and the entity life history (Pittman, 1993).

Another rule for finding objects is that if something is interesting to talk about, then it has to be an object (Thomas, 1989). When larger information systems are developed the method of finding objects and methods from nouns and verbs does not usually work, one reason for this is that there is probably no formal and correct requirements specification from which nouns and verbs can be sought (Rubin & Goldberg, 1992). Another reason why the method of finding objects and methods from nouns and verbs does not usually work is that the method is based too much on the tangible aspects of a problem; objects that can be seen, heard, felt, etc., are easy to find but conceptual objects are more difficult (Rubin & Goldberg, 1992). Another method than the grammatical method is therefore presented by Rubin & Goldberg (1992) and is called Object Behaviour Analysis; the method is based on system behaviours, and on the items that initiate and participate in the behaviours.

When starting with analysis one can use as a software analysis method, though it is not mandatory. The selection of software analysis method depends on several concerns such as the following: the chosen software development paradigm, the experience of software development methods, the type of information system under construction (Tengvall, 2001, p. 17), the immaturity of software development methods and the

support from software development tools for different software development methods, etc.

Mylopoulos et al. (1999) propose that object-oriented analysis is more powerful than traditional analysis like data flow diagrams and structural analysis. One reason is that object-oriented analysis does not suffer from the well-known problem that exists in traditional top-down software development (Cackowski et al., 2000). In traditional top-down software development software components in the structure become dependent on lower components in the structure, and if a component lower in the structure is modified this will affect the components in the structure, which eventually makes the maintenance of the information system more difficult (Koskimies, 1997, p. 3).

In object-oriented software analysis the components are more independent and usually a bottom-up approach is used. Therefore, a change in a component does not initiate large effects and usually the effects are limited to message passing features and inheritance features. There are, however, more traditional software development methods that are also based on the concept of rather independent parts, like the JSD (Jackson Structured Design) method where the system is modelled as processes that communicate with each other (Koskimies, 1997, p. 3).

Further reasons why object-oriented analysis is advantageous is that the object-oriented paradigm provides a more consistent approach to system modelling, it closely reflects the 'real world', it has an ability to model user interfaces to a system (nowadays this advantage is not that noteworthy, because designing user interfaces is possible through several different tools on the market), and finally, it has several reuse possibilities, like reuse of views from the class hierarchy (Wilkie, 1993, pp. 83-84). Further object-oriented analysis also significantly advances requirements modelling according to some researchers like Mylopoulos et al. (1999) and Wilkie (1993, p. 83).

According to Mylopoulos et al. (1999) object-oriented analysis is also popular because there are good analysis methods like the UML method (Rumbaugh et al., 2000; Solomon, 1999) and because it advances the state of practice in requirements modelling. Johnson et al. (1999) assert that there are better analysis and design models in the object-oriented world than in the traditional world.

UML (Unified Modeling Language) is an object-oriented software development method that is one result from a standardisation request that the Object Management Group (OMG) sent out in June 1996, asking for methods that can be considered as standardisation methods for object-oriented software development. UML combines the methods by Booch, Rumbaugh and Jacobson. Several important computer companies like Hewlett-Packard, Microsoft, Oracle and Texas Instruments support UML. Another standardised modelling language is the modelling language by the OPEN consortium, called the Open Modeling Language. Don Firesmith, Brian Henderson-Sellers and Ian Graham manage The Open Modeling Language (Lam, 1997). Note that Larman (2002, p. 4) has written a book on applying UML and argues that UML is a notation and not a software development method.

In addition, Noack & Schienmann (1999) propose that the UML method (which in fact covers analysis and design) in reality is only a *notion*, and this was agreed by leading IT vendors. Also Ramaswamy (2001) found that many object-oriented developers view object-oriented analysis methods and object-oriented design methods as nothing more than documentation techniques. But Caliò et al. (2000) are of a different opinion and argue that UML introduces an iterative and incremental software life cycle paradigm with a series of iterations that evolve into the final system. The iterations are incorporated in the phases in the Unified Process, and the Unified Process utilises the UML method. The phases in the Unified Process are inception, elaboration, construction and transition. (Jacobson et al., 1999) Every iteration has requirements analysis, domain analysis, system design, implementation and testing. Caliò et al. (2000) propose that UML is a clear and robust notation, with a good conceptual approach, a good methodological approach and a good modelling technique. Another benefit of UML is that it is used in connection with the popular Rational Rose software development tool.

From analysis the next steps are design and implementation: during these steps the objects are transformed into other objects that are slightly different and that make up the actual object-oriented information system. Note that according to Martin & Odell (1995, p. 11) analysis is not an approach that models reality; it is an approach that models the way people understand and process reality.

Design is the activity where the function of the data system is described, in other words, how the data system ought to work. Object-oriented analysis is concerned with the problem domain, and design is concerned with the solution domain (Monarchi & Puhr, 1992). The transition from object-oriented analysis in problem domain to object-oriented design in solution domain is, however, often so smooth that the boundary between object-oriented analysis and object-oriented design is equivocal (Henderson-Sellers, 1992, p. 26).

Mylopoulos et al. (1999) and Johnson et al. (1999) stress the fact that the whole software development process can be made easier when the designer has the same building artefacts from analysis to design and implementation. The artefacts are the objects, the classes, methods, messages and inheritance, etc. (Mylopoulos et al., 1999). Object-oriented analysis also supports two important structuring mechanisms, generalisation and aggregation (Mylopoulos et al., 1999).

Analysis, summary and discussion. Object-oriented analysis is a software development activity and not a pure benefit. However, object-oriented analysis is considered more powerful than traditional analysis and is thus classified as a benefit in object-oriented software development. The ‘benefit’ of object-oriented analysis has a connection to the following ‘real’ benefits and issues:

- Software components and Bottom-Up approach. Software components are more independent in the object-oriented paradigm, compared with the functional paradigm (Koskimies, 1997, p. 3), which makes a bottom-up approach more practicable. Due to the bottom-up approach, a change in

component does not affect as much as a change in a high-level top-down would.

- More natural. The object-oriented paradigm reflects the real world better than other software development paradigms (Wilkie, 1993, pp. 83-84).
- Reuse. Object-oriented analysis has several reuse possibilities (Wilkie, 1993, pp. 83-84). Not only components can be reused but also analysis. This issue is connected to the issues of frameworks and design patterns; if a certain type of information system has been developed then the analysis from the development project can be used perchance in another similar information development project.

Object-oriented analysis and object-oriented design are important activities when developing object-oriented information systems. Because object-oriented analysis is a software development activity, and not a pure benefit, there will be no connections where object-oriented analysis is involved.

3.3.2 Object-oriented design

Object-oriented design is not in itself a benefit of the object-oriented paradigm but the rather integrated object-oriented analysis and object-oriented design and the one model concept is considered a benefit (in comparison with traditional analysis and design which is more separated in the software life).

Design is the activity where the function of the data system is described, in other words, 'how' the data system ought to work. Object-oriented design is a step further from object-oriented analysis, and certain implementation issues are already considered (Eriksson, 1992, p. 30). Analysis results, like specifications, are expounded into hardware/software solutions that can be implemented (Coad & Yourdon, 1991, p. 5). Object-oriented design is often divided into two different subphases that require different skills and perspectives. The first subphase is architectural high-level design and the second subphase is low-level design or detailed design. (Ramaswamy, 2001)

Design is defined (Coad & Yourdon, 1991, p. 5) as (quotation):

> The practice of taking a specification of externally observable behaviour and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.

Larman (2002, pp. 6-7) gives the following definition (quotation):

> Design emphasizes a conceptual solution that fulfils the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented.
>
> As with analysis, the term is best qualified, as object design or database design.

In object-oriented information systems development the information that is developed in the analysis phase becomes an integrated part of the design, instead of only as the

'input' to the design (Johnson et al., 1999; Korson & McGregor, 1990). Here lies the benefit of object-oriented design in comparison with traditional structured design. In traditional structured design (in theory) analysis and design are strictly different activities, and in the transition from analysis to design some information and understanding of the system often get lost (Fagerström, 1995, pp. 16-17). In the object-oriented paradigm there is, however, a close connection between object-oriented analysis and object-oriented design (Mathiassen et al., 2000, p. 6). When there is a comprehensive representation of components in analysis, design and implementation, there will be a better internal consistency, and it will be easier to track analysis and design decisions in the code of the system or application which in turn makes the maintenance of the system easier (Fagerström, 1995, p. 13). There are classes that are used in object-oriented analysis for defining system requirements, and there are classes that are used for describing the information system itself (Mathiassen et al., 2000, p. 6).

There are also other benefits of object-oriented design that are presented by Morris et al. (1996, p. 65), Stevens & Pooley (2000, p. 218) and Wilkie (1993, p. 93), who propose that object-oriented design is associated with inherent modularity, and the client / server relationship between objects creates a framework for weak coupling and strong cohesion which is good. Coupling means the dependence of a module on other modules. A module which has complex and numerous interconnections with other modules is 'tightly coupled', and a module with few interconnections with other modules is 'weakly coupled'. A tightly coupled module is more difficult to maintain because of its heavy interdependences with other modules. (Wilkie, 1993, p. 93)

Cohesion means a measure of how well the parts of a software system 'hold together'. There are three different kinds of cohesion: functional cohesion (all the modules contribute towards the same purpose), sequential cohesion and coincidental cohesion. It is good especially if the functional cohesion is high. (Wilkie, 1993, p. 93)

Ramaswamy (2001) proposes that object-oriented design is a skill in its own right and not necessarily based on analysis skills. If a developer is good in analysis or programming, the developer is not necessarily good in design, though analysis and design are tightly coupled together in the object-oriented paradigm (Ramaswamy, 2001). Object-oriented design is not always easy, and therefore Kaindl (1999) presents the following guidelines for the proper use of the object-oriented analysis model in object-oriented design (quotation):

- Imagine zooming in on the black box representing the proposed system in the OOA model.
- Develop an architectural vision of the proposed system.
- For each object class in the OOA model, answer this question: Will the proposed program need information about corresponding real-world objects?
- For each association in the OOA model, answer this question: Do the object classes associated through it in the OOA model have corresponding object classes in the OOD model?
- Find out if the system needs additional associations.

- Define the architecture using the dynamic part of the OOA model, the architectural vision, and the OOD model under construction as guides.
- If the architecture requires it, define 'additional' object classes in the OOD model.
- Assign responsibilities to OOD objects to achieve the required functionality.

Following guidelines, like the ones above, will probably make it easier to gain the benefits of object-oriented design. Webster (1995, p. 106) also discusses how to do object-oriented design, and recommends that the software developers should specify the following when doing object-oriented design (quotation):

- Class and object internals.
- Abstract vs. concrete classes.
- Data management, including instance ownership and persistence.
- User interface and interactions with the system.
- Subsystems and modules, including interfaces and cohesion and coupling.

Of course different researchers recommend different activities for object-oriented design in the same manner they present different object-oriented design methods. Nevertheless, as one can see from the two recommendations above, the different approaches have some similarities. One difficult question is, however, which activities are analysis activities and which activities are design activities. Roughly, one could say that analysis becomes design when one starts to think of how to implement things (Nerson, 1992).

Performing object-oriented design does not presuppose that the design will be implemented in an object-oriented programming language. An object-oriented design can be implemented in almost any programming language (Sommerville, 1992, p. 216). Whether it is expedient to have a software development paradigm shift between design and implementation is, however, questionable. According to Fayad et al. (1996) software development paradigm shifts in the middle of an information system development project have usually failed. According to de Champeaux et al. (1993, p. 5) panels experts at the OOPSLA/ECOOP 1991 and OOPLSA 1991 conferences proposed that combinations of traditional and object-oriented analysis, design and implementation are problematic. In addition, Meyer (1995, p. 97) warns against a software development paradigm shift in the middle of a software development project, and recommends that one should apply the object-oriented paradigm from analysis to design, implementation and maintenance.

However, Henderson-Sellers & Edwards (1994, pp. 129-131) propose that some companies want to incrementally adopt the object-oriented paradigm in information systems design and therefore they might have traditional analysis and object-oriented design and implementation (F-O-O) or object-oriented analysis and design and

functional implementation (O-O-F). Nevertheless moving from traditional analysis to object-oriented design might not be a good solution.

Design is, however, often influenced by the forthcoming implementation: if there is no implementation language, there is no good context for design decisions. For example, Smalltalk users and C++ users will probably present the same problem with different design solutions because the implementation languages are so different. (Hohmann, 1996)

There are nevertheless some information system development methods that are hybrid; they are both structured and object-oriented. As an example, the PDIT method can be mentioned. The PDIT method has a structured development process with separate data and functions, but also consists of object principles supported by the development process (Repa, 1996).

Analysis, summary and discussion. Object-oriented design is a software development activity and not a pure benefit. The 'benefit' of object-oriented design has a connection to the following 'real' benefits and issues:

- Close connection between object-oriented analysis and object-oriented design. Several researchers and authors (like Mathiassen et al., 2000, p. 6) propose that object-oriented analysis is more closely connected to object-oriented design than functional analysis is connected to functional design. The closer connection is probably advantageous though some software developers might become irritated because of the 'fuzzy' borderline between object-oriented analysis and object-oriented design.
- Object-oriented design is associated with inherent modularity (Wilkie, 1993, p. 93). The modularity most likely has to do with components.
- Reuse. Object-oriented design has several reuse possibilities and it is actually recommended to utilise reuse when performing object-oriented design (Fagerström, 1995, p. 13). This issue is, like the issue of reusing analysis, connected to the issues of frameworks and design patterns; if a certain type of information system has been developed then the design from the development project can conceivably be reused in another similar information development project.

Object-oriented design and object-oriented analysis are important activities when developing object-oriented information systems. Object-oriented design, however, is not a 'real' benefit of the object-oriented paradigm. Because object-oriented design is a software development activity, and not a pure benefit, there will be no connections where object-oriented design is involved.

3.3.3 The one model concept

A significant benefit of the object-oriented paradigm is that the paradigm is a uniform paradigm throughout the information system development process (life-cycle) from analysis to implementation and maintenance (Henderson-Sellers & Edwards, 1990;

Radin, 1996). Because of the uniform paradigm an object identified in the analysis phase of the information systems development process will also be an object in the code, and there is traceability in the process (Eriksson, 1992, p. 27; Ramaswamy, 2001). Traceability can only be attained if there are connections between analysis, design and implementation (Eriksson & Penker, 1996, p. 101).

The one model benefit goes from the problem domain to code and maintenance (Coad et al., 1995, pp. 481-485). In a traditional approach data flow charts and hierarchy charts are often used in analysis and/or design, and then these different models are translated into a third model for detailed design and coding where the characteristics of the procedural programming language are considered (Henderson-Sellers, 1992, pp. 21-22). The change from analysis to design in a traditional approach (following, for example, the 'waterfall life cycle') is often difficult because there is often a need to dramatically change the structure and components of an information system, because the analysis and design do not necessarily always 'fit' together. Another problem is that often a change in the requirements is not integrated in the design, and therefore the design does not fully represent the information system anymore. (Radin, 1996)

Johnson (2000) found in his study that other benefits of the one model concept are easier modelling, more understandable analysis and design models and easier transition between phases. But Lauesen (1998) found that there was a good ‘one model’ solution mostly in technical applications and in business applications the object operations rarely transferred seamlessly from analysis to design.

Analysis, summary and discussion. The one model concept is a benefit that predominately is due to the following issue:

- The object-oriented paradigm. The object-oriented paradigm is a uniform paradigm throughout analysis to implementation and maintenance (Henderson-Sellers & Edwards, 1990; Radin, 1996). According to Lauesen (1998) this is, however, only true for technical applications.

The object-oriented paradigm can be used in combination with several life cycle models and the life cycle model affects the uniformity of the software development work. Nevertheless, one can probably argue that the object-oriented paradigm per se is more uniform than other software development paradigms. However, the study by Sheetz (2002) concluded that several information systems developers have difficulties to understand the one model concept when developing object-oriented information systems.

The following possible association between benefits has been identified:

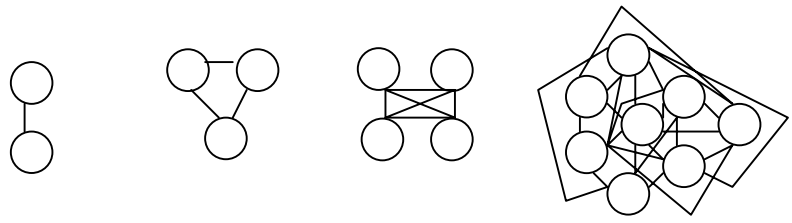
- THE OBJECT-ORIENTED PARADIGM is usually based on a -> ONE MODEL information systems development life cycle.

3.3.4 Management of complexity

According to Edsger Dijkstra who presented the famous paper "Programming considered as a human activity" in 1965, software systems beyond a certain size are too complex to be entirely comprehended by a single human (Dijkstra, 1965; cited by Webster, 1995, p. 22). The question of complexity is significant, and several software developers have been working on the problem of how to reduce complexity in information systems development, and structured programming can be considered as one result in this area (Webster, 1995, p. 22).

Pomberger & Blaschek (1996, pp. 177-178) give an example of how the complexity of an information system emerges; when a system consists of a multitude of individual parts like data, files and functions, and there are relationships between them, the number of possible interconnections rises more than linearly with the number of components, as the succession in Figure 7 demonstrates (the figure is from Pomberger & Blaschek, 1996, p. 178). The quadratic rise in the number of possible interconnections means that the number of potential relations quadruples when the number of components doubles. When developing information systems the software developers do not only need to understand the different components, they also need to understand the huge number of interconnections between the components. One solution to the dilemma of huge number of interconnections is the use of modules with encapsulated connections. (Pomberger & Blaschek, 1996, p. 178)

Figure 7: Rise in the complexity with increasing number of components



Components:			
2	2	4	n
Interconnections:			
1	3	6	$\frac{n * (n-1)}{2}$

The object-oriented paradigm is well suited for developing large *complex* business information systems (Brunet et al., 1994; Cackowski et al., 2000; de Champeaux et al., 1993, p. xiv; Graham, 2001, p. 41; Jacobson et al., 1995, pp. 45-48; Love, 1993, p. 90; Martin & Odell, 1992, p. xi; Wadden, 1999; Wegenast, 1998).

Business systems are often large and complex information systems, and according to Taivalsaari (1993, p. 272) the object-oriented paradigm usually makes it possible to make classes or objects that correspond to the real world, and by reusing these classes and objects, business applications can more easily be developed. This is feasible because although the number of reusable classes and objects might be limited, it is still possible to make enough reusable classes or objects for most business application needs. Typical objects in business applications are customers, accounts, manufacturing processes, reports, bills and orders, etc. (Eriksson & Penker, 1996, pp. 41-42). Also Henderson-Sellers & Edwards (1994, p. 5) propose that the object-oriented paradigm is better for developing business applications with business rules, products and services than traditional information systems development.

In fact, the object-oriented paradigm has several principles for dealing with complexity:

- *Abstraction* (Henders, 1998; Wadden, 1999). Blair & Lea (1992) and Fichman & Kemerer (1993) consider abstraction as perhaps the most important tool for managing complexity, by using the abstraction principle one tries to select only part of the thing under consideration. Note that polymorphism supports abstraction and reuse.
- *Application frameworks.* Application frameworks can also be used which means that reuse can be utilised and applications in complex business and other domains can be developed out of 'semi-complete applications' (Fayad & Schmidt, 1997). Because frameworks cover a large part of a system, reusing frameworks gives more often than not a notable pay back according to McClure (1996). Examples of areas that are often covered by frameworks are data access, user interfaces and security issues (McClure, 1996).
- *Association* can be used to tie together things (Coad & Yourdon, 1991, pp. 6-9).
- *Classification* is often part of the system structure (Nerson, 1992) and abstraction is the base for classification. Classification is the conception of grouping software ideas into classes of things (Henderson-Sellers, 1992, p. 21). Madsen (1995) proposes that a hierarchy of classes and subclasses through inheritance usually represents classification.
- *Communication with messages* (Gall et al., 1995).
- *Decomposition*. The possibility to decompose complex information into small and reusable parts (Jean, 1992).
- *Dynamic binding* which is advantageous because the number of condition cases is reduced (Fagerström, 1995, p. 26).
- *Encapsulation* supports abstraction and information hiding which makes rework easier when developing new information systems (Coad & Yourdon, 1991, pp. 6-9).
- *Hierarchy.* According to Booch (1994, pp. 16-25) and Rinat & Magidor (1996) the object-oriented paradigm is well suited for developing large complex business information systems because the complexity of a large and complex

business information system is often structured in a *hierarchy*. Booch (1994, p. 59) defines a hierarchy as a ranking or ordering of abstractions. A hierarchy is fairly easy to incorporate in the object-oriented paradigm, due to the possibility of developing equivalent hierarchies with classes and inheritance (Booch, 1994, pp. 16-25; Rinat & Magidor, 1996).

- *Inheritance* that makes it possible to specialise (Gall et al., 1995).
- *More natural.* Taylor (1990, p. 24) and Martin & Odell (1992, p. 78) propose that there is a similarity between the hierarchical structure of the object-oriented information system and the assumed human knowledge; both are structured in a hierarchy. Object-oriented systems are believed to *reflect* the real world more accurately (Webster, 1995, p. 58; Wilkie, 1993, p. 39).
- *Pervading methods of organisation* which means that object-orientation supports the way people organise their thinking (Coad & Yourdon, 1991, pp. 6-9).
- *Reuse*. Reuse of existing components makes it easier to develop more complex information systems (Hopkins, 2000).
- *Scale.* Scale means a principle that pertains to the whole-part principle that helps an observer to manage something very large without being overwhelmed (Coad & Yourdon, 1991, pp. 6-9).

According to Gehringer & Manns (1996) both programmers and managers recognise that the importance of the object-oriented paradigm increases as the information systems increase in complexity. Berg et al. (1995) describe a large development project of an operating system that used the object-oriented paradigm; the project contained 14,000 classes, 90,000 methods and 2 million lines of C++ integrated into 20 million lines of total code, and almost 10,000 classes were inherited from some base class. This complex project was a success by most measures. In fact, Berg et al. (1995) propose that the object-oriented paradigm is probably the best paradigm when building large and complex information systems.

However, Korson & Vaishnavi (1992) propose that some companies that have used the object-oriented paradigm when developing large and complex information systems have not experienced all its benefits, mostly due to a lack of knowledge about how to manage object-oriented projects. Also Lauesen (1998) proposes that there are experiences from the business world that illustrate that it is actually problematic to develop business information systems with the object-oriented paradigm.

In the software development community today, increasingly complex applications and information systems are needed. For example, Malan et al. (1995) mention plug-and-play strategies and off-the-shelf standard parts. They further propose that the object-oriented paradigm is viewed as a key enabler for agile and responsive software engineering, software development and software-driven product development.

Analysis, summary and discussion. According to previous studies better management of complexity is predominantly claimed to be due to the following issues:

- The object-oriented paradigm makes *abstraction* possible, which makes the management of complexity easier (Henders, 1998). Abstractions per se probably do not make anything easier, but skilful use of abstractions might. Skilful use of abstraction and reuse can, for example, be performed by polymorphism.
- Complex information systems are usually *hierarchical*, and the hierarchical structure of object-oriented systems fits well with hierarchical information systems (Booch, 1994, pp. 16-25). This claim is, however, rather restricted; first, there are perhaps not so many complex software systems that are 'hierarchical' (though Vossos et al. (1991) report of legal systems that are hierarchical), and second the hierarchy of an object-oriented system makes it probably more complex and difficult to understand.
- More *natural*. Object-oriented systems better reflect the real world, which makes the management of complex real world information systems easier (Webster, 1995, p. 58; Wilkie, 1993, p. 39). This is probably the case when developing information systems that consist of concrete things like machines, products and customers. Nevertheless, if the information system consists of more abstract things like customer relationships, manufacturing processes and cost analyses then the reflection is probably more difficult to obtain.
- *Reuse*. Reuse of existing components makes it easier to develop more complex information systems (Hopkins, 2000). Due to reuse the complexity of information systems can somewhat be controlled. Software developers can reuse several things, like design and components when developing new software systems. Because much can be reused, the software development work becomes easier and the complexity of the new software system can easier be controlled.

The following possible associations between benefits have been identified:

- The object-oriented paradigm is considered more NATURAL -> which makes the MANAGEMENT OF COMPLEXITY easier.
- The REUSE concepts makes it possible to reuse components and other artefacts -> which makes the MANAGEMENT OF COMPLEXITY easier.

3.3.5 Productivity, faster development and reduced costs

Time saving comparisons between different software development paradigms in information systems development can only be estimated roughly, because in order to achieve exact information on this issue, one would be forced to develop exactly the same information system with two or more different software development paradigms and then compare the results. At professional production sites this is impossible out of cost reasons, and even in research organisations this is difficult. One reason is that if an information system would first be developed with software development paradigm A (for example, the functional paradigm), and then the same information system would be developed with paradigm B (for example, the object-oriented paradigm), the experience from developing with paradigm A would effect the development with paradigm B. The

selection of new software developers for the work with paradigm B (with exactly the same experience as the software developers had when they started to work with paradigm A) would probably be almost impossible. (Pomberger & Blaschek, 1996, p. 282)

Information systems development can be made *easier* and *faster* if the work is done in an object-oriented way and if reuse is utilised (Bhattacherjee & Gerlach, 1998; Caliò et al., 2000; Henderson-Sellers & Edwards, 1990; Love, 1993, p. 85; Manhes, 1998; Meyer, 1995, p. 107; Musakka, 1996; Nowicki & Kosiak, 1996; Johnson et al., 1999; Sheetz & Tegarden, 1996; Smith & McKeen, 1996; Verschoor & Low, 1994). By reusing existing and tested artefacts like classes, components, design and database objects the information systems developers do not have to construct a solution for a common task repeatedly.

According to O'Connor et al. (1994) reuse will make it possible to achieve immediate and effective solutions for information systems development for customers. Without reuse object-oriented information systems development is usually slower than traditional development (Koskimies, 1997, p. 5). Räisänen (1997a, p. 12) claims that faster development is due to reuse, to the uniform object-oriented software development process and also to *object-oriented thinking*. One has also of course to remember that the experience of the programmers and software developers highly affects the productivity and development time in an information system development project. Out of all the stages in a software development life cycle, the programming stage is the stage that probably gains the most from using the object-oriented paradigm (because it becomes faster). The stages of analysis, design, testing, rollout, installation and training, etc., are not effected so much by the object-oriented paradigm as programming (and the stages are probably as time consuming as they were when developing functional information systems), and the speed of programming is very much effected by the experience of the programmers. (Cockburn, 1998, p. 25) According to Cockburn (1998, p. 25) the programming time will be reduced only if experienced programmers (with more than 12 months of active object-oriented programming behind them) are used.

Jenz (1999a) and Szyperski (1999, p. 7) point out that employing reuse and components *makes it possible* to achieve better information systems and software development productivity. Productivity gains are also achieved because a component or module is encapsulated and the software developer only needs to understand the interface of the component or module, which is much easier than to investigate how the component or module actually works (Stevens & Pooley, 2000, p. 9). Furthermore, the connection of CASE tools with class repositories and class libraries makes information systems development faster, because components in the class repository can be seen, customised and interlinked on the CASE tools screen (Martin & Odell, 1992, p. 33).

Coad et al. (1995, pp. 481-485), Gillach & Deyo (1993) and Radin (1996) propose that speed and frequent tangible working results are considered a benefit, and that when object-oriented development is used from analysis to implementation, it is easier to perform rapid prototyping and acquire tangible working results. Noack & Schienmann (1999) also point out that early delivery of software products can be achieved more effortlessly if the object-oriented paradigm is used.

Jenz (1999a) and Räisänen (1997a, p. 11) also propose that exercising reuse and using components will reduce costs and result in less need for financial resources for information systems development. In addition, patterns and architectures support lower development and maintenance costs and enforce a higher level of reuse (Bhattacherjee & Gerlach, 1998; Mellor & Johnson, 1997).

However, starting to apply reuse can result in higher *initial costs* and even slow return on investment (3-4 years) according to Joos (1994). Also Räisänen (1997a, p. 12) argues that one has to use the object-oriented paradigm for some time before one can experience that utilising reuse will actually lower costs. Bhattacherjee & Gerlach (1998) are of the same opinion as Joos (1994) and point out that the high initial costs are due to issues like extensive training, paying external consultants and buying expensive object-oriented information systems development tools, etc.

Page-Jones (1992b), Page-Jones (1998), Räisänen (1997a, p. 12) and Webster (1995, p. 60) also point out that the *first* object-oriented project in an organisation will probably get little productivity gain because new reusable classes have to be developed, and there is nothing or very little existing software to reuse. Usually the reusable classes are put in a special reuse library for reuse in the future according to McClure (1996).

Improved *productivity* was an experienced benefit of the object-oriented paradigm in the *Survey of Advanced Technology 1996* (Pickering, 1996). Object-oriented software development methods were also found to improve *productivity* by Aksit & Bergmans (1992) and Basili et al. (1996a). In 1996 Harrison et al. (1996) also evaluated functional and object-oriented programs. They found that functional (SML) was easier to debug than object-oriented code (C++), and that object-oriented code was faster to compile and run than functional code in the development process. In addition, Watanabe (1997) proposes that object-oriented software development contributes a great deal to software development productivity. Capper et al. (1994) present a case study where better productivity through the use of inheritance was experienced. However, inheritance is a part of the reuse concept, which is the key to better productivity. Better productivity from reuse does not, however, necessarily shorten the time for the product to the market according to Lim (1994), although Petre (2000, p. 2) is of an opposite opinion. In order to reduce the time for the product to the market reuse must be used effectively on the critical path of the information systems development project (Lim, 1994).

An interesting case is presented by Love (1993, pp. 95-95) where American Airlines developed a system for supporting dispatches with 200 classes and 2,000 methods with over 150,000 objects in active memory at a time. Only three software developers built this information system in only eight months. The productivity was amazing and one can argue that the quality of the system did not suffer because only two errors were found after deployment.

It might be difficult to estimate the increase in productivity; Page-Jones (1992b) has heard about an increase of 15:1, but the writer claims that the increase is actually only 5:1 and requires the development of a good library with classes and about five years of elapsed time. Martin & Odell (1992, p. 37) report that several organisations have been able to develop 80% of the code in new projects out of existing reusable classes, then

only 20% of the code is new. There are also of course reusability figures as high as 90%, but even 80% is still often difficult to achieve (Martin & Odell, 1992, p. 37). Pomberger & Blaschek (1996, p. 283) propose that the savings in code to be written when using the object-oriented paradigm is 25 – 50%, but the *total amount* of code will actually rise (120-300%) because of all the imported code.

One has also to notice that system development costs are not reduced by 5:1, because the requirements analysis is the same and object-oriented information development requires writing new code, understanding what can be reused and how to reuse it and building the library with classes or components, etc. (Page-Jones, 1992b). However, according to Nowicki & Kosiak (1996) the costs will be reduced.

Agarwal et al. (2000) argue that higher productivity is not gained automatically by adopting the object-oriented paradigm and by requiring information system developers to use the object-oriented paradigm. They often need some help on how to use specific reusable classes, and the owner or manager of the specific reusable class is often needed to guide the information systems developer in how to use the reusable class in question (McClure, 1996). The usability of the object-oriented paradigm must therefore be considered; otherwise the productivity of the information system developers might decrease because they consider the object-oriented paradigm as too complex (Agarwal et al., 2000; Malan et al., 1995). Another question is if the reusable class is appropriate for the information systems developer, but the owner (manager) of a reusable class can probably give some help (McClure, 1996).

Analysis, summary and discussion. According to previous studies higher productivity and faster development are be due to the following issues:

- *Reuse* (Bhattacherjee & Gerlach, 1998; Caliò et al., 2000; Henderson-Sellers & Edwards, 1990; Love, 1993, p. 85; Manhes, 1998; Musakka, 1996; Nowicki & Kosiak, 1996; Sheetz & Tegarden, 1996; Smith & McKeen, 1996), it is easy to understand that reuse leads to faster development and that faster development improves productivity. If one reuses then one does not have to develop everything from scratch. But if one reuses, one has to perform the reuse work carefully (Jézéquel & Meyer, 1997).
- *The object-oriented software development process and object-oriented thinking* (Räisänen, 1997a, p. 12). This argument is well known. It presumes that the object-oriented paradigm by itself leads to better productivity, which probably is not the case. If few things can be reused then the productivity gains might be rather minor.
- *The connection of CASE tools with class repositories and class libraries* (Martin & Odell, 1992, p. 33). This is most likely true in theory, but if no CASE tools that are connected to repositories can be found, then there will be no productivity gain in the software development work.
- *Object-oriented software development methods* (Aksit & Bergmans, 1992; Basili et al., 1996a). It might be extremely difficult to compare the efficiency between traditional functional software development methods and object-oriented software development methods. There are a vast number of both

traditional functional software development methods and object-oriented software development methods, and to compare them would be a tremendous task which would include difficult questions like ‘which methods ought to be compared?’ and ‘how does one compare the methods?’ etc. The conclusion is that this argument must be a rather subjective argument presented by the researchers mentioned above.

The reduced cost was predominantly due to the following reasons:

- *Reuse and use of components* (Jenz, 1999a; Räisänen, 1997a, p. 11), it is rather obvious that if something can be made faster by reuse it will also be cheaper because one can save in labour costs. Reuse shows the way to faster development, which leads to better productivity which in turn leads to better efficiency and lower costs. However, if the quality is suffering from faster software development work, then the maintenance costs might be higher so might the total costs for the life cycle of the information system.
- *Patterns and architectures* (Bhattacherjee & Gerlach, 1998; Mellor & Johnson, 1997), if these reuse related concepts are used correctly this is almost certainly the case. Nevertheless, if patterns and architectures are difficult to use then the costs will probably not be lower because the decrease in labour hours due to reuse will be counterbalanced by the extra effort to learn how the patterns and architectures are constructed.

Reuse seems to be the most important issue when attempting to achieve higher productivity. Different object-oriented software development methods, tools and patterns can of course also affect the productivity in a positive manner, but there are also good functional software development methods and tools on the market, so it is almost impossible to state that using object-oriented software development methods or tools, will increase productivity compared with using traditional ones.

The following possible association between benefits has been identified:

- The utilisation of REUSE -> results in FASTER DEVELOPMENT -> which results in better PRODUCTIVITY -> which affects the EFFICIENCY of the information systems development project -> which leads to REDUCED COSTS.

3.3.6 Quality and usability

When talking about information systems quality one can consider the issue of what is *meant* by quality. In a study by Capper et al. (1994) the concept of quality was divided into code quality, correctness, usability, adaptive maintainability, perfective maintainability and performance. Love (1993, p. 186) suggests that quality is ‘conformance to requirements’. Reeves & Bednar (1994) propose that the quality is ‘conformance to specification’.

Due to the object-oriented paradigm, the *quality* of the information system can be improved because programs are made of existing tested components and not developed

from scratch every time (Gillach & Deyo, 1993; Graham, 2001, p. 41; Jenz, 1999b; Johnson et al., 1999; Lim, 1994; Love, 1993, p. 80; Martin & Odell, 1992, p. 32; Räisänen, 1997a, p. 13; Sheetz & Tegarden, 1996; Sim & Wright, 2002; Smith & McKeen, 1996; Stevens & Pooley, 2000, p. 9; Taylor, 1990, p. 104). The quality in question is the 'conformance to specification' quality defined by Reeves & Bednar (1994).

By reusing tested components fewer faults and errors occur, the user interface becomes better and the quality of the information system becomes higher (Basili et al., 1996a; Graham, 2001, p. 41; Watson et al., 2004). This is due to the fact that when products are used several times the defect fixes from each reuse accumulate, which results in higher quality (Lim, 1994). Love (1993, pp. 188-189) proposes that the object-oriented paradigm produces software of better quality because of the following reasons (quotation):

- Objects accept only a finite number of messages as inputs. One object cannot access the data within another object. This encapsulation simplifies both debugging and testing. This fundamental structure makes test scripts relatively easy to develop and to use – tests are developed for each class at the same time the class is being developed.
- In languages that support dynamic binding, a message replaces numerous branching statements required to accomplish a desired behaviour. This can significantly decrease control flow complexity.
- Classes are stable chunks of methods and data structures that can be reused in a variety of systems and applications. Thus, their reliability can improve steadily. Reusability ratios between systems will increase from their current levels of 5-15% with traditional methods to 60-90% with object-oriented technology. This means that a lot of code will be used again and again.
- The size of systems decreases due to dynamic binding and inheritance, so errors are easier to find.
- The model in the designer's head is directly expressed in the software itself. Unlike procedural languages, designs are actually preserved in the source text of the software, making the code more comprehensible to future maintenance engineers.

Reliability is a further benefit of the object-oriented paradigm (Lim, 1994; Page-Jones (1992b). Coad et al. (1995, pp. 481-485) are of the opinion that information systems that are based on problem domain classes tend to be more stable over time compared to information systems based on functions and data.

Lim (1994) suggests that reuse provides incentives to remove bugs and prevent defects earlier in the life cycle of the product because the cost of prevention and debugging can be amortized over a larger number of users.

Coad & Yourdon (1991, p. 15) and Pomberger & Blaschek (1996, pp. 288-289) also propose that object-oriented design improves software quality. When one is using reuse, one can manage with less code, which probably means fewer bugs and the quality of the information system becomes better (Finch, 1998). In certain cases presented by Capper

et al. (1994) the better quality was primarily due to reuse from inheritance and from encapsulation; it applied to both object-oriented design and object-oriented information systems programming. Furthermore, design decisions can be encapsulated in the object-oriented paradigm, which reduces the damage of requirements changes and leads to higher quality (Cockburn, 1998, p. 2).

Another reason why encapsulation gives birth to better quality is because data structures have better integrity when they are encapsulated, and therefore more difficult to access for users (Martin & Odell, 1992, p. 33). This is, for example, important in many distributed, co-operative, and client-server systems. As a more concrete explanation one can present the case presented by Wilkie (1993, p. 39); since the only information about objects that is available to software developers is the information about messages and responsibilities of the objects, one can change the logic of the objects without fear of a ripple effect on any logic in another object.

Object-oriented application frameworks can also increase software quality and make development easier. Frameworks should, however, be used with great care in order to obtain the proposed benefits (Fayad, 2000).

Capper et al. (1994) then studied three cases with object-oriented information system development at IBM in the United Kingdom, and came to the conclusion that the quality of the developed object-oriented information systems improved more than the quality of similar functionally developed information systems. However, the information systems developed at IBM in the United Kingdom were not 100% object-oriented (Capper et al., 1994).

Bäumer et al. (1996) gives an account on experienced better software quality in a large banking software development project. Improved quality was also an experienced benefit of the object-oriented paradigm in the *Survey of Advanced Technology 1996* (Pickering, 1996). In 1996 Harrison et al. (1996) also evaluated functional and object-oriented programs. They found no significant differences between these programs regarding quality. Capper et al. (1994) observed that the issue of quality is based a great deal on the experience and skills of the information systems developers; using a specific information systems development paradigm does not as such produce better quality.

Analysis, summary and discussion. According to previous studies better quality and higher usability are due to the following issues:

- *Reuse.* Reuse is utilised and programs are made of existing, tested components and not developed from scratch every time. A presupposition for this contention is that the components themselves are very well tested and of high quality. Out of personal experience the author of this study claims that this is not always the case. When working as a systems analyst in 1990 in a major Finnish software company, an unpleasant error in a component that had been reused for more than ten years was found, and the component had been considered as high quality.
- *Encapsulation* (Love, 1993, pp. 188-189). Because an object cannot generally access data within another object, both testing and debugging is easier. The

encapsulation structure makes test scripts relatively easy to develop and use, and this easier testing and debugging should produce better quality. The encapsulation concept is not always easy to control and understand. If the software developers are not comfortable with encapsulation the testing will probably not be easier.

- *Dynamic binding* (Love, 1993, pp. 188-189). Due to dynamic binding the control flow complexity will decrease, which generates better quality. However, dynamic binding is probably the most challenging object-oriented concept and if the software developers do not manage dynamic binding, its use will probably make the quality of the information system lower, not higher.
- *Classes* (Love, 1993, pp. 188-189). Classes are examples of components and the reuse of tested classes through inheritance improves quality. This issue is very much like the first issue.
- *The size of an object-oriented information system is smaller* (Finch, 1998). The size of object-oriented information systems is smaller due to dynamic binding and inheritance, and smaller information systems are usually easier to comprehend, which makes maintenance, testing and debugging easier. This is not always the case; there are design issues that affect the understandability of programs. Therefore, a small but badly designed program is probably harder to understand than a large but well designed one.
- *The model in the designers mind can directly be expressed in the software itself* (Love, 1993, pp. 188-189). The object-oriented paradigm makes it possible for designs to be preserved in the programming code, which subsequently makes the information system easier to understand and maintain. This line of reasoning probably has its origin in the 'one model' concept in object-oriented analysis and design (Sheetz & Tegarden, 1996) that reduces the difficulty to map problem constructs from the problem domain with programming structures and program code. Someone could, however, argue that a 'one model' approach is fuzzier than a stepwise approach with analysis and design, although this is probably a subjective question.
- *Application frameworks* (Fayad, 2000). If the application frameworks are of high quality, their use most certainly increases software quality in the same manner as reuse increases software quality. One has, of course, also to understand how to use and manage application frameworks in order to achieve better quality when developing information systems; if application frameworks are used in the wrong way the quality improvements might be very moderate.

The reuse concept combined with software components and the 'one model' concept seems to be the most fundamental basis for achieving better software quality when doing object-oriented information systems development.

The following possible associations between benefits have been identified:

- Utilising REUSE and reusing tested components -> results in HIGHER QUALITY.

- The ONE MODEL concept that makes it possible to embed design issues in the programming code -> results in HIGHER QUALITY because the information system is easier to understand.
- Using SOFTWARE COMPONENTS like classes -> results in better USABILITY because tested components can be used.

3.3.7 Natural and better mapping to the problem domain

In the object-oriented paradigm *the connection to the mind of the human being* makes it easier for system developers to build object-oriented information systems than traditional information systems. This is the case because it is easier to look at the world as a collection of objects than it is to look at the world as a collection of functions and data (Booch, 1994, p. 287; Castelluccio, 1997; Mathiassen et al., 2000, p. 5; Räisänen, 1997a, p. 10). There is in other words a better mapping to the problem domain with things like machinery, bank issues, customers, products, sensors, markets and so on (Webster, 1995, p. 22). Castelluccio (1997) gives an unpretentious explanation as to why the object-oriented paradigm is more natural (quotation):

> Remember those intelligence tests in primary school? Those series of four or five pictures or objects where you had to pick out which one was different? Object-oriented programming depends on the human tendencies to look for similarities and to reduce complexity through classifying and "chunking". Similar objects are grouped into classes.

Bruegge & Dutoit (2000, p. 7) even propose that when object-oriented information systems development is performed the problem domain integrates into the solution domain. In other software development paradigms the problem domain and the solution domain are different, and other paradigms are often based on some mathematical models that users are neither familiar with nor interested in (Koskimies, 1997, p. 2).

Bozowski (1997), Koehler (1992), Korson & McGregor (1990) and Martin & Odell (1992, p. 31) argue that the parts that are produced by the object-oriented design process are more natural in the sense that they correspond more to the concepts of the real world. This claim is also supported by other researchers like Capper et al. (1994), Eriksson & Penker (1996, p. 43), Fagerström (1993, p. 15), Nerson (1992), Räisänen (1997a, p. 10), Sim & Wright (2002), Smith & McKeen (1996) and Taylor (1990, pp. 29-31) who propose that object-oriented software supports the way people view the real world, and the transition of this view into information systems becomes easier.

As an example of the arguments above, one can present the claim by Esch (1995) that there is a one-to-one mapping between the objects in the real world and the objects that are in a manufacturing information system. The correspondence between objects in the real world and objects in an object-oriented information system is easily understood if one examines an object-oriented user interface with objects as icons for different business objects like invoices, contracts, establishments and customers (Capper et al., 1994). However, the one-to-one correspondence between objects in the graphical user interface and the business objects does not come by itself. Though there is a relationship between them, there might be a different level of granularity (Capper et al., 1994).

There are also research results that imply that human beings usually think in a procedural way and not in an object-oriented way (Agarwal et al., 2000; Rosson & Alpert, 1990). Hatton (1998) presents an interesting model of how human beings use their memory; in the model the short-term memory, the long-term memory and a rehearsal buffer are considered. Out of the model and also psychological evidence from studies on Alzheimer's disease Hatton presents some material on how the memory of human beings works. The writer goes on and claims that the concept of encapsulation fits well in the memory of a human being. However, the concepts of inheritance and polymorphism do not.

In the study by Agarwal et al. (2000) there was also the result that people do not think in objects and objects are not natural representations of things in the problem domain. An earlier study by Rosson & Alpert (1990; cited by Sim & Wright, 2002) came to the same result as the study by Agarwal et al. (2000). The question of how well the object-oriented paradigm is connected with how human beings actually think is probably a difficult question and cannot easily be solved (Martin & Odell, 1995, p. 20). One fact that makes this issue difficult is that the mind of human beings is different from one human being to another, which signifies that people might think very differently in a similar situation (Barondes, 1998, p. 2-3).

In the empirical study by Johnson (2000) improved communication between information systems developers and end users was found to be a benefit. In addition, Davis & Morgan (1993), Gillach & Deyo (1993), Johnson (1997a) and Johnson et al. (1999) propose that using object-oriented information systems development makes it possible for the end users and information systems developers to speak the same language. Further, the communication between information systems developers and business executives becomes easier if the object-oriented paradigm is used (Cockburn, 1998, p. 2; Johnson et al., 1999). Martin & Odell (1992, p. 31) even claim that end users and business people think naturally in terms of objects, triggers and events. Gillach & Deyo (1993) also propose that better communication between information systems developers and end users results in information systems that represent the real world more adequately, and that the object-oriented information systems developers could deliver custom applications that are built specifically for the needs of end users. However, the benefit of better communication was not considered very important, probably because the users do not care how the information system is developed and because the new paradigm is rather radical compared with older paradigms (Johnson, 2000).

Noack & Schienmann (1999) say that the object-oriented paradigm makes closer communication with users easier. Agarwal et al. (2000) propose that the communication between information system developers and users is very important; otherwise the requirements analysis and the information system will not succeed. If the persons performing object-oriented analysis and design are skilled and knowledgeable then the benefit of better domain mapping is more palpable (Webster, 1995, p. 22).

It is also important to remember that objects used later on in the information system development process are often dynamic entities, which means that the natural view of them might have been lost (Bozowski, 1997). The natural view might also be difficult if

there are several names for one object that are natural. Another problem with the natural view is that there might be names among the users, but no explicit object for the names. These issues are discussed by Hanseth & Monteiro (1994).

Lauesen (1998) reports that users do not find object-oriented analysis natural; object-diagrams do not make sense to users, object-interaction diagrams are not easily understood, and objects per se are not natural for users. Objects like customers are perhaps relatively easy to understand, but objects like orders are hard to grasp as something that can perform operations and have responsibilities (Lauesen, 1998).

Analysis, summary and discussion. According to previous studies the concepts of 'natural' and 'better mapping to problem domain' are predominately due to the following issues:

- Good connection to the mind of the human being because it is easier to look at the world as a collection of objects than it is to look at it as a collection of functions and data (Booch, 1994, p. 287). The world is full of things as customers and products and these concepts are typically natural for a human being and can usually easily be found in the problem domain. This issue can be put in doubt, because different persons think in different ways, and something that is natural for one person might be rather unnatural for another. However, an object is a rather persuasive concept; typical examples with customers and products are often easy to understand.
- The parts that are produced by the object-oriented design process are more natural (Bozowski, 1997; Koehler, 1992, Korson & McGregor, 1990; Martin & Odell, 1992, p. 31). Lauesen (1998) was of the opposite opinion and reported that object-oriented diagrams do not make sense to end users. Probably objects per se might be easy to understand but object-oriented diagrams are not easy to comprehend. The question of how to model the "real-world" is also an interesting question. According to Isoda (2001) the real-world modelling can be divided into "genuine real-world" modelling and "pseudo real-world modelling" In other words, the object-oriented mapping to the problem domain is not very easy.

The question of what is natural is probably a very difficult one. End users are probably experts of the concepts in the domain where they are working. Information systems developers are perhaps not always very familiar with the same concepts. In a study reported by Johnson (2002) it was found that end users tend to think in terms of objects and experienced information systems developers tend to focus more on functional properties.

The following possible association between benefits has been identified:

- The OBJECT-ORIENTED PARADIGM with objects -> is more NATURAL for human beings.

3.3.8 Maintenance

Maintenance of information systems in organisations is a burdensome activity and more time is usually spent maintaining old information systems than in developing new ones (Swanson & Dans, 2000). Maintenance is the most costly part of an information systems life cycle and sources indicate that more than 50% of the working time is spent on maintenance out of the total time spent on software activities (Lientz & Swanson, 1981). According to Fagerström (1993, p. 13) and Hatton (1998) about 80% of all costs for software (from the whole life cycle of the software) come from maintenance, Erlikh (2000) proposes that the figure is about 90%.

Out of the maintenance costs one study showed that 65% were perfective (Lientz & Swanson, 1981). In another study it was found that 42% of the maintenance costs came from challenging new requirements specified by users (these changes are often challenging because one has to go back to the early stages in the life cycle of the software), 17% came from changes in the data, 12% came from emergency corrections and 9% came from normal daily corrections (Koskimies, 1997, p. 4).

If maintenance could be easier and cheaper due to an object-oriented information system, much could be gained. Easier maintenance is based on the ability to make changes easily and the in-depth understanding of how the information system or program is built (Wilde & Matthews, 1993).

Maintenance of the information system becomes easier (Agarwal et al., 2000; Booch, 1994, pp. 77-78; Caliò et al., 2000; Graham, 2001, p. 41; Johnson, 2000; Kozaczynski & Kuntzmann-Combelles 1993; Martin & Odell, 1992, p. 33; Nowicki & Kosiak, 1996; Radin, 1996; Sim & Wright, 2002) and cheaper (Gillach & Deyo, 1993; Page-Jones, 1992b) if object-oriented information systems development is used in comparison with functional information systems development. Koskimies (1997, p. 2) even proposes that easier maintenance is one of the most important benefits of the object-oriented paradigm.

Easier maintenance is based on several aspects, but one is the expectation that an object-oriented information system has high quality because of better data abstraction, better information hiding, better concurrency control and better management of changes in the real world, due to the mapping between objects in the real world and the objects in the application (Wilde & Matthews, 1993). Another aspect is localized maintenance, which means that changes in the object data or implementation is only modified in one single place, leading to lower costs and fewer errors (Graham, 2001, p. 41; Pressman, 2000, p. 526; Stevens & Pooley, 2000, p. 9; Watson et al., 2004). Because objects are encapsulated and viewed as black boxes, programs that use the objects only care about what the object is supposed to do, which in turn makes maintenance easier (Lam, 1997). According to Räisänen (1997a, p. 13) easier maintenance is a result of the real world model in object-oriented software development, better modularity and higher quality.

Hatton (1998) is, however, of a different opinion and argues that there is little data that support the claim that object-oriented information systems are easier to maintain than traditional functional information systems. Eeles & Sims (1998, p. 61) are of the same

opinion as Hatton (1998) and present an example of a customer class used in two different information systems (a car rental system and an apartment rental system); the customer class can be changed once but both information systems must be rebuilt, retested and redeployed.

Hatton (1998) and Wilde & Matthews (1993) propose that the complexity of object-oriented information systems is one reason why object-oriented information systems are more difficult to maintain than traditional functional information systems. However, if adequate design practices and passable support tools would be available, then maintenance of object-oriented information systems would also be easier (Wilde & Matthews, 1993),

Ambler (1998, pp. 134-135) proposes that patterns increase the consistency between applications, making the applications easier to understand and maintain. When applications are developed in a consistent manner it will become easier to do technical walkthroughs that make it easier to improve the quality of the applications (Ambler, 1998, pp. 134-135).

By developing information systems that are based very much on *components* the information systems are easier to maintain. Object-oriented information systems are more flexible and easier to modify (Fagerström, 1993, p. 9; Gillach & Deyo, 1993; Graham, 2001, p. 41; Smith & McKeen, 1996), they can easily be modified to keep up with rapidly changing business needs and business environments (Gillach & Deyo, 1993) and their maintainability is better than the maintainability of traditional functional information systems (Kozaczynski & Kuntzmann-Combelles, 1993).

Regarding maintenance, the object-oriented paradigm allows information systems to be extended over time when new functionality is needed; in other words, we can build smaller information systems that can easily be extended in the future (Rumbaugh, 1997; Taylor, 1992, p. 138). Adding extra components to object-oriented information systems is easier and safer than making extensions to traditional systems (Henderson-Sellers, 1992, p. 49). The object-oriented information system is more ‘open’ and new components like classes can be more easily added to the information system without damaging its overall structure (Eriksson, 1992, pp. 16-18; Jaime et al., 2000). Wolber (1997) furthermore proposes that objects are less likely to change than processes and requirements; therefore object-oriented information systems are more maintainable than traditional information systems.

The claim that the object-oriented paradigm makes maintenance easier rated 3.79 on a scale from 1 to 5 (1 =strongly disagree, 5 =strongly agree) in a study of managers who direct object-oriented information systems development projects (Gehringer & Manns, 1996). In the study by the Gothenburg School of Economics on the use of the object-oriented paradigm in Sweden, one finding was that 48% of the companies thought that maintenance costs decrease due to the object-oriented paradigm (Lotsson, 1996).

Reuse is the most important concept that makes maintenance easier (Lim, 1994; Wilde & Matthews, 1993). Easier maintenance is also based on the class structure that makes it easier to contain the effects of changes; the use of inheritance also makes it possible

to reuse existing classes and to extend the information system (Coleman et al., 1994, p. 7). The extension of the information system is usually based on adding new functionality to subclasses (Wilde & Matthews, 1993).

The independent and encapsulated objects also make maintenance easier because modifying one object will not affect other objects (Sommerville, 1992, p. 193; Wilde & Matthews, 1993). In a traditional approach global variables might be used which makes the modifications of procedures and functions more difficult to anticipate. Capper et al., (1994) also propose that the encapsulation of data and methods makes changes to the object-oriented information system easier, because it allows the scope of the changes to be found rapidly, and therefore there is a lower risk of accidentally introducing errors in other parts of the object-oriented information system.

However, if the problem in the object-oriented information system is due to improper messaging between objects then the maintainability will suffer because of the difficulty in analysing the message passing between several objects, and finding the troublesome object. (Capper et al., 1994) In addition, Wilde & Matthews (1993) talk about object-oriented information systems maintainers that have to trace through chains of dependencies produced either by inheritance or calling relationships. There are, however, tools that are based on dependence analysis that can be of great help for the object-oriented information systems maintainer working with these difficult issues. The browser can, for example, usually open a specific window for each link in the chain. If there are a lot of links there will also be a lot of windows, which probably make the analyses of the links and relationships harder. (Wilde & Matthews, 1993)

Lieberherr & Xiao (1993) propose that using the object-oriented paradigm makes maintenance actually *more difficult*, because details of the class structure are repeatedly encoded in the methods. The methods are in the classes and hard to change without consequences that are difficult to foresee. In addition, Wilde & Huitt (1992) argue that using object-oriented software development makes maintenance of the information system more difficult. This is because inheritance and polymorphism introduce difficulties in program analysis and understanding.

Analysis, summary and discussion. Easier maintenance is chiefly due to the following issues:

- The reuse mechanism, like inheritance (combined with the class structure), makes consistency easier to maintain (Winblad et al., 1990, pp. 213-222). In fact, reuse is probably the most important concept and issue that make maintenance easier (Lim, 1994; Wilde & Matthews, 1993). By inheritance and reuse of components, an information system developer can easily add new features to an information system that is maintained. In addition, modification can be achieved through inheritance, but here the information systems developer has to be more careful, there is a danger of unwanted effects, like the yo-yo effect, if an information system obtains complex inheritance structures.
- Software components that make information systems are more flexible to maintain (Gillach & Deyo, 1993; Smith & McKeen, 1996) and it is rather easy and safe to add new components (Henderson-Sellers, 1992, p. 49). Components

like tested objects are also not very likely to change (Wolber, 1997). Encapsulation is actually also an important concept that makes maintenance easier, because changes can be found rapidly and there is a lower risk of accidentally introducing errors in other parts of the object-oriented information system (Capper et al., 1994). If no adequate software components can be found, or the existing software components are of low quality the maintenance might, however, suffer.

- In-depth understanding of how an information system is built (Wilde & Matthews, 1993); this issue combined with reuse and components, also creates an ability to make changes easily (Wilde & Matthews, 1993). Regarding this matter one can hypothesize that it has to do with the opinion that object-oriented information systems are more 'natural' and better mapped to the problem domain, which makes the understanding of the information system easier. However, the object-oriented paradigm is complex and it might be difficult to get the 'in-depth understanding of how an information system is built'.
- High quality because of several reasons such as better data abstraction, better information hiding, better concurrency control and better management of changes in the real world (due to the mapping between objects in the real world and objects in the information system) (Wilde & Matthews, 1993). If an information system is of high quality it is certainly also easier to maintain. Whether the concepts mentioned above produce information systems of better quality is another question, as discussed in this study in the analysis of the 'quality' benefit.
- The object-oriented information systems are *smaller* (Rumbaugh, 1997; Taylor, 1992, p. 138). The maintenance of smaller information systems is not necessarily easier than the maintenance of larger information systems. Major issues regarding the maintainability of information systems are system design, program design and quality. The size is only one issue that might affect the maintainability of an information system.

Reuse is again the major concept that enhances maintenance; it is easier to maintain an information system or application if one can work with existing and tested components, and the information systems developer does not have to program all the changes at the commencement. The basic object-oriented concepts like classes, inheritance, dynamic binding, encapsulation and polymorphism are also argued to make maintenance easier, and if this is true then one can propose that the object-oriented paradigm as such is well suited for maintenance. The complexity of the object-oriented paradigm is, however, a hindrance for easy maintenance (Hatton, 1998). Because details of the class structure are repeatedly encoded in the methods the maintenance also becomes more difficult (Lieberherr & Xiao, 1993).

The conclusion is that reuse enhances maintenance of object-oriented information systems. However, the complexity of the object-oriented paradigm might affect the maintenance negatively.

The following possible associations between benefits have been identified:

- The utilisation of REUSE makes it easier to maintain the information systems because tested and existing components can be used -> MAINTENANCE becomes consequently easier.
- The use of SOFTWARE COMPONENTS makes it possible to avoid programming new parts for an information system, which makes -> MAINTENANCE easier and faster.

3.3.9 Software components

Pancake (1995) proposes that the greatest advantage of the object-oriented paradigm is the fact that objects can be used as *software components*. Components can also be analysis components, design components or programming components, etc. (Coad & Yourdon, 1991, p. 124).

According to Love (1993, p. 238) a software component (a class or object) usually consists of 10-15 methods, equivalent to 200-300 uncommented lines of programming code and it typically takes about two months for a person to build a commercial software component once the design is understood.

Software components are interesting because one vision of the object-oriented paradigm is that one could build information systems with software components in the same manner as one can, for example, build a radio out of premade and tested technical components. This is probably possible though some kind of programming code will probably be needed in order to integrate the premade software components.

The concepts of components and objects should not be considered the same. According to Petre (2000, p. 6) the difference is that objects are suitable for describing real world entities and components are suitable for describing the services of real world entities. Expressed differently, objects are suitable for describing the problem domain and components are suitable for describing the functionality of the problem domain.

Components and reuse of components have generated success in information system development in several companies. One good example of such a company is Castek in Toronto, Canada (Sparling, 2000). IT shops nowadays are building libraries of components and these components are often built for sale. Components are also derived from internal information systems development projects. (Carr, 1999)

Components have to be managed properly so that they can be reused, and some software companies have even created corporate support centres for software components to facilitate the internal sharing of the components. Each centre manages several activities like receiving the components, verifying the quality of the components, documenting the components, handling the maintenance of the components and shipping the components to places where they are needed. Every individual component normally has a formal corporate part number and the use of an individual component results in internal transfers (Love, 1993, p. 164).

Eriksson (1992, p. 54) argues that software components or modules are easier to develop because of the object-oriented paradigm. Kaasböll (1993) is of the same opinion and claims that the easier development of components is due to object-oriented software development *methods*. Caliò et al. (2000) also feel the same and present UML as such an object-oriented software development method.

According to Sparling (2000) it is also important that information systems developers accept the value of working with the components as encapsulated black boxes, and not try to rebuild them if it is not absolutely necessarily.

As maintained by Henderson-Sellers (1996, p. 16) and Thomas (1989) the time to code and test is usually less in object-oriented information system development, because the reused classes and software components have usually been more carefully designed and tested. Existing tested components can also safely be reused again and again (Airikkala, 1996), but there are some questions the information systems developer has to consider such as those in the following quotation from Binder (1999, p. 29):

> I'm a producer of reusable components. How can I test these components without knowing how they will be used?
>
> I'm a consumer of reusable components. How can I be sure that a reused component works correctly?
>
> I'm a consumer of reusable components. How can I be sure that a reused component hasn't caused other objects in my system to break?

Testing is important when developing components for reuse. The consequences of an error in a component are much more severe when the information system developer of the component in question is hard to locate and when the component is used in several places (Stevens & Pooley, 2000, p. 217).

Because of software components (modularity) object-oriented information systems are often more robust, more extensible, more flexible and have higher integrity. This is not only because of the components but also due to encapsulation that makes modifications safer and easier (Henderson-Sellers, 1992, p. 68; Henderson-Sellers & Edwards, 1994, p. 15; Petre, 2000, pp. 2-3).

A reusable component can be used in several different ways. It can, for example, be used as an attribute in other classes (aggregation or association), it can be used in inheritance, it can be inherited from to create subclasses (derived classes) and it can be used as an object and instantiated to create a set of runtime objects (Pant et al., 1996). However, in order to reuse a component the component must be locatable, consumable and extensible (Sparling, 2000). The components also have to be used correctly. One problem is that when the information system that uses components grows larger there will often be several different versions of one component (Jarzabek & Knauber, 1999).

Silveira (2000) presents one interesting advantage of using components when doing maintenance. He presents a "Web-based object computing paradigm" for supporting on-demand, dynamic distribution and integration of distributed reusable software artefacts on user environments during execution time. This paradigm is based on the concept that information systems do locate, retrieve, install and execute remotely available software

components on user desktops. This is done over the Web in the same manner as, for example, virus detection programs locate new components and install them on the user's computer. Another example is Java applets, which load remotely available classes during runtime. Examples of applications that are based on this new paradigm are Castanet, Netcaster, NetDeploy and WebCasing. The software that is based on this new paradigm is called 'spontaneous software'. (Silveira, 2000)

Analysis, summary and discussion. Software components as a ´concept' is a benefit from object-oriented software development and this benefit is due to the following issues:

- The object-oriented paradigm. The object-oriented paradigm per se is supporting the creation of software components. According to Eriksson (1992, p. 54) a proposition in the object-oriented paradigm is the use of software components. Because software components in the form of classes, objects and design parts, etc. are predominant in the object-oriented paradigm, the argument by Eriksson seems to be correct.
- Object-oriented software development methods. Object-oriented software development methods support the creation of software components (Kaasböll, 1993).

Software components are very central in the object-oriented world and one important base for reuse. Object-oriented information systems are also more robust, more extensive, more flexible and have higher integrity due to the software components (Henderson-Sellers, 1992, p. 68; Henderson-Sellers & Edwards, 1994, p. 15; Johnson et al., 1999; Petre, 2000, pp. 2-3).

The following possible associations between benefits have been identified:

- The OBJECT-ORIENTED PARADIGM enables the use of -> SOFTWARE COMPONENTS.
- Using SOFTWARE COMPONENTS results in higher -> FLEXIBILITY because one can reuse premade artefacts.
- Using tested SOFTWARE COMPONENTS results in higher - > ROBUSTNESS.
- Making use of SOFTWARE COMPONENTS leads to -> easier EXTENSIBILITY possibilities.
- Using SOFTWARE COMPONENTS results in higher -> INTEGRITY because the components are encapsulated without things like global variables.

3.3.10 Easier End-User Computing

During the 1980s the development of information systems by end users accelerated especially in the scientific/technical and business/commercial field (Brancheau & Brown, 1993), and it is estimated that by 2005 in the US alone, there will be 55 million end user developers (Sutcliffe & Mehandjiev, 2004).

Buxton (1993), Love (1993, p. 254), Pressman (2000, p. 891) and Winblad et al. (1990, p. 49) point out that the end users of today can probably develop and build information systems of their own easier in the future by using the object-oriented paradigm. Business people in some cases will be able to make changes in object-oriented information systems by themselves and will not need to consult programmers (Martin et al., 2001; Verity & Schwartz, 1991). However, Buxton (1993) argues that the rules of behaviour for objects will still have to be expressed in algorithmic terms and therefore object-oriented information systems will still need systems analysts and information system developers.

Brancheau & Brown (1993) give the following (quotation) definition of End-User computing:

> End-User Computing is the adoption and use of information technology by personnel outside the information systems department to DEVELOP software applications in support of organizational tasks.

Further Welke (1994) and Patriot Partners (presented in Martin & Odell, 1992, p. 50) claim that the age when information systems are developed by software developers is coming to an end. The new approach is where ’ordinary’ people select and acquire product components and assemble them into information systems (Mörch et al., 2004). The role of the information system developers will be to develop components for the needs of the users (Love, 1993, p. 254). This is the End-User computing concept. The term information system development will change and the change will be based on the object concept. Objects will be available for all kinds of information systems and combining objects from object-oriented platforms that are integrated in operating systems, will make it possible for users to build information systems on their own. The end users will combine the objects by using some sort of ‘glue’ (Alencar et al., 1998). There are research reports on component-based software engineering and some software engineering books like the one by Pressman (2000, pp. 738-763) that present what ‘glue’ could consist of.

Martin & Odell (1992, p. 33) talk about easier programming based on the object-oriented paradigm; this issue supports end-user development. Object-oriented programming is easier because programs can be developed in small pieces. However, Welke (1994) puts into question the view that users would start programming.

In a scenario by Gibbs et al. (1990) there are similar ideas to the ones above; a developer builds an information system by selecting generic software components and then composing these components. Eriksson & Penker (1996, p. 166) present a scenario where information systems will be developed from components, and the components will come from standard applications (like Word, Excel and Lotus 1-2-3), from libraries

with components developed in-house, from components in standard libraries, from components in the operating system and from components that are visual interface components (for charts and diagrams), etc. Eriksson & Penker (1996, p. 166), however, do not propose that end users will actually develop information systems.

However, according to Pree (1997) end-user computing in the future will be more based on component-based software engineering than on object-oriented software engineering and frameworks will be used as the building blocks in end user programming. Further Pree (1997) argues that visual, interactive composition tools will be available that make it possible for end users to develop information systems by handling components that are connected to convenient frameworks.

Further, according to Welke (1994) the *objects being manipulated will be business artefacts*, and the role of the information system developer in the future will mostly be to guide the end users on how to find objects. Information system development will be increasingly directed at the production of commercially available object components for general and more special information systems. (Welke, 1994) One benefit of the new idea that end users will develop their own applications and information systems is of course that information system developers will not be needed anymore; there will be no actual need anymore for end users to try to explain to information system developers what they need (Love, 1993, p. 255).

Gillach & Deyo (1993) also propose that end users will become more involved in information system development because of the object-oriented paradigm. The involvement will range from defining business processes to designing and developing solutions and further onto testing and refining so that the information system is according to the requirements (Gillach & Deyo, 1993).

Nevertheless, in 1998 end-user information systems development with objects was still very rare according to Finch (1998), but Staringer (1994) reports on a major information system that was built in co-operation with users in an end-user manner. The end users obtained some adequate tools and started to build the information system; this went well though the information system developers had some problems with the end users who could alter the source code of the information system developers, which sometimes caused the information system to behave differently than intended (Staringer, 1994). However, Lauesen (1998) claims that the object-oriented paradigm has not made it easier for end users to develop information systems.

Analysis, summary and discussion. Easier End-user computing is a benefit predominantly because of the following:

- Object-orientation. Easier programming (and implementation) because of object-orientation. This question is a very difficult and dubious question and is discussed below.

If end-user computing becomes more widespread, there will be less need for system analysts and information system developers according to Love (1993, p. 254). Information system development work will also change and probably be cheaper; the

savings in the decreased use of information system developers will, however, be balanced by the extra costs of having end users spend time developing information systems.

Winblad et al. (1990, p. 49) consider how the user can start programming if there is access to the object-oriented paradigm and some object-oriented class library. The scenario by Welke (1994) supports the claims by Winblad et al. but there are still many questions that have to be answered. How can the user find the objects? Can the user adequately utilise the objects without any understanding of how the objects have been developed? Is it really workable information system development to have users using objects or standard application packages and then developing the 'glue' between the objects? Perhaps the development of the 'glue' is difficult. Nevertheless, Zhang (1999, pp. 167-168) proposes that end users would be both willing and capable to carry out some information system development work, on the condition that the information system development methods the end users work with are simple and supported with easy-to-use tools. Help from experienced information system developers would also be necessary.

The end-user computing concept can further be criticised if the users, for example, use the C++ programming language one has to remember that C++ is a complex programming language (Koskimies, 1995) that is definitely not very well suited for beginners. In addition, Taylor (1992, p. 275) is of the opinion that end users should not program their own information systems although end-user computing has been an activity in several companies since the 1970's (Brancheau & Brown, 1993).

The following possible association between benefits has been identified:

- The OBJECT-ORIENTED PARADIGM with readymade components makes it easier to develop information systems, which result in -> better possibilities for END-USER COMPUTING.

3.3.11 Reuse

Reuse means the process of using existing software modules and other items instead of building everything from scratch (Basili et al., 1996a; Watanabe, 1997). Reuse has been a part of software development and programming since the early days of programming, but though traditional functional approaches allow for reuse of code, the object-oriented paradigm provides mechanisms that facilitate and put in force reuse (Fichman & Kemerer, 1993). Examples of these mechanisms are abstraction, encapsulation and inheritance.

Frakes & Isoda (1994) define reuse as follows (quotation):

> Software reuse is the use of engineering knowledge or artefacts from existing systems to build new ones. Software reuse is a technology for improving software quality and productivity.

Reuse is important in the object-oriented paradigm and makes it possible to move from a project oriented way of developing information systems to a product oriented way, where software modules are developed for several projects and not only for the ongoing project (Eriksson, 1992, pp. 348-349). In fact, the reuse concept is probably the most outstanding benefit of the object-oriented paradigm (Agarwal et al., 2000; Gillach & Deyo, 1993; Henderson-Sellers, 1992, p. 51; Koskimies, 1997, p. 2; Yourdon & Argila, 1996, p. 6).

One can develop object-oriented information systems without reuse (Koskimies, 1997, p. 5). However, developing object-oriented information systems without reuse might be appalling (McClure, 1996). Further Webster (1995, p. 215) proposes that one has to be aware of the fact that the benefits of reuse are not always realized.

Reuse can efficiently be combined with encapsulation, information hiding and inheritance (Wolber, 1997). Martin & Odell (1992, p. 51) claim that one of the best ways of achieving reuse is to use the object-oriented concepts that are connected to reuse mechanisms; classes, inheritance, polymorphism and frameworks. Gamma et al. (1995, p. 28) propose that using frameworks is the way that object-oriented systems achieve the most reuse and larger object-oriented information systems consist of layers of frameworks that cooperate with each other. Furthermore Koehler (1992) proposes that because of inheritance and reuse there is less code to write and test.

Reuse of software components can be accomplished through several concepts and mechanisms like class libraries, inheritance, design patterns and frameworks, etc. (Watanabe, 1997). As an example of a set of class libraries one can consider the Java class libraries that are produced by JavaSoft and Microsoft (Franz, 1998). The class libraries of Java are actually crowded with thousands of useful classes like classes for networking and encryption (Watson, 1999). Note that when using class libraries that they are usually written for a specific programming language like Java. A class library is not the only place for storing classes; another place is, for example, a repository (Jenz, 1999a). A repository is a tool for storing and retrieving development work. Source code in class libraries, documentation and analysis/design models can be stored by making a repository a more general storage mechanism than a class library. In practice a repository is a centralised database (Ambler, 1998, p. 217).

Reuse is reliant on artefacts to reuse. Classes and other components are usually reused, but also design and other object-oriented artefacts like business objects, subsystems and subroutine libraries etc. can be reused (Henderson-Sellers, 1992, p. 51; Radin, 1996). Joos (1994) presents designs and documentation as very reusable based on her experience at the company Motorola in the United States. Räisänen (1997b, p. 33) presents business plans, cost analyses, user manuals, project plans, test cases, requirements, designs and applications as reusable. Frakes & Terry (1996) present architectures, estimates (templates) and human interfaces as reusable. Mili et al. (1995) present data and programs as reusable. According to Gibbs et al. (1990) past experience like requirements, specifications, models, designs and software components and evolving software should be reused in order to improve the productivity of information systems development.

According to Nierstrasz et al. (1992) the reuse paradigm can very well be used, for example, for composing applications from already *packed software components*. Note however, that when building applications or information systems out of existing components the components can seldom be reused as they are, reusable components generally need to be adapted to match the system requirements. Component adaptation techniques should be transparent, black box, suitable for composition, configurable, reusable and efficient to use. (Bosch, 1997)

Reusable classes come from three sources, some come with the object-oriented programming languages, while others can be obtained from companies or developed by in house software analysts and programmers (Taylor, 1990, p. 90). Of course it might also be possible to find reusable components on the Internet that are originally from other sources like universities or even the personal libraries of programmers (Watanabe, 1997).

Reuse is dependent on the following (Hopkins, 2000):

- Components of good *quality.*
- Suitable components can be *found.*
- Components are *licensed.*
- Components that can be *used* without problems.
- The availability of a *platform* on which the components can be used and on which the components can send messages to each other.

However, the above listed requirements are seldom found, Grinzo (1998) reports of many programmers who have experienced that components are difficult to reuse, components do not include the source code (which means that they cannot be modified) and components are bound by absurd licensing restrictions, etc.

Reuse is based on software that is more general, which means that the software might be more cumbersome for users and more costly in terms of the number of CPU instructions (Deubler & Koestler, 1994). When developing general software components for reuse in the future, one has to consider what information systems will be built in the future and also to evaluate the straightforwardness needed for the development of new information systems, etc. (Nierstrasz et al., 1992).

Note that the word ‘composed’ is interesting, components using interfaces from each other are usually said to be composed together (Petre, 2000, p. 2). If the component will be unused then it is of course no idea to develop it as a reusable component, there has to be something that validates the effort (Sparling, 2000). One has also to consider the costs of reuse, which includes costs for creating or purchasing, reuse work, tools, product, libraries and implementing reuse related processes (Lim, 1994).

In addition, documentation is important when developing reusable classes; it might be more cumbersome and more expensive than the documentation of ordinary classes (McClure, 1996; Stevens & Pooley, 2000, p. 9). Webster (1995, p. 161) suggests that an on-line document should be created for every class, justifying its creation and design

and elucidating the postulations behind it as well as the future plans for its use and extension. The software developers should of course also update the documentation of a class when the class is modified (Webster, 1995, p. 161).

There are several *management issues* regarding reuse. Lawrence & Pfleeger (1995) found that doing reuse presupposes proper planning and measurement. Glass (1998) even proposes that if management is not working well then the whole reuse process might fail, which of course is in correspondence with the arguments that are presented by Meyer (1997b). Jenz (1999a) argues that reuse is so important that one should have a reuse manager who works with reuse issues.

Other management issues are planning and management of human issues (Bhattacherjee & Gerlach, 1998; McClure, 1996). One has to agree on the level of reuse, to promote reuse and to develop standards for building reusable components, etc. Furthermore Mili et al. (1999) propose that one also has to consider the costs of reuse, and these costs have to be weighed against the benefits. The costs of reuse are associated with the costs of finding and understanding classes in libraries and the costs of making modifications to existing classes (Henderson-Sellers, 1996, p. 20). If one is developing reusable classes, one has to be aware of the reality that reusable classes are usually more time consuming to develop and therefore also more expensive to develop (McClure, 1996). Software reuse is concerned with the trade-offs involved in such cost-benefit decisions; if the reuse costs a lot and the benefits are not obvious one should of course not go ahead with it (Mili et al., 1999).

Gehringer & Manns (1996) studied software reuse through consulting managers who directed object-oriented programming projects. The finding was as follows (quotation):

> Answer to the next question ‘yes’ or ‘no’.
>
> Does your company have an organized program to encourage software reuse?
>
> 34 Yes 19 No.
>
> If you answered yes to the question, is the reuse:
>
> a) Class libraries purchased from vendors? 25 Yes 8 No
> b) Class libraries developed in-house? 30 Yes 1 No

Reusing is also connected to education on how the components work, to testing, which means that the programmer gets to know how the component really works, and to working with implementation details outside the component, which means that the component has to fit the place were it is going to be used (Grinzo, 1998). Often software developers have to do more drastic things like reprogram or modify existing reusable classes, which is an activity that has to be carried out with great care (Casais, 1995, p. 201).

Finally, one can present some good arguments for reuse. According to Meyer (1995, p. 106) reuse enhances productivity, facilitates maintenance, improves reliability, efficiency and interoperability and capitalizes on software investment, which is a result of reuse as a customer and of reuse as a producer.

Analysis, summary and discussion. Reuse is a significant benefit and is predominately due to the following issue:

- The object-oriented paradigm. Core object-oriented concepts like classes, inheritance, polymorphism and frameworks support reuse (Cockburn, 1998, p. 25).

Reuse might constitute one solution to the problem with expensive and time-consuming software development. Instead of building everything from scratch one tries to reuse as much as possible. If good reusable components are used and developed, the information systems development process will not only gain from this, in fact the information systems developer that has developed the components that are often reused, will probably be famous (Watanabe, 1997). Nevertheless, several issues have to be considered when the reuse concept is discussed and analysed. For example, the issues of hierarchical systems and reuse, finding the appropriate components, the quality of the components, the reusability of the components, copyright and management of reuse, etc.

In order to utilise reuse dynamically the software developers have to be able to *find* safe and *high-quality* components that are easy to understand and modify as McClure (1996) proposes. Out of personal experience as a C programmer at the major Finnish software company Tietotehdas Oy (nowadays Tietoenator Oy), building a large money market information system for the Union Bank of Finland (nowadays Nordea) in the year 1990, the author of this study experienced, together with colleagues, an occasional difficulty in finding suitable components for the information systems development work. Moreover, it was often rather hard to understand how to use standard C components. Several of the information systems developers in the money market project felt somewhat like detectives searching for appropriate components in large C libraries.

When the reuse concept in the object-oriented model is discussed, one important issue is the *hierarchical* information system and the management of complexity. According to Booch (1994, pp. 59-65) the reuse concept can well be used when an information system is hierarchical (Booch, 1994, pp. 59-65). If an information system is not hierarchical, or if the information system is small, then the object-oriented paradigm probably does not give any noteworthy advantages because the reuse concept cannot probably be used to its full potential.

However, there are researchers who are of a different opinion. Gillach & Deyo (1993) claim that the object-oriented paradigm can be used very well for developing almost all kinds of information systems, and present a case where developing products and information systems are based on other products and information systems in a common family. In other words, information systems and software product families that share common functionality are particularly good targets for object-oriented information system development because there is a good potential for reuse.

Cartwright & Shepperd (2000) present a case with an object-oriented information-system development project. The object-oriented information system had 133,000 lines of C++ code but there was little use of reuse. In fact, there were only two inheritance trees in the system, and the trees consisted of very few classes. Perhaps the lack of

inheritance and reuse was due to the problem domain or to the object-oriented analysis and design method used as Cartwright & Shepperd (2000) state. On the other hand, it was perhaps due to something else like the development experience of the information systems developers who were accustomed to thinking in certain ways. They might have thought that inheritance made the information system harder to understand and therefore more difficult to maintain (Cartwright & Shepperd, 2000).

Verschoor & Low (1994) studied the perceived benefits of reuse in Australian organisations and their finding was that organisations generally perceived substantial benefits from reuse. Nevertheless, Glass (1998) presents several reasons why reuse has not always been a success because there was "little to reuse".

The problem with "little to reuse" seems to be connected to the fact that different information systems need different components, and it is hard to find suitable components for a specialised information system. If one wants a component that shows the time on the computer then finding the appropriate component is probably not very difficult. However, if someone wants to find a component that calculates the interest rate in a very country specific money market information system, it might be very hard to find such a component to reuse. Finding a suitable component that at that time needs a lot of rewriting is often not the solution, because as most experts agree, it is more effective to start from scratch if more than 20% of the component must be reworked for its new use (Glass, 1998). In order to find out if a component is easy or hard to reuse one can carry out reusability assessment as presented by Frakes & Terry (1996).

Another problem as to why there are so few components to reuse is that it is more difficult and takes more time to build *reusable components* compared with developing components for a specific information system. If the information systems developer has a picture of what there is in the company to reuse, the developer would probably prefer not to build another reusable component if something already exist that can be reused, even if it is not a very good component (Glass, 1998).

As a summary, one can propose that the reuse concept is probably the most promising concept in the object-oriented paradigm. If the reuse concept can be used in all its potency through significant productivity, quality, efficiency and reduced cost, etc. results can be obtained.

The following possible associations between benefits have been identified:

- The OBJECT-ORIENTED paradigm is connected to the -> REUSE concept.
- By utilising REUSE -> the MAINTENANCE of the information system becomes easier because most maintenance tasks come from challenging new requirements specified by users, and these requirements can be more easily developed if ready-made components can be reused.
- By performing REUSE -> MANAGEMENT OF COMPLEXITY can be controlled more easily because the complexity of an information system is often due to a hierarchy that can be built by using reuse.

- By utilising REUSE -> HIGHER QUALITY can be achieved because the information system is built out of readymade and tested components.
- REUSE results in -> FASTER DEVELOPMENT because readymade artefacts like components can be used, which results in higher -> PRODUCTIVITY -> and better EFFICIENCY -> which leads to REDUCED COSTS.

3.3.12 Portability

Portability of an information system means the ease with which the information system can be adapted to work on different computers; in other words on other computers than the computer that the information system was originally developed for. The portability depends on several factors such as the programming language, the extent of exploitation of specialized system functions and hardware properties. (Pomberger & Blaschek, 1996, p. 13) Portability can be considered a part of quality (Graham, 2001, p. 45), but in this study portability is presented in a sub section of its own.

Theoretically, portability can be considered a benefit of the object-oriented paradigm (Agarwal et al., 2000), although platform independence is not actually related to the object-oriented paradigm. The idea is that an object-oriented program can run on every computer with the assistance of a virtual machine, like the Java Virtual Machine (Franz, 1998). In order to achieve this goal one has to use design independence. Classes are then developed to be independent of platforms, hardware and software environments (Martin & Odell, 1992, p. 34). The independent classes should also employ requests and responses of standard formats so that they can be used with multiple operating systems, database managers, network managers and graphic user interfaces, etc. (Martin & Odell, 1992, p. 34).

Interesting is the platform independence that the programming language Java has (Tyma, 1998). Java is portable because of it's compiler targets the Java bytecode and not any part of the operating system or hardware. Java works in any environment, which means that all platforms will support Java programs and pure Java applications (Martin et al., 2001).

There are also of course other portability schemes like the Juice solution developed by University of California Irvine (Franz, 1998). One can write programs, compile them and run them on every computer without porting the program (Tyma, 1998). There are also interesting solutions in this area, for example, the Juice solution is transparent to end users and applets based on Juice can coexist with applets based on Java (Franz, 1998). However, all computers do not have a virtual machine, which makes the benefit less useful (Tyma, 1998). Another problem is that virtual machines put too much responsibility on the behaviour of the application, or information system, on the virtual machine (Watson, 1999). The Java Virtual Machine is also not the best solution according to Franz (1998) who proposes that there are better solutions for cross-platform portability.

Analysis, summary and discussion. In this study portability is seen as a benefit of object-oriented software development although this can be questioned, and this is due to the following issue:

- The object-oriented paradigm itself. Portability is considered a benefit of the object-oriented paradigm (Agarwal et al., 2000). Although portability in this study is considered a benefit of the entire object-oriented paradigm, it is often associated with the programming language Java and information systems developed with Java (the Juice solution developed by University of California Irvine is another example). Portability solutions have also been developed of course with the functional paradigm.

The following possible associations between benefits have been identified:

- The OBJECT-ORIENTED paradigm -> has a good support for PORTABILITY in the Java programming language (and some other programming languages)

3.3.13 Discussion of the benefits in general

There are several benefits of the object-oriented paradigm. Some of the benefits are comprehensive and some are more detailed. One general objective of the object-oriented paradigm is to support reuse and make it possible to develop information systems out of existing, high-quality software components. The reuse possibility then gives birth to better management of complexity, better productivity, faster development, reduced costs, better quality, better usability, better maintenance and easier End-User computing.

The second significant benefit of the object-oriented paradigm is the naturalness and better mapping to the problem domain. The end users, clients and business executives, etc. are supposed to experience the world (and problem domain) as a collection of objects, which of course fits well with the object-oriented paradigm.

The third benefit of the object-oriented paradigm is better integration between analysis and design using the object-oriented paradigm as compared with older paradigms. This "one model benefit" can be experienced when working with most object-oriented software development models. In object-oriented information systems development object-oriented analysis and object-oriented design are important activities and considered more powerful than traditional analysis and traditional design. Therefore they are presented as benefits in this study, although neither object-oriented analysis nor object-oriented design can be considered as a pure benefit of the object-oriented paradigm.

The fourth benefit presented is portability, despite the fact that this benefit is very much connected to the programming language Java (the Juice solution developed by University of California Irvine was presented as another example).

When starting to scrutinize the benefits of the object-oriented paradigm, the benefits might sometimes be difficult to recognise. For example, Coad & Yourdon (1991, p. 17)

present several benefits of object-oriented *design*. However, many of the benefits are rather common, and it might be difficult to understand them. A benefit like 'improve problem domain expert, analyst, designer and programmer interaction' is quite vague because different persons can apprehend these issues in various ways.

The findings of Gehringer & Manns (1996), however, support the claims of the benefits of the object-oriented paradigm. It is probably like Webster (1995, p. 23) proposes: "object-oriented development offers significant benefits in many problem domains, but those benefits must be considered realistically, as must the costs of object-oriented development".

3.4 Problems with the object-oriented paradigm

Taylor (1990, pp. 109-113) discusses the following potential concerns with the object-oriented paradigm: the maturity of the technology, the need for standards, the need for better tools, the speed of execution (discussed comprehensively by Pomberger & Blaschek, 1996, pp. 284-286), the availability of qualified personnel and the costs of conversion and support for large-scale modularity. Pancake (1995) studied the question of what problems there are with the object-oriented paradigm in information systems development, and why in 1995 there were still many companies that did not use the object-oriented paradigm. The author identified several obstacles that have to be overcome before the object-oriented paradigm becomes a standard for industry applications in the future. In addition, Johnson (2000) and Steinmann (1992) found several problems with the object-oriented paradigm. In 1992 Steinmann presented 17 pitfalls and recommends that one has to be very careful when moving into the object-oriented paradigm. However, in the rather recent study by Johnson (2000) the information systems developers that participated in the study viewed the presented problems as virtually nonexistent.

In this section only problems that are connected with the object-oriented paradigm are presented. Specific problems connected only to analysis, design, programming or execution are not dealt with.

The problems are presented and analysed one after the other and possible connections to other problems are identified in order to develop a basis for further research.

In this analysis different problems with the object-oriented paradigm are studied with a focus on the issues that are behind these problems. The problems are further divided into those that could be solved if the market would be 'mature' and those that are intrinsic to the object-oriented paradigm itself. According to Krajnc (1997; cited by Helton, 1998) the major setbacks of object-oriented information systems development are actually due to the problems that are intrinsic to the object-oriented paradigm. As an example of such an intrinsic problem one can present the risk of creating spaghetti like code because all objects may reference each other. These reference problems might give some information systems developers reasons for scepticism about the possibilities of developing large information systems.

3.4.1 Complexity

In the *Survey of Advanced Technology 1996* (Pickering, 1996, p. 6-2), it was found that the most significant difficulties with the object-oriented paradigm are complexity and compatibility with existing practices. Johnson (2000) proposes that complexity of object-oriented analysis and design methods and furthermore complexities of object-oriented programming languages are considered significant disadvantages with the object-oriented paradigm. Noack & Schienmann (1999) and Lauesen (1998) mention complexity and especially the complexity of object-oriented programming code as a disadvantage of the object-oriented paradigm. In another study by Harrison et al. (1996) object-oriented code was found more complex and more difficult to understand than functional code. Maring (1996) is of the same opinion as Harrison et al. (1996) and reports that classes are often so complex that only the programmers who have developed the classes are capable of debugging, enhancing and maintaining the software they wrote, and this is with difficulties.

Gamma et al. (1995, p. 1) also propose that object-oriented information system development as an activity from requirements analysis to maintenance might be difficult. The difficulty is primarily connected to object-oriented analysis and design (Johnson et al., 1999), predominantly because object-oriented analysis and design requires a new and different way of thinking. The information system developer has to find pertinent objects, put them into classes at the correct granularity, define inheritance hierarchies, define class interfaces and finally establish key relationships between objects. When doing all this, the information systems developer has to remember that the object-oriented software should be reusable. (Gamma et al., 1995, p. 1)

As an example of the complexity of an object-oriented application Webster (1995, p. 204) presents the following: let us assume that there is an average of eight methods per object class in the object-oriented application. For 10 objects the upper limit of message links is now 720 and the upper limit for 100 objects comes close to 80,000 distinct message links. This is not the end of the structure, one also has to deal with the possible ranges of parameter values as passed among objects for method calls, and even worse, one has to examine the state of the receiving objects when the method calls are made (Webster, 1995, p. 204).

Hu (2005) proposes that university teachers have thought that the object-oriented paradigm is complex for a rather long time. Eriksson (1992, p. 442-443) proposes that the object-oriented paradigm is often considered complex by information system developers, and that there is a lack of good practices and standards. If good tools were available for understanding the often more dynamic behaviour of object-oriented information systems, then its complexity would probably not be considered a major problem by information systems developers (Love, 1993, p. 189). One reason why the object-oriented paradigm is considered complicated by information system developers is probably that research and development of the theory of the object-oriented paradigm have been performed in the academic world, and this often creates a conflict because many information system developers are often fairly practical and the solutions (like multiple inheritance) that the systems developers have to use are rather theoretical (Eriksson, 1992, pp. 442-443).

It is also difficult to make good object-oriented design decisions and to reuse existing components because it is hard to document components and to understand existing software components (LaBoda & Ross, 1997). There are also many possibilities to develop complex software when using the object-oriented paradigm; one can easily create new classes, rearrange hierarchies, add data and function members to objects, construct new objects from old ones (by inheritance) and have objects to send messages to each other, etc. (Webster, 1995, p. 116).

However, Pittman (1993) claims that the object-oriented paradigm is not complex per se, the complexity usually arises out of the way the information systems developers manage complex structures. According to Sheetz & Tegarden (1996) complex problem domains and complex implementation environments imply more complex object-oriented information systems.

Pittman (1993) even proposes that the requirements for training and experience are larger for an object-oriented developer than for a conventional developer. Information systems developers are, however, very different regarding skills and productivity; there are studies that show productivity differences of 25:1 among information systems developers with comparable training and experience (Love, 1993, p. 220). Therefore the level of experience and training of staff are significant success factors in object-oriented information systems development (Pittman, 1993). Further Sheetz & Tegarden (1996) propose that developing a *reusable* object-oriented information system is usually more difficult than developing a "one-shot" information system. Polymorphism can also increase the complexity of the object-oriented information system (Sheetz & Tegarden, 1996). If the underlying semantics of the methods in polymorphism with the same name are different, this will probably make the object-oriented information system more complex (Sheetz & Tegarden, 1996).

Complexity in the distributed object-oriented information system also makes it difficult to know where to put the functionality among the objects in the hierarchy (Sheetz & Tegarden, 1996). The labelling of attributes and methods and the determination of the class protocol can also make the already defined classes even more complex (Sheetz & Tegarden, 1996). Moreover overuse of inheritance will make the object-oriented information system more complex, and developing reusable classes is of course more difficult than developing "one-shot" classes, which makes development work more complex (Sheetz & Tegarden, 1996).

Sheetz and Tegarden (1996) found that because the object-oriented paradigm was considered complex there were many other difficulties in object-oriented information systems development. Among these difficulties were communication between objects, designing methods, using methodologies and tools, using the existing class hierarchy, designing classes, incorporating reusability constraints and project management. Lieberherr & Xiao (1993) also consider object-oriented programs rigid and hard to evolve because object-oriented programs contain a lot of redundant information about class relationships.

Lauesen (1998) claims that some kinds of information systems cannot even be developed with the object-oriented paradigm. Especially business information systems

are difficult to develop with the object-oriented paradigm because the complex structure of the business domain forces the developers to use objects that are not purely object-oriented. These objects are called degenerate objects. (Lauesen, 1998)

Analysis, summary and discussion. Complexity is a problem with the object-oriented paradigm and this is mainly due to the following issues:

- Theoretical concentration. The object-oriented paradigm has been developed in the academic world (Eriksson, 1992, pp. 442-443). Information systems developers that are often more practical, however, do not always appreciate the academic focus. Probably there are also many information systems developers that have an academic education but not in the object-oriented paradigm. The object-oriented paradigm is then considered complex because it is based on different way of thinking than the functional paradigm. Instead of mainly working with algorithms and functions one has to work with objects, classes, inheritance and existing components.
- Difficulties in understanding existing software components and other software artefacts. It might be difficult to reuse existing components, especially if the components are deficiently documented (LaBoda & Ross, 1997). Because different information systems developers have different ways of programming (programming is considered as a specific *art* by many researchers and authors) it might be difficult to understand how an existing software component works. This is also the daily challenge for information systems developers working with maintenance of information systems.
- Possibilities in the object-oriented paradigm to create complex structures and object-type spaghetti code (Krajnc, 1997; cited by Helton, 1998; Webster, 1995, p. 116). The complex structures and object-type spaghetti code comes from objects referring to each other in an uncontrolled manner, rearranged class hierarchies and new (added) data and function members to objects, etc.
- Polymorphism. If the underlying semantics of the methods in polymorphism with the same name are different, this probably makes the object-oriented information system more complex (Sheetz & Tegarden, 1996). Polymorphism has been considered very complex by several authors (like Penker (1994)) and developing mediocre solutions based on polymorphism makes things even worse.
- Distributed systems. Distributed object-oriented information systems are usually complex and it might be difficult to know where to put new functionality among the objects in the hierarchy (Sheetz & Tegarden, 1996). The distribution concept usually makes information systems more complex, and distributed object-oriented information systems are often very complex.
- Overuse of inheritance. One should use the inheritance mechanism with considerable care; otherwise complex inheritance structures are easily developed (Sheetz & Tegarden, 1996). One can perhaps actually propose that inheritance hierarchies are usually complex per se.

- Lack of good tools for managing the dynamic behaviour of object-oriented information systems (Love, 1993, p. 189). If there would be adequate tools for managing complexity the problem would not exist. There is always a need for good information systems development tools. If there are no suitable information systems development tools (like CASE tools) then the information systems development work inevitably becomes more complicated.

Sheetz & Tegarden (1996) introduced several other risks of developing complex object-oriented information systems. These risks were inadequate communication between objects, deficient designing of methods, second-rate use of methodologies and tools, using the existing class hierarchy in an uncontrolled way, designing classes badly, incorporating reusability constraints in a unsatisfactory way and imperfect project management. One can conclude that there are many different possibilities for using the object-oriented concepts in an inferior manner, probably because the object-oriented concepts are rather difficult to understand, and one must be trained and experienced in order to perform high-quality object-oriented information systems development. As an example of a difficult object-oriented concept one can mention polymorphism, which often is considered as complicated and incomprehensive (Penker, 1994, p. 20).

As a summary, one can propose that the complexity problem seems to come from the object-oriented paradigm per se, although one has of course to remember that complexity is a rather subjective issue, different people with different backgrounds perceive different things as complex.

The following possible associations between problems have been identified:

- The OBJECT-ORIENTED paradigm is based on rather complex concepts like polymorphism and inheritance hierarchies. This results in higher -> COMPLEXITY in the information systems development work.

3.4.2 The object-oriented paradigm is still immature

The object-oriented paradigm is still considered immature by some researchers, and often object-oriented projects are criticised as promising too much and delivering too little (Bhattacherjee & Gerlach, 1998). Webster (1995, p. 39) recommends that if one starts with a first object-oriented information systems development project one should prepare to enter areas that are still rather undeveloped.

Some pure object-oriented programming languages (like Smalltalk) are still not very well supported and it might be difficult to find suitable compilers and environments. Hybrid object-oriented programming languages are better supported, but there is always a danger in using them. Moreover, existing programming languages are neither consistent nor interoperable. It might be difficult to connect information systems programmed in different languages. The object concept can also mean different things in different programming languages. (Pancake, 1995) Further there is still a lack of experienced object-oriented information systems developers although this problem is becoming less conspicuous the more wide spread the object-oriented paradigm becomes (Räisänen, 1997a, pp. 14-15).

There is still a lack of good and suitable object-oriented tools like CASE tools, object-oriented databases, object-oriented reuse tools and object-oriented project management tools and there is a lack of experience on how to use the available tools (Bhattacherjee & Gerlach, 1998; Henders, 1998; LaBoda & Ross 1997). There is also a lack of components to reuse, and companies have to put great effort in developing components and libraries that can be reused later on (Graham, 2001; p. 57; Henders, 1998; LaBoda & Ross, 1997; Smith & McKeen, 1996). It might also be rather difficult to find a suitable component to reuse (Graham, 2001, p. 57), and when one finds a component that would be appropriate for reuse there is still a considerable risk that the component has not been updated and is of an older version because version problems are not uncommon (Hopkins, 2000).

In the study by Johnson (2000) the object-oriented paradigm was *not* considered immature by object-oriented information system developers, although a problem with the unavailability of object-oriented CASE tools was pointed out. One has of course to remember that the word 'immature' is a rather subjective word; for example, Noack & Schienmann (1999) state that UML is still ’immature’, and therefore only a notion and not a full life cycle and process focused methodology.

The object-oriented paradigm also constantly matures as new solutions to earlier problems are developed. For example, the problem with connecting the object-oriented paradigm with relational databases can be considered as solved through many different approaches. However, Reinwald et al. (1996) present an example of a solution that made things even more complex.

Analysis, summary and discussion. This problem comes from the fact that the object-oriented paradigm became a major information systems development paradigm in the late 1980’s though the object-oriented paradigm itself is much older. The indications of its immaturity are disappearing all the time, but there are still some significant symptoms such as the following (summary):

- There are few compilers and environments for pure object-oriented programming languages (Pancake, 1995).
- There is a lack of experienced object-oriented information systems developers (Räisänen, 1997a, pp. 14-15).
- There is a lack of object-oriented tools like CASE tools, object-oriented databases, object-oriented reuse tools and object-oriented project management tools, etc. (Bhattacherjee & Gerlach, 1998; Henders, 1998).
- There is a shortage of components to reuse (LaBoda & Ross, 1997; Smith & McKeen, 1996).

It is surprising how many object-oriented areas were still considered immature in the late 1990’s; about 10 years after the object-oriented paradigm became an interesting area of study in the software engineering field. During recent years the immaturity is probably mostly connected to specific areas of the object-oriented paradigm or to specific object-oriented programming languages like Smalltalk.

The following possible association between problems has been identified:

- The OBJECT-ORIENTED paradigm is - > considered IMMATURE in some areas.

3.4.3 Poor support for testing and some other areas in information systems development

Information systems often have little information on object reliability, on performance, on resource utilisation and on security capabilities (Pancake, 1995). The object-oriented paradigm also had poor support for persistence issues and for stylistic guidelines for object-oriented programming (Henderson-Sellers, 1994, p. 21). Wolber (1997) proposes that important characteristics and processes as implementation and testing are poorly supported by most object-oriented information systems development methods.

Henders (1998) proposes that there is inadequate support for integrating the new object-oriented environment and paradigm with existing legacy information systems. Wrappers could, however, be a suitable solution. When the existing traditional legacy information system is a relational database then other solutions like Factory classes can also be used.

Malan et al. (1995) present experiences from Hewlett Packard where they state that most notations used in object-oriented information systems development methods (they used Fusion, OMT by James Rumbaugh, Coad & Yourdon and Shlaer-Mellor) pay little attention to how the method will cope with the size of the problem. Further, subsystems concurrency and real-time systems reuse and requirements were not well supported in the used methods. However, newer methods like the unified method (UML) probably address these missing characteristics in a better way.

Pree (1997) argues that classes/objects implemented in one object-oriented programming language cannot interoperate with classes/objects implemented in another. However, one has to remember that there is support in the object-oriented paradigm for many important characteristics and software issues. For example, the Common Object Request Broker Architecture (CORBA) is a framework defined by the Object Management Group to provide a unified communication layer for object-oriented information systems (Boulanger & Dubois, 1998). CORBA is an API (application programming interface); other APIs are, for example, JNI and RMI. APIs can be used, for example, for connecting legacy information systems with object-oriented information systems (Watson, 1999).

Testing the information system or software is important. According to Brooks' rule of thumb (Brooks, 1979; cited in Webster, 1995, p. 90) the time required for a project should be broken down into one-third for design and prototyping, one-sixth for implementation and half for testing. The object-oriented paradigm has little support for testing and several information systems software development methods are poor in providing guidance for testing (Malan et al., 1995). The major testing problems of an object-oriented information system are according to Kung et al. (1995) the following:

- **The understanding problem**. This problem is due to the encapsulation and information-hiding features. Because a member function of an object might

call another member function of another object, there is a 'delocalized plan' with an invocation chain of member functions. It is difficult to test such a structure where several member functions might call each other in a chain.

- **The complex dependency problem**. This problem comes from the complex relationships that exist in an object-oriented information system. Examples of such relationships are inheritance, association, aggregation, template class instantiation, class nesting, dynamic object creation, member function invocation, dynamic binding and polymorphism, etc. In other words, classes are dependent on each other and testing structures like these is not easy because it is difficult to understand a given class in a large object-oriented information system if the class depends on many other classes. Dynamic binding and polymorphism are also difficult to test.
- **The state behaviour testing problem**. The effect of an operation performed by a method on an object depends on the state of the object. The operation might also change the state of the object. The combined effect of the operations must be tested, which is difficult.
- **The tool support problem**. There are very few CASE tools for testing object-oriented systems.

Changes to an object-oriented program can have many effects that all have to be tested. However, testing is difficult due to the problems mentioned above. In addition, Ramaswamy (2001) argues that testing object-oriented programs can be more difficult and subtle than testing traditional functional programs. Lam (1997) proposes that testing object-oriented programs is somewhat different from testing traditional structural code. In object-oriented programs the information systems developer uses model tests, class tests, cluster tests, system tests, integration tests, regression tests and stress tests. Further the software developer tests iteratively and incrementally. There are of course also other testing strategies like white box testing and black box testing of components (Webster, 1995, p. 206). In white box testing the information systems developer has access to the code of the component, and in black box testing the information systems developer is forced to test all possible calls to the component because the information systems developer only has access to a compiled copy of the component (Webster, 1995, p. 206).

Stevens & Pooley (2000, p. 9) propose that in object-oriented information systems development bugs ought to be easier to find because then one could avoid examining irrelevant modules. Nevertheless, Hatton (1998) argues that testing in C++ is much harder than testing in C; he reports of a case where each C++ correction took more than twice as long to fix as each C correction. This could be the cause because C++ per se is more difficult to test than C, but Hatton (1998) does not think that this is the case, and argues that in general it appears to be more difficult to test object-oriented information systems than functional information systems. However, according to several researchers like Koskimies (1995) and Webster (1995, p. 138) C++ is hard to learn and use, and the circumstances are that some information systems developers have difficulties to perform good C++ programming.

However, one has to remember that C++ is a hybrid programming language and not a pure object-oriented programming language. Hatton (1998) also suggests that corrective maintenance costs are much higher in object-oriented information systems written in C++ than in conventional information systems written in C.

Analysis, summary and discussion. It seems that the object-oriented paradigm in the late 1990's still had poor support for the following issues:

- Object reliability (Pancake, 1995).
- Performance (Pancake, 1995).
- Resource utilisation (Pancake, 1995).
- Security (Pancake, 1995).
- Persistence (Henderson-Sellers, 1994, p. 21).
- Stylistic guidelines for object-oriented programming (Henderson-Sellers, 1994, p. 21).
- Testing (Wolber, 1997).

The reason why the issues above might be poorly supported by the object-oriented paradigm is that the object-oriented paradigm is still immature in some areas. For example regarding testing, it is somewhat difficult nowadays (2005) to understand that there would be poor support for the testing of object-oriented information systems. The extensive book on testing object-oriented information systems by Binder (1999) supports this view.

The following possible associations between problems have been identified:

- The object-oriented paradigm is still IMMATURE in some areas -> which results in POOR SUPPORT FOR SOME AREAS.

3.4.4 Difficulties in measuring object-oriented systems

In the sub section on measuring productivity and quality in information systems development the lack of appropriate object-oriented metrics was discussed. The lack of object-oriented metrics makes the measurement of object-oriented information systems development projects and object-oriented information systems difficult.

Analysis, summary and discussion. It would be favourable to be able to measure object-oriented information systems because one would like to compare object-oriented information systems with each other, as well as compare object-oriented information systems with similar traditional functional information systems.

There are some object-oriented metrics on the market. The problem seems to be the lack of experience in how to use these metrics (Räisänen, 1997a, p. 16) and the difficulties in finding suitable metrics for a specific object-oriented software development project.

The reuse concept seems to be a rather challenging concept when working with object-oriented metrics (Kan, 1995, p. 31). If a lot is reused it might be difficult to measure the development progress. Other concepts that affect the measurement are (Berard, 1998):

- Encapsulation and information hiding.
- Polymorphism (Webster, 1995, p. 96).
- Inheritance.
- Localisation.

If these object-oriented concepts would be integrated in good object-oriented metrics and the metrics would be commonly known and well tested, then there would be great progress in the area of object-oriented metrics.

The following possible association between problems has been identified:

- The object-oriented paradigm is still IMMATURE in some areas -> one area is software metrics which results in DIFFICULTIES IN MEASURING OBJECT-ORIENTED SYSTEMS.

3.4.5 Training & lack of experience

Many people consider the object-oriented paradigm hard to understand and use because it is based on new concepts like encapsulation and inheritance, new programming techniques like using classes and objects and new database modelling techniques, etc. The object-oriented paradigm might further introduce new working roles like class designers, object architects and object-oriented programmers. The birth of new working roles is of course dependent on how tasks are divided among the personnel. Further the work processes might change and new work processes such as reuse-based development are introduced. (Page-Jones, 1998)

Booch (1994, pp. 288-289) and Sheetz & Tegarden (1996) argue that it takes time for the information systems developers and managers to learn how to use the object-oriented paradigm. The object-oriented paradigm is very different from traditional paradigms (Lam, 1997) and one can talk about a change in the mindset and a new way of working and thinking (Henderson-Sellers, 1994, p. 21; Jenz, 1999a).

Bhattacherjee & Gerlach (1998) propose that it might take several years to learn how to carry out first-class object-oriented information systems development, though Berard (1998) proposes that it takes the average information systems developer about 6 months to become comfortable with the object-oriented paradigm. According to Johnson et al. (1999) it takes about six weeks for a programmer to become proficient with an object-oriented programming language.

According to findings presented by Berard (1998) an object-oriented information systems developer should plan on taking about one day per class to eventually fully understand all the classes in the class library under consideration. McClure (1996)

recommends two days of hands-on training per class. In addition, Fagerström (1995, p. 225) reports that information systems developers have often had difficulties in learning how to use class libraries, mostly because they consist of numerous classes and complex relations between the classes. It is difficult to navigate through existing libraries of classes (Henderson-Sellers, 1992, p. 59), it is difficult to analyse a problem in terms of objects and it is difficult to perform good object-oriented analysis and design (Bohrer et al. 1998). Further Henders (1998) suggests that some information systems developers do not understand the value of reuse.

Cockburn (1998, p. 2) proposes that it might take as much as nine months in order to fully earn back the salaries of the information systems developers; if nine months is multiplied with the hundreds or thousands of developers that companies will have to train, then the total cost is staggering, and many business executives will not consent to such extent. The problem with the lack of knowledge, training and experience with object-oriented projects might also be so substantial that managers might fear that adopting the object-oriented paradigm could have unanticipated impacts on mission-critical activities in the company (Bhattacherjee & Gerlach, 1998).

Further, many information systems developers are not very willing to shift from the old traditional functional paradigm to the object-oriented paradigm (Henders, 1998; Malan et al., 1995). This might be because they have considerable experience in functional information systems development and it is always hard and time consuming to become an expert in a new field. Some information systems developers think moreover that the traditional functional paradigm is simply the best for solving information systems development problems and for building information systems (Lam, 1997).

There might also be political reasons that make information systems developers unwilling to adopt the object-oriented paradigm; for example that they do not, nor wish to, understand it, that they are afraid that they will not be able to understand it or that they in fact do want to adopt it but in their own way, etc. (Webster, 1995, p. 58). Some information systems developers that are good at traditional structural information systems development might feel that they would be worse at object-oriented information systems development and are therefore not willing to move into this new area (Webster, 1995, p. 98). Sim & Wright (2002) even propose that it would be more difficult to learn the object-oriented paradigm for information systems developers that are experienced in the traditional functional paradigm, than it would be for novices.

Nevertheless, starting to use the object-oriented paradigm might also motivate the personnel, as reported by Davis & Morgan (1993). In addition, Capper et al. (1994) state that the real challenge for the information systems developers at IBM in the United Kingdom was actually the *change* from the functional paradigm to the object-oriented paradigm.

According to Noack & Schienmann (1999) and Radin (1996) it might be difficult to find experienced object-oriented information systems developers. This might be due to the fact that many developers are fully occupied by their current work and have no time to start studying a new software development paradigm (Fayad et al., 1996).

Many information systems developers do not know where to start when moving to object-oriented information systems development and are neither sure that it is worth the effort (Fayad et al., 1996; Malan et al., 1996, pp. 32-34). The management of object-oriented projects is important and only experienced managers should manage first-time object-oriented projects (Ramaswamy, 2001). Radin (1996) also claims that inexperienced information system developers tend to write object-oriented programs that are slow, difficult to test and hard to debug. Education, mentoring and gaining experience are therefore important issues. Not only working with information systems developers is important for management, management has also to consider issues like how this way of developing information systems might affect issues like the business strategies of the company (Jenz, 1999a).

Hohmann (1996) found that students at universities learn object-oriented concepts rather easily but have difficulties in creating solutions that are based on the concepts. Teachers that teach the object-oriented paradigm in first year courses have reported that they find teaching this concept more difficult than teaching the traditional functional paradigm (Kölling & Rosenberg, 2002). This is not due to the complexity of the object-oriented paradigm, but due to a set of other factors like a lack of experience and knowledge of how to teach these courses, unfamiliarity with the object-oriented paradigm, inadequate teaching materials, a lack of suitable software tools and problems with moving from the algorithm based view.

Love (1993, p. 162) presents some recommendations for training object-oriented information systems developers. First, the developers have to learn the concepts of object-oriented information systems development, which takes about one week. Then the traditional base programming language (of the hybrid programming language in question, if such is used) is studied, which takes about two weeks or more depending on the prior experience of the developers. The next step is to learn the object-oriented programming language and component library by using prototyping and developing sample programs; this step takes approximately 4-6 weeks for prototyping. Learning the component library takes about a further two weeks for 20 components (this is not true for larger libraries that must be learnt through experience). The last step is to plan a project (taking about one week) and to develop the first real object-oriented information system, taking about 20-40 weeks. During the latter stages guidance from a mentor or an experienced object-oriented information systems developer is usually needed.

Gehringer & Manns (1996) studied the problems of the object-oriented paradigm by asking managers who directed object-oriented programming projects. The findings were the following (quotation):

> Rate the next twelve statements on a scale of 1 (=strongly disagree) to 5 (=strongly agree)
>
> A roadblock we have faced in using the object-oriented paradigm has been
>
> | | |
> |---|---|
> | Retraining employees | 3.88 |
> | Few qualified recent graduates | 3.57 |
> | A lack of quality class-libraries | 3.47 |
> | The poor quality of O-O development environments | 3.06 |
> | Poor quality or an absence of needed tools | 3.32 |

The results indicate that in 1996 it was problematic to get trained information systems developers. If one faces a problem finding trained developers one has to develop the object-oriented skills among the developers in the company. Some experiences of how this can be done is presented by Malan et al. (1995) who present issues such as training, hands-on experience, mentoring, pilot projects, continued learning and identifying and solving problems, etc. Joos (1994) proposes that *education* is important and presents education forums like seminars, conferences and workshops. Steinmann (1992) recommends *pilot projects*.

One way of solving the problem with this lack of experienced object-oriented information systems developers is to let the object-oriented learning occur during project work with the help of a *mentor* (Ramaswamy, 2001; Sircar et al. 2001).

However, the learning of the object-oriented paradigm should not interfere with existing information system development work and ongoing projects (Bhattacherjee & Gerlach, 1998). Webster (1995, p. 35) proposes that a culture of learning the object-oriented paradigm is needed, and in order to achieve this the managers and developers that progress the most need to be rewarded. Books on object-oriented learning culture, relevant journals etc. are a necessity, and the developers should have time to read them (Webster, 1995, p. 35).

Analysis, summary and discussion. The object-oriented paradigm is not the only software development paradigm and a lot of information systems development work is still carried out consistent with the functional software development paradigm. The reason for this might be that the object-oriented paradigm is still *immature,* or perhaps there is a resistance to move into a new way of developing information systems. The *complexity* of the object-oriented paradigm might also be a notable hindrance for adopting object-oriented information systems development.

A lack of training in the object-oriented paradigm has been a problem, and although today a majority of universities and schools in the information systems development field teach the object-oriented paradigm, there are still companies that might experience a difficulty in finding trained object-oriented information systems developers. Further older information systems developers who are experienced in some other field are not always interested in starting to develop object-oriented information systems.

The following possible associations between problems have been identified:

- The object-oriented paradigm is still IMMATURE in some areas -> and it might be difficult to find TRAINED and EXPERIENCED information systems developers in these areas.
- The Object-oriented paradigm has high COMPLEXITY -> which makes TRAINING more difficult and there are several information systems developers that have a LACK OF EXPERIENCE.

3.4.6 Efficiency in two different areas

In this sub section two very different types of efficiency problems connected to object-oriented information systems development are presented.

1. There are often problems with efficiency because object-oriented information system development might take a lot of computer processing time (Booch, 1994, pp. 288-289; Johnson et al., 1999). Henderson-Sellers & Edwards (1994, p. 21) also propose that performance issues have to be solved before the object-oriented paradigm can be fully utilised. In the study by Johnson (2000) decreased system run-time performance was also considered a problem.

2. Page-Jones (1992b) discusses efficiency in another fashion and warns about starting to use the object-oriented paradigm if effective information systems software development is desired but there is no suitable repository with components for reuse. If the information systems developers are forced to develop reusable classes in connection with information system development work, the overall efficiency will probably be poor (Page-Jones, 1992b). Much extra effort is needed to develop reusable classes, and Page-Jones (1992b) presents a rule of thumb that proposes that it takes about 20 days for a person to develop a class for use immediately, and about 40 days for a person to develop a reusable class.

Analysis, summary and discussion. It seems that in the late 1990's the object-oriented paradigm was still suffering from efficiency problems in the following areas:

- Object-oriented information systems development takes up a significant amount of computer processing time and system run-time was poor because of the object-oriented paradigm.
- Developing *reusable* classes in connection with the ordinary information systems development work effects the efficiency of the information systems development project negatively.

The trend during recent years has been that the speed and efficiency of computers have become better all the time and hardware efficiency problems have usually decreased to a corresponding degree. However, what the future holds is difficult to predict.

The need for developing reusable components will also decline when the object-oriented paradigm becomes more mature and there will be more reusable components in companies, organisations and on the market.

The following possible associations between problems have been identified:

- The object-oriented paradigm is still IMMATURE in some areas, which results in -> EFFICIENCY problems.

3.4.7 Costs

The starting costs are often huge when one starts to develop a completely new object-oriented information system, because there is nothing that can be reused and everything has to be developed from scratch (Booch, 1994, pp. 288-289). Building for reuse can be very expensive and it can cost 3-10 times as much as merely developing the information system (LaBoda & Ross, 1997). Cost issues are connected to productivity issues but the productivity issues are presented in a sub section of their own in this study.

According to Jacobson (1993) the first object-oriented projects are often expensive and one has to see the object-oriented information systems development project as an investment for the future. Page-Jones (1992b) also presents this "technology trap" where the first object-oriented information systems development projects are costly and not very productive because there is nothing to reuse. The first projects are frequently expected to produce robust libraries with reusable components and classes, although these first projects cannot utilise the reuse concept because there is nothing to reuse.

Mili et al. (1995) present further the indirect and direct costs of including a component into a library of reusable components and the cost of integrating and/or adapting the component.

Analysis, summary and discussion. The starting costs for a new object-oriented information systems development project are usually high because there is nothing to reuse. As the object-oriented paradigm gets more mature this problem diminishes though there are also other costs with reuse as Mili et al. (1995) indicate.

Another issue that will most likely affect the costs negatively is the costs of finding components or other object-oriented software artefacts to reuse. This is an issue that Mili et al. (1995) discuss in detail; they also present the following formula for the average cost of attempting reuse in the following way (quotation):

$$[Search + (1-p) \times Development]$$

Where *Search* is the cost of searching for a component in a database, *Development* is the cost of developing the component from scratch, and p is the probability that the component is found in the database. The reuse option is feasible only if (quotation):

$$[Search + (1-p) \times Development] < Development \quad \text{or}$$

$$Search < p \times Development.$$

The following possible association between problems has been identified:

- The object-oriented paradigm is still IMMATURE in some areas, which results in -> higher COSTS.

3.4.8 Limited usability of components

Existing components and frameworks connected to products on the market are mainly intended for graphical user interfaces and other special areas (Schmidt & Fayad, 1997); therefore, it might be difficult to find reusable components in other areas (Garlan et al., 1994; Szyperski, 1999, p. 11). Another problem is that components are seldom built explicitly for reuse and therefore are difficult to reuse (Garlan et al., 1994; Radin, 1996). One has to plan for component reuse and anticipate the various ways one might use and reuse a component and build the component accordingly (Webster, 1995, pp. 223-225). It is also recommended that one builds class libraries and frameworks that are easy to find and use (Jolin, 1996).

Finding components for reuse is a notable problem in many object-oriented information systems development projects. Radin (1996) proposes that intelligent search engines might be a solution to this problem. Even if the information systems developer finds some promising components, significant problems often remain because the chosen components do not fit well together. Often this difficulty is due to low-level problems of interoperability like incompatibilities in operating systems, programming languages and database schemas, etc. (Garlan et al., 1994) Reusable components that are developed in different programming languages are usually the problem (Konstantas, 1995, p. 70) but there are also other mismatch problems, for example, different software architectures might have different suppositions about the reusable components, the operating system and the programming language might be the same, but mismatch problems still arise if the software architectures are not the same (Garlan et al., 1994).

Another problem is several different versions of a component. When a system that uses components grows and becomes larger, there will be a need for new functionality in the components because of new requirements, reuse, new software tools and new operating systems, etc. This results in several different versions of a component, or very complex single components and the usability of the components suffers. One solution to this problem is to use a distributed component platform like EJB (Enterprise JavaBeans from Sun Micro Systems), ActiveX (from Microsoft) or CORBA.

Analysis, summary and discussion. In order to utilise the reuse concept one has to be able to find appropriate and high-quality software components to reuse, and the components ought to be usable, which is not always the case. The problem is mainly due to the immaturity of the object-oriented paradigm, and has the following symptoms:

- Difficulties in finding components to reuse (Garlan et al., 1994).
- Reusable components are very complex (Jarzabek & Knauber, 1999).
- Potential reusable components are not built for reuse (Garlan et al., 1994).
- Reusable components are difficult to utilise (Radin, 1996).
- The reusable components do not fit well together and neither do they fit well with the existing components in the information system under development (Garlan et al., 1994).

- Reusable components can have several different versions and it is difficult to resolve which component is the most suitable for a specific reuse situation (Jarzabek & Knauber, 1999).

The following possible association between problems has been identified:

- The object-oriented paradigm is still IMMATURE in some areas, which results in -> LIMITED USABILITY OF COMPONENTS because there are no suitable components to use or existing components are complex or difficult to reuse.

3.4.9 Problems with reuse

The reuse concept has often been presented as the most promising concept in the object-oriented paradigm. However, several authors like Eeles & Sims (1998, p. 59) propose that reuse has not been the success one expected. However, reuse is the base for using objects and classes, and not using reuse when working with them might be worse than traditional information systems development (Maring, 1996).

Nierstrasz & Dami (1995, pp. 6-7) refer to Jon Udell (1994) who claims that components that are delivered with systems like Visual Basic are a more successful example of software reuse than object-oriented programming employing reuse. Nierstrasz & Dami (1995, p. 7) write about an interesting debate on the Internet that followed the claim by Jon Udell (1994). As a result of the discussion several researchers came to the conclusion that software reuse is a matter of methodology and design, and more than just technology. Object-oriented systems per se do not come with reuse though reuse can be utilised with the object-oriented paradigm.

Finch (1998) proposes that among companies that use the object-oriented paradigm there is actually little reuse; some industry watchers report 15 percent average reuse, whereas a major consulting firm reports 25 percent reuse, etc. Nevertheless, there are also figures of 80 percent reuse and therefore the real picture is actually a bit fuzzy (Finch, 1998). Mili et al. (1999) claim that although there are many examples of reuse, the fact is still that the promises of reuse remain for the most part unfilled. Jenz (1999c) goes even further and claims that reuse actually works very poorly. However, the claim by Mili et al. (1999) is based on the assumption that reuse will not occur if there are no reusable ‘assets’ and that good domain engineering is important in achieving them. The lack of assets is therefore the reason why reuse often fails to succeed. Further, if assets are found, some have to be reused more than thirteen times before the costs of developing the reusable assets are covered (Frakes & Isoda, 1994). What kinds of assets are there and what important issues have to be considered in order to perform proficient reuse? Mili et al. (1999) present the following:

Assets:

- Assets that abstract a function, for example, abstract data type implementations in Ada.
- Assets that abstract a structure, for example, design patterns.

Issues:

- How assets are represented; the properties of an asset should be represented well.
- How assets are matched; there are differences in matching assets based on functions with queries and matching assets based on structures with queries
- How assets are developed; black-box reuse is designed in another way than white-box reuse.
- How assets are used or reused; assets that embody a function might only be used for black-box reuse, assets that embody a structure are usually used for white-box reuse.

There are furthermore other more specific problems with reuse. According to Meyer (1995, p. 111), Nokso-Koivisto (1995) and Radding (1999) information system developers often avoid reusing existing modules, because they claim that the modules do not work, or it is not worth the effort to figure out how they work. There is a *not-invented-here* (NIH) problem where the information system developers do not trust the work of other information system developers (Coad & Yourdon, 1991, p. 137; Eriksson & Penker, 1996, p. 156; Henderson-Sellers & Edwards, 1994, p. 427; Schmidt & Fayad, 1997). The NIH problem usually concerns all artefacts that other information systems developers have developed, classes, frameworks and design patterns. The more distant the artefact is, the worse the NIH problem becomes. As a rule, one can thus argue that reuse within a project is fairly easy; across projects it is fairly hard and across an organisation it is exceedingly difficult. Further, there are some developers that think it is better to develop new components from scratch instead of reusing existing components (Sparling, 2000).

In fact, it is often hard to develop reusable classes (Nierstrasz et al., 1992) and the motivation for object-oriented information systems developers to build such modules is often poor (Eriksson, 1992, p. 86). Capper et al. (1994) report that developing the initial classes with reuse in mind is strenuous and time consuming, and one has to plan for this activity in order to get the object-oriented information systems developers motivated to develop reusable classes. In fact, in order to achieve better motivation among information system developers, a new ‘reuse culture’ should be incorporated in the company (Jenz, 1999b).

According to Pittman (1993) the quality of the components is seldom a serious problem. The manager of the information systems developer must be aware of the fact that the “not invented here” problem might effect the quality of the information systems being developed and therefore endorse the information systems developers to reuse components (Fichman & Kemerer, 1993; Pittman, 1993). If the information systems developers do not reuse components, expedient and tested functions and algorithms that are integrated in some components will not be utilised (Love, 1993, p. 220). Pittman (1993) proposes that a purchased component will most likely be of better quality than a component that is developed for the same price in the company. In practice individual information systems developers have probably developed components that they reuse; but successful reuse of other components in the company is rare (Maring, 1996).

However, according to Frakes & Fox (1995) most information system developers still prefer to reuse instead of developing new modules. Radding (1999) and Stevens & Pooley (2000, p. 218) also propose that using prebuilt components offers the best way to fast enterprise information systems development. However, buying components is not a self-evident matter; when buying a component a lot of new code comes to the company and the code has to be maintained (Love, 1993, p. 219). The quality of the commercial component that is bought from outside the company might, however, be better than a similar component developed within the company (Stevens & Pooley, 2000, p. 218).

However, by using existing components information systems development time can be reduced by 50% - 60% (Radding, 1999) and the information systems development costs can be reduced by 20%; and this in an industry where even a 1% saving could give a competitor a significant advantage (Graham, 2001, p. 46; Henderson-Sellers, 1996, p. 19). These opinions are supported by the experiences at AT&T Bell Labs, which moved into object-oriented information systems development, and experienced a total development time saving of 60% (Wilkie, 1993, p. 40). The question of how many changes to a component raise the reuse cost is also interesting. Pree (1997) proposes that only a few changes (12%) to a component raise the reuse cost to 55% compared to the costs of developing the same component from scratch; here the changes themselves do not cost that much, it is the required understanding of the component that generates costs.

Furthermore, Nokso-Koivisto (1995) and Radding (1999) found that reuse of components often cannot be carried out because *no adequate component can be found*. Henderson-Sellers (1993) and Coleman et al. (1994, p. 7) also present the problem of finding the desirable class. Class management is, however, dealing with this problem. For example, Gibbs et al. (1990) present issues like class management, class evolution, class packaging, class organisation and class selection. By learning how to work with class management, developers probably get enough assistance in selecting classes (Gibbs et al., 1990). Page-Jones (1992b) proposes that a special librarian is needed who maintains the quality of the libraries with reusable classes. The quality of the libraries is important; one has to work with questions like naming (several components might have the same name), configuration control (the functionality of the components changes, versioning questions), accessing (how to find components), dependencies (components are often dependent on other components), distribution (how updates etc. are distributed), responsibilities (who is responsible for a given component) and testing, etc. (Love, 1993, p. 218). In addition, a library consultant could be used who assists the information systems developers in their reuse of classes (Page-Jones, 1992b). If both these persons existed in the company then finding and using the reusable classes would probably be much easier.

The problem of finding the appropriate class is important, and it might occur when one is starting to use standard libraries that come with the environment of an object-oriented programming language. If there are hundreds of classes, a great deal of effort is required from the programmer before an understanding of all the standard library classes is achieved. The programmer might find it easier to program an adequate class

himself/herself than start searching for a suitable class among hundreds of standard library classes. (Gibbs et al., 1990)

There are also studies, of course, where there has been no problem in finding classes. Caliò et al. (2000) present one study and state that relevant architectural system components for user interfaces and data management were easily found among the components provided by Ms Windows and Visual Basic environments. Further Mili et al. (1999) argue that about 50 solutions are available to the problem of how to find a component. The authors present the retrieval method and the browsing method as the most important in finding assets in a component library. The retrieval method is based on queries and the browsing method is based on relevance guidelines (Mili et al., 1999).

Another obstacle to the reuse concept is the resistance to develop generalised classes because in such a case other teams in an information system development project could use these classes and in so doing become more productive than the team who initially developed the generalised classes (Henderson-Sellers, 1993). If, for example, a business line pays for an initial development, the benefits from reduced testing through reuse may actually be experienced by another - perhaps competing - business line (in the same or in another business unit) in later projects (Henderson-Sellers, 1996, p. 22). Productivity is also discussed by Coad & Yourdon (1991, p. 137) who argue that when productivity is measured by the number of code lines produced, reuse is not feasible. Jenz (1999a) and Maring (1996) argue that in order to achieve better productivity with reuse, companies have to give rewards to programmers who develop and use reusable components. Graham (2001, p. 56) also discusses problems with unwillingness or resistance to develop components that can be reused in the future because of cost reasons and because of a suspicion that the components will, in fact, not be reused.

Frakes & Fox (1995) did a study where it was found that the choice of programming language does not affect code reuse. Reuse in object-oriented and traditional languages was almost the same. Another issue to contemplate is the issue of reuse between programming languages; if a component has been programmed in one programming language like C++ it might be difficult to reuse it in an environment that is programmed in another programming language like Smalltalk (Eriksson & Penker, 1996, p. 154).

The hierarchy of classes can also be a hindrance for reuse. If a programmer needs a simple class that is down in the hierarchy and has perhaps four or five superclasses, then the programmer gets a lot of unnecessary classes and code when the whole hierarchy in the program has to be taken in just to get one class. If the resources of the computer are limited, the extra classes are probably unwanted and the programmer develops the wanted class himself/herself (Webster, 1995, p. 226). An object-oriented programmer (Wrede, 1998) also presented this problem to the author of this study. Further, a hierarchy in a class library can also be difficult to integrate into the existing class hierarchies in the company (Eriksson, 1992, p. 356). In a hierarchy there are often many classes and ‘unwanted’ classes easily come into the information system in the company when a class library is reused (Eriksson, 1992, p. 356). The hierarchy of classes that result from extensive reuse also often becomes difficult to document (Manhes, 1998).

Other obstacles to reuse might be the lack of good textbooks on how to perform reuse, unsuccessful experiences with reuse from the past (Coad & Yourdon, 1991, p. 137; Räisänen, 1997b, p. 35), technical difficulties in the form of information systems that are delivered as single executable programs in machine code (Eeles & Sims, 1998, p. 59), difficulties in understanding and using available components, a lack of tools that support reuse, no rewards for utilizing reuse (Räisänen, 1997b, p. 35), problems with the copyright for reusable components when working with several clients (Eeles & Sims, 1998, p. 59; Räisänen, 1997b, p. 35) and a shortage of time which means that immediate business needs have to be considered (LaBoda & Ross, 1997). Webster (1995, p. 216) proposes that reuse is difficult because reusable information systems must be more general (and therefore often become larger and more complex), similarities among projects are often small and that many of the things that can be reused already exist in the application environment or in the operating system. Verschoor & Low (1994) also argue that reusing classes usually results in information systems that are more complex, and the more a class is reused the more complex the information system becomes.

Further, Fayad et al. (2000) propose that in smaller start-up companies (less than 50 employees) the development of components for reuse might not be an advantage. The higher cost and increased time needed for developing reusable components are less important than releasing the product (Fayad et al., 2000). Lim (1994) refers to a number of projects at Hewlett-Packard where it cost twice as much to develop a reusable component than to develop an ordinary component. Jenz (1999c) thinks that is takes about 2-4 times more effort to develop reusable components than to develop ordinary components. However, if the company is successful then releasing the product is probably not the most important issue, and reusable components for the future can be developed (Fayad et al., 2000).

There are also other dangers with reuse that information systems developers have to be aware of (Murphy, 2001). These dangers are connected to the reuse of ready-made components in a new environment or context. Jézéquel & Meyer (1997) report on the catastrophic error that was actually a reuse error in the Ariane project that led to the destruction of the unmanned Ariane 5 rocket. The main cause was the failure of an assertion in a piece of code that was performing an unnecessary calculation (which exceeded its set of expected values). This came as a result of a crucial reuse of the Ariane 4 program code (Murphy, 2001). The error would have been avoided if the programmer had taken the Ariane 5 project into consideration (Murphy, 2001).

Analysis, summary and discussion. It appears that reuse is a very strong feature in the object-oriented paradigm. If the reuse feature is used accurately several benefits can be achieved. However, in order to carry out appropriate reuse one has to realize that reuse is not only about technology; reuse is also a question of economical, human and organisational factors, as Nierstrasz & Dami (1995, p. 7) argue.

One also has to be aware of the problems and pitfalls with reuse, and remember that the management of reuse is important. Actually, the management of object-oriented information systems projects has to develop a culture where the reuse of components is as palpable as programming in its own right.

The following problems connected to reuse were found (summary):

- There is a lack of components and other assets to reuse (Mili et al., 1999). This problem could probably disappear when the object-oriented paradigm becomes more mature.
- There is a not-invented-here problem (Radding, 1999). This is rather an old problem that has existed with functional programming as well (regarding modules, etc.). Everyone wants to be a master in information systems development and one does not 'trust' the work of others. A rather challenging problem that has no clear solution.
- Out of reuse a lot of programming code comes into the company that has to be maintained (Love, 1993, p. 219). If there is good documentation on the programming code or the 'black-box' components, then this problem is perhaps not that serious.
- No suitable component for reuse is found (Coleman et al., 1994, p. 7). The development of suitable tools for finding components will probably be available very soon. The problem is primarily connected to the immaturity of the object-oriented paradigm.
- Technical difficulties in the form of information systems that are delivered as single executable programs in machine code (.EXE file), and from which there is no possibility to take out a class that one wants to reuse (Eeles & Sims, 1998, p. 59). The solution is of course to try to get the source code for the class one wants to reuse.
- Resistance to develop generalised classes or other components because this means a demand on resources, and then there is a danger that someone else gets the credit (Henderson-Sellers, 1993). The solution to this problem is probably a system with some kind of rewards for developing artefacts for reuse (Henderson-Sellers, 1993).
- The hierarchy of classes might be a hindrance for reuse in the form of inheritance (Webster, 1995, p. 226). Careful and skilful use of inheritance is of course the solution, and experienced programmers can probably deal with this issue. The problem is mainly connected to the immaturity of the object-oriented paradigm.
- Reuse and inheritance often increase the complexity of an object-oriented information system (Sheetz & Tegarden, 1996).
- A lack of good textbooks on how to carry out reuse (Coad & Yourdon, 1991, p. 137). The more mature the object-oriented paradigm becomes, the less important this problem will turn out to be.
- A lack of tools that supports reuse (Räisänen, 1997b, p. 35). The more mature the object-oriented paradigm becomes, the less important this problem becomes.
- Problems with the copyright of reusable components when working with several clients (Eeles & Sims, 1998, p. 59; Räisänen, 1997b, p. 35). This is an

interesting problem when working in a software company with several clients. Should one have different reusable classes for different clients, or could the same components be used for different clients? This is both a legislative as well as a marketing problem.

- Reuse is not feasible because similarities among projects are often small (Webster, 1995, p. 216).

The following possible associations between problems have been identified:

- The object-oriented paradigm is still IMMATURE in some areas (for example, a lack of good textbooks), which results in -> PROBLEMS WITH REUSE.
- By using REUSE inheritance structures can be developed which increase the -> COMPLEXITY of the information system.

3.4.10 Problems with object-oriented analysis

Object-oriented analysis has been criticised by Höydalsvik & Sindre (1993) and Wilkie (1993, p. 85). Höydalsvik & Sindre (1993) found that object-oriented analysis does not fulfil the purposes of analysis (because, for example, requirements exist prior to object-oriented analysis), and that object-oriented analysis has no smooth transition to design. Wilkie (1993, p. 85) proposes that the mixing of analysis and design methods is a problem - especially for the information systems development project manager who must measure project progress - this is because the management (planning and control) of software development projects becomes more difficult, due to the mixing of analysis and design in object-oriented information systems development.

Kaindl (1999) proposes also that the object-oriented analysis model cannot be a part of the object-oriented design model. This is because there are differences in the representation of the objects in the different models, and the model of the domain inside the deployed program can be different from the object-oriented analysis model in many ways (Kaindl, 1999). However, Maring (1996) reports that some domains lead themselves to clear class definitions.

If one achieves a smooth transition to object-oriented design, there is the risk that analysis has not fulfilled its purposes, and vice versa. McGinnes (1992) identified some requirements that analysis has to fulfil; these were the following: ‘Multiple views of problem situation’, ‘Easily understood method’, ‘Relating views at different levels’, ‘Richness’, ‘Recognizable terminology’, ‘Technical content’, ‘Objectivity’, ‘Minimal solution’ and ‘Self-checking method’.

In many cases, there is a requirement capture phase before analysis in the information systems development life cycle; for example, de Champeaux et al. (1992) present such a phase. Because many object-oriented analysis methods assume that the requirements are identified before analysis starts (McGinnes, 1992), it is perhaps wrong to use the term *analysis* to characterise object-oriented analysis. The aim of analysis is to describe the problem and the user requirements, but what is there to analyse if the requirements are

already clear? Höydalsvik & Sindre (1993) argue that object-oriented analysis can thus be seen as a preliminary design rather than as pure analysis.

Kaindl (1999) proposes that it is difficult to move from object-oriented analysis to object-oriented design, because object-oriented analysis objects represent different things than object-oriented design objects do. Therefore an object-oriented analysis model cannot easily become an object-oriented design model. This leads to a situation where the information systems developers have to specify the architecture for the software and build a model of the domain that the information system is going to use. (Kaindl, 1999)

The transition to design from object-oriented analysis is based on the objectiveness of object-oriented analysis. It is possible to develop analysis models that have a clear connection to design, but this is not always the case, as Höydalsvik & Sindre (1993) and McGinness (1992) argue. If there is no distinction between object-oriented analysis and object-oriented design, then the objects in the system have to be equivalent to the objects in the real world. This is, however, seldom the case; often the real world has to be 'modified' to fit into the object model (McGinnes, 1992). The purpose of object-oriented analysis can be found in the general objectives of analysis; the smooth transition to design is questionable (McGinnes, 1992). However, in an information systems development project reported by de Champeaux et al. (1992) there was a smooth transition from object-oriented analysis to object-oriented design.

Many researchers claim that object-oriented modelling is natural and that objects are natural ensembles for many concepts in the real world (Booch, 1994, p. 78, Jacobson et al., 1992, p. 44). This means that the way people perceive the structure and behaviour of a system is connected to how the object-oriented model is constructed. Is the object-oriented analysis then problem-oriented? Höydalsvik & Sindre (1993) claim that this is probably not the case; in fact object-oriented analysis is quite domain oriented. This is due to several reasons, Höydalsvik & Sindre (1993) claim that good analysis does not arise simply from a model which is in accordance with the way humans think; object-oriented representation is not suitable for all kinds of knowledge; the analysis ought to be close to the user and not to the software engineer, and the main motivation for choosing object-oriented analysis is that the following steps in the software development process are also object-oriented. When discussing what is natural for users, interestingly Ellis & Gibbs (1989) claim that people will find it natural to think in terms of active object systems.

Höydalsvik & Sindre (1993) present business rules and processes as two examples of knowledge that are difficult to present in an object-oriented structure. It is important to remember that the real world should be modelled in the way that the users find accurate, if the users stress processes, a process-oriented view might be better than an object-oriented view. The users' way of thinking is important, and the information systems developer has to consider that the idea that object-oriented information systems are a natural representation of the world is probably an over-simplification (McGinnes, 1992).

Sommerville (1992, p. 66) claims that humans are flexible and can switch regularly between different ways of looking at the same information system, but the writer (1992, p. 193) also argues that it is difficult to find objects in object-oriented design because people's 'natural' view of many information systems is functional. Canning & Nethercott (1996, p. 125) propose that the development of the object model follows standard object-oriented principles in that the problem domain (the business) is examined to identify objects and relationships for inclusion in the model. Canning & Nethercott (1996, p. 125) go on and claim that users have little trouble understanding this approach. In other words, the practical finding supports the opinion of end users rather easily identifying objects in the real world.

Furthermore, Aksit & Bergmans (1992) found several obstacles in object-oriented software development, the problems related to preparatory work (in the analysis phase) and are inherent to object-oriented software development methods. Preparatory work means the mapping of the real world entities and the objects that are the entities in the analysis model. The problems found by Aksit & Bergmans (1992) are the following:

- Identification of Problem-Domain Structures.

  It was often difficult to identify classifications in the problem domain that could be mapped to inheritance hierarchies.

- Dealing with Excessive Domain Objects.

  Integrating the domain knowledge with the user's requirement specifications can yield a lot of objects. Only few of these objects may be relevant to the problem area.

- Early Decomposition.

  If subsystems are not identified before objects are identified problems will evolve, because objects have to be placed into some subsystem when identified. If the subsystems are identified before object identification, the boundaries of the subsystems may not be optimal.

- Subsystem-Object Distinction.

  In the analysis phase objects may act as subsystems if they are complicated. Subsystems can also be defined as objects if they can be structured in a hierarchy and reused.

- Commonality versus Partitioning.

  Because subsystems partition the system, classes that are members of the same hierarchy can be spread over several subsystems. Finding the suitable inheritance hierarchies becomes difficult.

- Subsystems Identification Using Object Interactions.

  Subsystems are often used for structuring interactions among objects; however, most object-oriented methods only have intuitive techniques for subsystem identification.

The problems above are discussed by Aksit & Bergmans (1992) who claim that they arise due to the size and complexity of the problem domain and how the problem domain is modelled. Aksit & Bergmans (1992) also have a solution to these problems; a new concept called *composition filters*. The rather rare Sina programming language adopts composition filters.

Finally, one can mention that it has been reported that object-oriented analysis is sometimes slower than traditional analysis (Koskimies, 1997, p. 5).

Analysis, summary and discussion. Object-oriented analysis is an information systems development activity and not a pure problem; however, object-oriented analysis is classified as a problem in object-oriented information systems development, because object-oriented analysis is considered problematic by several researchers and authors.

McGinnes (1992), Kaindl (1999) and Höydalsvik & Sindre (1993) take up several interesting and even very critical aspects of object-oriented analysis. For example, Kaindl (1999) proposes that the differences between what is modelled in the analysis phase and what is modelled in the design phase might initiate more deliberate development approaches. Another claim that is important is that object-oriented analysis does not fulfil all the tasks of analysis.

Analysis is, however, the stage in the information systems development life cycle that many researchers consider the most important, because if one starts to build the wrong product the consequences will be severe. The close connection between object-oriented analysis and object-oriented design is a major challenge in this context. However, the fountain like life cycle includes better opportunities to return to earlier work in design stage, and offers new potential for managing the whole analysis and design process. One has to remember that the fountain life cycle model is only one model for object-oriented information systems development; there are other models as well.

The following problems with object-oriented analysis were found (summary):

- The mixing of object-oriented analysis methods with object-oriented design methods is a problem (Wilkie, 1993, p. 85). It is difficult to determine where object-oriented analysis ends and object-oriented design starts when following the popular fountain life cycle in object-oriented information systems development (Monarchi & Puhr, 1992). This problem is probably worst for the project manager who has to work with milestones and resource allocations. For traditional information systems developers the problem is less important and some information systems developers probably think that it is practicable with a smooth transition from object-oriented analysis to object-oriented design.
- Object-oriented analysis does not fulfil the purposes of analysis because requirements exist prior to object-oriented analysis itself (Höydalsvik & Sindre, 1993). There are many different object-oriented analysis methods, and when object-oriented analysis is made in dexterity with all these various methods, object-oriented analysis is probably different from case to case. It might be a little bit treacherous to generalize and argue that there always exist requirements before object-oriented analysis.

- Object-oriented analysis has no smooth transition to design (Höydalsvik & Sindre, 1993). Korson & McGregor (1990), however, are of the opposite opinion and claim that the information in the analysis phase of the information systems development life cycle becomes an integrated part of the design.
- There are differences in the representation of objects in object-oriented analysis and object-oriented design (Kaindl, 1999); object-oriented analysis objects represent different things than object-oriented design objects. When carrying out object-oriented information systems development and working with analysis objects and design objects, it is probably true that the objects do not always fit very well together. One has to modify the object model (McGinnes, 1992). Whether this is a problem is an interesting question. Often one has to apply 'ad hoc' solutions when building different things like houses and machines, and software is probably no exception.
- Object-oriented analysis is sometimes slower than traditional analysis (Koskimies, 1997, p. 5). Perhaps object-oriented analysis is slower than traditional analysis because of the fountain like life cycle and the diffuse borderline with design. Perhaps parts of object-oriented analysis are actually designed already and the pure object-oriented analysis is in reality not much slower than traditional analysis.

The claim that the objects in the object-oriented model badly correspond to the objects in the real world is important. If this is true, much of the idea behind the object-oriented paradigm as a better way of modelling the real world can be questioned. But if the claim by Sommerville (1992, p. 66) that users can easily switch between different ways of looking at an information system, then the correspondence between objects in the real world and the objects in the object-oriented model seems less important.

In a study by Paetau (1995) activity based costing concepts were analysed in order to transform them into objects. This was, however, surprisingly difficult. Perhaps the question of how easy users find objects among business concepts is a question of what kind of information system is being developed and how experienced the user is in information systems development projects. Maring (1996) also reports that finding classes out of a general business domain is not always so easy. Maring (1996) goes on by saying that there might be different possibilities to model the problem domain into classes, and there is probably not any 'right model'.

The claim that object-oriented analysis, design and programming are more natural is also supported by Taylor (1990, pp. 29-31) who compares the structures of living organisms with object-oriented structures, and proposes that objects are like cells. The claim that a cell is natural is rather obvious, just as when a zoologist is looking at an animal like a lion, the lion is very natural. However, when a user like a financial dealer is looking at a financial process based on a formula for calculating an interest rate, the financial process might not be considered especially 'natural'.

Because object-oriented analysis is a software development activity, and not considered a problem of the object-oriented paradigm in this study, there will be no connections where object-oriented analysis is involved.

3.4.11 Problems with object-oriented design

The distinction between object-oriented analysis and object-oriented design is not always very clear. Many of the problems with object-oriented analysis can thus be recognized in object-oriented design as well. However, there are some specific problems with object-oriented design (quotation) reported by Pang (1996):

> - In modelling a complex information system, the problem domain involves too many objects interacting with each other in a complex way.
> - The problem domain of the applications is not well defined at the early stage of development. Coupled with constant changes in requirements, it is rather difficult to develop a proper object model.
> - Reuse is generally not addressed in the modelling and design phase.

Kaindl (1999) argues that there are few descriptions or rules of how object-oriented design objects should be defined. In fact object-oriented analysis objects and object-oriented design objects are often defined in the same way (Kaindl, 1999). Monroe et al. (1997) propose that there are limitations with object-oriented design because it is difficult to specify how groups of objects interact, and it is difficult to specify and package related collections of objects for future reuse. However, design patterns can solve the limitations (Monroe et al., 1997). In 1993 Wilkie (1993, p. 96) presented some shortcomings of object-oriented design; difficulties in identifying classes, blurred boundaries between design and both analysis and implementation, difficulties to find good CASE tools and elaborate and complex notations.

Kaindl (1999) proposes that there is often also another problem with design objects; they are both abstractions of something in the problem domain as well as an objects in the solution space It has also been reported that object-oriented design is slower than traditional design (Koskimies, 1997, p. 5).

Analysis, summary and discussion. Object-oriented design is a software development activity and cannot be considered as a problem. However, object-oriented design is classified as a problem in object-oriented information systems development, because object-oriented design is considered problematic by several researchers.

When presenting problems with object-oriented design, one has to remember that in object-oriented information system development analysis and design are strongly connected to each other, and that it might be difficult to draw a line between them. Therefore the problems with object-oriented analysis are probably found to some extent in the design phase too. In the design phase questions like “how shall this be implemented?” are common which distinguish design from analysis, because in analysis the questions are more like “what are we going to do?”

The following problems connected with object-oriented design were found (summary):

- Difficulties in finding good CASE tools that support object-oriented design (Wilkie, 1993, p. 96). This is a question of how mature the object-oriented

paradigm is. Nowadays (2005) the situation is better and several CASE tools are available.

- In object-oriented design it is difficult to specify how groups of objects interact (Monroe et al., 1997). This is an important issue because if the interactions between objects become indistinct, then the system becomes more complex and the understandability of the system suffers.
- In object-oriented design it is difficult to identify classes (Wilkie, 1993, p. 96). Although it is rather easy to identify classes when working with machines, customers, invoices and products, etc., it is more difficult to identify classes when working with more abstract things like relations and interests.
- In object-oriented design it is difficult to specify package related collections of objects for reuse in the future (Monroe et al., 1997). The activity of packaging collections of objects is connected to frameworks, and is not only a design issue.
- In most object-oriented design there are elaborate and complex notations (Wilkie, 1993, p. 96).
- Object-oriented design is slower than traditional functional design (Koskimies, 1997, p. 5). Perhaps this is because of the fountain like life cycle and the disseminated border with analysis. Perhaps parts of object-oriented design are actually analysis, so the pure object-oriented design is in actuality not much slower than traditional design.
- There are blurred boundaries between object-oriented analysis, object-oriented design and object-oriented implementation (Wilkie, 1993, p. 96). This problem is perhaps only a problem if projects are to be divided into milestones and then measured by project managers.

Because object-oriented analysis and object-oriented design are so closely connected, and the connection is influenced a great deal by the life cycle model that is followed, one has to probably study problems with object-oriented analysis when studying difficulties with object-oriented design, and vice versa.

3.4.12 Lack of object- oriented databases and common interfaces

Objects have to be stored somewhere because persistent objects are a necessity when performing object-oriented information systems development. Objects can be stored in files, relational databases, object databases and object/relational databases, etc. (Ambler, 1998, p. 341). Databases are mostly used.

Khoshafian & Abnous (1995, p. 24) present the following six approaches (quotation) for incorporation of the object-oriented paradigm in databases:

1. Use a novel database data model/data language approach.
2. Extend an existing database language with object-oriented capabilities.

3. Extend an existing object-oriented programming language with database capabilities.
4. Provide extendable object-oriented database management system libraries.
5. Embed object-oriented database language constructs within host language.
6. Use application-specific products with underlying object-oriented databases.

The unavailability of adequate object-oriented database systems is, however, a problem (Johnson, 2000). It is mostly due to a lack of an industry standard and a solid theoretical basis (Johnson et al., 1999). Because of the lack of appropriate object-oriented databases, relational databases are often used in object-oriented systems. Nevertheless, the use of a relational database within an object-oriented programming language often becomes problematic. The resulting information system is perhaps not optimal if object-oriented databases cannot be used, because theoretically one can achieve functionality with object-oriented databases that one cannot achieve with relational databases. One example of such functionality is the possibility to build intelligence in the form of methods in object-oriented databases. (Martin & Odell, 1992, p. 35) When working with complicated information structures object-oriented databases are also typically faster than traditional relational databases (Räisänen, 1997a, p. 15).

When using relational databases in object-oriented systems one common solution is to map a class with a table. In other words, each business class, like a security or a portfolio (Staringer, 1994), has a one-to-one correspondence with a relational table. In the table a row corresponds to an object. Usually this solution cannot be used if there are active objects in the application, because the active object has a more complex structure than a row in a relational table. Special arrangements are then needed as presented by Davis & Morgan (1993). Other solutions are based on using OBCD specifications. The Open Database Connectivity (ODBC) is an interface that several manufacturers like Microsoft supports that allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data (North, 1997). However, the solutions that are based on OBCD specifications are probably not possible because the database usually works differently with different operating systems. Because of this difficulty, not all database functions are in the OBCD and therefore one has to find another solution. Factory classes can be mentioned as such a solution (Rofrano, 1999).

Special solutions for combining the object-oriented paradigm with relational databases like the Strix object persistence engine can also be used (Perez, 2001). Reinwald et al. (1996) present another approach of combining relational databases and the object-oriented paradigm. In this approach an RDBMS extender called SMRC is used. SMRC provides the ability to store objects created in object-oriented programming languages like C++ into a relational database. One can read about the problems with storing objects in relational databases in the article by Reinwald et al. (1996), where problems with normalisation for objects, encapsulation and relational databases, inheritance and relational databases, etc. are considered.

Lauesen (1998) presents a solution where database wrappers are used to connect traditional databases with the object-oriented paradigm. Database wrappers mirror the traditional database and receive data from the traditional database. The database wrappers then write modified data back to the database. Database wrappers serve as well suited buffers for fast updating of screen objects. However, database wrappers are not pure objects, they are degenerate objects, which means that they are object-oriented objects that have been modified. (Lauesen, 1998)

When working with characteristic, pure, object-oriented databases everything is encapsulated. Therefore, ad hoc queries through a common interface like SQLCI cannot be made. This is a significant problem in many business applications (Miah, 1997; Ooil, 2002). The solution would be to develop predefined queries when using pure object-oriented databases (Miah, 1997). SQLCI is a term that is used in Non-Stop SQL on Tandem mainframes.

There are of course several object-oriented databases on the market. Wilkie (1993, p. 5) presents the following: Ontos, GemStone (used for example, in the HELIOS program, (Jean, 1992)), Objectivity, ObjectStore, Versant and O_2. Khoshafian & Abnous (1995, p. 23) present the object-oriented databases ObjectStore from ObjectDesign, Inc., OBD-II from Fujitsu, Objectivity / DB from Objectivity, POET and Itasca from Itasca Inc.

Object-oriented databases have advantages and disadvantages compared with relational databases. Wilkie (1993, pp. 248-249) proposes the following advantages (quotation):

- Potentially better performance than relational technology through the use of object IDs.
- Improved maintenance through the use of object-oriented techniques – conventional DBMS tend to offer very limited facilities for the expansion or modification of existing data structures because of the loose coupling between the database schema and the application programs. The tight coupling between applications and data in the object-oriented model offers considerably more scope for schema evolution through the extension and refinement of existing data structures and the effective use of application code through inheritance.
- More powerful modelling capabilities through the use of inheritance and user-defined types. The ability to store more semantic information within the database using abstract data types and unique identifiers. Also the ability to represent many-to-many relationships.
- A single language interface removes the problems of impedance mismatch associated with embedding SQL in a 3GL with a relational DBMS. This eliminates many of the inefficiencies, which occur in translating from one language to another.
- Applicability to environments in which relational technology is not suitable, such as computer-aided design (CAD), computer-aided software engineering (CASE), geographical information systems and office information systems (OIS).

Because of the potential advantages of using object-oriented databases, it would of course be a pity if an object-oriented database were desired but could not be found. Note also that the advantages presented above can naturally be discussed. For example, Bruegge & Dutoit (2000, p. 205) are of a different opinion to Wilkie (1993, pp. 248-249) regarding the performance of object-oriented databases, and argue that object-oriented databases are usually slower than relational databases for typical queries.

Analysis, summary and discussion. When building object-oriented information systems the persistence issue has always to be considered. Usually a database of some kind is used. When building object-oriented information systems an object-oriented database would usually be the most natural choice. However, there is still a lack of *tested* and *accepted* object-oriented databases on the market. Though Graham (2001, p. 231) proposes that object-oriented databases are presently in everyday commercial use, they are usually applied to applications where complex objects predominate, such as web servers, multimedia databases, geographical information systems and CAD/CAM systems.

One can always use a relational database when there is no suitable object-oriented database available. Different solutions on how to combine the object-oriented paradigm and relational databases are available.

The following possible associations between problems have been identified:

- The object-oriented paradigm is still IMMATURE in some areas (like databases), which results in -> A LACK OF OBJECT-ORIENTED DATABASES.
- The OBJECT-ORIENTED paradigm has resulted in few object-oriented databases and these object-oriented databases have a -> LACK OF COMMON INTERFACES for ad hoc queries.

3.4.13 Discussion of the problems with object-oriented paradigm in general

There are several problems with the object-oriented paradigm and some of them are quite obvious. Nevertheless, if these problems were solved the object-oriented paradigm could give a lot of new valuable strength to information system development. Different information systems development paradigms can be compared but it is difficult to conclude if one information systems development paradigm like the object-oriented paradigm is superior to others like the traditional functional paradigm. Hatton (1998) proposes that the object-oriented paradigm is a *new* paradigm but not necessarily a *better* one.

The finding from the *Survey of Advanced Technology 1996* (Pickering, 1996, p. 6-2) that states that using the object-oriented paradigm is complex is rather interesting. Are object technologies really so difficult? If they are, how can anyone use complex techniques to develop complex information systems? According to Booch (1994, pp. 3-25) the object-oriented paradigm is well suited for developing complex information systems.

The claim by Booch (1994, p. 289) that starting costs are often huge when one starts to develop a completely new object-oriented information system because there is nothing that can be reused and everything has to be developed from scratch is also interesting. This could be true. However, the experiences of the large and complex object-oriented information systems development project reported by Berg et al. (1995) were that the initial development costs amounted to less than if traditional methods had been used.

One also has to remember that the object-oriented paradigm matures from year to year; many problems have already disappeared and others will probably fade away in the future. In the study by Johnson (2000) the information systems developers did not recognize any real problems with the object-oriented paradigm.

4 EMPIRICAL STUDY

4.1 Introduction

When doing an empirical study there are several different research methods that can be used. Sometimes quantitative methods are the best choice for the research process in question, other times qualitative methods are better suited (Gummesson, 1991, pp. 2-3). It is also possible to use both quantitative and qualitative methods in the same research (Alasuutari, 1994, p. 23).

In information system research qualitative research methods are useful where information systems are studied in a *natural setting*. Quantitative research methods are useful when an area is *scanned* (Benbasat et al., 1987). In case studies the researcher can ask 'how' and 'why' questions and thereby gain some understanding of information systems and processes associated with them. Case studies in the information system area often concern questions regarding implementation and its success or failure. (Benbasat et al., 1987) In case studies the researcher has little control over the events and the focus is on contemporary phenomena within some real-life setting (Yin, 1994, p. 1).

4.2 Research method and research design

The research design is the sequence of events between the initial research questions and the eventual findings (Yin, 1994, p. 19). The sequence of events could include the research domains, asking meaningful research questions and using adequate research methodologies to address the research questions (Nunamaker et al., 1991). In this section both the research method and the research design will be presented.

After considering the different approaches presented in chapter 1 of this study, *the overall empirical research design and research method will be the evaluation research method with a combination of a survey and case studies*.

Surveys are a popular research method among many information systems researchers because they are easy to administer, easy to score, easy to code, allow the researcher to determine the values and relations of variables and constructs, provide responses that sometimes can be generalised, can be reused and therefore provide an objective way of comparing responses over different groups, times and places, help confirm and quantify the findings of qualitative research, can be used to predict behaviour and permit theoretical propositions to be tested in an objective fashion (Newsted et al., 1998). Surveys in this research were, however, chosen for other reasons, the main reason was that surveys are appropriate for scanning the market in order to get a general picture of the experienced benefits and problems with the object-oriented paradigm in Finnish software companies.

The author of this study concluded that case studies are especially useful when one wants to study experiences of benefits and problems of the object-oriented paradigm in

Finnish software companies. This is because case studies are suitable when one searches for some *understanding* of the benefits and problems being studied (Gable, 1994).

When two or more methods in social sciences are used for the same research problem in order to increase the reliability of the results, this is called triangulation (Gummesson, 1991, pp. 121-122). Triangulation and the combination of qualitative with quantitative evidence are recommended and discussed by Gable (1994), Jick (1979), Kaplan & Duchon (1988), Eisenhardt (1989, p. 538) and Yin (1994, pp. 90-94). Combining quantitative and qualitative research methods provides a richer contextual basis for validating and interpreting results and can lead to new insights and modes of analysis for the researcher while introducing testability and context to the research. Moreover, a more complete understanding of the phenomena being studied is achieved if different research methods are used. (Gable, 1994; Kaplan & Duchon, 1988)

Gummesson (1991, p. 122) gives an example of how a statistical quantitative survey can be supplemented by interviews. If the interviews are of a qualitative nature there is a combination of a quantitative and a qualitative method. This approach is often beneficial and frequently used (Alasuutari, 1994, p 23) especially for strengthening statistical results, for validation of results, for interpretation of statistical relationships and for clarification of puzzling findings (Jick, 1979). Mixing of methods utilises the strengths of the different methods (Jick, 1979). When computer systems are studied, it is often important to consider the cultural environment, social interaction and negotiation that could affect the outcome of the study and the phenomena being studied; qualitative research approaches are therefore often needed (Kaplan & Duchon, 1988).

Analysing the results of triangulation research might be difficult; the researcher is left to search for a logical pattern in the results of a mixed-method approach (Jick, 1979). In addition, Kaplan & Duchon (1988) experienced some difficulties and frustrations when carrying out research using quantitative survey and qualitative interview methods, especially in the analysing phase of the research. The triangulation approach also has some other problems, replication is difficult, the focus of the research and the research problems must be adequate and triangulation should not be used to legitimate a method that is preferred by the researcher (Jick, 1979).

Moreover, in a case study several different research approaches like observation, documentation, interviews and physical artefacts can also be combined (Gable 1994; Yin, 1994, pp. 79-94). If qualitative and quantitative research approaches are combined, they are usually combined so that first there is a qualitative pilot study and then there is a quantitative main study. This is because qualitative studies are often a good base from which hypotheses can be formulated. (Alasuutari, 1994, p. 203)

In this study, however, the survey was carried out before the case studies, mostly because it would probably produce information that could be taken into consideration when doing the interviews. Furthermore, the survey would also give some clues as to which types of companies would be most suitable for case studies.

The research design for the empirical study based on the evaluation study research method including a survey and case studies was the following:

1. Select the population. Find and select all Finnish software companies with more than four employees.
2. Send a letter by ordinary mail to all the companies in the population. Make a survey of the total population.
3. Expect a 15 % response rate.
4. Carry out the statistical analysis of the questionnaires.
5. Do the case studies with some of the companies. Difficult access might threaten this step.
6. Carry out the analysis of the case studies.
7. Compare the results of the survey with the results of the case studies and utilise the strengths of triangulation.
8. Write the conclusions and findings.

The response rate was considered the main problem with the survey; the pilot study in this study had this problem. There are of course problems with doing case studies after the survey; case studies cannot contribute to the model building exercise and to generating hypotheses for the survey (Gable, 1994). In this study, however, no hypotheses are generated and consequently this approach was not a problem. Note that previous studies revealed several suggestions and findings regarding the benefits and problems of the object-oriented paradigm. The research questions in this study were based on these and therefore formed the base for the questions in the survey and in the case studies.

4.3 Research questions

The research questions are the basic questions for the survey and the case studies. They are why-questions and other questions that have resulted from the investigation into previous studies in the field. Especially issues that are commonly known, but have not yet been scientifically studied are issues that can often be developed into research questions.

When formulating research questions some things have to be considered. The research questions ought to have both substance and form (Yin, 1994, p. 7). The questions cannot be trivial, as trivial questions are usually of no particular interest. The research questions also have to be questions to which there must be an answer otherwise they have to be rejected (Alasuutari, 1994, p 189).

Research questions are defined after previous studies have been considered. This means that many questions have a theoretical foundation and presumption of the answer, which is based on the findings from previous studies. The presumption can also be seen as a justification for selecting the research question. Some research questions could therefore be considered as hypotheses, instead of research questions. However, there is

a difference; research questions are often considered as why-questions, whereas hypotheses are in reality the answers to those why-questions, the validity of which are tested against the empirical material (Alasuutari, 1994, pp. 188-189).

The theoretical foundation is actually tested empirically, although it is only tested through one presumption, which is the presumption based on previous studies. It would be more rigorous to test the theoretical foundation through several presumptions (Lee, 1989). The presumption (and theoretical foundation) for the research questions is, however, rather comprehensive for most research questions in this study.

In this study the research questions are based on the research problems. Because the answers from the survey and the case studies are based on experiences and subjective opinions hypotheses cannot be developed. This is the case because hypotheses are based on answers that can be transformed into fixed figures or facts, and such answers are not obtained in this study.

When a research question has a presumption of the answer the researcher has to be very careful. The researcher has to avoid a "besserwisser" attitude where the truth (that the researcher thinks he or she knows, because of the presumption of the answers) is compared with the answers from the people participating in the (case-) studies (Alasuutari, 1996). The presumptions of the answers can of course be compared with the answers of the people studied, but the answers of the people studied can also be the 'truth'. In this study the subjective opinions of some people working in Finnish software companies are studied and compared with assertions found in previous studies. Whether the assertions or the subjective opinions are the truth is not analysed in this study.

When working out the research questions, some interesting research questions for object-oriented analysis and design have furthermore been developed. Object-oriented analysis and object-oriented design are presented in this study because these areas are important in object-oriented information systems development, they are claimed to be more powerful but also inferior to traditional analysis and design and can be seen as important parts of the object-oriented paradigm.

The research questions have also been analysed in order to become suitable questions in the survey or the case studies or in both. All research questions are included in the case studies, but not in the survey.

The research questions are not listed in this section as to prevent repetition. They are, however, all presented in the section on the analysis of theory and empirical findings.

4.4 Pilot study

A pilot study was made in co-operation with Christine Charpentier, who was an undergraduate student writing her master's thesis on the use of the object-oriented paradigm and software development methods, especially in the analysis phase (Charpentier, 2000). The author of this study was the supervisor of this work. The survey was carried out in such a way that it could also be used as a pilot study for this dissertation; part C of the survey on object-oriented projects was developed with this

study in mind. The aim with the pilot study was to get a general picture of the usage of the object-oriented paradigm, some experienced benefits and problems (the terms success and failure were used in the questionnaire) and some reasons why companies do or do not use the object-oriented paradigm.

Charpentier sent a questionnaire to 132 companies in the information systems development business. The population consisted of all Finnish information system development companies with more than 20 employees found in the database of Statistics Finland. The response rate was 15.2 %; 20 questionnaires were returned.

The questionnaire was divided into three parts:

- The company, part A.
- Software development projects, part B.
- Object-oriented projects, part C.

The third part was interesting for the purposes of this study. The questionnaire consisted of four pages with a total of 27 questions. Both open and scientific study based questions were used. The questionnaire was designed so that the respondent could answer it in a very short time. The questionnaire was sent on March 10, 2000 and the last answers were received on April 17, 2000. The questionnaire is found in the Appendix 1.

The results of the survey were the following:

The companies that answered the questionnaire were in the following fields of business:

Information systems, maintenance, e-business	6 companies
Trade, industry	9 companies
Insurance	6 companies
Public sector	2 companies
Other	3 companies

Some of the companies were in several businesses and therefore the total sum is more than 20.

Over 50% of the companies had a total sale of more than 8.3 billion Euros and most companies in the survey had 101-150 employees. The smallest company had 3 information systems developers, and the largest company had 217.

In the second part of the survey questionnaire the questions were about software projects. Interesting was the finding that 95% of the companies used some kind of information system development method. In addition, worthy of note was that only 35% of the companies used an object-oriented software development method and that 50% of the companies used two software development methods. The results also indicated that the object-oriented paradigm was mostly used in large projects and that most companies which used it had done so for only 1-2 years.

In the third part (part C) of the questionnaire the questions about the object-oriented paradigm were presented. When the respondent was asked to estimate the benefits of the object-oriented paradigm on a scale from 1 to 5 the average was 3.4. The following question was “Is the object-oriented paradigm your most important technique?” The results of this question were:

Now	5%
In one year	15%
Later	40%
Never	15%
No answer	30%

From these results one can estimate that the object-oriented paradigm will be more important in the future. One company answered both ‘later’ and ‘never’.

The three last questions in the questionnaire were open with no ready options to choose from. The questions and results were the following:

Question: What are the reasons for the success or failure of the object-oriented paradigm?

Answers:

- Know-how.
- No year 2000 support.
- The benefits of object directories become unused in commercial applications.
- Depends on the customer.
- The benefits of reuse are only found later.
- It takes a long time to learn the object-oriented paradigm.
- Training.
- Experience.
- The same as in traditional models.
- Difficult to say if it has been successful or unsuccessful in such an early phase.

Question: The most important reasons for using the object-oriented paradigm?

Answers:

- Requirements of customers.
- The integration of the PC with the mainframe.
- A part of the used document management program.
- Reuse.
- Standard solutions.
- Supports the development tools.
- Is a good part of the technology of today.
- The component architecture.
- Encapsulation of functions.
- Reuse.

- Learning.
- We see a remarkable future for object-oriented models and tools.

Question: Why is the object-oriented paradigm not used in all projects?

Answers:

- Requirements of customers.
- The construction model is based on components.
- Not needed.
- Not enough benefits/efficiency in the object-oriented paradigm.
- Difficult to implement the entire object-oriented paradigm, however, object-oriented effects are used.
- Difficult to say.
- Difficult to put object-oriented information systems development apart from development *based* on the object-oriented paradigm.
- Does not fit in all surroundings.
- We do not program that much.
- We do projects with traditional tools, however, we use the object-oriented paradigm in newer projects.
- Most of the development is on the mainframe, many projects are considered with existing systems.
- Enterprise resource planning.
- The time of objects is forthcoming.

When the results of the last three questions are examined, one can see that in 2000 the companies were not using the object-oriented paradigm as much as expected. Interesting was that reuse as a benefit was mentioned twice. Other benefits were know-how, support of development tools and encapsulation of functions, etc. Among the problems, training and learning the object-oriented paradigm were presented. Other problems mentioned were no year 2000 support, not enough benefits / efficiency and does not fit in all surroundings, etc.

As a conclusion, one can say that the companies had experienced some of the benefits and problems that are presented in other studies. However, the companies had not experienced all the proposed benefits and problems, and the picture of the situation in Finland was somewhat unclear. All in all the fact that only 7 companies out of 20 that answered were actually using object-oriented information development methods, means that the results obtained can only be used for identifying questions for more elaborate studies carried out later. The need for a more comprehensive study of the benefits and problems was therefore accentuated.

4.5 Survey, planning of the survey and statistical issues

The empirical study started with a descriptive quantitative survey of Finnish software companies. The author of this study analysed the survey by using descriptive statistics where the experienced benefits and problems of the object-oriented paradigm among

Finnish software companies were presented. Descriptive statistics consist of procedures to summarize the information in a set or sample and to describe the characteristics of the set or sample (Mendenhall et al., 1993, p. 6). In this section the planning of the survey is presented. Blom (1984, p. 164) presents several stages in planning a statistical examination and these stages have been considered in this study.

When analysing the research questions it was found that they were all classified (in other words on the nominal scale) and therefore qualitative. On a nominal scale, numbers are simply used as labels for groups or classes (Aczel, 1999, p. 10).

If a random sample is used then the sample from the population has to be picked in the correct way so the sampling design is important. In descriptive statistics, a random sample is required, and any disturbing factors are not allowed to occur (Blom, 1984, p. 165). In the survey there was no random sample because the questionnaire was sent to all Finnish software companies (with a few exceptions as presented in the sub section on the selection of the population).

The questionnaire is based on the research problems and the research questions. The connection between research problems, research questions and questions in the questionnaire are as follows: first some broad-spectrum research problems have been stated, then some more detailed research questions have been developed. These research questions are based on the research problems. Then the research questions were modified and became questions for the questionnaire. These research questions have no other theoretical background than the previous studies.

When doing the survey the next step after selecting the sample is to collect data. The collection of data can be performed in several different ways and there are several methods available. Körner & Wahlgren (2002) present the data collection methods of mail survey, phone interview and personal interview. Gunn (2002) presents web-based methods. The methods are presented below:

- Mail survey. A cheap alternative with a high risk of people not responding. Lundahl & Skärvad (1999, p. 172) mention the following advantages; usable for questions with many possibilities for answering, no effect from the interviewer, utilizable for sensitive questions, pictures and other visual material can be used. There are of course also disadvantages such as: time consuming, no control over the answering process (people might intentionally fill in wrong answers), difficulties in following up (if anonymous) and not very suitable for open questions (Lundahl & Skärvad, 1999, p. 172). Further, there are practical problems like writing envelopes and checking addresses, etc.
- Phone interview. The researcher might affect the respondent, which is a problem. There is in other words an interview effect. Phone interviews might also be rather expensive. It might be difficult to find the right person to talk with and several new connections and recalls might be necessary. According to Gunn (2002) respondents also tend to agree with the interviewer because of his or her presence. Of course, the privacy of the respondent is very low or even non-existent in phone interviews.

- Personal interview. Probably the alternative that demands most work. These kinds of interviews are presented more thoroughly by Lundahl & Skärvad (1999, p. 172). Personal interviews are based on the interviewer personally visiting the respondent in the respondent's office. These kinds of interviews have several advantages; they can be conducted rather quickly, they can be useful when one has complicated questions, one can also use pictures and other visual equipment. The interviewer can pose follow-up questions and the interview process is as a whole controlled. Among the disadvantages with these kinds of interviews are the rather high costs, the possible interview effect, the difficulty in asking sensitive questions and the difficulties in finding respondents willing to participate. (Lundahl & Skärvad, 1999, p. 172)

- Web-based surveys. Web-based surveys are presented and discussed by Gunn (2002). They are not dealt with here because web-based surveys were considered unsuitable for this study at rather an early stage. The main reason was that the author of this study had no former experience in how to build web-based survey instruments on the web. Another reason was the problem with finding the appropriate way of informing the potential respondents of the existence of the web-based survey. E-mails or regular post could have been used. However, if regular post had been used, the difference to a mail survey would have been diminutive.

As mentioned earlier the survey in this study is based on a mail survey. At first phone interviews were considered but after a few phone calls it was clear that it would be very difficult to get into contact with the right persons for the survey because they are often away from the office and seldom have the time to answer a long questionnaire by phone without prior arrangement. By sending questionnaires by mail the letter would hopefully be handed over to the right person and it could then wait until the person in question had the time to answer the questionnaire.

With all data gathering methods the problem with persons not responding has to be solved. The problem is associated with the gaining of access. Often companies or presumptive respondents do not want to participate in surveys or give access to organisations. This is due to several reasons, which Saunders et al. (2000, p. 114) present as follows:

- The respondent or the organisation cannot see any value in participating, and participating is time consuming and labour intensive.
- The research topic is sensitive to the respondent or organisation or they are concerned with the confidentiality of the information that is asked for.
- The respondent or the organisation has perceptions about the credibility of the researcher or they have doubts about the competence of the researcher.

Lundahl & Skärvad (1999, p. 119) recommend that one should promise to give the respondent something in order to encourage them to participate; for example, one can offer to give a copy of the final study. If the researcher also appears competent, the chances of 'getting in' are increased (Lundahl & Skärvad, 1999, p. 119).

Even if the researcher acquires access into an organisation, he or she has to be aware of some dangers. The researcher has to select a representative sample of organisational participants and has to be aware of the danger that the participants might lie, misunderstand the question or tell things that are mistrustful (Saunders et al., 2000, p. 115; Undheim, 1985, p. 19). Saunders et al. (2000, p. 115) call this issue cognitive access. Buchanan et al. (1998) further discuss the problems of gaining access into organisations and companies in their comprehensive article on the topic.

When analysing the research questions and the possible answers it was found as mentioned earlier that they are based on qualitative variables. The qualitative variables were also all *non-ranked categorical variables* on a *descriptive* (also called nominal) scale. For qualitative variables, the measurement of frequency is usually shown in a table also called a frequency distribution. Bar charts and pie charts are often used for qualitative variables (Körner & Wahlgren, 2002). For variables with a large number of categories, one has to group the data into categories of interest (Saunders et al., 2000, p. 338).

For qualitative variables that are measured on the nominal scale, there are several statistical tests that can be used for examining relationships among categorical descriptive data. According to Saunders et al. (2000, p. 357) there are two main statistical tests for categorical descriptive data; the Chi-Square test that is used to test whether two variables are significantly associated, and the Kolmogorov-Smirnov test that is used to study whether the distribution of an observed set of values for each category of a variable differs significantly from a specified population. However, the researcher must always be aware that relationships are usually complex and they should therefore be managed with great care (Undheim, 1985, p. 28).

In this study only descriptive data is used and no statistical tests were considered necessary in order to answer the research questions.

Surveys also have problems that have to be considered. They often take only a snapshot of the situation at a certain time giving little insight into the background of the data (Gable, 1994). The response rate might also be low and the reliability of the answers can be suspicious because the respondents might misunderstand the questions (Gable, 1994). The questions of *reliability* and *validity* also have to be considered. Such questions are discussed in connection with the case studies in this dissertation where most of the issues presented are valid for surveys as well. Some additional questions for surveys regarding reliability and validity are, however, shortly presented below.

Reliability and surveys. In order to draw conclusions, the reliability of the study must be high. Reliability can be defined as the freedom from random influence from an instrument of measurement, independently of what measurements the instrument of measurement is used for (Rudberg, 1990, p. 129). Boudreau et al. (2001) define reliability as a declaration on measurement correctness, in other words (quoting): ‘The extent to which an instrument produces consistent or error-free results’.

Reliability is concerned with matters like the exactitude among the observations and the sample, the accuracy of the figures that are based on the observations and the sample,

the registration, the usage and the treatment of the observations, etc. (Undheim, 1985, p. 18). High reliability is a presupposition but no guarantee for high validity (Rudberg, 1990, p. 133).

Validity and surveys. There are several definitions of validity. Boudreau et al. (2001) present *content* validity and *construct* validity. In order to achieve *content* validity the observations or the sample has to represent the issues that the researcher is working with (Undheim, 1985, p. 20). *Construct* validity is defined as the trustworthiness that an instrument is measuring the phenomenon it ought to measure (Rudberg, 1990, p. 130).

Validity can be further divided into several parts, of which the two following, according to Rudberg (1990, p. 131), are the most important for statistical studies:

- Contemporaneous validity, the reliability that an instrument (like a statistical test) can make diagnoses or specifically tell how something is now.
- Prognostic validity, the reliability that an instrument (like a statistical test) can make prognoses or expressly tell how something will be in future.

As a summary one can say that the higher the extent to which an instrument is measuring the item it is supposed to measure, the higher the validity is (Rudberg, 1990, p. 131). Note also that validity in a statistical test is expressed as a correlation coefficient that can be between 0 and +1 (Rudberg, 1990, p. 132). The validity correlation coefficient has of course to be recognized when administering statistical tests.

4.5.1 Selection of questions for the survey

When selecting questions for the survey out of the research questions, the author of this study tried to select questions that were important and suitable for the survey. All the 57 research questions are presented in Section 4.7. Some of the research questions are very difficult to study in a survey and therefore are only used in the case studies.

When planning a survey it is important to consider the statistical issues before the survey is conducted. In this study all the research questions were analysed and the following grouping was made:

1. The following research questions will not be included in the survey but only in the case studies:

   (Q5), (Q7), (Q9), (Q12), (Q13), (Q15), (Q16), (Q17), (Q18), (Q19), (Q22), (Q23), (Q24), (Q25), (Q26), (Q27), (Q28), (Q30), (Q38), (Q39), (Q42), (Q45), (Q48), (Q49), (Q50), (Q51), (Q52), (Q55) and (Q56).

2. The statistical issue that the answer can be ‘yes’, ‘no’ or ‘not sure’, for the research questions:

   (Q1), (Q3), (Q4), (Q6), (Q8), (Q10), (Q11), (Q14), (Q20), (Q21), (Q29), (Q32), (Q33), (Q34), (Q36), (Q37), (Q40), (Q41), (Q43), (Q44), (Q46), (Q47) and (Q53).

Data is qualitative, categorical, and descriptive. The analysis and presentation of this question is based on showing the proportion of occurrences of categories for one variable and therefore a pie chart is appropriate (Saunders et al., 2000, p. 339).

3. The statistical issue that data is qualitative, categorical and descriptive for the research questions:

   (Q2), (Q35), (Q36) and (Q54).

   The analysis and presentation of this question is based on showing the frequency of occurrences of categories for one variable so that the highest and lowest are clear; therefore, a bar chart is appropriate (Saunders et al., 2000, p. 339).

4. For the two open research questions (Q31) and (Q57) special statistical solutions will be used in the form of listing and grouping.

After excluding the research questions mentioned above the survey still consisted of 25 questions. The questionnaire for the survey is presented in Appendix 2.

4.5.2 Selection of population and carrying out the survey

For the survey population all software companies in Finland were first considered. In January 2003 the author of this study received a list from Statistics Finland with information regarding the number of software companies in Finland in different size categories. This information is presented in Table 2 and Table 3:

Table 2: The number of Finnish software companies in different turnover categories

Turnover €	**Number**	**Turnover €**	**Number**
Unknown	515	1 000 000 – 1 999 999	163
No turnover	1	2 000 000 – 9 999 999	176
1 –199 999	2915	10 000000 – 19 000000	28
200 000 – 399 999	358	20 000 000 -	17
400 000 – 999 999	317		

Table 3: The number of Finnish software companies in different size categories

Number of employees	Number of companies
Unknown	636
1 – 4	3046
5 – 9	295
10 – 19	238
20 – 49	179
50 – 99	56
100 – 249	37
250 – 499	1
500 – 999	2
1000 -	0
Total number of software companies	4490

When one sets up size categories for companies, one can take into consideration several definitions of small, medium and large sized companies. For example, Riihimaa (2004, p. 1) proposes that in the US manufacturing branch enterprises with less than 500 employees are considered small or medium sized. In Japan the figure is 300 and in EU 250. Fayad et al. (2000) propose that companies that have fewer than 50 employees are small, which corresponds well with the definition from EU presented in Table 4.

From these tables one can conclude that as a general rule software companies in Finland are small, especially when one compares the table (Table 3) with the criteria of SMEs in the EU (Table 4) as presented by Bradford (2002):

Table 4: Criteria of the SMEs in the EU

Criterion	**Micro sized**	**Small sized**	**Medium sized**
Max number of employees	10	50	250
Max annual turnover	-	7 Million Euros	40 Million Euros
Max annual balance sheet total	-	5 Million Euros	27 Million Euros
Max % owned by one, or jointly by several enterprise(s) not satisfying the same criteria	-	25 %	25%

As many as 3046 + 295 = 3341 of the Finnish software companies on the list had less than 10 employees and are classified as micro sized by the EU. The large number of micro sized Finnish software companies was reduced by the author of this study by only selecting those companies from the list with five or more employees; the subjective reasoning behind this decision is as follows:

- Very small Finnish software companies with 1-4 employees probably have limited experience of object-oriented information systems development.

- They are unlikely to answer the survey.
- They are numerous and would make the selected population too large to manage in an efficient way.

The population of Finnish software companies for the survey in this study thus became 295+238+179+56+37+1+2 = 808 software companies. Note that software companies with an unknown number of employees were excluded from the selected population.

The information regarding name, address and phone number for all Finnish software companies was obtained from Statistics Finland in the beginning of February 2003. The final population amounted to 799 Finnish software companies (9 software companies had gone out of business).

As many as 404 (50%) of the Finnish software companies in the selected population were from the greater Helsinki area (Helsinki, Espoo, Vantaa, Kauniainen, Klaukkala, Kerava, Tuusula and Järvenpää); the rest were located in other parts of Finland (85 from Tampere, 54 from Oulu and 42 from Turku).

For the survey a questionnaire was developed and based on the research questions in this study. The questionnaire was first developed and written in English and after that was translated into Finnish. The questionnaire for the survey in Finnish is presented in Appendix 3.

When the questionnaire was complete, it consisted of 16 pages and 25 questions. It was posted in a first class envelope with the name and logo of the Swedish School of Economics and Business Administration.

When some of the homepages of the selected Finnish software companies were studied, some interesting things were found:

- Some were obviously not carrying out software development; the companies were importers of software from abroad and consulting companies, etc.
- Some of the companies had gone out of business.

Because the actual number of software companies involved in software development was smaller (due to the reasons mentioned above) than expected, the author of this dissertation decided to do a study of all Finnish software companies in the selected population, in practice this meant all (see the exception above) Finnish software companies on the list from Statistics Finland that were still in business.

On April 25, 2003, the questionnaire with a cover letter was first sent to 100 software companies randomly picked from the list from Statistics Finland. The 100 software companies came from the Helsinki area and from the cities of Jyväskylä, Pori and Tampere. By searching through the homepages of the 100 selected software companies the most promising person to answer the questionnaire was selected (usually a production manager or a software development manager). When this person was found his or her name was used on the envelope. In other cases, the title ‘application development manager’ was used.

An email was also sent to all the persons who had an email address and to whom the questionnaire was directly addressed. In the email, the author of this study wrote that 'an important questionnaire' will soon arrive by regular post and that by participating in the survey one is supporting Finnish information systems research. It was also mentioned that one could participate in the survey anonymously.

Then the first 100 questionnaires were posted with the address hand written. After a few days, the author of this study received some emails from the persons who had been emailed, with rather positive remarks like 'nice questionnaire', 'nice that one can answer the questionnaire anonymously', and 'can I have the questionnaire in English?' etc. One respondent even asked for a copy of the questionnaire by email because he claimed that no questionnaire had come with the ordinary mail.

Four days after the questionnaire had been posted answers started to arrive. After one month 25 answers had been received, a response rate of 25%. In these 25 answers 20 had used the object-oriented paradigm and 5 had not. All 25 answers were well written and had to be considered relevant. On May 22, a reminder email was sent to all the persons that had an email address and to whom the first 100 questionnaires had been directly addressed. One answer by email came from a person who wrote that they are not involved in software development in the company in question and therefore they are not participating in the survey. Two further answers came later resulting in a final tally of 27.

On May 23, 2003 a new set of 200 questionnaires were posted, however, this time no homepages were checked. The 200 software companies had been selected from the population and list from Statistics Finland so that they were from the south, the east, the west and the north of Finland (but not from the central parts). The addresses were again all written by hand and the letter was addressed to the production manager. The production manager was chosen because among the 25 answers in the first set the title 'production manager' was the most frequent. The remaining 499 companies in the population and on the list from Statistics Finland were left to the third set.

On September 12, the third set of 488 questionnaires was posted and again no homepages were checked. The 488 software companies were the rest of the software companies in the population and on the list from Statistics Finland. Eleven software companies were left out because they were suspicious (exactly the same address as another company or odd address, etc.) or they had unquestionably gone out of business. The letters were addressed to the production manager.

A total of 130 answers were received from all three sets together, of these:

- 104 were valid responses.
- 24 were received as 'return to sender' presumably due to the following reasons; inaccuracies in the mailing list, companies had gone bankrupt or companies had moved, etc. Some information regarding this issue was obtained from the Finnish Post who returned the mail.
- 2 surveys received were incomplete.

Thus, a total of 104 valid answers were obtained. The 104 valid surveys out of 788 surveys sent, reflected a valid response rate of 13,2%. This was considered sufficient for this study because of the large number of answers (104) and because the questionnaire had been sent to all valid companies in the population (total survey).

The general quality of the answers received further increased the satisfactoriness of the responses. This quality was reflected in a review of the job titles of the respondents, which suggested that over 70% of the surveys were completed by managers, consultants and system analysts, etc. (Question IV).

One can argue that the validity of the survey was appropriate because Statistics Finland provided the population and the survey was made for all software companies in the population. The questions in the questionnaire were gathered from the review of previous studies and theory.

The reliability of the survey was adequate because the number of answers was sufficient and the quality of the answers was high.

Because the questionnaire was sent to all adequate software companies in the population and the number of answers was as high as 104, one can profess that one can generalise about all (more than 4 employees) Finnish software companies. No systematic drop out of software companies among the responses was found (for example, the questionnaires received after a remind message had been sent, were compared with the questionnaires received earlier), and one can argue that the software companies that participated in the survey most likely reflect a good sample of all (more than 4 employees) software companies in Finland involved in *software development*. However, one has to be aware of the danger that software companies involved in object-oriented information systems development were more willing to answer the survey than companies not involved.

As mentioned earlier, the author of this dissertation also studied the homepages of approximately 50 of the software companies in the population. Out of the studies on this randomly selected sample it was found that several of the companies classified as "software" companies by Statistics Finland were in fact not involved in software development. The companies imported software or were retail sellers of software produced by other companies. When considering this fact, one can conclude that the response rate among software companies actually involved in software development was actually higher than the total response rate.

The results cannot be generalized to populations outside Finland. However, the sample represented a wide variety of information systems developers. Respondents varied from those only slightly familiar with the object-oriented paradigm to those who were very experienced with it. There was a broad spectrum of jobs including executive chiefs, managers, analysts, programmers, consultants and even a 'share holder'.

Note further that the results obtained in the study were collected from:

- Information systems developers that had experience in conventional and object-oriented information systems development.

- Information systems developers in organisations of different size.

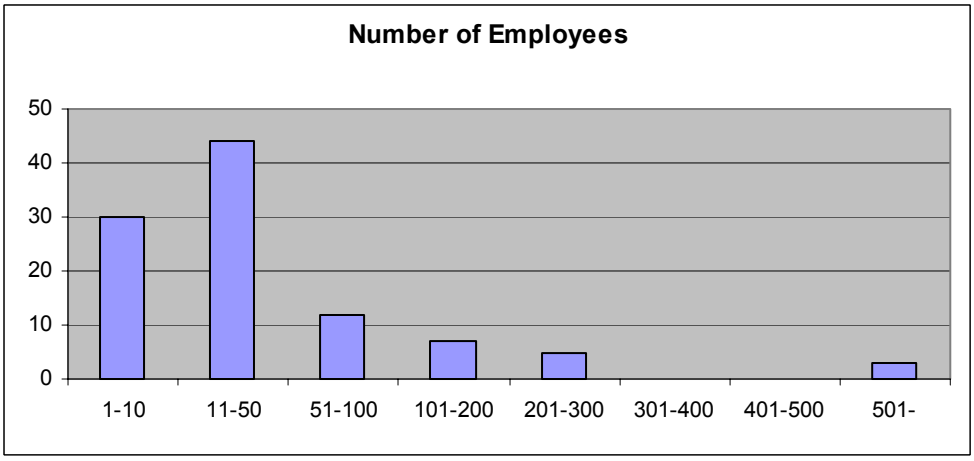
4.5.3 Survey results concerning the software companies

The results will be presented and analysed in Section 4.7. The questions in the questionnaire regarding the company are, however, presented here.

I. Approximate number of employees in your company:

The results are presented in Figure 8.

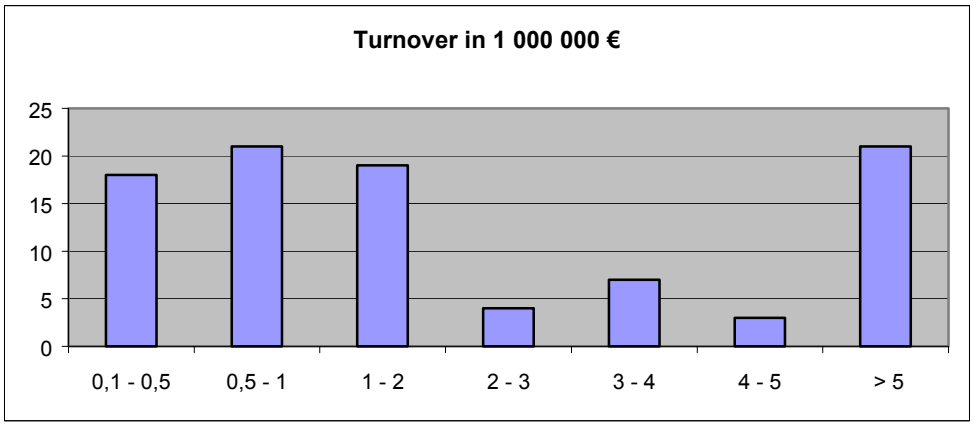
Figure 8: Number of employees



II. What is the approximate turnover of your company?

The results are presented in Figure 9.

Figure 9: Turnover of the companies in the survey



III. In what field are most of your clients?

The results are presented in Table 5.

Table 5: Main clientele in the survey

	Number of answers
Industry	15
Telecommunications	15
Several	13
Public Sector	7
Information Technology	4

There were additionally 9 fields with 3 answers and 9 fields with 2 answers and 30 fields with 1 answer.

IV. What is your position in the company?

The results are presented in Table 6. There were also 2 positions with 3 answers and 4 positions with 2 answers and 23 fields with 1 answer.

Table 6: Position of respondent in the survey

	Number of answers
Executive Chief	16
Production Manager	8
Software Development Manager	8
Technology Manager	7
Manager	5
Product Development Manager	5
Project Manager	5
CTO (Chief Technology Officer)	4
Program Analyst	4

4.6 Case study

After the survey was completed, the qualitative case study of some selected software companies in Finland started. The case study followed the steps that Yin (1994, p. 49) presents:

1. Develop Theory. During the review of previous studies several research questions were developed. These research questions are presented in Section 4.7.

2. Select Cases. Selecting cases has to be done carefully. The cases have to correspond to the population. Eisenhardt (1989) also recommends selecting cases that are extreme and that replicate or extend the emergent theory. Yin (1994, p.

46) proposes that cases must be selected so that the cases either predict similar results (in a literal replication) or produce contrary results but for predictable reasons in a theoretical replication. Lundahl & Skärvad (1999, p. 191) also discuss the selection of cases and present the ideas of selecting typical cases *and* special cases.

How many cases ought to be done in order to get the desired understanding and knowledge? According to Gummesson (1991, p. 85) the number of cases is determined by saturation (the diminishing marginal contribution of each additional case). When new cases give very little new knowledge or information there is no actual need to study any more. Lundahl & Skärvad (1999, p. 191) emphasize that the number of cases depends on the desired depth and width of the study. Eisenhardt (1989) also proposes that case studies can involve either single or multiple cases. A number between 4 and 10 cases usually works well when doing case studies for theory development (Eisenhardt, 1989). Yin (1994, p. 50) recommends a number of five, six or more cases. However, the number of cases is connected to the question of external validity. In an earlier similar study, seventeen method specialists and IS managers in eight Finnish companies were interviewed (Smolander et al., 1990). In this study six cases were selected.

3. Design Data Collection Protocol. The data collection protocol consists both of an instrument and of the rules and procedures that ought to be followed when using the instrument. The data collection protocol should have the sections that Yin (1994, pp. 63-74) presents. These sections were thoroughly considered when the author of this study developed the data collection protocol, which is presented in Appendix 4. When developing a data collection protocol a pilot case study can also be used (1994, pp. 74-77). In this study no pilot case study was performed. There is already a survey and a pilot study preceding the case studies, so doing a further pilot case study would not have added much more insight.

4. Conduct Case Studies. When conducting the case studies the answer to the research questions are of course sought, and some other matters regarding the experienced benefits and problems of the object-oriented paradigm in the software companies are examined in the open questions. When doing the case studies the chain of evidence has to be maintained so that one can follow the case studies from the early research questions to the final conclusions (Yin, 1994, pp. 98-99).

   The case studies consisted of interviews of one or two hours in length with either one or a couple of persons who were working with the object-oriented paradigm in information systems development. Of course, other techniques than interviews could have been used. The interview questions were open-ended in nature and the pre-planned interview questions were followed. The interview questions were the same as the research questions.

   The interviews were carried out in Finnish because it is important to consider the native-language of the person being interviewed. A tape recorder was used. The use of a tape recorder is discussed by Yin (1994, p. 86), who presents some circumstances when it should not be used. The interviews were transcribed from the tapes.

5-8. Write Individual Case Report from every Case Study, Draw Cross-Case Conclusions, Modify Theory and Develop Policy Implications.

After the case studies had been done and transcribed, the analysing and reporting phases started. Eisenhardt (1989) presents some analysis approaches and strategies. It is important, however, to have a general analytic strategy in the first place. Yin (1994, pp. 102-106) presents two general strategies: relying on theoretical propositions and developing a case description.

In this study there is a theoretical proposition: the previous studies. When considering different analysis techniques, the pattern-matching technique (Yin, 1994, pp. 106-119), seemed to be a good technique for this study. An empirical based pattern (the result of the case studies) is compared with a predicted one (the 'assumptions' from the review of previous studies). In this study, the cases are studied for observations that occur several times, in other words patterns. The observations are presented as observations and not as general rules or new theory.

Write Cross-Case Report. The final step is to compose the case study report. Important aspects and ethical topics to take into consideration when writing case study reports are presented and discussed by Altheide & Johnson (1994, pp. 491-492) and Yin (1994, p. 128).

When carrying out research and especially when doing case studies there are some key questions and problems that the researcher has to be aware of and remember. Qualitative studies like case studies have problems such as the inability to manipulate independent variables, the risk of defective interpretation, the lack of ability to randomise, the lack of controllability, the lack of possibilities for deduction, the lack of repeatability, the question of access, the question of generalisation, the question of reliability, the question of validity, the question of credibility, the question of which case study method to choose, the question of how to analyse the result from a case study and how to make a compound observation and the question of giving information about the case studies, etc. (Gable, 1994).

Access. When considering which Finnish software companies to study the question of *access* is important. The most interesting companies might not allow one to do case studies and interviews. There is also a possibility that the companies do not give correct information. Incorrect information can also be given in surveys of course. Alasuutari (1994, pp. 80-86) discusses the question of the truthfulness of the given information or honesty of the informant. Some practical advice on how to get in (getting access) the company is given by Buchanan et al. (1988).

Generalisation and case studies. According to Lee & Baskerville (2003) it is possible to make generalisations from empirical or theoretical statements and end up with empirical or theoretical statements in the generalisations. Yin (1994, pp. 35-36) discusses generalisation and external validity. He argues that from case studies analytical generalisation can be made where the researcher is attempting to generalise some results into some more common theory. There are, however, problems associated with making generalisations from individual case studies. If there is no need to make generalisations, there is also no problem (Alasuutari, 1994, p 207). In this study, no

generalisations were needed from the case studies because all generalisations are made from the survey.

Reliability and case studies. Reliability means that if another researcher would carry out the same research with the same methods, the results ought to be the same (Yin, 1994, p. 36). In order to make this possible the documentation of the case studies has to be made carefully, and the researcher has to remember that the case studies might be checked and read by several researchers later on (Yin, 1994, pp. 36-37).

Validity and case studies. Validity means that the evidence from the research really reflects the reality under examination (Gummesson, 1991, p. 159). According to Yin (1994, pp. 35-36) validity can be divided into internal validity and external validity. Internal validity is a problem only when doing explanatory case studies and when the researcher is trying to explain whether an event A led to an event B without knowing that some third factor C may actually have been involved (Yin 1994, p. 35). Internal validity is also a concern when making inferences (Yin 1994, p. 35). External validity is concerned with the problems of making generalisations from case studies.

Data collecting techniques used in the case studies. Several different techniques can be used when doing case studies. Case studies often consist of archives, questionnaires and observation, and the evidence can be in words (qualitative), or numbers (quantitative) or both (Eisenhardt, 1989). Yin (1994, pp. 79-90) presents the following case study techniques: documentation, archival records, interviews, direct observations, participant-observation and physical artefacts. In this study interviews were used. The other case study techniques did not seem appropriate with the exception of observations and perhaps documentation. These case study techniques and methods have already been succinctly discussed in the earlier sub section on selecting research design and research methods in this dissertation but are anyway briefly discussed below.

Documentation and archival records could be used because most information systems have some documentation that could be examined. This is a laborious procedure and the documentation is probably well protected and not available to the researcher. The design decisions and programming issues can also be difficult to understand for a person who has not been in the information systems development project. Observation will be used to some extent; if something interesting happens during the interviews, it will be written down. But systematic observations would probably give very little information because information system developers are mostly working with computers and one ought to see the screen and understand the context of the work in order to get some information from such an activity. Participant-observation and using physical artefacts are methods that are probably not suited for this study because information systems development is a complex activity and one cannot start doing it without adequate prior experience. Physical artefacts are hard to find because an information system is very intangible by nature. A software program could be considered a physical artefact but then the problem is that it might be difficult to analyse and understand the program if one is not familiar with the programming culture in the company or the programming language in question.

Result from case study and compound observation. What is important, however, is to remember that when doing a qualitative study and when working with raw observations obtained from, for example, case studies that all the raw observations have to be the same, or stand for the same thing, in order to join the raw observations to a compound observation (Alasuutari, 1994, p. 33). A single exception is enough to break the rule, and shows that one has to rethink the whole thing again. Often the level of abstraction is raised or the theoretical framework is changed in order to make compound observations (Alasuutari, 1994, p. 33). The theoretical framework is an explicitly defined view of the observations in question (Alasuutari, 1994, p. 69). After the raw observations have been compounded, the next step is to interpret the findings. This analysing phase means in qualitative research that based on the compound observations and other hints, we make an interpretation of the phenomena studied (Alasuutari, 1994, pp. 34-35).

When the companies for the case studies were selected, the largest software companies in Finland were considered. The largest software companies were selected because the author of this work thought that the there is a higher possibility that larger companies have experiences on object-oriented information systems development than that smaller companies have such. An email was sent to the 20 largest software companies and an answer was attained from nine companies. Six of the companies were willing to participate in an interview.

The case study protocol in Finnish is presented in Appendix 5.

The interviews were carried out as presented in Table 7:

Table 7: Interviews

Place	Date	Position	Time	Comments
Helsinki	8.12.2003	Development Manager	64 min.	Drew a picture
Helsinki	19.1.2004	Two Software Developers	78 min.	Very talkative
Helsinki	9.2.2004	Technology Manager	66 min.	
Vaasa	5.5.2004	Main Software Developer	40 min.	Integrated analysis and design with implementation; questions regarding analysis and design were omitted.
Espoo	18.6.2004	Manager	55 min.	Mainframes and PCs. Long time in business. Older software developers
Espoo	23.6.2004	Manager	65 min.	Business in Vietnam

The questions and answers can be found in the next chapter of this study. During the translation from Finnish to English great precision has been used. Still the answers were a bit modified when translated. Some words and sentences that the author of this study

found irrelevant were deleted in the answers. The mission was, however, to transcribe the interviews with constant great care.

4.7 Theoretical propositions and empirical findings

In this section the research questions will be analysed from a theoretical and an empirical view. The main concern is to compare the empirical findings with the theoretical statements found in the review of previous studies.

All companies (104) answered the first question (Q1) that concerned the use of the object-oriented paradigm in information systems development. As many as 89 companies out of 104 use the object-oriented paradigm, and the population N is 89 for the survey in the following research questions. If there are no answers to a survey question this is pointed out in the survey results.

When the possible associations between the benefits and the possible associations between the problems have been empirically checked, the following has been done:

1. First, all the respondents that have experienced the benefit or problem in question have been selected. The selected respondents become a population.

2. Out of the population, the respondents that have also experienced the connected benefit or problem have been selected.

Because the associations have been validated in this rather unpretentious way one can only consider them as hypotheses.

Note further that the one can discuss the direction of the associations. In this work the direction of the associations has been developed in the way the author of this study found most appropriate.

General

(Q1) Has the software company been using the object-oriented paradigm in information systems development?

Theory – Studies: According to Johnson et al. (1999) and Sircar et al. (2001) the adoption of object-oriented methodologies has progressed slowly. In a study by Glass (1999) that focussed on information systems managers, it was found that only 39% of the organisations had adopted the object-oriented paradigm in some form (Sircar et al. (2001). In addition, Zhang (1999, p. 66) found that many companies did not employ object-oriented information systems development.

Pilot study: In the pilot study it was found that 35% of the companies use an object-oriented information systems development method.

Survey: 85 % had been using the object-oriented paradigm.
1 % did not know.
14 % had not been using the object-oriented paradigm.

Case studies: All the companies used the object-oriented paradigm.

Discussion and conclusions: Since the time of the above-mentioned study in 1999 and the pilot study in 2000 a great majority of the companies have obviously started to work with the object-oriented paradigm. One can present a supposition that the object-oriented paradigm is nowadays a major information systems development paradigm.

(Q2) If the software company has not been using the object-oriented paradigm in information systems development, then why not?

Pilot study: The reasons for not using the object-oriented paradigm were the following:

- Requirements of customers.
- The construction model is based on components.
- Not needed.
- Not enough benefits/efficiency in the object-oriented paradigm.
- Difficult to implement the entire object-oriented paradigm, however, object-oriented effects are used.
- Difficult to say.
- Difficult to keep pure object-oriented development paradigm apart from development that is only *based* on the object-oriented paradigm.
- Does not fit in all surroundings.
- We do not program that much.
- We do projects with traditional tools, however, we use the object-oriented paradigm in newer projects.
- Most of the development is on a mainframe, many projects are considered with existing systems.
- Enterprise resource planning is our main business.
- The time of objects is forthcoming.

Survey: The results are presented in Table 8.

Table 8: Reason why companies in the survey have not used the object-oriented paradigm

	Number of answers
Don't know what the object-oriented paradigm is	4
Don't want to use the object-oriented paradigm	2
The object-oriented paradigm is too complex	1
The object-oriented paradigm is still too immature	0
Difficult to do object-oriented testing	0
Lack of software developers trained in the object-oriented paradigm	1
Lack of software developers that are experienced in the object-oriented paradigm	1
Object-oriented software development is too expensive	1
There is a lack of object-oriented components to reuse	0
Object-oriented reuse is problematic	1
Object-oriented analysis is problematic	1
Object-oriented design is problematic	1
Lack of object-oriented databases	2
Difficulties to integrate the object-oriented paradigm with traditional databases	1
Difficulties to integrate the object-oriented paradigm with legacy systems	4
Other reason	8
Don't know why the object-oriented paradigm is not used	1
Problems with efficiency and cross-platform support	1
No development environment	1
No programming	1
Slow and expensive to train old software developers	1
Software development is not a part of our business	1
The products of today are still character based	1
Use Progress software developed during a 15 years period	1

Case studies: All the companies use the object-oriented paradigm.

Discussion and conclusions: Many of the arguments for not using the object-oriented paradigm are rather subjective and expressive like ‘don’t know what the object-oriented paradigm is’ and ‘don’t want to use the object-oriented paradigm’.

Better-argued reasons are the reasons ‘difficulties to integrate the object-oriented paradigm with legacy systems’ and ‘lack of object-oriented databases’. The problem concerning the lack of object-oriented databases is discussed in this study. The problem with the integration with traditional procedural legacy systems is an old problem that is slowly diminishing when new information systems are replacing older legacy systems.

Benefits – Management of Complexity

(Q3) Has the object-oriented paradigm been found useful when developing large-scale and complex information systems?

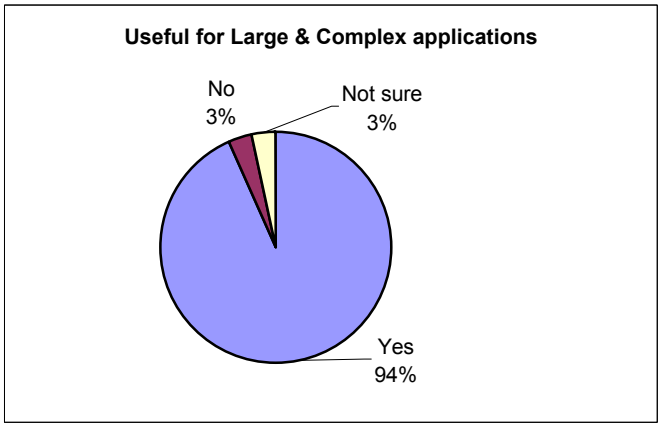
Theory – Studies: Due to the experiences of Berg et al. (1995) and the assertions by Booch (1994), Coad & Yourdon (1991, pp. 6-9) and Henderson-Sellers & Edwards

(1994, p. 5) the object-oriented paradigm is useful when developing large and complex systems.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 10.

Figure 10: The object-oriented paradigm is useful for large and complex applications



In the review of previous studies a possible association between reuse and easier management of complexity was identified. This is also the case according to 96% of the companies that had used a lot of reuse.

Case studies:

Company A: *Yes, the benefits come when there is a large system development project or a product development project. In a small customer project there is not much advantage.*

Company B: *With objects have come competent tools for managing several things, but many of the things that help maintenance and management are not purely connected to objects. We use a lot of Java and it consists of many other things than objects that we think are good. We not only handle large applications, we also take advantage of many good ways of doing things that are partly object-oriented, and partly have been learnt already with the programming language C. We have also put a lot of effort and money into configuration management and have to manage the maintenance of parallel versions.*

We have good experience of the object-oriented paradigm but when programming drivers connected to kernels one cannot always use the object-oriented paradigm; one has to use a lower level programming language like C. This is also the case when developing extremely resource critical (fast) programs.

Company C: *Yes, if one has taken the object-oriented technology into 'real' use. The difficulties come in different stages in large complex applications, depending on the complexity of the application. However, the usage of the object-oriented paradigm has often not been successful mostly because of a lack of full knowledge of the object-oriented paradigm or because of technical problems when the used software development method has not been taken into consideration. Typically, technical problems arise when interfaces and components are developed. As a conclusion, one can say that the challenges when developing a large application are not automatically handled by the object-oriented paradigm.*

Company D: *Yes, particularly when developing large applications.*

> **Company E:** *Yes we have experienced this, now we have in production systems with large overall solutions. We now have general components, both technical and domain specific; one can call them general reusable subsystems that consist of several components. There are benefits, also cost benefits, but the road is rather long when one starts from the very beginning, this does not happen during the first year, the second year nor the third year, but when one has the reusable parts.*
>
> **Company F***: We use the object-oriented paradigm in all information systems development projects where it is feasible. There are some things that are that simple that we do not go into the object-oriented paradigm, for example, simple support tools. When developing small applications with a short time limit the object-oriented paradigm is not our first choice. When carrying out more sophisticated product development we usually use object-oriented software development; in the beginning there was a lot to do but now we can utilise reuse, which makes things much easier.*

Summary of case studies: The object-oriented paradigm seems to be useful when developing large applications, though other programming languages and techniques are used when developing principally technical solutions.

Discussion and conclusions: The object-oriented paradigm has been useful when developing large-scale and complex information systems by software companies in Finland with five or more employees. This is the task-related belief that best corresponds to the review of previous studies and theory because as many as 94% of the respondents in the survey were of the opinion that the object-oriented paradigm seemed to be useful when developing large-scale information systems. No other question generated such high a percentage.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

Benefits – Productivity, faster development and reduced costs

(Q4) Has object-oriented information system development been more productive and faster than traditional information system development?

Theory – Studies: Improved *productivity* was an experienced benefit of the object-oriented paradigm in the *Survey of Advanced Technology 1996* (Pickering, 1996). According to Henderson-Sellers & Edwards (1990) object-oriented information system development is faster than traditional information system development. In the results of 12 empirical studies reported by Johnson (2002) better productivity was considered a major benefit.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 11 and Figure 12.

Figure 11: The object-oriented paradigm is more productive

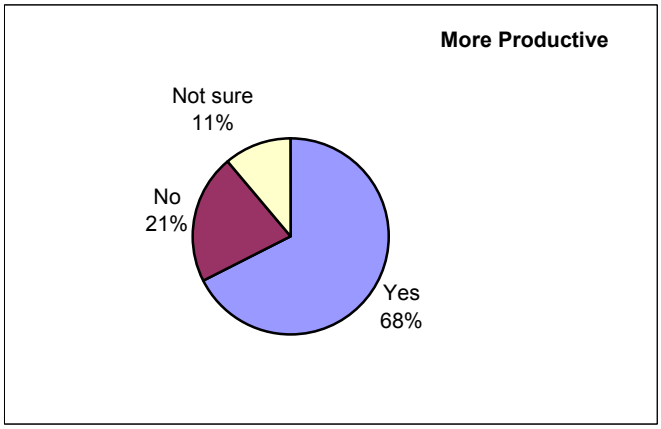
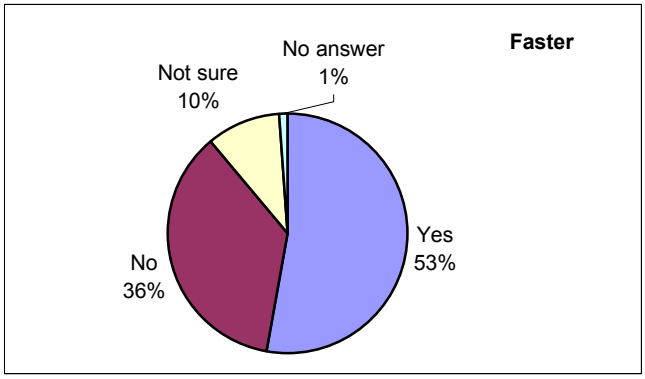


Figure 12: The object-oriented paradigm is faster



In the review of previous studies, a possible association between reuse and faster and more productive information systems development was identified. This is also the case according to 71% (for more productive) and 59% (for faster) of the companies that had used a lot of reuse (had given this answer in the survey).

Case studies:

Company A: *In a short time period the answer is: no. The time period should be longer before one can experience that object-oriented information systems development is more productive.*

Company B: *For applications suitable for the object-oriented paradigm, the productivity has been good, partly because of the object-oriented paradigm and partly because there are such mature tools in the object-oriented world.*

Company C: *With productivity the case is, especially in the J2EE projects where there is architecture with several layers, that the complexity of the environment and the large number of different options lead to the fact that productivity actually becomes worse. The problem can, however, be tackled by using frameworks. We still, however, have to build frameworks, and usually the frameworks are built in the first project, in the second project the frameworks are corrected and developed further, and finally in the third project the experience can be utilized. This situation then often generates frameworks that are not finished, and the benefits of these frameworks can therefore be questioned.*

The productivity issue is actually one reason why tailor-made applications are not as popular as before. One cannot get the same productivity with Java as with the fourth generation tools that were used before. However, it is more productive to perform information systems development with the object-oriented paradigm than with traditional software development tools and programming languages.

Company D: *Yes it has, one can work faster and when the software grows it can still be administered.*

Company E: *Let us say it like this; it is now turning into productive, in the beginning it was not productive. It will certainly be productive in the future***.**

Company F: *Yes I think it is more productive; it is more complex and more challenging but more productive, and even more interesting.*

Summary of case studies: For developing information systems where the object-oriented paradigm is suitable, the object-oriented paradigm is usually more productive than traditional information systems development, but not as productive as using fourth generation tools. However, one has to take into consideration issues like the learning curve and the experience of the information systems developers.

Discussion and conclusions: The case studies probably give a possible hint as to when object-oriented information systems development is more productive than traditional information systems development, in other words: 'when the object-oriented paradigm is suitable'. The knowledge and experience of the object-oriented paradigm probably also affects the productivity. Because a clear majority of the companies in the survey found object-oriented information systems development as more productive than traditional information systems development this is probably the case among software companies in Finland with five or more employees.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

(Q5) Has object-oriented information system development generated fewer lines of code than traditional information system development?

Theory – Studies: According to Cockburn (1993) object-oriented information system development generates fewer lines of code than traditional information system development.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *It differs from case to case; the system should be rather large before there are benefits.*

Company B: *It is often difficult to compare different ways of carrying out software development because they are on different levels. With C we program lower level programs and with Java, C++ and Python we program applications on a higher level, for example, GUI components.*

Company C: *See the answer to question 4.*

Company D: *There are more lines of codes but it looks better.*

Company E: *Yes, I have the view that this is the case. It probably depends on the developer.*

Company F: *It is difficult to say; one could say that if one makes use of reuse there is a smaller number of code lines. Nevertheless, using reuse leads to a program that gets larger with more feasibility.*

Summary of case studies: The companies are not in agreement about this contention. This is probably because it is difficult to compare traditional information systems development work with object-oriented information systems development work. The opinion which company E presented i.e. that it depends on the developer is interesting.

Discussion and conclusions: This question was only included in the case studies and because the companies were not in agreement one has to propose that this question had no appropriate answer. One reason for the uncertainty is probably that different programming languages give birth to different number of lines of code.

Benefits – Quality and usability

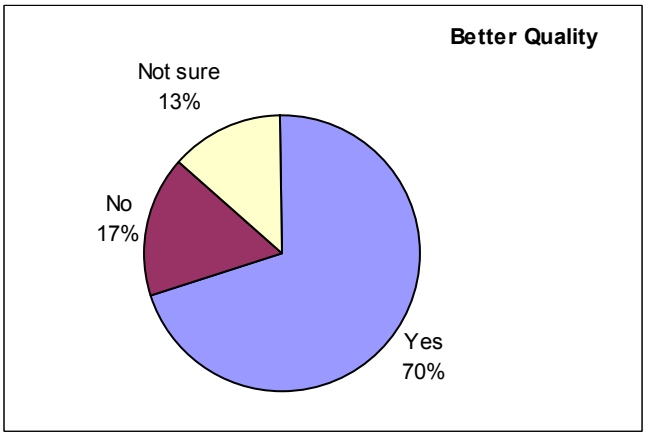
(Q6) Has the quality of object-oriented systems been better than the quality of traditional systems?

Theory – Studies: Due to the object-oriented paradigm, the *quality* of the information system can be improved, because programs are made of existing tested components and not developed from scratch every time (Gillach & Deyo, 1993; Sheetz & Tegarden, 1996; Smith & McKeen, 1996; Taylor, 1990, p. 104). Based on the results of 12 empirical studies reported by Johnson (2002) better quality was considered a major benefit. In the question a more general quality term is used, although one has to remember that there are several different types of quality (Reeves & Bednar, 1994).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 13.

Figure 13: The object-oriented paradigm is generating better quality



In the previous studies a possible association between reuse, the one model concept and better quality was identified. 68% of the companies that had used a lot of reuse had also experienced better quality in their information systems development projects. 72% of the companies that had used the one model concept had also experienced better quality in their information systems development projects.

Case studies:

Company A: *Yes.*

Company B: *It is difficult to compare the quality because in Java the quality problems have changed. All that Java solves on a lower level does not require so much time for testing because, for example, buffer overflows can be found more easily. Memory overflows are, however, just as troublesome in Java as in C.*

One thing that affects the number of errors is the fact that when Java was developed the developers tried to avoid constructs that cause problems. Java is in other words a rather 'secure' programming language.

In Java one can make things easily and quickly but this is not the same as productivity. Productivity means that one gets the expected result in a short time. In order to develop products of high quality in Java one has to be just as skilled in Java as in any other programming language. One can easily also do things wrong in Java.

Company C: *It is difficult to say because it is intricate to compare information systems (made only once) with each other. However, I think that a certain level of quality has become much better if one thinks of how different issues specified by users are handled. In J2EE projects data security is, for example, remarkably better, usage of the Web is better and the user interfaces become better when one uses the object-oriented paradigm.*

The decrease in productivity can probably be explained by the higher quality of object-oriented information systems.

Company D: *Very much***.**

Company E: *Lets say it like this; it depends very much on the developer. This is connected to the working skills of the developer; if there is an unskilled developer on the "old" side, the systems there also' fall' (there are dumps; if there is a skilled developer on the "new" (OO) side, then the systems do not 'fall'; but if there is a new and novice developer still learning object-oriented development on the new side, there will probably be problems all the way to production; especially the testing process has been difficult. But as a general rule I would say that the supporting tools for testing on the "new" side like "C test" and free open source tools better support the process that we have slowly been developing, but the process will still continue for a long time. I suggest that we will have code with fewer errors in the future.*

Company F: *Quality has certainly become better, but I am not sure that it is because of object-oriented information systems development. One can get information systems of both good and poor quality with both functional and object-oriented information systems development. However, the object-oriented paradigm has better possibilities for higher quality because of reuse; if a component has been tested and is then reused this should impose better quality. The use of reuse gives birth to some new requirements concerning testing.*

Summary of case studies: Although a certain amount of reservation exists one can argue that better quality is a result of the object-oriented paradigm. The issue of the role of the developer's working skills is interesting.

Discussion and conclusions: A clear majority of the Finnish software companies were of the opinion that object-oriented information systems are of better quality than traditional information systems. The case studies supported this finding.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies because as many as 70% of the respondents in the survey were of the opinion that using the object-oriented paradigm results in better quality.

(Q7) Has the *usability* of the object-oriented information systems in the software company been better than the *usability* of the information systems that have been developed with traditional software development methods and programming languages?

Theory – Studies: According to Sheetz & Tegarden (1996) using object-oriented analysis and design reduces the difficulty in mapping problem constructs from the problem domain with structures for the computer. This leads to higher quality and higher *usability* and maintainability (Sheetz & Tegarden, 1996). Therefore, the one model concept in analysis and design leads to higher quality, *usability* and maintainability. (Mellor & Johnson, 1997)

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No, they have been more difficult.*
>
> **Company B:** *We have no real experience of usability because we develop parts of larger applications. See the answer to the next question.*
>
> **Company C:** *Yes, because the user interfaces are better in object-oriented information systems.*
>
> **Company D:** *Yes I think that one can develop better programs with the object-oriented paradigm and that this can also be seen for the end- user.*
>
> **Company E:** *The information systems developers that come from the old side (non OO) experience the object-oriented support tools as being more difficult and more difficult to study than the tools on the old side. The threshold is very high. Nevertheless, the end users experience the object-oriented user interfaces as very pleasant. This is the general rule; there are, however, exceptions; some end users like the older, usually character based user interfaces.*
>
> **Company F:** *If I think of the situation for the end user I think the information systems today are much easier to use than, lets say 6-7 years ago. This is not due only to the object-oriented paradigm because other things have happened too; for example, the application environments also have many good components that one can use, especially when developing the user interface.*

Summary of case studies: Because the first company had a different opinion and two companies mixed user interfaces with the usability of the whole information system one can make no conclusions. That still there are end users that prefer the usually character based user interfaces was a small surprise.

Discussion and conclusions: No conclusions can be made**.**

Benefits – Natural and better mapping to problem domain

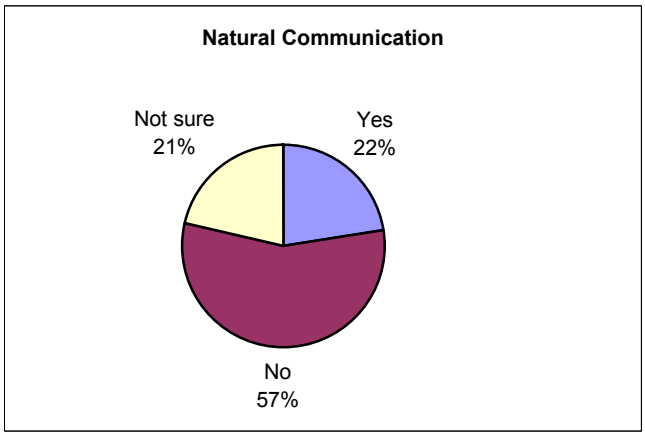
(Q8) Has there been a better and more 'natural' communication between information systems developers and end users because of using the object-oriented paradigm?

Theory – Studies: In the empirical study by Johnson (2000) improved communication with users was a found a benefit. In addition, Davis & Morgan (1993) and Gillach & Deyo (1993) propose that using object-oriented software development makes it possible for the users and software developers to speak the same language.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 14.

Figure 14: The object-oriented paradigm produces better communication between information systems developers and end users



Comments: No - Because of limitations of organisation. Yes / No - Classified as "Not sure": Among users there are experts on information systems development.

Case studies:

Company A: *No, there have been no benefits, in practice, we have been using "utilization" analysis and "usage" analysis and we have not been talking of objects at all.*

Company B: *We have not experienced this because we develop products that are part of larger applications and we therefore have very little communication with the end users.*

Company C: *If the end users are involved in the analysis work one can say that this has been beneficial for the project; however this is only true on a higher level of analysis, for example, when working with use cases. A presumption is also that the information systems developers really want to solve the problems of the end users, and not only present their software analysis method.*

One should also avoid discussing more complicated object-oriented issues like inheritance with end users (which information systems developers too often do) because this makes communication between information systems developers and end users more complicated.

Company D: *There has been no communication between the end users and the programmers and technical issues are totally internal. The end user does not see that the information system has*

been made using the object-oriented paradigm. From the customers we only get wishes regarding functions and needs, comments on what can be made better and of course feedback. The customers do not participate in our work.

Company E: *Yes we have the end users participating in the analysis stage and we use UML based information objects, which is an innovation that we have further developed from a product that the company Tietoenator sells. We always have the end users participating in the analysis phase when working with this tool, in the analysis process there is a stage where we define the use cases and in this stage the cooperation between end users and software developers is very good.*

Company F: *Our customers participate very much in the analysis stage, but I do not think the analysis is easier with the object-oriented paradigm. The result from the analysis is more a result of the software developers that carry out analysis than a result of the communication between software developers and end users. One has to remember that the end users do not necessarily know so much about software development. However, if the communication between software developers and end users is favourable then the results from the analysis are usually better.*

Summary of case studies: Because the end users do not work together with the information systems developers in most of the companies, this question cannot be answered properly. Only in company E and company F do the end users participate in the analysis work, and in these companies other factors are more important for the success than the object-oriented paradigm. In company E the analysis tool is the key factor and not object-oriented analysis per se. In company F the working skills of the software developers is the key factor.

Discussion and conclusions: A majority (57%) of the companies that participated in the survey were of the opinion that there is no better communication between information system developers and end users**.** It seems rather obvious that the findings by Johnson (2000) in the US cannot be compared with the results from the survey. In Finland the companies are generally smaller than in the US (as discussed in sub section 4.5.2) and the information systems projects are probably also smaller. In small information systems development projects there is often no communication between end users and information systems developers (finding from the case studies).

The findings from the empirical parts of this study contradict the proposition found in other studies regarding Finnish software companies. This is mostly due to the lack of co-operation between end users and information systems developers. When the end users cooperate with the information systems developers, the problem seems to be the lack of knowledge of software engineering among end users.

(Q9) Is object-oriented analysis more natural for users?

Theory – Studies: Objects are natural ensembles for many concepts in the real world according to Booch (1994, p. 78) and Jacobson et al. (1992, p. 44).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No.*

Company B: *No experience, see the answer in the previous question.*

Company C: *No.*

Company D: *No.*

Company E: *Yes, depending on how our tool is used the experiences are different, but one can argue that end users experience use cases as easier to develop than, for example, structured activity models.*

Company F: *No.*

Summary of case studies: Only one company was of the opinion that object-oriented analysis is more natural for users than traditional analysis. This might of course be due to the fact that in most companies the end users do not participate in analysis in the information systems development work that the companies carry out.

Discussion and conclusions: This question was asked only in the case studies, and because only one of the companies had actual experience of end users participating in the information systems development analysis work, one cannot answer this question.

Benefits – Maintenance

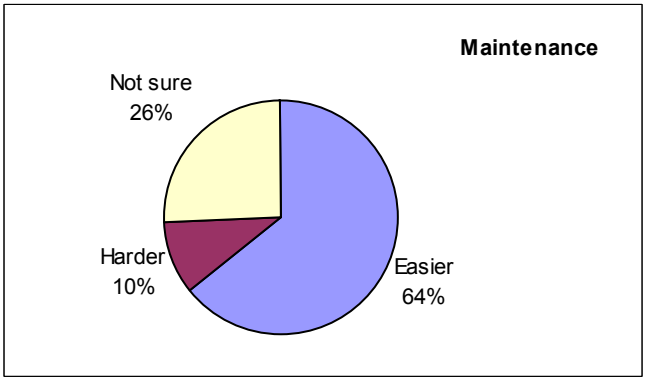
(Q10) Has maintenance of object-oriented applications been easier or harder than maintenance of traditional functional applications?

Theory – Studies: Many researchers like Agarwal et al. (2000), Booch (1994, pp. 77-78), Caliò et al. (2000), Johnson (2000), Nowicki & Kosiak (1996) and Radin (1996) argue that maintenance of object-oriented information systems is easier than maintenance of traditional functional information systems. However, researchers like Wilde & Huitt (1993) propose that maintenance of traditional functional information systems in reality is easier than maintenance of object-oriented information systems. Hatton (1998) and Wilde & Matthews (1993) propose that the complexity of object-oriented information systems is one reason why they are more difficult to maintain than traditional functional information systems.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 15.

Figure 15: The object-oriented paradigm generates more maintainable applications



Comments: One respondent answered "Not sure" because he/she had not experienced any difference.

In the review of previous studies a possible association between reuse and easier maintenance was identified. This is also the case according to 71% of the companies that had used a lot of reuse. In the review, another possible association between using software components and easier maintenance was also identified. This is also the case according to 66% of the companies that had used a lot of software components.

Case studies:

Company A: *Yes, the main reason is that because the object-oriented 'field' is harder, it is done in one way; among traditional development projects the divergence is larger, which affects maintenance.*

Company B: *It is easier to read C code than object-oriented code, but this might be due to the reader. In the object-oriented world, different parts of the application are more logical. The object-oriented paradigm gives a better probability of easier maintenance, but easier maintenance is due to several other things. In the object-oriented world, there are two things that effect maintainability, the amount of ready-made components and the lack of lower level routines and the structure of the object-oriented programming language that supports documentation. A large library supports maintenance, but this is the fact with C as well.*

Company C*: If the object-oriented information system is built without interfaces in a more traditional way, then maintenance is the same as before. However, if the object-oriented information system is built with real components and with a real application framework, then maintenance is easier. The situation today is that many organisations that made object-oriented software development in the late 1990's now have a lot of Java code that is hard to maintain.*

Company D: *It is much easier. If one makes the object oriented program thoroughly one knows that a change in one part of the program only affects the part in question and not any other parts of the program.*

Company E: *It depends on the information systems that we have in production and on the information system developer that works with the information system. There are information system developers that find maintenance very easy and there are those who find it difficult. This is probably an issue of working skills and how working skills improve when the information system developers become more experienced.*

Company F: *Maintenance is always difficult; we have a lot of products and some of the code is written by us and then integrated with the source code of mainstream products. However, the object-oriented paradigm makes maintenance a little bit easier.*

Summary of case studies: If the object-oriented paradigm is used appropriately then maintenance of the object-oriented information system is in all probability easier. The skills of the information system developer must also be taken into account.

Discussion and conclusions: It appears that the theoretical proposition is true for Finnish software companies. The maintenance of object-oriented information systems is easier than the maintenance of traditional information systems according to a clear majority of the Finnish software companies.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

Benefits – Software components

(Q11) Have readymade components been used and been considered beneficial for information system development?

Theory – Studies: Pancake (1995) claims that the greatest advantage of the object-oriented paradigm is the fact that objects can be used as *software components;* however, components can also be analysis components, design components or programming components (Coad & Yourdon, 1991, p. 124), etc.

Pilot study: In the pilot study one respondent answered 'Standard solutions' and another answered 'The component architecture'.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *We are using readymade components and they are beneficial; for example, we are using Stratch and in a way, Stratch supports the development of programming code that is of better quality.*
>
> **Company B:** *We have developed components in-house and we have bought ready made components. We have not been participating in any open source community. We also have a managed reuse of code, but this can also be made of course in a traditional programming environment.*
>
> *Some open source components can, however, be found in our company. The quality of the open source components is very different; out of ten components one might be very good and nine are rubbish and do not work. When using open source components one has to spend a lot of time in order to check the quality of the component. This time one can just as well spend in programming a new similar component.*
>
> **Company C:** *As a general rule we develop our own components and usually only for the needs of one or two projects. We have been evaluating readymade components for the last five years, and we have found several interesting components. Nevertheless, there are also several problems like the 'black box' phenomena where one is not allowed to get the source code, which means that customers cannot develop the components. When one cannot drop the readymade components and cannot further develop them one has often to pay expensive licence fees for a long period of time. Therefore, we have not found readymade commercial components as appropriate to use.*
>
> *Concerning open source components, the situation is that we use them, but we are not very fond of them because one can never be secure of their quality. An exception is components that are, for*

example, already in the product palette of IBM or some other known company. These components are usually reliable, but on the other hand, they are not necessarily components any more; they are often net services, application modules or something else. The Strux Framework is a typical example of this.

Company D: *We use readymade components to some extent, especially for network implementations and cryptology parts of the information system. However, we try to develop as much as possible by ourselves. We have used both open source components and components that we have bought.*

Company E: *We use technical readymade components that we have developed in-house and we have found these very useful; for example, error management components that we use in all new information systems that we develop. This results in a nice situation for the end users who switch between information systems; in all information systems the error management is performed in the same manner. We do not use commercial components that can be bought from other companies.*

Company F: *Yes we use components and try to develop them if we have enough time and resources. Time is indispensable because testing components that will be used in the future is important.*

Summary of case studies: All the companies use readymade components and they have found readymade components beneficial in their information systems development work. The companies are not very fond of open source components because of their varying quality.

Discussion and conclusions: One cannot generalize matters based on case studies but most probably readymade components are used a great deal and found beneficial by many software companies in Finland. The companies, however, seem to avoid open source components because of their varying quality.

(Q12) Have the companies developed software components? If they have, are the companies of the opinion that the object-oriented paradigm has made the development of software components easier?

Theory – Studies: Eriksson (1992, p. 54) argues that software components or modules are easier to develop due to the object-oriented paradigm. Kaasböll (1993) is of the same opinion and claims that the easier development of components is due to object-oriented software development methods for application development, Caliò et al. (2000) also feel the same and present UML as such an object-oriented software development method

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes we have, but less today, more before when our software development was more about developing software for the client.*

Company B: *Yes, it depends on the component, but as a rule, the object-oriented paradigm has not made the development of components easier. It is not easier to program components in Java than in C, though C might be an easier programming language to learn and use.*

Company C: *Yes but mostly small components. Some larger components have, however, been developed for banking applications.*

Company D: *Yes, we have, coding is faster, but one does have to know what one is doing.*

Company E: *Yes, we have. It is not easier or more difficult to develop components using the object-oriented paradigm than using some other older paradigm. The point is to find the "glue"- the interface between the components.*

Company F: *The object-oriented paradigm has made it a little bit easier. It was not that difficult before but is perhaps somewhat easier nowadays. The measurement and comparison of older functional and newer object-oriented component development is of course tricky.*

Summary of case studies: The companies have developed software components of their own; the question whether the object-oriented paradigm has made the development of components easier is rather unclear, probably because the companies have not developed many components using traditional information systems development techniques; only company F was an exception.

Discussion and conclusions: One cannot generalize matters based on case studies but most probably many software companies in Finland have developed software components of their own.

Benefits – End – User computing

(Q13) Are the software companies using End-User Computing? If the software companies are using End-Using Computing, has the object-oriented paradigm made it easier in the software company?

Theory – Studies: Winblad et al. (1990, p. 49) point out that perhaps in the future the users of today can develop and build applications of their own easier, using the object-oriented paradigm in an End-User Computing manner.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Some clients have started to maintain our software and continued to program our software. This has been done when the software is office based and not when it is independent and object-oriented.*

Company B: *We have active clients (often larger companies), and in these companies software developers further develop our programs. The quality of applications developed by end users is often a problem. There are, however, very skilful end users.*

Company C: *Yes, there is often some kind of special evolution to the information systems that we have developed. Nowadays the object-oriented technology is such that if end users start to build information systems of their own the whole work process goes into a 'knot', and this 'knot' cannot easily be opened. We have experienced cases where end users have made so much with Excel that the whole thing had became a huge 'knot', and then a software house is connected in order to get the 'knot' opened. For end users the technology is not mature enough for developing larger information systems. Nevertheless, they can of course develop small applications.*

Company D: *No, our customers have no access to the source code.*

Company E: *We have a rule that our customers are not allowed to develop information systems of their own. There are, however, some eager engineers that have developed some very small applications with Excel. The maintenance of end user information systems is always difficult and therefore end users are not allowed to develop information systems of their own.*

Company F: *Our customers do not develop much software anymore because most of the software development tasks are outsourced to us. There are some hobby programmers but the quality of work is usually poor.*

Summary of case studies: Most of the companies have experienced end users developing information systems. The quality of software that end users develop is, however, often rather defective. None of the companies commented whether the object-oriented paradigm has encouraged end-user information systems development.

Discussion and conclusions: End user computing is probably a fact in many Finnish companies (the 'clients'), though the quality of the information systems they develop is not always very good. Whether the object-oriented paradigm has promoted end-user development cannot be determined. According to one of the companies in the case studies, some end users develop information systems with Microsoft Excel. Because this question was not included in the survey, no generalisations can be made.

Benefits – One model

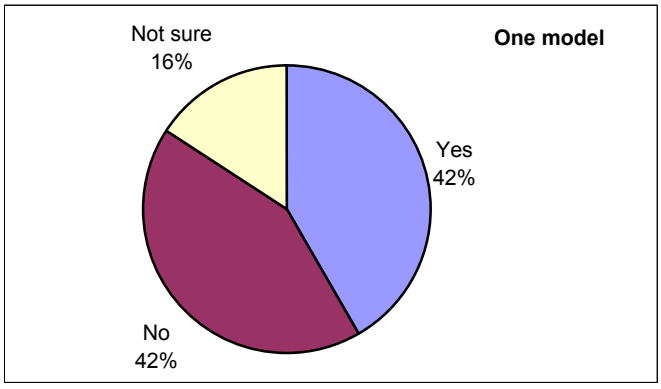
(Q14) Has the object-oriented system development process been seen as a uniform 'one model' from problem domain to code and maintenance in the software company?

Theory – Studies: The object-oriented paradigm has a uniform paradigm throughout development from analysis to implementation and maintenance (Coad et al., 1995, pp. 481-485; Henderson-Sellers & Edwards, 1990).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 16.

Figure 16: The object-oriented paradigm as one model



Comments: One respondent answered "Differs from one project to another", classified as "Not sure".

Case studies:

> **Company A:** *We have not been using the waterfall model; we have been using a model with boundaries that can be seen as one model though we have developed the system part by part, sometimes according to the waterfall model.*
>
> **Company B:** *The project management model is not connected to the software development paradigm. With pure object-oriented concepts one cannot manage all necessary things. With UML one can specify requirements but we also use "end-user concepts" in our work.*
>
> **Company C:** *Yes, in several projects we have experienced this phenomenon; artefacts from analysis have become components or objects with the same name in the implementation phase. However, the benefit from this is usually not very big because in the implementation phase the component does not only consist of the artefacts from the analysis phase; in fact often about 90% of the component (or object) consists of code for technical issues. The possibility to trace back from the component to the artefact in analysis is often not very clear.*
>
> **Company D:** *In our software house the analysis part is very limited. Design is also made at the same time as we program. We do not develop large class libraries and structures that we then start to implement.*
>
> **Company E:** *We have a very iterative way of performing information systems development. We might return to analysis or design, and check how to do something in a very iterative manner. The older software development work was much harder than the iterative work we do today.*
>
> **Company F:** *The process has become more straightforward, our company is so large that we have to use clear working procedures. We still see the different phases, but the boundaries between the different phases have become indistinguishable. The placement of the boundaries also depends on which software developer plans the information system; some software developers skilled in UML carry out pure requirements analysis and other software developers do a lot of design. However, the iteration concept is of course known and used.*

Summary of case studies: Because the companies use different information systems development practices this question cannot be properly answered. Two of the companies had, however, experienced the 'one model' phenomena.

Discussion and conclusions: Because of the results one cannot argue that the 'one model' development process is recognized a lot among Finnish software companies. Because the number of 'yes' answers were exactly the same as the number of 'no' answers the conclusions in the next paragraph are rather weak.

The findings from the empirical parts of this study contradict the proposition found in other studies regarding Finnish software companies. Though the case studies do indicate that the question is easily misinterpreted and therefore one ought to be careful when analysing the results. The lack of the 'one model' development process is probably due to the used company specific development processes. These company specific working processes often have a rather weak connection to a theoretical development process and propositions found for theoretical development processes are thus seldom very well supported.

(Q15) Have the companies found that there is a benefit because there are the same building artefacts in object-oriented analysis and object-oriented design?

Theory – Studies: Mylopoulos et al. (1999) stress the fact that the whole software development process can be made easier when the designer has the same building artefacts from analysis to design and implementation. The artefacts are the object, the classes, methods, messages and inheritance, etc. (Mylopoulos et al., 1999).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Yes, in one case objects were found rather early, but typically; no.*
>
> **Company B:** *In design the use of objects makes things easier to understand.*
>
> **Company C:** *Yes, the building artefacts in analysis and design are often very much alike. This is, however, not the case anymore when the information systems development project reaches the implementation phase. See question 14.*
>
> **Company D:** *No question was asked and no answer was received because the company does not carry out analysis.*
>
> **Company E:** *Yes.*
>
> **Company F:** *No real benefit.*

Summary of case studies: It seems that the companies that perform proper object-oriented information systems analysis have also experienced that one can recognize some objects that are the same objects in analysis and design.

Discussion and conclusions: In the case studies the respondents talked about 'normal' objects that come from true analysis and more 'technical' objects that are needed for network communication etc. Among the 'normal' objects there are objects that run from analysis through design into implementation. One cannot generalize matters based on case studies.

Benefits – Frequent tangible working results and reliability

(Q16) Has object-oriented information system development given frequent tangible working results?

Theory – Studies: Coad et al. (1995, pp. 481-485) and Radin (1996) propose that frequent tangible working results are considered a benefit of the object-oriented paradigm.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No.*
>
> **Company B:** *Yes, when ready-made components were used.*

Company C: *Only infrequently.*

Company D: *Not asked because of an error by the author of this study.*

Company E: *We have a number of software developers that are real virtuosos who are able to develop smaller applications in a few days.*

Company F: *Yes, especially when developing parts of systems. We have used commercial components and put them together and then further developed the mix. This is very productive and easy; one can buy the component and it's source code on WWW, pay with a credit car, and then use and modify the new component.*

Summary of case studies: This is not generally the case. Two companies had, however, experienced this. One company argued that this is more due to personal skills among software developers than it is due to the used software development paradigm (company E).

Discussion and conclusions: The possibility to gain frequent tangible working results is probably connected to reuse. Although reuse is performed (see answers on question 20) frequent tangible working results are not always accomplished. One cannot of course generalize matters based on case studies.

(Q17) Have the object-oriented information systems in the software company been more reliable than the information systems that have been developed with traditional software development methods and programming languages?

Theory – Studies: Lim (1994) and Page-Jones (1992b) also claim that reliability is a benefit of object orientation.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes.*

Company B: *No.*

Company C: *Yes.*

Company D: *Yes, they are, if one can develop objects that are isolated and only have connections with the environment through the procedures and parameters of the object. In traditional maintenance global variables were often difficult to manage.*

Company E: *No, it depends more on the skills of the software developer who has developed the information systems. If the software developers are not very skilled then sometimes the information systems are not that reliable.*

Company F: *Yes, if reuse is used properly.*

Summary of case studies: Four out of six of the companies had experienced this. Company E again presented the issue of the significance of the skills of the information systems developers. Company E is a software company with long experience in the information systems development field.

Discussion and conclusions: The four Finnish software companies in the case studies had experienced that the reliability of object-oriented information systems is better than the reliability of traditional information systems. One cannot of course generalize matters based on case studies.

Benefits – Suitability for embracing new technologies and sound academic basis

(Q18) Have the companies experienced that the object-oriented paradigm is a good tool for embracing new technologies like graphical user interfaces or client-server applications?

Theory – Studies: Regarding this benefit, graphical user interfaces and client-server applications are mentioned as new technologies that can be developed straightforwardly with the object-oriented paradigm (Coad et al., 1995, pp. 481-485).

Pilot study: In the pilot study it was found that the object-oriented paradigm has a significant future benefit regarding object-oriented models and tools.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Developing the user interface is a large part of the development work and in this work the benefits of the object-oriented paradigm are not that considerable. Few good ready made components have been found for developing graphical user interfaces.*
>
> **Company B:** *When developing graphical user interfaces Java is a very suitable tool. Generally, objects are suitable for developing graphical user interfaces. Regarding client – server applications objects are not necessary the right solution; lower level languages are usually more productive.*
>
> **Company C:** *Yes, or one can say that we try to use the object-oriented paradigm in all new information systems development projects. There are, however, still some customers that want information systems that have been built in a traditional way.*
>
> **Company D:** *Yes.*
>
> **Company E***: Yes I think this is the case. However, the model of thinking is important. One is forced, however, to comprehend the object-oriented way of thinking, if one still thinks in the old functional way then the object-oriented paradigm is not suitable for developing new information systems. Nevertheless, generally I think that all the 50 information systems developers who develop object-oriented software in our company would answer "Yes" to this question.*
>
> **Company F:** *Today I do not think there are many other alternatives than the object-oriented paradigm.*

Summary of case studies: The companies had all experienced this. One company, however, did not find the benefits especially extraordinary. Another company presented the problems with older software developers who had difficulties in starting to think in an object-oriented way.

Discussion and conclusions: When developing information systems today that are client – server based, and have graphical user interfaces, the choice of the object-oriented paradigm is not far away. Of course these kinds of information systems can

also be developed with traditional information systems development tools like the programming language C and D-Screen. One cannot of course generalize matters based on case studies.

(Q19) Are the companies of the opinion that the sound academic basis of the object-oriented paradigm is a benefit?

Theory – Studies: There is a strong theoretical background for the object-oriented paradigm. Academic research will also support the development of the object-oriented paradigm. (Smith & McKeen, 1996)

Pilot study: In the pilot study, one respondent answered "know-how".

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Yes, books and people can be found. Nowadays there are several students coming from universities with good knowledge of the object-oriented paradigm and these students have a positive attitude towards it.*
>
> **Company B:** *That there is knowledge based on experience is a benefit, but not all knowledge is of good quality.*
>
> **Company C:** *Yes, books (from Amazon, for example) and people can be found. Nowadays there are several students coming from universities that have studied an object-oriented programming language as their first programming language. They also have good knowledge of the object-oriented paradigm. Furthermore, students have a positive attitude towards it.*
>
> **Company D:** *Yes it is beneficial when studying that suitable academic material exists so that one learns the object-oriented way of thinking, because it is different from the traditional way of thinking.*
>
> **Company E***: Yes, definitely, for example, in different discussion groups there is a lot of information available. This possibility was not available before the Internet.*
>
> **Company F:** *Before we adopt any new technique, we always first evaluate the quality of the documentation of the technique.*

Summary of case studies: Companies A – D probably understood the question in the right way and had found the academic basis of the object-oriented paradigm beneficial. Companies E and F did not answer the question and probably misunderstood the question. What is interesting is that an academic base per se is not good; one has to understand the object-oriented way of thinking as well.

Discussion and conclusions: In order to learn the object-oriented paradigm and the object-oriented way of developing information systems one can use books and journals etc. There is no lack of material about the object-oriented paradigm on the market. This is the 'benefit' that was reported by the Finnish software companies that participated in the case studies. One cannot of course generalize matters based on case studies.

Benefits – Reuse

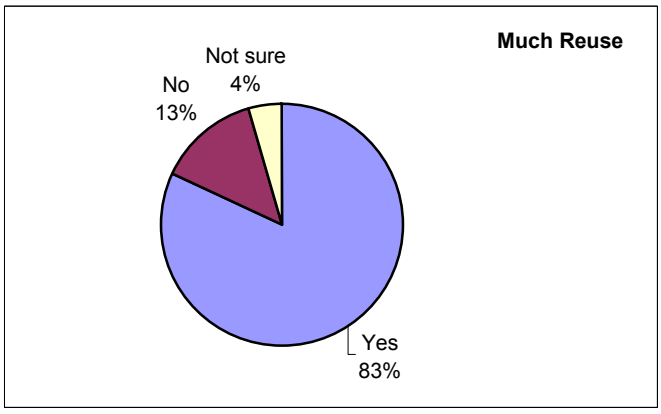
(Q20) Have the software companies used much reuse? Has reuse in the object-oriented paradigm been as beneficial as several researchers propose it to be?

Theory – Studies: Reuse often results in less rework in the development process (Basili et al., 1996a). However, Mili et al., (1999) propose that the benefits of reuse are not always realised.

Pilot study: In the pilot study it was found that the benefits of reuse are only found later, but anyway that it is still important.

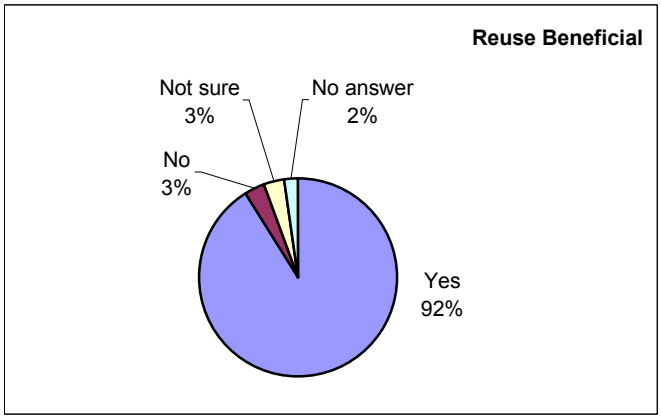
Survey: The results are presented in Figure 17 and Figure 18.

Figure 17: Companies used much reuse



Comments: one respondent answered "Yes/No" and one respondent answered "No/Not sure", both are classified as "Not sure"

Figure 18: Reuse considered beneficial



The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies because as many as 92% of the respondents were of the opinion that reuse is beneficial.

Case studies:

Company A: *Looking back, on can say that in practice it has not been beneficial, and we have not been using reuse. But now we are using another software development environment and nowadays reuse is used more and can be considered beneficial. The reuse of components is, however, seldom happening.*

Company B: *See question 21 and corresponding answer.*

Company C: *We have levels of reuse. For example, in the Open Method application development method that we use we have made reuse solutions that are much deeper than those that we have made in ordinary work. We have, for example, ready made document skeletons and use structures, and by using these we can rapidly develop analysis that is cohesive and highly useful. This way of working has supported reuse on the analysis and design levels.*

In the implementation phase the reuse of classes has not been successful. However, reuse has in practice been beneficial when working with Frameworks. The Framework includes the model for programming, a ready-made program skeleton and often there are also some 'generators' that can be utilized in the programming work. One can also 'glue' several things to the Framework like log components and user interface elements, etc. Reuse with Frameworks has been most successful. On the component level there are few good components to reuse, and reusable components do not pop up by themselves. Because we have a lack of time and the technology develops so fast we cannot start to develop reusable components.

Company D: *Yes, we have some components that can be used in almost all programs. We do not try to maximize reuse; we use reuse where it is suitable and where we can save some effort.*

Company E: *We have used reuse but not very much. We have found reuse good in the few cases when we have used it. Mostly we have used it in technical settings, but also in some business cases concerning customers. When utilizing reuse the interfaces have to be very properly defined; this definition work is not always very easy.*

Company F: *We use a lot of reuse; we have a substantial class library and we reuse classes from this library when possible.*

Summary of case studies: All the companies utilized reuse though the experiences of the benefits of reuse were different. Reuse is probably not considered as a major benefit for information development work.

Discussion and conclusions: Because a vast majority of the Finnish software companies had used reuse and found it beneficial, one can only argue that one of the most promising benefits of the object-oriented paradigm.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

(Q21) What do the software companies reuse?

Theory – Studies: Software reuse was studied by Gehringer & Manns (1996) by asking managers who directed object-oriented programming projects and according to the findings class libraries purchased from vendors and class libraries developed in-house were reused. Note that objects can also be reused because there are object-oriented programming languages like Smalltalk where everything is an object (Khoshafian & Abnous, 1995, p. 16). In programming languages like C++ and Java, objects are of course not reused.

Pilot study: The question was not included in the pilot study.

Survey: The results were the following:

	Number of answers
Objects	32
Classes	74
Class libraries purchased from vendors	25
Class libraries developed in-house	65
Analysis	25
Design	26
Components	62
Other	10
Of which:	
Documentation (for end users, etc.)	1
Frameworks	3
Free libraries	1
Inheritance of "schemas"	1
Open source libraries	1
Patterns	1
Teaching material	1
Testing material	1

Comment: Of those that had used components all considered them useful.

The respondents gave several answers and therefore no chart was made.

Case studies:

Company A: *We have a standard library that we reuse, all Java applications that we develop should be built upon this standard library, but Stratch is connected to the standard library, and it*

should also work in all projects. We have also our own library, which we use when developing products that are more complex.

Company B: *We develop applications in different phases and reuse components from the standard library. In our project management we have an important goal that components will be reused.*

Company C: *Analysis and design, see question 20.*

Company D: *We use, for example, the database components of Linux. We also use network components.*

Company E: *Mostly we have used components, but we have also developed some small subsystems for some special tasks.*

Company F: *We use classes, class libraries developed in-house, components, analysis, design and documentation, etc.*

Summary of case studies: The companies reuse components, components from standard libraries, analysis and design, database components of Linux, subsystems, network components and documentation.

Discussion and conclusions: The Finnish software companies reuse classes and components to a high degree. This is not surprising, because this is a base within the object-oriented paradigm. What is interesting, however, is that most Finnish software companies like to reuse components they have developed in-house. For example, open source components are not very popular for reuse. This finding is somewhat unexpected. One would expect that companies would share components with each other and not only share components inside the company.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

(Q22) Do information system developers prefer to reuse rather than to build from scratch, or do they consider reuse so difficult that they rather build components from scratch?

Theory – Studies: According to Frakes & Fox (1995) they prefer to reuse. However, according to Sparling (2000) many developers think that it is better to build a component from scratch than to reuse an existing one.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes, sometimes we have had to struggle in order to get the software developers to use readymade components. There is also a problem because some software developers are afraid to change components that are in the standard library, which leads to a way of working where the software developers take a copy of the component. This practice makes project administration more difficult and the component becomes "project specific" and not suitable for the standard library.*

Company B: *One has to be aware of the components, which means that components have to be documented. The second question is if the existing component is suitable for the job in question. One has to work in order to get reuse; reuse does not come by itself.*

Company C: *There are different types of programmers, but as a rule, the programmers prefer to develop their own components. See also question 20. Sometimes the programmers find interesting components but then they recognize that the components do not fulfil their requirements, which makes them even more unwilling to reuse existing components. However, project managers who are responsible for the timetable often try to find readymade components in order to get the work done faster.*

Company D: *If the information systems developers have a readymade component that can be used, they use this. However, it is the quality level requirement of the final product that determines what the information systems developers do. For example, when they used open source components they often got a lot of bugs (errors) and therefore they often had to develop components of their own.*

Company E: *Yes, we have experienced this problem in some projects, I do not know if this is a result of shortages in understanding, informing or something else. We have young enthusiastic men that like to develop components of their own, though we might have ready-made components that could be used.*

Company F: *They prefer to reuse.*

Summary of case studies: Three of the companies had experienced this problem. This problem is in other words not unknown, though it seems that it can be managed without any big difficulties.

Discussion and conclusions: This problem exists in Finnish software companies. Because this question was not included in the survey, no generalisations can be made.

(Q23) Are finding suitable components a hindrance for reuse?

Theory – Studies: According to Nokso-Koivisto (1995) reuse of components cannot often be carried out because no adequate component can be found. Glass (1998) proposes that in order for a component to be reusable it has to have 80% or more of the specifications and functionality that is needed.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes to some extent, but if this is due to the difficulty in finding a suitable component, or if this is due to the unwillingness to use a component, is hard to say.*

Company B: *Yes, one has to be aware of a component in order to be able to reuse it.*

Company C: *Yes.*

Company D: *No, we know very well what we have.*

Company E: *Yes, although we have scanned the component market.*

Company F: *No.*

Summary of case studies: When using libraries that have not been developed in-house, this seems to be a problem for most of the companies.

Discussion and conclusions: To find a suitable component from outside the software company seems to be a problem. Because this question was not included in the survey, no generalisations can be made.

(Q24) Have the producers of reusable components in the software company considered the needs of the future users of the components? (Both people and systems.)

Theory – Studies: According to Coleman et al. (1994, p. 230) producers of reusable components have to think about the needs of the future users of the components.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Software developers are not in contact with end users (people). The project managers are, however, considering this issue. Budgeting is the main obstacle. If the customer has to pay for the software development work, it is not fair to have the customer also pay for the extra work connected to the development of reusable components. When working with joint projects where there has been both software development and software enhancement this has been working.*
>
> **Company B:** *Yes, it is a part of the project management. This has been done very much when developing components for graphical user interfaces. It has been important to build a special level of components that makes the graphical user interfaces more alike.*
>
> **Company C:** *There is a fascinating phenomenon; when the information systems developers recognize a place for a ready made component they do not even search for such a component, instead they start to develop a new 'reusable' component by themselves, which means that the information systems developers do 'extra' work which will be a burden for the ongoing information systems development project. One can talk of 'reuse for the future' that is happening in the wrong direction; one makes reusable components though there is no real need for this. This problem must me managed by the project management.*
>
> **Company D:** *Yes, when we develop components we try to develop components that have as few constraints as possible. We do not consider all possible future needs, but we try to develop components that do not prevent forthcoming needs.*
>
> **Company E:** *Some projects can be called "harvest projects" because they can utilize components that have been developed in other projects. We also try to consider forthcoming projects when developing components.*
>
> **Company F:** *Yes.*

Summary of case studies: All the companies are aware of this issue, and also of the problem as to how to divide the costs for developing 'for the future'.

Discussion and conclusions: The Finnish software companies that took part in the case studies were aware of this dilemma. Because this question was not included in the survey, no generalisations can be made.

(Q25) Has multiple inheritance been used? If multiple inheritance has been used, has it been successful?

Theory – Studies: According to Koskimies (1995) multiple inheritance is considered by most researchers as having more disadvantages than advantages.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No.*
>
> **Company B:** *In Java one can use multiple inheritance only when working with interfaces, and this has been done. Pure multiple inheritance is not used in our company, and it is better so.*
>
> **Company C:** *Very seldom, mostly because Java does not support pure multiple inheritance. Multiple inheritance is a mechanism that is very difficult to master. We try to use delegation instead of multiple inheritance.*
>
> *However, theoretically we think that multiple inheritance is useful in some cases. For example, in C++ the Persistence Framework, one inheritance is for the Persistence implementer and the other inheritance is for the application hierarchy.*
>
> **Company D:** *We use object-oriented Delphi and there is no multiple inheritance. This is good because multiple inheritance might make things more convoluted.*
>
> **Company E:** *No. We use mostly Java.*
>
> **Company F:** *Yes, we use it to some extent in our C++ code. We have not experienced any notable problems with multiple inheritance.*

Summary of case studies: Only one company used multiple inheritance. Though most of the companies did not use multiple inheritance the answers indicated that they were probably aware of the dangers with it.

Discussion and conclusions: Most of the Finnish software companies avoided multiple inheritance and some of them used programming languages like Java where there is no multiple inheritance. The Finnish software companies seemed to be aware of the dangers with multiple inheritance. Because this question was not included in the survey, no generalisations can be made.

Benefits – Object-oriented analysis

(Q26) Can the users switch from the object-oriented paradigm to the functional paradigm and back in a smooth way?

Theory – Studies: According to Sommerville (1992, p. 66) users can switch from the object-oriented paradigm to the functional paradigm and back in a smooth way.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *The users are not aware of what approach is used; only the application manager is interested in this issue.*
>
> **Company B:** *We have no real contact with end users***.**
>
> **Company C:** *We have different kinds of information systems development projects. For example, in a large project that lasted for several years, the customer had developed the analysis phase by himself; with use cases several customers can develop analysis by themselves. Many customers have some understanding of the object-oriented technique today. However, we have also had information systems development projects where we have recognized that the object-oriented technique was too difficult for the customer to comprehend.*
>
> **Company D:** *We do not use analysis.*
>
> **Company E:** *The end users are only in one world; the functional or the object-oriented. There are a few exceptions, but then the end users have found it difficult to move from the functional world to the object-oriented world.*
>
> **Company F:** *Our customers and end users give us the requirements but do not participate in the pure analysis work. The end users, however, participate in issues related to the analysis of the requirements for the graphical user interface. When the customers work with the requirements they can define the requirements without problems.*

Summary of case studies: Only one company answered the question. The main reason is the limited contact between end users and the information system development project members. The company that answered the question was of the opinion that the users are not able to switch from the object-oriented paradigm to the functional paradigm and back in a smooth way. Another problem was that the companies talked about end users and the question involved all kind of users.

Discussion and conclusions: Unfortunately, no conclusions can be made because of the fact that the companies misunderstood the question, which was due to an error made by the author of this study who made the interviews.

(Q27) Have the companies used prototyping for finding requirements in object-oriented information systems development?

Theory – Studies: Prototyping is often used for finding the requirements in analysis (de Champeaux et al., 1993, pp. 7-8).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Prototyping has been used but it has not been especially beneficial.*
>
> **Company B:** *No comments.*
>
> **Company C:** *Yes, especially in large information systems development projects and in user interface and stress testing. We have, however, never built a prototype of a whole information system together with end users.*
>
> **Company D:** *No comments***.**

Company E: *No, though we are aware of the concept.*

Company F: *Yes, we use a lot of prototyping. We develop small prototypes that we present to our customers, especially when we develop large information systems. The small prototype is then a part of the larger system. Prototyping is especially useful when developing user interfaces.*

Summary of case studies: Prototyping is known and used by three of the companies, but it seems to be a rather uninteresting issue for all of them, with the exception of company F.

Discussion and conclusions: This question is not a core question regarding the object-oriented paradigm. The interest in this question was low among Finnish software companies. Because it was not included in the survey, no generalisations can be made.

Benefits – Object-oriented design

(Q28) Has the transition to object-oriented design from object-oriented analysis been easy or difficult?

Theory – Studies: The information that is developed in the analysis phase becomes an integrated part of the design instead of only being the 'input' to the design (Korson & McGregor, 1990). Here lies the benefit of object-oriented design in comparison with traditional structured design. In traditional structured information systems development (in theory), analysis and design are strictly different activities. However, some researchers like Kaindl (1999) think that this is not very true.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No differences have been found regarding the move from object-oriented analysis into object-oriented design, in comparison with the move from traditional analysis into traditional design.*

Company B: *No differences have been found regarding the move from object-oriented analysis into object-oriented design, in comparison with the move from traditional analysis into traditional design.*

Company C: *No problems in the transition process.*

Company D: *We do not use design.*

Company E: *It has been somewhat easier in the object-oriented world due to the possibility to iterate.*

Company F: *It depends on the software developer and the usage of UML, probably no real difference.*

Summary of case studies: The transition from object-oriented analysis to object-oriented design is considered as difficult or easy as the transition from traditional analysis to traditional design. Only one company was of the opinion that it is easier in the object-oriented world, due to the possibility to perform iteration.

Discussion and conclusions: It seems that the transition from object-oriented analysis to object-oriented design is as easy or as difficult as the transition from traditional analysis to traditional design. Note that not all the Finnish software companies worked with design. Because this question was not included in the survey, no generalisations can be made.

Benefits – Portability

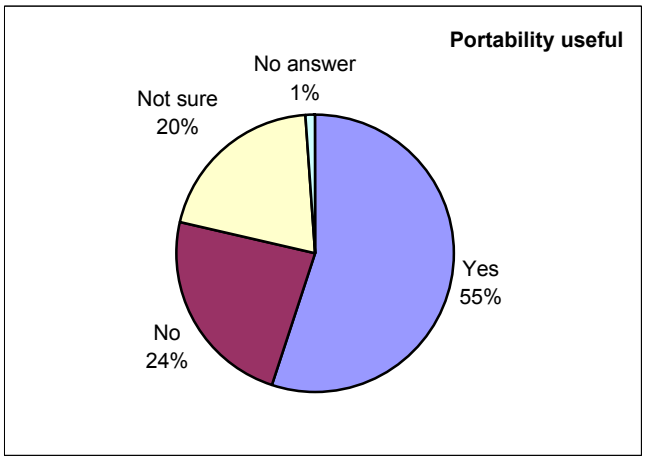
(Q29) Has portability of the object-oriented system been a benefit?

Theory – Studies: Theoretically portability is considered a benefit of the object-oriented paradigm (Agarwal et al., 2000). The idea is that an object-oriented program can run on every computer with the assistance of a virtual machine. This is the case if, for example, the programming language Java has been used.

Pilot study: In the pilot study it was found that the object-oriented paradigm does not fit in all surroundings.

Survey: The results are presented in Figure 19.

Figure 19: Portability useful



Case studies:

Company A: *Yes, mostly because one can perform software development on both Windows and Unix. Usually the customers want Windows based applications, but there are some that want Unix based applications.*

Company B: *One should not talk so much about portability; one can move traditional applications as well as object-oriented applications. Java has, however, a better portability than C.*

Company C: *Yes, there is a benefit, but the portability does not come up automatically; every information systems developer has to work with this issue and consider that the final information system might be used on a Unix or Aix computer. The possibility to relocate an information system that is in production to another operating system on another computer is never utilized.*

Company D: *Yes, on our server for the school administration we have the same source code for both Linux and Windows based information systems.*

Company E: *Yes, we have utilized portability in a pilot project and our experiences are that the mainframe "eats" the application surprisingly well.*

Company F: *Yes, we have utilized portability between Windows and Linux, but portability of an object-oriented system is not without problems. However, class libraries can be moved nicely, and user interfaces too.*

Summary of case studies: The portability issue is considered a benefit and the companies have moved information systems from the Windows platform to different kinds of Unix platforms and even to a mainframe platform (the operating system of the mainframe was not mentioned).

Discussion and conclusions: The portability of object-oriented information systems is undoubtedly considered a benefit among Finnish software companies. As an example, one software company in the case studies mentioned the transition of an information system from a Windows environment to a Unix environment.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

Benefits - Other

(Q30) Has the total independence of classes given advantages in system development compared with the traditional solution with modules with common data?

Theory – Studies: In traditional programming independent modules can be developed, but as long as these modules use common data with other modules, they are not totally independent of the environment in the same manner as the class with its objects that have both methods and data encapsulated.

Pilot study: In the pilot study it was found that the encapsulation of functions is a benefit of the object-oriented paradigm.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes, the quality of the product becomes better.*

Company B: *Yes.*

Company C: *Yes, it is a very good idea that the data of the object is in the object itself.*

Company D: *Yes***.**

Company E: *No actual benefits. It has been as difficult to develop object-oriented information systems, as it has been to develop traditional information systems.*

Company F: *No comments.*

Summary of case studies: All but one of the companies considered this a benefit. A manager that represented the company with a different view might have been unaware of practical programming issues.

Discussion and conclusions: The Finnish software companies found the solution with independent classes better than the traditional solution with global variables. Because this question was not included in the survey, no generalisations can be made.

(Q31) What other benefits than those already presented have the companies experienced in information systems development?

Pilot study: In the pilot study it was stated that the object-oriented paradigm is a part of the information systems development technology of today.

Survey: The following answers were reported:

General:

- Because the object-oriented paradigm is commonly accepted and used, this makes for better cooperation both in the home organisation and between organisations.
- The developer has integrated the theory in his thinking model.
- Programmers like the object-oriented paradigm.
- Generally, the possibility to develop systems that are more sophisticated compared with earlier solutions.
- Good uniformity, logical, good structure and lack of faults.
- Makes it easier to divide the working tasks (implementation of components) among the system analysts / programmers.
- Systematic approach, easier to distribute.
- Easy to expand.
- Easy maintenance.
- We still do not have much experience of the object-oriented paradigm. We have no good CASE tools for object-oriented software development.

Programming and design:

- Design & implementation of Model-View-Controller.
- Interfaces, hooks -> runtime switching of functionality.
- Modularity, two answers.
- In order to be able to reuse objects one has to use encapsulation and this has made it easier to be more disciplined; 'taking wrong paths' has become more difficult.
- Information encapsulation.
- One can concentrate better on the task and not on how the task is implemented.
- Patterns (design models); there is a connection between code cases and use cases.
- Possible to write code that is clearer and more intuitive. Management of large code masses becomes easier.
- The development of new objects by inheritance from existing objects.
- The development of unified conceptions is easier. With metaphors one can easier move the logics among persons. Using object-oriented languages makes it possible to use metaphors.
- The smaller amount of code due to the object-oriented paradigm.
- The modelling of the problem domain.
- The presentation of the domain and the analysis is easier with objects than with traditional sequential presentation. One has to remember that it is not feasible to make everything into objects.

- UML Design is more important than code in OOP. Design takes longer time but coding is predictable.

Case studies:

Company A: *Yes, in the traditional way of working, a project has been made for a specific client, and the project has then later on been used as a base for another project for another client. In object-oriented projects this has not been the case; instead the standard library has been used in several projects. The usage of layers and reuse has also been utilized. This makes it possible to start new projects with less work.*

The usage of the object-oriented paradigm also makes the administration of versions easier.

Company B: *In the object-oriented world, the solutions are compound and documentation is more straightforward.*

Company C: *The object-oriented paradigm is acclaimed in the community. If one wants to develop information systems using another paradigm one has to justify for this. The object-oriented paradigm is accepted and has good creditability, which means that there is no need to discuss the choice of technique. There is a remarkable benefit because 'all road-users are driving on the same side of the road'.*

Company D: *Difficult question; actually no, as the total picture I see is that the code is much more understandable, there are less bugs and the maintenance is easier.*

Company E: *No other benefits.*

Company F: *The transition from design into implementation is often easier, because many things are already clear when one starts with implementation.*

Summary of case studies: The companies did not present any real new benefits. The first company presented a benefit that is more a project management issue than an object-oriented issue. The third company presented an issue that has to do with acceptance on the market, which is not an object-oriented issue. The sixth company presented a benefit concerning the transition from design into implementation, but this benefit probably does not have much to do with the object-oriented paradigm per se.

Discussion and conclusions: One can analyse the benefits found in the survey and recognize that most of them are connected to benefits that have already been presented. There are also, however, some more detailed benefits and even some rather 'new' benefits like 'Management of large code masses becomes easier'.

Problems - Complexity

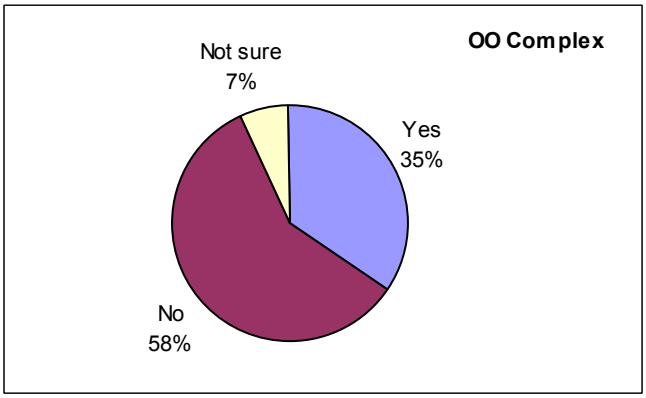
(Q32) Has the object-oriented paradigm been considered complex?

Theory – Studies: According to the findings from the *Survey of Advanced Technology 1996* (Pickering, 1996), the object-oriented paradigm is considered complex. According to the results of 12 empirical studies reported by Johnson (2002) object-oriented design was considered complex.

Pilot study: In the pilot study it was found that it takes a long time to learn the object-oriented paradigm.

Survey: The results are presented in Figure 20.

Figure 20: The object-oriented paradigm is considered complex



In the review of previous studies a possible association was established between difficult object-oriented concepts like reuse (difficulties for information systems developers to learn how a component works) and experienced complexity of the object-oriented paradigm. This association is not very well supported by the survey results. Only about half (43%) of the Finnish software companies that had considered reuse difficult also considered the object-oriented paradigm as complex.

Case studies:

> **Company A:** *Yes, some of the programmers with a background in traditional software engineering have had difficulties in starting to work in the object-oriented way.*
>
> **Company B:** *No, we develop so complex systems that in comparison the object-oriented paradigm cannot be considered complex.*
>
> **Company C:** *The complexity issue has to be compared with how things were done before; if one tries to see everything as it was seen before plus tries to see it in the object-oriented way then the complexity rises dramatically. If one keeps to the object-oriented world from the beginning, then one cannot argue that the object-oriented paradigm is complex.*
>
> **Company D:** *There is a certain obstruction for learning; when this obstruction has been managed then life is easy. Often one ponders why the object-oriented source code is so complicated and why one cannot make it simpler.*
>
> **Company E:** *It depends on the person asked. Most "older" software developers consider the move from traditional information systems development to object-oriented development as a huge step.*
>
> **Company F:** *If the software developer starts to work with the object-oriented paradigm from the beginning then the object-oriented paradigm is not complex. If the software developer moves from the "old" part to the new object-oriented part of software development there might be some hurdles in the beginning, but we have not experienced any real problems.*

Summary of case studies: For an information systems developer with training in the object-oriented paradigm, object-oriented systems development is not complex. Neither is the object-oriented paradigm considered complex if the information systems developer has no earlier experience in traditional information systems development.

Discussion and conclusions: The truth is probably close to findings from the case studies; for an information systems developer with training in the object-oriented paradigm and for an information systems developer with no "burden" of traditional information systems development, object-oriented systems development is not complex. The finding from the pilot study does not necessarily mean that the object-oriented paradigm has been seen as complex, the long time to learn it might be as a consequence of some other reason.

The findings from the empirical parts of this study contradict the proposition found in studies regarding Finnish software companies.

Problems – The object-oriented paradigm is still immature

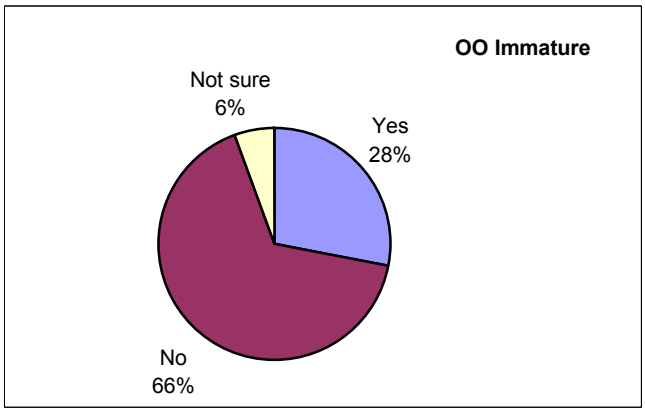
(Q33) Have the companies considered the object-oriented paradigm as being immature?

Theory – Studies: The object-oriented paradigm is still considered immature by some researchers. Object-oriented projects are often criticized as promising too much and delivering too little (Bhattacherjee & Gerlach, 1998).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 21.

Figure 21: The object-oriented paradigm is considered immature



Case studies:

Company A: *No.*

Company B: *It is in a good phase, some nice technical improvements are coming.*

Company C: *There are still some areas in the object-oriented world that are immature; one good example is the connection between relational databases and the object-oriented paradigm.*

Company D: *No, it is not immature according to me.*

> **Company E:** *No, the object-oriented paradigm is not immature. The newer versions of most tools on the market today (from IBM and other companies) support the object-oriented paradigm very well.*
>
> **Company F:** *No it is not immature. Object-oriented modelling is not difficult and we have found most tools we need.*

Summary of case studies: The object-oriented paradigm is not considered immature by the companies, though some areas can be found where the object-oriented paradigm can still be developed, like the connection between the object-oriented paradigm and relational databases.

Discussion and conclusions: A substantial majority of the Finnish software companies were of the opinion that the object-oriented paradigm is not immature.

The findings from the empirical parts of this study contradict the proposition found in studies regarding Finnish software companies. This is probably due to the fact that the previous studies were done in the 1990's and the empirical study was made in 2004. When the previous studies were completed the object-oriented paradigm was still considered immature but today it can be considered as mature.

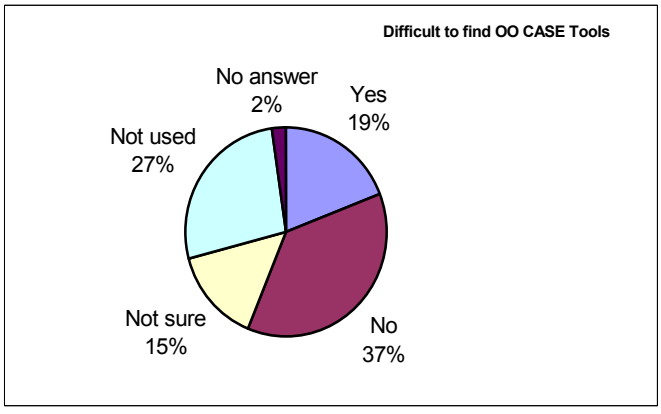
(Q34) Have the companies experienced difficulties in finding object-oriented CASE tools, object-oriented databases, object-oriented system development tools or perhaps even objects to reuse?

Theory – Studies: There is a lack of tools like CASE tools and object-oriented databases that support the object-oriented paradigm, and there is little experience of the tools available. There is also a lack of objects and components to reuse, and companies have to put a great deal of effort in developing objects and libraries that can be reused later on. (Bhattacherjee & Gerlach, 1998; Henders, 1998; LaBoda & Ross 1997; Smith & McKeen, 1996)

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 22, Figure 23, Figure 24 and Figure 25.

Figure 22: Difficulties in finding CASE tools



Comments: One respondent who answered that it is difficult to find CASE tools wrote that one could find them but that most are of poor quality.

Figure 23: Difficulties in finding object-oriented databases

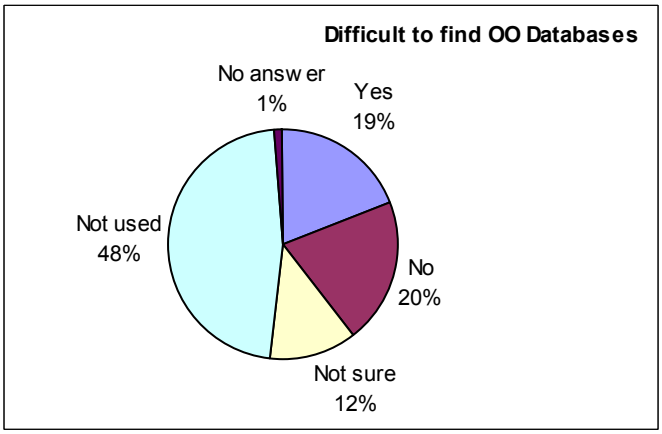


Figure 24: Difficulties in finding object-oriented software development tools

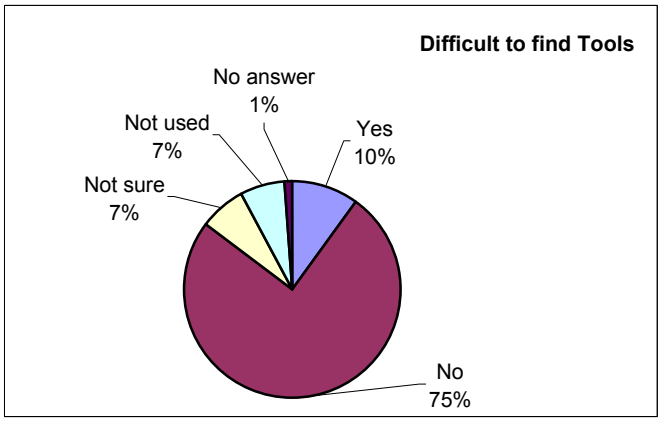
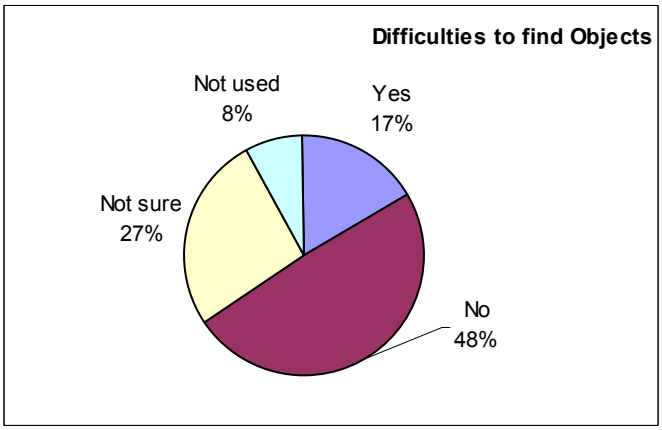


Figure 25: Difficulties in finding reusable objects



Case studies:

> **Company A:** *I would say that in practice, yes. The price is often too high. It is difficult to find affordable tools of good quality. Objects have been used although finding them has not been easy. Databases are also too expensive, we use relational databases, and most clients also want us to use relational databases.*
>
> **Company B:** *Yes, and regarding databases we are still working with compromises; our SQL databases do not work very well with objects, and we have not been working with object-oriented databases because no mature standard can be found.*
>
> **Company C:** *No real problems***.**
>
> **Company D:** *No problems.*
>
> **Company E:** *No problems anymore.*
>
> **Company F:** *No. Some products are expensive.*

Summary of case studies*:* The only problem seems to be the price of some object-oriented products. The availability of object-oriented tools is no problem.

Discussion and conclusions: Most of the companies did not use any object-oriented database. Whether this is due to a possible lack of object-oriented databases is difficult to say, there might be some other reason why the companies do not use any such database. Other object-oriented tools are not difficult to find. Neither is it difficult to find objects.

The findings from the empirical parts of this study contradict the proposition found in the other studies regarding the other issues and Finnish software companies. This is probably due to the evolution of the object-oriented paradigm that is more mature nowadays and therefore tools and reusable objects can be found more easily. Whether there is a good support for object-oriented databases cannot be confirmed because almost half (48%) of the companies did not use any.

Problems – No support for several important areas like testing

(Q35) Have the companies experienced that there are concepts in the object-oriented world that are not well supported?

Theory – Studies: Current systems have little information on object reliability, performance or resource utilisation. In addition, security capabilities are often poor. (Pancake, 1995)

Pilot study: In the pilot study it was found that the object-oriented paradigm does not have year 2000 support. A year 2000 support would have been a special feature of the object-oriented paradigm.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Today the situation is better, but still one can perceive problems like these.*

Company B: *No.*

Company C: *The complexity of the architecture of larger information systems is a problem. Nevertheless, nowadays performance problems and other 'minor' problems are more or less solved.*

Company D: *No.*

Company E: *No.*

Company F: *No.*

Summary of case studies: One company out of six ones answered that the complexity of the architecture of larger information systems is a problem; otherwise no problems were recognized.

Discussion and conclusions: The year 2000 support is not an interesting issue anymore. That the architecture of larger information systems becomes complex is no surprise. One can conclude that there is no substantial lack of anything in the object-oriented world according to five out of six Finnish software companies. Because this question was not included in the survey, no generalisations can be made.

The findings from the empirical parts of this study contradict the proposition found in the other studies regarding Finnish software companies. The object-oriented paradigm is more mature today than it was when the previous studies were completed.

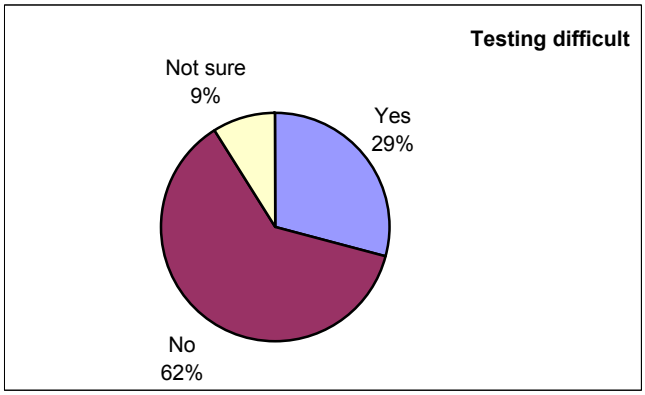
(Q36) Have the companies found testing object-oriented applications or information systems difficult? What testing problems have the companies experienced?

Theory – Studies: There is often little support for testing object-oriented systems in the object-oriented paradigm and in many object-oriented software development methods (Malan et al., 1995). Kung et al. (1995) present major testing problems.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 26.

Figure 26: Object-oriented testing is difficult



	Number of answers
Difficult to test structures, several member functions, call in a chain	23
Difficult to test complex relationships	27
Difficult to test because of few CASE tools for testing	13
Other testing problems	12

Of which the following were given:

Technical:

- The endurance of the load.
- Difficulties in automatically making calls for the compiler -> hard to find the errors.
- Making series of objects when testing distributed systems.

Graphical user interface:

- It is difficult to test the GUI and connections to other systems (this is not necessarily due to the object-oriented paradigm).
- GUI - difficult to "mechanize" the testing.

Programming:

- The same dangers with endless loops are present in the object-oriented world as in the traditional world.
- Finding the problem and slow starting of the application service.

Testing tools:

- There are no automatic object-oriented testing tools.
- Lack of good automatic tools for testing.

General:

- Difficulties with components from a third part, the components do not work as expected (problems with different versions, with dependence of operating systems, bugs).
- Special cases.
- Takes a lot of time.

Comment: One respondent answered that testing problems were much more dependent on language idioms than on object-oriented idioms.

Among the possible associations between problems found in the review of previous studies, the immaturity of the object-oriented paradigm was expected to result in poor support for several areas like testing. This association is, however, poorly supported by the results from this survey because only 40% of the respondents that had considered the object-oriented paradigm as immature were of the opinion that the immaturity had resulted in poor support for information systems development concepts like testing.

Case studies:

Company A: *No, even rather large systems can be tested without any particular difficulties.*

Company B: *Testing never comes automatically; one has to do code for testing and Java is good because one can make testing code faster.*

Company C: *Unit testing has become more difficult; there are good testing tools but the largeness of the application area is challenging when testing.*

Company D: *Testing has not been problematic.*

Company E: *Testing has been more problematic although we have good testing tools. Especially the end users have found the testing of object-oriented information systems as more difficult than testing traditional information systems. The comprehension of the whole information system has often been difficult when testing object-oriented information systems.*

Company F: *Object-oriented systems are more demanding to test. System testing is, however, easier.*

Summary of case studies: Testing is not considered difficult by most of the companies, although one company answered that unit testing is more difficult in object-oriented testing than in traditional testing and another company even reported that testing object-oriented information systems is more difficult than testing traditional information systems.

Discussion and conclusions: Testing object-oriented information systems is not complicated according to most of the Finnish software companies. Among the claimed testing problems, the most frequent was the "It has been difficult to test complex relationships that exist in an object-oriented system" problem.

The findings from the empirical parts of this study contradict the proposition found in studies regarding Finnish software companies. Nowadays the object-oriented paradigm is more mature than it was when previous studies were completed, so there are probably better testing tools on the current market and the software developer also has more experience in object-oriented testing.

Problems – Difficulties in measuring object systems

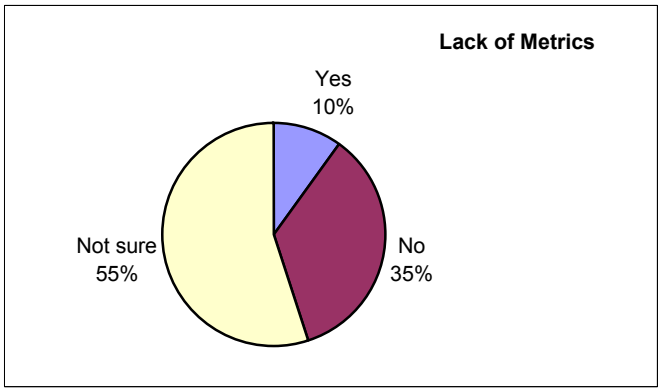
(Q37) Has a lack of metrics for measuring the object-oriented system been considered a problem?

Theory – Studies: According to Pancake (1995) there are no reliable measurement units for predicting progress, assessing productivity and evaluating costs of object-oriented systems. However, researchers like Chidamber & Kemerer (1994) and Henderson-Sellers (1994, Chapter 10) have made considerable contributions to the field of metrics for object-oriented systems.

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 27.

Figure 27: Lack of metrics



Among the possible associations between problems found in the review of previous studies, the immaturity of the object-oriented paradigm was expected to result in difficulties in measuring object-oriented systems. Unfortunately, this association cannot be studied because of the large amount of 'Not sure' answers.

Case studies:

> **Company A:** *No, metrics are not used.*
>
> **Company B:** *No metrics are used. It is very difficult. Experience is more important.*
>
> **Company C:** *We have a doctor's suitcase with metrics; when an information systems development project gets into trouble, we use metrics in order to find the problems so that we then can elucidate them. The metrics are included in the information systems development tools that we use.*
>
> **Company D:** *We use no metrics.*
>
> **Company E:** *We use no metrics, mostly because we do not find metrics reliable due to some earlier experiences on the mainframe side.*
>
> **Company F:** *No comments.*

Summary of case studies: Only one of the companies uses metrics; when solving problems that projects run into.

Discussion and conclusions: The Finnish software companies are not so aware of metrics. Probably this issue is too theoretical.

Problems – Training & lack of experience

(Q38) Has the software company been using a mentor in order to solve the problem with training of the software developers?

Theory – Studies: Using a mentor is recommended by, for example, Eriksson & Penker (1996, pp. 183-184) and Henderson-Sellers & Edwards (1994, p. 426).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No, we have only been using support people when inspecting programming code.*
>
> **Company B:** *No.*
>
> **Company C:** *Actually yes, we have continuous mentoring both internally and externally for our customers.*
>
> **Company D:** *No, we use Google; one can find a lot of information on the Internet as long as one knows what one is looking for.*
>
> **Company E:** *We use mentors; both from our company and mentors (consults) from other companies.*

Company F: *Our programmers are very active and enterprising and take full responsibility for their own progress. However, we have offered some training and education.*

Summary of case studies: Only one of the companies uses mentors. However, another of the companies uses support people to help customers, who are mentors in a way. Moreover, another of the companies uses support people to help information systems developers; and one even uses Google as a 'mentor'.

Discussion and conclusions: The Finnish software companies do not use genuine mentors, but some kind of support exists. Because this question was not included in the survey, no generalisations can be made.

(Q39) Has there been a resistance to learning the object-oriented paradigm because there is such a huge paradigm shift between the traditional functional paradigm and the object-oriented paradigm?

Theory – Studies: This might be the case according to Pancake (1995).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No, problems have been more organizational; the managers of object-oriented projects have not been able to induce the old programmers. Nevertheless, most old experts ('gurus') have started to work with the object-oriented paradigm with enthusiasm.*

Company B: *No.*

Company C: *Some resistance has been recognized. The step from traditional information systems development to object-oriented information systems development has been surprisingly large for some information systems developers. My understanding is that it takes about one year to move from traditional information systems development to object-oriented information systems development. It is not very difficult to learn Java, but it is significantly more difficult to master object-oriented programming and be productive.*

Company D: *Yes, but we have only one software developer that does not want to move to the object-oriented world.*

Company E: *Yes, we have a lot of "older" software developers who do not want to switch from traditional information systems development into object-oriented information systems development. The average age of our software developers is rather high, over 40, and this probably explains why we have this problem.*

Company F: *No.*

Summary of case studies: This problem is recognized by the companies, but not considered a major problem because in most companies (company E is an exception) there are only a few persons unwilling to move from traditional information systems development to object-oriented information systems development.

Discussion and conclusions: There seem to be only a few information systems developers that are dedicated to some older software paradigm left in the Finnish software companies who do not want to move into new information systems

development and programming paradigms. Because this question was not included in the survey, no generalisations can be made.

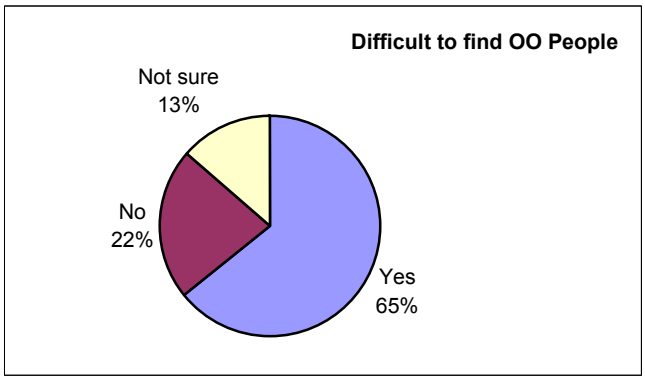
(Q40) Has it been difficult to find experienced object-oriented software developers and system analysts?

Theory – Studies: According to Noack & Schienmann (1999) and Radin (1996) it might be difficult to find experienced object-oriented information systems developers.

Pilot study: In the pilot study it was found that learning, training and experience regarding the object-oriented paradigm is important.

Survey: The results are presented in Figure 28.

Figure 28: Difficult to find people that know the object-oriented paradigm



In the review of previous studies a possible association between the immaturity of the object-oriented paradigm and the difficulties in finding software developers trained in the object-oriented paradigm was presented. This association is well supported by the results from the survey as 80% of the respondents who considered the object-oriented paradigm as immature had also had difficulties in finding software developers trained in the object-oriented paradigm.

In the review a possible association between the considered complexity of the object-oriented paradigm and the difficulties in finding software developers trained in the object-oriented paradigm was presented. This association is also well supported by the results from the survey as 78% of the respondents who considered the object-oriented paradigm as complex had also had difficulties in finding software developers trained in the object-oriented paradigm.

Case studies:

Company A: *No.*

Company B: *No, but differs from time to time. School only gives the basics.*

Company C: *No.*

> **Company D:** *No. A skilful programmer can also learn new programming languages and even object-oriented programming easily without formal education. The problem is more how to find skilful programmers. Students, who have studying programming and graduate, often have poor knowledge of programming if they have not had any experience before they start working.*
>
> **Company E:** *Today one can find trained software developers with knowledge of object-oriented information systems development issues. The situation has changed very much in recent years.*
>
> **Company F:** *No.*

Summary of case studies: This is not a problem according to the companies. What is interesting is the statement from one company that a skilful programmer can easily move from one programming language to another, even if there is a switch from one information systems development paradigm to another information systems development paradigm.

Discussion and conclusions: In the survey, but not in the case studies, it was found that it is difficult to find experienced object-oriented software developers and system analysts. It is worthy of note that the results from the survey are different from the results from the case studies. However, because the results from the survey can be generalized, the findings there are somewhat more interesting.

The findings from the survey in this study are in correspondence with the proposition found in the previous studies.

Problems – Efficiency

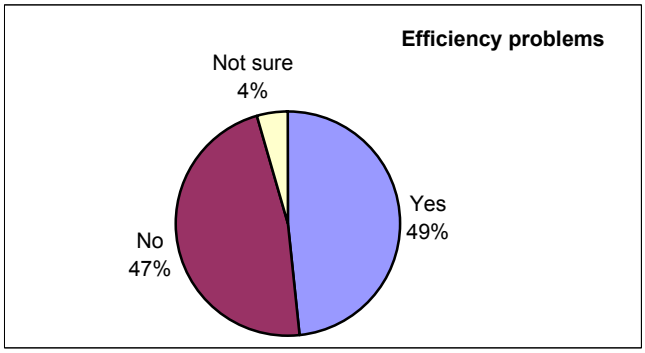
(Q41) Have the companies experienced computer efficiency problems in their object-oriented software development projects?

Theory – Studies: There are often problems with computer efficiency because some object-oriented designing takes up a lot of computer processing time (Booch, 1994, pp. 288-289).

Pilot study: In the pilot study it was found that there are not enough benefits/efficiency in the object-oriented paradigm.

Survey: The results are presented in Figure 29.

Figure 29: Experienced computer efficiency problems



Comment: One respondent who answered, "Yes," wrote that Java is a rather low-level language.

In the review of previous studies a possible association between the considered immaturity of the object-oriented paradigm and experienced efficiency problems was presented. This association is supported by a small majority (56%) of the respondents in the survey who considered the object-oriented paradigm as immature.

Case studies:

Company A: *Yes, to some extent.*

Company B: *Yes, but these kinds of problems have also been found in software that has been developed with traditional tools.*

Company C: *Yes, we have experienced efficiency problems that we have been working with on the framework level. We mostly develop large commercial information systems, and efficiency problems are often connected to the selection of data from very large databases. When these problems are handled then the efficiency problems also diminish.*

Company D: *No. However, when we have developed object-oriented software we have been working, in particular, with efficiency issues. If the object-oriented paradigm is used wrongly, the information systems become slower than traditional systems. In addition, the fragmentation of memory has been problematic sometimes.*

Company E: *No.*

Company F: *No.*

Summary of case studies: If not developed correctly it seems that object-oriented information systems have computer efficiency problems.

Discussion and conclusions: The results from the pilot study have to be omitted because they are too common. Probably the truth is somewhere in the direction that was identified in the case studies; if not developed correctly it seems that object-oriented information systems often have computer efficiency problems. The computer efficiency issue is probably also associated with the type of information system being developed.

(Q42) If there has been no suitable collection of objects to reuse, has it influenced the object-oriented development project efficiency in a negative way?

Theory – Studies: Project efficiency is discussed by Page-Jones (1992b) who warns about starting to use the object-oriented paradigm if effective information systems software development is desired and there is no suitable repository with objects for reuse available.

Pilot study: In the pilot study it was found that there are not enough benefits/efficiency in the object-oriented paradigm.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No, we have built our own objects when suitable objects have not been found.*
>
> **Company B:** *Yes, one has to consider the resources when starting a new project.*
>
> **Company C:** *Yes, we have very few objects that we reuse.*
>
> **Company D:** *We have not experienced any problems in finding objects to reuse.*
>
> **Company E:** *No comments.*
>
> **Company F:** *No comments.*

Summary of case studies: This question was difficult to understand for the companies, mostly because of the part about project efficiency. Probably project efficiency is often difficult to measure because one ought to compare projects that might be very different from each other. The companies were therefore more interested in talking about the availability of objects for reuse.

Discussion and conclusions: Once again the results from the pilot study have to be omitted because they are too common. No conclusions can be made. The question was only included in the case studies, and unfortunately the persons interviewed in all six Finnish software companies had no substantial experience of project efficiency.

Problems – Costs

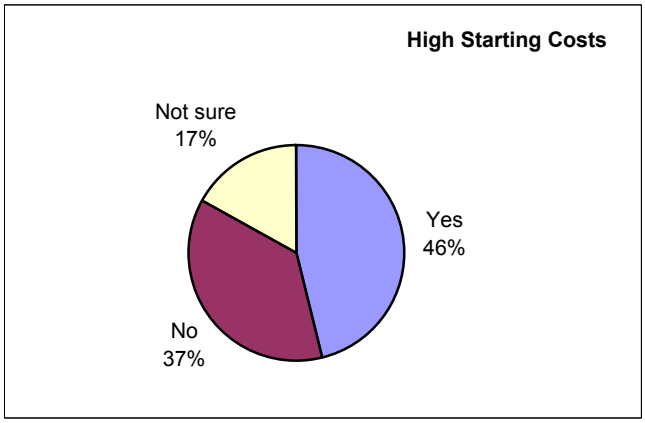
(Q43) Have the starting costs been high when launching a completely new object-oriented information system development project, due to a lack of artefacts to reuse?

Theory – Studies: The starting costs are often huge when one begins a new object-oriented information system project because there is nothing to reuse and everything has to be developed from scratch (Booch, 1994, pp. 288-289).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 30.

Figure 30: Experienced high starting costs



Comment: One respondent who answered "No" wrote that there are almost too many reusable parts.

In the review of previous studies there was a proposed association between the considered immaturity of the object-oriented paradigm and the experienced high starting costs. This is also the case according to a majority (64%) of the respondents who considered the object-oriented paradigm as immature.

Case studies:

Company A: *It is always important to manage costs, but starting costs are difficult to measure.*

Company B: *No comments.*

Company C: W*e have the application framework Open Frame and the building of this product has been very resource consuming and costly. This product makes software development more efficient, but we have to build many information systems before we get the invested money back.*

Company D: *We do not record how much it costs to start different projects.*

Company E: *No comments.*

Company F: *Starting costs have been high.*

Summary of case studies: Because the companies do no genuine recording of their starting costs this question was difficult to answer.

Discussion and conclusions: It seems that starting costs are high when launching a completely new object-oriented information system project, due to a lack of artefacts to reuse.

The findings from the empirical parts of this study are in correspondence with the proposition found in the previous studies.

Problems – Limited usability of components

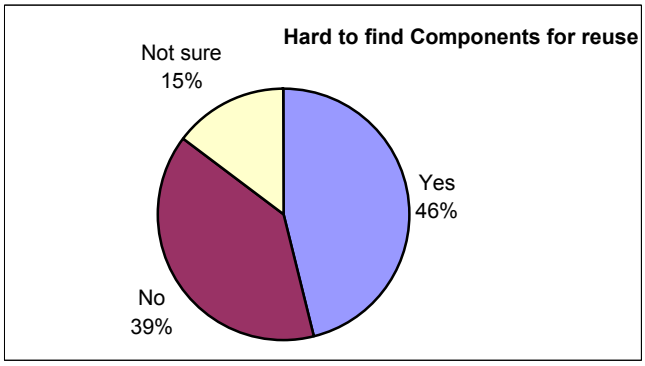
(Q44) Has the company had problems finding components to reuse?

Theory – Studies: Finding the components to reuse is a serious problem in many object-oriented projects. The usability of the components has to be good too, and for example, banks nowadays are defining usable standard business components. Nevertheless, it might still be difficult to find good components to reuse. (Radin, 1996)

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 31.

Figure 31: Hard to find components to reuse



In previous studies the considered immaturity of the object-oriented paradigm was estimated to result in experienced difficulties in finding components for reuse. This is also the case according to a clear majority (68%) of the Finnish software companies that considered the object-oriented paradigm as immature.

Case studies:

Company A: *No.*

Company B: *Sometimes.*

Company C: *Yes, see earlier questions.*

Company D: *No.*

Company E: *Sometimes.*

Company F: *No.*

Summary of case studies: Some companies have problems in finding components and others have no problems finding components to reuse. Of course, this is also something that has to do with the management of components in the company in question.

Discussion and conclusions: Probably the reality is close to the findings from the case studies; some Finnish software companies have had problems in finding components and some others have had no problems finding components to reuse. This has probably to do with the management of components in the software company in question. Another issue that has to be taken into consideration is the length of time a company has been involved in object-oriented information systems development. The longer the time the smaller the problems in finding components for reuse probably are.

(Q45) Has there been a problem in managing the different versions of a component?

Theory – Studies: An important problem with components is when there are several different versions of one component (Jarzabek & Knauber, 1999).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *No, we do not use different versions of objects.*
>
> **Company B:** *No.*
>
> **Company C:** *This issue has been handled with traditional methods. Earlier we had some problems with different versions of DLL, but not anymore.*
>
> **Company D:** *No.*
>
> **Company E:** *We have a very good version management system. We have no problems in managing different versions of components.*
>
> **Company F:** *No.*

Summary of case studies: The management of versions of a component is no problem.

Discussion and conclusions: The management of versions of a component is probably not a problem. Because this question was not included in the survey, no generalisations can be made.

Problems – Problems with reuse

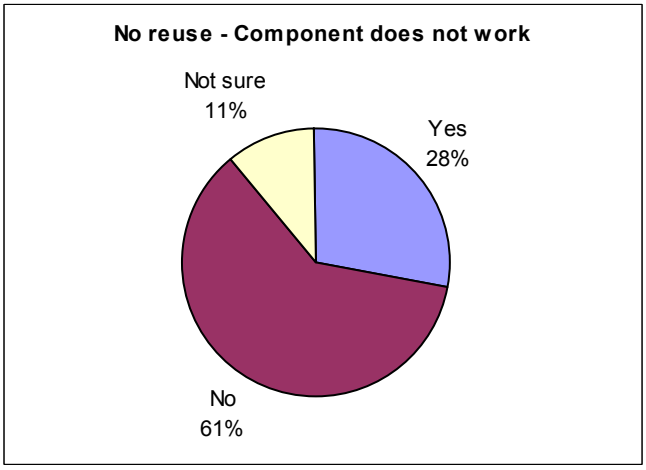
(Q46) Has there been a problem with reuse in the sense that software developers do not want to reuse a component, because they claim that it does not work, or it is too troublesome to learn how the component works?

Theory – Studies: According to Nokso-Koivisto (1995) and Radding (1999) system developers often avoid reusing existing modules, because they claim that the modules 'do not work anyway' or 'it is not worth the effort to figure out what the module (component) does and how it works'.

Pilot study: The question was not included in the pilot study.

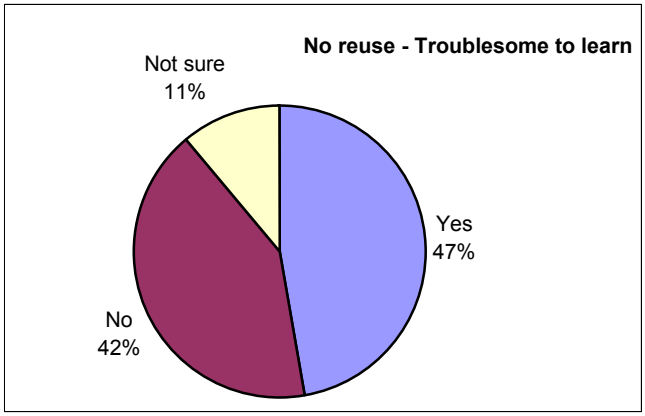
Survey: The answers are presented in Figure 32 and Figure 33.

Figure 32: No reuse because components do not work



Comment: One respondent who answered "Yes" wrote that the reason was that developers often do not "approve" others' code.

Figure 33: No reuse because troublesome to learn how a component works



Comment: One respondent who answered "Yes" wrote that it was, however, not troublesome if the source code and/or interface presentation is available.

In the review of previous studies the considered immaturity of the object-oriented paradigm was expected to result in experienced problems with reuse (that were connected to difficulties for information systems developers in learning how a component works). This is the case according 50% of the Finnish software companies that considered the object-oriented paradigm as immature.

Case studies:

Company A: *Yes, see earlier answers.*

Company B: *No.*

Company C: *Yes, see earlier answers.*

Company D: *No.*

Company E: *Yes.*

Company F: *No.*

Summary of case studies: Some of the companies have experienced this problem while others have not.

Discussion and conclusions: It seems that Finnish software companies do not have a problem with software developers not wanting to reuse a component because they feel that it does not work.

The findings from the empirical parts of this study contradict the proposition found in studies regarding Finnish software companies. There are probably a lot of 'approved' components on the market today and not using such components would be discomforting for an information systems developer.

A small majority of the Finnish software companies were of the opinion that it is troublesome to learn how components work; because of this one cannot conclude if the findings from the empirical study are in correspondence with the findings from the review of previous studies.

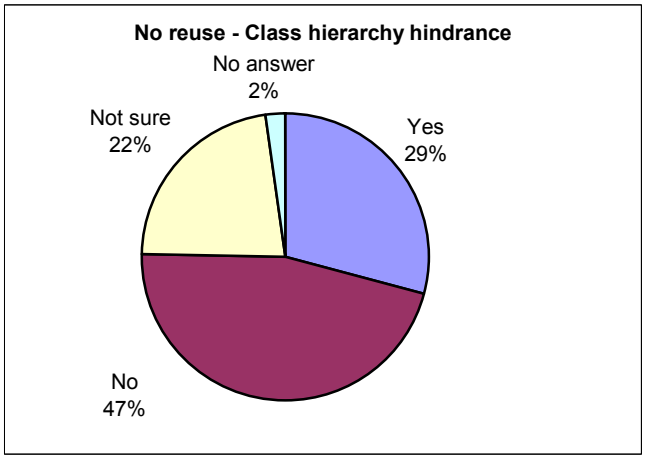
(Q47) Has the hierarchy of classes been a hindrance for reuse?

Theory – Studies: If a programmer needs a simple class that is down in the hierarchy and has several superclasses, then he or she might get a lot of unnecessary classes and code when taking in the whole hierarchy in the program just to get one class. Further, the hierarchy in a class library can be difficult to integrate into the existing class hierarchies in the software company (Eriksson, 1992, p. 356; Wrede, 1998).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 34.

Figure 34: No reuse because of class hierarchy



Case studies:

Company A: *Perhaps in debugging.*

Company B: *No, but deep hierarchies might have an affect on the efficiency and maintainability of the application.*

Company C: *No. On the framework side there are usually three levels in a hierarchy, and on the application side where these framework levels are used, there are at most two levels using the framework. However, on the application side there might be more levels in a hierarchy but it has not been a hindrance for reuse as long as the hierarchy does not become too deep.*

Company D: *If the hierarchy is very deep then one usually experiences problems. In debugging the hierarchies are sometimes difficult to visualize.*

Company E: *We have little experience of this problem, mostly because we do not use that much reuse.*

Company F: *We have hierarchies but we have not had any real problems with them or with reuse.*

Summary of case studies: The hierarchy of classes is no hindrance for reuse. What is interesting, however, is that in debugging the hierarchy of classes might be a problem.

Discussion and conclusions: The hierarchy of classes is probably no obstruction for reuse although as many as 29% of the Finnish software companies in the survey actually were of this opinion.

The findings from the empirical parts of this study contradict the proposition found in studies regarding Finnish software companies.

Problems – Problems with object-oriented analysis

(Q48) Has there been a problem with analysis when object-oriented analysis has been used?

Theory – Studies: According to Höydalsvik & Sindre (1993) object-oriented analysis does not fulfil the purposes of analysis. If this is true there ought to be documented problems with object-oriented analysis.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *Yes, it has been demanding to perform object-oriented analysis.*

Company B: *No.*

Company C*: No.*

Company D: *We carry out no real analysis.*

Company E: *No.*

Company F: *No.*

Summary of case studies: Object-oriented analysis is not problematic to perform. One company answered that it is demanding.

Discussion and conclusions: Object-oriented analysis is not problematic to perform according to the Finnish software companies that took part in the case studies. However, one cannot generalize here.

(Q49) Has object-oriented analysis been a good choice if the system that is to be developed has limited responsibilities, or it is a system with few classes (< 10) and objects?

Theory – Studies: According to Coad & Yourdon (1990, p. 32) this would not be the case.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *Principally one can develop such systems just as well with both the object-oriented paradigm as with traditional approaches, but we are moving into the object-oriented paradigm for all kinds of applications in order to get a uniform way of developing systems.*
>
> **Company B:** *No answer (lack of time).*
>
> **Company C:** *No answer (lack of time).*
>
> **Company D:** *We perform no real analysis.*
>
> **Company E:** *Yes, we use UML for all kinds of systems.*
>
> **Company F:** *Differs from case to case.*

Summary of case studies: This question cannot be answered because only one company answered this question clearly.

Discussion and conclusions: No conclusions can be made**.**

(Q50) Has the software company experienced one or several of the following problems with object-oriented information systems development in the analysis phase?

The problems are presented in the case study section.

Theory – Studies: Aksit & Bergmans (1992) found these obstacles in object-oriented software development. The problems related to preparatory work.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

a) Identification of Problem-Domain Structures has been difficult. It might often be difficult to identify classifications in the problem domain that could be mapped to inheritance hierarchies.

Company A: *Yes.*

Company B: *No answer (lack of time).*

Company C: *Finding the right structure is not easy. Using an object-oriented model might even have the result that one starts talking the wrong path. For example, one can find actual data and historical data; in the object-oriented data model these data are the same objects, but for the information system these data are of course very different.*

Company D: *No.*

Company E: *No.*

Company F: *No. Our programmers are experienced.*

Summary of case studies*:* This was considered a problem by two of the companies. The hierarchies are probably difficult to develop when working with very large information systems development projects.

b) Dealing with Excessive Domain Objects has been difficult. Integrating the domain knowledge with the user's requirement specifications can yield a lot of objects. Only few of these objects may be relevant to the problem area.

Company A: *No.*

Company B: *No answer (lack of time).*

Company C: *No, experienced information systems developers can manage a large number of objects in the analysis phase of the information systems development project. The "trash" objects can usually be found without problems.*

Company D: *No.*

Company E: *Yes, we have a problem when we do not know where objects should be, who should pay for them and who should maintain them, etc.*

Company F: *It is a small problem.*

Summary of case studies*:* Only one company recognized this as a real problem. Another company considered it as minor.

c) Problems with Early Decomposition. If subsystems are not identified before objects are identified problems might arise, because objects have to be placed into some subsystem when identified. If the subsystems are identified before object identification, the boundaries of the subsystems may not be optimal.

Company A: *No idea.*

Company B and C: *No answer (lack of time).*

Company D: *No.*

Company E: *See answer to question 50 b.*

Company F: *It is challenging but not problematic.*

Summary of case studies*:* Only one company recognized this as a problem (company E). Another company recognized the issue but did not recognize it as a problem.

d) Subsystem-Object Distinction has been difficult. In the analysis phase objects may act as subsystems if they are complicated. Subsystems can also be defined as objects if they can be structured in a hierarchy and reused.

Company A: *No idea.*

Company B and C: *No answer (lack of time).*

Company D: *No.*

Company E: *No idea.*

Company F: *No.*

Summary of case studies*:* This issue has probably not been a problem, though only two of the companies answered the question.

e) Problems with Commonality versus Partitioning. Because subsystems partition the system, classes that are members of the same hierarchy can be spread over several subsystems. Finding the appropriate inheritance hierarchies becomes difficult.

Company A: *No idea.*

Company B and C: *No answer (lack of time).*

Company D: *No.*

Company E: *No idea.*

Company F: *No idea.*

Summary of case studies*:* This issue cannot be discussed because none of the companies answered the question.

f) Subsystems Identification Using Object Interactions has been problematic. Subsystems are often used for structuring interactions among objects; however, most object-oriented methods only have intuitive techniques for subsystem identification.

Company A: *No idea.*

Company B and C: *No answer (lack of time).*

Company D: *No.*

Company E: *No idea.*

Company F: *No idea.*

Summary of case studies*:* This issue cannot properly be discussed because only one of the companies answered the question.

Discussion and conclusions: As can be read from the results of the case studies, only the first problem was recognized by two of the Finnish software companies.

Problems – Problems with object-oriented design

(Q51) Has the transition from object-oriented analysis to object-oriented design been easy or difficult?

Theory – Studies: According to Höydalsvik & Sindre (1993) object-oriented analysis has no smooth transition to design.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

> **Company A:** *The same.*
>
> **Company B:** *No answer (lack of time).*
>
> **Company C:** *It has been difficult; often the design phase becomes too short. It has even happened that the information systems developers skipped the design phase and started with programming right after the analysis phase. The larger the information systems development project is, the more important it is to design well.*
>
> **Company D:** *Because no analysis is carried out, this question is not relevant.*
>
> **Company E:** *Yes, but with growing experience this problem becomes smaller. One has to understand the issue of iteration appropriately in order to move properly from analysis to design and back.*
>
> **Company F:** *No. It depends, however, on the software developer.*

Summary of case studies: Only one of the companies considered the transition from object-oriented analysis to object-oriented design as difficult. Another company found this issue problematic in the first object-oriented information systems development projects but not later on.

Discussion and conclusions: One can argue that the transition from object-oriented analysis to object-oriented design has been easy. Because this question was not included in the survey, no generalisations can be made.

(Q52) If the transition from object-oriented analysis to object-oriented design has been difficult, why has it been?

Theory – Studies: Out of the theory and review of previous studies the following answers were expected:

- Difficulties in connecting concepts found in object-oriented analysis with concepts in object-oriented design.
- Problems with this issue in the chosen object-oriented information systems development method.
- Object-oriented analysis was poorly performed because it was difficult.
- Object-oriented analysis was poorly carried out because the object-oriented analysis method was insufficient.

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No comments.*

Company B: *No answer (lack of time).*

Company C: *Usually the problems are connected to the fact that in the design phase there are several unique objects that are technical. Therefore there are more objects in design than in analysis. For the tracing of the paths of objects this issue is problematic.*

Company D: *Because no analysis is performed this question is not relevant.*

Company E: *The utilization of the analysis in the design phase has been problematic.*

Company F: *No comments.*

Summary of case studies: One of the companies considered technical objects as a problem; these objects occur in design but are not present in analysis.

Discussion and conclusions: Because the transition from object-oriented analysis to object-oriented design has been easy this matter is of minor interest for this study. Because this question was not included in the survey, no generalisations can be made.

Problems – Lack of object-oriented databases and common interfaces

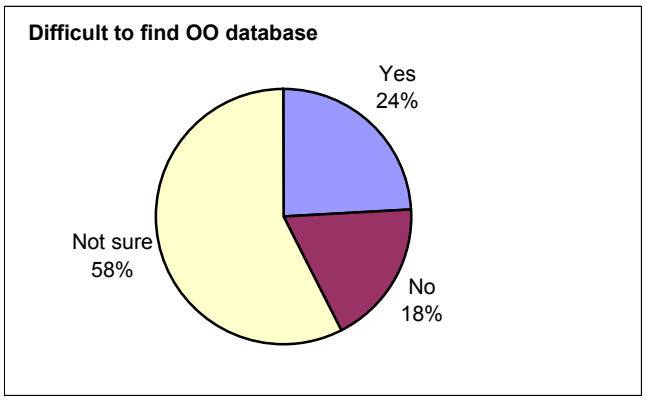
(Q53) Has it been difficult to find an appropriate object-oriented database?

Theory – Studies: Unavailability of adequate object-oriented database systems is usually a problem according to Johnson (2000).

Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 35.

Figure 35: Difficult to find object-oriented databases



Comments: One respondent who answered 'No' wrote that they have not used object-oriented databases because relational databases are the 'de facto' standard. One respondent who answered 'Not sure' wrote that they do not use object-oriented databases.

In the review of previous studies the considered immaturity of the object-oriented paradigm was expected to result in experienced difficulties in finding an object-oriented database. Because most of the Finnish software companies (58%) were not sure about this connection, no conclusions can be made.

Case studies:

Company A: *Yes.*

Company B: *Yes.*

Company C: *Yes, the object-oriented databases have not become commonly used. We have been evaluating Gemstone but not found it very suitable for our needs.*

Company D: *We use no object-oriented databases. We use file systems.*

Company E: *We do not use object-oriented databases, as we are not interested in them.*

Company F: *We use no pure object-oriented databases.*

Summary of case studies: The lack of appropriate object-oriented databases is considered a problem by two of the companies.

Discussion and conclusions: Because most of the Finnish software companies did not use any object-oriented databases, the number of "not sure" answers was high. Among those companies that used object-oriented databases, the lack of appropriate object-oriented databases was considered a problem by a small majority of the companies.

The findings from the empirical parts of this study are somewhat in correspondence with the proposition found in the previous studies.

(Q54) If a relational database has been used in the object-oriented system development work, which approach for connecting the object-oriented system with the relational database has been used?

Theory – Studies: Out of the theory and review of previous studies the following answers were expected:

- The solution of mapping a class to a table has been used.
- A solution with factory classes has been used.
- The Strix object persistence engine has been used.
- SMRC as presented by Reinwald et al. (1996) has been used.

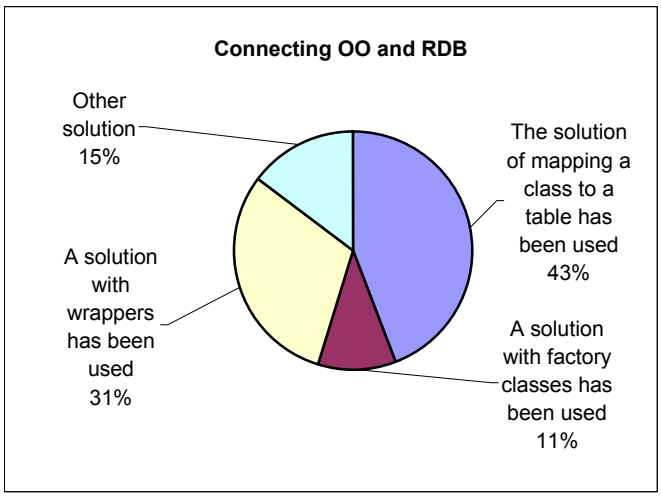
Pilot study: The question was not included in the pilot study.

Survey: The results are presented in Figure 36. The other solutions were the following:

- Own object-oriented database.
- Result of own development -> 1 object - N tables, development of the value – object.
- Object-Relational Mapping Frameworks (like Hibernate).

- Solution developed in company using Java and the "making of series"" of the database.
- Top link.
- Different solutions in different cases; two answers.
- Case tool takes care of it.
- JDBC.
- Object - Relational Mapping.
- OR - Mapping Layer (= a layer with persistence support classes and mappers).
- Other solution; two answers.
- SQL calls.

Figure 36: How to connect the object-oriented paradigm and RDB



Case studies:

Company A: *Another solution; we have not mapped the relational database with the object-oriented system.*

Company B: *Another solution; there are ready-made solutions based on object relational mapping that we have been using, but we have modified a solution for our needs.*

Company C: *Different solutions and mapping have been used; the Open Framework has a solution of its own; we have also used external frameworks, etc. When building simple information systems the solution of mapping a class to a table is a good one. The Strix object persistence engine has also been used.*

Company D: *We use file systems.*

Company E: *We use table specific procedures.*

Company F: *The solution of mapping a class to a table has been used. We have also used some other solutions.*

Summary of case studies: The solution of mapping a class to a table was considered a good one when working with small object-oriented information systems. One of the companies used the Strix object persistence engine. However, all the companies used other solutions than the ones mentioned in the question.

Discussion and conclusions: The solution of mapping a class to a table was the most used, followed by the solution with wrappers. The solution with factory classes had also

been used. Notable is that many of the Finnish software companies also had solutions of their own.

(Q55) Has the lack of a common interface for ad hoc queries been considered a problem when using pure object-oriented databases?

Theory – Studies: When working with most pure object-oriented oriented databases everything is encapsulated and therefore ad hoc queries through a common interface like SQLCI cannot be made (Ooil, 2002).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No answer (lack of time).*

Company B: *We use relational databases so one can make ad hoc queries.*

Company C: *We use relational databases and we avoid making it possible for users to do ad hoc queries.*

Company D: *No.*

Company E: *No.*

Company F: *No.*

Summary of case studies: Because none of the companies used pure object-oriented databases this is not a valid question.

Discussion and conclusions: This problem was recognized by some of the Finnish software companies that took part in the case studies, but it was of minor interest and because none of the companies used pure object-oriented databases, the question cannot be answered. No generalisations can be made.

Problems - Other

(Q56) Have there been difficulties in mixing classes developed in different object-oriented programming languages or produced by different vendors?

Theory – Studies: It is usually difficult to mix classes that have been developed in different programming languages or by different vendors, according to Lam (1997).

Pilot study: The question was not included in the pilot study.

Survey: The question was not included in the survey.

Case studies:

Company A: *No experience.*

Company B: *We do not mix components developed in different programming languages.*

Company C: *Yes, sometimes we have written the application code in Java and more hardware specific functions in C++. The integration of modules written in different programming languages is always difficult and has to be solved from case to case.*

Company D: *No.*

Company E: *We have experts who can easily solve problems like these.*

Company F: *We try not to mix classes developed in different programming languages. When connecting older classes with .NET, we have sometimes met this problem.*

Summary of case studies: One of the companies had experience of this and considered it challenging. Another company had also experienced it, but it was no problem because the experts there could easily deal with this kind of difficulty.

Discussion and conclusions: This is almost certainly done rather seldom but is probably demanding when it is done. Because this question was not included in the survey, no generalisations can be made.

(Q57) What other problems or obstacles of the object-oriented paradigm, other than those presented, have the Finnish software companies experienced?

Theory – Studies: Are some of the problems or obstacles the same as those presented by, for example, Taylor (1990, pp. 108-113) or Pancake (1995)?

Pilot study: The question was not included in the pilot study.

Survey: In the survey this question was divided into two parts (A and B). Not all 104 companies gave an answer. When more than one company gave the same answer this is noted after the answer.

A. Have you experienced a lack of support for any concepts in the object-oriented world? The results were the following:

Positive:

- Certainly nothing important is missing but there is some need for improvement.
- No; three answers.
- Don't know.

Tools:

- Lack of good automatic testing tool that is integrated with the design and would make it possible to systematically test the application and generate test cases.
- Lack of ready object-oriented and cheap tools that small companies can use.
- Lack of testing tools and testing processes (customs). A tool and process for connecting design, implementation, testing and documentation.
- More products and information.
- MS SQL relational object root hybrid. Not too expensive design tool that can be used with our IDE.
- Shortages in the design and modelling tools for embedded systems, problems to model parallel tasks.
- Commercial tools and components are expensive. The cheaper alternatives usually have shortages and have a poorer usability.
- UML is not suitable for the communication between software developers and end users.

- CASE tools.
- Programming languages and tools that increase productivity and quality. In this field, we are still in the early stages.

Documentation:

- The documentation of classes and solutions in IDE is often not sufficiently connected to reality.
- The documentation is too schematic or too simple.
- No systematic approach for documentation.

Programming and design:

- The move from functional/data oriented programming to the object-oriented paradigm has been hard.
- As a result, we have object-oriented programs that are more functional than object-oriented.
- Good class hierarchy: usually 3-4 iterations before one can get it right.
- Hard to build components in some circumstances.
- It should be possible to generate the interfaces of the database classes more easily.
- Natural division of work between objects and OODB – in which level should the business logic be situated?
- Not from the object-oriented world but from the implementation of object-oriented programming languages.
- New modelling of user interfaces.

General:

- The lack of understanding and skill to compare software systems and architectures among the managers.
- Lack of clear guidelines.
- Lack of good and skilful developers.
- Object oriented databases.
- Silver bullet.

B. What other problems or obstacles of the object-oriented paradigm, other than those presented, have you experienced? The results were the following:

Positive:

- No problems.

Programming and design:

- Bugs and strange behaviour in tools and classes in IDE. One has to use special solutions.
- Compulsory class hierarchy; should be: 'object has these features' not 'class has these features'.
- Should reflect the real world better; for example, it should be possible to give 'value €' though the object has not inherited the 'value' class.
- Hard to present and design the sequence of the messages.
- Large systems have so large libraries that often one builds one's own components though ready-made components are available. This usually happens when the original model ages and one is building new things in the model, then old objects in the model have to be updated and the functionality of the objects changes.
- MS XML -> Sun XML move.
- Object-oriented development gives a picture view of the problem that the software developer has in his mind. Often one thinks in a too complex manner, and one developer has difficulties in understanding the picture view of another developer

- One can write "sausage" code with object-oriented tools like Java.
- Problems when developing real-time systems (especially when developing with Java).
- Problems with the usage of objects that are only needed in certain circumstances.
- The development and evaluation of components takes a lot of time (and costs).

Databases:

- The object-oriented databases are still in the early stages compared with relational databases.
- Often the information behind objects is not sufficient. Often traditional relational tables are needed in order to integrate the application with other applications.

Tools:

- The object-oriented tools have shortages. Testing could be improved.
- The prices of the tools are usually too high for our small company.
- High prices on system tools and object-oriented tools. The object-oriented world has a rather good theoretical background, but there is a lack of CASE tools.

General:

- Customers do not always like it because of its complexity.
- First it was difficult to come into the 'object world thinking'.
- Many think that the object-oriented paradigm is something special
- Mostly the limitations of one's own knowledge. We have used the object-oriented paradigm very little.
- Normal attitude problems that can be solved.
- Resistance against change to the object-oriented paradigm.
- Slow application service. The difficulties in connecting different application services.
- The change in thinking is the most important challenge in object oriented thinking for the developer.
- The object-oriented paradigm is not a "miracle medicine"; with the object-oriented paradigm one can develop just as bad or as good solutions as without it. All depends on the working skills of the developer.
- The shortages of applications that have been developed by non-professional developers.

Case studies:

Company A: *In the Microsoft environment one has to use too many tools, J#, C# and Visual Basic; this is too much. Programming in XML has been problematic.*

Company B: *People who have not been working in large object-oriented projects often forget that memory management must still be performed.*

Company C: *No, the problems can be divided into those that are connected to knowledge and experience of the object-oriented paradigm, those that are technical, those that are due to project management flaws and those that are industrial; nowadays one has to produce information systems faster and faster and the customers are more and more demanding.*

Company D: *No other problems.*

Company E: *No other problems.*

Company F: *No other problems.*

Summary of case studies: Some of the six companies mentioned problems like 'too many tools to manage', memory management and a lack of knowledge of the object-

oriented paradigm. No real problems connected to the object-oriented paradigm were presented.

Discussion and conclusions: In the survey many problems connected with the object-oriented paradigm were presented. Most of the problems were, however, variations of the problems with the object-oriented paradigm that are presented earlier in this study. No generalisations can be made.

5 RESULTS AND ANALYSIS

This chapter first summarises the major findings regarding the benefits and problems with the object-oriented paradigm. Then the major empirical findings are discussed with a focus on the explanation of the results.

5.1 Summary of empirical findings

When reading the summary one should note that the companies that did not answer the survey question are presented in a "no answer" category. The population in the survey is therefore always the 89 companies that use the object-oriented paradigm in information systems development.

Benefits. The empirical study showed that most of the Finnish software companies are very positive towards the object-oriented paradigm. One has, however, to take into consideration the risk that companies that are more positive towards the object-oriented paradigm also might be more willing to answer the survey. The response rate of 13,2% was not very high.

A substantial majority of the Finnish software companies had experienced the following benefits. The percentage figure in parentheses is the "yes" category.

1. The object-oriented paradigm is useful when developing large-scale and complex information systems (94%).
2. Reuse is beneficial (92%).
3. The quality of object-oriented systems is better than the quality of traditional systems (70%).
4. Object-oriented information systems development is more productive than traditional information systems development (68%).
5. Maintenance of object-oriented information systems is easier than maintenance of traditional information systems (64%).

The Finnish software companies had not experienced the following proposed benefits with the object-oriented paradigm:

1. The companies had not experienced a better and more 'natural' communication between information systems developers and end users due to the use of the object-oriented paradigm. Only 22% of the companies had experienced a more natural communication, when as many as 57% of the companies had NOT experienced this. The case studies, however, indicate that this is a question that is easily misunderstood and therefore one has to be careful when interpreting the results.
2. The companies had not experienced that the object-oriented system development process could be seen as a uniform 'one model' from problem

domain to code and maintenance. 42% of the companies had experienced a uniform one model whereas the same number (42%) had NOT experienced this.

The two exceptions above reflect unexpected empirical evidence when compared with the findings in the review of previous studies.

Problems. It was also much unexpected that the Finnish software companies had experienced so few of the proposed problems.

For the problems, the following were most commonly agreed:

1. It has been difficult to find experienced object-oriented software developers and systems analysts (**65**%).
2. Companies have experienced computer efficiency problems in object-oriented information systems development projects (**49**%).

When analysing the results regarding experienced problems, there are several problems sighted in the review of previous studies that the Finnish software companies have not experienced to a significant degree. The following are the most important:

1. Difficult to find object-oriented system development tools (10% 'Yes', **75**% **'No'**, 7% 'Not Sure', 7% 'Not used' and 1% No answer).
2. The object-oriented paradigm is still immature (28% 'Yes', **66% 'No'** and 6% 'Not sure').
3. Testing object-oriented information systems has been difficult (29% 'Yes', **62% 'No'** and 9% 'Not sure').
4. There has been a problem with reuse for the reason that software developers do not want to reuse a component, because they claim that it does not work (28% 'Yes', **61% 'No'** and 11% 'Not sure').
5. The object-oriented paradigm is considered complex (35% 'Yes', **58% 'No'** and 7% 'Not sure').

The exceptions above are interesting. These empirical results differ from the findings in the previous studies. This discrepancy with mainstream studies on the object-oriented paradigm seems, however, to be in agreement with recent findings by other researchers (Johnson, 2000; Johnson, 2002), and may have to do with the fact that tools and experience have changed in ten years.

5.2 Analysis of empirical findings

A look at scientific studies revealed several findings established earlier by other researchers regarding the benefits and problems with the object-oriented paradigm. The research questions in this study were then based on these.

In the empirical part of this study, some interesting findings regarding experienced benefits and problems with the object-oriented paradigm in Finnish software companies were found. By then comparing the findings from the empirical part of this dissertation with other scientific studies, some issues on the benefits and problems with the object-oriented paradigm could be discussed and presented. The major findings were the following:

Experienced benefits:

The object-oriented paradigm is useful when developing large-scale and complex information systems. 94% of the Finnish software companies had experienced this benefit. Whether this result is based on the overall pre-eminence of the object-oriented paradigm over the older functional paradigm or if it is based on the fact that most companies use the object-oriented paradigm and find it suitable is difficult to say. Probably a proper use of reuse has done the information systems development easier.

A proper use of reuse is beneficial and makes easier development of information systems possible. 92% of the Finnish software companies reported that reuse is beneficial. By reusing objects, classes, components, etc. one must not develop everything from scratch, which is advantageous. However, in the older functional paradigm reuse is also possible, one can, for example, reuse modules that might be procedures, functions or subprograms. However, probably the companies have experienced that the object-oriented paradigm has more reuse possibilities.

According to 70% of the Finnish software companies the quality of object-oriented systems is better than the quality of traditional systems. Reuse of tested components ought to initiate better quality. The complexity of the object-oriented paradigm might threaten the quality aspects. However, the object-oriented paradigm includes also the one model concept and the more natural concept that might affect the quality of the information system favourably. Finally, one could expect that the concept of encapsulation might produce higher quality of the object-oriented information system.

Object-oriented information systems development is more productive than traditional information systems development. A majority (68%) of the Finnish software companies were of this opinion. By reusing objects, classes, components, etc. the productivity of the information systems development work should get higher. The skills of the information systems developer is, however, a major factor when discussing productivity. A trained, experienced and skilful information systems developer might be very productive in traditional functional information systems development as well as in object-oriented information systems development.

A majority (64%) of the Finnish software companies were of the opinion that maintenance of object-oriented information systems is easier than maintenance of

traditional information systems. Once more reuse is the concept that most likely affects the maintenance work positively. One would think that the complexity of the object-oriented paradigm would affect maintenance negatively. However, one must take into consideration the fact that 58% of the companies had not experienced that the object-oriented paradigm would be complex. The 'one model' concept might also affect the maintability because one can study the information system more easily from analysis to implementation and documentation.

Not experienced benefits:

The Finnish software companies had not experienced a better and more 'natural' communication between information systems developers and end users due to the use of the object-oriented paradigm. Merely 22% of the companies had experienced a more natural communication, when as many as 57% of the companies had not experienced this. This question is difficult because in many cases the end users are probably not involved in the information systems development work. In many companies end users are involved in the work but almost certainly to a rather low degree. Because the question of whether the communication between information systems developers and end users is 'better' is a very subjective one that might differ from one information system developer to another, this result can be criticised.

The Finnish software companies had not experienced that the object-oriented system development process could be seen as a uniform 'one model' from problem domain to code and maintenance. 42% of the companies had experienced a uniform one model whereas the same number (42%) had not experienced this. Because the number of 'yes' answers is the same as the number of 'no' answers, this question has no other actual response than that one can argue that the 'one model' benefit is not widely recognised. The used analysis and design method in object-oriented information systems development might further affect the comprehension of the transition from analysis and design into implementation.

Experienced problems:

A majority (65%) of the Finnish software companies were of the opinion that it has been difficult to find experienced object-oriented software developers. Because the object-oriented paradigm nowadays is taught in most universities and polytechnics one can assume that this problem will disappear in the future. However, it might still in the near future be difficult to find *experienced* object-oriented information systems developers.

Though the number is not high, 49% of the Finnish software companies have experienced computer efficiency problems in object-oriented information systems development projects. The used computer, operating system, programming language, and database, etc. might be the factors that affect the efficiency the most and not the object-oriented paradigm per se. The more mature the object-oriented paradigm becomes the better the computer efficiency might also become due to better operating systems, better tools, etc.

Not experienced problems:

As many as 75% of the Finnish software companies were of the opinion that it has not been difficult to find object-oriented system development tools. Because the object-oriented paradigm was more mature when the survey was done than when the previous studies were done, one can argue that the maturity of the object-oriented paradigm is nowadays rather high, which means that object-oriented system development tools are now available. An interesting question is, however, whether the companies can afford these tools.

A majority (66%) of the Finnish software companies were of the opinion that the object-oriented paradigm is not immature. The reason is the same as presented above; the object-oriented paradigm was more mature when the survey was done than when the previous studies were done. Further, there are nowadays several years of experience of object-oriented information systems development in many Finnish software companies.

62% of the Finnish software companies were of the opinion that testing object-oriented information systems has not been difficult. The experience of the information system developer, used testing tools, testing strategies, etc. are issues that might affect the difficulty to test object-oriented information systems more than the used software engineering paradigm.

There has been a problem with reuse for the reason that software developers do not want to reuse a component, because they claim that it does not work. 61% of the Finnish software companies had not experienced this problem. If one has no background in functional information systems development then one probably has an obvious understanding of the need for reuse and software component quality, which an information system developer that is used to work in co-ordinance with the functional paradigm where reuse is utilised but probably to a smaller extent, does not have.

A small majority (58%) of the Finnish software companies were of the opinion that the object-oriented paradigm is not considered complex. Whether a paradigm is complex or not is a rather subjective issue where the information system developer's background and training are factors that most likely affect the view of the developer, which purports that an inexperienced information systems developer might consider the object-oriented paradigm as complex, when an experienced developer considers it rather simple.

6. DISCUSSION

This chapter consists of the parts that Järvinen (2004, p. 172) recommends: repetition of results, limitations, recommendations to practitioners and recommendations to researchers.

6.1 Repetition of results

As mentioned earlier in this study the empirical results differ from the findings in the previous studies. This divergence with mainstream studies on the object-oriented paradigm seems, however, to be in agreement with recent findings by other researchers (Johnson, 2000; Johnson, 2002).

However, though the results of this study are in agreement with the results of the studies by Johnson (2000) and Johnson (2002) there are differences in how this study was carried out and how the other two studies were conducted.

Some interesting differences are presented in Table 9.

Table 9: Differences in studies on object-orientation

	Present study	**Johnson 2000**	**Johnson 2002**
Topic	Benefits and problems with the object-oriented paradigm	Benefits and problems with object-oriented systems development	Object-oriented analysis and design
Type of survey	Mail	Internet	Study of 12 empirical studies
Number of answers	104	150	-
Country	Finland	The US	-
Source - Respondents	Statistics Finland	US subscribers to Communications and OOPS Messenger plus registrants at recent OOPSLA conferences (in the US).	Miscellaneous
Selection	All in the population	Randomly selected sample	-

As can be read from the table above there are some interesting differences in how this study was carried out in comparison with the two other studies. The study by Johnson (2002) was very different from this study because it included no pure empirical study, but a study of some empirical studies, and it was focused mainly on object-oriented

analysis and design. The study by Johnson (2000) is rather similar to this study although some significant differences exist:

- This study can be generalised over all Finnish software companies (with the exception mentioned earlier in this study) because the population was all companies. The study by Johnson (2000) can only be generalised over the rather limited selected population.

- This study was a mail survey. The study by Johnson (2000) was Internet based.

- In this study all companies in the population were selected. In the study by Johnson (2000) a sample was selected from the population.

- This study is concerned with software companies in Finland and the study by Johnson (2000) is concerned with companies in the US.

One could compare the studies regarding validity, reliability, generalisation, etc. nevertheless, no such comparison will be made. However, one can conclude from the comparison in Table 9 that this study most probably is an adequate study and in any case not of inferior quality to the study by Johnson (2000).

As a summary, one can state that a major finding from the empirical part of this study is the following:

> *It is interesting to note how positive the Finnish software companies that took part in the empirical part of this study are towards the object-oriented paradigm. Most of the companies had experienced many of the proposed benefits but had not experienced many of the proposed problems. Out of these results, one can argue that nowadays (2005) the object-oriented paradigm is a leading information systems development paradigm among Finnish software companies, and that the Finnish software companies do not experience significant problems with it.*

One could further argue that many of the benefits of the object-oriented paradigm are not always so strong and obvious. One could also further conclude that many of these problems have been solved, or at least partly.

The results of this study will hopefully be used for improving the understanding of how the benefits with the object-oriented paradigm can be realised, and correspondingly, how the problems with the object-oriented paradigm can be avoided in object-oriented information system development in the future.

The most important outcomes of this study are twofold:

1. To improve the knowledge of the benefits and problems with the object-oriented paradigm.
2. To offer some rather practical advice to organisations on the possibilities and problems with the object-oriented paradigm.

Finally, another quite concrete result of this study is the comprehensive presentation of benefits and problems of the object-oriented paradigm. In none of the studies investigated was a presentation of the same kind found.

However, probably the most significant outcome of this study was the difference between the results of previous studies on the benefits and problems of the object-oriented paradigm and the empirical findings concerning Finnish software companies.

6.2. Limitations

When doing a study like this several limitations have to be considered. Already in the introduction chapter of this study some boundaries were stated. There are, however, some other limitations connected to this study of which the following are the most significant:

1. When the literature review was made great care was taken in order to find all major benefits and problems presented in previous studies. However, there is certainly always a possibility that some major proposed benefits and / or problems have not been found.

2. The selection of research questions could have been done in another way, now some proposed benefits and problems are associated with several research questions and some other benefits and problems are associated with only one research question.

3. All research questions were included in the case studies but the selection of research questions for the survey was difficult and could probably have been done in another way.

4. In the population for the survey all software companies with five or more employees in Finland were considered. In the case studies five software companies were selected from the Helsinki area and one from Vaasa. There is a difference in the distribution of the answers.

5. When doing the case studies the interview was rather long and in the end some of the respondents become tired which reflected the answers.

6. Out of the benefits, problems, connections between benefits and connections between problems, two theoretical models are presented. These theoretical models have not been acceptably proven in this study and have consequently been developed as hypotheses for further research.

Other limitation could undoubtedly be found. In this study the author of this work has strived to lay emphasis on the reporting of the research process and the empirical study, and subsequently the reader can hopefully make some conclusions of his or her own.

6.3 Recommendations to practitioners

The major contribution for practitioners is the identification and presentation of the major benefits and problems with the object-oriented paradigm in information systems development. In order to do feasible software development the software developer has to be aware of the possibilities and dangers with the chosen software development paradigm. Although one can be read from the results of this work that most practitioners seem to be aware of the major benefits and problems it is always good to have them arranged in one source for repetition and further studies.

Hopefully the two theoretical models with the associations of the 57 questions considered give some understanding for practitioners on how different benefits (and problems) are connected. This knowledge makes it hopefully easier to perform object-oriented information systems development work in the future.

6.4 Recommendations to researchers

6.4.1 A look in the future

The era of programming and building information systems in a predominantly functional way or in some other ‘older’ way is probably coming to an end. Information system developers are now realizing that there are several interesting ways of building information systems, and the object-oriented way is one. The main task in the future will probably be to select the most appropriate information systems development approach for a specific information system development project. The solution can also be of course a mixture of several information system development approaches. When selecting the appropriate information system development approach or the appropriate mixture of information systems development approaches there are many things to take into consideration. One of the things to consider is the benefits and problems with an information system development paradigm. (Martin & Odell, 1995, p. 1) If problems with the object-oriented paradigm can be managed, and benefits of the object-oriented paradigm can be realised, information systems development can certainly be promoted by it. Therefore, this study can hopefully be of some help for both managers and information system developers.

One also has to keep in mind of course that new ways of carrying out information system development are evolving all the time. The object-oriented paradigm has a connection to the traditional functional approach in several ways; for example, object orientation stores the segments of code (the methods) in a similar way as the functional approach uses functions. Out of object orientation, for example, grew a new way of developing software with agents. Software agents have their own control mechanism and work with code, state and invocation of the software agents. The software agents can even have individual rules and can be considered active objects with initiative. (Odell, 2000) Software agents will not of course be the last information systems development approach. New ways of performing information systems development will undoubtedly come in the future.

This investigation can be seen as a beginning of research on how one can perform more efficient object-oriented information systems development. There are many other issues that can be studied; for example, what shortages are there in the object-oriented paradigm theory and what new theory is needed in order to improve the object-oriented paradigm? Other study issues could address the questions of how to develop new tools for object-oriented information system development, as according to Bhattacherjee & Gerlach (1998), there still seems to be a lack of efficient and easy-to-use object-oriented information systems development tools.

6.4.2 Two theoretical models

In this section two theoretical models for future research are presented. These theoretical models can be seen as interesting topics for future academic research.

When pondering about theory, one must remember that theory is the answer to questions (Sutton & Staw, 1995). Sutton & Staw (1995) propose that theory considers the connections among phenomena; why acts, events, structure and thoughts occur; the nature of causal relationships is discussed as well. The most interesting questions in this study consider the realisation of benefits and the avoidance of problems when working with the object-oriented paradigm.

The connections (associations) between different benefits and problems are also of interest in this study; if a software company has experienced a benefit, it is interesting to study further benefits connected to the initial benefit. In the empirical part of this study the connections (associations) presented in the models were evaluated. It was found that the empirical survey results in this study support the theoretical models rather well, though one has to remember that the number of respondents in some of the presented connections was rather low.

Because of the low number of respondents and the often rather low percentage of respondents that had experienced the following benefit, the connections (associations) must be considered as weak and they have to be interpreted as hypotheses and not proven connections.

One also has to take into consideration that the possible connections were investigated in an unpretentious way where:

1. All respondents that had experienced a certain benefit or problem were selected.

2. Out of this population the number of respondent that also had experienced the following (connected) benefit or problem was selected.

In this section, theoretical models of the connections between the benefits (CBB model) and the connections between the problems (CBP model) are developed as hypotheses for further research. The models describe how one benefit gives birth to another benefit and how one problem gives birth to another problem and so on. The connections are selected from the identified connections presented in the sections on benefits and

problems with the object-oriented paradigm. One can anticipate that the models will give some kind of answer to the questions 'Why are there benefits?' 'Why are there problems?' 'How are benefits connected?' and 'How are problems connected?'

6.4.2.1 The CBB model - Connection Between Benefits

When the CBB model was developed, the main issue was to utilise the findings from the review of previous studies concerning connections between benefits. Some of the connections were further tested in the empirical study.

In the section on benefits of the object-oriented paradigm, object-oriented analysis and object-oriented design were presented but regarded as activities and not benefits. They are consequently not discussed here.

The first benefit that was presented, discussed and analysed in the above-mentioned section was the one model benefit. This benefit has its roots in the object-oriented paradigm because it is a uniform paradigm from analysis to implementation and maintenance (Henderson-Sellers & Edwards, 1990; Radin, 1996).

> THE OBJECT-ORIENTED PARADIGM is usually based on a -> ONE MODEL information systems development life cycle.

The next benefit dealt with was the management of complexity benefit. Object-oriented systems better reflect the real world (Webster, 1995, p. 58; Wilkie, 1995, p. 39), which decreases complexity.

> The object-oriented paradigm is considered more NATURAL -> which makes MANAGEMENT OF COMPLEXITY easier.

When the benefit of management of complexity was further analysed it was found that by utilising reuse, one can manage complexity, because the information system under construction can be built out of existing tested building blocks that are reused.

> The REUSE concepts makes it possible to reuse components and other artefacts -> which makes MANAGEMENT OF COMPLEXITY easier.
>
> *This connection was tested in the empirical study and found valid.*

The following benefit that was considered in the section on the benefits of the object-oriented paradigm was the benefit of productivity, faster development and reduced costs. According to Bhattacherjee & Gerlach, 1998; Caliò et al., 2000; Henderson-Sellers & Edwards, 1990; Love, 1993, p. 85; Manhes, 1998; Musakka, 1996; Nowicki & Kosiak, 1996; Sheetz & Tegarden, 1996; Smith & McKeen, 1996) reuse leads to faster development and better productivity. Reuse further leads to better efficiency and reduced costs. It also makes maintenance easier, and easier maintenance decreases the costs. Faster development also leads to lower costs of course (Jenz, 1999a; Räisänen, 1997a, p. 11).

> The utilisation of REUSE results in -> FASTER DEVELOPMENT because readymade artefacts like components can be used, which results in higher -> PRODUCTIVITY -> which affects the EFFICIENCY of the information systems development project -> which leads to REDUCED COSTS.
>
> *The connection between the reuse benefit and the more productive benefit was tested in the empirical study and found valid.*
>
> *The connection between the reuse benefit and the faster development benefit was tested in the empirical study and also found valid.*

The next considered benefit was the benefit of quality and usability. Reuse of existing software components is actually connected to two benefits (reuse and software components), and these two benefits give birth to better quality and usability (Gillach & Deyo, 1993; Jenz, 1999b; Lim, 1994; Love, 1993, p. 80; Martin & Odell, 1992, p. 32; Räisänen, 1997a, p. 13; Sheetz & Tegarden, 1996; Smith & McKeen, 1996; Taylor, 1990, p. 104). This is only true of course if one reuses tested components of high quality. The one-model benefit is also connected to the benefit of quality and usability, because as Love (1993, pp. 188-189) proposes, the model in the software designers mind can be expressed in software itself.

> Utilising REUSE and reusing tested components -> results in HIGHER QUALITY.
>
> *This connection was tested in the empirical study and found valid.*
>
> The ONE MODEL concept that makes it possible to save design issues in the programming code -> results in HIGHER QUALITY because the information system is easier to understand.
>
> *This connection was tested in the empirical study and found valid.*
>
> Using SOFTWARE COMPONENTS like classes -> results in better USABILITY because tested components can be used.

Natural and better mapping to the problem domain was the following benefit that was presented, discussed and analysed. The software parts that are a result of object-oriented design are proposed by several researchers to be more natural (Bozowski, 1997; Koehler, 1992, Korson & McGregor, 1990; Martin & Odell, 1992, p. 31). However, because object-oriented design was not considered a benefit in the section on benefits of the object-oriented paradigm, the connection between object-oriented design and the object-oriented paradigm as more natural is excluded. On the other hand, it is rather obvious that the benefit of ‘natural’ comes from the object-oriented paradigm itself so it is possible to identify the following connection:

> The OBJECT-ORIENTED PARADIGM with objects are considered as more - > NATURAL for human beings.

The next benefit that was considered was the benefit of enhanced maintenance. The easier maintenance because of the object-oriented paradigm is a significant benefit that

is connected to the reuse of existing components. Object-oriented analysis and design also makes maintenance easier if the appropriate classes have been identified (Gillibrand, 2000). However, as mentioned above, in the presentation, discussion and analysis of object-oriented benefits, object-oriented analysis and object-oriented design were considered activities and not benefits, and they are therefore omitted here.

> The utilisation of REUSE makes it easier to maintain the information systems because tested and existing components can be used -> MAINTENANCE becomes consequently easier.
>
> *This connection was tested in the empirical study and found valid.*

> The use of SOFTWARE COMPONENTS makes it possible to avoid programming new parts for an information system, which makes -> MAINTENANCE easier and faster.
>
> *This connection was tested in the empirical study and found valid.*

Software components are very central in the object-oriented paradigm, and one important base for reuse. Object-oriented information systems and applications are also more robust, more extensive, are more flexible and have higher integrity due to software components (combined with some other issues like encapsulation), according to Henderson-Sellers (1992, p. 68), Henderson-Sellers & Edwards (1994, p. 15) and Petre (2000, pp. 2-3). With this in mind, the following connections can be presented:

> The OBJECT-ORIENTED PARADIGM uses -> SOFTWARE COMPONENTS.

> Using SOFTWARE COMPONENTS results in higher -> FLEXIBILITY because one can reuse premade artefacts.

> Using tested SOFTWARE COMPONENTS results in higher -> ROBUSTNESS.

> Making use of SOFTWARE COMPONENTS leads to -> easier EXTENSIBILITY possibilities.

> Using SOFTWARE COMPONENTS results in higher -> INTEGRITY because the components are encapsulated without things like global variables.

Easier End-user computing was the following benefit that was dealt with. Easier End-user computing is a benefit that has its foundation in the object-oriented paradigm itself. The basic idea is that end users can more easily start building their own applications by reusing existing software components.

> The OBJECT-ORIENTED PARADIGM with readymade components makes it easier to develop information systems which results in -> better possibilities for END-USER COMPUTING.

The next benefit that was considered was reuse. Reuse is probably the most interesting and important benefit of the object-oriented paradigm. It is also a benefit that has its origin in the object-oriented paradigm itself. Earlier in this section a connection

between reuse and faster development, better productivity, efficiency and reduced cost was presented. Earlier in this section another connection between reuse and maintenance was presented as well. However, reuse gives birth to several other benefits, and the following connections were found:

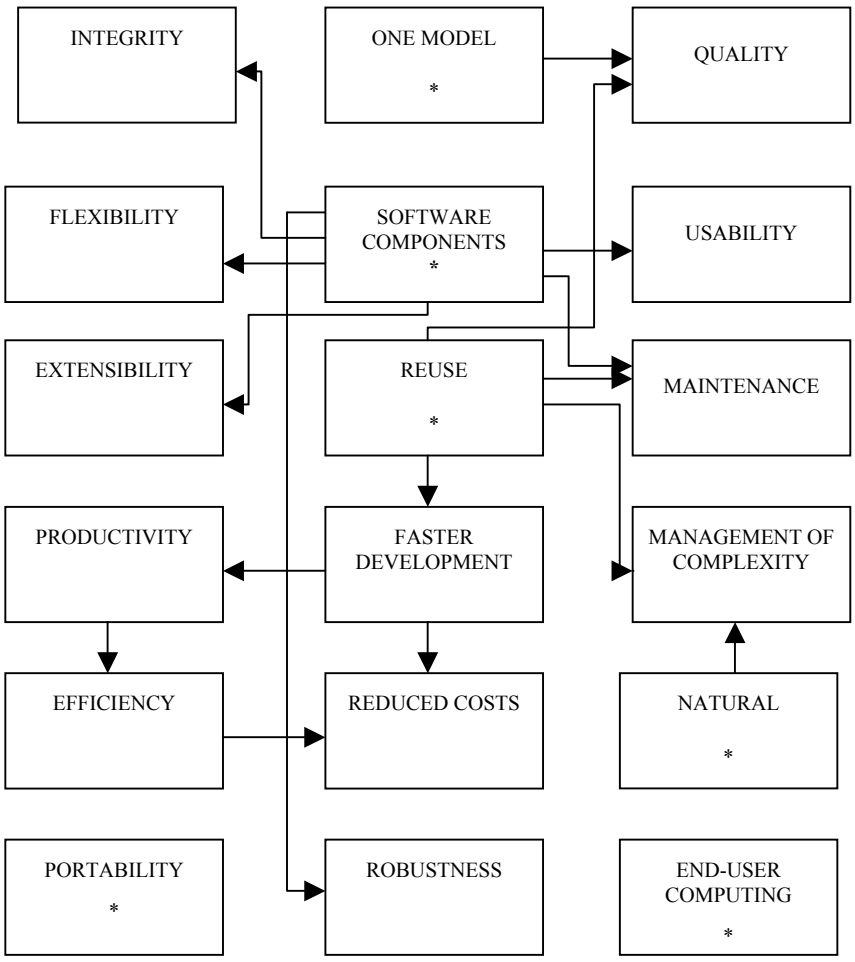
> The OBJECT-ORIENTED paradigm is connected to the -> REUSE concept.
>
> By performing REUSE -> MANAGEMENT OF COMPLEXITY can be controlled more easily because the complexity of an information system is often due to a hierarchy that can be built by using reuse.
>
> By utilising REUSE -> HIGHER QUALITY can be achieved because the information system is built out of readymade and tested components.

The last benefit was portability. Portability is considered a benefit that has its basis in the object-oriented paradigm itself (Agarwal et al., 2000).

> The OBJECT-ORIENTED paradigm -> has a good support for PORTABILITY in the Java programming language (and some other programming languages).

The whole model is presented in Figure 37.

Figure 37: The CBB Model



Note that the * symbol stands for the fact that the benefit comes from the object-oriented paradigm itself. There are thus benefits (boxed in the diagram) with no connections. These benefits come from the object-oriented paradigm itself and have no connections to other benefits.

Discussion. It is a subjective approach of course to connect the benefits to each other, though the connections do have a modest verification in this study. Some of the benefits are connected in a rather obvious way and other benefits are more difficult to connect. The model that is presented here is the result of an analysis of the benefits that was carried out by the author of this study. One cannot perceive the model as the only "right" model. Several other models could certainly be developed out of the benefits. For example, Räisänen (1997a, p. 11) has developed a slightly different model, albeit more limited.

An interesting issue one can notice when looking at the model is that there are two benefits that act like 'spiders in the net', and that out of these two benefits most other benefits come. The two 'spiders' are REUSE and SOFTWARE COMPONENTS. One

could certainly conclude that these two benefits are the 'main' benefits of the object-oriented paradigm. By reusing software components several other benefits arise.

6.4.2.2 The CBP model – Connections Between Problems

When the CBP model was developed the main issue was to utilise the findings from the review of previous studies concerning connections between problems. Some of the connections were tested in the empirical study.

In the section on problems with the object-oriented paradigm, complexity was first presented, discussed and analysed. The object-oriented paradigm is complex according to several researchers like Harrison et al. (1996), Maring (1996), Noack & Schienmann (1999) and Johnson (2000). In the aforementioned section, it was found that the complexity of the object-oriented paradigm comes from the object-oriented paradigm per se.

> The OBJECT-ORIENTED paradigm is based on rather complex concepts like polymorphism and inheritance hierarchies. This results in higher -> COMPLEXITY in the information systems development work.
>
> *This connection between problems was tested in the empirical study and found rather valid because 43% answered "yes" and 47% answered "no".*

Many researchers like Bhattacherjee & Gerlach (1998) and Webster (1995, p. 39) claim that the object-oriented paradigm is still immature, though this is an issue that is improving. On the other hand, some researchers are of a different opinion, and think that nowadays the object-oriented paradigm is more mature. Having said that, one can still propose that the object-oriented paradigm is still somewhat immature.

> The OBJECT-ORIENTED paradigm is still -> IMMATURE in some areas.

Poor support for several important areas like testing is a problem that is rather broad. Again, this poor support is most likely based on the immaturity of the object-oriented paradigm (Henderson-Sellers, 1994, p. 21; Pancake, 1995; Wolber, 1997). Therefore, the following connection between problems was identified:

> The object-oriented paradigm is still IMMATURE in some areas -> which results in POOR SUPPORT FOR SOME AREAS.

The following problem that was considered was the problem of difficulties in measuring object-oriented systems. This is a problem that will probably disappear when the object-oriented paradigm becomes more mature. The connection between problems is therefore:

> The object-oriented paradigm is still IMMATURE in some areas -> one area is software metrics which results in DIFFICULTIES IN MEASURING OBJECT-ORIENTED SYSTEMS.

The following problem that was dealt with was the problem of training & lack of experience. There are two problems that are connected to this difficulty, namely the immaturity and complexity of the object-oriented paradigm. Numerous software developers have not become skilled at the object-oriented paradigm yet.

> The object-oriented paradigm is still IMMATURE in some areas -> and it might be difficult to find TRAINED and EXPERIENCED information systems developers in these areas.
>
> *This connection between problems was tested in the empirical study and found valid because 80% answered "yes".*
>
> The object-oriented paradigm has high COMPLEXITY -> which makes TRAINING more difficult and there are several information systems developers that have a LACK OF EXPERIENCE.
>
> *This connection between problems was tested in the empirical study and found valid because 78% answered "yes".*

Efficiency was the following problem that was presented, discussed and analysed in the section on problems with the object-oriented paradigm. This problem has to do with its immaturity.

> The object-oriented paradigm is still IMMATURE in some areas, which results in -> EFFICIENCY problems.
>
> *This connection between problems was tested in the empirical study and found rather valid because 56% answered "yes".*

The starting costs for a new object-oriented software development project are still often high today because there are not many components that can be reused. However, when the object-oriented paradigm becomes more mature there will be more reusable components both on the market and in-house, and therefore the costs will become lower.

> The object-oriented paradigm is still IMMATURE in some areas, which results in -> higher COSTS.
>
> *This connection between problems was tested in the empirical study and found valid because 64% answered "yes".*

The next problem considered was the limited usability of components. There are problems like how to find components (Garland et al., 1994) and how to handle the complexity of components (Jarzabek & Knauber, 1999), etc. When the object-oriented paradigm becomes more mature, these problems will probably disappear.

> The object-oriented paradigm is still IMMATURE in some areas which results in -> LIMITED USABILITY OF COMPONENTS because there are no suitable components to reuse or existing components are complex or difficult to reuse

The connection between the problems of immaturity and finding components for reuse was tested in the empirical study and found valid because 68% answered "yes".

Reuse or problems with reuse, was the following problem presented, discussed and analysed. As was said in an earlier sub section of this study reuse is probably one of the main benefits of the object-oriented paradigm. However, this major benefit also has some troubles. Several different problems with reuse were analysed and the conclusion was that many of these would disappear when the object-oriented paradigm becomes more mature. Reuse is a complex activity, which also often makes the object-oriented paradigm more complex.

The object-oriented paradigm is still IMMATURE in some areas (for example, a lack of good textbooks), which results in -> PROBLEMS WITH REUSE.

This connection between problems was tested in the empirical study and found valid to a certain extent because 50% answered "yes".

By using REUSE inheritance structures can be developed which increases the -> COMPLEXITY of the information system.

In the section on problems with the object-oriented paradigm, object-oriented analysis and object-oriented design were presented but regarded as activities and not problems. They are accordingly not presented here.

The last problem that was dealt with was the problem with the lack of object-oriented databases and common interfaces. The conclusion was that the lack of object-oriented databases is a result of the immaturity of the object-oriented paradigm, and that the lack of common interfaces is due to the object-oriented paradigm itself.

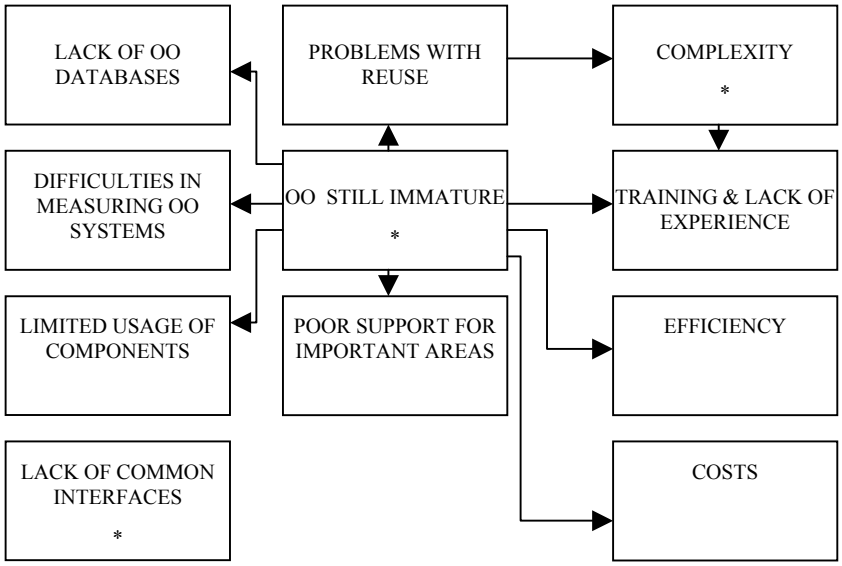
The object-oriented paradigm is still IMMATURE in some areas (like databases), which results in -> A LACK OF OBJECT-ORIENTED DATABASES.

This connection between problems was tested in the empirical study and NOT found valid because 32% answered "yes" and 68% answered "not sure", probably because so few of the respondents use object-oriented databases.

The OBJECT-ORIENTED paradigm has resulted in few object-oriented databases and these object-oriented databases have a -> LACK OF COMMON INTERFACES for ad hoc queries.

The whole model is presented in Figure 38.

Figure 38: The CBP Model



Note that the * symbol stands for a problem that has its origin in the object-oriented paradigm itself.

Discussion. It is a subjective approach of course to connect the problems to each other, although a modest verification of the connections in this study does exist. As one can further notice it is probably not even possible to connect all the problems to each other. The model that is presented here is the result of an analysis of the problems that was carried out by the author of this study. One cannot perceive the model as the only "right" model. Several other models could certainly be developed out of the problems.

When looking at the model one can notice that the problem of immaturity is a considerable one. Immaturity means the difficulty in finding appropriate information system development tools and artefacts like object-oriented databases. Many other problems originate from the object-oriented paradigm itself. Whether the object-oriented paradigm is immature or not is a difficult question; one result of the empirical part of this study was that a majority of the Finnish software companies were of the opinion that the object-oriented paradigm is not immature.

REFERENCES

Aczel, A. (1999): *Complete Business Statistics*. Singapore: McGraw-Hill Book Co.

Agarwal, R., De, P., Sinha, A. & Tanniru, M. (2000): On the Usability of OO representations. *Communications of the ACM*, Vol. 43, No. 10, pp. 83-89.

Airikkala, J. (1996): Comptel pitää oliosta. *SAS Institute oy:n asiakaslehti*, No. 2, p. 5. In Finnish.

Aksit, M. & Bergmans, L. (1992): Obstacles in Object-Oriented Software Development. OOPSLA'92, ACM SIGPLAN Notices, Vol. 27, No. 10, pp. 341-358.

Al-Ahmad, W. & Steegmans, E. (2000): Inheritance in Object-Oriented Languages: Requirements and Supporting Mechanisms. *Journal of Object-Oriented Programming*, Vol. 12, No. 8, January 2000, pp. 15-24.

Alasuutari, P. (1994): *Laadullinen tutkimus*. Tampere, Finland: Vastapaino. 3 Ed. In Finnish. English edition: Researching Culture: Qualitative Method and Cultural Studies. London: Sage Publications Ltd. 1995, 1 ed. based on the earlier version of the Finnish original from 1993.

Alasuutari, P. (1996): Personal presentation. Course in qualitative methods at Tampere University in Finland on October 27, 1996.

Alencar, P., Cowan, D., Lucena, C. & Nova L. (1998): A Model for Gluing Components. Weck, W., Bosch, J. & Szyperski, C. (Eds.). Proceedings of the WCOP'98 Third International Workshop on Component-Oriented Programming, Brussels, Belgium, July 21, 1998. Turku Centre for Computer Science, TUCS General Publication, No 10, September 1998.

Alexander, C. (1979): *The Timeless Way of Building*. New York: Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fixdahl-King, I. & Angel S. (1977): *A Pattern Language*. New York: Oxford University Press.

Altheide, D. & Johnson, J. (1994): Criteria for assessing interpretative validity in qualitative research. Denzin, N. & Lincoln, Y. (eds.): *Handbook of Qualitative Research.* London: Sage, pp. 485-499.

Ambler, S. (1994): In search of a generic SDLC for object systems. *Object Magazine*, Vol. 4, No. 6, pp. 76-78.

Ambler, D. (1998): *Building Object Applications That Work*. Cambridge, UK: Cambridge University Press. Copublished by the press syndicate of the University of Cambridge and SIGS books.

Andersen, E. (1996): *Systemutveckling – principer, metoder och tekniker*. Lund, Sweden: Studentlitteratur. In Swedish.

Bansiya, J. & Davis, C. (1997): Automated Metrics and Object-Oriented Development. *Dr. Dobb's Journal*, Vol. 272, December 1997, pp. 42-48.

Bansiya, J., Davis, C. & Etzkorn, L. (1999): An Entropy-Based Complexity Measure for Object-Oriented Designs. *Theory and Practice of Object Systems*, Vol 5, No 2, pp. 111-118.

Barondes, S. (1998): *Mood Genes*. London: Penguin Books Ltd.

Basili, V., Briand, L. & Melo, W. (1996a): How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM*, No. 10, pp. 105-116.

Basili, V., Briand, L. & Melo, W. (1996b): A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, pp. 751-761.

Benbasat, I., Goldstein, D. & Mead, M. (1987): The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, Vol. 11, No. 3, September 1987, pp. 369-386.

Berard, E. (1992): *Essays on Object-Oriented Software Engineering*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Berard, E. (1998): Metrics for Object-Oriented Software Engineering. The Object Agency, Inc. Available from: http://www.toa.com/pub/moose.htm [Accessed 11 October 2002].

Berg, W., Cline, M. & Girou, M. (1995): Lessons Learned from the OS/400 OO Project. *Communications of the ACM*, Vol. 38, No. 10, pp. 55-64.

Bhattacherjee, A. & Gerlach, J. (1998): Understanding and Managing OOT Adoption. *IEEE Software*, Vol. 15, No. 3, May/June 1998, pp. 91-96.

Binder, R. (1999): *Testing Object-Oriented Systems. Models, Patterns and Tools*. Reading, Massachusetts, US: Addison-Wesley Longman, Inc.

Blair, G. S & Lea, R. (1992): The impact of distribution on support for object-oriented software development. *Software Engineering Journal*, Vol. 7, No. 2, pp. 130-138.

Blom, G. (1984): *Statistikteori med tillämpningar*. Lund, Sweden: Studentlitteratur. In Swedish.

Boehm, B. (1986): A spiral model of software development and enhancement. *ACM Software Engineering Notes*, Vol. 11, No. 4, pp. 14-24.

Bohrer, K., Johnson, V., Nilsson, A. & Rubin, B. (1998): Business Process Components for DISTRIBUTED OBJECT APPLICATIONS The San Francisco project offers developers extendible components to simplify the transition to distributed, easily customizable applications. *Communications of the ACM*, Vol. 41, No. 6, pp. 43-48.

Booch, G. (1994): *Object-oriented Analysis and Design with Applications*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc.

Booch, G. & Vilot, R. (1990): Object-oriented design, Inheritance relationships. *C++ Report*, Vol. 2, No. 8, pp. 11-13.

Bosch, J. (1997): Adapting Object-Oriented Components. Weck, W., Bosch, J. & Szyperski, C. (Eds.). Proceedings of the WCOP'97 Second International Workshop on Component-Oriented Programming, Jyväskylä, Finland, June 9, 1997. Turku Centre for Computer Science, TUCS General Publication, No 5, September 1997.

Bosch, J., Szyperski, C. & Weck, W. (1997): Summary of the Second International Workshop on Component-Oriented Programming (WCOP'97). Weck, W., Bosch, J. & Szyperski, C. (Eds.). Proceedings of the WCOP'97 Second International Workshop on Component-Oriented Programming, Jyväskylä, Finland, June 9, 1997. Turku Centre for Computer Science, TUCS General Publication, No 5, September 1997.

Boudreau, M-C., Gefen, D. & Straub, D. (2001): Validiation in Information Systems Research: A State-of-the-Art Assessment. *MIS Quarterly*, Vol. 25, No. 1, March 2001, pp. 1-16.

Boulanger, D. & Dubois, G. (1998): An Object Approach for Information System Cooperation. *Information Systems*, Vol. 23, No. 6, pp. 383-399.

Bozowski, N. (1997): Philosophizing about objects. *Computing Canada*, October 14, p. 48-49.

Bradford (2002): Web pages of Overseas Trade Support for SMEs (OTSS) – project, without named editor, The University of Bradford Management Centre. Available from: http://www.brad.ac.uk/acad/mancen/otss/geninfo.htm#team [Accessed 11 February 2005].

Brancheau, J. & Brown, C. (1993): The Management of End-User Computing: Status and Directions. *ACM Computing Surveys*, Vol. 25, No. 4, pp. 437-482.

Brooks, F., Jr. (1979): The Mythical Man-Month. Reading, Massachusetts, US: Addison-Wesley.

Brooks, F., Jr. (1987): No Silver Bullet; Essence and Accidents of Software Engineering. *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.

Bruegge, B & Dutoit, A. (2000): *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*. Upper Saddle River, New Jersey, US: Prentice Hall, Inc.

Brunet, J, Cauvet, C., Meddahi, D. & Semmak, F. (1994): Applying Object-Oriented Analysis on a Case Study. *Information Systems*, Vol. 19, No. 3, pp. 199-209.

Buchanan, D., Boddy, D. & McCalman, J. (1988): Getting In, Getting On, Getting Out and Getting Back. Bryman, A. (ed.), *Doing Research in Organizations*. London: Routledge, pp. 53-67.

Buchholz, J. (1999): Component-Based Development: Methodik und Toolunterstutzung. Diplomarbeit im Fachbereich Wirtschaftsinformatik an der Fachhochschule Furtwangen. Available from: http//: www.joachimbuchholz.de/ cbd/_abstract_o-bg.html [Accessed 19 September 2002}. Abstract. In German.

Buxton, J. (1993): On the Decline of Classical Programming. Sommerville, I. & Paul, M. (Eds.). Fourth European Software Engineering Conference (ESEC). Garmisch-Partenkirchen, Germany, September 13-17, 1993. Proceedings, Springer-Verlag, Lecture Notes in Computer Science, pp. 1-9.

Bäumer, D., Knoll, R., Gryczan, G. & Zullighoven, H. (1996): Large Scale Object-Oriented Software-Development in a Banking Environment. An experience Report. Pierre Cointe (Ed.). ECOOP '96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 1996. Proceedings, Springer Verlag, Lecture Notes in Computer Science 1098, pp. 73-90.

Cackowski, D., Najdawi, M. & Chung, Q. (2000): Object analysis in organizational design: A solution for matrix organisations. *Project Management Journal*, Vol. 31, No. 3, pp. 44-51.

Calió, A., Autiero, M. & Bux, G. (2000): Software Process Improvement by Object Technology. 22nd International Conference on Software Engineering (ICSE), University of Limerick, Ireland, 4-11 June 2000. Proceedings of ACM and SigSoft, pp. 641-647.

Canning, L. & Nethercott, R. (1996): Using Fusion for Commercial Fixed-Price Bespoke Development. Chapter 4 in Malan, R., Letsinger, R. & Coleman D. (1996): *Object-Oriented Development at Work. Fusion in the Real World*. Upper Saddle River, Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Capper, N., Colgate, R., Hunter, J. & James, M. (1994): The impact of object-oriented technology on software quality: Three case histories. *IBM Systems Journal*, Vol. 33, No. 1, pp. 131-157.

Carr, D. (1999): Building with blocks. *Internet World*, Vol. 5, No. 26, p. 69.

Cartwright, M. & Shepperd, M. (2000): An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Engineering*, Vol. 26, No. 8, pp. 786-795.

Casais, E. (1995): Managing Class Evolution in Object-Oriented Systems. Nierstrasz, O. & Tsichritzis, D. (Eds.), *Object-Oriented Software Composition,* pp. 201-244. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Castelluccio, M. (1997): Why all the noise over OOP (Object-oriented programming)? *Management Accounting*, Vol. 79, No. 3, pp. 53-56.

Champeaux de, D., Anderson, A. & Feldhousen, E. (1992): Case Study of Object-Oriented Software Development. OOPSLA'92, ACM SIGPLAN Notices, Vol. 27, No. 10, pp. 377-391.

Champeaux de, D., Lea, D. & Faure, P. (1993): *Object-Oriented System Development*. Reading, MA: Addison-Wesley Publishing Company.

Champeaux de, D. (1996): *Object-Oriented Development Process and Metrics*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Charpentier, C. (2001): Kravanalys. En empirisk undersökning om användningen av systemutvecklingsmodeller i företag. *Svenska handelshögskolan*. Masters thesis. In Swedish.

Chidamber, S. & Kemerer, C. (1994): A Metrics Suite for Object-oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493.

Clements, P. (1995): From Subroutines to Subsystems: Component Based Software Development. *American Programmer*, Vol. 8, No. 11, November 1995.

Coad, P., North, D. & Mayfield, M. (1995): *OBJECT MODELS Strategies, Patterns & Applications*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Coad, P. & Yourdon, E. (1990): *Object-Oriented Analysis*. Englewood Cliffs, New Jersey, US: Prentice-Hall, Inc.

Coad, P. & Yourdon, E. (1991): *Object-Oriented Design*. Englewood Cliffs, New Jersey, US: Prentice-Hall, Inc.

Cockburn, A. (1993): The impact of object-orientation on application development. *IBM Systems Journal*, Vol. 32, No. 3, pp. 420-444.

Cockburn, A. (1998): *Surviving Object-Oriented Projects: a Manager's Guide*. Reading, Massachusetts, US: Addison-Wesley Longman, Inc.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. & Jeremaes, P. (1994): *Object-Oriented Development - The Fusion Method*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Covaleski, M. & Dirsmith, M. (1990): Dialectic tension, double reflexivity and the everyday accounting researcher: on using qualitative methods. *Accounting, Organizations and Society*, Vol. 15, No. 6, pp. 543-573.

Dahl, O-J. & Nygaard, K. (1966): Simula – an algol-based simulation language. *Communications of the ACM*, Vol. 8, No. 9, pp. 671-678.

Dahl, S. & Lindqvist, K. (1993): *Objektorienterad programmering och algoritmer i Simula.* Lund, Sweden: Studentlitteratur. In Swedish.

Davis, J. & Morgan, T. (1993): Object-Oriented Development at Brooklyn Union Gas. *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 67-74.

Davis, T. (2000): Object-Oriented Design in Procedural Environments. *Dr. Dobb's Journal*, June 2000, pp. 68-72.

Deubler, H-H. & Koestler, M. (1994): Introducing Object Orientation into Large and Complex Systems. *IEEE Transactions on Software Engineering*, Vol. 20, No. 11, pp. 840-848.

Dijkstra, E. (1965): Programming considered as a human activity. Proceedings of the 1965 IFIP Congress, North-Holland Publishing Co., Amsterdam, the Netherlands.

Eden, A. (2002): A Theory of Object-Oriented Design. *Information Systems Frontiers*, Vol. 4, No. 4, pp. 379-391.

Eeles, P. & Sims, O. (1998): *Building Business Objects*. New York, US: John Wiley & Sons, Inc.

Eisenhardt, K. (1989): Building Theories from Case Study Research. *Academy of Management Review*, Vol. 14, No. 4, pp. 532-550.

Eliëns, A. (2000): *Principles of Object-Oriented Software Development*. Harlow, England: Addison-Wesley Publishing Company.

Ellis, C. & Gibbs, S. (1989): Active Objects: Realities and Possibilities. Kim, W. & Lochovsky, F. (eds.), *Object-Oriented Concepts, Databases and Applications*. New York: ACM press, Addison-Wesley Publishing Company, pp. 561-572.

Eriksson, H-E. (1992): *Objektorienterad programutveckling med C++*. Lund, Sweden: Studentlitteratur. In Swedish.

Eriksson, H-E. & Penker, M. (1996): *Objektorientering - handbok och lexikon*. Lund, Sweden: Studentlitteratur. In Swedish.

Erlikh, L. (2000): Leveraging legacy system dollars for E-business. *IT Pro* (IEEE), May/June 2000, pp. 17-23.

Esch, J. (1995): A Fine MESS. Real-time manufacturing execution systems bridge the gap between planning and the plant floor. *Byte*, December, pp. 67-75.

Fagerström, J. (1993): *Objektorienterad systemutveckling – en introduktion*. Lund, Sweden: Studentlitteratur. In Swedish.

Fagerström, J. (1995): *Objektorienterad analys och design – en andra generationens metod*. Lund, Sweden: Studentlitteratur. In Swedish.

Fayad, M. (2000): Introduction to the computing surveys' electronic symposium on object-oriented application frameworks. *ACM Computing Surveys*, Baltimore, Vol. 32, No. 1, March 2000, pp. 1-11.

Fayad, M., Laitinen, M. & Ward, R. (2000): Software Engineering in the small. *Communications of the ACM*, Vol. 43, No. 3, pp. 115-118.

Fayad, M. & Schmidt, C. (1997): Object-Oriented Application Frameworks. *Communications of the ACM*, Vol. 40, No. 10, pp. 32-38.

Fayad, M. & Tsai, W-T. (1995): Object-Oriented Experiences. *Communications of the ACM*, Vol. 38, No. 10, pp. 51-53.

Fayad, M., Tsai, W-T. & Fulghum, M. (1996): Transition to Object-Oriented Software Development. *Communications of the ACM*, Vol. 39, No. 2, pp. 109-121.

Fernandes, H. (1998): Important Concepts of Object Orientation. Free Speech Online. Blue Ribbon Campaign. Available from: http://www.geocities.com/fernanesh/ OT_Concepts.html [Accessed 11 October 2002]

Fichman & Kemerer (1993): Adoption of Software Engineering Process Innovations: The Case of Object Orientation. *Sloan Management Review.* Vol. 34, No. 2, Winter 1993, pp. 7-22.

Finch, L. (1998): So Much OO, So Little Reuse. *Dr. Dobb's Journal*, Online op-Eds. Available from: http://www.ddj.com/documents/ [Accessed 23 September 2002]. Journal: May 7, 1998.

Fitzgerald, B. (1995): The Use of Systems Development Methods: A Survey. ESRC Research and Discussion Papers. Cork: University College Cork Ireland.

Fogarty, K. (2004): Object-Oriented Cobol. *Baseline*, Vol 1, No. 30, May 2004, p. 78.

Frakes, W. & Fox, C. (1995): Sixteen Questions About Software Reuse. *Communications of the ACM*, Vol. 38, No. 6, pp. 75-87.

Frakes, W. & Isoda, T. (1994): Success Factors of Systematic Reuse. *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 15-19.

Frakes, W. & Terry, C. (1996): Software Reuse: Metrics and Models. ACM Computing Surveys, Vol. 28, No. 2, pp. 415-435.

Franz, M. (1998): The Java Virtual Machine: A Passing Fad? *IEEE Software*, Vol. 15, No. 6, November/December 1998, pp. 26-29.

Gable, G. (1994): Integrating case study and survey research methods: an example in information systems. *European Journal of Information Systems*, Vol. 3, No. 2, pp. 112-126.

Gall, H., Klösch, R. & Mittermeir, R. (1995): Object-Oriented Re-Architecturing. Schäfer, W. & Botella, P. (Eds.). 5th European Software Engineering Conference (ESEC), Sitges, Spain, September 1995. Proceedings, Springer Verlag, Lecture Notes in Computer Science 989, pp. 499-519.

Galliers, R. (1992): Choosing Information Systems Research Approaches. Galliers, R.D. (ed.), *Information Systems Research. Issues, methods and practical guidelines*. Oxford: Blackwell Scientific Publications, pp. 144-162.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995): *Design Patterns. Elements of Reusable Object-Oriented Software*. Reading, Massachusetts, US: Addison-Wesley Longman, Inc.

Garlan, D., Allen, R., & Ockerbloom, J. (1994): Architectural Mismatch: Why Reuse is so hard. *IEEE Software*, Vol. 12, No. 6, November 1995, pp. 17-26.

Gehringer, E. & Manns, M. (1996): OOA/OOD/OOP: What programmers and managers believe we should teach. *Journal of Object-Oriented Programming*, Vol. 9, No. 6, October, pp. 52-60.

Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O. & Pintado, X. (1990): CLASS Management for Software Communities. *Communications of the ACM*, Vol. 33, No. 9, pp. 91-102.

Gillach, J. & Deyo, N. (1993): Empowering the IT-business relationship with objects. *Object Magazine*, Vol. 3, No. 3, September-October, pp. 69-71.

Gillibrand, D. (2000): Essential business objects design. *Communications of the ACM*, Vol. 43, No. 2, pp. 117-119.

Glaser, B. & Strauss, A. (1967): The discovery of grounded theory: Strategies of qualitative research. London: Wiedenfeld and Nicholson.

Glass, R. (1998): Reuse: What's Wrong with This Picture? *IEEE Software*, Vol. 15, No. 3, May/June, 1998, pp. 57-59.

Glass, R. (1999): A Snapshot of Systems Development Practice. *IEEE Software*, Vol. 16, No. 3, May/June, 1999, pp. 111-112.

Graham, I. (1995): A non-procedural process model for object-oriented software development. *Report on Object Analysis and Design*, Vol. 1, No. 5, pp. 10-11.

Graham, I. (2001): *Object-Oriented Methods. Principles & Practice*. Third Edition. Harlow, England: Addison-Wesley Publishing Company.

Grinzo, L. (1998): The Unbearable Lightness of Being Reusable. *Dr. Dobb's Journal*, Online op-Eds. Available from: http://www.ddj.com/documents/ [Accessed 23 September 1998].

Gummesson, E. (1991): *Qualitative Methods in Management Research*. Newbury Park (US): Sage Publications, Inc.

Gunn, H. (2002): Web-based Surveys: Changing the Survey Process. *First Monday*, Vol 7, No. 12. Available from: http://firstmonday.org/issues/issue7_12/gunn/index.html [Accessed 16 December 2002].

Hamilton, S. & Ives, B. (1992): MIS Research Strategies. Galliers, R.D. (ed.), *Information Systems Research. Issues, methods and practical guidelines*. Oxford: Blackwell Scientific Publications, pp. 132-143.

Hanseth, O. & Monteiro, E. (1994): Modelling and the Representation of Reality: Some Implications of Philosophy on Practical Systems Development. *Scandinavian Journal of Information Systems*, Vol. 6, No. 1, pp. 25-46.

Harmon, P. (1995): Object-Oriented AI: A Commercial Perspective. *Communications of the ACM*, Vol. 38, No. 11, pp. 80-86.

Harrington, J. (1995): *C++ and the Object-Oriented Paradigm*. An IS Perspective. New York: John Wiley & Sons, Inc.

Harrison, R., Samaraweera, L., Dobie, M. & Lewis P. (1996): Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal*, Vol. 11, No. 4, July 1996, pp. 247-254.

Harrison, W. & Ossher, H. (1993): Subject-Oriented Programming (A Critique of Pure Objects). OOPSLA'93, *ACM SIGPLAN Notices*, Vol. 28, No. 10, pp. 411-428.

Hatton, L. (1998): Does OO Sync with How We Think? *IEEE Software*, Vol. 15, No. 3, May/June 1998, pp. 46-54.

Heller, M. (2003): The Water Language (actually untitled). *Byte.com*, 2/10/2003. Business Source Premier Database on the Web.

Helton, D. (1998): The Impact of Large-Scale Component and Framework Application Development on Business. Weck, W., Bosch, J. & Szyperski, C. (Eds.) Proceedings of the WCOP'98 Third International Workshop on Component-Oriented Programming, Brussels, Belgium, July 21, 1998. Turku Centre for Computer Science, TUCS General Publication, No 10, September 1998.

Henders, R. (1998): An Evolutionary approach to application development with object technology. *IBM Systems Journal*, Vol. 37, No. 2, pp. 181-188.

Henderson-Sellers, B. (1992): *A Book of Object-Oriented Knowledge. Object-Oriented Analysis, Design and Implementation: A new approach to software engineering.* Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Henderson-Sellers, B. (1993): The economics of reusing library classes. *Journal of Object-Oriented Programming*, Vol. 6, No. 4, July-August, pp. 43-50.

Henderson-Sellers, B. (1996): *Object-Oriented Metrics. Measures of complexity*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Henderson-Sellers, B. & Edwards, J. (1990): Object-Oriented Systems Life Cycle. *Communications of the ACM*, No. 9, pp. 143-159.

Henderson-Sellers, B. & Edwards, J. (1993): The fountain model of object-oriented system development. *Object Magazine*, Vol. 3, No. 2, pp. 71-79.

Henderson-Sellers, B. & Edwards, J. (1994): *BOOK TWO of Object-Oriented Knowledge: The Working Object. Object-Oriented Software Engineering: Methods and Management.* Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Hohmann, L. (1996): The First Step in Training: Analysis & Design or Implementation Language? *Journal of Object-Oriented Programming*, Vol. 9, No. 6, October 1996, pp. 61-63.

Holm, P. (1998): *Objektorienterad programmering och Java*. Lund, Sweden: Studentlitteratur. In Swedish.

Hopkins, T. (1992): Object-oriented systems. *Software Engineering Journal*, Vol. 7, No. 2, pp. 82-83.

Hopkins, J. (2000): Component Primer. *Communications of the ACM*, Vol. 43, No. 10, pp. 27-30.

Hu, C. (2005): Dataless Objects Considered Harmful. *Communications of the ACM*, Vol. 48, No. 2, pp. 99-101

Höydalsvik, G. & Sindre, G. (1993): On the purpose of Object-Oriented Analysis. OOPSLA'93, *ACM SIGPLAN Notices*, Vol. 28, No. 10, pp. 240-255.

Isoda, S. (2001): Object-oriented real-world modelling revisited. *The Journal of Systems and Software*, Volume 59, Issue 2, November 2001, pp. 153-162.

Ives, B., Hamilton, S. & Davis, G. (1980): A Framework for Research in Computer-Based Management Information Systems. *Management Science*, Vol. 26, No. 9, September 1980, pp. 910-934.

Jacobson, I. (1993): Is Object Technology Software's Industrial Platform? *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 24-30.

Jacobson, I., Christerson, M., Jonsson, P. & Övergaard, G. (1992): *Object-Oriented Software Engineering, A Use Case Driven Approach*. Reading, Massachusetts, US: Addison-Wesley, Publishing Company.

Jacobson, I., Booch, G. & Rumbaugh, J. (1999): The Unified Process. *IEEE Software*, Vol. 16, No. 3, May/June, 1999, pp. 96-102.

Jacobson, I., Ericsson, M. & Jacobson, A. (1995): *The Object Advantage: Business Process Reengineering with Object Technology*. Reading, Massachusetts US: Addison-Wesley, Publishing Company. ACM Press.

Jaime, A., Irastorza, A. & Diaz, O. (2000): Dimensiones en el diseño basado en componentes. IScDIS' 2000, I^{er} Taller de Trabajo en Ingeniería del Software basada en Componente Distribuidos. Valladolid, 9 de noviembre 2000, pp. 1-4. In Spanish.

Jarzabek, S. & Knauber, P. (1999): Generative Approaches. Nierstrasz, O. & Lemoine, M. (Eds.). 7th European Software Engineering Conference, Toulouse, France, September 1999. Proceedings Springer Verlag, Vol. 24, No. 6, Nov. 1999, pp. 429-445.

Jean, F-C. (1992): EEC embraces OT for hospital information systems. *Object Magazine*, Vol. 2, No. 2, July-August 1992, pp. 49-53.

Jenz, D. (1999a): Einführung der Komponentenbasierten Anwendungsentwicklung – Auswirkungen auf die Organisation. Jenz & Partner – Views on organization. Available from: http://www.jenzundpartner.de/download/voo.reuse.pdf [Accessed 19 September 2002]. In German.

Jenz, D. (1999b): Komponentenbasierte Anwendungsentwicklung – aber nicht ohne organisatorischen Unterbau. Jenz & Partner – Views on organization. Available from: http://www.jenzundpartner.de/download/voo.reuse.pdf [Accessed 19 September 2002]. In German.

Jenz, D. (1999c): Komponentenbasierte Anwendungsentwicklung – ist Return on Investment erzielbar? Jenz & Partner – Views on organization. Available from: http://www.jenzundpartner.de/download/voo.reuse.pdf [Accessed 19 September 2002]. In German.

Jézéquel, J-M. & Meyer, B. (1997): Design by Contract: The Lessons of Ariane. Eiffel Software. Available from: http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html [Accessed 1 January 2003]. A variant of the article appeared in the journal *IEEE Computer*, as a part of the Component and Object Technology department in the February 1997 issue.

Jick, T. (1979): Mixing Qualitative and Quantitative Methods: Triangulation in Action. *Administrative Science Quarterly,* Vol. 24, No. 4, pp. 602-611.

Johnson, L. (1997a): Object-oriented technology eases development. *National Underwriter* (Life & health/financial services ed.), Feb 17, Vol. 101, No. 7, pp 40-42.

Johnson, R.E. (1997b): FRAMEWORKS = (Components + Patterns). *Communications of the ACM*, Vol. 40, No. 10, pp. 39-42.

Johnson, R.A. (2000): The Ups and Downs of Object-Oriented Systems Development. *Communications of the ACM*, Vol. 43, No. 10, pp. 69-73.

Johnson, R.A. (2002): An Examination of Empirical Research in Object-Oriented Analysis and Design. *Journal of Computer Information Systems*, Vol. 42, No. No. 3, Spring 2002, pp. 11-15.

Johnson, R.A., Hardgrave, E. & Doke, E. (1999): An Industry analysis of developer beliefs about object-oriented systems development. *Database for Advances in Information Systems*, Vol. 30. No. 1, pp. 47-64.

Jolin, A. (1996): Usability and Class Library Design. *Dr. Dobb's Journal*, October, pp. 16-22.

Joos, R. (1994): Software Reuse at Motorola. *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 42-47.

Järvinen, P. (1994): On Research Methods. Tampere, Finland: Opinpajan Kirja.

Järvinen, P. & Järvinen, A. (1995): Tutkimustyön Metodeista. Tampere, Finland: Opinpaja Oy. In Finnish.

Kaasböll, J. (1993): Object-oriented Models of Functionally Integrated Computer Systems. Bansler, J., Bödker, K., Kensing, F., Nörbjerg, J. & Pries-Heje, J. (Eds.). Department of Computer Science, University of Copenhagen. Proceedings of the 16th IRIS, 1993, pp. 236-251.

Kan, S. (1995): *Metrics and Models in Software Quality Engineering.* Reading, Massachusetts, US: Addison-Wesley Publishing Company.

Kaindl, H. (1999): Difficulties in the Transition from OO Analysis to Design. *IEEE Software*, Vol. 16, No. 5, September/October 1999, pp. 94-102.

Kaplan, B & Duchon, D. (1988): Combining Qualitative and Quantitative Methods in Information Systems Research: A Case Study. *MIS Quarterly*, Vol. 12, No. 4, December 1988, pp. 571-586.

Khoshafian, S. & Abnous, R. (1995): *Object Orientation*. New York, US: John Wiley & Sons, Inc.

King, R. (1989): My Cat is Object-Oriented. Kim, W. & Lochovsky, F. (eds.), *Object-Oriented Concepts, Databases and Applications*. New York, US: ACM press, Addison-Wesley Publishing Company, pp. 23-30.

Koehler, S. (1992): Objects in insurance. Gaining the competitive edge in financial services. *Object Magazine*, Vol. 2, No. 2, July-August 1992, pp. 37-41.

Konstantas, D. (1995): Interoperation of Object-Oriented Applications. Nierstrasz, O. & Tsichritzis, D. (Eds.), pp. 69-95, *Object-Oriented Software Composition*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Korson, T. & McGregor, J. (1990): Understanding Object-Oriented: a Unifying Paradigm. *Communications of the ACM*, Vol. 33, No. 9, pp. 40-60.

Korson, T. & Vaishnavi, K. (1992): Managing Emerging Software Technologies: A Technology Transfer Framework. *Communications of the ACM*, Vol. 35, No. 9, pp. 101-111.

Koskimies, K. (1995): Olio-ohjelmointi ja oliokielet. Tampere, Finland: Tampereen Yliopisto Tietojenkäsittelyopin Laitos, Julkaisusarja C. C-1994-2, Syyskuu 1995. University of Tampere, Department of Computer Science. In Finnish.

Koskimies, K. (1997): *Pieni Oliokirja*. Espoo, Finland: Suomen Atk-kustannus. In Finnish.

Kozaczynski, W. & Kuntzmann-Combelles, A. (1993): What it Takes to Make OO Work. *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 20-23.

Krajnc, M. (1997): *Why Component-Oriented Programming?* Oberon Microsystems, Zurich, Switzerland.

Kung, D., Gao, J., Hsia, P., Toyoshima, Y, Chen, C., Kim, Y. & Song, Y. (1995): Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of the ACM*, Vol. 38, No. 10, pp. 75-87.

Kölling, M. & Rosenberg, J. (2002): BlueJ – *The Hitch-Hikers Guide to Object Orientation*. Technical Reports 2002, No 2, September 2002, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark.

Körner, S. & Wahlgren, L. (2002): Statistiska metoder. University College of Borås, Sweden. Available from: http://www.hb.se/vhb/student/schema/stat.htm [Accessed 3 December 2002]. In Swedish.

LaBoda, D. & Ross, J. (1997): Travellers Property Causality Corporation: Building an Object Environment for Greater Competitiveness. SIM International Paper Awards Competition. Available from: http://www.simnet.org/public/ programs/capital/97paper/paper3.html [Accessed 3 December 2002].

Lam, J. (1997): Object-Oriented Technology. Available from: http://disc.cba.uh.edu/~rhirsch/spring97/lam1/hope.htm [Accessed 11 October 11, 2002].

Larman, C. (2002): *APPLYING UML AND PATTERNS. An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, New Jersey, US: Prentice Hall PTR.

Lauesen, S. (1998): Real-Life Object-Oriented Systems. *IEEE Software*, Vol. 15, No. 2, March/April 1998, pp. 76-83.

Lawrence, R. & Pfleeger, S. (1995): Reuse Measurement and Evaluation. *American Programmer*, Vol. 8, No. 11, pp. 25-30.

Lee, A. (1989): A Scientific Methodology for MIS Case Studies. *MIS Quarterly*, Vol. 13, No. 1, March 1989, pp. 33-59.

Lee, A. & Baskerville, R. (2003): Generalizing Generalizability in Information Systems Research. *Information Systems Research*, Vol. 14, No. 3, pp. 221-243.

Liao, S., Cheung, L. & Liu, W. (1999): An Object-Oriented System for the Reuse of Software Design Items. *Journal of Object-Oriented Programming*, Vol. 11, No. 8, January 1999, pp. 22-28.

Lieberherr, K., Holland, I. & Riel, A. (1988): Object-Oriented Programming: An Objective Sense of Style. Conference on Object-Oriented Programming Systems, Languages & Applications, (OOPSLA)'88. Proceedings, pp. 323-334.

Lieberherr, K. & Xiao, C. (1993): Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, pp. 313-343.

Lieberherr, K. (2005): Law of Demeter. Available from: http://www.ccs.neu.edu/home/lieber/LoD.html [Accessed 8 August 2005].

Lientz, B. & Swanson, E. (1981): Problems in application software maintenance. *Communications of the ACM*, Vol. 24, No. 11, pp. 763-769.

Lim, W. (1994): Effects of Reuse on Quality, Productivity and Economics. *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 23-31.

Lotsson, A. (1996): Mycket objekt men lite orientering. *Computer Sweden*, 15.11.1996, No. 71, P. 18. In Swedish.

Love, Tom (1993): *Object Lessons*. New York, New York, US: SIGS Books, Inc.

Lundahl, U. & Skärvad, P-H. (1999): *Utredningsmetodik för samhällsvetare och ekonomer*. Lund, Sweden: Studentlitteratur. In Swedish.

Madsen, O. (1995): Open Issues in Object-oriented Programming - A Scandinavian Perspective. *Software - Practice and Experience,* Vol. 25, No. 4, December 1995, pp. 3-43.

Malan, R., Coleman, D. & Letsinger, R. (1995): Lessons from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard. Conference on Object-Oriented Programming Systems, Languages & Applications, (OOPSLA)'95, Austin, Texas, US. Proceedings, pp. 33-46.

Malan, R., Letsinger, R. & Coleman, D. (1996): *Object-Oriented Development at Work. Fusion in the Real World*. Upper Saddle River, Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Manhes, S. (1998): La réutilisabilité: Patterns et Frameworks. Les patterns métiers: extraction dans l'existant logiciel, rapport de DEA, IRIN, université de Nantes. Available from: http://www.stm.tj/reuse/Accueil.htm [Accessed 11 October 2002]. In French.

Maring, B. (1996): Object-Oriented Development of Large Applications. *IEEE Software*, Vol. 13, No. 3, May 1996, pp. 33-40.

Martin, J. & Odell, J. (1992): *Object-Oriented Analysis and Design.* Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Martin, J. & Odell, J. (1995): *Object-Oriented Methods: A Foundation.* Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Martin, S., Yen, D. & Chou, D. (2001): Object-oriented technology for electronic commerce systems. *Human Systems Management*, Vol. 20, No. 2, pp. 161-169.

Mathiassen, L., Munk-Madsen, A., Nielsen, P. & Stage, J. (2000): *Object-oriented analysis and design*. Aalborg, Denmark: Marko Publishing ApS.

McCabe, T. & Butler, C. (1989): Design Complexity Measurement and Testing. *Communications of the ACM,* December 1989, Vol. 32, No. 12, PP. 1415-1425.

McClure, C. (1996): Experiences from the OO Playing Field. Extended Intelligence, Inc., US. Available from: http://www.reusability.com/papers6.html [Accessed 11 October 2002].

McGinnes, S. (1992): How Objective is Object-Oriented Analysis? Loucopoulos, P. (Ed.). Advanced Information Systems Engineering, 4th International Conference CAiSE '92, Manchester, UK, May 1992. Proceedings, Springer-Verlag, pp. 1-16.

McGregor, M. (1996): Too Many Cooks could spoil the Broth. Object World UK 1996 Report and directory. EvolveIT.net. Available from: http://markmcgregor.com/art_cook.htm [Accessed 11 October 2002].

Mellor, S. & Johnson, R. (1997): Why Explore Object Methods, Patterns and Architectures? *IEEE Software*, Vol. 14, No. 1, January/February 1997, pp. 27-30.

Mendenhall, W., Reinmuth, J. & Beaver, R. (1993): *Statistics for Management and Economics*. Belmont, California, US: Duxbury Press.

Meyer, B. (1988): *Object-oriented Software Construction*. Hemel Hempstead, UK: Prentice Hall International Ltd.

Meyer, B. (1995): *Object Success. A Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process*. Hemel Hempstead, Hertfordshire, UK: Prentice Hall International (UK) Ltd.

Meyer, B. (1997a): A Really Good Idea. Eiffel Software. Available from: http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/idea/page.html [Accessed 1 January 2003]. A variant of the article appeared in the journal *IEEE Computer*, as a part of the Component and Object Technology department in the February 1997 issue.

Meyer, B. (1997b): The next software breakthrough. Eiffel Software. Available from: http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/breakthrough/page.html [Accessed 1 January 2003]. A variant of the article appeared in the journal *IEEE Computer*, as a part of the Component and Object Technology department in the February 1997 issue.

Meyer, B (1998): The role of object-oriented metrics. Eiffel Software. Available from: http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/metrics/ page.html [Accessed 1 January 1 2003}. A variant of the article appeared in the journal *IEEE Computer*, as a part of the Component and Object Technology department in the November 1998 issue.

Miah, S. (1997): Critique of the Object-oriented Paradigm: Beyond Object-Orientation. Part of a book. Available from: http://members.aol.com/shaz7862/critique.htm [Accessed 11 October 2002].

Mikhajlov, L. (1999): *Software Reuse Mechanisms and Techniques: Safety Versus Flexibility*. Turku, Finland: Turku Centre for Computer Science TUCS Dissertations No 21, Åbo Akademi University.

Mili, H., Mili, F. & Mili, A. (1995): Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, June 1995, pp. 528-561.

Mili, A., Yacoub, S., Addy, E. & Mili, H. (1999): Toward an Engineering Discipline of Software Reuse. *IEEE Software*, Vol. 16, No. 5, September/October 1999, pp. 22-30.

Monarchi, D. & Puhr, G. (1992): A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, Vol 35, No. 9, pp. 35-47.

Monroe, R., Kompanek, A., Melton, R. & Garlan, D. (1997): Architectural Styles, Design Patterns and Objects. *IEEE Software*, Vol. 14, No. 1, January/February 1997, pp. 43-52.

Morris, D., Evans, G., Green, P. & Theaker, C. (1996): *Object Oriented Computer Systems Engineering*. London: Springer-Verlag.

Murer, T. (1997): The Challenge of the Global Software Process. Weck, W., Bosch, J. & Szyperski, C. (Eds.). WCOP'97 Second International Workshop on Component-Oriented Programming, Jyväskylä, Finland, June 9, 1997. Proceedings, Turku Centre for Computer Science, TUCS General Publication, No 5, September 1997, pp. 359-363.

Murphy, N. (2001): What Have the Romans Ever Done For Us? *Embedded Systems Programming*, Vol. 14, No. 10, September 2001, pp. 41-46.

Musakka, L. (1996): Oliotekniikka nopeuttaa sovelluskehitystä huimasti. *SAS Institute oy:n asiakaslehti*, No. 2, p. 5. In Finnish.

Mylopoulos, J., Chung, L. & Yu, E. (1999): From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, Vol. 42, No. 1, pp. 31-37.

Mörch, A., Stevens, G., Won, M., Klann, M., Dittrich, Y. & Wulf, V. (2004): Component-Based Technologies for End-User Development. *Communications of the ACM*, Vol. 47, No. 9, pp. 59-62.

Nandhakumar, J. & Jones, M. (1997): Too close for comfort? Distance and engagement in interpretative information systems research. *Information Systems Journal,* Vol. 7, No. 2, pp. 109-131.

Nerson, J-M. (1992): Applying Object-Oriented Analysis and Design. *Communications of the ACM*, Vol. 35, No. 9, pp. 63-74.

Newsted, P., Huff, S. & Munro, M. (1998): Survey Instruments in Information Systems. *MIS Quarterly*, December 1998, pp. 553-554.

Nierstrasz, O. (1989): A Survey of Object-Oriented Concepts. Kim, W. & Lochovsky, F. (Eds.), *Object-Oriented Concepts, Databases and Applications*. New York: ACM press, Addison-Wesley Publishing Company, pp. 3-21.

Nierstrasz, O. & Dami, L. (1995): Component-Oriented Software Technology. Nierstrasz, O. & Tsichritzis, D. (Eds.), *Object-Oriented Software Composition*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc., pp. 3-28.

Nierstrasz, O., Gibbs, S. & Tsichritzis, D. (1992): Component-Oriented Software Development. *Communications of the ACM*, Vol. 35, No. 9, pp. 160-165.

Noack, J. & Schienmann, B. (1999): Introducing OO Development in a Large Banking Organization. *IEEE Software,* Vol. 16, No. 3, May/June 1999, pp. 71-81.

Nokso-Koivisto, M. (1995): Oppi tulee kantapään kautta. *Tietoviikko*, No. 35, P. 15. In Finnish.

North, K. (1997): ODBC Drivers. Available from: http://www.sqlsummit.com/ODBCVend.htm [Accessed 7 February 2005].

Nowicki, A. & Kosiak, M. (1996): Premises of Object-Oriented Approach Adoption in Information Systems. Research Experiences. Wrycza, S. & Zupancic, J. (Eds.). Proceedings of the Fifth International Conference on Information Systems Development, Gdansk/Poland, September 24-26, 1996.

Nunamaker, J., Chen, M. & Purdin, T. (1991): Systems Development in Information Systems research. *Journal of Management Information Systems,* winter 1990-1991, Vol. 7, No. 3, pp. 89-106.

O'Connor, J., Mansour, C. & Campbell, G. (1994): Reuse in Command – and – Control Systems. *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 70-79.

Odell, J. (2000): Objects and Agents Compared. *Journal of Object Technology,* Vol. 1, No. 1, May–June 2002, pp. 41-53.

Ooil, S. (2002): Object-oriented Programming Oversold! Laboratory Report. Available from: http://www.geosities.com/SiliconValley/Lab/6888/oopbad.htm [Accessed 11 October 2002].

Paetau, P. (1995): En studie av det objektorienterade paradigmets tillämpbarhet för utvecklandet av en activity-based costing applikation. Licentiate thesis in Computer Science and Information Systems at Swedish School of Economics and Business Administration, Helsinki, Finland. In Swedish.

Page-Jones, M. (1992a): Comparing Techniques by Means of Encapsulation and Conscience. *Communications of the ACM*, Vol. 35, No. 9, pp. 147-151.

Page-Jones, M. (1992b): Object orientation: the importance of being earnest. *Object Magazine*, Vol. 2, No. 2, July-August 1992, pp. 11-14.

Page-Jones, M. (1998): The basic pitfalls of adopting object orientation. Object Orientation: Making the Transition. Wayland Systems, Inc, Washington, US. Available from: http://www.elj.com/elj/v1/n3/mpj/ [Accessed 11 October 2002}.

Pancake, C. (1995): The Promise and the Cost of Object Technology: A Five-Year Forecast. *Communications of the ACM*, Vol. 38, No. 10, pp. 33-49.

Pang, C. (1996): Systems modelling and design with objects and patterns. *Journal of Object-Oriented Programming*, Vol. 9, No.2, May 1996, pp. 32-41.

Pant, Y., Henderson-Sellers, B. & Verner, J. (1996): Generalization of object-oriented components for reuse: Measurements of effort and size change. *Journal of Object-Oriented Programming*, Vol. 9, No. 2, May 1996, pp. 19-31.

Parnas, D. (1972): On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058.

Parson, J. & Wand, Y. (1997): USING OBJECTS for Systems Analysis. *Communications of the ACM*, Vol. 40, No. 12, pp. 104-110.

Pawson, R. (2002): The Naked Truth About Business Systems. Computer Sciences Corporation. Available from: http://www.csc.com/aboutus/cscworld/summer02/nakedtruth.shtml [Accessed 11 October 2002].

Penker, M. (1994): *Praktikfall av objektorienterad systemutveckling*. Lund, Sweden: Studentlitteratur. In Swedish.

Perez, C. (2001): The Strix Object Persistence Engine. *Dr. Dobbs's Journal*, August 2001, pp. 40-47.

Petre, L. (2000): *Components vs. Objects*. Turku, Finland: Turku Centre for Computer Science TUCS Technical Report No 370. TUCS is a centre for University of Turku, Åbo Akademi University and Turku School of Economics and Business Administration.

Pickering, C. (1996): *Survey of Advanced Technology 1996*. Overland Park, US: Systems Development, Inc.

Pidd, M. (1995): Object-orientation, Discrete Simulation and the Three-Phase Approach. *Journal of the Operational Research Society*, Vol. 46, No. 3, pp. 362-374.

Pittman, M. (1993): Lessons Learned in Managing Object-oriented Development. *IEEE Software*, Vol. 10, No. 1, January, pp. 43-53.

Pomberger, G. & Blaschek, G. (1996): *Object-Orientation and Prototyping in Software Engineering*. Hemel, Hempstead, Hertfordshire, UK: Prentice Hall Europe.

Prata, S. (1991/1992): *C++ Programmering*. Göteborg, Sweden: Pagina International, AB. In Swedish.

Pree, W. (1997): Component-Based Software Development – A New Paradigm in Software Engineering? *Software – Concepts and Tools*, Vol. 18, No. 18, pp. 169-174.

Pressman, R. (2000): *Software Engineering. A Practitioner's Approach*. European Adaptation. London, UK: McGraw-Hill International.

Putkonen, A. (1994): *A Methodology for Supporting Analysis, Design and Maintenance of Object-oriented systems*. Kuopio, Finland: Kuopio University Printing Office. Doctoral thesis.

Radding, A. (1999): Fast track to app success. *InformationWeek*, Manhasset, July 26.

Radin, G. (1996): Object technology in perspective. *IBM Systems Journal*, Vol. 35, No. 2, pp. 124-127.

Ralston, A. (editor), Reilly, E. & Hemmendinger, D. (2003): Encyclopedia of Computer Science. New York, US: John Wiley & Sons Inc.

Ramaswamy, R. (2001): Mentoring Object-Oriented Projects. *IEEE Software*, Vol. 18, No. 3, May/June, 2001, pp. 36-40.

Reeves, C. & Bednar, D. (1994): Defining Quality: Alternatives and Implications. *Academy of Management Review*, Vol. 19, No. 3, pp. 419-445.

Reinwald, B., Lehman, T., Pirahesh, H. & Gottemukkala, V. (1996): Storing and using objects in a relational database. *IBM Systems Journal*, Vol. 35, No. 2, pp. 172-189.

Repa, V. (1996): Object Life Cycle Modelling in the Client-Server Applications Development Using Structured Methodology. Wrycza, S. & Zupancic, J. (Eds.). Proceedings of the Fifth International Conference on Information Systems Development, Gdansk/Poland, September 24-26, 1996, pp. 617-620.

Riihimaa, J. (2004): *Taxonomy of information and communication technology system innovations adopted by small and medium sized enterprises*. Tampere, Finland: Tampere University Press. Doctoral thesis.

Rinat, R. & Magidor, M. (1996): Metaphoric Polymorphism: Taking Code Reuse One Step Further. Pierre Cointe (Ed.). ECOOP '96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 1996. Proceedings, Springer Verlag, Lecture Notes in Computer Science 1098, pp. 449-471.

Rofrano, J. (1999): Java Portability by Design. *Dr. Dobb's Journal*, June 1999, pp. 34-41.

Rosson, M. & Alpert, S. (1990): The Cognitive Consequences of Object-oriented Design. *Human-Computer Interaction*, Vol. 5, No. 4, pp. 345-379.

Rothering, D. (1994): Development of an OO Infrastructure for Mainframe Database Applications. OOPSLA'94, *ACM SIGPLAN Notices*, Vol. 29, No. 10, pp. 205-211.

Rubin, K. & Goldberg, A. (1992): Object Behaviour Analysis. *Communications of the ACM*, Vol. 35, No. 9, pp. 48-62.

Rudberg, B. (1990): *De första stegen i statistik*. Stockholm, Sweden: Bokförlaget Natur och Kultur. In Swedish.

Rumbaugh, J. (1996): A matter of intent: How to define subclasses. *Journal of Object-Oriented Programming*, Vol. 9, No. 5, September 1996, pp. 5-9 & p. 18.

Rumbaugh, J. (1997): OO Myths: Assumptions from a language view. *Journal of Object-Oriented Programming*, Vol. 9, No. 9, February 1997, pp. 5-7 & p. 48.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991): *Object-Oriented Modelling and Design*. Englewood Cliffs, New Jersey, US: Prentice Hall, Inc.

Rumbaugh, J., Jacobson, I. & Booch G. (2000): *The Unified Modelling Language Reference Manual*. Reading, Massachusetts, US: Addison-Wesley Longman, Inc.

Räisänen, S. (1997a): Olioajattelun onnistunut käyttöönotto. Haapa-aho, H., Hakulinen, H., Hirvonen, A., Kupias, T-K., Laine, H., Niinistö, H., Räisänen, S. & Virkki, P. (Eds.). *Olioiden Maihinnousu*. Espoo, Finland: Suomen Atk-kustannus. In Finnish.

Räisänen, S. (1997b): Uudelleenkäyttö. Haapa-aho, H., Hakulinen, H., Hirvonen, A., Kupias, T-K., Laine, H., Niinistö, H., Räisänen, S. & Virkki, P. (Eds.). *Olioiden Maihinnousu*. Espoo, Finland: Suomen Atk-kustannus. In Finnish.

Sanguinetti, J. (2000): Future is Object-Oriented. *Electronic Engineering Times*, July 24, No. 1123, p. 72.

Saunders, M., Lewis, P. & Thornhill, A. (2000): *Research Methods for Business Students*. Essex, UK: Pearson Education Limited.

Schmidt, D. & Fayad, M. (1997): Lessons Learned. Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM*, Vol. 40, No. 10, pp. 85-87.

Selic, B., Gullekson, G. & Ward, P. (1994): *Real-Time Object-Oriented Modelling*. New York, US: John Wiley & Sons Inc.

Sheetz, S. (2002): Identifying the difficulties of object-oriented development. *The Journal of Systems and Software*, Volume 64, Issue 1, October 2002, pp. 23-36.

Sheetz, s. & Tegarden, D. (1996): Perceptual complexity of object-oriented systems: a student view. *Object-oriented Systems*, Vol. 3, December 1996, pp. 165-195.

Shlaer, S. & Mellor, S. (1988): *Object-Oriented Systems Analysis Modelling the World in Data*. Englewood Cliffs, New Jersey, US: Yourdon Press, Prentice-Hall, Inc.

Shlaer, S. & Mellor, S. (1992): *Object Lifecycles Modelling the World in States*. Englewood Cliffs, New Jersey, US: Yourdon Press, Prentice-Hall, Inc.

Silveira da, G. (2000): Spontaneous Software: A Web-based, Object Computing Paradigm. 22nd International Conference on Software Engineering (ICSE), University of Limerick, Ireland, June 4-11, 2000. Proceedings of ACM and SigSoft, pp. 719-722.

Sim, R. & Wright, G. (2002): The difficulties of learning object-oriented analysis and design: An exploratory study. *Journal of Computer Information Systems*, Winter 2001-2002, pp. 95-100.

Sircar, S., Nerur, S. & Mahapatra, R. (2001): Revolution or evolution? A Comparison of object-oriented and structured system development methods. *MIS Quarterly*, Vol. 25, No. 4, pp. 457-472.

Sklenar, J. (1997): Introduction to OOP in Simula. Available from http://staff.um.edu.mt/jskl1/talk.html#Classes. [Accessed 13 October 2004].

Smith, H. & McKeen, J. (1996): Object-Oriented Technology: Getting Beyond the Hype. *The DATA BASE for Advances in Information Systems*, Vol. 27, No. 2, Spring 1996, pp. 20-29.

Smolander, K, Tahvanainen, V-P. & Lyytinen, K. (1990): How to Combine Tools and Methods in Practice - a field study. Proceedings of the Advanced Information Systems Engineering, 2nd Nordic conference CAiSE '90, Stockholm, Sweden, May 8-10, 1992 / Steinholz, B., Sölvberg, A., & Bergman, L., (eds.), Springer-Verlag, pp. 195-214.

Snyder, A. (1993): The Essence of Objects: Concepts and Terms. *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 31-42.

Solomon, H. (1999): Modelling tools a software must. *Computing Canada*, Vol. 25, No. 18, pp. 25-26.

Sommerville, I. (1992): *Software engineering*. 4th edition. Reading, Massachusetts, US: Addison-Wesley Publishers Ltd.

Sommerville, I. (1996): *Software engineering*. 5th edition. Reading, Massachusetts, US: Addison-Wesley Publishers Ltd.

Sparling, M. (2000): Lessons Learned Through Six Years of Component-Based Development. *Communications of the ACM*, Vol. 43, No. 10, pp. 47-53.

Staringer, W. (1994): Constructing Applications from Reusable Components. *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 61-68.

Steinmann, J, (1992): The Overselling of Object Technology, or How to Fail On Your First Object Project. Object Magazine, Vol. 2, No. 3, September 1992. With postscript 1995 & 1999, available from: http://www.bytesmiths.com/ pubs/9209Overselling.html [Accessed 11 October 2002].

Stevens, P. & Pooley, R. (2000): *Using UML: Software Engineering with Objects and Components*. Harlow, England: Addison-Wesley Publishing Company.

Sutcliffe, A. & Mehandjiev, N. (2004): End-User Development. *Communications of the ACM*, Vol. 47, No. 9, pp. 31-32.

Sutton, R. & Staw, B. (1995): What Theory is Not. *Administrative Science Quarterly,* Vol. 40, No. 3, pp. 371-384.

Swanson, E. & Dans, E. (2000): System life expectancy and the maintenance effort: Exploring their equilibration. *MIS Quarterly*, Vol. 24, No. 2, June 2000, pp. 277-298.

Szyperski, C. (1999): *Component Software. Beyond Object-Oriented Programming*. England, Harlow: Addison-Wesley Publishing Company, Inc.

Taenzer, D., Ganti, M. & Podar, S. (1989): Object-Oriented Software Reuse: The Yoyo Problem. *Journal of Object-Oriented Programming*, Vol. 2, No. 3, September/October 1989, pp. 30-35.

Taivalsaari, A. (1993): *A Critical View of Inheritance and Reusability in Object-oriented Programming*. Jyväskylä, Finland: Jyväskylä studies in computer science, economics and statistics, University of Jyväskylä. Doctoral thesis.

Taylor, D. (1990): *Object-Oriented Technology: A Manager's Guide*. Reading, Massachusetts, US: Addison-Wesley. Second Printing, January 1992.

Taylor, D. (1992): *Object-Oriented Information Systems: Planning and Implementation*. New York, US: John Wiley & Sons, Inc.

Tengvall, J. (2001): *Ohjelmistotuotannon nopeuttaminen*. Oulu, Finland: Department of Electrical Engineering, University of Oulu. Diploma Thesis.

Tepfenhart, W. & Cusick, J. (1997): A Unified Object Topology. *IEEE Software*, Vol. 14, No. 1, January/February 1997, pp. 31-35.

Thomas, D. (1989): In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming*, Vol. 2, No. 1, May/June 1989, pp. 60-63.

Tyma, P. (1998): Why are we using JAVA AGAIN? *Communications of the ACM*, Vol. 41, No. 6, pp. 38-42.

Törnebohm, H. (1997): In the *Focus 97 Dictionary*. Stockholm, Sweden: Nordstedts förlag AB.

Udell, J. (1994): Componentware. *Byte*, May 1994, Vol. 19, No. 6, pp. 46-56.

Undheim, J. (1985): *Statistik från ord till formel*. Lund, Sweden: Studentlitteratur. In Swedish.

VanDoren, E. (1997): Cyclomatic Complexity. Available from: http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html. [Accessed 13 October 2004].

Verity, J. & Schwartz, E. (1991): SOFTWARE MADE SIMPLE; WILL OBJECT-ORIENTED PROGRAMMING TRANSFORM THE COMPUTER INDUSTRY? *Business Week*, September 30, 1991, pp. 58-63.

Verschoor, R. & Low, G. (1994): Software Reusability in Australia. *The Australian Computer Journal*, Vol. 26, No. 4, November 1994, pp. 134-142.

Villeneuve, A. & Fedorowicz, J. (1996): Usage Benefits of Object Orientation. Boston University, School of Management, Boston, US. Presented on the INFORMS Conference on Information Systems & Technology, Washington DC, US, May 5-8, 1996.

Vossos, G., Dillon, T., Zeleznikow, J. & Taylor, G. (1991): The use of object oriented principles to develop intelligent legal reasoning systems. *The Australian Computer Journal*, Vol. 23, No. 1, February 1991, pp. 2-10.

Wadden, G. (1999): Object-oriented technology helps reduce programming risks. *Instrumentation & Control Systems*, Vol. 72, No. 3, p. 83.

Walsham, G. (1995): Interpretative case studies in IS research: nature and method. *European Journal of Information Systems,* Vol. 4, No. 2, pp. 74-81.

Watanabe, S. (1997): Professionalism through OO and Reuse. *IEEE Software*, Vol. 14, No. 1, January/February, 1997, p. 26.

Watson, D. (1999): Java: No Longer A Systems Language. *Byte*, September 2, 1999. Available from: http://www.byte.com/print [Accessed November 18, 2002].

Watson, R., Zinkhan, G. & Pitt, K. (2004): Object-Orientation: A Tool For Enterprise Design. *California Management Review*, Vol. 46, No. 5, Summer 2004, pp. 89-110.

Webster, B. (1995): *Pitfalls of Object-Oriented Development*. New York, New York: M&T Books.

Webster's Encyclopaedic Unabridged Dictionary of the English Language. 1996. New York, US: Gramercy books.

Weck, W. (1997): Inheritance Using Contracts & Object Composition. Weck, W., Bosch, J. & Szyperski, C. (Eds.). Proceedings of the WCOP'97 Second International Workshop on Component-Oriented Programming, Jyväskylä, Finland, June 9, 1997. Turku Centre for Computer Science, TUCS General Publication, No 5, September 1997, pp. 1-21.

Wegenast, D. (1998): Object models speed system development. *Computing Canada*, Vol. 24, No. 40, pp. 31-32.

Welke, R. (1994): The Shifting Software Development Paradigm. *Data Base*. November, Vol. 25, No. 4, pp. 47-56.

Wieringa, R. (1998): A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, Vol. 30, No 4, pp. 459-527.

Wilde, N. & Huitt, R. (1992): Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038-1044.

Wilde, N. & Matthews, P. (1993): Maintaining Object-Oriented Software. *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 75-80.

Wilkie, G. (1993): *Object-Oriented Software Engineering. The Professional Developer's Guide.* Wokingham, England: Addison-Wesley Publishing Company.

Winblad, A., Edwards, S. & King, D. (1990): *Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.

Wirfs-Brock, R. & Johnson, E. (1990): Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, Vol. 33, No. 9, pp. 104-123.

Wirfs-Brock, R., Wilkerson, B. & Wiener, L. (1990): *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey, US: Prentice-Hall, Inc.

Wolber, D. (1997): Reviving Functional Decomposition in Object-Oriented Design. *Journal of Object-Oriented Programming*, Vol. 10, No. 6, October 1997, pp. 31-39.

Wrede, G. (1998): Using old classes in a hierarchy. Interview with programmer and acting senior assistant George Wrede at the Swedish School of Economics and Business Administration, October 1998.

Wybolt, N. (1992): Can it forge a partnership with object technology? *Object Magazine*, Vol. 2, No. 2, July – August 1992, pp. 27-29.

Xenos, M., Stavrinoudis, D., Zikouli, K. & Christodoulakis, D. (2000): Object-Oriented Metrics – A Survey. Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain.

Yin, R. (1994): *Case Study Research. Design and Methods.* Second edition. Thousand Oaks, CA, US: Sage Publications, Inc.

Yourdon, E. (1994): *Object-Oriented Design: An Integrated Approach.* Englewood Cliffs, New Jersey, US: Yourdon Press.

Yourdon, E. & Argila, C. (1996): *Case studies in object-oriented analysis and design*. Upper Saddle River New Jersey, US: Yourdon Press.

Zhang, X. (1999): *User Participation in Object-Oriented Contexts – From Methodological and Practical Perspectives*. Lund, Sweden: Lund Studies in Informatics, No. 13. Lund University. Doctoral thesis.

Zhou, X., Zaslavsky, A., Rasheed, A. & Price, R. (1998): Efficient Object-Oriented Query Optimisation in Mobile Computing Environment. *The Australian Computer Journal*, Vol. 30, No. 2, May 1998, pp. 65-74.

Zilles, S. (1973): Procedural encapsulation: a linguistic protection technique. *ACM SIGPLAN Notices*, Vol. 8, No. 9, September 1973, pp. 142-146.

Appendix 1 – Pilot study: Survey of the use of software development techniques

The questionnaire was sent in Finnish and translated into English at a later stage.

Kysely eri sovelluskehitystekniikoiden käytöstä

Questionnaire on the use of software development techniques

Tämän kyselyn tarkoitus on kerätä teidän näkemyksenne oliotekniikan ja muiden sovelluskehitysmetodien käytöstä yrityksessänne varsinkin määrittelyvaiheessa. Kysely on tehty niin että pystytte vastaamaan siihen nopeasti. Kaikki Vastaukset käsitellään luottamuksellisesti.

The aim of this study is to gather your views on the use of the object-oriented paradigm and other software development techniques, especially in the analysis phase. The questionnaire is designed so that you can answer it rapidly. All the answers will be treated confidentially.

Osa A: Yrityksestänne
Part A: About Your Company

1. Mikä toimialanne on?
   What is your field of business?

2. Kaupunki tai kunta missä yrityksenne sijaitsee
   Town or municipality where your company is situated

3. Yrityksenne koko arvioituna bruttomyyntinä (tai kokonaisbudjetti sellaisille yrityksille jotka eivät ole myyntiyrityksiä)
   The gross sale of your company (or total budget for companies that are not selling)

4. Arvioitu määrä työntekijöitä
   Estimated number of employees

5. Asiakaskuntanne tai ala johon erikoistutte?
   The field of your customers or the field that you are specialized in?

6. Arvioitu määrä työntekijöitä tietojärjestelmien sovelluskehityksessä
   Estimated number of employees performing software development

---

Osa B: Tämänhetkiset Sovelluskehitysprojektinne - The software development projects of today

1. Mitä menetelmiä käytätte sovelluskehityksessä?
   What techniques (methods) do you use in software development?

	Arvioitu prosentti projekteista joissa käytetään mainittua menetelmää Estimated percentage of the projects where the method is used
Talon sisäinen menetelmä In-house method	
Strukturointimenetelmä Structured method	

Prototyyppimenetelmä
Prototyping method

Oliomenetelmä
Object-oriented method

Muu (kuvaile)
Other (describe)

Menetelmää ei käytetä
No method is used

2. Jos ette käytä menetelmää, niin mitkä ovat tärkeimmät syyt siihen?
If you do not use any method, what are the main reasons for this?

3. Mikäli ette käytä menetelmää tällä hetkellä, oletteko aikeissa ottaa sellainen käyttöön?
If you do not presently use any method, do you intend to start using one?

Vuoden sisällä	Myöhemmin	Ei tulevaisuudessa
In one year	Later	Not in the future

Mikäli aiotte, minkä menetelmän olette ottamassa käyttöönne?
If you intend, which method do you plan to use?

4. Minkä tyyppiset (*sovellukset)* projektit Teillä on meneillään tällä hetkellä
What kind of (applications) projects do you have right now?

Kirjanpito
Accounting

Markkinointi
Marketing

Yrityshallinto
Management

Taloushallinto
Finance

Henkilöstöhallinto
Human resource management

Muu, Mikä?
Other, Which?

5. Minkälaisissa projekteissa käytätte menetelmää?
In what kind of projects do you use a method?

Kirjanpito
Accounting

1	2	3	4	5
aina				ei koskaan
always				never

Markkinointi
Marketing

1	2	3	4	5
aina				ei koskaan
always				never

Yrityshallinto
Management

1	2	3	4	5
aina				ei koskaan
always				never

Taloushallinto
Finance

1	2	3	4	5
aina				ei koskaan
always				never

Henkilöstöhallinta
Human resource management

1	2	3	4	5
aina				ei koskaan
always				never

Muu, mikä?_____
Other, which?

1	2	3	4	5
aina				ei koskaan
always				never

6. Kuinka suuret projektinne ovat?
How large are your projects?

Arvioikaa projektien määrä joka kokoluokalle (sekä valmiit että keskeneräiset projektit)
Estimate the number of projects for every size category (both completed projects and ongoing projects), time ought to be estimated so that it is how much time it would take for one person to finish the project

	Projektien määrä Number of projects	Käytetty menetelmä Used method
Vähemmän kuin 3 mieskuukautta Less than 3 months		
3-6 mieskuukautta 3-6 months		
6-12 mieskuukautta 6-12 months		
1-3 miesvuotta 1-3 years		
enemmän kuin 3 miesvuotta more than 3 man-years		

7. Tärkeimmät työkalut joita käytätte sovelluskehityksessä
The most important tools that you use in software development

Ohjelmointikieli, mikä?
Programming language, which?

CASE

Tietokantasovellus
Database application

Sovelluskehitin
Application generator

Muu, mikä?
Other, which?

8. Käyttämänne aika sovelluskehityksen eri vaiheisiin, prosentteina kokonaisajasta
Used time for the different phases of software development in percentage of the total time

Määrittely
Analysis

Suunnittelu
Design

Käyttöönotto
Implementation

Ylläpito
Maintenance

9. Määrittelyn eri vaiheet, käyttämänne aika prosentteina kokonaisajasta
The usage of time in percentage of the total time for the different analysis phases

Vaatimusten määrittely
Requirements analysis

Vaatimusmäärittelyraportti
Requirements analysis report

Vaatimusten vahvistaminen
The conformation of the requirements

10. Mitä tiedonkeruumetodeja käytätte määrittelyvaiheessa ja kuinka menestyksellisiä ne ovat olleet?
What information gathering methods do you use in the analysis phase and how successful have they been?

	1= epäonnistunut 1= Unsuccessful				5= Menestys 5= Successful
Haastatteluja Interviews	1	2	3	4	5
Ryhmähaastatteluja Group interviews	1	2	3	4	5
Kysymyslomakkeita Questionnaires	1	2	3	4	5
Tarkkailu Observation	1	2	3	4	5
JAD – Joint Application Design	1	2	3	4	5

GSS – Group Support Systems	1	2	3	4	5
Prototyping	1	2	3	4	5
Videonauhoitus Video recording	1	2	3	4	5
Olemassa olevien dokumenttien analysointi Analysis of existing documents	1	2	3	4	5
Muu mikä?_____ Other, which?	1	2	3	4	5

11. Onko Teillä käytössä vaatimusten kirjaamiseen valmista dokumenttia?
Do you use a ready-made document for the writing of requirements?

Kyllä	Ei
Yes	No

12. Kuinka tärkeänä pidätte käyttäjien osallistuminen sovelluskehitykseen?
How important do you think it is that the end users participate in software development?

1	2	3	4	5
Ei Ollenkaan Tärkeänä Not important at all				Erittäin tärkeänä Very important

---

Osa C: Olioprojektit
Part C: Object-oriented projects

1. Mitkä ovat työkalut joita käytätte oliokehityksessä?
What tools do you use in object-oriented software development?

Ympäristö
Environment

Kieli
Language

CASE

Tietokanta
Database

Muu, Mikä?
Other, Which?

2. Minkälaisissa projekteissa käytätte oliotekniikkaa?
In what kind of projects do you use the object-oriented paradigm?

	Jakauma prosentteina Percentage
Kehitys, Suunnittelu Development, Design	
Prosessien valvonta Supervision of processes	

Hallinto
Management

3. Minkälaisia luokkia käytätte?
What kind of classes do you use?

	Jakauma prosentteina Percentage
Käyttäjätehtävä User task	
Käyttöliittymä User interface	
Systeemitehtävä System task	

4. Arvioikaa kuinka pitkään organisaationne on käyttänyt oliokehitysmenetelmää
Estimate for how long your organisation has been using the object-oriented paradigm.

5. Arvioikaa havaitsemanne hyödyn oliomenetelmästä organisaatiossanne tällä hetkellä
Estimate the benefits of the object-oriented paradigm in your organisation at the moment

1	2	3	4	5
Epäonnistuminen Unsuccessful				Menestyksellinen Successful

6. Onko oliotekniikka tärkein sovelluskehitysmenetelmä
Is the object-oriented paradigm your most important software development technique?

Right now	In one year	Later	Never
Tällä hetkellä	Vuoden sisällä	Myöhemmin	Ei koskaan

7. Mitkä ovat mielestänne tärkeimmät syyt oliotekniikan onnistumiselle tai epäonnistumiselle yrityksessänne?
According to you, which are the most important reasons for the success or failure of the object-oriented paradigm in your company?

8. Mitä ovat tärkeimmät syyt oliotekniikan käytöstä projekteissanne?
What are the most important reasons for the use of the object-oriented paradigm in your projects?

9. Jos ette käytä oliotekniikkaa kaikissa projekteissanne, niin miksi ette?
If you do not use the object-oriented paradigm in your company, please say why not?

Appendix 2 – Questionnaire for the survey

Survey of experienced benefits and problems with object-orientation in information system development

The purpose of this study is to collect your perceptions of experienced benefits and problems with object-orientation in **information systems development** in your organisation. The questionnaire has been designed so that you can answer it quickly. All responses are confidential. *Thank you for taking the time to complete this survey.*

Section A: General questions

I. Approximate number of employees in your company:

II. What is the approximate turnover of your company?

III. In what business field are most of your clients?

IV. What is your position in the company?

1.Have you been using object-orientation in information systems development?

Yes	□
No	□
Not sure	□

If the answer to this question is 'Yes', please go to question 3.

If the answer to this question is 'No', please answer question 2, and then the interview is complete.

2. If you **have not** been using object-orientation in information systems development, please state why.

- □ Don't know what object-orientation is
- □ Don't want to use object-orientation
- □ Object-orientation is too complex
- □ Object-orientation is still too immature
- □ Difficult to carry out object-oriented testing
- □ Lack of software developers trained in object-orientation
- □ Lack of software developers who are experienced in object-orientation
- □ Object-oriented software development is too expensive
- □ There is a lack of object-oriented components to reuse
- □ Object-oriented reuse is problematic
- □ Object-oriented analysis is problematic
- □ Object-oriented design is problematic
- □ Lack of object-oriented databases
- □ Difficulties in integrating object-orientation with traditional databases
- □ Difficulties in integrating object-orientation with legacy systems
- □ Other reason, which?

Section B: Benefits

Management of Complexity

3. Have you found the object-oriented paradigm useful when developing large-scale and complex information systems?

Yes	□
No	□
Not sure	□

Productivity and faster development

4. Have you found that object-oriented information system development has been more productive and faster than traditional information system development?

I. Has object-oriented information system development been:

	Yes	No	Not sure
More productive?	□	□	□
Faster?	□	□	□

Quality and usability

5. Have you experienced that the quality of object-oriented systems has been better than the quality of traditional systems?

Yes	□
No	□
Not sure	□

Natural and better mapping to the problem domain

6. Has there been a better and more 'natural' communication between information systems developers and end users because of using the object-oriented paradigm?

Yes	□
No	□
Not sure	□

Maintenance

7. Has maintenance of object-oriented applications been easier or harder than maintenance of traditional functional applications?

Easier	□
Harder	□
Not sure	□

One model

8. Have you seen the object-oriented system development process as a uniform 'one model' from the problem domain to code and maintenance?

Yes □
No □
Not sure □

Reuse

9. Have you used much reuse? Has reuse in the object-oriented paradigm been considered beneficial?

I. Have you used much reuse?

Yes □
No □
Not sure □

II. Has reuse been considered beneficial?

Yes □
No □
Not sure □

10. What do you reuse?

□ Objects
□ Classes
□ Class libraries purchased from vendors
□ Class libraries developed in-house
□ Analysis
□ Design
□ Software components

IF you have used software components:

Have the software components been considered beneficial?

Yes □
No □
Not sure □

□ Other, what?

Portability

11. Have you experienced portability of object-oriented systems as a benefit?

Yes □
No □
Not sure □

Other

12. What other benefits of the object-oriented paradigm, other than those already presented, have you experienced in information systems development?

Section C: Problems

Complexity

13. Do you consider the object-oriented paradigm complex?

Yes	□
No	□
Not sure	□

The object-oriented paradigm is still immature

14. Do you consider the object-oriented paradigm as immature?

Yes	□
No	□
Not sure	□

15. Have you experienced difficulties in finding object-oriented CASE tools, object-oriented databases, object-oriented system development tools or perhaps even objects to reuse?

Have you experienced difficulties to find:

	Yes	No	Not sure	Not used
Object-oriented CASE tools?	□	□	□	□
Object-oriented databases?	□	□	□	□
Object-oriented system development tools?	□	□	□	□
Objects?	□	□	□	□

16. Have you found testing object-oriented information systems, applications or systems as difficult? What testing problems have you experienced?

I. Has testing been difficult?

Yes	□
No	□
Not sure	□

II. What testing problems have you experienced?

- □ It has been difficult to test structures where several member functions call each other in a chain.
- □ It has been difficult to test complex relationships that exist in an object-oriented system. Examples of such relationships are inheritance and polymorphism.
- □ It has been difficult to test because there are very few CASE tools for testing object-oriented systems.
- □ **Other** testing problems, which?

Difficulties in measuring object systems

17. Do you think that a lack of metrics for measuring the object-oriented system is a problem?

Yes □
No □
Not sure □

Training & lack of experience

18. Has it been difficult to find experienced object-oriented software developers and system analysts?

Yes □
No □
Not sure □

Efficiency

19. Have you experienced computer efficiency problems in your object-oriented information system development projects?

Yes □
No □
Not sure □

Costs

20. Have the starting costs been enormous when starting a completely new object-oriented information system or application, due to the lack of artefacts to reuse?

Yes □
No □
Not sure □

Limited usability of components

21. Have you had problems with finding components to reuse?

Yes □
No □
Not sure □

Problems with reuse

22. Has there been a problem with reuse for some of the following reasons:

	Yes	No	Not sure
Software developers do not want to reuse a component, because they feel that it does not work	□	□	□
It is troublesome to learn how the component works	□	□	□
The hierarchy of classes has been a hindrance for reuse	□	□	□

Lack of object-oriented databases

23. Has it been difficult to find an appropriate object-oriented database?

Yes	□
No	□
Not sure	□

24. If a relational database has been used in the object-oriented system development work, which approach for connecting the object-oriented system with the relational database have you used?

- □ The solution of mapping a class to a table has been used
- □ A solution with factory classes has been used
- □ A solution with wrappers has been used
- □ Other solution, which?

Other & lack of support for several important areas like testing

25. Have you experienced a lack of support for something in the object-oriented world?

What other problems or obstacles of the object-oriented paradigm, other than those presented above have you experienced?

Appendix 3 – Questionnaire for the survey in Finnish

Kysely oliotekniikan hyödyistä ja ongelmista tietojärjestelmien kehittelyssä

Tämän kyselyn tarkoitus on kerätä teidän näkemyksenne kokemistanne oliotekniikan hyödyistä ja ongelmista **tietojärjestelmien kehittelyssä**. Kysely on tehty niin että pystytte vastaamaan siihen nopeasti. Kaikki vastaukset käsitellään luottamuksellisesti. *Kiitos ajastanne.*

Osa A: Yleisiä kysymyksiä

I. Yrityksenne arvioitu työntekijämäärä:

II. Yrityksenne arvioitu liikevaihto:

III. Miltä toimialalta asiakkaanne enimmäkseen ovat:

IV. Asemanne yrityksessä:

1. Oletteko käyttäneet oliotekniikkaa tietojärjestelmätyössänne?

Kyllä □
Ei □
En osa sanoa □

Jos vastasitte edelliseen kysymykseen "Kyllä", olkaa ystävällinen ja menkää kysymykseen 3.

Jos vastasitte edelliseen tähän kysymykseen "Ei", vastatkaa vain kysymykseen 2.

2. Miksi **ette ole käyttäneet** oliotekniikkaa tietojärjestelmätyössä?

□ Emme tiedä, mitä oliotekniikka on
□ Emme halua käyttää oliotekniikkaa
□ Oliotekniikka on liian monimutkaista
□ Oliotekniikka on vielä liian kehittymätöntä
□ Oliojärjestelmiä on vaikea testata
□ Oliotekniikkaan koulutetuista sovelluskehittäjistä on pula
□ Kokeneista, oliotekniikan hallitsevista sovelluskehittäjistä on pula
□ Oliopohjainen sovelluskehitystyö on liian kallista
□ Uudelleenkäytettävistä oliokomponenteista on pula
□ Oliopohjainen uudelleenkäyttö on ongelmallista
□ Oliopohjainen määrittely on ongelmallinen
□ Oliopohjainen suunnittelu on ongelmallinen
□ Oliotietokannoista on pula
□ On vaikeata yhdistää oliotekniikkaa tavanomaisten tietokantojen kanssa
□ On vaikeata yhdistää oliotekniikkaa olemassa olevien tietojärjestelmien kanssa
□ Muu syy, mikä? _____

Osa B: Hyödyt

Monimutkaisuuden hallinta

3. Oletteko kokeneet oliotekniikan hyödylliseksi, kun olette kehittänyt laajoja ja monimutkaisia tietojärjestelmiä?

Kyllä □
Ei □
En osa sanoa □

Tuottavuus, nopeampi kehitystyö ja alhaisemmat kustannukset

4. Oletko kokenut, että oliopohjainen sovelluskehitystyö on ollut tuottavampaa tai nopeampaa kuin perinteinen sovelluskehitystyö?

I. Onko oliopohjainen sovelluskehitystyö ollut

	Kyllä	Ei	En osa sanoa
Tuottavampaa:	□	□	□
Nopeampaa:	□	□	□

Laatu ja käyttökelpoisuus

5. Oletteko kokeneet, että oliopohjaiset järjestelmät olisivat parempilaatuisia kuin perinteiset järjestelmät?

Kyllä □
Ei □
En osa sanoa □

Luonnollisuus ja parempi yhteensopivuus sovellusalueen kanssa

6. Onko oliomenetelmien käyttö sovelluskehitystyössä parantanut sovelluskehittäjien ja loppukäyttäjien välistä kommunikaatiota?

Kyllä □
Ei □
En osa sanoa □

Ylläpito

7. Onko oliopohjaisten tietojärjestelmien ylläpito ollut helpompaa tai vaikeampaa kuin perinteisten funktionaalisten tietojärjestelmien ylläpito?

Helpompaa □
Vaikeampaa □
En osa sanoa □

Yksi malli

8. Oletteko pitäneet oliopohjaisen kehitysprosessin yhtenäisenä ”yhtenä mallina’ sovellusalueesta ohjelmointikoodaukseen ja ylläpitoon?

Kyllä □
Ei □
En osa sanoa □

Uudelleenkäyttö

9. Oletteko käyttäneet paljon uudelleenkäyttöä (reuse)? Onko uudelleenkäyttö oliopohjaisessa sovelluskehitystyössä ollut hyödyllinen?

Kyllä □
Ei □
En osa sanoa □

II. Onko uudelleenkäyttöä pidetty hyödyllisenä?

Kyllä □
Ei □
En osa sanoa □

10. Mitä käytätte uudelleenkäytössä?

□ Olioita
□ Luokkia
□ Luokkakirjastoja joita olette ostaneet yrityksiltä
□ Luokkakirjastoja joita olette kehittäneet omassa yrityksessä
□ Määrittelyä
□ Suunnittelua
□ Komponentteja

Jos te olette käyttäneet:

Onko valmiita komponentteja pidetty hyödyllisenä?

Kyllä □
Ei □
En osa sanoa □

□ Muuta, mitä? _____

Siirrettävyys (portability)

11. Oletteko kokeneet, että oliopohjaisten järjestelmien siirrettävyys on hyöty?

Kyllä □
Ei □
En osa sanoa □

Muut

12. Mitä muita kun nyt esille tulleita oliotekniikan hyötyjä olette kokeneet sovelluskehitystyössä?

Osa C: Ongelmia

Monimutkaisuus

13. Pidättekö olioteknikoita monimutkaisina?

Kyllä □
Ei □
En osa sanoa □

Oliotekniikka on vielä kehittymätöntä

14. Pidättekö oliotekniikkaa vielä kehittymättömänä?

Kyllä □
Ei □
En osa sanoa □

15. Oletteko kokeneet, että on vaikeata löytää oliopohjaisia CASE työkaluja, oliopohjaisia tietokantoja, oliopohjaisia sovelluskehitystyökaluja, tai jopa olioita jotka sopivat uudelleenkäyttöön?

Oletteko kokeneet että on vaikeata löytää:

	Kyllä	Ei	En osa sanoa	Emme käytä
Oliopohjaisia CASE työkaluja	□	□	□	□
Oliotietokantoja	□	□	□	□
Oliopohjaisia sovellus-kehitystyökaluja	□	□	□	□
Olioita	□	□	□	□

16. Oletteko kokeneet, että oliopohjaisia tietojärjestelmiä, oliopohjaisia sovelluksia tai oliopohjaisia järjestelmiä ovat vaikeita testata?

I. Onko testaus ollut vaikeata?

Kyllä □
Ei □
En osa sanoa □

II. Mitä testausongelmia olette kokeneet?

□ On ollut vaikeaa testata rakenteita, joissa useamman olion funktiot kutsuvat toisiaan ketjussa.
□ On ollut vaikeaa testata oliojärjestelmän monimutkaisia riippuvuuksia. Esimerkkejä ovat periytyminen ja polymorfismi.
□ On ollut vaikeaa testata, koska oliojärjestelmien testaukseen tarkoitettuja CASE työkaluja ei ole.
□ **Muita** testaukseen liittyviä ongelmia, mitkä?

Oliojärjestelmiä on vaikeita mitata

17. Pidättekö puutteena oliojärjestelmien mittaustavoissa olevia ongelmia?

Kyllä □

Ei □

En osa sanoa □

Koulutuksen ja kokemuksen puute

18. Onko ollut vaikeata löytää kokeneita oliotekniikkaan perehtyneitä sovelluskehittäjiä ja suunnittelijoita?

Kyllä □
Ei □
En osa sanoa □

Tehokkuus

19 Oletteko kokeneet tehokkuusongelmia oliopohjaisissa projekteissanne?

Kyllä □
Ei □
En osa sanoa □

Kustannukset

20. Ovatko aloituskustannukset olleet suuret, kun olette aloittaneet täysin uuden oliopohjaisen tietojärjestelmän tai sovelluksen, koska uudelleenkäytettävistä osista on ollut puute?

Kyllä □
Ei □
En osa sanoa □

Rajoitettu komponenttien käytettävyys

21. Onko teillä ollut vaikeuksia löytää sopivia komponentteja uudelleenkäyttöön?

Kyllä □
Ei □
En osa sanoa □

Ongelmia uudelleenkäytön kanssa

22. Oletteko kokeneet uudelleenkäyttöön liittyvän ongelmia joistakin seuraavista syistä:

	Kyllä	Ei	En osa sanoa
Sovelluskehittäjät eivät halua uudelleenkäyttää komponenttia, koska he väittävät että komponentti ei toimi	□	□	□
On vaikeata oppia miten komponentti toimii	□	□	□
Luokkahierarkia on ollut uudelleenkäytön este	□	□	□

Oliotietokantojen ja yleisten liittymien puute

23. Onko ollut vaikeata löytää sopivaa oliotietokantaa?

Kyllä □
Ei □
En osa sanoa □

24. Kun olette käyttäneet relaatiotietokantaa oliopohjaisessa sovelluskehitystyössä, niin mitä ratkaisua olette käyttäneet, kun olette liittäneet oliojärjestelmän käyttämäänne relaatiotietokantaan?

□ Olemme käyttäneet ratkaisua missä luokka liitetään tauluun
□ Olemme käyttäneet yritysluokkia
□ Olemme käyttäneet kuoria (wrapper)
□ Muu ratkaisu, mikä? _____

Muut & puuttuva tuki monelle tärkeälle osalle kuten testaukselle

25. Puuttuuko oliomaailmasta mielestänne jokin oleellinen osa?

Mitä muita kuin edellä esiteltyjä oliotekniikan ongelmia tai esteitä olette kokeneet?

Appendix 4 - Data Collection Protocol of the Case Study

Data Collection Protocol

1. Overview of the case study project

When doing the case studies the researcher has to keep the aim of the study in mind. The aim is the following:

> The aim of this study is to investigate and gain some understanding into what *benefits and problems* there are with the object-oriented paradigm in software development.

2. Field procedures

The first step is to gain access to the companies. This issue has been presented in the section on the case studies in this study.

When conducting the interviews the investigator needs to remember that they are made on the premises of the interviewee. The investigator also needs to remember that the interviewee might refuse to cooperate fully in answering the questions.

A tape recorder will be used, but if the investigator feel that the interviewee is uncomfortable because of it, or refuses to cooperate if a tape recorder is used, then written notes will be made.

The investigator should of course be polite and helpful when conducting the interview, and remember to do everything possible in order for the interviewee to feel comfortable. The investigator is of course forbidden to use threatening language or show what might be considered as unprofessional behaviour.

3. Case study questions

The case study questions are the same as the research questions, but have been modified in order to suit the interview and the case study protocol. Note that all research questions are considered, and not just a selection, as is the case in the questionnaire and survey.

General questions

1. Please tell me, has your company been using object-orientation in information systems development?

2. If your company has not been using object-orientation in information systems development, then why do you think this is so?

Benefits – Management of Complexity

3. Has the object-oriented paradigm apparently been found useful in your company when developing large-scale and complex information systems?

Benefits – Productivity, faster development and reduced costs

4. Do you think that object-oriented information system development has been more productive than traditional information system development?

5. Do you think that object-oriented information system development has been faster and generated fewer lines of code than traditional information system development?

Benefits – Quality and usability

6. Do you consider the quality of the object-oriented systems in your company better than the quality of the traditional systems?

7. What about usability? Have the object-oriented information systems in your company been more usable than the information systems that have been developed with traditional software development methods and programming languages?

Benefits – Natural and better mapping to the problem domain

8. Have you experienced that there has been a better and more 'natural' communication between information systems developers and end users because of using the object-oriented paradigm?

9. Do you think that object-oriented analysis is more natural for users than traditional analysis?

Benefits – Maintenance

10. According to your experiences, has maintenance of object-oriented applications been easier or harder than maintenance of traditional functional applications?

Benefits – Software components

11. Have you been using readymade components in your company? If so, have they been considered beneficial for information system development in your company?

12. Have you developed software components of your own in your company? If you have, are you of the opinion that object-orientation has made the development of software components easier?

Benefits – End – User computing

13. Have your clients been using End-User Computing? If they have, according to your opinion, has object-orientation made End-User computing easier?

Benefits – One model

14. Has the object-oriented system development process been seen as a uniform 'one model' from the problem domain to code and maintenance in your company?

15. Have you found it a benefit that there are the same building artefacts in object-oriented analysis as in object-oriented design?

Benefits – Frequent tangible working results and reliability

16. Has object-oriented information system development given frequent tangible working results?

17. Have the object-oriented information systems in your company been more reliable than those that have been developed with traditional software development methods and programming languages?

Benefits – Suitability for embracing new technologies and sound academic basis

18. Have you experienced that object-orientation is a good tool for embracing new technologies like graphical user interfaces or client-server applications?

19. What is your opinion about the statement that the sound academic basis of object-orientation is a benefit?

Benefits – Reuse

20. Has your company used much reuse? What can you tell me about the benefits gained from reuse in the object-oriented paradigm?

21. What do you reuse in your company? Classes? Objects? Components? Analysis? Design? Class libraries purchased from vendors? Class libraries developed in-house?

22. Do information system developers in your company prefer to reuse rather than to build from scratch; or do the information system developers consider reuse so difficult that they rather build components from scratch?

23. Are finding suitable software components a hindrance for reuse in your company?

24. Do you think that the producers of reusable software components in your company usually consider the needs of future users of the components? Future users of the components mean both people and systems.

25. Has multiple inheritance been used in your company? If so, was it successful?

Benefits – Object-oriented analysis

26. Have you noticed that users can switch from the object-oriented paradigm (a way of thinking in an object-oriented way) to the functional paradigm (a way of thinking in a functional way) and back in a smooth way?

27. In your company have you used prototyping for finding requirements in object-oriented information systems development?

Benefits – Object-oriented design

28. Has the transition to object-oriented design from object-oriented analysis been easy or difficult in your company when carrying out object-oriented software development?

Benefits – Portability

29. Has portability of object-oriented systems been considered a benefit in your company?

Benefits - Other

30. Have you experienced in your company that the total independence of classes has produced advantages in system development, compared with the traditional solution of modules with common data?

31. What further benefits of the object-oriented paradigm, other than those presented above, have you experienced in information systems development in your company?

Problems - Complexity

32. Has the object-oriented paradigm been considered complex in your company?

Problems – The object-oriented paradigm is still immature

33. Do you consider object-orientation as immature?

34. Has your company experienced difficulties in finding object-oriented CASE tools, object-oriented databases, object-oriented system development tools or perhaps even objects to reuse?

Problems – No support for several important areas like testing

35. Have you experienced a lack of support for any concepts in the object-oriented world? If you have, what is lacking according to your opinion? Is it a lack of support for objects? Reliability? Better performance? Or is it a lack of support for resource utilisation or security capabilities?

36. Have you found testing object-oriented information systems, applications or systems being difficult? What testing problems have you experienced?

Problems – Difficulties in measuring object systems

37. Has a lack of metrics for measuring the object-oriented system been considered a problem in your company?

Problems – Training & lack of experience

38. Have you in your company been using a mentor in order to solve the problem with training of the software developers?

39. Has there in your company been a resistance to learning the object-oriented paradigm because there is such a huge shift between the traditional functional paradigm and the object-oriented paradigm?

40. Has it been difficult to find experienced object-oriented software developers and system analysts?

Problems – Efficiency

41. Have you experienced computer efficiency problems in your company's object-oriented software development projects?

42. If there (in your company) has been no suitable collection of objects to reuse, has this lack been defective for the efficiency of the object-oriented information systems development project in question?

Problems – Costs

43. If there has been a lack of artefacts to reuse, have the starting costs been enormous when starting a completely new object-oriented information system or application, due to the lack of artefacts to reuse?

Problems – Limited usability of components

44. Have you had problems finding components to reuse in your company?

45. Has there been a problem managing the different versions of a component?

Problems – Problems with reuse

46. Has there in your company been a problem with reuse for the reason that software developers do not want to reuse a component, because they claim that it does not work, or it is too troublesome to learn how the component works?

47. Has the hierarchy of classes been a hindrance for reuse in your company?

Problems – Problems with object-oriented analysis

48. Have there been any kinds of problems with analysis when object-oriented analysis has been used in your company? If there have been problems, what kind have they been?

49. According to you, has object-oriented analysis been a good choice, if the system that is to be developed has limited responsibilities or is a system with few classes (< 10) and objects?

50. Have you in your company experienced one or several of the following problems with object-oriented information systems development in the analysis phase?

    Identification of Problem-Domain Structures has been difficult. It might often be difficult to identify classifications in the problem domain that could be mapped to inheritance hierarchies.

    Dealing with Excessive Domain Objects has been difficult. Integrating the domain knowledge with the user's requirement specifications can yield a lot of objects. Only few of these objects may be relevant to the problem area.

    Problems with Early Decomposition. If subsystems are not identified before objects are identified problems might arise, because objects have to be placed into some subsystem when identified. If the subsystems are identified before object identification, the boundaries of the subsystems may not be optimal.

    Subsystem-Object Distinction has been difficult. In the analysis phase objects may act as subsystems if they are complicated. Subsystems can also be defined as objects if they can be structured in a hierarchy and reused.

    Problems with Commonality versus Partitioning. Because subsystems partition the system, classes that are members of the same hierarchy can be spread over several subsystems. Finding the appropriate inheritance hierarchies becomes difficult.

    Subsystems Identification Using Object Interactions has been problematic. Subsystems are often used for structuring interactions among objects; however, most object-oriented methods only have intuitive techniques for subsystem identification.

Problems – Problems with object-oriented design

51. Has your company found the transition from object-oriented analysis to object-oriented design easy or difficult?

52. If the transition from object-oriented analysis to object-oriented design has been difficult, then why has it been difficult according to you? Difficulties to connect concepts found in object-oriented analysis with concepts in object-oriented design? Problems on this issue in the chosen object-oriented software development method? Object-oriented analysis was poorly performed because the object-oriented analysis was difficult? Object-oriented analysis was poorly performed because the object-oriented analysis method was insufficient?

Problems – Lack of object-oriented databases and common interfaces

53. Has it been difficult to find appropriate object-oriented databases in your object-oriented software development projects?

54. If a relational database has been used in the object-oriented system development project, which approach for connecting the object-oriented system with the relational database have you been used? A solution of mapping a class to a table? A solution with factory classes? Has the Strix object persistence engine been used?

55. Do you think that the lack of a common interface for ad hoc queries has been considered a problem when using pure object-oriented databases in your company?

Problems - Other

56. Have you experienced difficulties in mixing classes developed in different object-oriented programming languages or classes produced by different vendors?

57. What further problems or obstacles of the object-oriented paradigm, other than those presented above, has your company experienced?

4. Guide for the case study report with analysis plan

The analysis of the case study has been discussed in the case study sub section of the research method and research design section of this study.

Appendix 5 - The Case Study Protocol in Finnish

Tapausselostusten keräysprotokolla

1. Tapausselostustutkimuksen yleiskatsaus

Tehdessään tapausselostustutkimusta tutkijan on pidettävä tutkimuksen tavoitetta muistissa. Tutkimuksen tavoite on seuraava:

Tämän työn tavoitteena on tutkia ja saada ymmärrystä yleisistä oliotekniikan *hyödyistä* ja *haitoista* sovelluskehitystyössä.

Tavoitteena on myös oliotekniikan hyötyjen ja haittojen eri näkökantojen esittäminen, jos sellaisia näkökantoja on löydetty kirjallisuustutkimuksessa tai tämän työn empiirisessä osassa. Eri näkökantojen esittäminen voidaan pitää esitellyiden hyötyjen ja haittojen vaatimattomana analyysina.

Toisin sanoin, tutkijan joka tekee tapausselostustutkimusta, tulisi yrittää saada jonkun verran ymmärrystä yleisistä oliotekniikan hyödyistä ja haitoista sovelluskehitystyössä. Tutkijan tulisi myös yrittää löytää oliotekniikan hyötyjen ja haittojen näkökantoja sovelluskehitystyössä. Esimerkkinä eri näkökannoista voitaisiin mainita mahdollisuus että vastaaja antaa uuttaa tietoa liittyen hyötyihin ja haittoihin.

2. Kenttämenettelyt

Kenttämenettelyiden ensimmäinen askel on miten saada kontakti yrityksiin. Tämä seikka on esitelty tämän työn tapausselostustutkimusluvussa.

Tehdessään haastattelut haastattelijan on muistettava että haastattelut tehdään haastatellun ehdoilla. Haastattelijan on myös muistettava että haastateltu voi kieltäytyä täydestä yhteistyöstä kun hän vastaa kysymyksiin.

Haastattelijalla on aikomus käyttää nauhuria, mutta jos haastattelija huomaa että haastatellulla on epämiellyttävä olo, tai jos haastateltava kieltäytyy yhteistyöstä (mikäli nauhuria käytetään), niin haastattelija tekee tavallisia muistiinpanoja.

Haastattelijan tulee tietysti olla ystävällinen ja auttavainen tehdessään haastattelun. Haastattelijan tulee myös muistaa tehdä kaikkensa jotta haastateltavalla olisi mukava olo. Haastattelija ei tietenkään saa käyttää uhkaavaa kieltä tai asennetta.

3. Tapausselostustutkimuksen kysymykset

Tapausselostustutkimuksen kysymykset ovat samat kuin tutkimuskysymykset. Tapausselostustutkimuksen kysymykset on kuitenkin muokattu niin että ne olisivat haastatteluun ja tapauskysymysprotokollaan sopivia. Huomatkaa että kaikki tutkimuskysymykset ovat mukana. Lomakekyselyssä ja kyselylomakkeessa oli mukana vain valittu määrä tutkimuskysymyksiä.

Yleisiä kysymyksiä

1. Pyydän teitä ystävällisesti kertomaan onko yrityksenne käyttänyt oliotekniikkaa sovelluskehitystyössä?

2. Jos yrityksenne ei ole käyttänyt oliotekniikkaa sovelluskehitystyössä, niin miksi teidän mielestänne näin on?

Hyödyt - Monimutkaisuuden hallinta

3. Oletteko kokeneet oliotekniikan hyödylliseksi, kun olette kehittänyt laajoja ja monimutkaisia tietojärjestelmiä?

Hyödyt – Tuottavuus, nopeampi kehitystyö ja alhaisemmat kustannukset

4. Oletteko kokeneet, että oliopohjainen sovelluskehitystyö on ollut tuottavampaa kuin perinteinen sovelluskehitystyö?

5. Oletteko kokeneet, että oliopohjainen sovelluskehitystyö on ollut nopeampaa ja tuottanut vähemmän koodirivejä kuin perinteinen sovelluskehitystyö?

Hyödyt - Laatu ja käyttökelpoisuus

6. Oletteko kokeneet, että oliopohjaiset järjestelmät olisivat parempilaatuisia kuin perinteiset järjestelmät?

7. Miten on käyttökelpoisuuden kanssa? Ovatko teidän oliotekniikkaan perustavat järjestelmänne olleet käyttökelpoisempia kuin järjestelmät jotka on kehitetty tavanomaisilla sovelluskehitysmenetelmillä ja ohjelmointikielillä?

Hyödyt – Luonnollisuus ja parempi yhteensopivuus sovellusalueen kanssa

8. Onko oliomenetelmien käyttö sovelluskehitystyössä parantanut sovelluskehittäjien ja loppukäyttäjien välistä kommunikaatiota?

9. Pidättekö oliopohjaista määrittelyä käyttäjille luonnollisempana verrattuna tavanomaiseen määrittelyyn?

Hyödyt – Ylläpito

10. Onko teidän mielestä oliopohjaisten tietojärjestelmien ylläpito ollut helpompaa tai vaikeampaa kuin perinteisten funktionaalisten tietojärjestelmien ylläpito?

Hyödyt – Ohjelmistokomponentteja

11. Oletteko käyttäneet valmiita ohjelmistokomponentteja yrityksessänne? Jos olette käyttäneet valmiita ohjelmistokomponentteja, niin ovatko nämä olleet hyödyllisiä sovelluskehitystyössä yrityksessänne?

12. Oletteko kehittäneet omia ohjelmistokomponentteja yrityksessänne? Jos olette, niin oletteko sitä mieltä että oliotekniikka on helpottanut ohjelmistokomponenttien kehitystyön?

Hyödyt – End – User computing

13. Ovatko asiakkaanne käyttäneet End-User Computing menettelyä? Jos asiakkaanne ovat käyttäneet End-Using Computing:iä, onko mielestänne oliotekniikka tehnyt End-User computingin helpommaksi?

Hyödyt – Yksi malli

14. Oletteko pitäneet oliopohjaisen kehitysprosessin yhtenäisenä ”yhtenä mallina’ sovellusalueesta ohjelmointikoodaukseen ja ylläpitoon?

15. Oletteko kokeneet hyödyksi että oliomäärittelyssä on samat rakennusosat kuin oliosuunnittelussa?

Hyödyt – Nopeita ja näkyviä työtuloksia ja luotettavuus

16. Onko oliotekniikka tuottanut nopeita ja näkyviä työtuloksia?

17. Ovatko oliojärjestelmät yrityksessänne olleet luotettavampia kuin järjestelmät jotka on kehitetty tavanomaisilla sovelluskehitysmenetelmillä ja ohjelmointikielillä?

Hyödyt – Soveltavaisuus uusien teknologien omaksumiseen ja perusteltu akateeminen perusta

18. Oletteko kokeneet oliotekniikan soveltuvan uusien teknologien omaksumiseen? Uusilla teknologioilla tarkoitetaan teknologioita kuten graafiset käyttöliittymät ja client – server sovellukset.

19. Mitä mieltä olette väittämästä että oliotekniikan akateemista perustaa pidetään hyötynä?

Hyödyt – Uudelleenkäyttö

20. Oletteko käyttäneet paljon uudelleenkäyttöä (reuse)? Onko uudelleenkäyttö oliopohjaisessa sovelluskehitystyössä ollut hyödyksi?

21. Mitä käytätte uudelleenkäytössä? Olioita? Luokkia? Luokkakirjastoja joita olette ostaneet yrityksiltä? Luokkakirjastoja joita olette kehittäneet omassa yrityksessä? Määrittelyä? Suunnittelua? Komponentteja?

22. Käyttävätkö sovelluskehittäjät yrityksessänne mieluimmin uudelleenkäyttöä kuin rakentavat alusta alkaen? Pitävätkö sovelluskehittäjät uudelleenkäyttöä niin vaikeana että he mieluimmin rakentavat komponentteja alusta alkaen?

23. Onko oikeiden komponenttien löytäminen este uudelleenkäytölle yrityksessänne?

24. Oletteko sitä mieltä että uudelleenkäyttöön tarkoitettujen komponenttien tuottajat yrityksessänne ottavat huomioon komponenttien tulevaisuuden käyttäjien tarpeet? Tulevaisuuden käyttäjillä tarkoitetaan sekä ihmisiä että järjestelmiä.

25. Onko yrityksessänne käytetty moniperiytymistä? Jos moniperiytymistä on käytetty niin onko moniperiytyminen onnistunut hyvin?

Hyödyt – Oliomäärittely

26. Oletteko kokeneet että loppukäyttäjät pystyvät siirtymään oliomaailmasta (loppukäyttäjät ajattelevat oliotermein) funktionaaliseen maailmaan (loppukäyttäjät ajattelevat käyttäen funktioita) ja takaisin helposti ja ongelmitta?

27. Onko teidän yrityksessänne käytetty perusmuotoilua (prototyping) vaatimusten löytämiseen sovelluskehityksen oliomäärittelytyössä?

Hyödyt – Oliosuunnittelu

28. Onko siirtyminen oliomäärittelystä oliosuunnitteluun ollut vaikeaa tai helppoa yrityksessänne?

Hyödyt – Siirrettävyys (portability)

29. Oletteko kokeneet, että oliopohjaisten järjestelmien siirrettävyys on hyöty?

Hyödyt - Muut

30. Oletteko kokeneet yrityksessänne että luokkien täydellinen itsenäisyys on ollut hyödyksi sovelluskehitystyössä? Vertailukohteena on tavanomainen ratkaisu missä käytetään moduuleita joilla on yhtenäistä tietoa.

31. Mitä muita kun nyt esille tulleita oliotekniikan hyötyjä olette kokeneet sovelluskehitystyössä?

Ongelmia - Monimutkaisuus

32. Pidättekö olioteknikoita monimutkaisina?

Ongelmia – Oliotekniikka on vielä kehittymätöntä

33. Pidättekö oliotekniikkaa vielä kehittymättömänä?

34. Oletteko kokeneet, että on vaikeata löytää oliopohjaisia CASE-työkaluja, oliopohjaisia tietokantoja, oliopohjaisia sovelluskehitystyökaluja, tai jopa olioita jotka sopivat uudelleenkäyttöön?

Ongelmia – Puuttuva tuki monelle tärkeälle asialle kuten testaukselle

35. Monilta nykyisiltä oliojärjestelmiltä on vaikeata saada tietoja olioiden luotettavuudesta, oliojärjestelmien suorituskyvystä ja oliojärjestelmien resurssien käytöstä.

    Oletteko kokeneet puutteita jostakin oliomaailmassa? Mikäli te olette kokeneet puutteita, niin mistä? Puutteen olioista? Puutteen luotettavuuden tuesta? Puutteen paremmasta suorituskyvystä? Puutteen resurssien käytön tuesta? Puutteen turvallisuuden mahdollisuuden tuesta?

36. Oletteko kokeneet, että oliopohjaisia tietojärjestelmiä, oliopohjaisia sovelluksia tai oliopohjaisia järjestelmiä on vaikeata testata? Mitä testausongelmia olette kokeneet?

Ongelmia – Oliojärjestelmiä on vaikeita mitata

37. Pidättekö puutteena oliojärjestelmien mittaustavoissa olevia ongelmia?

Ongelmia – Koulutuksen ja kokemuksen puute

38. Oletteko yrityksessänne käyttäneet ohjaajaa tai neuvonantajaa (mentor) jotta pystyisitte ratkaisemaan sovelluskehittäjien kouluttamiseen liittyviä ongelmia?

39. Onko yrityksessänne ollut vastustusta oppia oliotekniikkaa koska oliotekniikan ja tavanomaisen sovelluskehitystekniikan ero on niin suuri?

40. Onko ollut vaikeata löytää kokeneita oliotekniikkaan perehtyneitä sovelluskehittäjiä ja suunnittelijoita?

Ongelmia – Tehokkuus

41. Oletteko kokeneet tehokkuusongelmia oliopohjaisissa projekteissanne?

42. Mikäli uudelleenkäyttöön sopivia olioita ei ole ollut, niin onko tämä puute ollut haitallinen kyseisen oliopohjaisen projektin tuottavuudelle?

Ongelmia – Kustannukset

43. Ovatko aloituskustannukset olleet suuret, kun olette aloittaneet täysin uuden oliopohjaisen tietojärjestelmän tai sovelluksen, koska uudelleenkäytettävistä osista on ollut puute?

Ongelmia – Rajoitettu komponenttien käytettävyys

44. Onko teillä ollut vaikeuksia löytää sopivia komponentteja uudelleenkäyttöön?

45. Onko ollut ongelmallista hallita komponenttien eri versioita?

Ongelmia – Ongelmia uudelleenkäytön kanssa

46. Oletteko kokeneet uudelleenkäyttöön liittyvän ongelmia koska sovelluskehittäjät eivät halua uudelleenkäyttää komponenttia, koska he väittävät että komponentti ei toimi, tai on vaikeata oppia miten komponentti toimii?

47. Onko luokkahierarkia ollut uudelleenkäytön este yrityksessänne?

Ongelmia – Oliomäärittelyn ongelmia

48. Oletteko kokeneet ongelmia kun olette käyttäneet oliomäärittelyä yrityksessänne? Mikäli olette kokeneet ongelmia, niin minkälaisia ongelmia olette kokeneet?

49. Onko oliomäärittely ollut mielestänne hyvä ratkaisu jos kehitettävällä järjestelmällä on vähän vastuuta tai jos järjestelmällä on pieni määrä luokkia ja olioita (< 10)?

50. Oletteko joskus kokeneet yhden tai useampia seuraavista ongelmista sovelluskehitystyössä määrittelyvaiheessa?

    Sovellusalueen struktuurit ovat vaikeita löytää. On usein vaikeata löytää sovellusalueen luokittelut joita voidaan yhdistää periytymishierarkioihin.

    Sovellusalueen ylimääräisiä olioita on vaikeita hallita. Sovellusalueen tietämyksen yhdistäminen loppukäyttäjien vaatimusmäärittelyiden kanssa voi tuottaa suuria määriä olioita. Yleensä vain pieni osa näistä olioista on sovellusalueelle kuuluvia.

    Ongelmia aikaisella hajottamisella. Mikäli olioita löydetään ennen alajärjestelmiä niin ongelmia voi syntyä koska oliot on sijoitettava alijärjestelmiin kun ne löydetään. Jos alajärjestelmiä taas löydetään ennen olioita niin alajärjestelmien rajat eivät välttämättä ole optimaaliset.

    Olioiden ja alajärjestelmien ero on epämääräinen. Määrittelyvaiheessa monimutkaiset oliot voivat toimia alajärjestelminä. Alajärjestelmät voivat taas toimia olioina jos alajärjestelmät voidaan strukturoida hierarkiaan ja uudelleenkäyttää.

    Ongelmia: yleistää vai ositella? Koska alajärjestelmät osittavat järjestelmää, niin luokat jotka ovat samassa hierarkiassa voidaan sijoittaa eri alajärjestelmiin. Oikean periytymishierarkian löytäminen vaikeutuu.

    Alajärjestelmien löytäminen käyttäen hyödyksi olioiden vuorovaikutusta on ollut vaikeata. Alajärjestelmiä käytetään usein olioiden vuorovaikutusten strukturointiin, mutta useimmilla oliomenetelmillä on ainoastaan vaistonvaraisia tekniikoita alajärjestelmien löytämiselle.

Ongelmia – Oliosuunnittelun ongelmia

51. Onko yrityksenne kokenut siirtymisen oliomäärittelystä oliosuunnitteluun helpoksi tai vaikeaksi?

52. Mikäli siirtyminen oliomäärittelystä oliosuunnitteluun on ollut vaikea, niin mikä on mielestänne syy tähän? Ongelmia yhdistää oliomäärittelyn konseptit oliosuunnittelun konseptiin? Ongelmia liittyen tähän kysymykseen sovelluskehityksen oliomenetelmässä? Oliomäärittely oli huonosti tehty koska oliomäärittely on vaikea? Oliomäärittely oli huonosti tehty koska oliomäärittelymenetelmä oli puutteellinen?

Ongelmia – Oliotietokantojen ja yleisten liittymien puute

53. Onko ollut vaikeata löytää sopivaa oliotietokantaa sovelluskehitystyöhön, edellyttäen että olette etsineet sellaista?

54. Kun olette käyttäneet relaatiotietokantaa oliopohjaisessa sovelluskehitystyössä, niin mitä ratkaisua olette käyttäneet, kun olette liittäneet oliojärjestelmän käyttämäänne relaatiotietokantaan? Oletteko käyttäneet ratkaisua missä luokka liitetään tauluun? Oletteko käyttäneet tehdasluokkia? Oletteko käyttäneet kuoria (wrapper)? Oletteko käyttäneet valmiita luokkia jotka olette ostaneet? Oletteko käyttäneet “The Strix object persistence engine”?

55. Oletteko sitä mieltä että yleisen liittymän puute (jolla voi tehdä tilapäisiä kyselyitä) on ongelma kun käytätte oliotietokantoja yrityksessänne?

Ongelmia - Muita

56. Oletteko kokeneet ongelmia kun olette yhdistäneet luokkia joita on kehitelty eri ohjelmistokielillä tai jotka eri valmistajat ovat kehittäneet?

57. Mitä muita kuin edellä esiteltyjä oliotekniikan ongelmia tai esteitä olette kokeneet?

4. Tapausselostustutkimuksen raportin ohje ja analyysin suunnittelu

Tapausselostustutkimuksen analyysi on käsitelty aikaisemmin tässä työssä.

Appendix 6 - Some secondary definitions for this study

Application domain. The organisation that administers, monitors, or controls a problem domain (Mathiassen et al., 2000, p. 3). The application domain is a part of the user organisation (Mathiassen et al., 2000, p. 6). For example, in a payroll information system the application domain includes the personnel office, while the problem domain includes the employees, the contracts and the salary information, etc.

Business objects. OMG (Object management group) gives the following (quotation) definition (cited in Räisänen, 1997b, p. 32):

> A business object is a representation of a thing active in the business domain, including at least its business name and definition, attributes, behavior, relationships and constraints. A business object may represent, for example, a person, place or concept. The representation may be in a natural language, a modeling language, or a programming language.

One can of course develop business objects of different complexity, from simple plain objects like phone calls to more complicated objects like companies.

Component. A component is a unit of software (or something else, see below) designed to integrate and work with other units of software (Webster, 1995, p. 25).

According to Nierstrasz & Dami (1995, p. 4) a component is a component because it has been designed to be used in a compositional way together with other components. A component is usually developed in isolation, but a component can be one part of a framework of collaborating components (Nierstrasz & Dami, 1995, p. 4). Nevertheless, more formally, Nierstrasz & Dami (1995, p. 5) give the following definition of a component (quotation):

> A component is a static abstraction with plugs

Radin (1996) and Sparling (2000) give another definition of a component; they propose that components can be seen as encapsulated black boxes with specified behaviour. Examples of components are classes, objects, functions, macros, procedures, templates, modules, specifications, documentations, test data and applications. Examples that are more specific are buttons and lists (Räisänen, 1997b, p. 28).

The concepts of components and objects should not be considered the same, according to Petre (2000, p. 6); the difference is that objects are suitable for describing real world entities, and components are suitable for describing the *services* of real world entities. Expressed differently, objects are suitable for describing the problem domain of a system, and components are suitable for describing the *functionality* of the problem domain (Petre, 2000, p. 6).

The First International Workshop on Component Oriented Programming (WCOP '96), which was a part of the ECOOP96, produced the following definition of a *software component* (Helton, 1998; Eliëns, 2000, pp. 178-179):

> A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties.

Interesting is thus that a component's specification is required, but the implementation of the component is not required. An 'interface' conceals the particular implementation details, and the interface provides a group of service specifications for the component. (Helton, 1998) Note that in this study, the object-oriented paradigm is the main area of research, but component software is also considered in some cases because component software and the object-oriented paradigm are considered the same by many researchers. One has, however, to remember that some researchers like Szyperski (1999) consider component software as a unique paradigm that should not be mixed with the object-oriented paradigm, though he points out that there is a comprehensible connection because components are likely to come to life through objects (Szyperski, 1999, p. 31).

Component-based software engineering. In component-based software engineering, applications are built out of existing components that are reused (Pressman, 2000, p. 738). The following presentation of component-based software engineering (quotation) is given by Clements (1995; cited by Pressman, 2000, p. 738):

> Component-based software engineering is changing the way large software systems are developed. Component-based software engineering embodies the "buy, don't build" philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, component-based software engineering shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus. At its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.

Component-based software engineering is not presented or discussed in this study more thoroughly because it is not a part of the purpose of this study. See also definition of component.

Composition. A composition is a fact when an object is built out of other objects. A composition defines the has_a relationship (Webster, 1995, p. 24). Madsen (1995) says that with composition is an object that can contain components that are part objects or references to other objects. For example, composition is supported in object-oriented languages so that concerning part objects a class attribute can be an instance of another class in C++ (Madsen, 1995).

CORBA. Konstantas (1995, p. 72) says that CORBA stands for Common Object Request Broker Architecture and has its origin at Object Management Group (OMG). The Object Management Group is a consortium of over two hundred technology vendors and users that can be characterised as technology sponsors for the coordination of standards, subsidization of early adopters and promotion of the object-oriented paradigm (Fichman & Kemerer, 1993). Helton (1998) says that Object Management Group is a consortium with over 700 companies in 1998, and that it is usually used for plugging together components in component oriented programming (other standards for plugging together components are Microsoft's Component Object Model, COM and Sun Microsystems' JavaBeans).

The Object Request Broker (ORB) provides interoperability between applications or information systems on different computers in distributed environments. ORB is a common layer through which objects transparently exchange messages and receive replies. The Interface Definition Language (IDL) describes both the interfaces that the client objects requests and the interfaces that the object implementations provide. (Konstantas, 1995, p. 72) According to Watson (1999) CORBA is an API, which can be used, for example, for binding legacy systems with new applications.

Design patterns. Design pattern and a pattern usually mean the same thing, though there are many different types of patterns in the object-oriented world. Gamma et al. (1995) are probably the most widely known persons when talking about design patterns. In the book by Gamma et al. (1995) the famous statement by Christopher Alexander (often regarded as the father of design patterns) can be found, and it goes like this (quotation):

> Each pattern describes a problem, which occurs over and over again in our environment, and describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same time twice.

In the article by Eden (2002) another definition (quotation) is given (Alexander et al., 1977, Alexander, 1979):

> A "pattern" is a prescription for solving a category of problems in a specific manner. This prescription is intended for the dissemination of specialized knowledge and to create an instrumental vocabulary.

Though Alexander was talking about design patterns connected to building houses and towns, what the writer says is true about object-oriented design patterns (Gamma et al., 1995, p. 2). In the book by Gamma et al. (1995, p. 3) the design patterns are (quotation):

> Descriptions of communicating objects and classes that are customized to solve general design problems in a particular context.

Design pattern or object design pattern is a predefined design structure, which is used as a building block when building a software architecture (Tepfenhart, & Cusick, 1997). According to Pressman (2000, p. 607) design patterns may become the software analogue of small circuits made from components (classes would be seen as components). Often design patterns actually consist of classes, or objects, or the communication between objects or the contracts between objects, etc. (Räisänen, 1997b, p. 27).

Monroe et al. (1997) state that the basic idea behind design patterns is that common idioms are found several times in object-oriented software design and that these patterns should be made explicit, codified, and applied properly to similar problems or tasks. Gamma et al. (1995, p. xi) state that design patterns capture solutions that are considered good and that have been developed and evolved over time. According to Tepfenhart, & Cusick (1997) there is a difference between design patterns and object design patterns, because object design patterns can be architectural frameworks, design patterns and idioms.

There is often confusion between design patterns and frameworks. Gamma et al. (1995, p. 28) explain the differences between design patterns and frameworks (quotation):

> 1. *Design patterns are more abstract than frameworks*. Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, the design patterns in this book have to be implemented each time they're used. Design patterns also explain the intent, trade-offs and consequences of a design.
> 2. *Design patterns are smaller architectural elements than frameworks*. A typical framework contains several design patterns, but the reverse is never true.
> 3. *Design patterns are less specialized than frameworks*. Frameworks always have a particular application domain. A graphical editor framework might be used in a factory simulation, but it won't be mistaken for a simulation framework. In contrast, the design patterns in this catalog can be used in nearly any kind of application. While more specialized design patterns than ours are certainly possible (say, design patterns for distributed systems or concurrent programming), even these wouldn't dictate an application architecture like a framework would.

However, Pressman (2000, p. 607) say that there are still surprisingly few patterns catalogued on the market, though this is an area where progress is made incessantly.

Domain. A domain is a given area of functionality or certain problem area (Webster, 1995, p. 25).

Frakes & Isoda (1994) offer the following definition; a domain is an application area, or more formally, a set of systems that share design decisions.

Martin & Odell (1995, p. 20) define a domain as a selected area of interest that consists a set of objects that are instances of the domain specification. The domain specification is then the collection of concepts that apply to a domain.

Shlaer & Mellor (1992, p. 133) offer the following definition of a domain; a domain is defined as a separate world inhabited by a distinct set of objects that behave according to rules and characteristics of the domain

Framework. A framework is a set of classes (as a rule the classes are abstract classes) that collaborate to carry out a set of common responsibilities (Gamma et al, 1995, p. 26; Taivalsaari, 1993, p. 159). There is usually some kind of interconnection between the classes and to the classes there are often components or subclasses connected (Johnson, 1997b; Räisänen, 1997b, p. 30). The code of the framework is in the framework, and the framework calls the code outside the framework in the Hollywood style 'Don't call us, we'll call you' (Räisänen, 1997b, p. 30).

Webster (1995, p. 25) almost offers the same definition when he writes: "a framework is a collection of related objects that work together to provide a certain class of functionality".

According to Pree (1997), “a framework is a collection of several single components with predefined co-operations between them, for the purpose of accomplishing a certain task”. Here one can also note that Pree (1997) proposes that frameworks require enormous development effort; in fact, Pree (1997) proposes that the costs of developing a framework are significantly higher compared to the costs of developing a similar application.

Frameworks make up, for example, a reusable design for an exclusive software class that provides the entire domain-independent infrastructure you need to implement a system (Tepfenhart, & Cusick, 1997). Frameworks can consist of design patterns, and mature frameworks often incorporate several design patterns (Gamma et al., 1995, p. 27). The opposite is, however, never the case; a design pattern never consists of frameworks (Gamma et al., 1995, p. 28).

Another definition of a framework is that it is an application specific class library (Winblad et al., 1990, p. 34); in other words, a grouping of classes that is developed for a specific application, but still is so general that it can be widely reused (Henderson-Sellers, 1992, p. 59). The frameworks can be application frameworks or company frameworks, etc. (Fayad, 2000).

A further definition of a framework states that a framework is a reusable object-oriented analysis and design for an application or subsystem; an application framework thus provides a template for an entire application (Coleman et al., 1994, p. 219). However, Coleman et al. (1994, p. 219) go on and argue that a framework is connected to its problem and solution space classes, which include abstract classes. Therefore, the definition by Coleman et al. (1994, p. 219) supports the definition by Taivalsaari (1993, p. 159). In an article Liao et al. (1999) writes about a generic application for reuse when developing new applications, this generic application can probably be classified as a kind of framework.

Problem domain. The part of a context that is administered, monitored or controlled by a system (Mathiassen et al., 2000, p. 3). The problem domain describes the system’s purpose, as well as the parts of reality that the system should help administer, monitor or control (Mathiassen et al., p. 6).

Repository. According to Jenz (1999a) a repository is almost the same as a library. Martin & Odell (1992, p. 7) give a more common definition of a repository: “The repository is a mechanism for defining, storing and managing information about an enterprise, its data and systems”. Martin & Odell (1992, p. 7) goes on and propose that a repository stores models, specifications, designs and reusable constructs that are used in software development. This gives the ultimate definition that a repository is an object-oriented database, storing information about objects, storing information about libraries of reusable classes and facilitating reusable design (Martin & Odell, 1992, p. 48).

Subject. Subjects are important when performing object-oriented software development. Subjects are something that act upon objects like users, systems and other objects (Webster, 1995, p. 110).

Subsystem. A subsystem is a general term for a set of objects that interact with each other (Webster, 1995, p. 25). A subsystem is defined as a set of classes (and possibly other subsystems) working together to fulfil a set of responsibilities (Wirfs-Brock et al., 1990, p. 30).

Wrappers. Wrappers are interesting concepts; if one wants to connect a traditional functional application with an object-oriented application, this can often be done using object-oriented wrappers (Martin & Odell, 1992, p. 35). Because there are a lot of traditional functional applications and traditional functional information systems in society today one often has to use wrappers. Wrapping is a technique where traditional functional software gets a layer of object-oriented code around it, and then the traditional functional software appears to be object-oriented (Ambler, 1998, p. 343). A wrapper is a collection of one or more classes that encapsulates access to technology that is not object-oriented. The following wrapping techniques are presented by Ambler (1998, pp. 348-357); C APIs, dynamic shared libraries, screen scraping, peer-to-peer and CORBA (CORBA is actually predominantly used for other purposes).

EKONOMI OCH SAMHÄLLE

Skrifter utgivna vid Svenska handelshögskolan

Publications of the Swedish School of Economics
and Business Administration

123. DAVID BALLANTYNE: A Relationship Mediated Theory of Internal Marketing. Helsingfors 2004.

124. KRISTINA HEINONEN: Time and Location as Customer Perceived Value Drivers. Helsingfors 2004.

125. CHRISTINA NORDMAN: Understanding Customer Loyalty and Disloyalty – The Effect of Loyalty-Supporting and -Repressing Factors. Helsingfors 2004.

126. MATTS ROSENBERG: Essays on Stock Option Compensation and the Role of Incentives and Risk. Helsingfors 2004.

127. BENJAMIN MAURY: Essays on the Costs and Benefits of Large Shareholders in Corporate Governance. Helsingfors 2004.

128. HONGZHU LI: Conditional Moments in Asset Pricing. Helsingfors 2004.

129. MIREL LEINO: Value Creation in Professional Service Processes. Propositions for Understanding Financial Value from a Customer Perspective. Helsingfors 2004.

130. MARKO S. MAUKONEN: Three Essays on the Volatility of Finnish Stock Returns. Helsingfors 2004.

131. JAN ANTELL: Essays on the Linkages between Financial Markets, and Risk Asymmetries. Helsingfors 2004.

132. HELENA ÅKERLUND: Fading Customer Relationships. Helsingfors 2004.

133. KIM SKÅTAR: Faktorer som initierar och påverkar prat i långsiktiga relationer. Factors that Initiate and Influence Word of Mouth in Long-Term Relationships. English Summary. Helsingfors 2004.

134. ROBERT WENDELIN: The Nature and Change of Bonds in Industrial Business Relationships. Helsingfors 2004.

135. LI LI: Knowledge Transfer within Western Multinationals' Subsidiary Units in China and Finland. The Impact of Headquarter Control Mechanisms, Subsidiary Location and Social Capital. Helsingfors 2004.

136. MINNA HIILLOS: Personnel Managers and Crisis Situations – Emotion-Handling Strategies. Helsingfors 2004.

137. ÅKE FINNE: Hur den aktiva kunden konstruerar budskap. Ett synsätt inom relationskommunikation. How the Active Consumer Constructs Messages. A Relationship Communication Perspective. English Summary. Helsingfors 2004.

138. MARTIN SEPPÄLÄ: A Model for Creating Strategic Alliances. A Study of Inter-Firm Cooperation in the North European ICT Sector. Helsingfors 2004.

139. HANNELE KAUPPINEN: Colours as Non-Verbal Signs on Packages. Helsingfors 2004.

140. TEEMU KOKKO: Offering Development in the Restaurant Sector - A Comparison between Customer Perceptions and Management Beliefs. Helsingfors 2005.

141. BERNARD BEN SITA: Essays on the Role of Time in Price Discovery. Helsingfors 2005.

142. MARTIN FOUGÈRE: Sensemaking in the Third Space – Essays on French-Finnish Bicultural Experiences in Organizations and Their Narratives. Helsingfors 2005.

143. PERNILLA GRIPENBERG: ICT and the Shaping of Society: Exploring Human – ICT Relationships in Everyday Life. Helsingfors 2005.

144. TUA HALDIN-HERRGÅRD: Hur höra tyst kunskap? Utveckling av en metod för studier av tyst kunnande. Helsingfors 2005.

145. SARI SALOJÄRVI: Increasing Knowledge Focus – a Means for Entrepreneurs to Remain on a Growth Path. Essays on the Role and Nature of Knowledge Management in Finnish SMEs. Helsingfors 2005.

146. MARJUT JYRKINEN: The Organisation of Policy Meets the Commercialisation of Sex. Global Linkages, Policies, Technologies. Helsingfors 2005.

147. HENRI MÄNTTÄRI: Short-Term Price Behavior and the Effect of Foreign Investors in Finnish Equity Markets. Helsingfors 2005.

148. MIA ÖRNDAHL: Stories of Survival. Knowledge Intensive Organisations and the Finnish 1990s Recession. Helsingfors 2005.

149. OLGA KARAKOZOVA: Modelling and Forecasting Property Rents and Returns. Helsingfors 2005.

150. TOMI HUSSI: Essays on Managing Knowledge and Work Related Wellbeing. Helsingfors 2005.