Faculdade de Engenharia da Universidade do Porto

WARP - Speeding Up The Software Development Process

José Filipe Barbosa de Carvalho

Report of Project
Master in Informatics and Computing Engineering

Supervisor: António Carvalho Brito (Assistant Professor)

July 2008

WARP - Speeding Up The Software Development Process

José Filipe Barbosa de Carvalho

Report of Project

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (Associate Professor)

External Examiner: César Analide (Assistant Professor)

Internal Examiner: António Carvalho Brito (Assistant Professor)

July 17th, 2008

Contact Information:

José Filipe Barbosa de Carvalho
Mestrado Integrado em Engenharia Informática e Computação

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n
4200-465 Porto
PORTUGAL

Tel: +351 916498040
Email: jose.carvalho@fe.up.pt
URL: http://www.fe.up.pt/~ei03067/

to my mother Maria da Conceição
and to my father José Armindo

to my girlfriend Teresa

Abstract

Nowadays business operations happen very fast and they are really complex, so almost every organization supports its activities using information systems and other technological instruments. Because sometimes there are no final products that fit their needs, it is common that companies are enrolled in software development activities. The complexity and abstract character of software implies new challenges for these enterprises.

The success of software development process is determined by several factors, for example the value created by the product, technical difficulties, team skills or communication between stakeholders. If the organization adopts an organized process, choosing the correct methodologies and tools to support it, we can speed up it and, at the same time improve its quality.

This project is all about improving the software development process, looking for three activities that improve communication: business modeling, object-relational mapping and prototyping user interfaces.

Business modeling is a complex activity, involving the knowledge of problem domain and some technological background. However their common outputs, texts and visual diagrams, are usually informal and ambiguous. WARP solution proposes developing formal and executable business models, using an object-oriented programming language. These models are testable and can be reused in following development stages.

Most of business entities have a long life, requiring that their knowledge and data must be safely saved. Object-relational mapping software helps the developer in the implementation of persistency concerns, decreasing the effort to connect relational schemas and objects.

Prototyping user interfaces provides earlier discussion with clients about concrete things, which would detect fails on requirements or on system design. Moreover, nowadays there are methods for automatic generation of user interfaces from models, which increase significantly the developer's productivity.

The WARP project studies current techniques and tools associated with these three activities, choosing an approach to be integrated in current software development processes. Finally this approach is applied in the development of Equipment Database, a product that solves a Qimonda's specific problem, to evaluate its applicability in real projects.

Resumo

Nos dias de hoje os negócios acontecem muito rapidamente e são cada vez mais complexos. Para suportar as suas actividades a maioria das organizações utilizam sistemas de informação e outros instrumentos tecnológicos. Porque muitas vezes não existem produtos à medida das suas necessidades, estas vêem-se envolvidas em actividades de desenvolvimento de software. A complexidade e o carácter abstracto do software implicam novos desafios para estas empresas.

O sucesso do processo de desenvolvimento de software é determinado por diversos factores, como por exemplo o valor criado pelo produto, as dificuldades técnicas, as capacidades da equipa ou a comunicação entre os diferentes interessados. Se a organização adoptar um processo organizado, escolhendo as metodologias e ferramentas certas para o suportar, é possível acelerar e ao mesmo tempo melhorar a sua qualidade.

Este projecto tem como objectivo melhorar o processo de desenvolvimento, explorando três actividades que melhoram a comunicação: modelação de negócio, mapeamento objecto-relacional e prototipagem de interfaces de utilizador.

Modelar o negócio é uma actividade complexa, envolvendo a compreensão do domínio do problema mas também de eventuais restrições tecnológicas. No entanto, os artefactos produzidos nesta actividade, descrições textuais e diagramas, são informais e ambíguos. A solução WARP propõe o desenvolvimento de modelos de negócio formais e executáveis, usando uma linguagem orientada a objectos. Desta forma, os modelos são verificados através de ferramentas de teste, bem como reutilizados nas fases seguintes do processo de desenvolvimento.

A maioria das entidades de negócio tem um tempo de vida longo, obrigando as aplicações a guardar o seu conhecimento e a sua informação. As ferramentas de mapeamento objecto-relacional ajudam o programador na implementação da persistência, diminuindo o esforço para relacionar os objectos com as estruturas das bases de dados.

A prototipagem de interfaces de utilizador permite realizar mais cedo discussões sobre o produto com os clientes, ajudando a encontrar falhas nos requisitos ou no desenho da aplicação. Além disso, nos dias de hoje é possível gerar automaticamente interfaces de utilizador a partir de modelos, aumentando a produtividade de quem desenvolve.

O projecto WARP estuda técnicas e ferramentas actuais relacionadas com estas três actividades, definindo uma abordagem a ser integrada nos processos actuais de desenvolvimento de software. Por fim, esta abordagem é utilizada na criação da *Equipment Database*, um produto para resolver um problema específico da Qimonda, para medir a sua aplicabilidade em projectos reais.

Preface

During the elaboration of this report, the enthusiasm and expectations were enormous for two reasons: I am about to finish my master course and I am contributing with my expertise to solve an enterprise problem.

The former has been a life objective, which I pursued many years ago. I strongly believe that my course contributed to enrich me, personally and intellectually. But it was a long endeavor. Five years were elapsed from the day I started my course and I wrote this document. There are many memorable stories that can be told: nightly work, night parties, interesting courses, boring lessons, and so on. At the end, two important things remain: friends and knowledge.

I have been working to improve the software development process during the last four months. By improving I mean speeding up the process with more quality. My contribution is modest, but I would expect to change your mind about some topics about software engineering with my visions and recommendations. I will focus more on practice, giving examples and recommending, rather than other more theoretical works.

This work was challenging and very motivating. The impact of my research can be large, improving the productivity of software industry, or even inspiring others with my ideas. Additionally a large technical expertise was acquired during this effort.

My ideas are sympathetic with lean engineering, agile and test-driven development, and they are quite influenced by Open Source ideals. That does not mean it cannot be implemented, completely or partly, in business environments or with proprietary tools. As you will see in this project, they can be integrated in a company and within a Microsoft-based development environment.

However, I strictly agree with the principles of the Context-Driven School [1], which emphasizes the importance of the context. This school starts in software testing domain but I believe that applies to other fields of software development. This means that any presented good practice or interesting tool could not be the best in other contexts. Moreover corrections and comments are welcome; please use the contact information to get in touch.

I hope this document will be an enjoyable reading for you, providing new trends and concepts about software development.

Vila do Conde, July 2008 *José Carvalho*

Acknowledgments

I would like to express my gratitude for all who contributed, directly or indirectly, during this project. I would like to thank my family, friends and colleagues for those expertise, understanding and patience, adding value and providing guidance into my work. In particular I would like to thank my parents, Maria and José, and my girlfriend, Teresa, who gave me their personal guidance and love.

I would acknowledge Professor António Carvalho Brito of *Faculdade de Engenharia da Universidade do Porto*, for his coordination, suggestions and revisions. I would also like to thank other college professors that contributed with their suggestions: Ana Paiva and Ademar Aguiar.

I would like to thank my English professor, Célia Arezes, for her dedication on reviewing the syntax and semantics consistency of this report.

I have a special acknowledge word to João Cortez. He envisioned the project and provided me the appropriate orientation during the entire project. His contributions strongly influenced this project, from his references and books, to meeting discussions and his document revisions.

Finally, I would like to thank *Qimonda Portugal, S.A.*, which provides me knowledge, infrastructure and financial support to complete this journey.

“Programming languages are very good
in expressing what the computer should do;
they are less effective in expressing
the developer intent when the code was written.”

Ayende Rahien [2]

Contents

List of Figures

Part I

Overview

Chapter 1

Introduction

This project aims the improvement of the software development process: create faster a better product. This objective has been pursued constantly by all industries. Information technology sector is not an exception. In the last years, many strategies emerged to accomplish the common goals: increase developers' productivity, reduce the costs, focus on the customer, raise innovation, and so on. This has lead to the creation of new methodologies, tools and standards since the first computers were created, namely from software engineering, software quality and management disciplines.

All these solutions, however, do not stop the research in this field. First, the problem was not solved. Software projects are famous by its high failure rates. A study of Standish Group in 1996 revealed that 31,1% of the information system projects are canceled before finishing and more than 50% cost almost double of the estimations [3]. Although a similar study in 2000 reveals enhancements [4], anyone enrolled in software development knows that there is a lot to do. Second, it is a natural look for solutions that produce more value and makes life of collaborators easier.

WARP drive is the faster-than-light movement in science fiction. Therefore WARP project intends speeding up the software process, providing a general understanding about software development process, and then showing techniques that can improve it. These techniques concentrate on business modeling, object-relational mapping and user interface prototyping.

This report is targeted for anyone involved in software development: project managers, architects, developers and even clients. However a strong technical background would be required to understand some parts, namely related with design and programming activities.

1.1 Project context

The work was developed during the Project course, performed in an entrepreneurial context, in the company *Qimonda Portugal, S.A*, to improve specific aspects of its internal software development, looking for new trends to integrate on future projects.

This course is the final unit of studies of *Mestrado Integrado em Engenharia Informática e Computação*, at Faculdade de Engenharia da Universidade do Porto

1.2 Methodology

The methodology adopted to achieve intended results follows the scientific method [5]. It is an organized way of thinking and appears obvious for a research and development project. In fact, lean development philosophies argue that is a good way to solve problems within business world, as Toyota has been doing with its Toyota Production System problem-solving method [6].

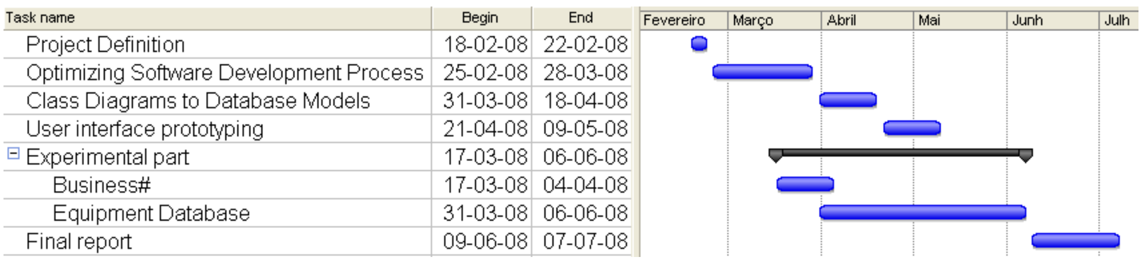
The scientific method adds more formality and organization to the process, and it has the following steps:

1. Define the problem;
2. Observe, collecting data to understand better the problem;
3. Formulate a hypothesis to explain and, possibly, solve the problem;
4. Perform experiments and collect results;
5. Draw and publish conclusions.

The work went along through these phases, as it is explicit in its document structure 1.4. First, problems to be addressed were identified and prioritized. Then an analysis moment succeeded, by studying available techniques and tools. The formulation of hypothesis was accompanied by reasoning, discussion and small experiments. After that, two major experiments were done: Business# and Equipment Database. Business# is a framework to develop evolutionary business models. Equipment Database is an application which integrates information about manufacturing equipments. It has been developed for research purposes, to discuss and test Business#. This report outlines the conclusions during the research.

1.3 Project management

The project schedule was tightly related with the methodology. The project occurred during four months, following an estimated schedule presented in 1.1. The first week was focused in a specific company formation and in the project definition. Following five weeks looking for improvements in software development processes, mainly centered in business modeling. Then three weeks were dedicated to mapping classes and relational schemas and more three about generating automatically user interfaces from classes. During that time and in the following four weeks, were built two implementations: Business# and Equipment Database. Finally four weeks were dedicated to write the final report.



Task name	Begin	End
Project Definition	18-02-08	22-02-08
Optimizing Software Development Process	25-02-08	28-03-08
Class Diagrams to Database Models	31-03-08	18-04-08
User interface prototyping	21-04-08	09-05-08
⊟ Experimental part	17-03-08	06-06-08
Business#	17-03-08	04-04-08
Equipment Database	31-03-08	06-06-08
Final report	09-06-08	07-07-08

Figure 1.1: Project plan

Weekly meetings occurred to verify the progress and discuss ideas. Similarly, every week, a report was written with the accomplishments, disappointments, critical items and next steps. And meetings with the college coordinator took place to supervise the performed work.

1.4 Document structure

The structure of the report follows a traditional research document, starting by an introduction, which contextualizes the work. Then it is presented the state of the art in software development and the carried out experiments in the project. Finally the achieved conclusions are presented.

This document is organized in four parts with several chapters in each part:

Part 1 introduces the project and which problems are addressed. It also presents the context, methodology used, work plan and controlling activities, and document structure.

Part 2 provides a general understanding about software development process, and explains solutions to improve it. There is a focus on business modeling, object-relational mapping and user interface prototyping.

Part 3 details performed experiments to prove the applicability of previous described solutions.

Part 4 presents a list of lessons learned about software development process and conclusions.

In the end of the report, there is a glossary, references and appendices.

Part II

WARP approach

Chapter 2

The problem

Motto: Create Faster, Create Better

The goal of this project is to speed up the software development process. Speed means shorter time from vision to market, with similar resources and without sacrificing quality. Therefore a better process is needed, to eliminate waste and discipline the development environment. In the same way, every process must fit to specific business constraints and involved people.

However finding a good process is a hard mission. First because software development is an intensive knowledge activity, where it is hard to measure productivity [7] . Projects barely follow estimations, and metrics hardly represent the real value produced. The artifacts are abstract entities instead of concrete items, like raw materials and final outputs on regular industries, which tangle the problem. Measuring a report in number of pages or an application in number of lines of code appears to be too naive. These types of metrics always have a relative value. Similarly, an estimation based on product potential value could be performed, giving weight percentages to activities or produced artifacts. But defining these weights is also relative and is not free of error.

Second software development occurs on varied contexts, from companies to universities, with different objectives, constraints and people. Although business objectives are usually centered on generating revenues and reducing costs, others would be valid, like personal or organization improvement or help humanitarian causes. Moreover context constraints influence decisions and development, as labor regulations or organizational hierarchies.

The multiplicity of stakeholders enrolled to build software products is huge: clients, final users, managers, analysts, architects, developers, testers, system administrators, quality people, technical writers, sales people, researchers, students, and so on. Furthermore, most of the time it could be more complicated because:

- The objectives and preferences are not the same for everyone;

- People have different scientific background and professional experience;
- They work in different places and companies;
- They speak using specific vocabulary, or even different languages.

Problems of communication and different interests are common, and many times the management contributes more to increase than to mitigate them. Discussions around specifications or between people that share responsibilities can arise easily.

Communication impedance mismatches is a term to refer barriers and noises introduced during information transfers, derived from electrical impedance mismatch concept. The mismatches generate misunderstandings between involved elements [8].

The process of developing software has three important communication mismatches, which are explicit in figure 2.1. Enrolled elements are very different and speak different languages. The analyst must listen to the customer and capture the requirements. But the client knows much more about problem domain, and the analyst knows much more about computers. Then the analyst transmits to the developer the requirements. But he only gives parts of his own vision of customer objectives. Finally, the developer must convert desired functionalities into code. However *programming languages are very good in the expressing what the computer should do; they are less effective in expressing the developer intent when the code was written* [2].

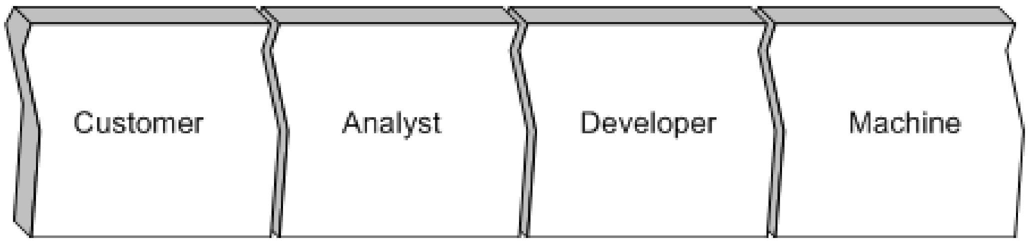


Figure 2.1: Communication mismatches on software development

Bringing and implementing a disciplined process in these complex contexts is challenging. Communication and synchronization difficulties happen every day, interfering on the battle for speeding up processes.

2.1 Objectives

The motto is *Create faster, Create better*. Faster means a shorter time to develop a system. Better means a disciplined way to create a product with higher quality. Better also means simple, lean and more automatic, and thus quicker. Therefore the main objective is to study techniques and tools, which could be incorporated on and which improve current software development process, instead of creating a totally new methodology. Namely, it advocates the importance of following activities:

- Connecting development phases and its results;
- Business modeling;
- Object-relational mapping;
- Prototyping the user interface.

The idea is to create a business model that guides implementation. The business model should be formal and executable, avoiding the subjectivity inherent to textual descriptions or visual diagrams. Informal descriptions cause misunderstandings. But formal methods demands higher levels of expertise and it is hard to reuse them on implementation. This way, WARP tries to establish a framework to solve these questions. The framework should be sufficiently simple to be understood by both business analysts and developers. But powerful enough to be used in a production programming platform like .NET framework.

This high-level design can also be used to generate the persistence layer and the graphical user interface. Therefore all these components are executable and testable. The envisioned WARP approach for software development is summarized in the figure 2.2. The triangle represents the lower levels of V-Model for software development (described on sub section 3.2.2).

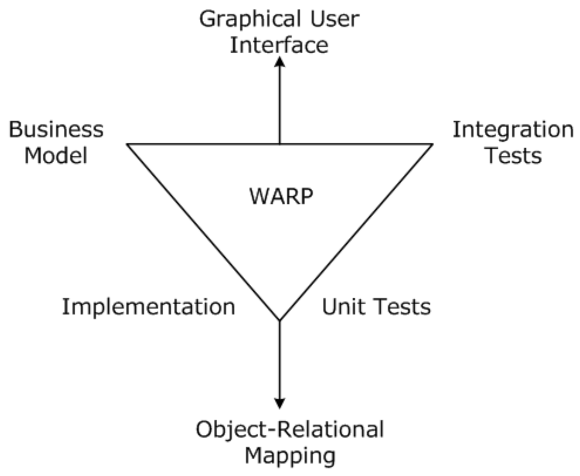


Figure 2.2: WARP approach

Additionally, a strong emphasis is given about new trends and tools on software engineering, like task automation or artifact traceability.

2.2 Scope

The WARP approach does not provide a complete methodology, neither a radical new philosophy to create software. But it provides techniques to be incorporated in existent software development processes, which can improve them.

The work focuses on improving communication and integration through the software development cycle. However the context is mainly technical and practical. That is to say, it concentrates on designing, programming and testing activities, rather than other conceptual solutions, e.g. improvement of requirements analysis or implementation of process standards. The practical component aims concretize the concepts, giving specific tools and explaining the implementation of executable samples, which are unfortunately many times forgotten.

The study of tools is constrained to Microsoft development environments, but most of the time, similar applications could be applied to other platforms.

Chapter 3

The software development process

Motto: Opportunity ⇒ Product ⇒ Value

A *software development process* is a systematic approach on the development of a software product. It is also referred as *software development life cycle* and *software process* [9]. The process starts from an identified opportunity, with available resources and management encouragement, to achieve a product. The product would create additional value in different forms, e.g. rising revenues, reducing costs or creating knowledge.

Every software process is unique, because it occurs in a circumstantial moment, with different people, within different organizations, different objectives, and so on. Custom development is becoming more and more important, namely in the business world [6], as it provides suitable solutions for companies requirements. That contrasts with traditional pre-developed software packages, targeted usually for a big number of unrelated users, with a regular rhythm of releases and with less client support, as in the games industry.

3.1 Software process activities

The software development life cycle is composed by several activities with different objectives, which could be done by the same or different people. There are six activities commonly accepted: requirements, architecture, design, implementation, testing and deployment.

3.1.1 Requirements

Software requirements analysis is an effort to determine the necessities and conditions in the creation of a new product or an alteration of an existent product. The analysis takes inputs from stakeholders to output artifacts that contain a high-level description of desired system. Examples of stakeholders would be the product visionary, the manager of actual or potential users, the acquiring customer, the marketing department, the funding sponsor and so on [10].

There are several techniques to capture requirements, for example interviews, group meetings, creation and discussion of visual models, and prototyping. The final outputs of this activity could be a mixture of text documents, use cases, models, prototypes, descriptive storyboards, event-response tables, more or less detailed, accordingly to the adopted methodologies. The level of requirements varies from a high-level description of product objectives, the business requirements, until functional and system requirements, which describe desired functionalities, non-functional requirements and its subcomponents.

This phase is famous by misunderstandings, because it implies combining different interests and points of view. For example, *a customer's definition of requirements might sound like a high-level product concept to the developer. The developer's notion of requirements might sound like detailed user interface design to the user* [10]. People have different backgrounds, some know more about the problem domain, whereas others know more about information systems. Natural language is ambiguous, causing multiple interpretations. This diversity of definitions leads to confusing and frustrating communication problems, during and in following stages.

In fact, some advocate that different teams to capture requirements and to do development is not so good as it appears, because intermediaries introduce noise between the client and developers [6]. In a similar way, many argue that requirements should not be too much detailed, as they could change quickly or never implemented, and represent a big effort.

3.1.2 Architecture

Software architecture is an activity concentrated on identifying and documenting the sub-components that composes the system and its relationships. The term is also applied to the documentation produced during effort [11]. The definition and structure of software architecture process is imprecise, but it concerns on a top-down analysis to build a system that fits the requirements, easy to develop but prepared to grow. Its outputs could be paper diagrams, class models, interface descriptions and a set of quality attributes (e.g. availability or reliability).

During the last years there was an attempt to collect the knowledge of this discipline, with architecture patterns, architectural styles, best practices, description languages and formal methods. Examples of results of this effort are the notable catalog of Martin Fowler [12] and Patterns and Best Practices for Enterprise Integrations [13].

3.1.3 Design

Software design is an associated activity of architecture, preparing and detailing a solution to be implemented. It includes reasoning about the components and their relationships, finding solutions and establishing their interfaces, allowing the concurrent development. It concerns about lower levels of detail, as well, algorithms, class diagrams, platform-specific constraints, and suitable programming paradigms and tools. It also aims improvement in several software aspects, like extensibility, compatibility and reliability.

As software architecture, this activity was undisciplined and only based on designer experience and intuition. There is no defined process that explains how to achieve a good architecture or a good design. However nowadays there is a good common knowledge, mainly organized in form of patterns or best practices, following its well-known predecessor *GoF Design Patterns* [14]. Nevertheless many knowledge remains dispersed around the Web and technical documentation, so the designer experience maintains its important role.

3.1.4 Implementation

Software implementation, also called software programming, concretizes all the reasoning and analysis into code, which can be executable and deployed to the customer's environment. In this phase the developer writes code using one or more programming languages and related platforms, e.g. Java and its class libraries. This task is usually supported by an integrated development environment, which provides several functionalities, as an editor, a compiler, a debugger, a source code control application and a testing framework. Choosing a good set of tools is a critical decision, with consequences in all aspects, including project progress and product quality.

Implementation is a difficult task, as *the programming languages are good to express what the computer should do, but they are less effective in expressing the developer intent* [2]. It demands expertise and knowledge in many areas, including product domain, algorithms, data structures and formal logic; sometimes a certain degree of imagination, creativity and passion to solve new problems. That leads to many different programmer backgrounds, joining mathematicians, physicians, engineers, visual designers and even biologists. Although this activity emerged in scientific and engineering environments and the effort in the last years to bring it discipline and automation, some continue defending that programming is a type of art.

3.1.5 Testing

Software testing is an activity to evaluate and ensure the quality of software products. Quality is an abstract concept, but usually it measures how much a product fulfills agreed requirements, functional and non-functional, and to verify faults or defects in any artifact. Nowadays it is mainly concentrated on implementation outputs[15]. However testing should be an ongoing activity, as eliminating defects early reduce the cost of their repairing.

The traditional testing methods are divided into black box and white box. The former stands for testing software without knowing its internals, to check if requirements were accomplished. The second advocates the importance of comprehend the data structures and algorithms, as well how code coverage can find an unused code. The current practice is a mix of these two [16], known as grey box testing .

Different types of tests have emerged, with specific objectives and scope:

- Unit testing verifies small components or functionalities, e.g. methods or classes behavior;
- Integration testing examines and identifies defects in the interfaces and between integrated components;
- Acceptance testing, also called system testing, checks if the product meets its requirements;
- Graphical user interface testing focuses on testing user interfaces, following the record and playback method, or providing inputs and defining expected changes on interface;
- Non-functional aspect testing tries to evaluate some characteristics of the system, as performance, usability and security.

There is a enormous controversy in software testing, namely about who takes the responsibility for testing, programmers or specialized testers, about the tradeoffs of using scripted tests instead of exploratory, and between manual and automated tests [17].

3.1.6 Deployment

Software deployment is a set of related tasks to put the product into production, that is to say, available for use. As any software product is unique, the precise processes or procedures to perform deployment are difficultly defined [18]. However, they usually include a release of a stable version of the code repository and install it on targeted machines. This installation could occur in a server machine or in scattered workstations, and usually requires some type of configuration. In some cases it even requires changing implementation to accommodate specific incompatibilities. Updating and uninstalling are also deployment activities.

3.2 Software life cycle models

Despite the disparity of situations on software development, since the first years of software engineering have appeared many models which try to give a common structure to the projects, improving their management and its organization. Some are formal and imply heavy and long processes, while others are considered lightweight methodologies. For example, some have less focus on documentation or in management tasks. This section intends to present the best-known models, providing a critical analysis about them.

3.2.1 Waterfall model

Waterfall model is the best-known and oldest software process model, where activities occur one after another, in discrete stages. The next phase should not start until results of previous are approved. Freezing parts of the development permits an ongoing development, where detected issues on previous stages were ignored or left for later resolution. That makes easier planning, tasks concurrency, specialization, and reduces the rework of producing and approving new documents and the discussion about specifications.

The waterfall life cycle is composed by five activities: requirements definition, system and software design, implementation and unit testing, integration and system testing, operation and maintenance. Figure 3.1 outlines these activities and their sequence. As you could see, the activities described in the section above are tightly related with this model.

Curiously the first mention of the term waterfall was published in an article of Winston Royce, which criticizes it and presents it as non-working model [19]. It is based on concepts of industries that produce physical items where the value of raw materials and the cost of redoing is prohibitive, as in the construction sector. However software development produce abstract products, composed by a set of zero and ones to be executed in computers. At least, it is possible completely to change them after they are built (although that does not mean easier). This abstract character also leads to communication problems and incorrect specifications. Therefore the software development has evolved to more iterative models, which consent changes and the unpredictability.

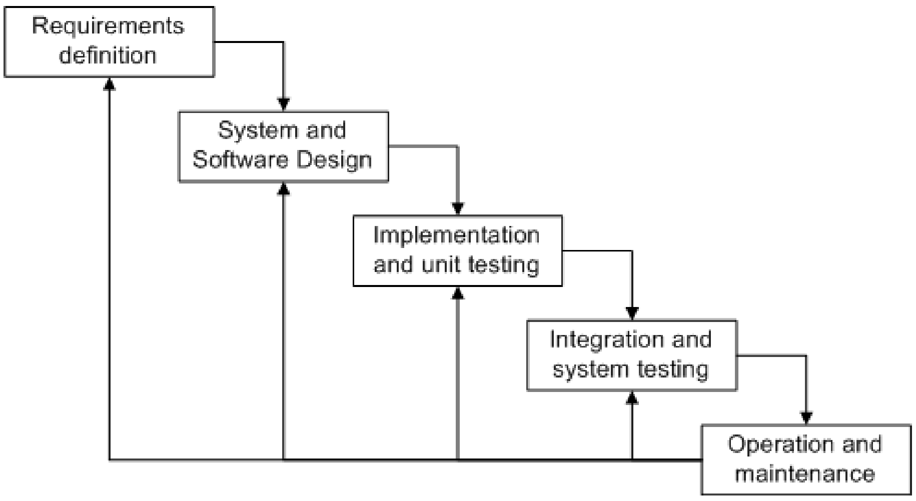

Requirements definition
System and Software Design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 3.1: Waterfall life cycle, based on [15]

Waterfall model is a classic model and it is taught in all software engineering courses. However nowadays it is barely used, at least, in a strict way, which does not admit modifications on results of previous activities. Nevertheless it is a good didactic concept, giving a general understanding about software development for beginners.

3.2.2 V-Model

V-Model makes explicit the importance of verification and validation phases, and how they relate each other. All testing activities have a correspondent development phase, where accordingly tests are planned and defined. So, for example, requirements are used to create acceptance tests, and integration tests are defined during the architecture stage. To show these relations a chart in form of V was created (figure 3.2). However the stages occur sequentially, which made V-Model an extension of waterfall.

V-Model was developed by German Federal Armed Forces, and is widely used in Germany, namely in the public sector [20]. It was a good step to show the importance of relating phases, and to emphasize the importance of testing. However it is a weight model, composed by sequential activities, which do not accept easily modifications.

3.2.3 Iterative models

Iterative models were created to accept changes, while maintaining a structured model. They are based on two main ideas: incremental delivery and iterative development. The former defends that any activity occurs by increments, each one adding value to the project, e.g. a release for any added functionality. The second

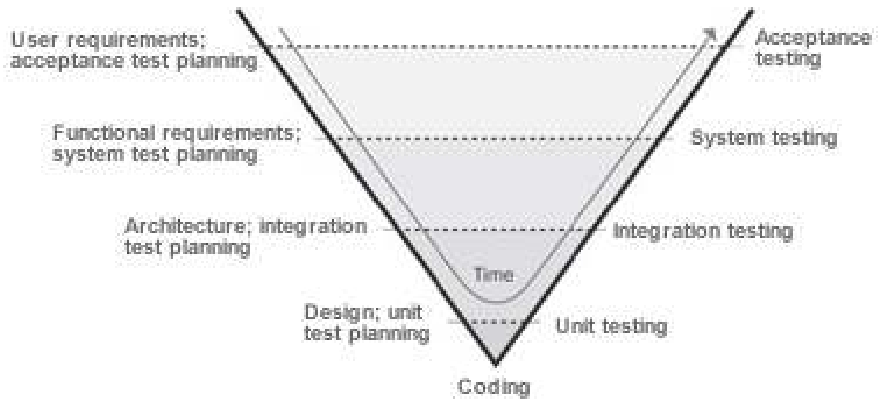


Figure 3.2: V-Model life cycle, retired from [10]

defends the importance of producing through iterations, which are repeated in a constant cadence, and providing feedback each other.

Two well-know iterative models are Spiral and Rational Unified Process (RUP) . The Spiral model was proposed by Boehm [21], integrating the concept of iteration with the waterfall model. Each iteration is like an entire waterfall process, recognizing the importance of planing and giving feedback to next iterations. He also alleged the importance of design and prototyping in software development. Figure 3.3 shows how these ideas are joined.

Rational Unified Process (RUP) is a framework created by Rational Software Corporation. It includes an iterative model, recommendations and tools. It is based on the six key principles of Business-Driven Development [23]:

1. Adapt the process;
2. Balance stakeholder priorities;
3. Collaborate across teams;
4. Demonstrate value iteratively;
5. Elevate the level of abstraction.

The RUP model is composed by four phases: inception, elaboration, construction and transition. Inception phase establishes product objectives, based on its potential value. Elaboration phase consists of thinking about the problem without any implementation, including domain analysis, business modeling, risk estimation, elaboration of use cases, prototyping, architecture and design of components. RUP also includes descriptions of engineering disciplines, similar to process activities described above, recommendations and tools to guide and bring discipline. Rational

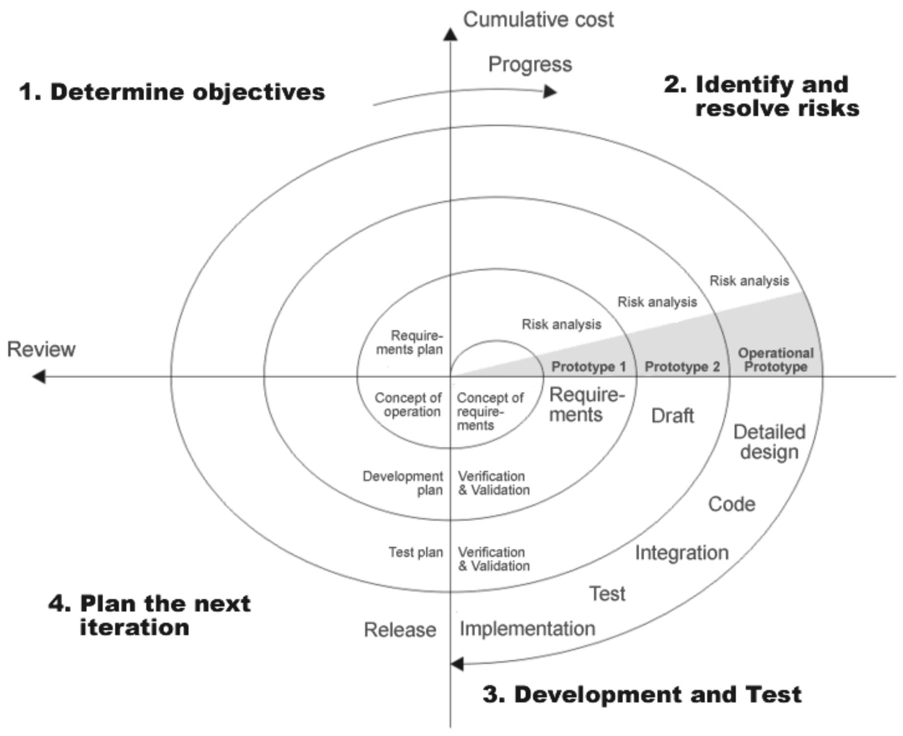


Figure 3.3: Spiral life cycle [22]

Software provides many different tools, like Rational Software Analyzer and Rational Rose.

3.2.4 Agile models

Iterative models are more adaptive to change, but some of them are so heavy and complex processes, that reduce the focus on producing valid software: people have lost with the burden of documentation and management activities. Agile models are iterative models which argue the importance of simplicity, communication and adaptability.

Agile methodologies are more people-centric, relying on trust, motivation and teamwork. The highest priority is the customer satisfaction, delivering him continuously working software for approval and discussion. Changing requirements are welcome, and the capability to accept them is seen as competitive advantage. Agile models favors oral communication, rather than written documentation, putting together business people and developers to work daily throughout the project. It also advocates the importance of technical excellence and good design, as well as continuous improvement. The Agile Manifest, promoted by some honorable and senior specialists, contains a good set of their principles [24].

Notable agile models are Scrum and Extreme Programming (XP). Scrum divides the process in sprints, a fifteen-thirty day period, where the team produces an executable increment to the product . Requirements are prioritized in the product backlog, and on each sprint the team takes from it the features that believe they could accomplish - the sprint backlog [25]. During the period the sprint backlog does not change (figure 3.4).

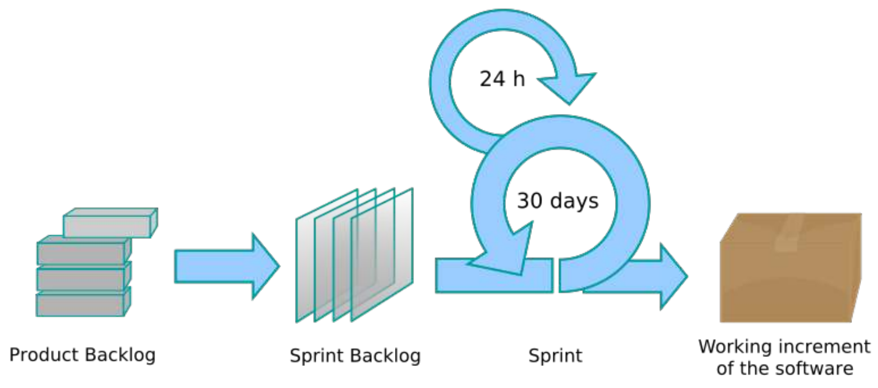


Figure 3.4: The Scrum model [26]

The model has some interesting curiosities, like the group of roles: pigs (product owner, facilitator and the team) for who are engaged on produce the software and chickens (users, customers and managers) which are related to the project. This nomenclature came from a joke, where a pig and a chicken would open a restaurant named *Hams and Eggs*, but the pig strongly refuses because it is committed while the chicken is only involved [26]. It also advocates realization of daily meetings to discuss the progress and difficulties arise. In these meetings people stands to ensure that they will be short. Scrum is remarkably simple, easing its implementation.

Extreme Programming follows the agile philosophy, but it is really critic about documentation tasks. It has four basic activities: coding, testing, listening and designing. Coding is the main activity of XP, which advocates that the code is most important result of software development. Without any runnable piece of software, the process does not produce anything with real value. Coding would be any task that creates executable code: class diagrams that will generate code, work flows that will be converted to running algorithms, and scripting code to be interpreted or compiled.

Testing is an important activity too: without perform tests, unit or acceptance tests, there is not certainty about the feature works as expected. Unit tests checks small units of code, while acceptance tests verify customer requirements. Therefore, Extreme Programming is usually implemented with test-driven development. Listening involves the understanding of problem domain, through communication between developers and business people, and designing is an ongoing process to improve implemented system, easing future changes and eliminating dependencies.

Other practices are included in Extreme Programming, like pair programming, continuous integration and refactoring. However it remains a controversial model, which many companies avoid implementing it completely, mainly because it is too risky and it is targeted for small teams. And managers tend to run away from risks. Meanwhile many of these techniques have been adapted to traditional models, with considerable benefits.

Agile models are seen as unmanageable and undisciplined; however their goal is the reverse. They argue that planning in detail the next phases, which are uncertain to happen and where requirements could change, is costly and unnecessary. Moreover their simplicity eases their implementation, in contrast with heavy processes that most of the times are partially followed. And many support the idea that they cannot be applied to large or distributed teams. However usually a project could be decoupled and its sub projects allocated to small teams [27]. Similarly, the distance between them could be minimized by techniques like kick-off meetings in one location, travelers that transmit information from one point to another and sharing tools [28]. Nevertheless agile philosophies usually do not succeed on command-and-control company cultures and when they are forced to people [29].

3.2.5 Microsoft Solutions Framework process model

Microsoft Solutions Framework (MSF) is *a deliberate and disciplined approach to technology projects based on a defined set of principles, models, disciplines, concepts, guidelines, and proven practices from Microsoft* [30]. Its objectives are to develop information technology solutions faster, with fewer people, less risk and higher quality results. It results from well-known industry best practices, and either 25 years of Microsoft experience.

MSF includes descriptions of process models fitted to Microsoft technologies, from the management concepts up to providing tools like Visual Studio Team System. These descriptions are very concrete, explaining potential artifact types, team roles, and actions to take. These process models are foundations, which could be customized to fit organization culture.

Although most of MSF concepts are not really new, it resumes many of the knowledge about software development. Tool-supported methodologies have many advantages, establishing rules, automating routine tasks and raising cooperation. Microsoft software development tools, namely the Visual Studio family of products, have pioneered many ideas into software engineering. This approach to improve the software development process is remarkably, as Microsoft has a strong influence in the information technology sector.

3.2.6 Cowboy coding

Cowboy coding is a term to refer the absence of any structure in software development, without organized teams or an adopted process model [31]. The decisions are left for coders, raising the uncertainty and unpredictability. Therefore it usually applies to individuals or small teams, where their passion and effort supersede the standards and best practices.

Surprisingly many successful entrepreneurial projects emerged in this type of environments, as academic and hobbyist projects. A good example is Adobe Photoshop which was created by Knoll brothers, because of their enthusiasm on photography [32]. In fact many organizations create small spin-offs to develop specific ideas and surpass organizational barriers and constraints on innovation [33]. That makes questionable the real value of bureaucracy, hierarchies, standards and regulations, and even process models.

3.3 Trends on software development

Software development is a recently activity, and it has been suffering a rapid maturation. There are a lot of ideas and techniques to improve development, however they are not really models. This section presents some of today trends about the software process.

3.3.1 Guidance on improvement

In the last years, models or techniques to guide process improvement have appeared within organizations. This sub-section presents two of these models for software engineering: lean software development and Capability Maturity Model Integration.

Lean development

Lean software development is a translation of manufacturing and supply chain management to software development, performed by Mary and Tom Poppendieck [6]. It is mainly adapted from the Toyota Product Development System, bringing terms like Just-in-Time and *Autonomation* into software development. Just-in-Time advocates the importance of producing only when products are necessary. And *Autonomation*, also called *stop-the-line*, is a way of organizing things that minimizes human intervention, where the work only stops when some irregularity happens. Their book is full of concepts and real stories, from both industries, which help to understand it.

Lean ideas are agile-friendly, and arguing that software development is an empirical process that must create some type of value. The seven principles of lean software development are:

1. Eliminate waste;
2. Build quality in;
3. Create knowledge;
4. Defer commitment;
5. Deliver fast;
6. Respect people;
7. Optimize the whole.

These principles drive multiple lessons in lean philosophy. For example, eliminating waste try to identify common problems on software development, which have similar meaning in manufacturing sector: partially done work, extra features, time between phases, relearning, handoffs, task switching, delays and defects. This principle provides mottos like *Write Less Code* and *Justify Every Feature*. In the same way defer commitment said that every product should be designed and built with change tolerance in mind. It also focuses the importance of create and maintain the knowledge, through scientific method, Kaizen events and simplicity on documentation.

Lean software development is more a philosophy, a set of principles that guides the improvement, rather than a true methodology. However most of their key ideas could be incorporated into any organizational culture.

Maturity models

Capability Maturity Model Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes. It is composed by proven models and methods to be adapted accordingly business objectives. In most of the cases, the implementation of CMMI raises productivity, quality and customer satisfaction; while it decreases the costs. However many criticize its complexity, and small companies are less likely to get benefits from it [34].

Many organizations have been submitted to CMMI appraisals, namely the larger ones, to identify areas of improvement and, in some cases, because clients require a certain level of maturity.

There are subsets of CMMI applied to specific aspects of software engineering. Personal Software Process (PSP) aims to improve the quality and productivity of individual engineers [35]. Team Software Process (TSP) aims to improve team work and management of software projects [36]

3.3.2 Forever beta products

Forever beta products have been a new trend on software development, where companies deliver incomplete and unfinished applications to users for a long time. *The idea behind is that solving 80% of a problem provides much more value than improving remainder 20%* [33]. Additionally users would test the product, giving their feedback. The unique compromise is that the product will improve if users like it. Estimations about the product value and repair of problems were done under user testing, much more accurately than research markets or technical experts could do.

The beta state does not mean, however, worse quality. All updates are carefully performed and documented. They work better in web-based applications, which ease deployment and frequently upgrades. Sometimes older versions were maintained, giving the user hypothesis to choose. And users have time to learn the new incremental functionalities. Beta state merely means under ongoing development, which in many cases means forever.

The most famous examples are Google applications. Google News [37] (English version) was created in 2001 and left beta stage only in 2006. Gmail [38], the Google mail service, starts in 2004 and remains beta. Google Docs [39], which provides editing of word documents, spreadsheets and presentations, is beta too. And there is another level of product development: Google Labs projects [40]. They are a first public appearance of favorite Google ideas, and could be considered an alpha version. Most of new features and products appeared firstly in Google Labs, and then the most successful graduated to beta version.

Many view the beta products as an excuse to avoid commitment and customer's support. In the same way, they are usually targeted for the general public products: signing contracts about forever beta products appears unrealistic in the business world. Nevertheless exposing their state is a way to inform users and establish expectations. Interestingly the users are enthusiastic about beta products, following the product roadmap and being happy to contribute to create a better tool. After all, most of applications, targeted for entertainment or for professionals, are under development forever.

3.3.3 Metrics and estimation

Estimation is widely used in software processes. Estimation tries to forecast the future and to do planning, based on historical archive of previous projects and their metrics. These metrics measure things like lines of code, function points, median time to perform some type of tasks or common delay in risky projects.

Metrics are also used to evaluate software quality. For example, NDepend [41] analysis assemblies to find dependencies between components and to remove complexity, while NCover [42] measures the percentage of code not tested.

Metrics and estimation are popular techniques, which help people manage their teams and projects. However they should be used carefully. Estimation techniques transform the work in quantities, forgetting the importance of qualities. Comparing numbers, e.g. the number of lines of code, to evaluate developer's productivity is too naive. That approach does not capture the potential value, the level of difficulty and quality of implementation [7]. In the same way, software with less dependencies does not mean necessarily better software.

3.3.4 Tool supported and automation

Nowadays most software development processes are supported by tools, which help the developer, automate routine tasks and raise communication. That leads the team focus on customer and problem solving, increasing overall productivity.

Project management has a lot of tools available too. Microsoft Project [43] is a general and widely used application. Mingle [44] is a solution of ThoughtWorks for management and team communication on agile environments.

In the same way, the process of compiling and joining components is usually automated using building tools, like Apache Ant [45] or MSBuild [46].

Software testing has improved very much in the recent years, particularly due to automation on tests. Figure 3.5 shows that today most tests could be scripted and then executed without any human intervention. Acceptance tests, or story tests, measure the customer's satisfaction. Unit tests check the validity of small pieces of code. Property tests evaluate general characteristics of the system, like performance. The exceptions are some quality attributes as usability, which requires understanding human beings.

Examples of these tools are:

- **FitNesse** [47] - is an acceptance testing framework, integrated with a wiki for team and customer collaboration. Tests are based on visual tables with inputs and expected outputs, and fixture code that calls correspondent Java or C# classes;

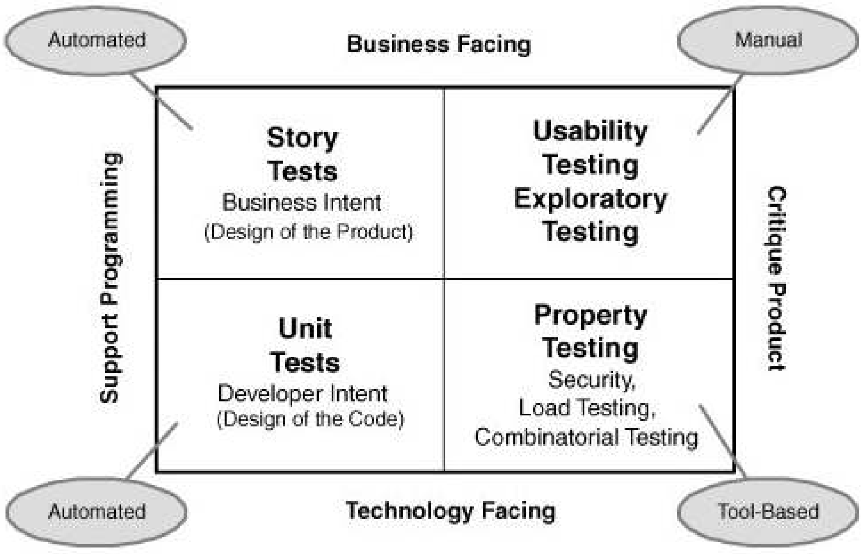


Figure 3.5: Types of testing [6]

- **NUnit** [48] - is a unit-testing framework for all .NET languages (for a more complete list of available unit-testing frameworks see [49]);

- **Selenium** [50] - is a test framework for web applications, based on record and play concept. It works like a normal browser, making requests and verifying the responses;

- **WatiN** [51] - is another test framework for web applications, but where the tests are scripted in C#;

- **Visual Studio Team System Test Edition** [52] - is the testing framework included in Visual Studio Team System, including support for unit-tests, web tests, load tests and code coverage analysis.

Human code reviews to enforce standards and find defects is a waste[6]. Today code analysis tools, as FxCop [53], do that in an automatic and systematic manner. And they could be customized adding specific rules of the organization.

However many software engineering tools are still too expensive, erecting barriers to their utilization. Fortunately there are alternatives from Open Source, and some could be considered better than proprietary solutions. For example, Subversion (SVN) is a widespread, free and usable tool to source control. Tools do not substitute people, but they can make their life much easier.

3.3.5 Integrated collaboration environment

Integrated collaboration environments (ICEs) [54] are an extension of tool supported processes, providing a set of related tools which support development, including team communication and source control. They are a natural evolution of integration development environments (IDEs). Automation and information integration rises when different types of tools communicate with each other. For example, when an item is added to bug list, the project planning is automatically updated and warn involved people.

Process templates

One of the notable characteristics of Microsoft Solutions Framework is process templates [55]. They bring discipline to software development based on tools that assist and follow the adopted process model. Process templates establish working rules and constraints in development, including the types of artifacts and team roles.

Microsoft Team Foundation Server and its client Visual Studio Team System provide tools like source control, item tracking, bug lists, status report, guidance, and planning tools. Their behavior is based on the adopted process template. MSF version 4.0 includes two different templates:

- MSF for Agile Software Development;
- MSF for Capability Maturity Model Integration (CMMI) Process Improvement.

As expected MSF for agile is simpler than MSF for CMMI, with less roles and less types of artifacts. Other templates are created by Microsoft partners and Open Source world, for Scrum, Extreme Programming and V-Model XT (the successor of V-Model). These models are customizable to fit specific aspects of organizations, using the Process Template Editor, included in Visual Studio 2005 Team Foundation Server Power Tools.

Artifact traceability

The number of artifacts used and produced in the context of software development is enormous, and usually there are too many disparate types of items: interview summaries, requirements documents, project plan, technical reviews, code files, compiled assemblies, XML files, relational schemas, test reports, etc. And usually they have multiple versions. The challenge is to relate the artifacts to each other and manage and update these relationships afterwards. The traditional approach is textual references between items, which imply manual updates and spare readings around different files. Finding and exploring relationships is a hard effort in this context.

Fortunately, nowadays there are tools that relate artifacts. Microsoft Team Foundation Server has a tracking feature for work items, that is, artifacts which are not code, and source code. Work items could be requirements, bugs, tasks or change requests. For example, when the developer checks in his code, he can notify that one bug is resolved. The team easily follows the progress, by visualizing the lists of work items and summarizing charts. That makes the communication and management much easier.

Continuous integration

Continuous integration is a widespread software development practice promoted by Martin Fowler. It advocates that developers should integrate their work frequently, at least every day [56]. An automated build and the subsequent tests are performed to find integration errors as quickly as possible. This approach substitutes big-bang integration phases, which are usually long and risky. The system is viewed as a whole, where everyone knows how components interact. And it enables frequent deployment, providing new features quickly, and then a faster customer's feedback.

The key practices of continuous integration are:

- Maintain a single source repository;
- Automate the build;
- Make your build self-testing;;
- Everyone commits every day;
- Every commit should build the mainline on a integration machine;
- Keep the build fast, e.g. creating different building stages or using mock objects;
- Test in a clone of the production environment;
- Make it easy for anyone to get the latest executable, e.g. from integration machine;
- Everyone can see what is happening;
- Automate deployment.

ThoughtWorks, where Martin Fowler works, has developed two continuous integration servers: CruiseControl and CruiseControl.NET. The former runs on Java platform and the second on .NET framework, however both have similar functionality. They works with many different tools, as Subversion, NAnt, NUnit, FitNesse, FxCop, NDepend and NCover.

3.3.6 New programming paradigms

Programming paradigms have an important role in software development. There are a lot of paradigms, with multiple levels of abstraction and different objectives. Each paradigm usually has a large list of possible programming languages and related frameworks.

From the first days of low-level programming, where the code was tightly related with hardware, much has changed. Compilers or interpreters transform more abstract languages into machine code. Today it is common that developers do not care about processor, memory, network, and even operating system constraints.

Higher abstraction boosts developer productivity. Most of the time, the performance penalty of abstraction is neglected due to advantages during development. And after all, it is not guaranteed that human made better resource management than automatic algorithms. Therefore, these new paradigms have been adopted by software industry increasingly. Even in embedded systems like mobile phones, which usually have very limited resources, the development is moving to different paradigms.

Procedural and imperative programming is widely adopted, with *ancient* languages as C and Cobol, and some recent languages which tolerate this paradigm, like C++ and PHP. Nowadays, however, object-oriented programming (OOP) became a "de facto" standard. It uses objects and their interactions as base concepts to design applications, including techniques like inheritance, encapsulation, modularity, and polymorphism. Java and C#, two object-oriented languages, are largely used and documented. In fact, nine of the first ten languages listed on TIOBE Programming Community Index for June 2008 [57], a ranking which measures language popularity on the web, has some kind of support for objects (the exception is C).

Aspect Oriented Programming (AOP) has appeared to solve some problems of object-oriented paradigm, namely on implementation of cross-cutting concerns. They are concerns scattered around the code and hard to isolate in components, which *cut* across multiples components. Examples of cross-cutting concerns are logging and consistent error handling. Aspects are implemented defining the code and when that code is executed. However they do not substitute objects: they are a complement of them. AspectJ and their associated tools constitute the most notable aspect framework, to weave aspects to Java objects. Nowadays adoption of aspect oriented paradigm is quite risky, because it is a technology in development, with lack of tool support and people with knowledge about it.

Very high-level programming languages have emerged, like Pyhton, Ruby and Scheme [58]. They are usually minimalist and multi-paradigm including procedural, functional and object-oriented. The developer chooses the more appropriate paradigm accordingly to his context. With few lines the developer can create powerful applications. However some advocate that this flexibility leads indiscipline to coding activities and makes hard code understandability.

There are a lot of other programming paradigms [59]: event-driven, logic programming, process-oriented, based on constraints, and so on. Some are created to accomplish particular problems, while others maintain a research status. Although most are confined into specific areas, all of them deserves attention when evaluate new paradigms to adopt on software development process.

Service Oriented Architecture (SOA) is not really a programming paradigm, but it is a way of thinking to organize and guide software development adopted by many organizations. Business processes are modeled as services, favoring separation of concerns and distributing.

The multiplicity of programming paradigms, and associated technologies, confuse software engineers. The controversy, with disparate comparisons and definitions, around them are messy. That makes it harder to choose the most suitable solution to fulfill business requirements. Additionally, the effort to move to different paradigms is considerable, because it implies a change on developer's mindset. Nonetheless, all these new paradigms have been contributing to increase developer's productivity and software quality.

3.3.7 Code as design

Code as design was an idea exposed in 1994 by Jack W. Reeves about software development [60]. The article is highly controversial, arguing that coding is design. The main idea is devaluate the importance of activities that occur before coding. He advocates that programming is *still part* of design activities. The code is the detailed design, which is based and influences the high-level design.

Good top level design is important, but never should include too much detail. Formal documentation should be delayed as much possible, namely design documents. It is frequent that a programmer finds a problem that implies changes on high-level design. Designers can use anything that helps them in reasoning and communication: structure charts, class diagrams, and so on. But for him that is not software design.

This idea emerged as a response from who felt that bureaucracy has been slowing software engineering processes. People spend much of their time in management and documentation activities, which add nothing into customer environment. In today changing environments, producing detailed documentation before coding appears ingenuous. Businesses, technologies and people change every day. And feedback provided by implementation probably will change the design. Revising documentation, or even throwing away it, after implementation is a waste. Moreover it is known that many people spent their precious time producing documentation which nobody will read or retrieve value from it.

Albeit the author was exaggerated in some points of views, and he was too scrupulous about definitions, his text explains how programming and the built/test cycle are central to the process of engineering software, instead of remaining activities.

In fact, this idea has been reinforced is the last years. Extreme Programming centers on coding activities. Interestingly many Open Source projects have minimum requirement and design stages, their management is lightweight, and code repositories plays a central role. And Microsoft strategy for Visual Studio 2005 recognizes the limitations of visual modeling, e.g. UML diagrams, good for sketching and expose conceptual abstractions, but bad for detailed specifications [61].

3.3.8 Software prototyping

Prototyping has been used since the first software days. It is a technique of fast developing raw products, that could be presented to the customer and analyzed, but are not prepared to be deployed on customer's site. That technique is used in many domains, from building architecture to electrical engineering. Prototyping became a popular and widely used technique due to the abstract value of the software.

Most of process models advocate the importance of prototyping. When the development starts many unexpected problems arise, which must be discussed and solved. Prototypes could be functional, or they only expose the interfaces. A visual prototype eases its understandability. Prototypes could be throwaway, or used as code basis to start the development. Creating prototypes raises awareness of software constraints and improves the knowledge about problem domain, which consequently helps in project estimation and scheduling.

Nowadays new techniques have arisen to help prototyping. For example, scaffolfing techniques generate simple graphical user interfaces based on classes or relational tables. Other tools create prototypes using visual and drag-and-drop features, without any coding involved. Techniques like these have been reducing the costs of prototyping.

Prototyping reduces the costs and lowers the risk of software process. However, as any technique, its misusing can be disadvantageous. Many times customers think that a prototype is an almost finished system, and they do not understand that is a lower quality output. Moreover people lose many time prototyping or lose focus on other analysis and design problems. Nevertheless prototyping is a powerful technique, recommended for most of software projects.

3.3.9 Pattern & Practices

Resuming the knowledge about software development is not an easy task. The experience and intuition yet plays an important role. Each software has unique characteristics, making hard to communicate the expertise acquired in previous works.

Patterns and best practices descriptions on software development appeared to describe and transmit knowledge owned by the specialists. They give a general solution for a common problem, showing examples and possible variations. Additionally they provide a common vocabulary for developers. Design and architectural patterns are the best-known in software development. Good references on these topics are [12] [13] [14] [62]. Interestingly, anti-patterns emerged too, describing obvious but inefficient solutions.

The initiative Patterns & Practices [63], from Microsoft, guide architects and software developers on their tasks. Although they are fitted to Microsoft technologies, most of them provide good lessons to every software engineer. In the same context Guidance Automation Toolkit (GAT) [64] provides tools and methods to distribute expertise and educate developers, including recipes and wizards.

3.4 Discussion

There is not a silver bullet in software development. Many processes and techniques were experimented, in many different contexts, and there is not a unique solution. The controversy about this topic was scattered around books, thesis and the Web. Some defend their approach as a religion, without recognizing their disadvantages. Others neither remember the importance of improving the processes, and blame developers for the lower productivity.

The discussion about process models happens mainly because they are abstract concepts, which generate different interpretations, and therefore different implementations. However is not necessarily bad, as the business, people and tools are different; a different process realization is expectable. But this confusion generated strange implementations, e.g. the requirements phase is long and weight but then development processes are agile.

Although iterative processes are commonly accepted as better than standard waterfall processes, and agile philosophies have been embraced in the last years, there is no general agreement. *Post-Agilism* [65] and *Context Driven School* [1] have emerged, as informal movements which prefer adopting methods and ideas from various methodologies, avoiding constraints and dogmas of existent models, as well as the discussions about which is better. These schools propose that people can create their own processes, models and techniques.

There is not the best process. The trick is to study possible alternatives, and adopts the ones that resolve a specific development problem. The methodologies should not be copied, but criticized and adapted to the context, beware of their tradeoffs - this is the motto of context-driven school. For example, in a team of two developers, using a complex project management tool is nonsense. Besides, a good software development process does not guarantee success: involved people and potential product value play also important roles in this equation.

Chapter 4

Business modeling

Motto: Words ⇒ Logic ⇒ Code

In the context of software development, business modeling is an activity that captures and transforms the organizational reality into conceptual entities. Its objective is mapping textual and oral visions to some type of logic and more formal specifications, which then would be converted to code. The business is summarized in form of models and rules. The capacity to understand the problem is a determining factor to the success of any project.

WARP approach advocates that business modeling should drive software development. An abstract and formal business model could build a communication bridge between conceptual specifications, created by analysts, and source code, produced by developers. The formal business model can be executable, creating an opportunity for reusing and testing. Implementation reuses the executable business logic and a new type of tests can be created over business model, more focused on business concerns.

As business model is executable, it can be used to drive the creation of persistence layer and simple user interfaces. In this way two techniques are studied: object-relational mapping and user interfaces prototyping. The former minimizes the effort to manipulate persistent data, mapping objects to relational schemas. The second creates user interfaces which help understanding the system.

In fact, this idea is not new. Since the 1980s, model-driven engineering (MDE) [66] have promoted the systematic use of models through the software process, supported by CASE tools. The models are visual, e.g. UML use cases, and can be easily understood. However most of these models are not executable and are not easily synchronized with production code. All changes must occur in both models and code. Maintaining these models afterwards could be a tedious and worthless task. Therefore models rapidly become outdated. Although these techniques are very popular, in practice model-driven engineering tools are more fitted to sketch and explain certain aspects of software, rather than create full specifications.

Meanwhile Eric Evans suggested domain-driven design (DDD) in his book [67]. Domain-driven design advocates the importance of knowing the whole domain and understands domain logic to create better applications. Creating a good domain model is a master key to achieve success. It is a way of thinking and creating a ubiquitous language about the problem domain, easing the communication between team members, linking model and implementation more deeply. It explains how to model, but it is technological agnostic. WARP approach was strongly influenced by domain-driven design ideals.

4.1 Types of business modeling

Business modeling is an important activity of software development. However two different types could be perceived: throw-away and evolutionary.

4.1.1 Throw-away business modeling

The traditional techniques of business modeling are mainly based on textual descriptions and visual charts. They create a general understanding of the system, but become easily desynchronized with the implemented system. Therefore these types of business modeling must be updated in later stages of software development, or even thrown-away.

4.1.2 Evolutionary business modeling

Evolutionary business modeling is an effort to integrate business model with design and coding activities. It relies on mechanisms to automatically convert between models and code, easing the maintenance and synchronism of both. There are two approaches to synchronize them: code generation and reverse engineering. Both could exist in the same environment.

Code generation

Code generation techniques create automatically a code to targeted platforms, e.g. C++ or Java, from charts and higher level specifications. For example, in figure 4.1, one UML class diagram origins a set of class skeletons with right attributes and methods. Then these skeletons could be edited or called as independent components.

Model-driven architecture (MDA) has promoted this approach to solve the problems of model-driven engineering. The abstract specifications and visual diagrams have an internal syntax, which could be converted into code using appropriate tools

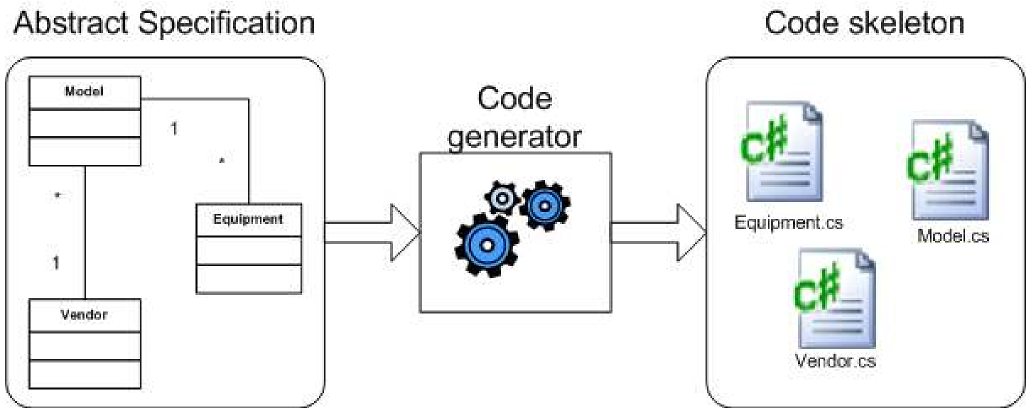


Figure 4.1: Code generation from a UML class diagram

[68]. Rational Rose applications [69] have integrated several generative techniques. Likewise Generative Modeling Technologies (GMT) [70] project of Eclipse Foundation has been producing a set of prototypes on model-driven engineering area.

Most of the time, however, the generated code is a simple code skeleton, which is easily coded in a few hours. Likewise most of the time a change on business model breaks completely the compatibility with an older code. Finally, in some cases reflecting changes on code into specifications is a nearly impossible or painful task.

Reverse engineering

Reverse engineering techniques follow the reverse direction: they generate visual diagrams and other specifications from source code. The source code is the primary representation. Any change on code is immediately reflected on models, as any change on models affects directly the code.

For example, Visual Studio Class Designer supports this type of modeling. The code is the original representation, which could be edited through a richer graphical user interface. It provides the following features:

- A visual representation of classes, synchronized with the code;
- Visually edition of fields, properties, methods and even events;
- The *object test bench* feature permits instantiate objects and call methods;
- Supports the managed languages: C#, VB.NET and J# in VS 2005, and VS2008 adds support for native C++ code with some limitations [71].

Figure 4.2 shows a simplified chart with entities involved in bank business.

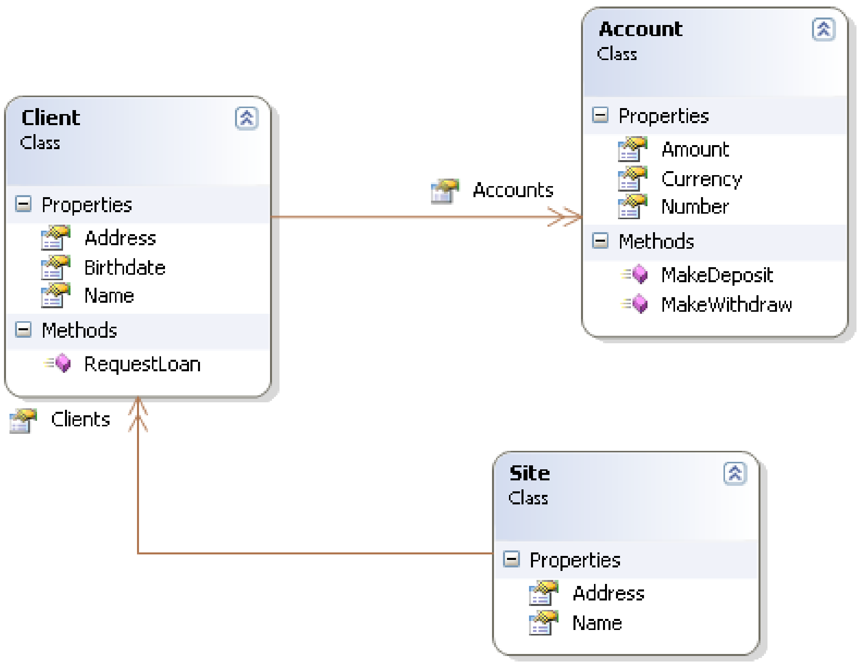


Figure 4.2: Simple class diagram created using Visual Studio Class Designer

Reverse engineering techniques is always synchronized with a code, and helps people understand it. However reverse engineering techniques are highly coupled with technological constraints. Creating more complex, higher-level and imaginative models is restricted, namely because of programming languages limitations. For example, generate work flows or use cases diagrams from code is quite challenging.

4.2 Modeling techniques

4.2.1 Textual descriptions

The easiest way to transmit intentions is through words. Oral communication is great, but sometimes it is necessary to record and to summarize these intentions. Therefore people have written texts, as vision and requirement documents, which explain the problem domain and the desired software. The business is modeled with descriptive descriptions, e.g. goal lists, use cases statements, or executive summaries.

Nowadays almost everyone can read texts, so this is a universal technique. Every customer and every developer understands them, better or worse. Ultimately the communication between people occur using words, even when they talk about complex models. Although the inherent ambiguity of natural language causes multiple interpretations, the power of textual descriptions should not be underestimated.

4.2.2 Visual models

Visual modeling is a way to explain and organize knowledge through images. After all, an image can transmit much more information than thousands of words. Creating images to model the business is a clever option.

Probably the best-known modeling technique in software development is Unified Modeling Language (UML), and their associated CASE applications [72]. UML is a visual language promoted by Object Management Group (OMG), targeted to do specification and documentation of software systems. It provides a common representation to describe structure and behavior of systems. Figure 4.3 shows the diagram types available in UML 2.0.

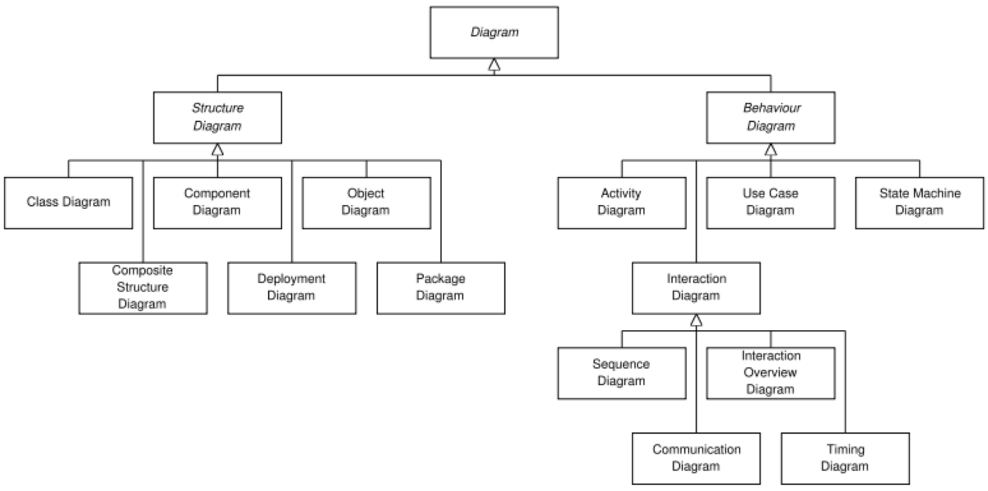


Figure 4.3: Types of diagrams in UML 2.0 [73]

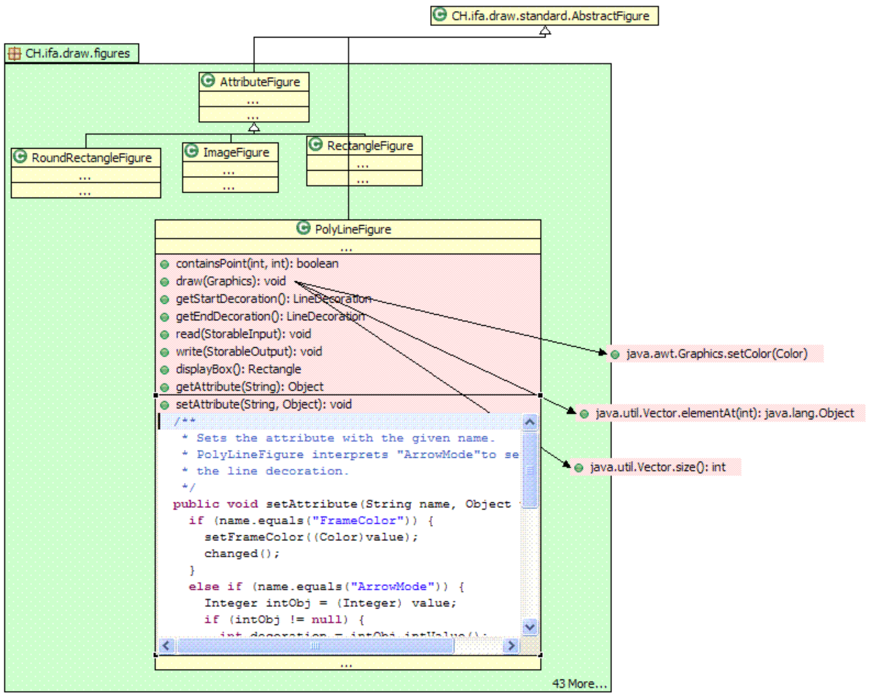
The creation of UML diagrams is usually supported by CASE software, like Rational Rose Modeler [69] or Microsoft Office Visio [74].

Model-driven engineering is highly focused on visual models. People draw and maintain their models, while implementing the application. Generative techniques produce automatically code from models, reducing the mismatch between them. Structure diagrams are appropriate for generative techniques, because most of the time there are correspondent elements in the programming platform. For example, a UML class diagram is easily mapped to object-oriented languages. However translating behavior diagrams is trickier, mainly because they are ambiguous and it is hard to transform those behaviors into executable code.

Nevertheless there are interesting initiatives. Windows Workflow Foundation (WF) uses work flow diagrams to map application behavior. The Workflow Designer, one of their components, allows a visual creation of these work-flows [75]. It is a part of .NET framework 3.0 and could be used to develop .NET applications. In contrast with other work-flow engines, Workflow Foundation is quite reliable and customizable [76].

Business Process Modeling Notation [77] is a graphical language for defining business processes in work-flows. It is promoted by Object Management Group, as a possible standard for communication between all stakeholders. Although it has the same lacks of all visual models, it appears to be more complete and more business-oriented rather than its competitors, namely UML.

Reverse engineering techniques enabled the possibility to create models from source code. Visual Studio Class Designer shows and edits .NET code in a visual way. Likewise Relo [78] is a code visualization application to Java and Eclipse, developed by Massachusetts Institute of Technology. The developer cans explore the code, navigating through packages, classes, attributes and methods. He cans zoom out to get a broader view of the system, or zoom in until reading the correspondent code. Figure 4.4 shows a screenshot of Relo user interface.


CH.ifa.draw.standard.AbstractFigure
CH.ifa.draw.figures
AttributeFigure
RoundRectangleFigure
ImageFigure
RectangleFigure
PolyLineFigure
containsPoint(int, int): boolean
draw(Graphics): void
getStartDecoration(): LineDecoration
getEndDecoration(): LineDecoration
read(StorableInput): void
write(StorableOutput): void
displayBox(): Rectangle
getAttribute(String): Object
setAttribute(String, Object): void
java.awt.Graphics.setColor(Color)
java.util.Vector.elementAt(int): java.lang.Object
java.util.Vector.size(): int
43 More...

```
/**
 * Sets the attribute with the given name.
 * PolyLineFigure interprets "ArrowMode"to se
 * the line decoration.
 */
public void setAttribute(String name, Object
  if (name.equals("FrameColor")) {
    setFrameColor((Color)value);
    changed();
  }
  else if (name.equals("ArrowMode")) {
    Integer intObj = (Integer) value;
    if (intObj != null) {
```

Figure 4.4: Screenshot of Relo user interface [78]

Visual models are very popular in software development. Many people, namely those with business background, are very reluctant on using programming languages. But they understood the limitations of textual descriptions. Visual models support reasoning and knowledge organization. That is, they are great to establish general understandings, namely to explain complex issues.

However visual models are not formal, executable or testable, which means that they are ambiguous like natural language. An activity diagram could be interpreted differently by the analyst and the developer. Moreover maintaining models and code synchronized is a heavy and tedious task. It is impossible to use diff tools with visual models, and it is difficult to perform searches. Therefore driving the development only through these models could be prejudicial. Likewise, *code as design* apologists argue that they are only a support for reasoning (for more see section 3.3.7).

Visual models with constraints

The specifications based on visual models are ambiguous and incomplete. Therefore new languages appear to extend them. The best-known example is Object Constraint Language (OCL), which enables the definition of rules about UML items.

OCL is mainly based in constraints about classes' attributes and methods. A constraint has a context, properties, operations and languages keywords. Figure 4.5 shows an example of an OCL constraint. It defines an invariant for the income of every person. If the person is employed, the income is equal or higher than 100. If not, the income is less than 100.

```
context Person inv:
let income : Integer = self.job.salary->sum() in
if isUnemployed then
   income < 100
else
   income >= 100
endif
```

Figure 4.5: Example of an OCL constraint [79]

There are three types of constraints in OCL: invariants, preconditions and postconditions. The former are eternal truths, which must be ensured during all object life. The second are conditions that must be ensured before executing a certain code. And the third defines conditions that must be matched after execute the code.

The development of OCL is supported by Object Management Group, and it is highly documented. And there a lot of tools which support OCL [80]. However it is questionable to learn a quite complicated language only to write or read documentation. The process of map OCL constraints to software code is complex and error prone. Therefore using traditional programming languages to define these constraints could be a better approach.

A great effort has been done to create OCL compilers and parsers, which transform it to another language, e.g. C#, or to machine binaries. OCL Compiler for

.NET is an example of this effort [81]. Alternatively, programming languages have started to support definitions of constraints, namely through programming by contract approaches (see more on 4.2.5).

4.2.3 Formal methods and model-based testing

Formal methods appeared to help analysts in the specification and validation of software. They are highly based on mathematics concepts. Business analysts produce executable models, to understand better the behavior of the system and discover potential defects. Vienna Development Method (VDM) is a well-known formal method. It is composed by a formal and object-oriented language and tools to test the validity of software. The language also support programming by contract techniques (see more on sub-section 4.2.5).

Similarly, model-based testing helps analysts to be aware of all system behaviors and consequences. It advocates the generation of test cases to verify models [82]. However unit tests are not enough: exploratory testing techniques should be used, to ensure that a good set of possibilities will be tested.

Microsoft Research has developed a tool to do model-based testing: Spec Explorer [83]. It generates finite states machines, with possible application states and transitions. These diagrams help finding many errors, but also generate confusion due to their complexity. Figure 4.6 shows the state machine produced for by Spec Explorer for a system with two actions: turn on and turn off.

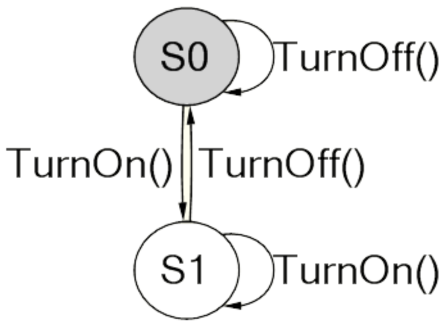


Figure 4.6: Example of a model-based test produced by Spec Explorer

Unfortunately, both methods, formal methods and model-based testing, have emerged to do thrown-away business modeling. That makes them expensive techniques, making them more applicable in the production of critical software. They are great to do business modeling, as well as to discover failures. But by now, they are not integrated with traditional development environments, and map has the same risks of other modeling techniques.

4.2.4 Domain specific languages

A domain specific language (DSL) is a computer language that is targeted to a particular kind of problem, rather than a general purpose language that is aimed at any kind of software problem[84]. It is implemented for particular contexts, reducing the complexity of general-purpose languages and defining the vocabulary of the problem. For instance, SQL is a domain-specific language for data manipulation.

Domain specific languages have become popular because they are very good in their tasks, and usually are quite simple to use. However creating one domain specific language is quite more complex. But it is becoming more and more accessible, with extensive documentation and easier frameworks to create compilers.

Domain specific languages could be divided in four categories [2]:

1. External DSL;
2. Graphical DSL;
3. Fluent Interfaces;
4. Internal/Embedded DSL.

External DSL

An external domain specific language is a separate language used to solve a specific problem, but used in cooperation with traditional programming languages. XML, SQL and Regular Expressions are examples of external domain specific languages.

The key characteristic of external DSLs is that they were written using traditional and complex compiler technology. That provides flexibility on the language syntax, but increases implementation complexity. Besides they lack symbolic integration with other programming languages, that is, they could not be linked into base language [85].

Graphical DSL

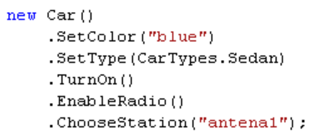
Graphical domain specific languages explain the domain through visual models. They are very good for documentation, but they are less effective to implement software (see section 4.2.2).

Microsoft has been promoting a framework to develop graphical DSLs: Domain-Specific Language Tools. It is based on the concept of Visual Studio Class Designer: the developer uses a graphical designer for editing domain models, within Visual Studio 2008. These models are saved as XML files and can generate code. New domain specific languages are defined using Visual Studio 2008 SDK[86].

Domain Builder [87] is an envisioned project to create a graphical domain-specific language to guide software development, using Microsoft Domain-Specific Language Tools. Additionally it would use CSLA [88] to generate the business persistence layer and NHibernate [89] to create the data access layer. Although it is an interesting project, it is in a preliminary state of development.

Fluent Interfaces

Fluent interface is a technique to organize application programming interfaces in a way that operations flow in a natural manner, making the code more readable and powerful. For example, figure 4.7 shows how a fluent interface is useful in the context of a car simulator.



```
new Car()
    .SetColor("blue")
    .SetType(CarTypes.Sedan)
    .TurnOn()
    .EnableRadio()
    .ChooseStation("antena1");
```

Figure 4.7: Example of a fluent interface in C#, based on [2] examples

However creating fluent interfaces is a hard effort, involving not only a strong knowledge about complex language features, as generics and operator overloading, but also an extra design effort to accept operations through a flow.

Internal DSL

Internal domain specific languages, also called embedded domain specific languages, are defined on base languages, e.g. Java or Python. That provides symbol integration and better access to development environment, including available frameworks and tools.

Many specialists recommend internal domain specific languages to model domains. They could be more abstract, with the semantics of problem domain, but maintain the same syntax of base language. That eliminates synchronization issues, because the models are represented with final code.

It is commonly accepted that mainstream languages as Java and C# are not suitable for internal DSLs. More flexible languages, like Ruby and Lisp, are often cited as the best to create an internal domain specific language. However not always developers change easily their mindset and adopt new programming paradigms.

This section is a short summary about domain specific languages, more details about them could be found in [2] and [90].

4.2.5 Programming by contract

Programming by contract is an approach to design software based on compromises between the components. It is also known by Design by Contract (DbC), a registered trademark of Eiffel Software (previously known as Interactive Software Engineering, Inc.) in the United States [91] [92].This company promotes the *Eiffel Development Framework*, strongly influenced by programming by contract techniques.

Definition of contracts in the code is a way to avoid misunderstandings between components. Defining conditions and expectations for every method, that is to say, their possible inputs, outputs, exceptions and side effects, limits the chances of implementation errors. For example, a contract could be that the amount of a bank account must be always equal or higher than zero.

Programming by contract is very similar to OCL approach, including the focus on constraints: preconditions, postconditions and invariants. The advantage of using programming by contract is that models are executable and easily connected with the implementation code.

Programming languages can support programming by contract natively or using third-party tools. The former is easy to use and more supported by tools and documentation. However few languages have natively included programming by contract. Obviously, Eiffel language supports it natively. A Microsoft research project created Spec#, a language for .NET that supports it too (see more in section A.1.1). Nevertheless they are not widespread languages.

Traditional programming languages have simpler contracts techniques, like code assertions, which throw exceptions or exit the current method. Most of their concepts, like preconditions and postconditions could be encapsulated in classes. The exception is invariants, which require a static verifier during program execution.

Nevertheless it would be preferable to use a framework which support programming contract. Common approaches include preprocessors or libraries. Preprocessors analyze the code, to find special contract syntax, which must be converted into regular code. Libraries expose interfaces to establish contracts. For example, Contract4J [93] provides programming by contract for Java. Contracts are defined using Java annotations. The static verifier was implemented using aspects written in AspectJ. Java Modeling Language (JML) [94] adds contracts in form of Java code comments, which can be processed and transformed to Java bytecode, using the JML compiler.

The most powerful technique of programming by contract is invariants. After all, utilization of preconditions and postconditions is an organized and systematic technique, but does not add much more of common checking done before and after executing a method. Essentially, this programming technique shines because it avoids erroneous interactions through detailed code specifications, improving the software quality.

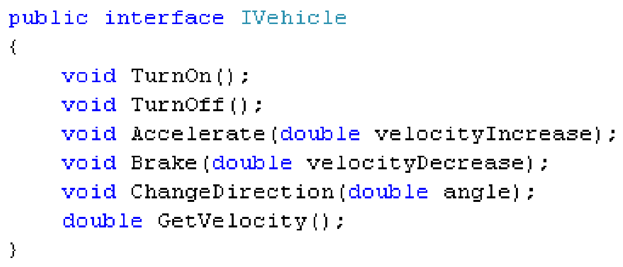
4.2.6 Inheritance as contract

This technique appeared in context of object-oriented programming. Inherit the properties and methods of a parent class or interface is a kind of contract. The component obtains a status, but it must implement the rules defined by predecessors. This idea is highly aligned with programming by contract. But it is barely outlined as a business model technique, because it is inherent to object-oriented paradigm.

These kinds of contracts are stronger, because components share similar *genetic code*. The advantage is more commitment between them, but the trade off is less separation of concerns.

Interface based

Interfaces appeared in object-oriented languages exactly to establish contracts rules. Objects must implement certain interfaces to participate in certain relationships. For example, all vehicle share the same operations: turn on, turn off, accelerate, brake, and so on (figure 4.8). Defining these common characteristics is a way to organize the knowledge.



```
public interface IVehicle
{
    void TurnOn();
    void TurnOff();
    void Accelerate(double velocityIncrease);
    void Brake(double velocityDecrease);
    void ChangeDirection(double angle);
    double GetVelocity();
}
```

Figure 4.8: Example of an interface in C#

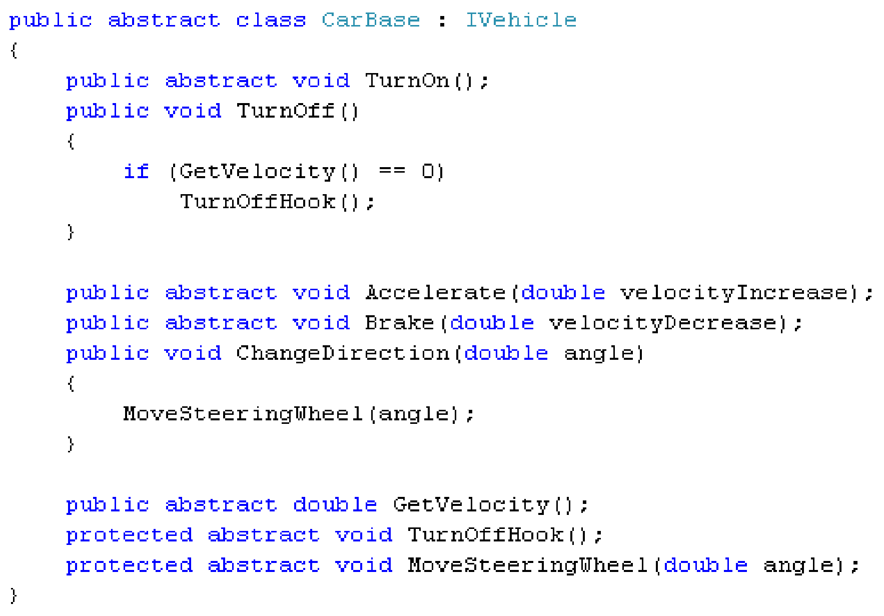
An interface resumes the common behavior and properties of a different set of objects. The process to establish an interface could be long, because time is needed to understand enough about problem domain. Therefore building an interface could be considered a modeling activity: it captures the knowledge about real objects, establishing connections between them.

Interface inheritance is supported by most object-oriented languages, including multiple inheritance. As some languages, e.g. C# and Java, forbid multiple class inheritance, interfaces are a way to establish multiple contracts by the same class.

Class based

Inheritance based on interfaces is limited to definition of methods and properties signatures. However it does not allow add some functionality or business constraints into the code. Therefore, class inheritance is used to maintain business logic shared by multiple types in one place, favoring code reusing.

The vehicle interface, presented in figure 4.8, let to create a base class for cars. Obviously there are specific restrictions. The car cannot be turned off while is moving. And the direction is changed using the steering wheel. Figure 4.9 shows a possible generic class for a car, where additional code defines the appropriate constraints.



```
public abstract class CarBase : IVehicle
{
    public abstract void TurnOn();
    public void TurnOff()
    {
        if (GetVelocity() == 0)
            TurnOffHook();
    }

    public abstract void Accelerate(double velocityIncrease);
    public abstract void Brake(double velocityDecrease);
    public void ChangeDirection(double angle)
    {
        MoveSteeringWheel(angle);
    }

    public abstract double GetVelocity();
    protected abstract void TurnOffHook();
    protected abstract void MoveSteeringWheel(double angle);
}
```

Figure 4.9: Example of a class in C#

4.2.7 Dynamic rules

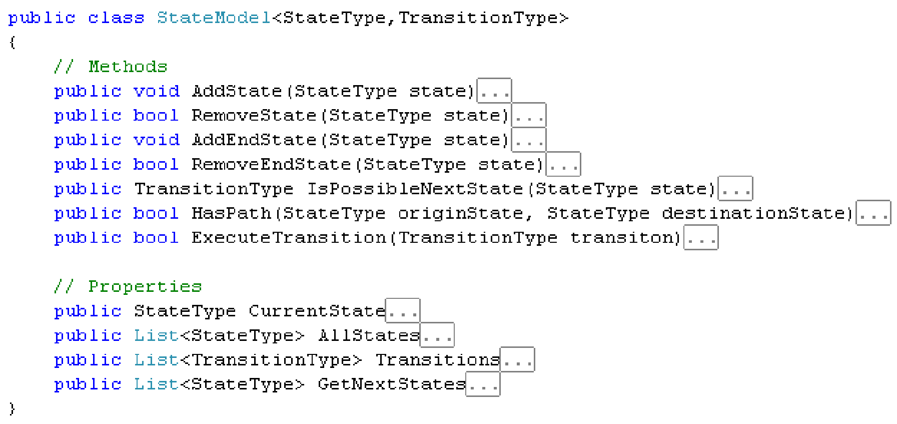
Dynamic rules are a technique to add and remove business rules during software execution. After all, *business rules change much more frequently than the rest of the application code* [95]. This is the idea behind business rule engines.

Implementing a business rule engine is a frightening task, but in some cases it is strictly necessary. For instance, imagine a customer relationship management software that processes clients accounts. Admit that premier clients are allowed to have a maximum debit of 5000 €, but now management has decided to change the limit to 2000 €. Change code is not practical, when business rules change too quickly. A small rule engine should be implemented to accommodate these types of problems. The rules of these engines can be defined using a new domain specific language [2].

Modeling a system with dynamic rules is preparing it for coming changes, which could emerge during implementation or afterwards. Together with adaptive object models[96] , dynamic business rules could make a system very extensible. However this flexibility has critical impacts on system performance, understandability and stability, which must be considered.

4.2.8 Generic programming

Business modeling is all about capturing the business logic without going into many details. As previous demonstrated it does not mean that coding activities should be avoided. Generic programming focuses on the class behavior, delaying the selection of primitive data types. For example, generic programming is suitable to define the behavior of a state finite machine, independently of data types that represents its states and transitions (figure 4.10).



```
public class StateModel<StateType,TransitionType>
{
    // Methods
    public void AddState(StateType state)...
    public bool RemoveState(StateType state)...
    public void AddEndState(StateType state)...
    public bool RemoveEndState(StateType state)...
    public TransitionType IsPossibleNextState(StateType state)...
    public bool HasPath(StateType originState, StateType destinationState)...
    public bool ExecuteTransition(TransitionType transiton)...

    // Properties
    public StateType CurrentState...
    public List<StateType> AllStates...
    public List<TransitionType> Transitions...
    public List<StateType> GetNextStates...
}
```

Figure 4.10: Example of generic programming in C#

Generic programming provides a good way to create a high-level design without defining data types. However generic programming syntax is not technological neutral and makes the code harder to understand and maintain, due to its higher-level of abstraction.

4.3 WARP solution

Every modeling technique has advantages and drawbacks. However most of them are neglected because they are costly and hard to synchronize with final code. The WARP solution proposes a mixture of previous techniques, connecting high-level design and source code.

The high cost of modeling techniques is tolerable, because to understand new domains is a complex and not automated activity. However the utilization of different techniques, tools and people to do analysis and development can raise this cost significantly.

High-level design tries to define the components that compose a system, including class model and component relations. Although technological restrictions play an important role in this level of analysis, people are afraid of testing their ideas in the target platform. They prefer to use all modeling techniques presented above, from visual models to formal methods, but never the development platform.

WARP solution proposes the utilization of a programming language to do business modeling. Like formal methods, WARP suggests the creation of an executable and testable business model, to explore components behavior and their dependencies. In this model the business logic and rules are defined, in a high-level way. Techniques as programming by contract would be useful and then implementation stage reuses this business logic, using inheritance as contract or object composition.

WARP also advocates the separation between business and implementation code: logical design and physical design (figure 4.11). This division eases the reusing of business code and changing the implementation part. In fact, Martin Fowler defends this approach with the *Separated Interfaces* architecture pattern [97].

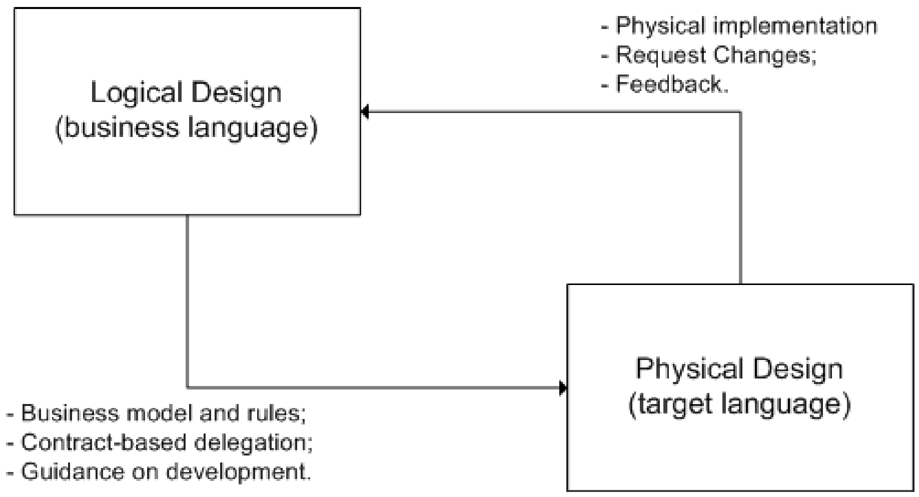

Logical Design
(business language)
- Physical implementation
- Request Changes;
- Feedback.
Physical Design
(target language)
- Business model and rules;
- Contract-based delegation;
- Guidance on development.

Figure 4.11: Logical design and physical design on WARP

In the object-oriented paradigm, logical design is composed by business classes. They define proper properties and methods, as well some logic and constraints of the business. Physical design is composed by concrete classes, which deal with more technical concerns as object persistence, security or user interfaces. Usually a concrete class inherits a business class, establishing a contract between them. Ideally the interface exposed by the business model would be the same interface exposed by implementation code.

The synchronization between two parts occurs as in code development, using source repositories and diff tools. At any moment, developers can request changes on high-level design. Because it is represented as source code, they can submit a patch through source repository, which analysts would accept or not. Similarly analysts can change high-level design and evaluate immediately the consequences on implementation part.

Reverse engineering techniques allow generating visual models from executable business model, which help understanding the system. But the source code is the primary representation of business models. Visual Studio Class Designer shows classes and their relations in a visual way, through analysis of source code. Maybe in the future, it will be possible generate more complex charts, as work-flow diagrams, from the source code.

4.3.1 Modeling language

The language used to do modeling should be as much similar as possible with target language. The utilization of two different languages, one for logical design and other to physical design, creates difficulties on communication. It is like writing the requirements in Portuguese and then they are given to French people to do implementation.

Nevertheless two different languages is a tempting approach: the logical language is more flexible and focuses in domain rules, while physical language takes care of all implementation details. In this scenario, there are two interesting hypothesis:

- A logical language based on OCL syntax;
- An internal domain specific language.

However these approaches imply staff training, both for analysts and developers. They need time to change their mindset and practice to consolidate their knowledge in new languages.

Therefore WARP proposes the utilization of the target programming language, e.g. C# or Java, to do business modeling. The language syntax should be restricted, to maintain the focus on modeling activities and to avoid going into too many implementation details. Obviously developers understand the language and analysts can easily learn it, due to its simplicity. Moreover, it is not exceptional that business analysts have some technological background. Many of them have done software development in the past, or at least they had one programming course in college.

The utilization of the target programming language supports an important advantage: the business model can be easily reused during implementation, without any complex mapping process. In addition developers can criticize and request changes on business model in an easier way. That is, it simplifies the process of synchronization between business models and source code.

4.3.2 Testability

One of strengths of WARP solution is testability. In comparison with other techniques (figure 4.12), WARP offers a way to do analysis and design, in a formal and testable way. That is to say, both logical and physical designs are formal and testable.

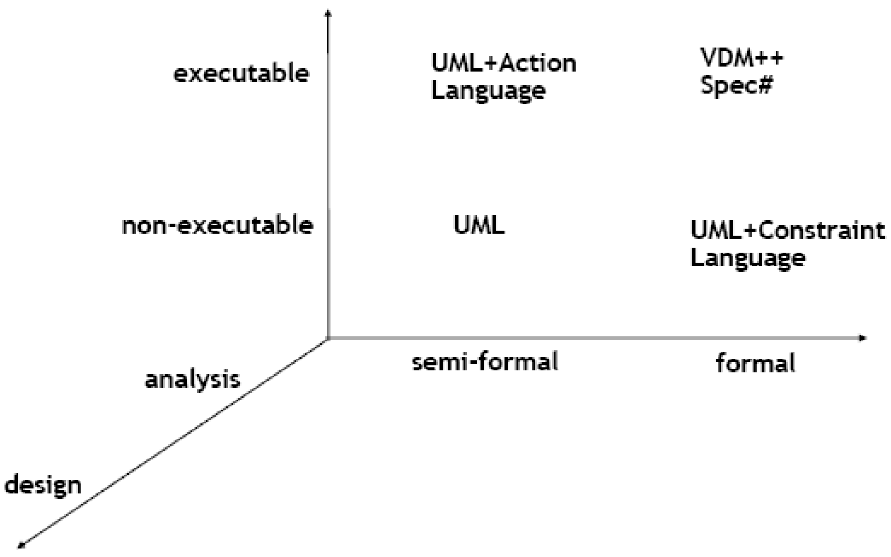


Figure 4.12: Comparison of some business modeling techniques [82]

The ability to write tests to both parts, open a door to create tests that validate the business classes: logical tests. Afterward, this type of tests can also verify the implementation part, by running over concrete classes. Using Continuous Integration techniques, developers and analysts can test easily the consequences of their changes.

Traditional programming languages have several tools to do automated testing, namely unit tests frameworks, which are easily adapted to implement logical tests.

Logical tests verify classes' behavior, accordingly business rules. They are written by analysts with two possible purposes: ensure the validity of achieved business model or ensure the validity of concrete classes. The former pretends ensure the validity of analyst work, while second look for errors on developer efforts.

Although logical tests can be considered a type of integration tests, the boundary between them is quite blurred. A logical test can test a small functionality, so it would be named a unit test. But it can test a business rule, and therefore it is a logical test. In the same way, developers can write integration tests that are not logical tests. Moreover not only analysts but also developers can write logical tests. Strictly, all tests that verify business logic in an agnostic technical way are logical tests.

When logical tests are written, business classes are not implemented and their behavior is quite undefined. This uncertainty decreases the testability options. Mock frameworks [98] can address this problem, providing a way to build a fake world where the business model can run. Moreover they are very practical in testing abstract classes.

Connected tests

WARP approach advocates the importance of separate logical and physical parts in different components, e.g. different assemblies. This division guarantees a real separation between business and concrete classes. In the same way, and accordingly with good practices on software development, tests are written in independent assemblies (figure 4.13).

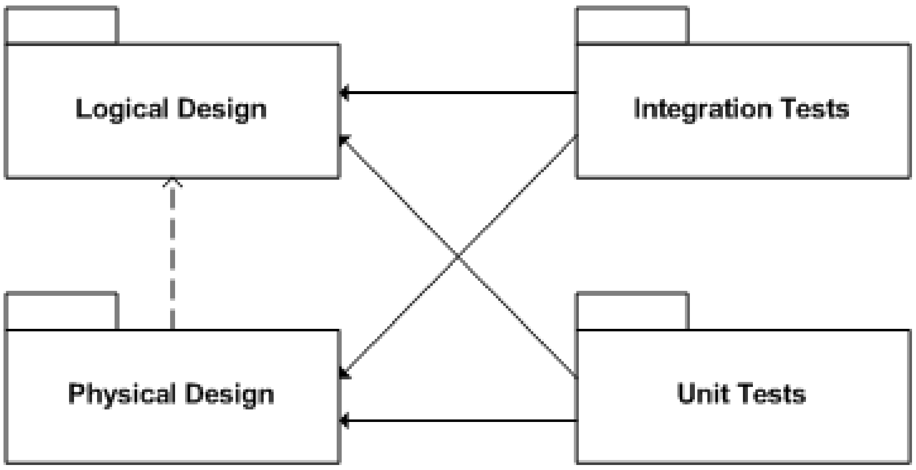
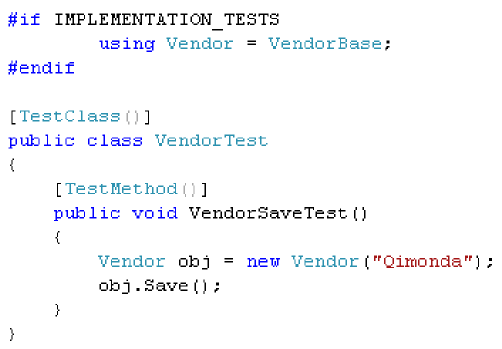


Figure 4.13: Testing on logical and physical designs

An interesting aspect of WARP is that analysts contribute with tests for the final software. These are logical tests, which could be integration or unit tests. In the same way, developers write their own tests, to check the validity of their work. In this situation tests can verify both logical and physical designs. That is, they are connected.

A decision arise in this scenario: which tests should test physical part, and which tests should test logical part. However some tests are useful to test both parts. But that does not mean write two different tests. The tests could be reused in both parts.

For example, using Visual Studio and C#, it is possible to change the used classes with the preprocessor directives #if/#endif (figure 4.14). Therefore, a simple change on project options allows the choice between business and concrete classes. The change of classes would not produce big problems, as business and concrete classes have similar public interfaces.



```
#if IMPLEMENTATION_TESTS
        using Vendor = VendorBase;
#endif

[TestClass()]
public class VendorTest
{
    [TestMethod()]
    public void VendorSaveTest()
    {
        Vendor obj = new Vendor("Qimonda");
        obj.Save();
    }
}
```

Figure 4.14: Using C# preprocessor directives to change used classes

4.4 Discussion

Business modeling is a hard effort, and it will continue to be. The number of techniques is large and seems natural that people do not know what adopt. WARP solution was a compromise between previous presented techniques and industry constraints, helping to bring part or completely some techniques into software development process. However WARP solution is a complement to existent techniques, and it will not pretend substitute them entirely.

Chapter 5

Object-relational mapping

Motto: Objects ⇒ Tables

Software executes instructions on computers in the context of threads or processes. Applications manipulate data on main memory, which is quicker but also volatile and it disappears after to finish the process. The only way to guarantee is save data on permanent storage supports, as hard disks or magnetic tapes. Therefore, developers must care of persistence of data.

Most of programming frameworks provide low-level methods of persistence, as saving data in text files or using object serialization. However they are not reliable, being inadequate to most business applications [99]. Data access is slow and complicate, and they lack support for security, replication, data consistency and concurrency access. Database management systems (DBMS) came out to address this problem. They were widely adopted by software industry, due to their capacity to manage persistent data. Oracle, Microsoft SQL Server and MySQL are examples of well-known databases.

Database systems are based on relational algebra, organizing the information in logical tables, views and stored procedures. They expose their interface through a domain specific language: Structured Query Language (SQL). This interface encapsulates and hides the internals of databases, which ensure the correct processing and saving of data.

Although databases have proved their utility, integrate them with software applications is not easy. Access and save persistent data are common operations, leading to the creation of many SQL code scattered in the system. Layered architectures have tried to solve this issue, isolating database accesses on persistence layers. But maintain this component is a complex and prolonged mission, taking time and focus of developers in the problem domain.

This concern becomes more challenging on the context of object-oriented programming. Coders represent the business model in form of objects, writing classes.

Some of these objects are persistent, that is, they exist between different process instances. Save and retrieve them on databases appear to be natural. However mapping between classes and tables is not easy, as they have different characteristics. For example, data types available in the two systems are different. Relational schemas do not support inheritance, and associations are static and globally defined. Object identity is usually a reference. In relational schemas, identity is ensured through primary keys. After all, object-oriented programming focuses on behavior, while relational schemas centers on structure. This difficulty in relate classes and tables is known as object-relational impedance mismatch[100].

Object-relational mapping (ORM) software try to mediate the interaction between databases and object-oriented programming. Object-relational mappers provide a way to map classes to tables, defining the points where they are related, and hiding the specificities of relational schemas. For instance, a integer property in a class could be the primary key in the table. Therefore, developers only do operations over objects, which are reflected in the persistence layer. The data loaded from database is returned as objects and a simple call to a method can directly delete table tuples. In fact, object-relational mappers provide intermediary and object-oriented querying languages, thus the developer does not need to known SQL.

Additionally most of ORM tools have support for transactions, caching, concurrent control, class associations, querying, dirty checking, lazy loading and pagination of results. A more complete list of features provided by object-relational mappers could be found in [101].

5.1 Data access layer and ORM

Confusion data access layer and object-relational mapping concepts is a common mistake. Data access layer is a set of components that provides communication between classes and relational schemas. It maps data types of original programming languages to database types, and it provides methods to access and save data. That is to say, it abstracts the database for classes use.

A data access layer can do object-relational mapping, but that is not mandatory. ADO.NET and Enterprise Library Data Access Application Block provide access to data, but they are not object-relational mapping software, because do not help to map database entities directly to classes. Real object-relational mapping software avoids the knowledge about SQL language and the underlying relational schema. The developer can work with persistent objects as with normal objects.

5.2 Advantages and disadvantages

Object-relational mapping techniques have several advantages and disadvantages, which make them a controversial topic on software development. The common referenced advantages are:

- Rising of developer's productivity;
- Decrease of costs in maintaining the data access layer;
- Abstraction of database operations;
- A more maintainable system, with less lines of code;
- Interchangeability of underlying database.

Nevertheless there are some drawbacks generally well accepted:

- The effort to learn the tool could be large;
- The developer is not aware of performance consequences, because he does not known how database accesses occur;
- Batching operations usually perform better inside database environments, for example using stored procedures;
- Recent technique, in ongoing development;
- Lack of support by software industry.

A common criticism about object-relational mappers is performance degradation. Critics said that manual persistence could perform better rather than automated persistence through an ORM. However this is a false question, as programming assembly code could perform better than with higher level languages [99]. That is, not always more abstraction means lower performance. In fact, with some features included as caching and unit of work pattern, object-relational mapping software can really improve performance.

The lack of support by software industry, especially large vendors like Microsoft and Oracle, was pointed as a drawback of this technique. In fact, many of mature object-relational mappers arose in the Open Source community, like Hibernate and NHibernate. Information technology people is usually afraid to accept the risk of choosing a software without support. However this technique has been adopted by large vendors, as Microsoft and its LINQ to Entities project [102].

5.3 Applicability

Although object-relational mapping is a useful programming technique, there are cases where they are not applicable.

In most of new object-oriented software, object-relational mapping would be recommended. That is true especially for applications that need data persistence and have a richer business model. However, in data intensive software, which do simple data accesses and saves, their utilization has little benefits. Similarly, the benefit for small-size applications should be evaluated, due to the effort to learn new technologies.

Another situation where object-relational mapping is not recommended is in the software that has the business logic in the database, using stored procedures or object-relational databases. Most of the ORM tools do not support these techniques, making their mapping painful, usually requiring cross-cutting calls.

The adoption of an ORM in a legacy software depends on its quality. In an ideal situation, if software was developed using object-oriented design, with a good level of encapsulation in the database access, it could easily move for a ORM. But these ideal situations are quite rare.

5.4 Types of ORMs

Object-relational mapping software could be classified according their working principles. One type of classification describes the way how the relation is established:

- Top-down: Classes $\Rightarrow$ Tables;
- Bottom-up: Tables $\Rightarrow$ Classes;
- Bidirectional: Classes $\Leftrightarrow$ Tables.

In the top-down approach, the developer writes classes, and then the ORM produces the relational schema. In the bottom-up type, the developer scripts relational schema, and then classes are generated. And in the bidirectional, the developer writes both classes and tables, and then uses the object-relational tool to relate each other.

Another way to classify object-relational mapping tools is to analyze how they provide their functionality. Code generators and libraries are the common approaches. Code generators analyze the primary representation and produce respective code in the other representation. Libraries expose an interface to define relations between the two representations.

5.5 Object-relational mapping tools

At the moment of start using object-relational mapping tools, the developer must choose between to use an existing framework or to build a new one. The temptation to build custom object-relational mapping software could be high. Most of current tools are under development, missing features or stability. However, implementing a functional and complete object-relational mapping tool is a hard task. Barely a company has the necessity to implement one. After all, the productivity gains with their adoption would be lost on producing that new component.

The range of available object-relational mappers is large, with different working models, advantages and drawbacks. A presentation of six ORM tools for .NET development environment is presented below: SubSonic, NHibernate, Castle ActiveRecord, LINQ to Entities, EUSS and LLBLGen.

5.5.1 SubSonic

SubSonic [103] generates the data access layer from relational schema (figure 5.1). It could be considered a bottom-up object-relational mapper, as it generates objects from tables. The data access layer uses SubSonic classes and ADO.NET providers. If the developer wants the Enterprise Library provider could be used. The business logic could be implemented extending data access objects, using partial classes or object composition.

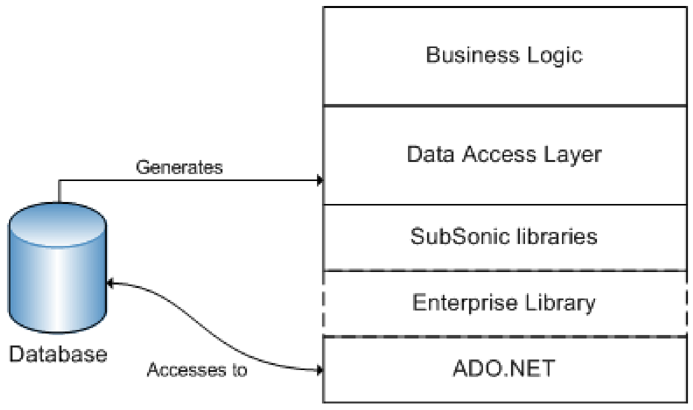


Figure 5.1: SubSonic working model

SubSonic is open source and it is incredibly simple. Setting up the application is easy and the configuration requires few lines of XML. It does not introduce many layers on the software, avoiding big performance penalties. The code of data access layer could be viewed, to understand its behavior or to tune its performance. Nowadays it supports Microsoft SQL Server 2000/2005, MySQL and Oracle. Appendix A contains a review of SubSonic (see section A.1.10).

However its bottom-up approach does not make much sense. The developer must define first the business model, and then implement manually the relational schema to generate the data access layer.

5.5.2 LLBLGen

LLBLGen [104] is object-relational mapper that generates the data access layer and part of business objects tier from an existent relational schema. The working model is very similar with SubSonic. The generated code is available to be analyzed and tuned. However, in contrast with SubSonic, stored procedures and database views could be easily accessed through simple method calls, and it has a visual interface to configure and execute the code generation.

LLBLGen is a mature and well-tested framework, with five years on the market. It supports the following databases: Microsoft SQL Server, Oracle, PostgreSQL, FireBird, Microsoft Access, IBM DB2 UDB, MySQL and SyBase. However it is proprietary software and the license costs are not insignificant.

5.5.3 NHibernate

NHibernate [89] is an open source port of Hibernate, a object-relational mapper for Java. It acts as a intermediary between classes and the relational schema (figure 5.2). The developer writes XML files that define how classes are mapped to existent tables, specifying the properties to map and their respective data types.

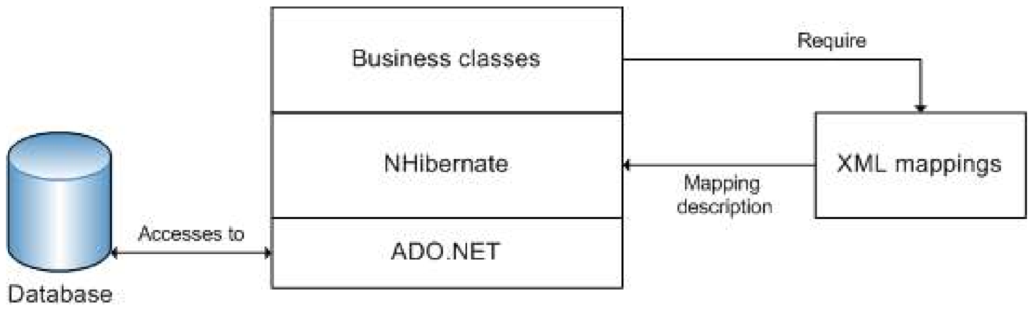


Figure 5.2: NHibernate working model

The maturity and completeness of NHibernate make it an excellent tool. The separation between business classes and mapping files provides a better maintainability. Similarly, bidirectional approach eases both the initial creation and future changes.

The control over mapping is bigger, but it demands more effort to do. Moreover the excess of XML code, which is not easily debuggable, could burden developer's work. For more information, consult the tool review of NHibernate in the appendices (section A.1.9).

NHibernate supports following database systems: Microsoft SQL Server 2000/2005, Oracle, Microsoft Access, Firebird, PostgreSQL, DB2 UDB, MySQL and SQLLite [105].

5.5.4 Castle ActiveRecord

Castle ActiveRecord [106] implements the Active Record pattern[12] on top of NHibernate. The developer writes the business classes with attributes that define the mapping to the relational schema. For example, to say that property name does not allow duplicate items, an attribute signs that it is a unique property. Castle ActiveRecord generates the NHibernate XML mappings and respective relational schema (figure 5.3). A tool review of Castle ActiveRecord is available as appendix (section A.1.8)

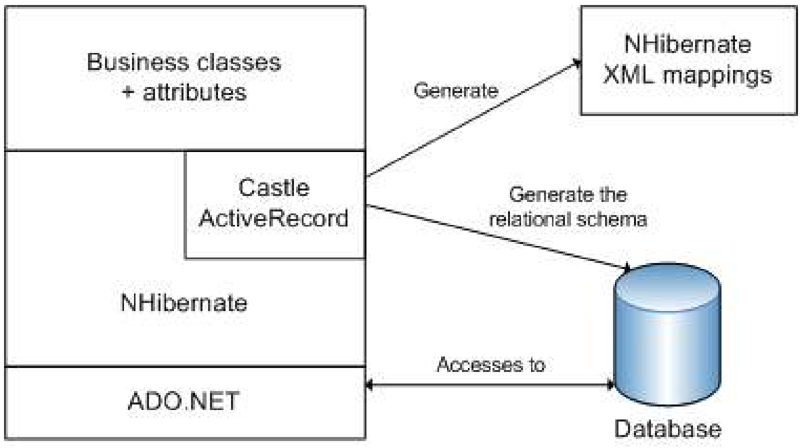


Figure 5.3: NHibernate working model

The developer work with objects transparently, using the exposed interface without know nothing about relational schema. Castle ActiveRecord provides a set of classes for CRUD operations and data validations. For more complex queries, the developer would use NHibernate. As Castle ActiveRecord is built on top of NHibernate, it supports the same collection of databases.

The top-down process is very natural: write business classes and then generate automatically underlying layers. It seems like a *coding and compiling* process. And it frees developers of write complicate XML files and SQL code. Although the mapping had woven with code, imposing design constraints, Castle ActiveRecord really accelerate the implementation of persistence concerns. Note, however, that the top-down process is not mandatory: it could define mappings for existent relational schemas.

5.5.5 ADO.NET Entity Framework

ADO.NET Entity Framework [107] is a framework from Microsoft to solve the object-relational impedance mismatch. The developer defines an entity model, through XML files, which are a object representation of relational schema. And then he could use entities in the code as they are normal types (figure 5.4). Their strong points are ADO.NET Entity Designer and LINQ for Entities.

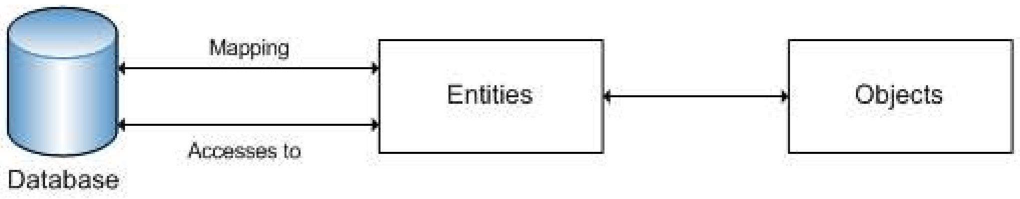


Figure 5.4: Simplified ADO.NET Entity Framework working model

ADO.NET Entity Designer is a graphical user interface to manage entity models, with a level of completeness that is uncommon in object-relational mapping software. The entities are created in a visual way, relating them with existent tables and stored procedures. Next-and-finish wizards are common, like automatic generation of entities from relational schema and interfaces to update entities.

Language Integrated Query (LINQ) allows to write queries over different types of data using the same syntax, directly from C# 3.0 or Visual Basic 9.0 code. It was released as part of .NET Framework 3.5, and its roadmap foreseen LINQ to Entities. Its major advantage is that queries are compilable and Intellisense could help in their development. By now there are support for querying objects, XML, SQL and ADO.NET DataSets. Nevertheless anyone could implement a provider to query a specific item.

Microsoft announced support only to Microsoft SQL Server, but some companies declared their intentions to implement providers for others databases, like Oracle, PostgreSQL and MySQL. For more details see the tool review of ADO.NET Entity Framework on section A.1.3.

The framework follows the Microsoft policy of have good visual interfaces to help the developer. However, in contrast with other tools, the framework is quite immature, without any stable release and with lack of documentation. Therefore the risk of adopt it now is high, but certainly it would be a excellent software engineering tool.

5.5.6 EUSS

Evaluant Universal Storage Services (Euss) [108] is a open source persistence infrastructure promoted by Evaluant. Its concepts are very similar with NHibernate: the developer writes XML files to map objects to the persistence items. However, its concept of persistence engine is notably. Developers define entities, relationships and attributes that could be mapped to any support, according chosen persistence engine. That is to say, the mapping could occur not only with databases, but also with memory databases or XML files.

EUSS supports Microsoft SQL Server, Microsoft Access, Oracle, MySQL and SQLite.

5.6 Discussion

Object-relational mapping increases de facto developer's productivity. The effort and cost to build and maintain persistence layer is considerably reduced. Bottom-up approaches seem very good for legacy software, which already have a relational schema. On the contrary top-down tools appear to be very natural in the creation of new systems, creating first the business model and then generating the schema. The best way would be the bidirectional approach, which permits mapping between two existent sides, combined with code generative techniques of bottom-up and top-down.

The choice of the best object-relational mapping software is questionable. Moreover those tools are evolving quickly. But, by now, NHibernate and Castle ActiveRecord together could be considered the best framework to do object-relational mapping in .NET environments. They are open source and mature projects. Both are well-documented and widely used. Castle ActiveRecord avoids the complicate mapping of NHibernate with XML files, as well as a set of useful development features.

Chapter 6

Prototyping the user interface

Motto: Objects $\Rightarrow$ User interfaces

Prototyping is a widely software engineering technique, as a way of communication between different stakeholders. One of the difficulties of people enrolled on software projects is to talk about abstract things. Textual project vision, class diagrams, work-flow charts, domain specific languages are examples of conceptual descriptions that are hard to understand, especially for those without software engineering background. The customers and analysts understand much better a system when they see the user interface. Final users easily provide feedback about a tangible thing, finding defects or improvements. Their feedback would correct problems not only in the user interface but also in the whole system.

Prototyping the user interface has costs, especially if it is a graphical user interface (GUI), requiring the definition of application look and feel and components layout. However the saving costs of founding defects prematurely definitely supersede prototyping expenditures.

Nevertheless two techniques were emerged to do prototyping: visualization and scaffoling. The first focuses on the creation of images of user interface, while second produces automatically an interface from existent models.

6.1 Visualization techniques

Visualization techniques emerged since first days, to sketch the user interface and to define how the visual components are displayed. The simplest way is to draw in paper the application layout. The utilization of a painting program could make more professional the aspect of drawings. But they do not capture the flow of activities, neither the system behavior.

The option is to use existent development environments. In the case of web applications, a simple HTML editor could be used to do that. Similarly Visual

Studio enables prototyping windows forms or web applications, by dragging and drop controls and coding some simple functionality. But that demands a considerable amount of time and technical knowledge.

Interestingly, model-driven development was adapted to the creation of user interfaces. Himalia (see A.1.2) allows to model user interface visually and then it generates the code of user interfaces. However the project is in a preliminary stage and similar tools are not known.

Fortunately some simpler applications appeared to let business analysts build prototypes of users interfaces. For example, iRise [109] is a proprietary simulation tool that works as a CAD system for software, through inserting text and drag and drop controls. The flow of activities could be modeled through work flow diagrams, as well some business rules, without any code involved. The aspect of application could be configurable in a similar way of what you see what you get (WYSWYG) editors for web applications, allowing the utilization of templates and master pages. The prototype could be packaged and executed in client machines, using a plug-in. Figure 6.1 shows a screenshot of IRise Studio, the application to develop the prototypes.

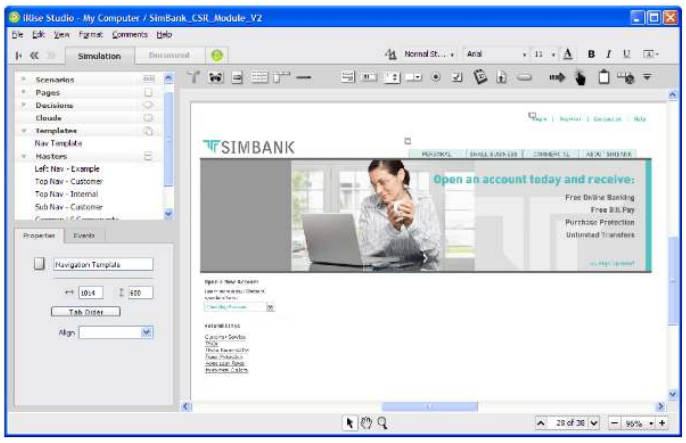


Figure 6.1: Screenshot of IRise Studio

Visualization techniques help people in the understanding of software requirements, without requiring too much technical background. Analysts could build prototypes in the earlier stages of project, even before create the development team. These prototypes could be a reference for the following phases. Similarly, they provide a way to evaluate system usability before build it.

6.2 Scaffolding techniques

Establishing the flow of user's activities in the system is a modeling activity. Unfortunately, most of the time, prototyping starts without any knowledge about business logic. Additionally the absence of an underlying structure, namely an executable business model, complicates this effort. That leads to a undesirable waste: throw-away prototypes.

Scaffolding techniques allow create user interface prototypes based on underlying structure, creating a code skeleton to the developer. These prototypes are useful for discussions with final users and to insert testing data by developers. These techniques were popularized by Ruby on Rails framework and they have been ported to other platforms [110].

Scaffolding provides the automatic generation of user interfaces from existent models, usually from relational schema or from a set of classes. At least, it builds an interface for the basic CRUD operations (Create, Read, Update and Delete). Commonly it detects the relations between entities and allows define them, e.g. an interface to choose a existent client to fulfill a order.

6.2.1 Types of scaffolding

Scaffolding techniques could be categorized according their working model and the original model.

Following the Ruby on Rails denomination, there are two working models: dynamic scaffolding and scaffolding generation. The first type creates at runtime the user interface based on configuration points and on the original model. The scaffold is based on a monolithic piece and it is hardly modifiable. In the scaffolding generation, the user interface is generated as code, allowing future changes.

Another classification of scaffolding defines the original model from it produces the user interface: relational schema or classes. Although creating a user interface from tables and stored procedures would be interesting, producing from classes could be more powerful. Relational schema defines the data structure, specifying data types and relations. However it does not capture objects behavior.

For example, in a hypothetical *Employee* class there is a method to calculate the salary, based on years of experience and on job position. Similarly when the company admits a new employee, a new account is created in the internal information system. A user interface based on classes would be much richer. But doing that is also much more challenging. Objects are dynamic entities, which would disappear after program execution.

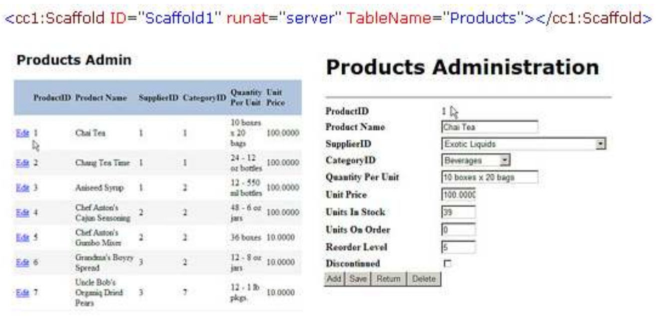
Moreover the degree of freedom on object programming paradigm complicates generation techniques. The developer could use inheritance, object composition, collections, generics, recursive methods, and so on. Likewise methods signatures are not enough descriptive to produce user interfaces. Although they are relatively clear about user inputs, describe their consequences is more tricky. For example, the output of a method that returns a boolean type can represent a answer of the system or a failed operation.

6.2.2 Scaffolding tools

In the case of Ruby on Rails, the scaffolding support is quite mature. However, to other platforms, namely for .NET development, there is a scarcity of tools. Moreover most of them are experimental and in ongoing development. This section presents three tools that have some kind of scaffolding techniques for web applications: SubSonic, Castle MonoRail and ASP.NET Dynamic Data.

SubSonic

SubSonic is an object-relational mapping software that generates the data access layer from relational schema (see section 5.5.1). It supports dynamic scaffolding through a user control, a user interface component for ASP.NET framework. That user control allows to create, read, update and delete items saved on the underlying database through a web interface. The 6.2 shows a sample of ASP.NET code to call the Scaffold control, and respective produced user interface.



```
<cc1:Scaffold ID="Scaffold1" runat="server" TableName="Products"></cc1:Scaffold>
```

Products Admin

	ProductID	Product Name	SupplierID	CategoryID	Quantity Per Unit	Unit Price
Edit	1	Chai Tea	1	1	10 boxes x 20 bags	100.0000
Edit	2	Chang Tea Time	1	1	24 - 12 oz bottles	100.0000
Edit	3	Aniseed Syrup	1	2	12 - 550 ml bottles	100.0000
Edit	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	100.0000
Edit	5	Chef Anton's Gumbo Mixer	2	2	36 boxes	10.0000
Edit	6	Grandma's Boyry Spread	3	2	12 - 8 oz jars	10.0000
Edit	7	Uncle Bob's Organiq Dried Pears	3	7	12 - 1 lb pkgs.	10.0000

Products Administration

ProductID	1
Product Name	Chai Tea
SupplierID	Exotic Liquids
CategoryID	Beverages
Quantity Per Unit	10 boxes x 20 bags
Unit Price	100.000
Units In Stock	39
Units On Order	0
Reorder Level	5
Discontinued	☐

Figure 6.2: Sample of utilization of SubSonic scaffolding [111]

Although the available user control is very simple and easy to configure, it is a monolithic component, hardly changeable. Therefore it is perfect to fulfill data on the database and to build throwaway prototypes, but it is not recommendable to create the initial skeleton of user interface.

Castle MonoRail

Castle MonoRail [112] is an open-source Model-View-Controller (MVC) framework similar to Ruby on Rails. This type of frameworks aims separation of concerns, separating models, views and controllers. The model represents a business entity. The view defines how to render the page. And the controller processes and responds to events, typically user actions, invoking models. MVC frameworks are emerging as a standard for web applications in software industry. In fact, Microsoft have released ASP.NET MVC [113], which is very similar to Castle MonoRail.

Castle MonoRail works over ASP.NET and could run with a small configuration in Internet Information Services (IIS). It provides many interesting techniques like data binding from form inputs to objects, or caching. Figure 6.3 explains briefly how it works: a client send a request for a page, the IIS recognizes that need to call the MonoRail handler which creates a controller and invoke a method, accordingly the requested URL. This method could interact with models and pass data to the view. Finally the view is processed and the result is sent back to IIS, which forward the response to the client.

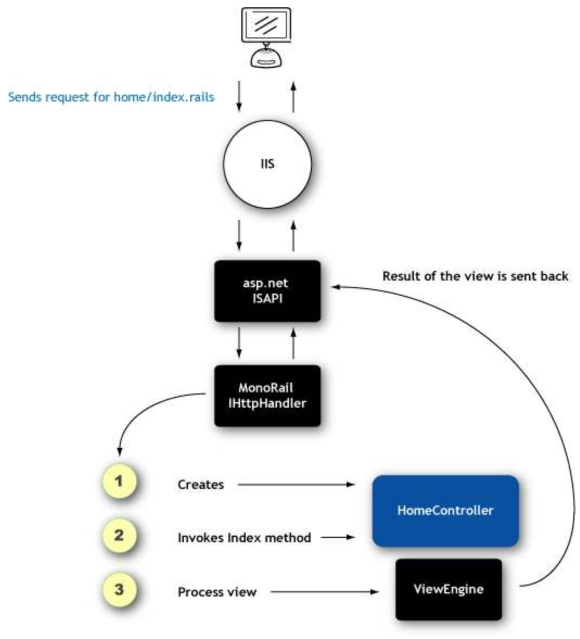


Figure 6.3: How Castle MonoRail works [114]

Primarily Castle MonoRail have supported dynamic scaffolding. Using models produced with Castle ActiveRecord, an object-relational mapping software (section 5.5.4), it generates at runtime the controllers and their views to do the basic CRUD operations. Figure 6.4 shows how to create a scaffold controller for Equipment class, while figure 6.5 shows the result.

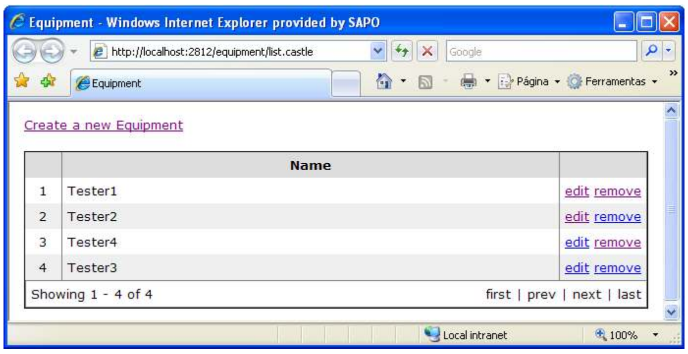
```
using Castle.MonoRail.Framework;
using MonoRailApp.Models;

namespace MonoRailApp.Controllers
{
    [Scaffolding(typeof(Equipment))]
    public class EquipmentController : BaseController
    {

    }
}
```

Figure 6.4: Code to create a scaffold for Equipment class

As dynamic scaffolding is hardly extensible, there is an effort of Castle MonoRail community to support scaffolding generation, through Generator project started by Marc-Andre Cournoyer [115].


Equipment - Windows Internet Explorer provided by SAPO
http://localhost:2812/equipment/list.castle
Google
Equipment
Página
Ferramentas
Local intranet
100%

Create a new Equipment

	Name	
1	Tester1	edit remove
2	Tester2	edit remove
3	Tester4	edit remove
4	Tester3	edit remove

Showing 1 - 4 of 4 first | prev | next | last

Figure 6.5: Screenshot of scaffold generated by Castle MonoRail

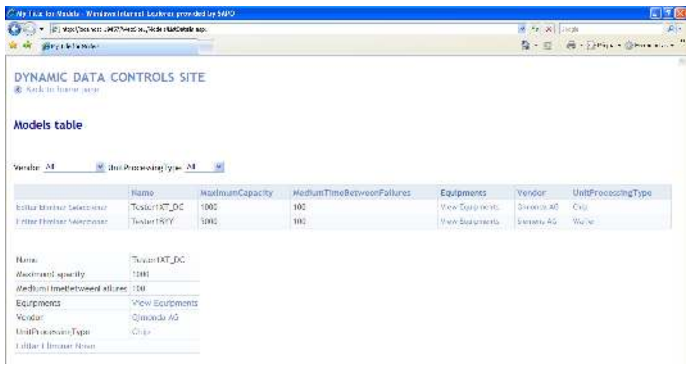
ASP.NET Dynamic Data

ASP.NET Dynamic Data [116] is a scaffolding framework, included in the .NET Framework 3.5 Service Pack 1 Beta [117]. The interest of Microsoft in scaffolding techniques reflects how software industry is receptive for this concept. Although ASP.NET Dynamic Data is under development and only with a beta release, the actual functionality is very promising.

ASP.NET Dynamic Data includes an object-relational mapping designer that generates classes from an existent relational schema. That data model could be extended using partial classes and attributes, e.g. to perform data validation. The graphical user interface is generated based on the model and a set of customizable templates. It is a dynamic scaffolding, but quite customizable.

Figure 6.6 shows a user interface created with ASP.NET Dynamic Data and Visual Web Developer 2008 Express Edition.

ASP.NET Dynamic Data is very simple and tightly integrated with Visual Studio development environment. Several wizards are available and the documentation has



DYNAMIC DATA CONTROLS SITE

[illegible]

Models table

Vendor: All · UnitProcessingType: All

	Name	MaximumCapacity	MediumTimeBetweenFailures	Equipments	Vendor	UnitProcessingType
[illegible]	TesterIXT_DC	1000	100	View Equipments	[illegible] AG	Chip
[illegible]	Tester [illegible]	1000	100	View Equipments	[illegible] AG	Wafer

Name	TesterIXT_DC
MaximumCapacity	1000
MediumTimeBetweenFailures	100
Equipments	View Equipments
Vendor	Qimonda AG
UnitProcessingType	Chip
[illegible]	

Figure 6.6: Screenshot of scaffold generated by ASP.NET Dynamic Data

been growing during the last year. Microsoft partners have been developing components based on ASP.NET Dynamic Data. For example, Infragistics, a specialized enterprise on developing components for presentation layers, announced plans to support ASP.NET Dynamic Data in their controls [116]. Therefore this technology probably will be highly supported by software industry.

6.3 Discussion

Prototyping the user interface is an expensive activity. But the quickly feedback given by it avoids many misunderstandings and defects. Late repairing has an exponential cost, much greater than prototyping costs. Unluckily, by now most of prototypes are thrown away, because they have extensibility problems.

Visualization techniques that have not coding activities could be cheaper, but it is impossible reuse them in later stages. And if they have, they are expensive and bad designed: not prepared to change, neither to accommodate the technological constraints.

Scaffolding techniques are a way to create more real prototypes, according existent models. Dynamic scaffolding is practical but it is not extensible. Scaffolding generation would be the more convenient approach, as it produces changeable code. But it seriously constraints the system design and the range of solutions is small and under development. Moreover the current available tools are unstable and have several defects.

The drawback of current scaffolding techniques is that most of them are based on structure rather than on behavior. That is to say, they are mainly based on the relational schema, which does not capture all business logic. A more convenient approach would be scaffolding based on classes, but there are some challenges to solve as showed in this chapter.

Nevertheless scaffolding is a singular tool, as it is quite inexpensive and quick technique. And maybe in the near future the technology allows more component reusing.

Part III

Practical work

Chapter 7

Business#

The WARP solution is a theoretical description of an approach to do business modeling. However lacking experimentation, it would be an unsupported idea, without any proof of concept. Therefore Business# emerged as a framework to support business modeling and subsequent development according WARP vision. Remember that WARP advocates the utilization of a traditional programming language, e.g. Java or C#, to do business modeling. However this high-level design should only capture business logic, without any implementation detail. This approach allows a easier synchronization between the model and the code.

Business# provides a high-level language to do business modeling, accompanied by a set of tools to assist analyst work. It would be compatible with Microsoft .NET Framework, allowing to build applications in this platform afterwards.

The intent of Business# is to become the basis infrastructure to develop an application according WARP way. It would be the core of application, while business model defines the behavior and functionality of the system. The implementation components expose the software interface, based on underlying layers (figure 7.1). Ideally the public methods are only defined in business classes but there are times where that is impractical. However, as rule of thumb, the added logic in the implementation should be as minimal as possible.

Business classes inherit from BaseObject. Similarly implementation classes inherit business classes (figure 7.2).

Business classes define components and their relationships. Some business rules could be defined, by implementing some methods and using programming by contract. Similarly they could use template methods to delegate into subclasses some behavior. The implementation classes use business logic using inheritance, overriding properties and methods, and using object composition.

One of the great properties of Business# is lack of ambiguity. Its language is formal and clearly defined. Business# models are represented as code, avoiding textual

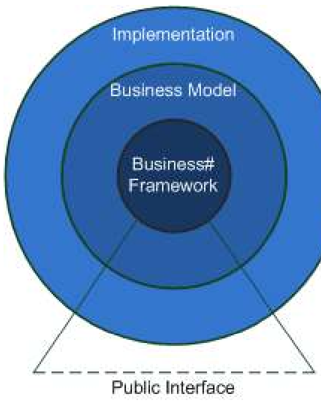


Figure 7.1: A layered architecture using Business#

and visual descriptions that are differently understood by people. The communication through code eliminates many misunderstandings. Additionally modeling with code means easier change management. Source control and diff tools could be user to manage the evolution of business code. The developer can request an inconsistency in business model through submitting a patch in the code repository.

7.1 Objectives

The main goal of Business# is to prove the utility of WARP solution for business modeling, as well as its applicability. The challenge of WARP is to integrate high-level design with low-level design and at the same time provide a testing framework. Therefore the Business# would be composed by:

- A simple, high-level and formal language;
- Useful classes to do modeling;
- Guidance on coding;
- Automatic generation of visual models from code;
- A testing framework.

7.2 Language Syntax

As explained in the WARP solution (section 4.3), traditional programming languages are an adequate syntax to do business modeling. As they are used in later stages of development, they allow the integration between business code and implementation code.

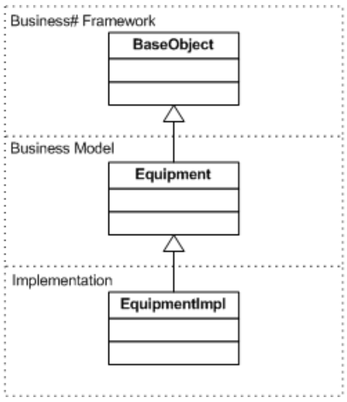


Figure 7.2: A typical class hierarchy using Business#

In the context of .NET platform, the most famous language is C#. It is a mature language, well documented and it is widely used. This syntax is formal and easily understandable by people with experience in object-oriented languages, as Java and C++. However Business# classes must be simple, without bundled implementation details. Therefore a restricted C# language emerges as a natural choice to do modeling.

The syntax of the Business# language is constituted by:

- C# operators (arithmetic, logical, assignment, and so on);
- C# basic types;
- C# statements, namely selection statements, e.g. *if* and *switch*, and iteration statements, e.g. *while* and *foreach*;
- C# classes, interfaces, enumerations and structures.

Although Business# wants to restrict the C# usage, its syntax is flexible. The analyst is allowed to include drawing algorithms in the business code, using according .NET libraries, if he considers that they are critical to the application. Nevertheless the framework would emit warnings about the consequences of mixing up business logic and implementation code.

Generation techniques have been thought since first days of the project. The analyst would do models in one language, which it is convertible to the target language of implementation. This method is widely used to generate skeletons of code from visual models in the Model-driven architecture . But if the analysis language should be formal, using two different languages would create a mismatch point between high-level and low-level designs [118].

Formal languages, as VDM++ and Spec#, were not adapted because they would maintain analysts and developers talking in different idioms. Moreover VDM++ has not a automatic method to convert their classes for .NET assemblies and Spec# is not a mature framework.

7.3 Results

The modeling uses a new framework, named Business#, which provides to the analyst a set of tools to create a better business model, namely:

- A base class that guides the persistence implementation;
- Generic and richer containers;
- A class that supports the design by contract development;
- Visualization of business classes through class diagrams;
- A code analysis component which guides the analyst to avoid going further than logical design.
- Testable models.

7.3.1 Framework library

The framework library supports the development according WARP solution. There is available a class that would be the parent class of business classes, named *BaseObject*. Although it does not provide a special functionality, it establishes a standard interface to implement persistence. In addition it provides a method that checks if any field or property is in the *null* state.

The framework provides rich containers: List, Map, Queue, Set and Stack. They are similar to collections available in .NET framework class library. However they have quantifiers' methods, which allows verify some statements about the collection items. For example, there are a method named *ForAll* that checks if all collection elements verify a condition. This condition is defined as a C# delegate method.

Additionally, a class was built with common rules on business modeling. For example, there are method to validate an email and a generic method to verify ranges.

Programming by contract

C# does not support programming by contract natively, thus a custom implementation is needed. The Business# library supports programming by contract through a class to define preconditions and postconditions and a technique to implement invariants.

The conditions are configurable, allowing to raise assertions or exceptions. Moreover they could be disabled, through .NET conditional compilation [119].

The support for invariants is more complex, as it requires runtime checking of application state. The common approaches to implement them are:

- A Check method that is explicitly called in the beginning and ending of each operation;
- Macros and preprocessors;
- Inversion of control containers with interception features;
- Aspect-oriented programming languages.

Calling explicitly the method is impractical as it produces many lines of code, which are hardly maintainable. Writing macros to be processed and add automatically the checking methods in the code would be interesting, but requires a new tool, which analysts and developers must learn. Some inversion of control containers have interception features that detect when certain parts of the code is executed. But this technique seriously constrains the software design, as objects must be registered in the container. An aspect-oriented language would be perfect, where people define the invariants as aspects.

Unfortunately .NET platform misses a mature and complete aspect-oriented programming framework. However there is an interesting tool, that is a mixture of preprocessor and aspect-oriented programming: PostSharp Laos. PostSharp is a post compiler processor of assemblies that could manipulate directly intermediate code, which could be integrated with MSBuild, thus with Visual Studio (figure 7.3). PostSharp Laos is a subset to support aspect-oriented programming. It provides a high level and programmatic interface to add cross-cutting functionalities, based on attributes.

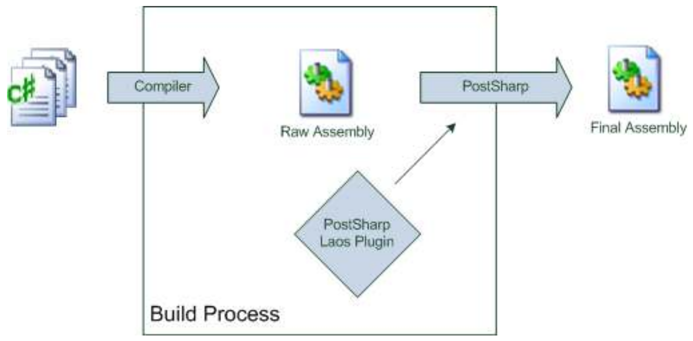
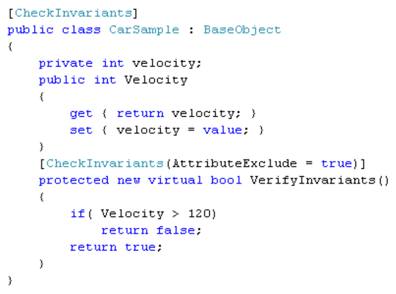


Figure 7.3: Building a project with PostSharp Laos

The analyst could define the class invariants in a method. Then he decorates the class with the aspect CheckInvariants, which runs the integrity check method every time a method or property of an object of this class is accessed. Figure 7.4) shows an example of this technique.



```
[CheckInvariants]
public class CarSample : BaseObject
{
    private int velocity;
    public int Velocity
    {
        get { return velocity; }
        set { velocity = value; }
    }
    [CheckInvariants(AttributeExclude = true)]
    protected new virtual bool VerifyInvariants()
    {
        if( Velocity > 120)
            return false;
        return true;
    }
}
```

Figure 7.4: Implementation of an invariant in Business#

7.3.2 Visual models synchronized with code

Class or activity diagrams are not formal languages. They could be misinterpreted as they are ambiguous. However visual models are important because they help people get a general image of the system. Using the Visual Studio Class Designer developers could view and edit the class diagram of the existent code. Moreover in the future more tools would appear with other types of visual models.

7.3.3 Code analysis

The business modeling should not care of implementation details. But limiting the syntax of language takes the liberty of analysts. Therefore a code analysis module was developed to warn analysts that they are going further than business modeling. For example, persistence and security concerns are out of scope of analysts.

It was implemented using FxCop, a framework to analyze .NET assemblies and emit warnings to developers. FxCop can be extended implementing new modules and integrating them with Visual Studio. In fact, the Business# rules applied in each verification are configurable in Visual Studio (figure 7.5). For more information, consult the tool review in section A.1.4.

The Business# rules implemented are:

- Recommend using BaseObject as base class;
- Recommend do not use abstract classes as they are harder to test;
- Warning the utilization of classes from .NET namespaces that are commonly used to do implementation: System.Collections, System.IO, System.Configuration,

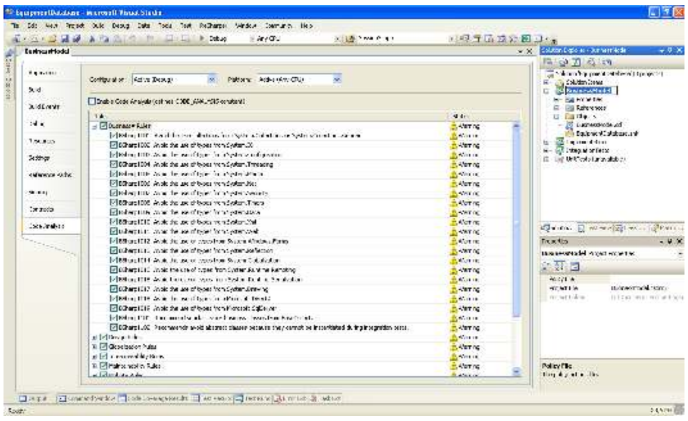


Figure 7.5: Business# rules in the Visual Studio

System.Threading, System.Media, System.Net, System.Security, System.Timers, System.Data, System.Xml, System.Web, System.Windows.Forms, System.Reflection, System.Globalization, System.Runtime.Remoting, System.Runtime.Serialization, System.Drawing, Microsoft.DirectX and Microsoft.SqlServer.

7.3.4 Testable model

Business# models the business through C# classes. Therefore, any unit testing framework compatible with .NET would be acceptable to test these classes. In the context of the experimentation, the Visual Studio Team System for Testers was used.

7.4 Discussion

Business# is an instance of WARP solution for .NET platform. It has proved that WARP vision could be implemented in a existent development environment. In fact, WARP ideas could be adapted to any object-oriented environment, e.g. Java or PHP, with more or less effort.

Nevertheless the feasibility of WARP does not means real world applicability. Only creating a real project that uses Business# is possible evaluate their advantages and drawbacks.

Business#

Chapter 8

Case-Study: Equipment Database

Equipment Database is a master database, which stores representation of all equipments used in memory manufacturing (Qimonda's core business), integrating sparse information around different information systems, such as:

- FAB300 for equipment tracking, scheduling and dispatching;
- SAP for plant maintenance, purchasing and asset depreciation;
- EBS for engineering data analysis.

Equipment Database is a case study, which serves as proof of concept of Business# and WARP solution ideas. That is, it is like the "white rat" used in the laboratory. Nevertheless it would provide value for Qimonda, its implementation has only research purposes.

8.1 Objectives

The application has no intention to be released to production and its main goal was to experiment and learn the impact introduced by some techniques on software development process, namely:

- Evaluate the applicability of Business#;
- Develop executable and testable business models;
- Testability on all stages of software development lifecycle according the V-model;
- Design by contract methodology;
- Creating adaptable, pattern-based and decoupled application architectures;

- Utilization of object-relational mapping tools;
- Prototyping and scaffolding techniques;
- Impact of code analysis and refactoring tools;
- Aspect oriented programming.

8.2 Functionality

Equipment Database manages the information about the manufacturing equipments, allowing multiple clients through different interfaces. Equipment Database would implement the following features:

1. **Information editor** - A simple user interface to create, list, update and delete equipments and associated entities: model, equipment type, vendor, unit processing type and persons;
2. **Equipment clustering** - The equipment can be composed by other sub equipments. Provide an easy way to attach, detach and move equipments;
3. **Equipment searching and filtering** - The equipments can be searched and filtered by attributes, e.g. by vendor, equipment type, and any state from any state model;
4. **Equipment type attributes** - The equipment type has a list of user-defined attributes (numeric or string) which user could add and remove;
5. **Equipment state models** - The equipments can have multiple state models, that is, finite state machines, which represents possible state flows. A finite state machine has an entry state, multiple target states, optional end-states and finite state transitions. The user could change any state model according predefined transitions;
6. **Equipment transactions logging** - Each modification on equipments should be logged, and can be consulted through user interface;
7. **Business service** - Separate the business logic and data access layers from user interface/presentation layers creating a web service;
8. **Web client** - Create one user interface to perform operations, based on built business service;

9. **General logging** - The system records history of all performed transactions, including state transitions and attribute updates, registering the user that performs that operation;

10. **Authentication** - There is no separate login for the users. They should use the existing Windows account;

11. **Role management and special permissions** - Any operation or transaction could require special permissions by role from current user to be executed;

12. **Create a executable business model based on Business# framework** - The application uses an executable, and thus testable, business model that can be easily reused in next projects. This business model is independent of implementation and represents a common knowledge about part of company business.

8.3 Documentation

This case-study is documented by two internal files: requirements specification [120] and system specification [121]. The former contains a detailed description of requirements. The second presents the architecture, the design and other technical decisions. The information presented in this chapter is a summary of these documents and it is only for research purposes. For further details about Equipment Database read these documents.

Additionally code comments were written using standard XML notation. Then they are used to produce automatically documentation of the application programming interface (API). Sandcastle Help File Builder was used to generate this documentation.

8.4 Architecture and Design

The chosen system architecture and design options have a profound impact in its modularity, maintainability and capacity to grow. Simultaneously, it affects the process of capture adequately the business rules and provide high value to customer. As a research project, the main goal is to get a well designed system rather than a robust and reliable system. The driven characteristics to build the system were:

- Repository architecture style [122];
- Independent components architecture style [123];
- Modular system, composed by several decoupled components;

- Rich, reusable and testable business model.

The creation of architecture was a careful and iterative process. Fundamentals about software architecture were studied to get the best software possible. In fact, the architecture has bundled several application patterns, as showed in appendix B.5.

8.4.1 Logical architecture

The logical architecture describes the system in terms of logical components. The separation in independent components is a hard task, but rewards the team with more maintainability and concurrent development. A useful technique is to create layers horizontally and vertically, creating boundaries that separate the pieces that compose the system.

Horizontal layers

Horizontal layers separate an application by level of abstraction: a layer call lower layers to provide a new higher-level functionality. A layer maintains clean and stable interfaces with its neighbors, avoiding cross-cutting calls, which means that changes on a layer have low impact on remainder layers.

Equipment Database is organized in five layers: data and storage management, data access, business logic, user interface and presentation (figure 8.1). The lower level ensures data persistence, concretized by a database management system. The data access is a tier that provides to upper levels a way to communicate with database. Business logic contains the domain entities and business rules, providing services in a high-level way. The user interface provides a human friendly application to interact with business logic, while presentation layer defines how display the information. Note the calls described by arrows, representing calls that occurs using messages between different processes, even different machines.

The business logic layer is composed by three internal components: business objects, a lightweight façade and a web service. Business objects are a representation of problem domain entities. The façade is a thin layer that transforms the fine-grained interface of business objects into a coarse-grained interface, composed by stateless and transactional methods. The web service exposes the façade through a standard interface.

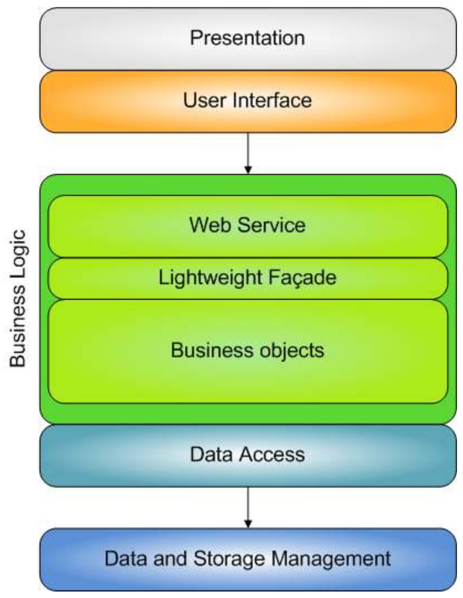


Figure 8.1: Equipment Database horizontal layers

Vertical layers

Vertical layers separate an application by functionalities and system properties, which cross all horizontal layers. They are deeply connected with functional requirements, categorizing them. Equipment Database involves, mainly, changing information of different entities. Thus six layers were created to represent these entities: equipment, model, vendor, equipment type, unit processing type and person (functional layers). Additionally there are layers concerned with management and security (infrastructural layers): authentication, role management and logging (figure 8.2).

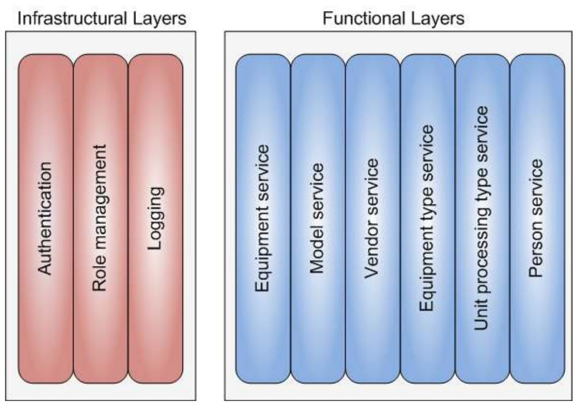


Figure 8.2: Equipment Database vertical layers

8.4.2 Physical architecture

The Equipment Database is a typical client-server application, where one web server responds to client requests through intranet/internet, using HTTP protocol. A web service holds all business logic, and a web server request its services through SOAP, to create the graphical user interface, a web application. The web service uses a database management system, to store and retrieve data quickly and safely (figure 8.3). In all the cases, the interaction starts by the element at left in the figure, that is, the caller are in the left side of connection. For example, the client starts the communication to web server, which answers based in calls to components at right.

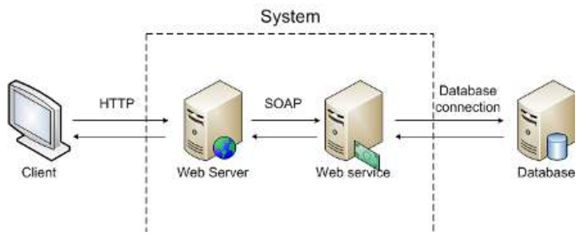


Figure 8.3: Equipment Database physical architecture

The connection between physical components through industry standards guarantees compatibility, and their possible independent deployment favors decoupling and scalability.

All business services are provided by the web service, in a high-level way. The authentication and role management is performed from there, which eases creation of new clients, like Windows and mobile applications.

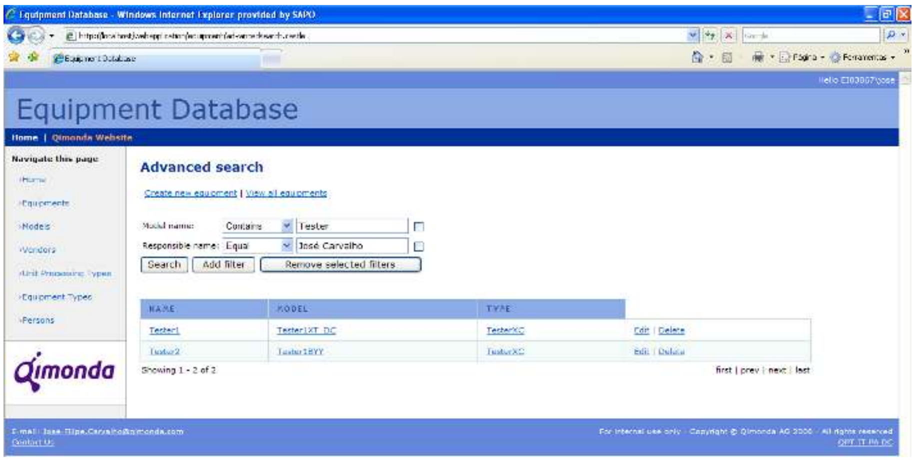
8.5 Results

The Equipment Database software was implemented successfully. João Cortez, the project supervisor, acted as a fictitious customer, participating in frequently discussions about requirements, architecture and design concerns. The application was developed incrementally, favoring oral communication and delaying detailed documentation. Although equipment transaction logging was not implemented and parts of the system were not well tested, the system is quite mature and stable, even when it was not meant for productive release.

The business model was coded using Business# framework and WARP ideas. The early building of models as code found prematurely requirement and design errors. Programming by contract and inheritance as contract established a bridge between business code and implementation code, finding fails of communication and synchronization. During the development, it was common that business preconditions detected an error on implementation code.

Equipment Database implementation was quite complex, especially by the range of new technologies explored. Castle ActiveRecord and NHibernate were used as object-relational mapping software. The implementation of user interface was supported by Castle MonoRail framework. General logging was created through a aspect-oriented programming tool, PostSharp Laos. Moreover the implementation of security concerns was challenging, based on specific ASP.NET technology. Chapter B contains more technical details about Equipment Database implementation.

The resulting application is composed by three physical components: base library, web service and web application. The former contains business classes and respective implementation classes. The web service exposes a coarse-grained interface over base library. The web application connects with the web service to provide a graphical user interface. Figure 8.4 shows the web application interface of Equipment Database.



Model name: Contains Tester
Responsible name: Equal José Carvalho

NAME	MODEL	TYPE	
Tester1	Tester1XT_DC	TesterXC	Edit | Delete
Tester2	Tester1BYY	TesterXC	Edit | Delete

Showing 1 - 2 of 2

E-mail: Jose-Filipe.Carvalho@qimonda.com
Contact Us
For internal use only · Copyright © Qimonda AG 2006 · All rights reserved

Figure 8.4: Graphical user interface of Equipment Database

8.6 Discussion

Equipment Database case-study provided an enormous feedback about Business# framework and WARP ideas. In fact, during Equipment Database implementation, Business# was changed to accommodate some flexibility. For example, some methods are left as virtual, allowing their re-implementation by subclasses.

Surprisingly, Equipment Database revealed that modeling problem domains as code is quite simple. These models are very understandable, exposing clearly and formally the interface and relations between objects. They miss the common "plumbing code" of implementation, e.g. code to ensure persistency or security. Business analysts can understand them as they understand UML class diagrams. In fact, the generated visual models are useful to create a general view of classes.

Nevertheless the project found some limitations in WARP ideas. Firstly their application constraints severely the software design. The lack of multiple inheritance in most of today object-oriented languages, as C#, complicates the development. As implementation classes inherit business classes, they cannot inherit from classes of other frameworks. For example, Castle ActiveRecord has a base class with generic persistency methods. However implementation classes cannot use it, because they already inherit business classes.

A fragile aspect of WARP exposed during this case study is testing. Logical tests intend to verify business logic, independently of implementation. But the boundary between logical and physical tests is very blurred. Most of the time logical tests need an underlying structure to run, that is, the implementation code.

Lastly, Equipment Database demonstrated that WARP business modeling is not suitable for every project. Some applications are very simple, requiring only few classes and the basic CRUD operations. In these cases, the risk of creating an *anemic domain model* [124] is very high. That is, the software will probably have a model without any business logic. For example, Equipment Database business logic is quite simple, mainly with persistency methods to retrieve and save data. In these cases, separate logic from implementation has no real advantage.

The studies about Equipment Database architecture achieved a good general software structure, logical and physical, that can be reused in future applications. Moreover the range of new technologies explored can be integrated in next projects. Castle ActiveRecord and NHibernate proved that are a mature and complete object-relational mapping solution. Similarly, Castle MonoRail emerged as a very good MVC framework. Finally PostSharp Laos, an aspect-oriented programming tool, decreased significantly the effort to do general logging.

Finally, the case study demonstrated the value of code analysis and refactoring tools. Visual Studio Code Analysis, the integrated version of FxCop, verifies if .NET assemblies follow good design practices. Resharper [125] is a refactoring tool, with code completion and showing suggestions to improve and to clean the code. Both are used in the project, really improving developer productivity and software quality.

Part IV

Wrap up

Chapter 9

Lessons learned on software development

This chapter presents some good practices on software development learned during WARP project. In opposition of most texts about this topic, it would give simple and practical advices that can be applied in everyday developments. Most of them are applicable on Qimonda development activities but also to most of software companies. Although it is only presented examples of tools and technologies explored in the context of the project, probably there are similar applications on other platforms.

9.1 Software development process

Context-driven school
This ideas follow the Context-Driven School [1]. Any practice recommended below can be good or bad according the context.

Keep it simple
The processes should be simple. A small set of simple guidelines for your software development process would be enough. The development team must be aware and follow these guidelines. If some guideline proves that is unfeasible in the context, it is better discarding it. Sometimes the team profile or project specificities make harder implement some guidelines. Similarly, a small number of artifact types is a good sign. As a reference when more than 20 different types of artifacts are identified in the development process or one person must work with more than 5 types, maybe something has been going wrong.

Agile requirements
Avoid to perform a long requirements analysis phase or to create large requirement documents without any implementation. After create the system vision, one or two

weeks to create one small requirement document, with less than 30/40 pages, would be enough. Requirements change very quickly, so detailing requirements that will never be implemented is a waste. Moreover, it is very probably the first prototypes showed to the client would change the requirements.

Keep it fun
Fun and creative environments increases the innovation, a competitive advantage for every company. For example, putting two Java lamps [6], red and green, in the development room to inform that build server has errors or not makes the environment more relax. The famous Google white boards around the office, with sketches and new concepts would make the environment more fun and creative. However developers should be heard during the implementation of these measures: they can think that it is a measure to control them, like in a factory. Instead, a better solution would be give a list of possible measures and ask for new ideas.

Keep it automated
A person should not do repetitive tasks, as running tests or deploying systems. A software development process using continuous integration techniques provides a real competitive advantage, leaving developers to focus on problem domain and customer satisfaction.

9.2 People

Training in software engineering tools
Teaching developers how to use software engineering tools is critical. It is not rarely that people do not understand how use a certain software in their development tasks. Moreover many do not understand why that tool is adopted. Therefore training must explain the specific features that make the company adopt it.

Technology experts
Every company has people that know everything about one programming language or a database system. However when their knowledge is needed, most of people do not know how to find them. Creating the role of technology experts and establishing a simple process to request their help would solve this problem.

Everyone knows how to design
Software design is an ongoing activity. Therefore teaching design to developers would improve software quality. If they know something about design they will understand architecture compromises and they will collaborate to improve software design. "Design Patterns: Elements of Reusable Object-Oriented Software" [14] and "Patterns of Enterprise Application Architecture" [12] are two good books to create small presentations about this topic.

9.3 Knowledge

Do not reinvent the wheel
A common error on software development is to create things that already exist. This applies both for new products and for used software engineering tools. In most cases there are existing applications, open source or commercial, that provides desired functionality.

Information repositories
Information repositories stores the knowledge acquired in the past. They should be available for all users with different levels of detail. Ideally their content can be searched. Repositories increase reusing in all aspects of software development. For example, the following archives would be interesting: code repositories (e.g. Google Code), architecture repositories, business object repositories, requirement documents repositories and portfolio of products.

Distribute information transparently
The information should be published as much quickly and transparently as possible. If developers understand the global context of business, they can make smarter choices. For example, a dashboard of organization can be created, including current projects and product portfolio.

Bottom-up and Top-Down approaches
Bottom-up approach is usually better to produce documentation and coding, but top-down approach is easier to read and to create a common understanding[126].

9.4 Programming

Test carefully a new tool
A new tool must be tested carefully before adoption. Sometimes the product seems magic, but then problems arise when it needs some type of customization. ASP.NET controls are a good example these type of situations.

Code coverage
A testing framework with a code coverage feature provides a simple way to know how much of the code is tested and to find unused code.

Code analysis
A code analysis tool inspects the code for potential errors on design, naming, globalization, maintainability, security and so on. It provides to developers a full set of coding good practices without much effort, eliminating potential errors automatically. For example, Microsoft Visual Studio 2005 Team System has bundled a code analysis feature. Similarity, FxCop provides the same functionality in a standalone and free application.

Refactoring tools
Developers spend much time with code layout, checking imports, creating class or methods skeletons and so on. Refactoring tools help them in this task. Although most of integrated development environments (IDEs) have some features, many of that are too simple. For instance ReSharper adds to Visual Studio powerful refactoring features.

Code comments
Formal documents do not substitute code comments, which are critical on the maintainability of application. Commenting the code with a standard syntax allows generating documents with application programming interface (API), e.g Sandcastle Help File Builder for .NET languages and Javadoc for Java. Moreover code scattered in the code explains used data structures and algorithms.

Coding standards
Coding standards are important, as they improve code readability and cleanness. Developers should learn internal coding standards, like they learn other organizational policies. Ideally these rules would be added into a code analysis tool. The documents [127] and [128] are examples of coding standards.

Common Libraries
Common libraries are a set of classes and methods that can be reused in other applications. These libraries are shared from different projects, but they are not a specific product. Usually they are too general to have an allocated team and too company specific to exist as third-party software. They can be managed as an Open Source project, where programmers of different teams collaborate and discuss changes. They could be called the *common code knowledge*.

9.5 Artifact control

Document control
Nowadays it is mandatory use source control software, not only for code but also for documents. For example, Visual Studio Team Foundation Server or Subversion would be perfect for that.

Artifacts traceability
Text artifacts must be related with code, using a tool, e.g. Team Foundation Server, or simple comments on code. For instance, if you have a non-functional requirement as application security you should explain in your code that some variable checks are related with this requirement.

Chapter 10

Conclusion

The WARP approach is a set of techniques that definitely can improve existent software development processes. The study of current techniques and experimentation of new concepts allowed to consolidate ideas and to achieve a general approach.

WARP intended to be one step to increase the level of abstraction in software development. Since the first days of software engineering, people have created new languages and tools, trying to be closer of human thinking and far off hardware behavior. Nowadays compilers allow developers write code in higher-level languages, increasing their productivity. Now it is the moment to look for new ways of abstraction. That does not mean to eliminate completely programming tasks, but create new tools and methods that automate or reduce the effort of certain activities. WARP dreams a process where a formal business model generates the basis of persistency and user interface layers. Although it is not completely successfully, it can lay the foundations for further developments in this domain.

Business modeling through programming languages proved to soften the separation between analysts and programmers. The reutilization of business models in the implementation stage establishes a contract between the two parts, highly manageable afterwards. Especially in the case of complex business logic, decoupling the business model and implementation code can really improve discipline and reusing. However, in the case of small and simple applications, the benefits would be lesser than their drawbacks.

Although WARP ideas are not appropriate to build all software, part of these ideas is widely accepted. The formality of design through the code allows eliminating ambiguity, the separation of interfaces definition and implementation in different assemblies is a famous architecture pattern and programming by contract is adopted as a way to define expectations.

Object-relational mapping demonstrated to be a very useful programming technique, improving developer's productivity significantly. The process of mapping is simple and systematic, providing an object-oriented interface to communicate with persistency layer. The developer can implement most of operations without be aware about specificities of relational schemas. Moreover the business logic is defined as objects instead of as tables and stored procedures. Therefore, applications become independent of underlying databases, reducing the risk of a later port.

Prototyping the user interface attested as a way to improve communication between development team and customers. Scaffolding techniques emerged to avoid throwaway prototyping, generating automatically the code of user interfaces from models. Ideally these models would be classes with embedded business logic, but there are several challenges to address. Therefore current tools generate from static relational schemas and they are quite immature. Nevertheless this technique will evolve quickly in the next years.

The adoption of open source software is not necessarily risky. Most of the time, it is very mature and well-documented, produced by large communities and tested on multiple different projects. Erroneously software industry is still reluctant in adopting this type of software. NHibernate, Castle ActiveRecord and Castle MonoRail have proved their qualities, being a reference in their domain.

Finally, these ideas are more conceptual than technical, which means they can be applied to other development platforms. Although WARP, with Business# and Equipment Database implementations, tried to adapt to .NET framework applications, its theories can easily ported for other environments, as Java or C++.

The integration of WARP concepts and explored technologies in existent software processes will definitely speed up and improve the quality of produced systems.

10.1 Summary of contributions

This project contributes with a summary of current knowledge about the software development process. The listing of software engineering trends is an original text, resulting from the effort during the project analysis stage. In fact, this updated information can be included in lessons of a software engineering course.

The project outlines the importance of communication and how intermediate and ambiguous specifications are potential sources of errors. In the same way, it explained the central role of business modeling activity in software development. A possible path to improve it was presented, together with generative techniques to produce persistency and user interface layers from business models. Additionally it provides a common vocabulary and classification for multiple different and unrelated techniques in these topics.

The exploration of technologies and tools emerged a set of discussions and recommendations, which can help development teams to choose the suitable environment for them.

The project focus was not creating a new product, ready for production. However Equipment Database software achieved a good level of maturity, after a disciplined and iterative process. Moreover it made available references for documentation and code samples of explored technologies.

In addition, Equipment Database brought a new architecture for integration systems, with defined components and technologies. Surprisingly, most of the time experts are reluctant in presenting a general good architecture with a respective implementation. However Equipment Database was proposed this solution, based on WARP ideas and architecture patterns.

The lessons learned on software development presents some general advices for companies enrolled in this activity and it was a result of the knowledge and experience acquired during the project.

10.2 Future developments

The WARP approach is a result of a four months master project. Obviously there is place for future developments, which would mature the ideas and techniques presented. Publishing and discussing these ideas with other people probably will originate conceptual and technical improvements.

Although WARP methodologies have proven to be beneficial, they should be tested in bigger projects. In Equipment Database only one person played as analyst and programmer. Moreover the project conditions are favorable, with a stable vision by one customer. In fact, WARP approach for business modeling would be more valuable in large projects, with more complex requirements and more people enrolled. The formality of WARP fits perfectly in this scenario, where misunderstandings emerge easily.

Business# is a very young framework, which needs more usage and consolidation. The integration between business and implementation classes can be improved, for example using new types of contract. Specialized editors would appear to support coding business classes, as well as tools to generate different types of visual models, e.g. work-flow diagrams. In the same way, ports to other platforms would be interesting.

The utilization of a domain specific language to do business modeling, instead of a traditional programming language, should be evaluated. The simplicity and flexibility of domain specific languages would be interesting. The WARP project was considered that, but the effort of creating a new domain specific language, compatible with target platform, is considerable.

The WARP techniques can be considered types of Model-driven engineering, as both advocate the central role of models. The main difference is that WARP requires formal and executable models. The body of knowledge on Model-driven engineering is very large, so some of these ideas can be used with WARP concepts.

Finally, Equipment Database can evolve to production software, providing value for Qimonda with little effort.

Part V

Glossary, Appendices and References

Glossary

Annotation (programming) - it is a code declarative tag, which adds additional information and behavior to certain entities, like classes or methods. The correspondent in C# is attribute.

Attribute (programming) - it is a code declarative tag, which adds additional information and behavior to certain entities, like classes or methods. The correspondent in Java is annotation.

Business-Driven Development - it is a methodology to create information technology systems that satisfy the business requirements. It starts from business strategy, requirements and goals, and then transforms then into a solution.

Computer Aided Software Engineering (CASE) tool - it is a tool that assists the developer in their tasks.

Code analysis - it is an automated inspection of the code to find common design or programming errors not found by the compiler. Usually they are based on recommended practices for adopted framework, finding prematurely many defects without the human effort of manual reviews.

Code coverage - it is a metric that outlines the executed code during testing, providing related statistics. It could be good to detect useless or untestable code.

Code repository - it contains large amounts of code, for public or private use. Usually it contains a way to control changes and handle multiple versions, and some have a search feature.

Delegate method - it is a type that references a method. It allows the selection of a method to call at runtime.

Design by contract (DbC) - it is an approach to design computer software, that defends the creation of precise and verifiable interface specifications, that ensures contracts between system components.

Domain Driven Design (DDD) - it is a school in software engineering that emphasizes the importance of domain and domain logic in software development.

Event-response table - it is a way of organize and document user requirements, describing all events and expected system responses [10].

Function points - it is a unit to measure how much business functionalities are implemented functionality, from a point of view of customer.

Integrated Development Environment (IDE) - it is a software application that helps the programmer in its tasks, providing a source code editor, a compiler/interpreter, build automation tools, etc.

Inversion of control - it is a programming technique that changes the way how components are called. Objects are registered in containers, allowing that any object could find other [129].

Kaizen event - Kaizen is a Japanese word for improvement. Kaizen events puts together people to discuss and solve a key problem of the organization.

Mock framework - it provides methods to create fake objects, which simulate real objects.

Namespace (programming) - it is a concept used by several programming languages to group related logical entities, as classes or interfaces.

Object-Relational Mapping (ORM) - it is a programming technique for converting data between relational databases and object-oriented programming languages [130].

Pair programming - it is a software development technique where two programmers work side by side, using the same computer. One of them typing whereas the other review the code.

Partial class - it is a feature supported by some object-oriented languages, where a class definition could be scattered in multiple files.

Scaffolding - it is a type of prototyping that aims to generate automatically user interfaces from domain models or relational schemas.

Software pattern - it is a structured way to describe a good solution to solve a specific problem. It resumes the knowledge and experience captured in previous projects, giving to software engineers guidance in their work.

Stakeholder - represents all the persons and organizations interested in or affected by a product.

Task concurrency - this term could be used in two different contexts: project management and computation. In the former, it means the possibility to do things at the same time, dividing small pieces of work among different people. In computation, task concurrency is related with different processes or threads that struggle for the system resources.

Template method - it is a method that classes leave unimplemented, expecting that subclasses concretize it (see Template Method pattern [14]).

Test-Driven Development (TDD) - it is a software development technique that advocates the importance of write before test cases and then produce the code to pass these tests.

WARP - it is the short name of this project, which aims accelerate the software development process. This name was inspired on WARP drive, the faster-than-light movement in science fiction, as used in Star Trek [131].

Appendix A

Technical Reviews

During the project, several techniques and tools were assessed to evaluate their potential utility. This chapter presents summaries of these reviews.

A.1 Tool Reviews

A tool review summarizes the knowledge about it, after read documentation and trying to use it. This section presents the tool reviews performed during the project.

A.1.1 Spec#

Summary

The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. The Spec# system consists of:

- The Spec# programming language. Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and postconditions as well as object invariants;
- The Spec# compiler. Integrated into the Microsoft Visual Studio development environment for the .NET platform, the compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools;
- The Spec# static program verifier. This component generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

A unique feature of the Spec# programming system is its guarantee of maintaining invariants in object-oriented programs in the presence of callbacks, threads, and inter-object relationships [132].

Date of review: 28th February 2008 (updated on 15th June 2008)
Homepage: `http://research.microsoft.com/specsharp/`

Actual version: preliminary
Last release: 11th April 2008
Tested: Yes
Owner/Sponsor: Microsoft Research
License: Microsoft Research Shared Source License Agreement

Advantages:

- Several tools and documentation available that support Spec# utilization;
- Spec# is a superset of C# (easy to learn);
- OCL-like constraints supported (object invariants, method pre- and post-conditions);
- Support for checked exceptions (that C# doesn't support);
- Tight integration with Visual Studio.

Disadvantages:

- It is a research project: is quite risky incorporate a research tool in production software;
- There are no guarantees about the future developments and support;
- There is no guarantee that callees from other languages of framework (C#, VB.NET, etc.) will work;
- A new language: demands time to learn;
- Lower performance: the constraints require run-time checks;
- It is not clear if Spec# is a programming language only for specification, or to implement all the application using it.

Conclusion:

The Spec# programming system is a promising tool, but for now it is clever not use it for production systems. Probably Spec# will be incorporated in future releases of Visual Studio, as new language or as a new feature of C#. However some concepts, namely Design by Contract and OCL constraints, could be incorporated in Warp framework.

A.1.2 Himalia

Summary:

Himalia provides a way to develop application focusing in user interfaces, establishing paths that the user will follow (Navigation Model), division of screen and relationships among different areas (layout behavior) and themes and styles. It is based in Microsoft DSL Tools.
Date of review: 5th March 2008 (updated on 21st June 2008)

Homepage: http://www.himalia.net
Actual version: 0.8 (beta)
Last release: 12th December 2006
Tested: Yes, it was tested in the laptop, as it requires .NET framework 3.0. A new project was created using the open Pet Shop example. The diagrams in Pet Shop example are pretty, but when trying work with application, multiple problems have arisen. It is quite unstable, crashing Visual Studio 2005 several times, and many features are not implemented, e.g. Navigation Model Toolbox does not exist.
Owner/Sponsor: Leonardo Vernazza
License: not founded

Advantages:

- Provides good practices to create user interfaces;
- The concept, model-driven user interfaces, is quite interesting, namely for capture user requirements. User interfaces are very concrete items that user can easily observe and discuss;
- Integrated with Visual Studio 2005 and created using Microsoft DSL Tools;
- Free to download, however license is not specified.

Disadvantages:

- It is closed source and apparently it is an individual project: forum community has 4 or 5 posts, most of them from author;
- The list of known issues is big, and include sentences like "Some controls may not behave as expected under certain circumstances";
- During testing the application is very disappointing - many bugs and not implemented features;
- Most of planned releases had been failed last year and appears that project was abandoned.

Conclusion:

The concept, Model-Driven user interfaces, is really interesting and could be applied to real software projects, namely where user interfaces have an important role. However, the tool available is not recommendable: it is in a early beta version, closed source, with many bugs and features that are not implemented. Besides Himalia appears to be a discontinued project.

A.1.3 ADO.NET Entity Framework

Summary:

The ADO.NET Entity Framework, part of the ADO.NET components of the .NET Framework, is an object-relational mapping technology from Microsoft. It is geared towards solving the mismatch between the formats in which data is stored in a database and in which it is consumed in an object-oriented programming language or other front ends.

Date of review: 5th March 2008 (updated on 16th June 2008)

Homepage: `http://msdn2.microsoft.com/en-us/data/default.aspx`

Actual version: Beta 3 (final version will be available in 2008 1st semester)

Last release: 12th June 2007

Tested: No. It requires installing Visual Studio 2008 and then installing ADO.NET Entity Framework Tools (see below). However several demonstrations were watched to understand this tool.

Owner/Sponsor: Microsoft - ADO.NET team

License: Microsoft Pre-Release Software License (until final release)

Associated Tools

ADO.NET Entity Framework Tools CTP 2 (for Beta 3): contain tools to enable developers making use of Entity Framework within Visual Studio 2008 (Visual Studio 2005 hasn't support in last releases), namely the ADO.NET Entity Designer. However this tool does not support (yet) all the features of Entity Framework, but it is expected that it will support.

Advantages:

- The Entity Data Model (EDM) Designer provides a simple, flexible and graphical way to make mappings;
- Intellisense support that helps the developer;
- The tool addressed maintainability issues;
- The code to access database is succinct, understandable and object oriented: we iterate objects instead of iterate rows;
- Related objects can be loaded using references;
- Lazy loading: the framework only retrieves data when needs the data;
- Common interface to query different artifacts: LINQ to Objects, LINQ to DataSets, LINQ to SQL, LINQ to Entities and LINQ to XML;
- The project sponsor: Microsoft;
- Planned support for other DBMS within 3 months the release of ADO.NET Entity Framework: some companies manifested their intentions to implement ADO.NET providers to Oracle, MySQL, PostgreSQL, IBM DB2, Informix and others [133];
- Drag-and-Drop an GridView or DataSet can be considered a scaffolding technique (an old feature of Visual Studio);
- It works very well with stored procedures;
- It helps very much in software maintenance and evolution.

Disadvantages:

- The tool is, in its essence, a mapping tool: relates objects to tables, or reverse-engineering database schemas (tables, views and stored procedures) to create non-behavioural objects (only with methods to access database). Business logic is added using partial classes feature, but it is questionable if that is a disciplined way;
- Microsoft only supports Microsoft SQL Server;
- Requires C# 3.0 and Visual Studio 2008 to compile and .NET Framework 3.5 to run. The license costs and deployment costs could be significant to upgrade legacy systems to support these frameworks;
- Traditionally, Microsoft data access technologies change (sometimes, a huge change) every 3 years.
- Final version is not released;
- It is not possible generate one database schema from business classes - only do mapping with existent tables. In July 2006, the authors did not plan provide this feature [134].

Conclusion:

The framework is very good and, in my opinion, should be adopted in future projects - namely whom connects Microsoft SQL Server or in projected upgrades in software that uses ADO.NET. However, there are uncertainties about how it would perform in a disciplined software development process.

A.1.4 FxCop

Summary:

FxCop is an automatic code analysis tool for .Net assemblies which targets eliminate developments errors during implementation, providing to developer useful guidelines when detects a possible error.

Date of review: 31st March 2008 (updated on 16th June 2008)
Homepage: `http://code.msdn.microsoft.com/codeanalysis`
Actual version: 1.35 (stable) or 1.36 Beta (development)
Last release: 10th October 2007
Tested: Yes. It was used during the entire project, and custom rules was implemented to support Business# development.
Owner/Sponsor: Microsoft
License: Microsoft Public License (Ms-PL)

Advantages:

- Easy to use: any developer could start using it;
- Using the tool, the developer learns most of guidelines inside it;

- If the developer follows its advices, he will avoid many programming errors;
- User-friendly interface, both in Visual Studio or in standalone application;
- Sponsored by Microsoft;
- Support for all .NET languages: it runs over assemblies;
- Extensible: creating custom rules is quite easy [135];
- Supports the new generation to improve quality: avoid the defects using automatic code reviews, instead of manual reviews;
- Tight integration with Visual Studio Team System.

Disadvantages:

- Sometimes warnings can be annoying. However they could be disabled in project properties menu (all, by type or individually);
- It slows the building process;
- This tool removes some of developer's freedom. Although that could be good (improvised solutions would be eliminated), it would decrease chances for possible innovative solutions;
- Documentation is incomplete and sparse. It is hard find information to create custom rules, and sometimes that can be a tricky experience;
- Although sponsored by Microsoft, it had lack of advertising and visibility in the last years;
- Some advices do not mention the line/file where the problem occur. It is a known bug of FxCop, caused because it relies over PDB files generated in Visual Studio projects;
- No integration for other Visual Studio versions.

Conclusion:

FxCop is a great tool and it is recommendable for any software project. Its included guidelines are great lessons of programming in .NET environment, and extensibility features are great to create custom rules that fit specific organization rules. It eliminates errors in first phases of development: is like have a code review in every compilation.

A.1.5 Aspect.NET

Summary:

Aspect.NET is an Aspect-Oriented Programming framework for .NET, based in AspectJ principles. It is tightly integrated with Visual Studio 2005 and is supported by Microsoft Research.

Date of review: 28th March 2008
Homepage: https://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6801
Actual version: 2.1
Last release: 21th April 2007
Tested: Yes. It was installed, as such Microsoft Phoenix, which is required to run this tool. One aspect was implemented but it do not weave into application: Visual Studio add-in does not work properly.
Owner/Sponsor: Vladimir O. Safonov, St. Petersburg University. Supported by Microsoft Research.
License: Not found.

Advantages:

- Follow the AspectJ approach, creating a new language named Aspect.NET.ML to implement aspects. It is simpler for people that know AspectJ;
- Supported by a big team of researchers at St. Petersburg University and by Microsoft Research;
- Tight integration with Visual Studio 2005;
- Good user's guide.

Disadvantages:

- Closed project;
- Lack of information available in Internet;
- During testing, the aspect was not woven into application.

Conclusion:

Aspect.NET could be a good tool in the future. The similarities with AspectJ are obvious and support by Microsoft Research reveals that it could be integrated within future releases of Visual Studio. For now it is a research project and does not work so well, so it is only recommendable for research activities.

A.1.6 Aspect#

Summary:

Aspect# is an Aspect-Oriented Programming framework for .NET, based on Windsor/MicroKernel features of Castle Project.
Date of review: 31st March 2008 (updated on 21st June 2008)
Homepage: http://www.castleproject.org/aspectsharp/
Actual version: Dead project [136]
Last release: Not found.
Tested: No. After reading documentation, it was discovered that the project was discontinued.

Owner/Sponsor: Open source project (Castle Project)
License: Not found.

Advantages:

- Open project, someone could restart it in future;
- Castle support/community.

Disadvantages:

- Dead project (the most relevant);
- Lack of documentation.

Conclusion:

Aspect# was a promising project but now is a dead project. Nowadays Castle supporters defend interception capabilities of Windsor/MicroKernel project as alternative to Aspect#. However these features imply saving most of the objects in the inversion control container, one requirement that is hard to meet for all software projects. Moreover, this option does not provide a full AOP framework and there are lack of development tools. So, neither Aspect# nor Windsor/MicroKernel are acceptable AOP frameworks.

A.1.7 PostSharp

Summary:

PostSharp is a post compiler for .NET, that is, it could change IL code after compilation of one assembly. It includes PostSharp Laos library, a simple Aspect-Oriented Programming (AOP) framework to add aspects to our applications using custom attributes.
Date of review: 28th March 2008 (updated on 21st June 2008)
Homepage: http://www.postsharp.org
Actual version: stable version 1.0.8.316 (Release Candidate 2) and development version 1.0.8.331
Last release: 11th March 2008 (version 1.0.8.331)
Tested: Yes. It was tested and a used to implement invariants.
Owner/Sponsor: Open source project, lead by Gael Fraiteur
License: GPL/LGPL v3

Advantages:

- PostSharp Laos really works: it is the first AOP framework for .NET where I could implement something;
- Language agnostic: it works over IL code;
- It is simple to use and it is easy understand its underlying principles;

- Based in one .NET feature: custom attributes;
- Alive project: last release this month;
- Mature project: more than 2 years of developments;
- Open source project;
- It has a growing community and one forum that provides good support.

Disadvantages:

- Documentation available is quite small and lacks of examples;
- Sometimes it is difficult debug our applications (but it is a common problem in AOP development);
- Although integrated with Visual Studio, one tool to explicit the pointcuts would be great;
- Subclasses do not inherit aspects of their parents;
- The performance implications of PostSharp are uncertainty. It degrades the application execution because it adds many operations in several points of assembly and it uses reflection. But that does not mean that is worse than manual code.

Conclusion:

PostSharp is a good tool and deserves a special attention of .NET community. At least that I can find, it is the best AOP tool for .NET. However it should be used only when it is really necessary. It was used on invariants implementation of Business#.

A.1.8 Castle ActiveRecord

Summary:

Castle ActiveRecord is an open source ORM (object-relational mapping) library that relies on NHibernate. It uses class inheritance and attributes to decorate a class with persistent methods, avoiding XML configuration files of NHibernate.

Date of review: 11th April 2008 (updated on 21st June 2008)

Homepage: `http://www.castleproject.org/activerecord/`

Actual version: stable version is Release Candidate 3 and development version the Build 880

Last release: 19th June 2008

Tested: Yes. It is used in the implementation of Equipment Database case study.

Owner/Sponsor: Open source project (Castle Project)

License: Apache License, Version 2.0

Advantages:

- The process makes sense: develop business classes and then relational schema;
- It is simple and clear;
- Open source project;
- Mid-size community and good forum;
- Schema generation - it is possible create a simple application without learn SQL;
- Almost XML free;
- Scaffolding possibilities, with easy Castle MonoRail integration;
- Number of databases supported: Microsoft SQL Server 2005/2000, Oracle, Microsoft Access, Firebird, PostgreSQL, DB2 UDB, MySQL and SQLite (the same that NHibernate);
- Open source contributors use it in several production applications, so it is relatively well tested.

Disadvantages:

- The mapping is *code intrusive*: it relies on inheritance and attributes;
- The library have been evolving very quickly and because that many of its interfaces have not frozen yet. Likewise some minor bugs were found (or what could be called an unexpected behaviour) and unimplemented features;
- The performance could worsen, as it introduces several abstraction layers (NHibernate + CastleActiveRecord). However that would be minimized by other ORM features, like caching and lazy loading;
- For the beginner it is hard understand how it works and relationships among their components.

Conclusion:

Castle ActiveRecord is a great tool and it really abstracts relational operations for a developer. It seems work very well in mid-size applications, and when they release version 1.0 could be used by any enterprise application. It is a really developer productivity booster.

A.1.9 NHibernate

Summary:

NHibernate is an ORM (object-relational mapping) tool. It uses XML mapping files to relate classes and relational tables, providing a common API to connect to databases.

Date of review: 11th April 2008 (updated on 21st June 2008)
Homepage: `http://www.hibernate.org/343.html`
Actual version: stable version is NHibernate 1.2.1GA and development version is NHibernate 2.0.0.Alpha1
Last release: the stable version was released on 27th November 2007 and the development version was released on 30th March 2008
Tested: No, directly. In fact, it is used when during Castle ActiveRecord utilization to perform complex queries and debugging. But it is not used directly through its API, neither XML mappings were performed.
Owner/Sponsor: Open source project (port of Hibernate for Java)
License: LGPL v2.1

Advantages:

- Mature project and big community;
- The business classes and mapping documents are separated (no code intrusive);
- The control over mapping is bigger, however demands more effort to learn;
- Well documented;
- Number of databases supported: Microsoft SQL Server 2005/2000, Oracle, Microsoft Access, Firebird, PostgreSQL, DB2 UDB, MySQL and SQLite;
- Open source project.

Disadvantages:

- XML burden;
- The framework is enormous and sometimes it is hard know how to start;
- No scaffolding possibilities.

Conclusion:

NHibernate is a powerful and mature tool, in contrast with most of their competitors. It provides a good set of classes to access databases and to provide object persistence. However its XML burden is quite annoying and decreases developer productivity. Fortunately there are applications as MyGeneration [137], which could generate automatically those XML.

A.1.10 SubSonic

Summary:

SubSonic is tool that generates the Data Access Layer from relational schema, which can be extended to incorporate business logic using partial classes. It has an awesome scaffolding control.
Date of review: 11th April 2008 (updated on 21st June 2008)

Homepage: http://subsonicproject.com/
Actual version: stable version is SubSonic 2.0.3
Last release: 11th July 2007
Tested: Yes, but only with a simple sample.
Owner/Sponsor: Open source project (mainly maintaned by Rob Conery)
License: Mozilla Public License 1.1

Advantages:

- It is simple;
- Scaffold feature is great;
- Its performance could be better than its competitors;
- It could rely on Microsoft Enterprise Library providers;
- Open source project;
- Almost XML free.

Disadvantages:

- The process does not make sense: start creating relational schema and then write business classes;
- Young project and smaller community;
- Less databases supported: Microsoft SQL Server 2005/2000, MySQL and Oracle.

Conclusion:

Subsonic is great tool, but it has a design disadvantage: it starts from relational schema and not from business logic. Because ADO.NET Entity Framework would probably do the same with pretty wizards, probably SubSonic will have a limited success.

A.2 Technical Comparisons

Technical comparisons is a way to resume and compare the characteristics of similar tools, to help people decide the best one in their context.

A.2.1 ORM tools

Object-relational mapping tools (ORM) help the programmer to relate objects to relational schemas. This technical comparison tries find out the similarities and differences between some well-known ORM tools for .NET environments, which are explicit in table A.1.

Date of review: 11th April 2008 (updated on 21st June 2008)

Table A.1: Table with object-relational mapping tools comparison

	Castle ActiveRecord	SubSonic	Hibernate
Method to relate objects with tables and objects	inheritance and attributes	Generate code of Data Access Layer from relational model	XML documents
ORM input	business classes with attributes	relational schema	mapping XML documents
ORM output	business classes with aditional methods that provide persistency	Generated namespace with classes that access original relational schema	Interface to provide persistency for business classes pre-configured on XML documents
Requires that business class inherits certain class	no, but is preferable	no	no
Create relational tables from classes	yes	n/a	no
Create a DDL script (relational schema) from classes	yes	n/a	yes (you can generate schema from mapping files, using Schema Export tool)
Create business objects from relational schema	yes (using Active Record Class Generator - third part tool)	yes (using partial classes you could extend generated classes)	n/a
Easy binding to .NET controls	yes	yes	yes
Scaffolding	yes (integrating with Monorail)	yes (using Scaffold control or AutoScaffold)	no
Support stored procedures	no (although you can use ADO.NET methods or Named Queries, their use is not so easy)	yes (configuring xml files and generating wrapper classes)	yes (using Named Queries)
Tested	yes	yes	no
Open source	yes	yes	yes
Databases supported	The same that NHibernate	Microsoft SQL Server 2005/2000, MySQL, Oracle	Microsoft SQL Server 2005/2000, Oracle, Microsoft Access, Firebird, PostgreSQL, DB2 UDB, MySQL, SQLite

Appendix B

Equipment Database: implementation details

This chapter presents implementation details of Equipment Database application. Although Equipment Database was not the focus of the project, the technical achievements are useful for any development project.

Section B.1 describes the structure of the system in more detail, while section B.2 defines its interfaces. Section B.3 presents used technologies and potential alternatives. Section B.4 explains security aspects of the system. Finally section B.5 shows some implemented architecture patterns.

B.1 System structure

This section describes, component by component, its responsibilities, applied techniques and used technologies.

B.1.1 Horizontal layers

Data and Storage Management

The data and storage management layer is responsible to save and retrieve persistent data, in a relational schema. It will be ensured by a database system, Microsoft SQL Server 2005, which provides data consistency, quickly access, security, transactions and concurrency support.

The database should not include any business logic avoiding, as possible, to build database objects, stored procedures or product-specific features. The upper layers should communicate with it using standard SQL, easing interchangeability to others databases.

Data Access

The data access layer is a bridge between the data layer and business logic. It provides a meaningful interface to help mapping objects to tables and to abstract the data layer, easing the change of database.

The Equipment Database access layer uses two related object-relational mapping tools: Castle ActiveRecord and NHibernate. Castle ActiveRecord is built on top of NHibernate. At the other hand, NHibernate is based on ADO.NET, the default access layer of .NET framework (figure B.1).

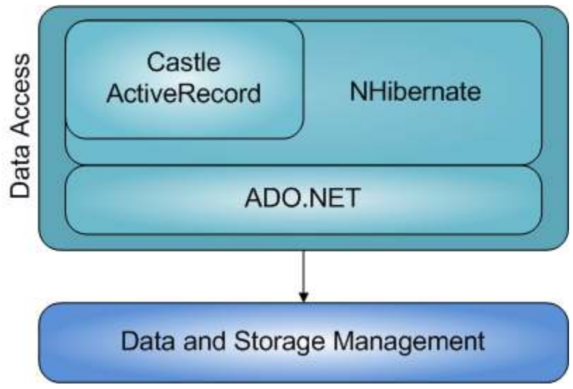


Figure B.1: Data access layer components

Castle ActiveRecord provides an interesting way to map objects to tables, using attributes, going together with a set of useful methods to save and retrieve data, in a very object-oriented way. It helps the developer focus on business logic, relying on ActiveRecord to ensure the persistence. However, to implement more complex queries, knowledge of NHibernate was required.

Business Logic

The business logic contains the knowledge about company organization and operations, providing high-level services to upper layers.

The business logic is composed by a web service, a façade and business objects (figure B.2). The web service exposes business services, which are transactional and stateless. The lightweight façade coordinates business objects to provide a stateless interface to the web service. Ideally, the façade would provide the same interface than web service, so a component can directly access it to improve performance. Business objects represent the knowledge about the business, representing the entities, their behavior and relationships that make part of it. They were implemented following WARP approach and using Business#.

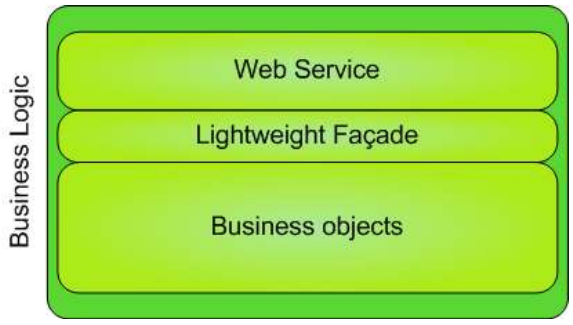


Figure B.2: Business logic layer components

The lightweight façade is group of stateless and atomic methods, which instantiates and coordinates the business objects. The web service exposes the façade to external and independent components, using the ASP.NET web services technology.

User Interface

User interface is an independent component, communicating with business layer using SOAP protocol. It means that it is easy create new and different graphical user interfaces in Equipment Database. For a proof-of-concept, it was implemented a web application, using ASP.NET framework. However WebForms methodology was discarded, in favor of a promising Model-View-Controller (MVC) framework, Castle MonoRail.

An MVC framework helps the separation in models, views and controllers. In Equipment Database, models mean services provided by business logic layer. Views define how the information is displayed. And controllers manage the interaction with the user, views and models, coordinating the page flow. So, in this context, the user interface is composed by controllers. Section 6.2.2 explains in more detail Castle MonoRail working model.

Presentation

Presentation layer define how to render the information to the user. This layer is usually tightly coupled with user interface, but the used MVC pattern helps creating a more independent component. The presentation layer is composed by views, images, JavaScript and CSS documents. The views contain a mix of XHTML 1.0 and rendering instructions processed by a view engine, NVelocity, generating a browser compliant file.

B.1.2 Vertical layers

Vertical layers components have strong connection with functional requirements and use cases described in [120]. So the description here is merely contextual and do not intends to be a full definition. Vertical layers cross all horizontal layers, namely in the following cases:

- In data and storage management component, they create relational tables to persist data;
- They use the data access layer to connect with databases;
- They use the business model to create stateless and atomic methods, exposable through web services;
- The controllers call the business logic, to provide the services in a user interface;
- The views are processed to show information returned by the services.

Equipment Service

Equipment service provides a set of methods to consult and manage equipments, including clustering, searching, and support for state models.

Model Service

Model service provides a set of methods to consult and manage models.

Vendor Service

Vendor service provides a set of methods to consult and manage vendors.

Equipment Type Service

Equipment type service provides a set of methods to consult and manage equipment types, including adding and removing attributes.

Unit Processing Type Service

Unit processing type service provides a set of methods to consult and manage unit processing types.

Person Service

Person service provides a set of methods to consult and manage persons, which can be responsible for equipments.

Authentication

The authentication guarantees that only permitted users accede to the system. Equipment Database relies on Integrated Windows Authentication (IWA) to authenticate users based on current account running and configured users in the machine. Because it uses IWA, it provides an easy way to integrate with existent active directories that manage company Windows accounts.

The web service performs the authentication, allowing ASP.NET impersonation, a feature to control the identity under which code is executed. This is a technique that provides to clients of web services, e.g. the web application, authenticate users and pass its data to web services. That means maintain the authentication control on the web service and simplify the implementation of authentication through upper layers.

Role Management

The role management verifies if the current user is allowed to execute the requested service. In Equipment Database, it is implemented using ASP.NET Role Management on the web service. The web service defines, for each method, permissions. These permissions can be defined using attributes or calling methods to check user roles. The roles and users are stored in the database, using a custom role provider.

Integrated Windows Authentication and impersonation ensures that role management is always performing about the user that runs in the client.

Logging

There are two types of loggers in the system: general and specific for equipments. The former records all business operations performed in a text file, and it was designed due security reasons. The second stores all operations over equipments in the database, to maintain their history.

The implementation of general logging could be simple, but tedious. Add a call to the logger in all operations is a manual task and it is hard to maintain. Because that, the system uses an aspect that adds a log per any method call of business classes. For that reason, it was used an aspect-oriented programming (AOP) framework, named PostSharp Laos. The logger was implemented using the log4net framework.

Equipment logging was not implemented, but it also would use log4net, with unique messages for each operation.

B.2 Interfaces

This section presents Equipment Database interfaces between components. Due to the fact that it is research project the interfaces are loosely defined. An interesting point to look on interfaces is that they usually occur between horizontal layers, to build a vertical layer. The interfaces related with security concerns are described in section B.4.

B.2.1 Internal interfaces

Business Objects - Data access layer

A business object interacts with data access layer using the .NET API (Application Programming Interface) exposed by Castle ActiveRecord and NHibernate, by object composition or class inheritance (figure B.3).

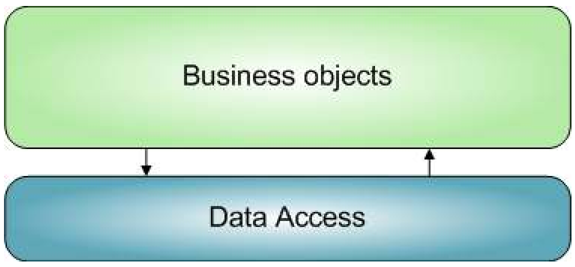


Figure B.3: Interface between business objects and data access components

Lightweight Façade - Business Objects

The lightweight façade instantiates business objects to invoke methods exposed by them, providing a transactional and stateless interface to the upper layers (figure B.4).

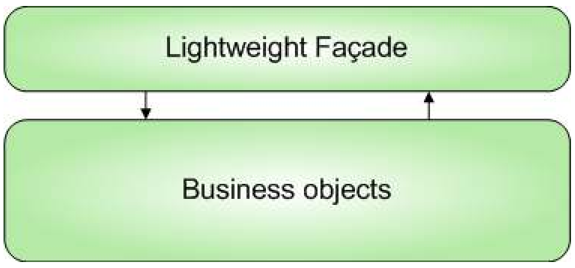


Figure B.4: Interface between lightweight façade and business objects components

The interface exposed by business objects represents logical entities and their relationships, which are related with company domain. Figure B.5 shows the domain model of Equipment Database.

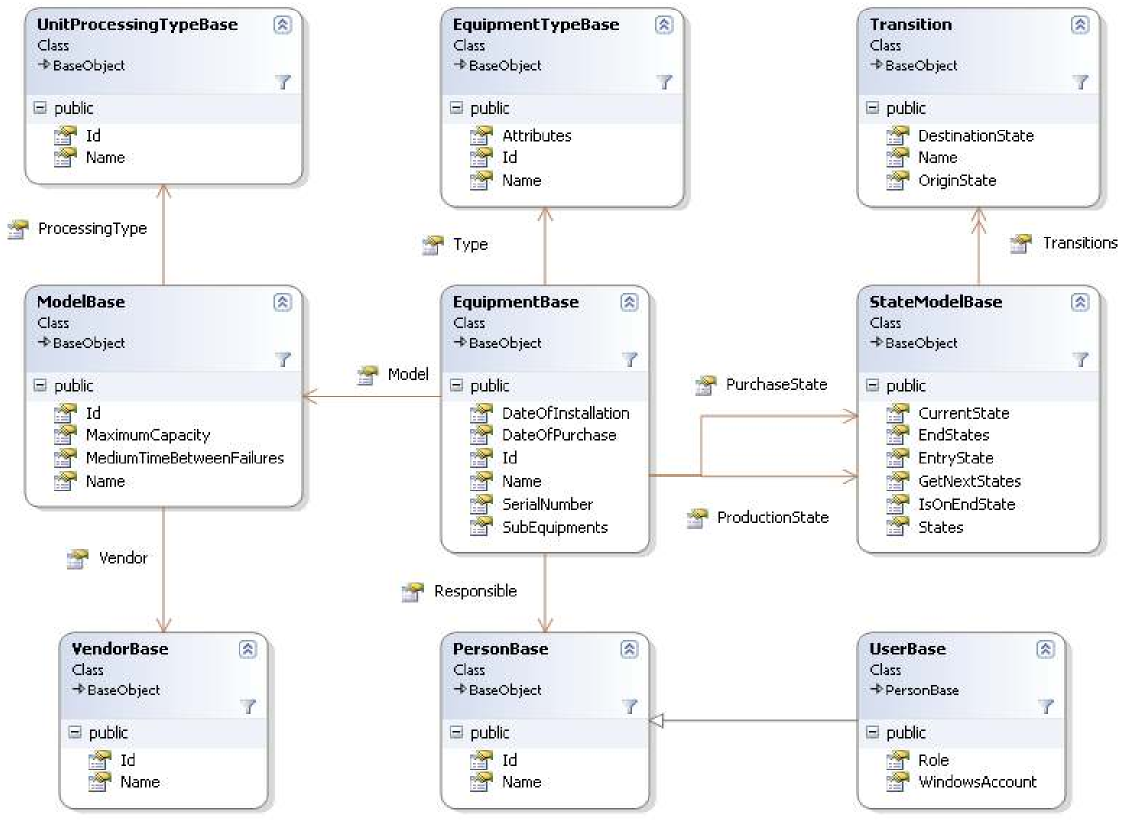


Figure B.5: Equipment Database business model

Web Service - Lightweight Façade

The web service uses the lightweight façade API to implement its services, avoiding implement any logic in the web service. Ideally, the interface exposed by the web service is equal to interface exposed by façade, in a one-to-one relationship (figure B.6).

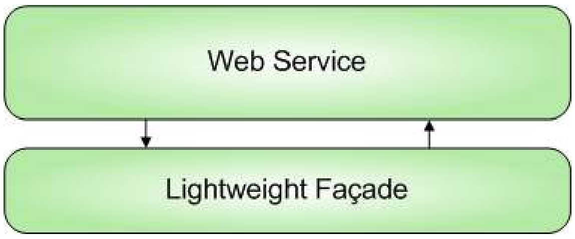


Figure B.6: Interface between the web service and lightweight façade components

Web Application - Web Service

The web application communicates with the web service using ASP.NET web services infrastructure. The underlying protocol is SOAP, which enables any application able to establish HTTP connections and parse XML to be compliant with the web service (figure B.7).

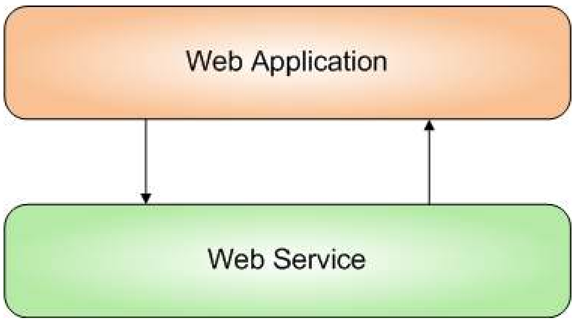


Figure B.7: Interface between web application and web service components

B.2.2 External interfaces

Data access layer - Data and storage management layer

The data access layer calls the data storage management layer via available protocols by adopted database (figure B.8). In case of SQL Server 2005 it connects using shared memory, named pipes or TCP/IP. The data access layer produces compatible SQL to store and retrieve data in the current database. But it can be configured to generate SQL compliant with other databases.

Web Client - Web Application

The web client is a browser that communicates through HTTP protocol (figure B.9). The web application generates XHTML 1.0 files, which may include standard JavaScript, CSS 2.0 and multimedia files.

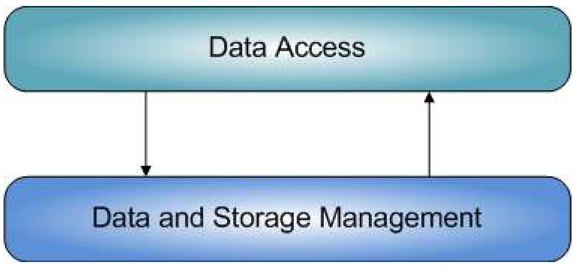


Figure B.8: Interface between data access and data storage components

B.3 Technologies

This chapter describes technologies used in Equipment Database project, explaining their adoption and potential alternatives.

B.3.1 Microsoft-based Development Environment

The chosen development environment is based in Microsoft tools, namely Microsoft Visual Studio Team System and .NET framework 2.0. That follows an internal company strategy, which believes these tools would be the better to accomplish organization objectives, favoring technological specialization.

Additionally the project was a special interest to study specific software engineering tools and get practical conclusions. It makes sense study tools targeted for the principal company development environment. For example, the study of object-relational mapping, scaffolding, features of Visual Studio Team System for Testers, code analysis, and aspect-oriented programming.

Potential alternatives

The potential alternatives, which can compete in maturity, quality and completeness, are scarce. The first alternative would be Java framework, using Eclipse as integrated development environment. Another possibility is use MonoDevelop or SharpDevelop, development environments based on open-source project Mono, which tries to create a .NET framework that runs in different operating systems. All these alternatives are open-source, without any license costs.

B.3.2 Programming language: C#

The choice of adopted programming language usually has great impact, not only on programmer productivity but also on many design issues. However in the case of .NET framework this is not a problem, because it abstracts the programming language to an intermediate language, Microsoft IL. So it is possible create components in different languages. In fact all generated assemblies in the project are compatible to be called by different .NET languages. The C# was chosen mainly because:

- Well known language;
- Extensively documented;

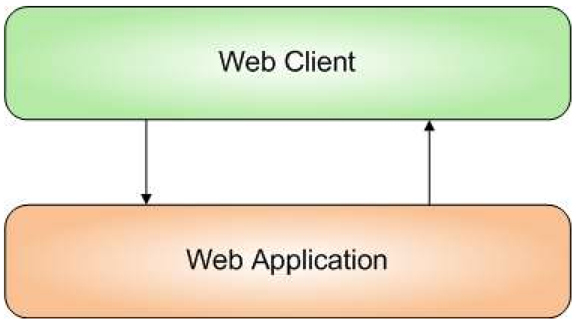


Figure B.9: Interface between web client and web applicant components

- Easy from who migrates from other well known languages like C++ and Java;
- Code cleanness;
- Personal experience.

Potential alternatives

Nowadays there are a few alternative languages targeted to .NET environment. Despite well known and supported VB.NET and Managed C++, there are other new languages appearing like Boo, Spec# and F#, and Microsoft encourages that with initiatives like Phoenix Academic Program [138].

B.3.3 Business# Framework

Equipment Database serves as proof of concept of WARP solution and Business# framework, to evaluate their applicability and utility in a real software project.

B.3.4 Microsoft SQL Server 2005

Microsoft SQL Server 2005 was chosen because it is easier to deploy and configure in development environment. It is a know database management system, well documented and has good support in the industry. Moreover it has good management and tuning tools, like SQL Server Management Studio and SQL Server Profiler.

Potential alternatives

The natural choice for a database management system was Oracle systems, widely used by company software. However the utilization of an object-relational mapping tool minimizes this decision. Despite a small experience was performed to check compatibility with Oracle databases.

B.3.5 Castle ActiveRecord/NHibernate

Castle ActiveRecord is an open source object-relational mapping tool, connecting classes and relational schemas through programmatic attributes, and providing database interchangeability. It relies on NHibernate, but avoids the creation of metadata as XML, which usually is more difficult to learn and to maintain, and provides a set of useful methods to save and retrieve data.

Castle ActiveRecord is very intuitive for programmers with object-oriented background, but can be strange for who has more database knowledge: the way your work with your persistent objects is really simple and transparent. Moreover, it includes advanced features as lazy loading and data validation. However Castle ActiveRecord does not provide all functionality and it is expected use NHibernate to implement complex queries.

Potential alternatives

The principal advantage of Castle ActiveRecord is starting from domain model to generate the relational schema, unlike other tools that go from relational schema (SubSonic) or only help doing the mapping (ADO.NET Entity Framework). Other options would be using NHibernate only, building XML metadata manually or with some available tools, or do not use any object-relational mapping tool, performing accesses directly to ADO.NET.

B.3.6 ASP.NET

ASP.NET is a web application framework developed by Microsoft, to build dynamic web sites and web services. Equipment Database uses ASP.NET to implement web services and to perform authentication and role management; however it does not use its famous Web Forms development model to create web sites but uses Castle MonoRail.

B.3.7 Castle MonoRail

Castle MonoRail is an open-source framework, that allows the creation of web user interfaces according the Model-View-Controller pattern. Its working mode is described in section 6.2.2. As it is a promising tool, integrated with Castle ActiveRecord, seems natural choose it to implement the Equipment Database project.

Potential alternatives

The obvious alternative is the utilization of Web Forms development model. However MVC frameworks had been supporting by more and more experts. The major reason is because it improves the system testability, but also because provide higher separation of concerns. Microsoft had recognized and had launched an ASP.NET MVC framework, but it is under development (at the moment, it is in version Preview 3).

B.3.8 NVelocity

NVelocity is the used view engine used by Castle MonoRail to render the contents back to the browser, that is, to render views. It is a port of Velocity, used in Java projects, and provides a language named Velocity Template Language (VTL), with simple operators to display logic (e.g. iterators and logical operators). It is chosen because it has good documentation, with many samples, and it is easy to learn.

Potential alternatives

Castle MonoRail has several other available view engines: WebForms, Brail, AspView and StringTemplate. In fact, you could implement your view engine. See [139] for a discussion about these alternatives.

B.3.9 PostSharp Laos

In Equipment Database PostSharp Laos was used to provide general logging, registering all calls to every methods of business model. For more about PostSharp consult sections 7.3.1 and A.1.7.

Potential alternatives

Aspect oriented programming has not a very good tool in .NET environment, as Java has AspectJ [140]. There are the research and promising project Aspect.NET and dead Aspect# project, but both are immature and lack many functions.

B.3.10 Apache Log4net

Log4net [141] is a framework to help the programmer output log statements to a variety of output targets. It is port of log4j to the .NET runtime. It is chosen due its simplicity, extensibility and thread-safety.

Potential alternatives

There are a lot of logging frameworks, which the most famous for .NET runtime would be the Logging block of Enterprise Library. However it is complex to learn and introduce some dependencies.

B.3.11 jQuery

Creating usable and simple web applications is not easy only with HTML. JavaScript is a powerful language to add some functionality in the client side but it does not provide a richer API. jQuery [142] is a library with many built-in effects and animations. Equipment Database has used jQuery Datepicker to allow a more friendly selection of dates by user.

B.4 Security aspects

The security aspects are mainly related with authentication and role management. They are based on following Microsoft technologies:

- Integrated Windows Authentication (IWA);
- ASP.NET impersonation;
- ASP.NET role management and a custom role provider.

The Integrated Windows Authentication provides a way to integrate the application authentication with Windows authentication. ASP.NET impersonation allows change the user which the code is executed. And ASP.NET role management helps adding special permissions for each method, based on roles. Role management uses a custom role provider to get and change the user's roles (figure B.10).

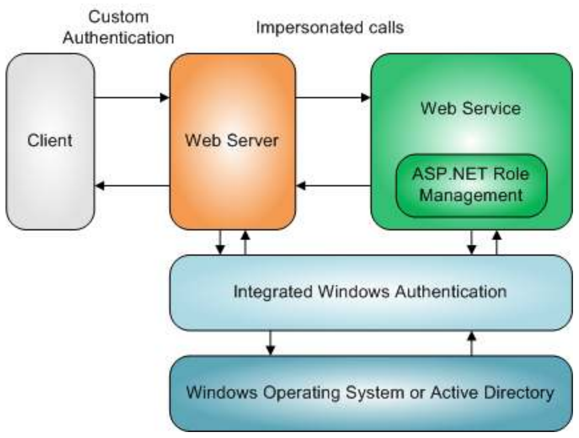


Figure B.10: Equipment Database security components

The authentication guarantees that only permitted users accede to the system. Equipment Database relies on Integrated Windows Authentication (IWA) to authenticate users based on current account running and configured users in the machine. Because it uses IWA, it provides an easy to integrate with existent active directories that manage company Windows accounts.

The web service performs the authentication, allowing ASP.NET impersonation, a feature to control the identity under which code is executed. This is a technique that provides to clients of web services, e.g. the web application, authenticate users and pass its data to web services. That means maintain the authentication control on the web service and simplify the implementation of authentication through upper layers.

The role management controls if current user is allowed to execute the requested service. In Equipment Database, it is implemented using ASP.NET Role Management on the web service. The web service defines, for each method, permissions. These permissions could be defined using attributes or calling methods to check user roles. The roles and users are stored in the database, using a custom role provider.

Integrated Windows Authentication and impersonation ensures that role management is always performed about the user that runs in the client.

B.5 Patterns of Enterprise Application Architecture

A well-designed system is commonly rich of patterns, rules-of-thumbs that guide the resolution of common problems. There are a lot of published patterns on software industry, namely about system architecture and design. They are very useful, because transmit knowledge about techniques applied in previous systems that could work your new application. They should not be abused. They should only be used when really needed.

This chapter aims to describe the architecture patterns of Fowler's catalog [12] identified in Equipment Database, after their implementation, to resume learned lessons and to teach the reader more about these patterns. Excluded patterns do not mean they are not there: just means they are not identified or considered relevant.

B.5.1 Domain Model

The domain model captures the behavior and relationships of entities related with business where building system would operate. It is composed by objects, where each one is an abstract representation of a real element, like a medicament prescription or a customer profile. These objects should be sufficient abstract and hide system specificities: it would be much preferable work with a meaningful object than with a relational record in a database.

Equipment Database project was focused to look the way we get a domain model and how it is mapped in the implementation. It also aims advocating its advantages: captures the business logic, raises its understandability in all stages of software development, and favors reusing and maintainability. The domain model of Equipment Database is concretized in Business Objects component.

B.5.2 Service Layer

The service layer operates over domain model to provide a set of available operations, responding to other components, as user interfaces and external applications. It defines a boundary, encapsulating the business logic, which increases the component decoupling and interchangeability. Equipment Database creates a service layer to achieve these objectives.

B.5.3 Active Record

An active record is an object that wraps a record in a database table or view, encapsulating the database access, and adding business logic on that data. This pattern was applied on Equipment Database using a framework targeted to do this task: Castle ActiveRecord.

B.5.4 Lazy Load

An object that supports lazy loading do not holds all the data, but knows how to get it. This pattern is particularly useful when we have complex objects, which persists their data in several different resources, like different database tables. To increase performance, the object only retrieves the data when it needs it. The Castle ActiveRecord supports lazy loading easily, configured through attributes. Currently the system does not use this feature but could be enabled with small effort.

B.5.5 Data Mapper

A data mapper is layer that separates the in-memory objects and persistency repositories, namely databases. It tries smoothing problems as object-relational impedance mismatch [100], caused by different representations, for instance the standard relational schemas have not inheritance neither pointers.

Equipment Database uses the data mapper NHibernate, which transfers data between objects and a database, keeping them independent of each other. However Castle ActiveRecord abstracts the mapping process of NHibernate, making its use practically transparent. In fact, Castle ActiveRecord can also generate the database schema, hiding to the programmer many details of the data layer.

B.5.6 Identity Field

Objects are usually distinguished by reference while table rows by identifiers, also known by primary keys, making harder map them. This pattern recommends saving the database identifier in the in-memory object to maintain identity. Equipment Database uses this pattern, somewhat because one component of the persistency layer, Castle ActiveRecord, enforces its application.

B.5.7 Foreign Key Mapping

This pattern suggests mapping an association between objects to a foreign key reference between tables, ensuring data integrity on the database and simplifying the management of related data in the database, as performance tuning or cascade deletes. Equipment Database uses this pattern, somewhat because one component of the persistency layer, Castle ActiveRecord, recommends its application.

B.5.8 Query Object

A query object represents a database query, in an object oriented style, hiding to the programmer SQL language details and existent relational schema. That increases maintainability and interchangeability.

Equipment Database uses a namespace of NHibernate, NHibernate.Criterion, to implement complex queries over the mapped business model.

B.5.9 Model View Controller

The model view controller (MVC) is a popular presentation pattern, which advocates the separation of user interface in three distinct roles: View, Controller and Model (figure B.11). The view defines how to render the information to the user. The model characterizes the entity data and behavior. And the controller processes and responds to events, using models and views.

Relationships between these three types of components can differ significantly in distinct implementations.

Equipment Database uses a framework targeted to the implementation of this pattern to the web: Castle MonoRail. Views were written using Velocity Template Language (VTL) and XHTML 1.0, and controllers are classes that inherit from MonoRail framework classes. The model is the service layer.

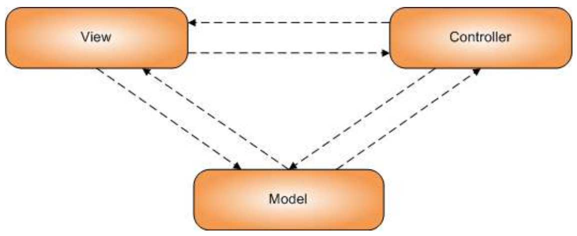


Figure B.11: Model View Controller pattern

B.5.10 Remote Façade

A remote façade transforms a fine-grained interface, desirable in oriented object paradigm (OOP) due its flexibility and extendibility, to a coarse-grained interface, minimizing the calls between the layers, due performance reasons. It should not add any functionality, but only instantiate, coordinate and invoke underlying fine-grained objects.

Equipment Database has an instance of this distribution pattern inside the business logic (figure B.12). The lightweight façade defines a interface fitted to remote calls, using the business objects.

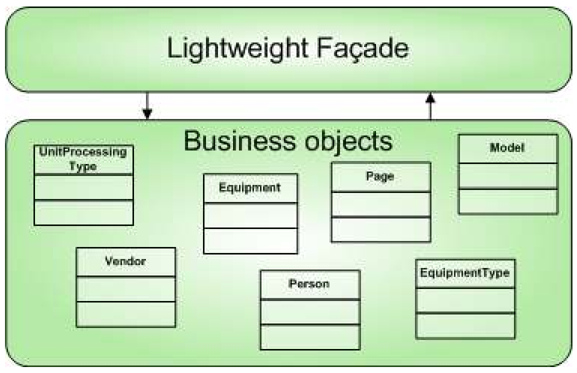


Figure B.12: Remote Façade in Equipment Database

B.5.11 Data Transfer Object

A data transfer object carries data between processes in order to reduce the number of method calls. When we need to transmit data with a remote interface, like a remote façade, we should reduce the number of calls. So, the data is encapsulated in an object, avoiding pass multiple parameters and resolving the limitations of some languages that only permits return single values, like Java or C#. This object needs to be serializable to be transmitted over connections.

Equipment Database makes use of these objects intensively, namely in the web service. The ASP.NET web services framework generates stub objects that clients use to pass data and the web services to return the results.

One interesting example is PageSimple class (figure B.13). It returns a page of objects, for instance a list of equipments. It is used to perform pagination in database, passing to user interface the list of objects (Page), the number of the page (CurrentPage) and the total number of items in the database (TotalItems).

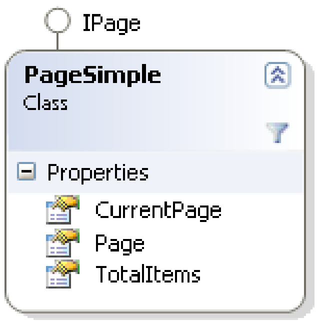


Figure B.13: PageSimple interface

B.5.12 Layer Supertype

A layer supertype is a type that is parent of all types in its layer (or most of them). It is useful when these types have similar behaviors. Equipment Database uses this pattern extensively, as presented in figure B.14.

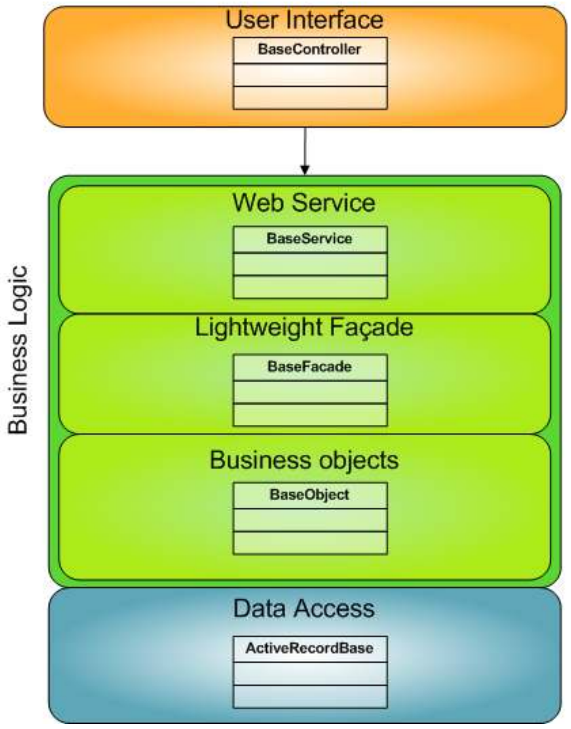


Figure B.14: Layer supertypes in Equipment Database

B.5.13 Gateway

A gateway is an object that encapsulates access to an external system or resource, usually involving multiple processes and machines, or communication through messages.

The web service in Equipment Database can be considered a gateway, where several different clients can connect to accede to business logic. The data access layer can be considered a gateway too, which encapsulates the database.

B.5.14 Separated Interface

The separated interface pattern advocates the creation of an interface in a separate package from its implementation. Equipment Database project was tightly connected with this pattern, due to WARP solution defends this approach.

References

[1] Unknown author. The Seven Basic Principles of the Context-Driven School. http://www.context-driven-testing.com. [Online; accessed 11-June-2008].

[2] Ayende Rahien. *Building Domain Specific Languages in Boo*. Manning Publications Co. http://www.manning.com/rahien/, unedited draft edition, 2008. [Online; accessed 17-June-2008].

[3] Standish Group. *Chaos Charting the Seas of Information Technology*. The Standish Group International, West Yarmouth: MA, 1996.

[4] António Miguel. *Gestão de Projectos de Software*. FCA, segunda edição actualizada edition, 2006.

[5] Steven D. Schafersman. An Introduction to Science - Scientific Thinking and the Scientific Method. http://www.freeinquiry.com/intro-to-sci.html, January 1994. [Online; accessed 11-June-2008].

[6] Tom Poppendieck Mary Poppendieck. *Implementing Lean Software Development - From Concept to Cash*. Addison-Wesley Professional, 2006.

[7] Martin Fowler. Cannot Measure Productivity. Martin Fowler Bliki http://martinfowler.com/bliki/CannotMeasureProductivity.html, 2003. [Online; accessed 16-June-2008].

[8] Kailash Awati. Communication impedance mismatches on projects. Eight to Late http://eight2late.wordpress.com/2008/06/07/communication-impedance-mismatches-on-projects/, 2008. [Online; accessed 25-June-2008].

[9] Wikipedia Contributors. Software development process. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Software_development_process, 2008. [Online; accessed 16-June-2008].

[10] Karl E. Wiegers. *Software Requirements*. Microsoft Press, second edition edition, 2003.

[11] Software Engineering Institute. How Do You Define Software Architecture? Carnegie Mellon http://www.sei.cmu.edu/architecture/definitions.html, 2008. [Online; accessed 18-June-2008].

[12] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[13] Gregor Hohpe. Patterns and Best Practices for Enterprise Integration. http://www.enterpriseintegrationpatterns.com/. [Online; accessed 18-June-2008].

[14] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[15] Ian Sommerville. *Software Engineering*. Pearson Education Limited, seventh edition edition, 2004.

[16] Jiantao Pan. Software testing. Carnegie Mellon University http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/, 1999. [Online; accessed 18-June-2008].

[17] Wikipedia Contributors. Software testing. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Software_testing, 2008. [Online; accessed 19-June-2008].

[18] Antonio Carzaniga et all. A Characterization Framework for Software Deploymnet Technologies. http://www.inf.unisi.ch/carzaniga/papers/CU-CS-857-98.pdf, 2008. [Online; accessed 18-June-2008].

[19] Winston W. Royce. Managing the development of large software systems. *Proceedings of IEEE WESCON*, 1970. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf [Online; accessed 19-June-2008].

[20] Manfred Rieck. Project (Change) Management News from Germany. PM World Today http://www.pmforum.org/library/RegionalReports/2008/PDFs/Rieck-5-08.pdf, 2008. [Online; accessed 20-June-2008].

[21] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. IEEE Computer http://www.cs.usu.edu/~supratik/CS5370/r5061.pdf, 1988. [Online; accessed 19-June-2008].

[22] Wikipedia Contributors. Spiral model. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Spiral_model, 2008. [Online; accessed 19-June-2008].

[23] Wikipedia Contributors. IBM Rational Unified Process. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process, 2008. [Online; accessed 19-June-2008].

[24] Ward Cunningham et all. Manifesto for Agile Software Development. http://agilemanifesto.org, 2001. [Online; accessed 20-June-2008].

[25] Softhouse Consulting. Scrum in five minutes. http://www.softhouse.se/Uploades/Scrum_eng_webb.pdf, 2006. [Online; accessed 20-June-2008].

[26] Wikipedia Contributors. Scrum (development). Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Scrum_(development), 2008. [Online; accessed 20-June-2008].

[27] Scott Ambler. Supersize Me. Dr. Dobb's Portal http://www.ddj.com/architect/184415491, 2006. [Online; accessed 20-June-2008].

[28] Scott Ambler. Bridging the Distance. Dr. Dobb's Portal http://www.ddj.com/architect/184414899, 2002. [Online; accessed 20-June-2008].

[29] Wikipedia Contributors. Agile software development. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Agile_software_development, 2008. [Online; accessed 20-June-2008].

[30] Geoffrey Lory et all. Microsoft Solutions Framework version 3.0 Overview. http://www.microsoft.com/technet/solutionaccelerators/msf/default.aspx, 2003. [Online; accessed 20-June-2008].

[31] Tim Dysinger Lasse Mirkovic. Cowboy coding. Cunningham & Cunningham, Inc. http://c2.com/cgi-bin/wiki?CowboyCoding, 2007. [Online; accessed 19-June-2008].

[32] Tom Hormby. How Adobe's Photoshop Was Born. Internet Archive http://web.archive.org/web/20070717193315/http://siliconuser.com/?q=node/10, 2007. [Online; accessed 19-June-2008].

[33] Gary Hamel. *O Futuro da Gestão, como a gestão 2.0 ultrapassará mentalidades que limitam a inovação estratégica.* Actual Editora, 2007.

[34] Wikipedia Contributors. Capability Maturity Model Integration. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/CMMI, 2008. [Online; accessed 23-June-2008].

[35] Personal Software Process (PSP). Software Engineering Institute, Carnegie Mellon http://www.sei.cmu.edu/tsp/psp.html. [Online; accessed 23-June-2008].

[36] Team Software Process (TSP). http://www.sei.cmu.edu/tsp/tsp.html. Software Engineering Institute, Carnegie Mellon [Online; accessed 23-June-2008].

[37] Google News. http://news.google.com/. [Online; accessed 21-June-2008].

[38] Gmail. http://mail.google.com/. [Online; accessed 21-June-2008].

[39] Google Docs. http://docs.google.com/. [Online; accessed 21-June-2008].

[40] Google Labs. http://labs.google.com/. [Online; accessed 21-June-2008].

[41] NDepend. http://www.ndepend.com/. [Online; accessed 23-June-2008].

[42] NCover. http://www.ncover.com/. [Online; accessed 23-June-2008].

[43] Project Homepage Microsoft Office Online. http://office.microsoft.com/project. [Online; accessed 22-June-2008].

[44] Mingle: Team Collaboration and Project Management Made Easy. ThoughtWorks Studios http://studios.thoughtworks.com/mingle-project-intelligence. [Online; accessed 22-June-2008].

[45] Apache Ant. http://ant.apache.org/. [Online; accessed 22-June-2008].

[46] MSBuild Reference. http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx. [Online; accessed 22-June-2008].

[47] FitNesse Acceptance Testing Framework. http://fitnesse.org/. [Online; accessed 22-June-2008].

[48] NUnit. http://www.nunit.org/. [Online; accessed 22-June-2008].

[49] Wikipedia Contributors. List of unit testing frameworks. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks, 2008. [Online; accessed 18-June-2008].

[50] Selenium. http://selenium.openqa.org/. [Online; accessed 22-June-2008].

[51] WatiN, Web Application Testing in .NET. http://watin.sourceforge.net/. [Online; accessed 22-June-2008].

[52] Visual Studio Team System Test Edition. http://msdn.microsoft.com/en-us/library/ms182409.aspx. [Online; accessed 22-June-2008].

[53] FxCop. http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx. [Online; accessed 22-June-2008].

[54] Wikipedia Contributors. Integrated Collaboration Environment. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Integrated_Collaboration_Environment, 2008. [Online; accessed 23-June-2008].

[55] Process Templates and Tools. MSDN Team Suite Developer Center http://msdn.microsoft.com/en-us/vsts2008/aa718795.aspx. [Online; accessed 23-June-2008].

[56] Martin Fowler. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html, 2006. [Online; accessed 23-June-2008].

[57] TIOBE Programming Community Index for June 2008. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2008. [Online; accessed 24-June-2008].

[58] Wikipedia Contributors. Very high-level programming language. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Very_high-level_programming_language, 2008. [Online; accessed 24-June-2008].

[59] Wikipedia Contributors. Programming paradigm. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Programming_paradigm, 2008. [Online; accessed 24-June-2008].

[60] Jack W. Reeves. What Is Software Design? C++ Journal and available in http://www.developerdotstar.com/mag/articles/reeves_design.html, 1992. [Online; accessed 17-June-2008].

[61] Unknown author. Visual Studio 2005 Team System Modeling Strategy and FAQ. Microsoft Visual Studio 2005 Technical Articles http://msdn.microsoft.com/en-us/library/ms379623(VS.80).aspx, 2005. [Online; accessed 17-June-2008].

[62] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition edition, 2004.

[63] patterns & practices. Microsoft patterns & practices Developer Center http://msdn.microsoft.com/pt-br/practices/default(en-us).aspx, 2008. [Online; accessed 24-June-2008].

[64] Guidance Automation Extensions and Guidance Automation Toolkit. Team Suite Developer Center http://msdn.microsoft.com/en-us/vsts2008/aa718948.aspx, 2008. [Online; accessed 24-June-2008].

[65] Jason Gorman. Post-Agilism - Beyond the Shock of the New. ITarchitect http://www.itarchitect.co.uk/articles/display.asp?id=280, 2006. [Online; accessed 19-June-2008].

[66] Wikipedia Contributors. Model-driven engineering. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Model_Driven_Engineering, 2008. [Online; accessed 25-June-2008].

[67] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[68] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. *ECOOP 2001*, 2001.

[69] Rational Rose Product Line. http://www-306.ibm.com/software/awdtools/developer/rose/index.html, 2008. [Online; accessed 26-June-2008].

[70] Generative Modeling Technologies project. http://www.eclipse.org/gmt/, 2008. [Online; accessed 25-June-2008].

[71] Unknown author. Working with Visual C++ Code in Class Designer. Microsoft Visual Studio 2008 Developer Center http://msdn.microsoft.com/en-us/library/bb385735.aspx, 2008. [Online; accessed 17-June-2008].

[72] Carlos Videira Alberto Silva. *UML - Metodologias e Ferramentas CASE*. Centro Atlântico, Rua da Misericórdia ,76 - 1200-273 Lisboa - Portugal, segunda edição edition, 2005.

[73] Wikipedia Contributors. Unified Modeling Language. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Unified_Modeling_Language, 2008. [Online; accessed 26-June-2008].

[74] Microsoft Office Visio. http://office.microsoft.com/visio, 2008. [Online; accessed 26-June-2008].

[75] Windows Workflow Foundation. http://netfx3.com/content/WFHome.aspx, 2007. [Online; accessed 26-June-2008].

[76] José Formiga. As vantagens reais do Windows Workflow Foundation(WF). José Formiga's Blog http://mdalab.wordpress.com/2007/11/11/as-vantagens-reais-do-windows-workflow-foundation-wf/, 2007. [Online; accessed 26-June-2008].

[77] OMG Object Management Group. Business Process Modeling Notation, V1.1. http://www.omg.org/docs/formal/08-01-07.pdf, 2008. [Online; accessed 17-June-2008].

[78] Relo. Massachusetts Institute of Technology http://relo.csail.mit.edu, 2007. [Online; accessed 26-June-2008].

[79] OMG Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf, 2003. [Online; accessed 17-June-2008].

[80] José L. Alemán Ambrosio Toval, Victor Requena. OCL Tools. Ingeniería del Software, Universidad de Murcia http://www.um.es/giisw/ocltools/, 2006. [Online; accessed 17-June-2008].

[81] Hassan Charaf László Lengyel, Tihamér Levendovszky. Implementing an OCL Compiler for .NET. *.NET Technologies'2005*, 2005. http://dotnet.zcu.cz/NET_2005/Papers/Full/A19-full.pdf [Online; accessed 27-June-2008].

[82] Ana Paiva. Model-based testing. Teste e Qualidade de Software, Faculdade de Engenharia da Universidade do Porto http://paginas.fe.up.pt/~jpf/teach/TQS0708/ModelBasedTesting.pdf, 2007. [Online; accessed 25-June-2008].

[83] Spec Explorer. Microsoft Research http://research.microsoft.com/specexplorer/, 2008. [Online; accessed 28-June-2008].

[84] Martin Fowler. Domain Specific Language. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html. [Online; accessed 27-June-2008].

[85] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html, 2005. Work in progress book [Online; accessed 28-June-2008].

[86] Visual Studio SDK. Visual Studio Extensibility Developer Center http://msdn.microsoft.com/en-us/library/bb166441.aspx. [Online; accessed 27-June-2008].

[87] Domain Builder. CodePlex http://www.codeplex.com/domainbuilder, 2008. [Online; accessed 25-June-2008].

[88] Rockford Lhotka. *Expert C# 2005 Business Objects*. Apress, second edition edition, 2006.

[89] NHibernate for .NET. http://www.hibernate.org/343.html, 2008. [Online; accessed 27-June-2008].

[90] Martin Fowler. *Domain Specific Languages*. 2008. Work in progress book, http://martinfowler.com/dslwip/ [Online; accessed 27-June-2008].

[91] Wikipedia Contributors. Design by contract. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Design_by_contract, 2008. [Online; accessed 27-June-2008].

[92] Eiffel Software. http://www.eiffel.com/, 2008. [Online; accessed 27-June-2008].

[93] Contract4J: Design by Contract for Java. http://www.contract4j.org/, 2008. [Online; accessed 25-June-2008].

[94] Yoonsik Cheon Gary T. Leavens. Design by Contract with JML. 2006. http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf [Online; accessed 27-June-2008].

[95] Wikipedia Contributors. Business rules engine. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Business_rules_engine, 2008. [Online; accessed 26-June-2008].

[96] Federico Balaguer and Joseph W. Yoder. Adaptive Object-Model Architecture. *OOPSLA 2001*, 2001. http://www.adaptiveobjectmodel.com/OOPSLA2001/AOMoopsla2001Tutorial.pdf [Online; accessed 26-June-2008].

[97] Martin Fowler. Separated Interface. Catalog of Patterns of of Enterprise Application Architecture http://martinfowler.com/eaaCatalog/separatedInterface.html, 2008. [Online; accessed 4-July-2008].

[98] Martin Fowler. Mocks Aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html, 2007. [Online; accessed 29-June-2008].

[99] Christian Bauer Pierre Henri Kuaté and Gavin King. *NHibernate in Action*. Manning Publications Co. http://www.manning.com/kuate/, unedited draft edition, 2007.

[100] Scott W. Ambler. The Object-Relational Impedance Mismatch. Ambysoft Inc. http://www.agiledata.org/essays/impedanceMismatch.html, 2006. [Online; accessed 29-June-2008].

[101] Ayende Rahien. 25 Reasons Not To Write Your Own Object Relational Mapper. http://www.ayende.com/Blog/archive/7615.aspx, 2006. [Online; accessed 29-June-2008].

[102] Billy McCafferty. NHibernate Best Practices with ASP.NET, 1.2nd Ed. http://www.codeproject.com/KB/architecture/NHibernateBestPractices.aspx, 2008. [Online; accessed 30-June-2008].

[103] SubSonic. http://subsonicproject.com/, 2008. [Online; accessed 30-June-2008].

[104] LLBLGen Pro. http://www.llblgen.com/, 2008. [Online; accessed 30-June-2008].

[105] Databases supported by NHibernate. http://www.hibernate.org/361.html, 2006. [Online; accessed 30-June-2008].

[106] Castle ActiveRecord. http://www.castleproject.org/ActiveRecord/, 2008. [Online; accessed 30-June-2008].

[107] The ADO.NET Entity Framework Overview. Visual Studio 2005 Technical Articles http://msdn.microsoft.com/en-us/library/aa697427.aspx, 2006. [Online; accessed 30-June-2008].

[108] Evaluant Universal Storage Services. http://euss.evaluant.com/, 2008. [Online; accessed 30-June-2008].

[109] iRise - Innovation Through Innovation. http://www.irise.com/. [Online; accessed 22-June-2008].

[110] Wikipedia Contributors. Scaffold (programming). Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Scaffold_(programming), 2008. [Online; accessed 1-July-2008].

[111] Jon Galloway. Microsoft should ship SubSonic (formerly called ActionPack) with Atlas. http://weblogs.asp.net/jgalloway/archive/2006/08/30/SubSonic-_2800_formerly-ASP.NET-ActionPack_2900_-_2D00_-Microsoft-should-ship-this-with-Atlas.aspx, 2006. [Online; accessed 2-July-2008].

[112] Castle MonoRail. http://www.castleproject.org/monorail/index.html. [Online; accessed 2-July-2008].

[113] Microsoft ASP.NET MVC. http://www.asp.net/mvc/, 2008. [Online; accessed 2-July-2008].

[114] How it works, Castle MonoRail Documentation. http://www.castleproject.org/monorail/documentation/trunk/manual/howitworks.html, 2008. [Online; accessed 26-June-2008].

[115] Generator. Castle Project http://www.castleproject.org/others/contrib/generator/index.html, 2006. [Online; accessed 2-July-2008].

[116] Microsoft ASP.NET Dynamic Data. http://www.asp.net/dynamicdata/, 2008. [Online; accessed 2-July-2008].

[117] ASP.NET in .NET 3.5 Service Pack 1 Beta. http://www.asp.net/downloads/3.5-SP1/default.aspx, 2008. [Online; accessed 2-July-2008].

[118] Wikipedia Contributors. Model-driven architecture. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Model-driven_architecture, 2008. [Online; accessed 3-July-2008].

[119] Conditional compilation in C# - Explaining System.Diagnostics.ConditionalAttribute. B# .NET Blog http://bartdesmet.net/blogs/bart/archive/2006/08/30/4368.aspx, 2006. [Online; accessed 3-July-2008].

[120] José Carvalho. Equipment Database - Requirements Specification. Qimonda Internal Document, 2008.

[121] José Carvalho. Equipment Database - System Specification. Qimonda Internal Document, 2008.

[122] David Garlan and Mary Shaw. An Introduction to Software Architecture. School of Computer Science, Carnegie Mellon University http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf, 1994. [Online; accessed 3-July-2008].

[123] Rick Kazman Len Bass, Paul Clements. *Software Architecture in Practice*. Addison-Wesley Professional, first edition edition.

[124] Martin Fowler. Anemic Domain Model. Martin Fowler Bliki http://martinfowler.com/bliki/AnemicDomainModel.html, 2003. [Online; accessed 25-June-2008].

[125] ReSharper 4.0. http://www.jetbrains.com/resharper/, 2008. [Online; accessed 4-July-2008].

[126] Wikipedia Contributors. Top-down and bottom up design. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design, 2008. [Online; accessed 18-June-2008].

[127] Vic Hartog and Dennis Doomen. Coding Standard: C#. Philips Medical Systems - Software / SPI http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf, 2005. [Online; accessed 17-June-2008].

[128] Juval Lowy. C# Coding Standard, Guidelines and Best Practices. IDesign Inc. http://www.idesign.net, 2008. [Online; accessed 17-June-2008].

[129] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html, 2004. [Online; accessed 2-July-2008].

[130] Wikipedia Contributors. Object Relational Mapping. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Object-relational_mapping, 2008. [Online; accessed 11-June-2008].

[131] Wikipedia Contributors. Warp Drive. Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Warp_drive, 2008. [Online; accessed 11-June-2008].

[132] K. Rustan M. Leino Mike Barnett and Wolfram Schulte. The Spec# Programming System: An Overview. 2004. [Online; accessed 16-June-2008].

[133] ADO.NET team. The ADO.NET Entity Framework - Not Just For SQL Server! http://blogs.msdn.com/adonet/archive/2007/12/17/the-ado-net-entity-framework-not-just-for-sql-server.aspx, 2007. [Online; accessed 16-June-2008].

[134] ADO.NET team. ADO.NET Entity Framework: What. How. Why. http://channel9.msdn.com/shows/Going+Deep/ADONET-Entity-Framework-What-How-Why/#217633&d=0, 2006. [Online video; accessed 16-June-2008].

[135] Jason Kresowaty. FxCop and Code Analysis: Writing Your Own Custom Rules. http://www.binarycoder.net/fxcop/, 2008. [Online; accessed 16-June-2008].

[136] Best AOP Framework with Castle. Castle Project Support Forum http://forum.castleproject.org/viewtopic.php?t=2690, 2007. [Online; accessed 19-June-2008].

[137] MyGeneration - Code Generation, O/R Mapping, and Architectures. http://wwww.mygenerationsoftware.com/, 2008. [Online; accessed 2-July-2008].

[138] Phoenix Academic Program. Microsoft Research http://research.microsoft.com/Phoenix/default.aspx, 2008. [Online; accessed 26-June-2008].

[139] View Engines Comparison. Castle MonoRail documentation http://www.castleproject.org/monorail/documentation/trunk/viewengines/comparisson.html, 2008. [Online; accessed 2-July-2008].

[140] The AspectJ Project. http://www.eclipse.org/aspectj/, 2008. [Online; accessed 2-July-2008].

[141] Apache log4net. http://logging.apache.org/log4net/. [Online; accessed 2-July-2008].

[142] jQuery. http://jquery.com/, 2008. [Online; accessed 2-July-2008].

Index