Algorithm Simplification through Object Orientation

Eliezer Kantorowitz
Computer Science Department, Technion-Israel Institute of Technology, 32000 Haifa, Israel(email: kantor@cs.technion.ac.il)

SUMMARY

The object oriented (O-O) approach is claimed to have a number of advantages. Some support to these claims appeared during an O-O redesign of a legacy CAD system. A surprisingly simple and efficient solution algorithm was discovered for a change propagation problem. The analysis of the case employs the new concepts of implementation and extension complexity, which indicate the amount of code (software costs) required for the implementation and for later extensions. These two complexities are functions of the problem complexity expressed by the number N of object types employed to model the problem domain. Moving from the old system to the new O-O system reduced the implementation complexity from $O(N^2)$ to $O(N)$, the extension complexity is reduced from $O(N)$ to $O(1)$. The two systems have the same space and time complexities. The CAD system is employed for designing structures composed of parts. The O-O analysis attempts to analyze each part type separately, which proved possible. The corresponding N different object types can therefore be developed independently of each other. The top down analysis employed for the old algorithm did not discover the simple architecture, because it is not geared to look for this kind of solutions. Instead it analyzes the N^2 different change propagation cases. The methodical search for independent modules is an important reason for preferring O-O analysis.

Key words: object oriented; software architecture; complexity; implementation complexity; extension complexity

1. INTRODUCTION

Solution architectures should be simple in order to avoid high software development and maintenance costs. At the same time these architectures should also involve low execution costs. It is, however, a common experience in system development that optimizing one goal often compromises other goals. In the case described in this paper, the use of an O-O approach enabled the design of an architecture that involved relatively low implementation costs, and has the same reasonably low execution costs as the original top down designed system. This paper analyzes the case and explains why and when such an advantage can be achieved with O-O. The case involves the re-engineering of a legacy CAD (Computer Aided Design) system. Some parts of this legacy system are about 20 years old, and it does not meet some current needs.

Extending the system to satisfy these new needs was however estimated to be very difficult, if not impossible. The reason is that the old code is quite complicated, and has a number of difficulties in removing bugs. It was estimated that O-O could alleviate some of these problems. It was therefore decided to begin developing a new O-O system, that will gradually, over a number of years, replace the old system.

2. THE CHANGE PROPAGATION PROBLEM

The purpose of the CAD system is to enable a team of engineers to design the approximately 10^5 different parts of a large steel structure. During this process, the design is sometimes changed. For example, the thickness of a particular part is increased. This change may require changes also in the parts connected to the first part. These changes may cause further changes and so on. This *change propagation problem* proved to be very difficult. The old program was therefore very complicated. The O-O solution developed by the author surprised everyone with its simplicity. In order to understand what happened here, we must first learn something about the change propagation problem.

The 3D parts employed in this industrial system are divided into 9 distinct types. Some of these types are quite complicated. For the purposes of this paper, we consider a highly simplified 2D case, in which all parts are rectangles whose faces are parallel to the coordinate axes of an orthogonal coordinate system. The faces of these rectangles are denoted by the compass directions, N, E, S, and W. A part may be rotated by any multiple of $\pi/2$ radians. It is therefore possible that the N face of one part interfaces, i. e. touches, with the N face of another part that has been rotated through π.

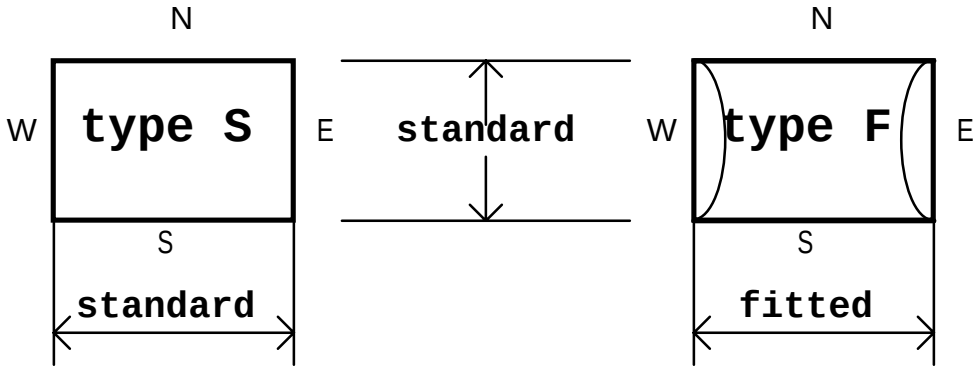

Fig. 1 Parts of type S and F

The parts are manufactured from rectangles supplied by steel mills in standardized dimensions. Parts that are used as they come from the steel mill, without any change, belong to type S (Standard), because they have standard dimensions. An example of an S part is shown in the left side of Fig. 1. The right side of Fig.1 shows a part of type F (fitted), i.e. a part where the distance between the E and W faces is non standard (fitted), while the distance between the S and N faces is a standard dimension. The E and W faces are marked by arcs in order to indicate that they correspond to a fitted

dimension. An application of an F part is shown in Fig. 2, where it is required to fit a part P between the two existing parts called L and R. The distance 2.17 between these two parts is nonstandard. The part P is therefore produced by cutting down a standard 3.00×1.00 part to the required 2.17×1.00. Fitting is done only by cutting down the distance between the E and W faces of the part, and not between the N and S faces. The locations of the E and W faces of F parts are determined by fitting and are, therefore, denoted *fitted faces*. The location of the N and S faces are determined by some standard dimension and are therefore, called *standard faces*. The N, E, S, and W faces of S parts are thus all standard. It makes no sense to fit a fitted face of one part to a fitted face of another part. It is therefore not permitted to interface the E or W face of an F part with the E or W face of another F part.

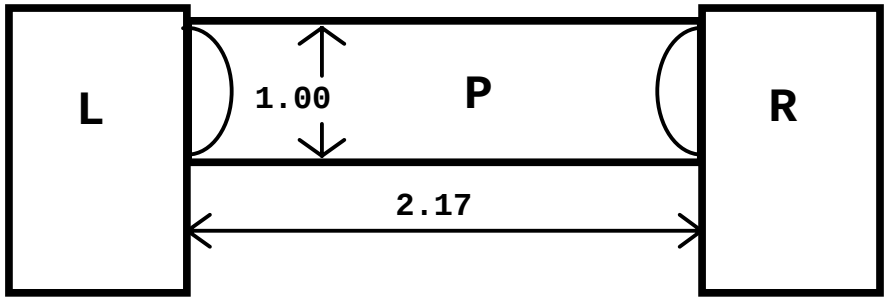

Fig. 2 Part P of type F is fitted in the space between parts L and R of type S.

During the design process the designer sometimes wishes to make changes. Fig. 3, for example, shows a design composed of 5 parts of type S. The leftmost part is denoted by 1 and has the length *t*. The designer wishes for some reason to increase this length to $t+d$. We assume that the W face of part 1 has a fixed location in some absolute coordinate system. The E face of part 1 will therefore move in this coordinate system the distance *d* to the right. This face is, however, welded to the W face of Part 2. As a result, part 2 will move *d* to the right. Eventually, so will all the four rightmost parts of the structure (2, 3, 4, and 5). In this example, the change propagates through the entire structure. Let us now consider the case where the designer wishes to shorten part 1 by d. The W face of part 2 is in this case pushed d to the left. Part 2 will therefore move d to the left and so will all the parts to the right of it. We can now formulate the first rule.

Rule 1. When a standard face of a part is pushed a distance *d*, the entire part will move *d* in the push direction. The shape of the part will not change.

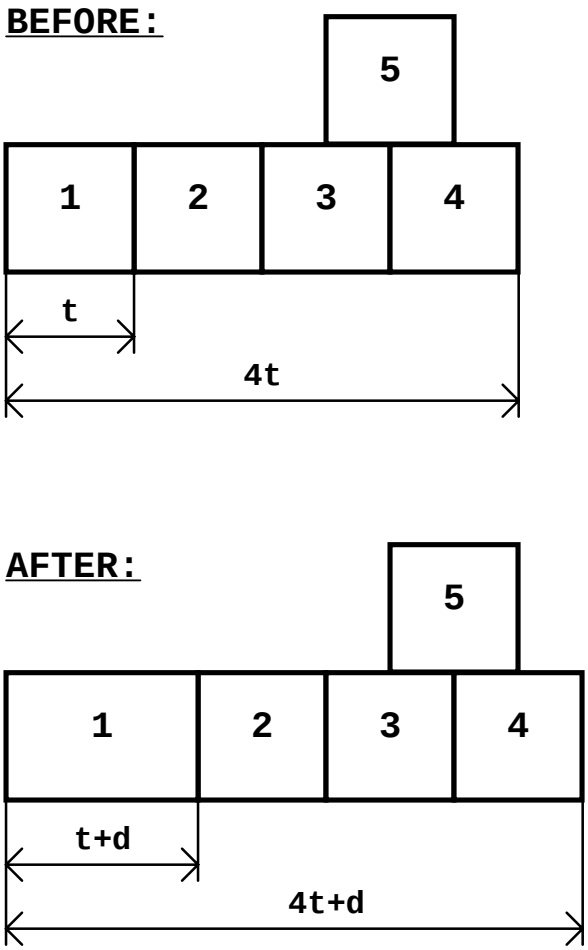

Fig. 3 Changes propagate troughout standard dimensions

Consider now a similar structure (Fig. 4), in which part 3 is of type F and the remaining parts 1, 2, 4, and 5 are of type S. In this example assume that the locations of both the W face of part 1 and the E face of part 4 are fixed in the absolute coordinate system. Again, consider what happens when the designer increases the length of part 1 from *t* to *t+d*. The E face of part 1 moves *d* to the right. Part 2 will therefore also move *d* to the right. The length of part 3 will now be fitted to the space between part 2, in its new location, and part 4, whose location in the absolute coordinate system is fixed. The result is that the length of 3 is reduced by *d*. It is also noted that the propagation of the change stops at part 3. The fitted face absorbs the change and thus stops further propagation.

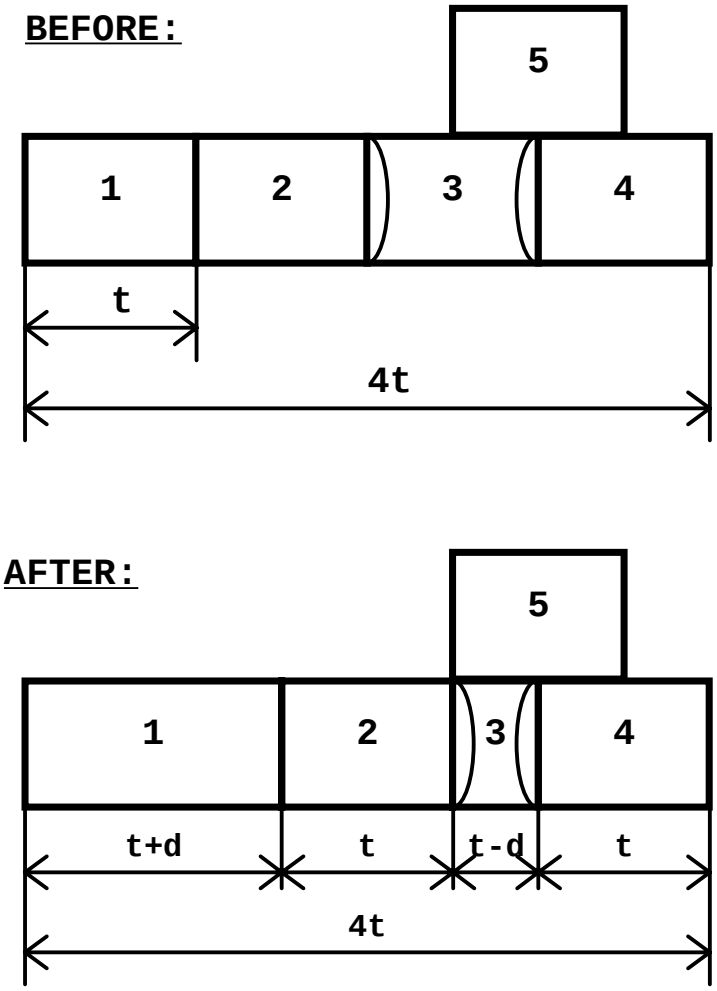

Fig. 4 Change propagation stops in a fitted dimension

We can now formulate the second rule.

Rule 2. When a fitted face, i.e. an E or W face of an F part, is pushed a distance *d*, the length of the part, i.e., the distance between the W and E faces, will be reduced by *d*. The part will not move and the change propagation stops.

3. TRADITIONAL TOP DOWN DESIGN

This section discusses the architecture of a solution algorithm using a traditional top-down approach [1] . This solution will be contrasted with an O-O solution developed later.

In a general formulation of the change propagation problem, an architecture may be modeled by a *structure graph*, whose nodes represent the parts and the arcs represent the interfaces between them. Fig. 5 shows the graph that corresponds to the structure shown in Fig. 4.

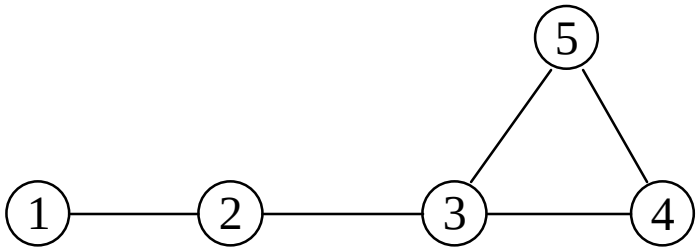

Fig 5 Structure represented by a graph

The change propagation problem can now be formulated as a calculation of how a change propagates over any given structure graph. In a traditional top down analysis this formulation can be considered as the problem statement at the highest level of abstraction. At the second level of abstraction, the problem is "How does any given change propagate from any given node in the graph to any node that is connected directly to the first one?". In order to answer this question, it is necessary to analyze how a change propagates through any of the possible kinds of interfaces, i.e. through graph arcs. Recall that it is permitted to rotate any part a multiple of $\pi/2$ radians, so that each of the $N_{faces} = 4$ faces (N, E, S or W) of a part may interface with any of the $N_{faces} = 4$ faces of an interfacing part. The two interfacing parts may, however, belong to any of the $N_{types} = 2$ different part types (fitted and standard). There are thus,

$$N_{face\_types} = N_{faces} N_{types} = 4N_{types} = 8$$

kinds of faces that may interface with each other. This gives

$$N_{\text{int}erface\_types} = N^2_{face\_types} = 4N^2_{types} = 64$$

different interface types. Table 1 shows how a change propagates through any of these 64 different kinds of interfaces. The table was calculated with the help of rules 1 and 2. The leftmost column and the uppermost row in the table show all the kinds of faces that exist in the system. "S-N", for example, means "face N of a part of type S". Every internal cell in the table contains a letter (**m, s,** or **e**) that tells what happens when a part with a face of the kind indicated by the row name pushes another part on a face of the kind indicated by the column name. The letter **m** (move) means that the part moves. An example is shown in Fig. 4, in which the E face of part 1 pushes the W face of part 2, which therefore moves to the right. This case is shown in Table 1, by the letter **m** at the intersection of the row S-E and the column S-W. The

letter **s** (shorten the part and stop propagation) is employed when a fitted dimension is changed. The letter **e** (error) is for an illegal interface between two fitted faces.

	S-N	S-E	S-S	S-W	F-N	F-E	F-S	F-W
S-N	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
S-E	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
S-S	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
S-W	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
F-N	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
F-E	**s**	**s**	**s**	**s**	**s**	**e**	**s**	**e**
F-S	**m**	**m**	**m**	**m**	**m**	**s**	**m**	**s**
F-W	**s**	**s**	**s**	**s**	**s**	**e**	**s**	**e**

Table 1. Change propagation throughout all possible interfaces.

In the industrial system there are $N_{types} = 9$ types of 3D parts, and each has $N_{faces} = 6$ faces, i.e.

$$N_{face\_types} = N_{faces} N_{types} = 6 \times N_{types} = 54$$

The number of entries in the equivalent of Table 1 in the industrial system is

$$N_{interface\_types} = N^2_{face\_types} = 36 \times N^2_{types} = 2916$$

Each one of these $36 \times N^2_{types} = 2916$ interface types was managed in the old industrial system by a separate piece of code. The amount of code that implements the old change propagation algorithm is thus a function of N^2_{types}, and may therefore be represented by an *implementation complexity* of

$$O(N^2_{types}).$$

The implementation complexity concept suggested in this paper is an indication of code size as a function of the problem complexity, which in this case is represented by the number of different part types N_{types}. This complexity measure, which will be discussed later in the paper, is analogous to the well known time and space complexities, which are employed as indicators for computing time and memory as function of the amount of data involved.

The 2916 pieces of code that implemented the old code were poorly documented and its designers were no longer available. It is, therefore, not surprising that even a continual debugging effort could not remove some damaging bugs. Typically, the repair of one bug caused new ones to appear. Another problem with the old system was that a needed extension with new part types was judged to be nearly impossible. To understand why such an extension is difficult, we calculate the number of code pieces that must be added when the number of part types is increased by 1 ‘ (from 9 to 10):

$$36 \times (N_{types} + 1)^2 - 36 \times N^2_{types} = 72 \times N_{types} + 36 = 684$$

which involves a considerable effort. The above formula shows that the number of code pieces grows linearly as N_{types} increases. This may be expressed by an *extension complexity* of

$$O(N_{types})$$

This complexity measure, which is suggested in this paper, is an indication of the amount of code to be added when the complexity of the problem is increased.

4. OBJECT ORIENTED DESIGN

This section outlines the O-O development process that produced the new program with the simple architecture. The O-O approach was introduced around 1967 with the SIMULA programming language[2,3] , and became quite popular after the introduction of SMALLTALK[4] around 1980. Later several books on O-O software development appeared, for example [5,6,7,8,9]. We consider the approach of Jacobson [9], where a program is composed of *entity*, *control*, and *interface objects*. Entity objects are program modules that model the problem domain. Each worthy real world object in this domain is represented in the software by an entity object. Interface objects are the program modules used to communicate with human users or with other programs. When a user or another program communicates a request for a service to an interface object, the latter will call a control object to do the job. In order to accomplish this, the control object will typically employ a number of entity objects. One of the important stages in the O-O software development process is to identify the problem domain objects to be modeled by entity objects. In our simplified change propagation problem, the entity object types are the two part types, fitted and standard (Fig. 1). In a later refinement step of the O-O process, the methods, and attributes of these two object types are specified. The attributes are the types of the four faces of the part. The methods required correspond to the messages that a part may receive from other parts. The messages relevant to the change propagation problem are "you are being pushed a distance *d* on face *i* (*i* = N, E, S, or W); do what is required". In the design of the method that processed this message we followed the O-O approach, i.e attempting to process the message solely by the means of the object that received it. This attempt succeeded, and as a result the rules 1 and 2, presented earlier in this paper, were produced. When a standard face is pushed, rule 1 is applied. That is, the part moves, and if it interfaces with other parts it will push them. Pushing is coded as sending messages to these parts saying that they are also being pushed. How these parts handle these messages is not known to the part. If, on the other hand, a fitted face is pushed, rule 2 is employed. This means that the length of the part is shortened, and the propagation stops, i.e. the part sends no messages to other interfacing parts. The software that manages the change propagation thus comprises of one method for each face of each object type. The industrial system has 9 part types each having 6 faces that is

$$N_{methods} = N_{face\_types} = N_{faces} N_{types} = 6 \times N_{types} = 54,$$

which is dramatically less than the 2916 pieces of code employed in the old legacy system. Moving to O-O technology thus reduced the number of cases to be managed by the code from $N^2_{face\_types}$ to N_{facee_types}. The implementation complexity of the O-O architecture is thus

$$O(N_{types})$$

to be compared with $O(N_{types}^2)$ in the old non O-O system. The simplicity of the O-O architecture is due to the fact that the object being considered need not know anything about the object that pushes it. The analysis is solely based on the type of the face, which is being pushed. Only one method per face type is therefore required giving a total of N_{face_types} methods. The old system, on the other hand, analyses the way in which a change propagates through the interface between the pushing and the pushed faces, i.e. all possible pair of face types must be considered. The number of different analyses in the old program is therefore $N_{face\_types}^2$.

Adding a new part type to the O-O system requires a new software object type for this part with 6 methods for the 6 faces. This is also a dramatic reduction from the 684 code pieces required in the old system. Note that the amount of code required for each new part type (6 methods) is independent of the value of N_{types} . The extension complexity of the new O-O system may therefore be expressed as

$$O(1)$$

compared with $O(N_{types})$ in the old system. True to its goal the new system is thus considerably easier to extend than the old one. In retrospect, after having followed the O-O approach, it is possible to see a different initial abstraction that would allow a top down development of a code with the same architecture, implementation complexity and extension complexity, as the O-O solution. The advantage of the O-O analysis is that it led directly to the simple solution.

5. WHEN CAN O-O ANALYSIS PRODUCE A SIMPLE ARCHITECTURE

A part of the explanation for why O-O analysis can lead to simple architectures is that one of its goals is to divide the responsibilities of the program among its objects. A further goal is that, when possible, an object should process the messages it receives solely without using other objects. O-O analysis thus strives at reducing the amount of coupling between the different program modules (the objects), which brings down the complexity of the architecture. The ideal O-O design is a program composed of a number of cooperating objects, each of which takes sole care of its responsibilities. This kind of program has typically a distributed control of flow. To illustrate this, let us consider the example shown in Fig. 4. When part 1 is extended from *t* to *t+d*, it will send a message to part 2 saying "you are being pushed ". Part 2 reckons that it has to move to the right, and sends a message to part 3 that it is being pushed, and so on. The decisions on where to go next are made by the different objects. This distributed control of flow contrasts with the tree structured control with a main function in the root, which is typical for programs designed by traditional top-down methods.

The traditional top down analysis of the change propagation problem did not reveal the simple architecture that was discovered later using O-O analysis. To understand why, we recall the steps in the top-down analysis. At the highest level of abstraction, we asked how a given change propagates over a structure represented by its structure graph. To answer this question, we formulated the sub-question "How does a given

change propagate from any given part to any part that interfaces with it?". This led to an analysis of how change propagates through anyone of the N_{types}^2 possible interfaces. No stage of this process focused on the faces of the parts. Thus, there was no chance that the useful properties of the faces would be discovered and exploited. Another way of explaining this, is that traditional top-down analysis is controlled by the functionality of the system, while the data involved are not inherently visible. This contrasts with O-O analysis, in which both the data elements and the functions of the objects are explicitly analyzed. Following the O-O approach produces the question "What happens when a face of a part is pushed". The next step according to the O-O method is to try to solve this problem without any knowledge of the part that pushes our part, which fortunately proved possible. The O-O approach leads, when possible, to a program composed of modules that can function independently of each other.

6. IMPLEMENTATION AND EXTENSION COMPLEXITIES

The quality of an algorithm is traditionally characterized by its time and space complexities, that are useful as rough indicators for the cpu and memory requirements. They are not intended for accurate estimates, which require detailed analysis. This paper suggests complexity indicators for the costs of software implementation and extension. These complexities are also intended to be only rough indicators. More accurate cost estimates require elaborate software metrics analysis, as described for instance in [10,11] . A theoretical approach to complexity analysis based on an abstract O-O machine is found in [12]. The complexities suggested in this paper are indicators for the amount of code to be written as function of the number of different entity object types employed by the algorithm. The reason for this approach is that solving a problem in a more complicated problem domain, which is characterized by the number of entity object types involved, is expected to require more code. The amount of required code was in our change propagation problem measured by counting the number of required methods. In other problems other units for code measurement may be appropriate.

One problem in this case study was how to count the number of different object types in the old non O-O program. It was done by considering the code and data specifications that correspond to a particular part type as an implicitly defined object type. The 9 different part types employed in this old non O-O program were therefore considered as 9 different object types.

7. OBJECT ORGANIZATION AND PERFORMANCE

The legacy system is composed of a number of distinct parts, one of which is for the designing of the steel parts. The first version of the O-O system corresponded to the steel component of the legacy system, and had therefore only 9 different object types (C++ classes) for the different kinds of steel parts. The system was later extended with additional object types for different kinds of equipment, such as pipes and pumps, that are attached to the steel structure. In this way, the number of entity object types increased over a number of years from 9 to 85. This large extension was possible only because adding a new object type proved to be relatively easy. The effort required to define a new object type was independent of the number of object types that were already defined, consistent with the estimated extension complexity of $O(1)$. Class derivation and inheritance were useful in organizing this large number of classes systematically. All the classes are in a single derivation tree with a sub-tree for each

kind of class, e.g. steel parts, pipes, pumps. This structure also enabled some code reuse through inheritance. The number of objects is around 200,000, and a commercial database system was employed to manage these data.

Each of the 85 entity classes has a function for displaying itself on the screen. These functions have all the same name, making use of polymorphism. A picture of a number of different objects, which may belong to different classes, is produced by calling their display functions. When a new class is introduced, a display function must be implemented for it. Writing this single function is all that is required in order to extend the graphical system to include the new class. Polymorphism was thus useful in designing a graphical algorithm with an extension complexity of only $O(1)$. It is noted that both the change propagation algorithm and the algorithm for graphical display employ the same set of 85 entity object types, i.e. the two algorithms have the same *entity domain*. Fortunately, both involve the same low extension complexity of $O(1)$.

The legacy system and the new O-O system do exactly the same geometrical computations when solving the same given change propagation problem. The time complexity is in both cases $O(n_{parts})$ where n_{parts} is the number parts that participate in the change propagation. Typically, n_{parts} is quite small, and computation time of up to a few seconds were observed in the O-O system. This is acceptable for the users. Most of the time is spent for disk data storage and retrieval and for updating the screen picture of the modified structure. The legacy system, on the other hand, has bugs that causes a large number of superfluous manipulations. Attempts to remove them caused worse problems. Computation times measured in minutes are therefore quite common. The principal reason for the performance advantage of the O-O system is its simplicity, which helped in avoiding the bugs that produced the superfluous computations. The legacy CAD/CAM system is composed of a large number of different programs, and only a part of it has been re-engineered. Many tens of person years are required in order to re-engineer the rest of it. A large investment is required also for training the users in the operation of the new system. The process of modernizing the system will probably take several years.

8. CONCLUSIONS

The implementation and extension complexities suggested in this paper were useful indicators for the properties of the old and new change propagation programs. The old difficult-to-extend program has a relatively high extension complexity of $O(N_{types})$, while the new easy-to-extend system has an extension complexity of only $O(1)$. The large and difficult-to-debug old system has an implementation complexity of $O(N_{types}^2)$, while the new smaller system has a lower implementation complexity of only $O(N_{types})$. More experience must be gained with these complexity measures in order to assess their usefulness in general .

Brooks [13] noted that one of the major source of problems with software is due to its complexity, which in most cases increases much more than linearly with the size. This was the case with the old legacy system with its implementation complexity of

$o(N_{types}^2)$. This complicated architecture resulted from a top down analysis. The problem of this analysis was that it considered the system as a whole with all relationships between all the parts (the structure graph of Fig. 5). This line of thought leads to a complex system, where the propagation of changes through all possible types of part face combinations are considered. In the O-O approach, on the other hand, the goal is to distribute the responsibilities of the program among a number of independent objects, and such that each object takes care of its responsibilities without any knowledge of other objects. In problems where this is possible, for example our change propagation problem, the O-O analysis therefore leads to a simple architecture with relatively low implementation and extension complexities. This observation supports the opinion that O-O analysis should in general be preferred over traditional top down analysis.

REFERENCES

1. N. Wirth, 'Program Development by Stepwise Refinement', *Communication of the ACM*,14, 221-227,(4-1971)

2. O.J. Dahl, B. Myrhaug, and K. Nygaard, '*SIMULA Common Base* ', Norwegian Computing Center s-22, Oslo Norway, 1970

3. O.J. Dahl and C.A.R. Hoare '*Hierarchical Program Construction' in 'Structured Programming'*, Academic Press, New York , 1972, 174-220

4. A.Goldberg, and D. Robson, '*SMALLTALK 80 - The Language and its Implementation'*, Addison-Wesley, Reading. Massachusetts, 1983.

5. Booch, '*Object Oriented Analysis and Design with Applications'*, Addison-Wesley, Reading. Massachusetts, 1994.

6. D. de Champeaux, , D., Lea and, P. Faure., '*Object-Oriented System Development'*, Addison-Wesley, Reading. Massachusetts, 1993.

7. B. Meyer, *'Object oriented Software Construction'*, Hertfordshire,England, Prentice Hall International, 1988.

8. J.Rumbaugh, M.Blaha, W. Premerlani, F. Eddy. and W. Lorensen, '*Object Oriented Modeling and Design'*, Prentice hall International, Englewood Cliffs, 1991

9. I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard, '*Object Oriented Software Engineering'*, Addison Wesley, Reading. Massachusetts, 1992

10. N.E. Fenton, '*Software Metrics a Rigorous Approach'*, Chapman Hall, New York, 1991

11. H. Zuse, '*Software Complexity Measures and methods'*, Walter & Gruyter, New York 1992

12. H. Schmidt, and W. Zimmermann, ‘A Complexity Calculus for Object Oriented Programming’, *Object Oriented Systems*,1, 117-147, (2-1994)

13. F.P. Brooks,. ‘No Silver Bullet Essence and Accidents of Software Engineering’ *Computer*,29, 10-19, (2-1987)