Chapter 1

The Need for Object-Oriented Reengineering

Reengineering legacy systems has become a vital matter in today's software industry. In the past few years, most of the reengineering efforts were focussed on systems written in traditional programming languages such as COBOL, Fortran and C. But recently an increasing demand for reengineering object-based systems has emerged. This recent evolution is not caused by failure of the object-oriented paradigm. Rather, it illustrates that the mere application of object-oriented techniques is not sufficient to deliver flexible and adaptable systems. This is due to a number of obvious problems:

- *Lack of experience*. It requires several years of experience to fully exploit the potential of the object-oriented paradigm. Such experience is often built up during the initial stages of a project, at the time when the most crucial parts of the system are implemented.
- *Hybrid programming languages.* The use of hybrid languages –like C++ and Ada–, combined with a "learn on the job" approach, prevents programmers from making the necessary paradigm shift.
- *Technology expansion.* Legacy systems could not benefit from emerging standards (e.g., UML, CORBA), technological advancements (e.g., design patterns, architectural styles) and extra language features (e.g., C++ templates, Ada inheritance).

These problems are accidental in nature: given proper training and sufficient tool support they will eventually be resolved. So why should one worry about object-oriented reengineering, since within a few years there won't be any more object-oriented legacy systems? In fact there a more fundamental problem.

The law of *software entropy* dictates that even when a system starts off in a well-designed state, requirements evolve and customers demand new functionality, frequently in ways the original design did not anticipate. A complete redesign may not be practical, and a system is bound to gradually lose its original clean structure and deform into a bowl of "object-oriented spaghetti" [WILD 92], [CASA 98], [BROW 98].

Many of the early adopters of the object-oriented paradigm have experienced such software entropy effects. Their systems are developed using object-oriented design methods and languages of the late 80s and exhibit a range of problems that prohibits them meeting the evolving requirements imposed by their customers. Their systems have become overly rigid thus compromising thier competitive advantage and as a consequence object-oriented reengineering technology has become vital to their business.

1.1 The FAMOOS Project

The need for object-oriented reengineering technology has been recognised by two of the leading European companies, namely Daimler-Benz and Nokia. Together with the University of Berne, Forshungszentrum

Informatik, SEMA Spain and Take5 they started a research project –named FAMOOS [1] – to investigate tools and techniques for dealing with object-oriented legacy systems.

The handbook you are reading right now is one of the main results of the FAMOOS project. It collects techniques and knowledge on the problem of software evolution with a special emphasis on object-oriented software. Most of the subject matter is not "new" in the sense that it represents new discoveries. Rather the handbook regroups much of the knowledge about redesign, metrics and heuristics into a single work that is focused on object-oriented reengineering.

1.1.1 Case Studies

All the techniques described in the handbook have been verified on six industrial case-studies, ranging from 50.000 lines of C++ up until 2,5 million lines of Ada. Figure 1.1 provides a quick overview of all of the FAMOOS case studies.

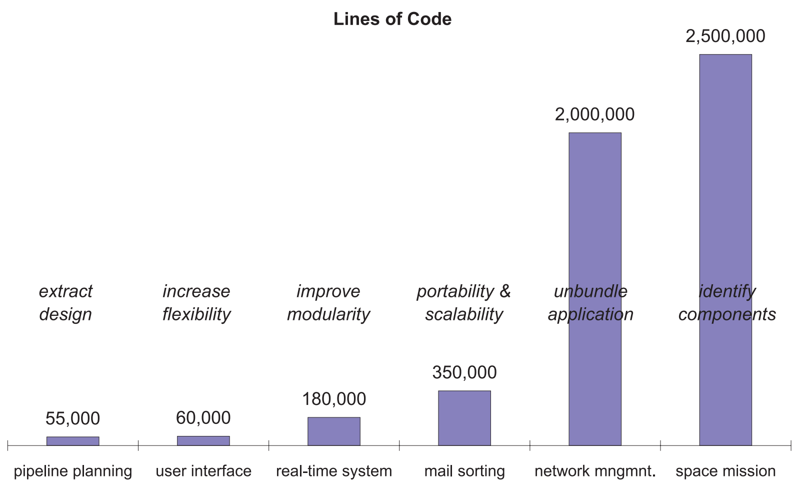

Figure 1.1: Overview of the FAMOOS Case Studies

- *Pipeline Planning.* The system supports the planning of liquid flow in a pipeline between multiple stations. The reengineering goal was to extract design from source-code, in order to reduce the cost of implementing similar systems, probably in other languages. The system is written in C++ and is a candidate for being rewritten in Java or Smalltalk.
- *User Interface.* This software provides graphical representations of telecommunication networks to telecom operators. The reengineering goal was to increase the flexibility of the software, i.e. improve its portability, facilitate addition of functionality and enhance tailorability towards customers. The system is written in C++.
- *Real-time System.* This software provide operating system features for embedded real-time controlling of hardware. The reengineering goal was to improve modularity for gaining shorter edit-compile-run cycles. The system is written in a mixture of C and C++.
- *Mail Sorting.* A control system for machines sorting mail envelopes. The software is highly configurable, to deal with the different ways countries over the world handle letters. The software itself is based on an internally developed distributed architecture which hindered the future evolution. The

[1] If you want to read more about the FAMOOS project and its results, we suggest to browse the web-sites offered by the respective project partners: http://dis.sema.es/ projects/ famoos/; http://www.iam.unibe.ch/ famoos/; http://www.fzi.de/ prost/

reengineering goal was to investigate how new technology could improve the portability and scalability (e.g. CORBA, Java, HTML). The system is written in a mixture of C and C++.

- *Cellular Network Management.* This case-study concerned a management system for digital networks. The main goal of the reengineering project was to unbundle the application, i.e. split the system into sub-products that can be developed and sold separately. The system is written in a mixture of C and C++.

- *Space Mission Management.* A set of applications that in different combinations form systems to support the planning and execution of space missions. The reengineering goal was identify components in order to improve reliability and facilitate system maintenance. The system is written in Ada.

1.1.2 Reengineering Goals

From this list of case studies some interesting information can be learned. First of all, the goals and motivations for reengineering the software systems are quite diverse, yet some common themes emerge.

- *Unbundling.* Unbundle the software system into subsystems that can be tested, delivered and marketed separately.

- *Performance.* Improving performance is sometimes a goal and sometimes considered as a potential problem once the system is reengineered.

- *Port to other Platform.* Porting to other (user-interface) platforms, sometimes requiring overall changes to the system.

- *Design Extraction.* Always a necessary step in understanding the system; sometimes even an explicit reengineering goal.

- *Exploitation of New Technology.* This may range from new features of the programming language up until upcoming standards (CORBA and UML).

1.1.3 Architectural Problems

Besides the motivations for reengineering problems, the case studies experience recurrent problems that are perceived as key obstacles for achieving the stated reengineering goals. Solving these problems requires significant human intervention since it involves an intimate understanding of and considerable changes to the architecture of the legacy system.

- *Insufficient Documentation.* All of the case studies face the problem of non-existent, unsatisfactory or inconsistent documentation. Tools to document module interfaces, maintain existing documentation and visualise the static structure and dynamic behaviour are required.

- *Lack of Modularity.* Most of the case studies suffer from a high degree of coupling between classes / modules / sub-systems that hampers further software development (compilation, maintenance, versioning, testing). A solution will involve metrics to help detect such dependencies and refactoring tools to help in resolving them.

- *Duplicated Functionality.* In many of the case studies several modules implement similar functionality in a slightly different way. This common functionality should be factored out in separate classes / components, but tools are missing which help in recognising similarities and in restructuring the source code.

- *Improper Layering.* In a few case studies the user-interface code is mixed in with the ”basic” functionality, creating problems in porting to other user-interface platforms. A general lack of separation, or layering, is observed with regard to other aspects (distribution, database, operating system) in other case studies. In contrast to a lack of layering, one case study suffers from unnecessary layers. Overly layered modules resulted from each successive developer encapsulating the module with a new concept instead of revising it. This problem needs tool support for defining layers and subsequent correction of broken layers.

1.1.4 Code Clean Up

There are quite a number of problems that have to do with ”code clean up”. Many of these problems arise from the lack of familiarity with the new object-oriented paradigm. But several years of development with sometimes geographically dispersed programming teams that change over time exacerbate these problems. Since they involve behaviour preserving restructuring of code only, those problems could be identified and repaired almost mechanically.

- *Misuse of Inheritance.* Inheritance is used as a way to add missing behaviour to one superclass. This is a often a result of having a method in a subclass being a modified clone of the method in the super-class.
- *Missing Inheritance.* In some cases, programmers have duplicated code instead of creating a subclass. In other parts, long case statements that discriminate on the value of a variable are used instead of method dispatching on a type.
- *Misplaced Operations.* Operations on objects were defined outside the corresponding class. Sometimes this was necessary in order to patch ”frozen” designs.
- *Violation of Encapsulation.* This was observed in extensive use of the C++ friend mechanism. Also, software engineers rely on the strong typing of the compiler to ensure certain constraints, and afterwards use typecasts to circumvent the safety-net. In some cases this leads to redundant type definitions which contaminate the name space.
- *Class Misuse.* This problem has been named ”C style C++”, although it is observed in Ada as well. It refers to the usage of the classes as a structuring mechanism for namespaces only. Sometimes this is necessary to interface with external non object-oriented systems.

1.1.5 Requirements

Last but not least, the case studies impose a number of constraints on the techniques and heuristics presented in this book.

- *Language Independent.* All material in this handbook is applicable on all major object-oriented languages, in particular C++ and Ada, Java and Smalltalk.
- *Scalable.* Some techniques and heuristics scale better than others. Rather than restricting ourselves to those techniques that can deal with small as well as large systems, we choose to specify for every technique the scale of systems it can be applied upon.
- *Tool Support.* There is a heavy emphasis on available tool support for all techniques covered in the book.

1.2 Basic Terminology

Before diving in the specific solutions for object-oriented reengineering, it is useful to agree on some terminology. We rely on the taxonomy of Chikofsky and Cross which is well-accepted within the reengineering community [CHIK 90]. For terminology specific to the object-oriented paradigm, we draw upon the design pattern book [GAMM 95].

- *Reverse engineering.* Originally used for the process of analysing hardware to discover its design, the term refers to the process of recovering information from an existing software system. In general reverse engineering seeks to recover information at a higher level of abstraction such as design information from code. Reverse engineering does not involve modifying the software system: it may be done as a stage in the reengineering process (model capture), as part of an effort to document the system, or as an attempt to extract reusable components from the software.

- *Forward engineering.* Refers to the usual process of software engineering: moving from requirements to high-level design, to progressively lower design levels and to implementation. While it may seem unnecessary to introduce a new term, the adjective ”forward” has come to be used where it is necessary to distinguish from reverse engineering and reengineering.

- *Reengineering.* Reengineering is the modification of a software system which in general requires some reverse engineering to be done. That is, reengineering requires that we first recover a view of the system at a higher level of abstraction than the code itself, then make changes to this view and implement these changes at the code level again. Simplistically, reengineering thus involves moving from code to model (reverse engineering), making modifications to the model, and then moving to ”better” code (forward engineering).

  There is some discussion as to whether or not reengineering involves a change in the functionality of the system – what it does for the user – since practically speaking reengineering almost always modifies the existing behaviour of the system, and indeed is usually motivated by a need to meet new requirements.

- *Restructuring.* Restructuring refers to transforming a system from one representation to another while remaining at the same abstraction level. At implementation level, this usually means changing the code structure without changing the semantics. However, even if the semantics are not changed at implementation level, restructuring might affect higher levels of abstraction (changing design vocabulary without affecting the implementation).

- *Refactoring.* Refactoring is restructuring within an object-oriented context. Refactoring involves tearing apart classes into special and general purpose components and rationalising class interfaces. The principle behind refactorings is that some relatively simple transformations (e.g., renaming a class, renaming a method, moving a method or attribute to another class) are combined into quite powerful semantic preserving transformations (i.e., componentise parts of a class, introduce a bridge design pattern).

1.3 The Reengineering Life-cycle

In this section we present the FAMOOS reengineering life-cycle. We regard reengineering as an evolutionary process consisting of the following six stages (see also figure 1.2):

1. Requirements analysis: identifying the concrete reengineering goals.

2. Model capture: documenting and understanding the design of a legacy system.

3. Problem detection: identifying violations of flexibility and quality criteria.

4. Problem analysis: selecting a software structure that solves a design defect.

5. Reorganisation: selecting the optimal transformation of the legacy system.

6. Change propagation: ensuring the transition between different software versions.

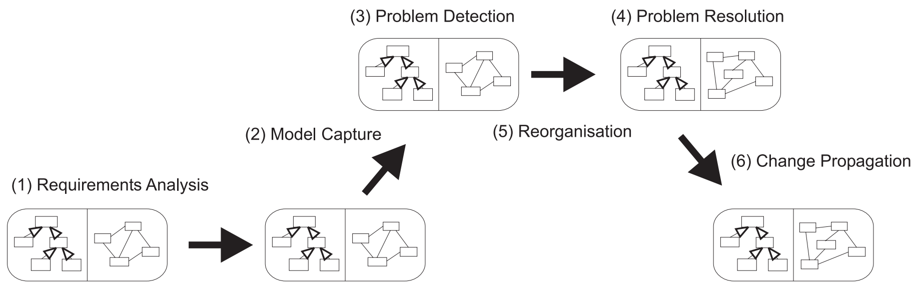

Figure 1.2: The Reengineering Life-cycle

Several iterations of these re-engineering stages might be needed before achieving a stable system with the desired degree of generality and adaptability.

Requirements Analysis. The specification of the criteria that the new, reengineered software must fulfill (for example, faster network performance).

Model Capture. In order to understand and to manipulate an object-oriented legacy system, it is necessary to capture its design, its architecture and the relationships between different elements of its implementation. A common problem in legacy systems is the lack of documentation. As a consequence, a preliminary model capture is often unavoidable, in order to document the system and the rationale behind its design. This requires reverse-engineering the legacy system to extract design information from the code.

Problem Detection. According to the reengineering requirements, problem areas within the legacy systems need to be detected. This requires methods and tools to inspect, measure, rank and visualise software structures. The problem areas have typically properties that deviate strongly form the properties as they are defined in the requirements. Detecting the problems with respect to flexibility requires a definition of these deviations (for example through thresholds on metrics). Problem detection can be based on a static analysis of the legacy system (i.e. analysing its source code or its design structure), but it can also rely on a dynamic usage analysis of the system (i.e. an investigation of how programs behave at run-time).

Problem Analysis. Upon detection of possible defects in the legacy system, software developers have to analyse them; that is, match detected problems against unmet requirements and understand how they concretely affect the software. Because applications are organised as intricate webs where classes, objects and methods may participate in various interactions, a detected problem may have to be decomposed into elementary sub-problems. A selection follows of appropriate target software structures - such as design patterns - that impart the software with the desired flexibility and functionality. A combination of such structures may be necessary to handle the particular design defect at hand. A prerequisite for problem analysis is an identification and specification of software structures to serve as the targets of reengineering, and a classification that allows to look for target structures corresponding to particular flexibility criteria or functional requirements.

Reorganisation. This phase of reengineering consists in physically transforming software structures according to the operations selected previously. This requires methods and tools to manipulate and edit software systems, to reorganise and recompile them automatically, to debug them and check their consistency, and to manage versions of software.

Change Propagation. The process of establishing a revised system throughout a corporate software environment. This might involve reengineering methodology that supports dissemination of improvements in more than one step.