For Ann, Sara and Niels.
For Florence, Quentin and Thibaut.
For Angela and Frida.
# Table of Contents

Preface................................................................................................................... ix  

Part I  Introduction

1. **Reengineering Patterns** ................................................................................. 3
   1. Why do we Reengineer? .................................................................................. 3
   2. The Reengineering Lifecycle ......................................................................... 6
   3. Reengineering Patterns ............................................................................... 10
   4. The Form of a Reengineering Pattern ......................................................... 11
   5. A Map of Reengineering Patterns ............................................................... 13

Part II  Reverse Engineering

2. **Setting Direction** ......................................................................................... 17
   Agree on Maxims .............................................................................................. 18
   Appoint a Navigator .......................................................................................... 19
   Speak to the Round Table ............................................................................... 19
   Most Valuable First ........................................................................................... 20
   Fix Problems, Not Symptoms .......................................................................... 22
   If It Ain’t Broke, Don’t Fix It ........................................................................... 22
   Keep It Simple .................................................................................................... 23

3. **First Contact** ............................................................................................... 25
   Chat with the Maintainers ............................................................................... 29
   Read all the Code in One Hour ........................................................................ 35
   Skim the Documentation ................................................................................... 40
   Interview During Demo ..................................................................................... 45
   Do a Mock Installation ....................................................................................... 51

4. **Initial Understanding** .................................................................................. 55
   Analyze the Persistent Data ............................................................................ 58
   Speculate about Design .................................................................................... 65
   Study the Exceptional Entities .......................................................................... 72

5. **Detailed Model Capture** ............................................................................. 79
   Tie Code and Questions .................................................................................. 82
   Refactor to Understand .................................................................................... 87
   Step Through the Execution ............................................................................ 91
   Look for the Contracts ...................................................................................... 94

*Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz*
Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz

Learn from the Past ........................................................................................................ 97

Part III  Reengineering

6. Tests: Your Life Insurance! ...................................................................................... 103
   Write Tests to Enable Evolution .............................................................................. 106
   Grow Your Test Base Incrementally ........................................................................ 110
   Use a Testing Framework ....................................................................................... 113
   Test the Interface, Not the Implementation .......................................................... 119
   Record Business Rules as Tests ............................................................................ 122
   Write Tests to Understand .................................................................................... 125

7. Migration Strategies ............................................................................................. 127
   Involve the Users ................................................................................................. 130
   Build Confidence ................................................................................................ 132
   Migrate Systems Incrementally .......................................................................... 134
   Prototype the Target Solution ............................................................................. 136
   Always Have a Running Version .......................................................................... 138
   Regression Test After Every Change .................................................................... 140
   Make a Bridge to the New Town ........................................................................... 142
   Present the Right Interface .................................................................................. 144
   Distinguish Public from Published Interface ...................................................... 146
   Deprecate Obsolete Interfaces ............................................................................. 148
   Conserve Familiarity ............................................................................................ 150
   Use Profiler Before Optimizing .......................................................................... 151

8. Detecting Duplicated Code ................................................................................. 153
   Compare Code Mechanically .............................................................................. 155
   Visualize Code as Dotplots .................................................................................. 160

9. Redistribute Responsibilities ............................................................................... 165
   Move Behavior Close to Data ............................................................................. 168
   Eliminate Navigation Code .................................................................................. 175
   Split Up God Class ............................................................................................. 183

10. Transform Conditionals to Polymorphism ...................................................... 189
    Transform Self Type Checks ............................................................................... 192
    Transform Client Type Checks .......................................................................... 199
    Factor out State ................................................................................................ 207
    Factor out Strategy ............................................................................................ 210
    Introduce Null Object ....................................................................................... 213
    Transform Conditionals into Registration ......................................................... 216

Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Part IV Appendices

11. Thumbnail patterns ................................................................. 225

1. Testing Patterns ................................................................. 225
   Retest Persistent Problems .............................................. 225
   Test Fuzzy Features ...................................................... 226
   Test Old Bugs .............................................................. 226

2. Refactorings .............................................................. 226
   Encapsulate Field .................................................. 226
   Extract Method ..................................................... 226
   Move Method ......................................................... 227
   Rename Attribute .................................................. 227
   Rename Method ..................................................... 227
   Replace Conditional with Polymorphism ....................... 227

3. Design Patterns .............................................................. 227
   Abstract Factory ...................................................... 227
   Adapter ................................................................. 228
   Facade ................................................................. 228
   Factory Method .................................................... 228
   Flyweight ............................................................ 228
   Null Object ........................................................... 228
   Quantity ............................................................... 229
   Singleton ............................................................. 229
   State ................................................................. 229
   State Patterns ........................................................ 229
   Strategy .............................................................. 229
   Template Method .................................................. 230
   Visitor ................................................................. 230

12. References .......................................................................... 231
Preface

A Fairy Tale

Once upon a time there was a Good Software Engineer whose Customers knew exactly what they wanted. The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customers’ problems now and for decades. When the Perfect System was designed, implemented and finally deployed, the Customers were very happy indeed. The Maintainer of the System had very little to do to keep the Perfect System up and running, and the Customers and the Maintainer lived happily every after.

Why isn’t real life more like this fairy tale?

Could it be because there are no Good Software Engineers? Could it be because the Users don’t really know what they want? Or is it because the Perfect System doesn’t exist?

Maybe there is a bit of truth in all of these observations, but the real reasons probably have more to do with certain fundamental laws of software evolution identified several years ago by Manny Lehman and Les Belady. The two most striking of these laws are [Lehm85a]:

- The Law of Continuing Change: A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- The Law of Increasing Complexity: As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

In other words, we are kidding ourselves if we think that we can know all the requirements and build the perfect system. The best we can hope for is to build a useful system that will survive long enough for it to be asked to do something new.

What is this book?

This book came into being as a consequence of the realization that the most interesting and challenging side of software engineering may not be building brand new software systems, but rejuvenating existing ones.
From November 1996 to December 1999, we participated in a European industrial research project called FAMOOS (ESPRIT Project 21975 — Framework-based Approach for Mastering Object-Oriented Software Evolution). The partners were Nokia (Finland), Daimler-Benz (Germany), Sema Group (Spain), Forschungszentrum Informatik Karlsruhe (Germany), and the University of Berne (Switzerland). Nokia and Daimler-Benz were both early adopters of object-oriented technology, and had expected to reap significant benefits from this tactic. Now, however, they were experiencing many of the typical problems of legacy systems: they had very large, very valuable, object-oriented software systems that were very difficult to adapt to changing requirements. The goal of the FAMOOS project was to develop tools and techniques to rejuvenate these object-oriented legacy systems so they would continue to be useful and would be more amenable to future changes in requirements.

Our idea at the start of the project was to convert these big, object-oriented applications into frameworks — generic applications that can be easily reconfigured using a variety of different programming techniques. We quickly discovered, however, that this was easier said than done. Although the basic idea was sound, it is not so easy to determine which parts of the legacy system should be converted, and exactly how to convert them. In fact, it is a non-trivial problem just to understand the legacy system in the first place, let alone figuring out what (if anything) is wrong with it.

We learned many things from this project. We learned that, for the most part, the legacy code was not bad at all. The only reason that there were problems with the legacy code was that the requirements had changed since the original system was designed and deployed. Systems that had been adapted many times to changing requirements suffered from design drift — the original architecture and design was almost impossible to recognize — and that made it almost impossible to make further adaptations, exactly as predicted by Lehman and Belady’s laws of software evolution.

Most surprising to us, however, was the fact that, although each of the case studies we looked at needed to be reengineered for very different reasons — such as unbundling, scaling up requirements, porting to new environments, and so on — the actual technical problems with these systems were oddly similar. This suggested to us that perhaps a few simple techniques could go a long way to fixing some of the more common problems.

We discovered that pretty well all reengineering activity must start with some reverse engineering, since you will not be able to trust the documentation (if you are lucky enough to have some). Basically you can analyze the source code, run the system, and interview users and developers to build a model of the legacy system. Then you must determine what are the obstacles to further progress, and fix them. This is the essence of reengineering, which seeks to transform a legacy system into the system you would have built if you had the luxury of hindsight and could have known all the new requirements that you know today. But since you can’t afford to rebuild everything, you must cut corners and just reengineer the most critical parts.

Since FAMOOS, we have been involved in many other reengineering projects, and have been able to further validate and refine the results of FAMOOS.

In this book we summarize what we learned in the hope that it will help others who need to reengineer object-oriented systems. We do not pretend to have all the answers, but we have identified a series of simple techniques that will take you a long way.
Why patterns?

A pattern is a recurring motif, an event or structure that occurs over and over again. Design patterns are generic solutions to recurring design problems [Gamm95a]. It is because these design problems are never exactly alike, but only very similar, that the solutions are not pieces of software, but documents that communicate best practice.

Patterns have emerged in recent years as a literary form that can be used to document best practice in solving many different kinds of problems. Although many kinds of problems and solutions can be cast as patterns, they can be overkill when applied to the simplest kinds of problems. Patterns as a form of documentation are most useful and interesting when the problem being considered entails a number of conflicting forces, and the solution described entails a number of tradeoffs. Many well-known design patterns, for example, introduce run-time flexibility at the cost of increased design complexity.

This book documents a catalogue of patterns for reverse engineering and reengineering legacy systems. None of these patterns should be applied blindly. Each patterns resolves some forces and involves some tradeoffs. Understanding these tradeoffs is essential to successfully applying the patterns. As a consequence the pattern form seems to be the most natural way to document the best practices we identified in the course of our reengineering projects.

A pattern language is a set of related patterns that can be used in combination to solve a set of complex problems. We found that clusters of patterns seemed to function well in combination with each other, so we have organized this book into chapters that each presents such a cluster as a small pattern language.

We do not pretend that these clusters are “complete’ in any sense, and we do not even pretend to have patterns that cover all aspects of reengineering. We certainly do not pretend that this book represents a systematic method for object-oriented reengineering. What we do claim is simply to have encountered and identified a number of best practices that exhibit interesting synergies. Not only is there strong synergy within a cluster of patterns, but the clusters are also interrelated in important ways. Each chapter therefore contains not only a pattern map that suggests how the patterns may function as a “language”, but each pattern also lists and explains how it may be combined or composed with other patterns, whether in the same cluster or a different one.

Who should read this book?

This book is addressed mainly to practitioners who need to reengineer object-oriented systems. If you take an extreme viewpoint, you could say that every software project is a reengineering project, so the scope of this book is quite broad.

We believe that most of the patterns in this book will be familiar to anyone with a bit of experience in object-oriented software development. The purpose of the book is to document the details.
Acknowledgments

We would like to thank first and foremost our FAMOOS partners at Nokia, Daimler, FZI and Sema who provided the context for discovering these patterns. Persons like Juha (Julho) Tuominen, Roland Trauter, Eduardo Casais and Theo Dirk Meijler played a crucial role while starting the project. We would especially like to thank our co-authors of the prototype for this book, *The FAMOOS Object-Oriented Reengineering Handbook*: Holger Bär, Markus Bauer, Oliver Ciupke, Michele Lanza, Radu Marinescu, Robb Nebbe, Michael Przybilski, Tamar Richner, Matthias Rieger, Claudio Riva, Anne-Marie Sassen, Benedikt Schulz, Patrick Steyaert, Sander Tichelaar and Joachim Weisbrod.

We gratefully acknowledge the financial support of the European Union towards ESPRIT project 21975 (FAMOOS) as well as that of the Swiss Government towards projects NFS-2000-46947.96 and BBW-96.0015. The University of Antwerp provided financial support in terms of a grant entitled “Object Oriented Reengineering” while the Fund for Scientific Research in Flanders sponsored by means of a research network named “Foundations of Software Evolution”.

Some of the material in this book was presented in the graduate course “Object-Oriented software Reengineering” held at the University of Berne in the winter semesters of 1998 and 1999, and at several tutorials at OOPSLA. We would like to thank the participants of the courses and tutorials for their feedback and input. We also would like to thank members of the Software Composition Group at the University of Berne for participating in several pattern workshops and giving valuable feedback on many of the patterns in this book: Michele Lanza, Pietro Malorgio, Robbe Nebbe, Tamar Richner, Matthias Rieger and Sander Tichelaar.

Several of the patterns in this book have been presented elsewhere. We would like to thank our EuroPLoP shepherds Kent Beck (1998), Kyle Brown (1999), Neil Harrison (2000), Mary Lynn Manns (2000), Don Roberts (1998) and Charles Weir (1998) and all participants of the writers’ workshops where these patterns have been discussed. Special thanks go to Jens Coldewey for helping us out with pattern forms and forces.

We would like to thank the members and friends of Ralph Johnson’s Software Architecture Group who workshoped several chapters of this book: John Brant, Brian Foote, Alejandra Garrido, Peter Hatch, Ralph Johnson, Brian Marick, Andrew Rosenfeld, Weerasak Witthawaskul and Joe Yoder. Downloading and playing voluminous megabytes of workshop recordings in mp3 format truly made each of us feel like a “fly on the wall”!

We would like to thank Tim Cox, our editor, and Stacie Pierce, his assistant, both at Morgan Kaufmann, for following our project with such dedication. Also, thanks to Christa Preisendanz at DPunkt Verlag for putting us in touch with Tim in the first place! We especially appreciated the two very thorough rounds of reviews that this book underwent, and we only regret that the final draft of this book is nothing like the definitive work some of reviewers clearly hoped it would be! We thank our reviewers for reading between the lines and helping to explain many of these patterns to us: Kyle Brown, Thierry Cattel, Oliver Ciupke, Koen De Hondt, Jim Coplien, Gert Florijn, Neil Harrison, Mary Lynn Manns, Alan O’Callaghan, Don Roberts and Benedikt Schulz.

*Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz*
Part I

Introduction
Chapter 1

Reengineering Patterns

1. Why do we Reengineer?

A legacy is something valuable that you have inherited. Similarly, legacy software is valuable software that you have inherited. The fact you have inherited it may mean that it is somewhat old-fashioned. It may have been developed using an outdated programming language, or an obsolete development method. Most likely it has changed hands several times, and shows signs of many modifications and adaptations.

Perhaps your legacy software is not even that old. With rapid development tools and rapid turnover in personnel, software systems can turn into legacies more quickly than you might imagine. The fact that the software is valuable, however, means that you do not just want to throw it away.

A piece of legacy software is critical to your business, and that is precisely the source of all the problems: in order for you to be successful at your business, you must constantly be prepared to adapt to a changing business environment. The software that you use to keep your business running must therefore also be adaptable. Fortunately a lot of software can be upgraded, or simply thrown away and replaced when it no longer serves its purpose. But a legacy system can neither be replaced nor upgraded except at a high cost. The goal of reengineering is to reduce the complexity of a legacy system sufficiently that it can continue to be used and adapted at an acceptable cost.

The specific reasons that you might want to reengineer a software system can vary significantly. For example:

- You might want to unbundle a monolithic system so that the individual parts can be more easily marketed separately or combined in different ways.
- You might want to improve performance. (Experience shows that the right sequence is “first do it, then do it right, then do it fast”, so you might want to reengineer to clean up the code before thinking about performance.)
- You might want to port the system to a new platform. Before you do that, you may need to rework the architecture to clearly separate the platform-dependent code.
- You might want to extract the design as a first step to a new implementation.
- You might want to exploit new technology, such as emerging standards or libraries, as a step towards cutting maintenance costs.
- You might want to reduce human dependencies by documenting knowledge about the system and making it easier to maintain.

Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Though there may be many different reasons for reengineering a system, as we shall see, however, the actual technical problems with legacy software are often very similar. It is this fact that allows us to use some very general techniques to do at least part of the job of reengineering.

**Recognizing the need to reengineer**

How do you know when you have a legacy problem?

Common wisdom says, “If it ain’t broke, don’t fix it.” This attitude is often taken as an excuse not to touch any piece of software that is performing an important function and seems to be doing it well. The problem with this approach is that it fails to recognize that there are many ways in which something may be “broken”. From a functional point of view, something is broken only if it no longer delivers the function it is designed to perform. From a maintenance point of view, however, a piece of software is broken if it can no longer be maintained.

So how can you tell that your software is going to break very soon? Fortunately there are many warning signs that tell you that you are headed towards trouble. The symptoms listed below usually do not occur in isolation but several at a time.

**Obsolete or no documentation.** Obsolete documentation is a clear sign of a legacy system that has undergone many changes. Absence of documentation is a warning sign that problems are on the horizon, as soon as the original developers leave the project.

**Missing tests.** Even more important than up-to-date documentation is the presence of thorough unit tests for all system components, and system tests that cover all significant use cases and scenarios. The absence of such tests is a sign that the system will not be able to evolve without high risk or cost.

**Original developers or users have left.** Unless you have a clean, well-documented system with good test coverage, it will rapidly deteriorate into an even less clean, more poorly documented system.

**Inside knowledge about system has disappeared.** This is a bad sign. The documentation is out of sync with the existing code base. Nobody really knows how it works.

**Limited understanding of the entire system.** Not only does nobody understand the fine print, but hardly anyone has a good overview of the whole system.

**Too long to turn things over to production.** Somewhere along the line the process is not working. Perhaps it takes too long to approve changes. Perhaps automatic regression tests are missing. Or perhaps it is difficult to deploy changes. Unless you understand and deal with the difficulties it will only get worse.

**Too much time to make simple changes.** This is a clear sign that Lehman and Belady’s Law of Increasing Complexity has kicked in: the system is now so complex that even simple changes are hard to implement. If it takes too long to make simple changes to your system, it will certainly be out of the question to make complex changes. If there is a backlog of simple changes waiting to get done, then you will never get to the difficult problems.

**Need for constant bug fixes.** Bugs never seem to go away. Every time you fix a bug, a new one pops up next to it. This tells you that parts of your application have become so complex,
that you can no longer accurately assess the impact of small changes. Furthermore, the architecture of the application no longer matches the needs, so even small changes will have unexpected consequences.

**Maintenance Dependencies.** When you fix a bug in one place, another bug pops up *somewhere else*. This is often a sign that the architecture has deteriorated to the point where logically separate components of the system are no longer independent.

**Big build times.** Long recompilation times slow down your ability to make changes. Long build times may also be telling you that the organization of your system is too complex for your compiler tools to do their job efficiently.

**Difficulties separating products.** If there are many clients for your product, and you have difficulty tailoring releases for each customer, then your architecture is no longer right for the job.

**Duplicated code.** Duplicated code arises naturally as a system evolves, as shortcut to implementing nearly identical code, or merging different versions of a software systems. If the duplicated code is not eliminated by refactoring the common parts into suitable abstractions, maintenance quickly becomes a nightmare as the same code has to be fixed in many places.

**Code Smells.** Duplicated code is an example of code that “smells bad” and should be changed. Long methods, big classes, long parameter lists, switch statements and data classes are few more examples that have been documented by Kent Beck and others [Fowl99a]. Code smells are often a sign that a system has been repeatedly expanded and adapted without having been reengineered.

**What’s special about Objects?**

Although many of the techniques discussed in this book will apply to any software system, we have chosen to focus on *object-oriented legacy systems*. There are many reasons for this choice, but mainly we feel that this is a critical point in time at which many early adopters of object-oriented technology are discovering that the benefits they expected to achieve by switching to objects have been very difficult to realize.

There are now significant legacy systems even in Java. It is not age that turns a piece of software into a legacy system, but the *rate* at which it has been developed and adapted without having been reengineered.

The wrong conclusion to draw from these experiences is that “objects are bad, and we need something else”. Already we are seeing a rush towards many new trends that are expected to save the day: patterns, components, UML, XMI, and so on. Any one of these developments may be a Good Thing, but in a sense they are all missing the point.

One of the conclusions you should draw from this book is that, well, objects are pretty good, but you must take good care of them. To understand this point, consider why legacy problems arise at all with object-oriented systems, if they are supposed to be so good for flexibility, maintainability and reuse.

First of all, anyone who has had to work with a non-trivial, existing object-oriented code base will have noticed: *it is hard to find the objects*. In a very real sense, the architecture of an object-
oriented application is usually hidden. What you see is a bunch of classes and an inheritance hierarchy. But that doesn’t tell you which objects exist at run-time and how they collaborate to provide the desired behavior. Understanding an object-oriented system is a process of reverse engineering, and the techniques described in this book help to tackle this problem. Furthermore, by reengineering the code, you can arrive at a system whose architecture is more transparent, and easier to understand.

Second, anyone who has tried to extend an existing object-oriented application will have realized: reuse does not come for free. It is actually very hard to reuse any piece of code unless a fair bit of effort was put into designing it so that it could be reused. Furthermore, it is essential that investment in reuse requires management commitment to put the right organizational infrastructure in place, and should only be undertaken with clear, measurable goals in mind [Gold95a].

We are still not very good at managing object-oriented software projects in such a way that reuse is properly taken into account. Typically reuse comes too late. We use object-oriented modelling techniques to develop very rich and complex object models, and hope that when we implement the software we will be able to reuse something. But by then there is little chance that these rich models will map to any kind of standard library of components except with great effort. Several of the reengineering techniques we present address how to uncover these components after the fact.

The key insight, however, is that the “right” design and organization of your objects is not something that is or can be evident from the initial requirements alone, but rather as a consequence of understanding how these requirements evolve. The fact that the world is constantly changing should not be seen purely as a problem, but as the key to the solution.

Any successful software system will suffer from the symptoms of legacy systems. Object-oriented legacy systems are just successful object-oriented systems whose architecture and design no longer responds to changing requirements. A culture of continuous reengineering is a prerequisite for achieving flexible and maintainable object-oriented systems.

2. The Reengineering Lifecycle

Reengineering and reverse engineering are often mentioned in the same context, and the terms are sometimes confused, so it is worthwhile to be clear about what we mean by them. Chikofsky and Cross [Chik92a] define the two terms as follows:

“Reverse Engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

That is to say, reverse engineering is essentially concerned with trying to understand a system and how it ticks.

“Reengineering ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

Object-Oriented Reengineering Patterns © 2002 Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Reengineering, on the other hand, is concerned with restructuring a system, generally to fix some real or perceived problems, but more specifically in preparation for further development and extension.

The introduction of term “reverse engineering” was clearly an invitation to define “forward engineering”, so we have the following as well:

“Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.”

How exactly this process of forward engineering can or should work is of course a matter of great debate, though most people accept that the process is iterative, and conforms to Barry Boehm’s so-called spiral model of software development [Boeh88a]. In this model, successive versions of a software system are developed by repeatedly collecting requirements, assessing risks, engineering the new version, and evaluating the results. This general framework can accommodate many different kinds of more specific process models that are used in practice.

If forward engineering is about moving from high-level views of requirements and models towards concrete realizations, then reverse engineering is about going backwards from some concrete realization to more abstract models, and reengineering is about transforming concrete implementations to other concrete implementations.

Figure 1 illustrates this idea. Forward engineering can be understood as being a process that moves from high-level and abstract models and artifacts to increasing concrete ones. Reverse engineering reconstructs higher-level models and artifacts from code. Reengineering is a pro-
cess that transforms one low-level representation to another, while recreating the higher-level artifacts along the way.

The key point to observe is that reengineering is not simply a matter of transforming source code, but of transforming a system at all its levels. For this reason it makes sense to talk about reverse engineering and reengineering in the same breath. In a typical legacy system, you will find that not only the source code, but all the documentation and specifications are out of sync. Reverse engineering is therefore a prerequisite to reengineering since you cannot transform what you do not understand.

**Reverse engineering**

You carry out reverse engineering whenever you are trying to understand how something really works. Normally you only need to reverse engineer a piece of software if you want to fix, extend or replace it. (Sometimes you need to reverse engineer software just in order to understand how to use it. This may also be a sign that some reengineering is called for.) As a consequence, reverse engineering efforts typically focus on redocumenting software and identifying potential problems, in preparation for reengineering.

You can make use of a lot of different sources of information while reverse engineering. For example, you can:

- read the existing documentation
- read the source code
- run the software
- interview users and developers
- code and execute test cases
- generate and analyze traces
- use various tools to generate high-level views of the source code and the traces
- analyze the version history

As you carry out these activities, you will be building progressively refined models of the software, keeping track of various questions and answers, and cleaning up the technical documentation. You will also be keeping an eye out for problems to fix.

**Reengineering**

Although the reasons for reengineering a system may vary, the actual technical problems are typically very similar. There is usually a mix of coarse-grained, architectural problems, and fine-grained, design problems. Typical coarse-grained problems include:

- **Insufficient documentation**: documentation either does not exist, or is inconsistent with reality.
- **Improper layering**: missing or improper layering hampers portability and adaptability.
- **Lack of modularity**: strong coupling between modules hampers evolution.
- **Duplicated code**: “copy, paste and edit” is quick and easy, but leads to maintenance nightmares.
The most common fine-grain problems occurring in object-oriented software include:

- **Misuse of inheritance**: for composition, code reuse rather than polymorphism
- **Missing inheritance**: duplicated code, and case statements to select behavior
- **Misplaced operations**: unexploited cohesion — operations outside instead of inside classes
- **Violation of encapsulation**: explicit type-casting, C++ “friends”.
- **Class abuse**: lack of cohesion — classes as namespaces

Finally, you will be preparing the code base for the reengineering activity by developing exhaustive test cases for all the parts of the system that you plan to change or replace.

Reengineering similarly entails a number of interrelated activities. Of course, one of the most important is to evaluate which parts of the system should be repaired and which should be replaced.

The actual code transformations that are performed fall into a number of categories. According to Chikofsky and Cross:

> “Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the system’s external behavior.”

Restructuring generally refers to source code translation (such as the automatic conversion from unstructured “spaghetti” code to structured, or “goto-less”, code), but it may also entail transformations at the design level.

**Refactoring** is restructuring within an object-oriented context. Martin Fowler defines it this way:

> “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”  [Fowl99a]

It may be hard to tell the difference between software “reengineering” and software “maintenance”. IEEE has made several attempts to define software maintenance, including this one:

> “the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment”

Most people would probably consider that “maintenance” is routine whereas “reengineering” is a drastic, major effort to recast a system, as suggested by figure 1.

Others, however, might argue that reengineering is just a way of life. You develop a little, re-engineer a little, develop a little more, and so on [Beck00a]. In fact, there is good evidence to support the notion that a culture of *continuous* reengineering is necessary to obtain healthy, maintainable software systems.

Continuous reengineering, however, is not yet common practice, and for this reason we present the patterns in this book in the context of a major reengineering effort. Nevertheless, the reader should keep in mind that most of the techniques we present will apply just as well when you reengineer in small iterations.
3. Reengineering Patterns

Patterns as a literary form were introduced by the architect Christopher Alexander in his landmark 1977 book, *A Pattern Language*. In this book, Alexander and his colleagues presented a systematic method for architecting a range of different kinds of physical structures, from rooms to buildings and towns. Each issue was presented as a recurring pattern, a general solution which resolves a number of forces, but must be applied in a unique way to each problem according to the specific circumstances. The actual solution presented in each pattern was not necessarily so interesting, but rather the discussion of the forces and tradeoffs consisted of the real substance they communicated.

Patterns were first adopted by the software community as a way of documenting recurring solutions to design problems. As with Alexander’s patterns, each design pattern entailed a number of forces to be resolved, and a number of tradeoffs to consider when applying the pattern. Patterns turn out to be a compact way to communicate best practice: not just the actual techniques used by experts, but the motivation and rationale behind them. Patterns have since been applied to many aspects of software development other than design, and particularly to the process of designing and developing software.

The process of reengineering is, like any other process, one in which many standard techniques have emerged, each of which resolves various forces and may entail many tradeoffs. Patterns as a way of communicating best practice are particularly well-suited to presenting and discussing these techniques.

*Reengineering patterns* codify and record knowledge about modifying legacy software: they help in diagnosing problems and identifying weaknesses which may hinder further development of the system, and they aid in finding solutions which are more appropriate to the new requirements. We see reengineering patterns as stable units of expertise which can be consulted in any reengineering effort: they describe a process without proposing a complete methodology, and they suggest appropriate tools without “selling” a specific one.

Many of the reverse engineering and reengineering patterns have some superficial resemblance to design patterns, in the sense that they have something to do with the design of software. But there is an importance difference in that design patterns have to do with choosing a particular solution to a design problem, whereas reengineering patterns have to do with *discovering an existing design*, determining what problems it has, and *repairing* these problems. As a consequence, reengineering patterns have more to do with the *process of discovery and transformation* than purely with a given design structure. For this reason the names of most of the patterns in this book are process-oriented, like *Always Have a Running Version* [p. 138], rather than being structure-oriented, like *Adapter* [p. 228] or *Facade* [p. 228].

Whereas a design pattern presents a solution for a recurring design problem, a reengineering pattern presents a solution for a recurring reengineering problem. The artifacts produced by reengineering patterns are not necessarily designs. They may be as concrete as refactored code, or in the case of reverse engineering patterns, they may be abstract as insights into how the system functions.
The mark of a good reengineering pattern is (a) the clarity with which it exposes the advantages, the cost and the consequences of the target artifacts with respect to the existing system state, and not how elegant the result is, (b) the description of the reengineering process: how to get from one state of the system to another.

Reengineering patterns entail more than code refactorings. A reengineering pattern may describe a process which starts with the detection of the symptoms and ends with the refactoring of the code to arrive at the new solution. Refactoring is only the last stage of this process, and addresses the technical issue of automatically or semi-automatically modifying the code to implement the new solution. Reengineering patterns also include other elements which are not part of refactorings: they emphasize the context of the symptoms, by taking into account the constraints that reengineers are facing, and include a discussion of the impact of the changes that the refactored solution may introduce.

4. The Form of a Reengineering Pattern

In figure 2 we see an example of a simple pattern that illustrates the format we use in this book. The actual format used may vary slightly from pattern to pattern, since they deal with different kinds of issues, but generally we will see the same kind of headings.

The name of a pattern, if well-chosen, should make it easy to remember the pattern and to discuss it with colleagues. (“I think we should Refactor to Understand or we will never figure out what’s going on here.”) The intent should communicate very compactly the essence of a pattern, and tell you whether it applies to your current situation.

Many of the reengineering patterns are concerned with code transformation, in which case a diagram may be used to illustrate the kind of transformation that takes place. Typically such patterns will additionally include steps to detect the problem to be resolved, as well as code fragments illustrating the situation before and after the transformation.
If It Ain’t Broke, Don’t Fix It

Intent: Save your reengineering effort for the parts of the system that will make a difference.

Problem
Which parts of a legacy system should you reengineer?

This problem is difficult because:

• Legacy software systems can be large and complex.
• Rewriting everything is expensive and risky.

Yet, solving this problem is feasible because:

• Reengineering is always driven by some concrete goals.

Solution
Only fix the parts that are “broken” — that can no longer be adapted to planned changes.

Tradeoffs

Pros  You don’t waste your time fixing things that are not only your critical path.

Cons  Delaying repairs that do not seem critical may cost you more in the long run.

Difficulties  It can be hard to determine what is “broken”.

Rationale
There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

Known Uses
Alan M. Davis discusses this in his book, 201 Principles of Software Development.

Related Patterns
Be sure to Fix Problems, Not Symptoms.

What Next
Consider starting with the Most Valuable First.

Figure 2: The format of a typical reengineering pattern
5. A Map of Reengineering Patterns

The patterns in this book are organized according to the reengineering lifecycle presented earlier. In figure 3 we can see the chapters in this book represented as clusters of patterns along the lifecycle. The diagram suggests that the patterns may be applied in sequence. Though this may well be the case, in practice you are more likely to iterate between reverse engineering and reengineering tasks. The diagram is simplistic in the same sense that the “waterfall” lifecycle is simplistic: it may be a useful way to keep track of the different software engineering activities and their relationships, even though we know that they are not carried out sequentially but iteratively.

Each cluster of patterns is presented as a simple “pattern language” — a set of related patterns that may be combined to address a common set of problems. As such, each chapter will typically start with an overview and a map of the patterns in that chapter, suggesting how they may be related.

Setting Direction contains several patterns to help you determine where to focus your reengineering efforts, and make sure you stay on track. First Contact consists of a set of patterns that may be useful when you encounter a legacy system for the first time. Initial Understanding helps you to develop a first simple model of a legacy system, mainly in the form of class di-
ograms. **Detailed Model Capture** helps you to develop a more detailed model of a particular component of the system.

**Tests: Your Life Insurance** focuses on the use of testing not only to help you understand a legacy system, but also to prepare it for a reengineering effort. **Migration Strategies** help you keep a system running while it is being reengineered, and increase the chances that the new system will be accepted by its users. **Detecting Duplicated Code** can help you identify locations where code may have been copied and pasted, or merged from different versions of the software. **Redistribute Responsibilities** helps you discover and reengineer classes with too many responsibilities. **Transform Conditionals to Polymorphism** will help you to redistribute responsibilities when an object-oriented design has been compromised over time.