

Object-Oriented Reengineering: Patterns and Techniques

Serge Demeyer

University of Antwerp, Lab On REengineering (LORE)
<http://www.lore.ua.ac.be/>

Stéphane Ducasse, Oscar Nierstrasz

University of Berne, Software Composition Group (SCG)
<http://www.iam.unibe.ch/ducasse/>

Abstract

Surprising as it may seem, many of the early adopters of the object-oriented paradigm already face a number of problems typically encountered in large-scale legacy systems. Software engineers are now confronted with millions of lines of industrial source code, developed using object-oriented design methods and languages of the late 80s and early 90s. These systems exhibit a range of problems, effectively preventing them from satisfying the evolving requirements imposed by their customers.

This tutorial will share our knowledge concerning the reengineering of object-oriented legacy systems. We will draw upon our experiences, to show you techniques and tools we have applied on real industrial OO systems to detect and repair problems. In particular, we will discuss issues like reverse engineering, design extraction, metrics, refactoring and program visualisation.

1. Introduction

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:

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.

2 What is this tutorial ?

This tutorial (and the accompanying 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 un-bundling, 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 tutorial 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.

3 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. 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 tutorial explains a catalogue of patterns for reverse engineering and reengineering legacy systems. None of these patterns should be applied blindly. Each pattern 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.

We do not pretend that our catalogue of patterns is “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 these patterns 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. And by cataloguing them, we hope to help reengineers all over the world in their daily struggle to revive their legacy systems.