Object-oriented Reengineering Patterns An Overview[⋆]

Oscar Nierstrasz[1], Stéphane Ducasse[2], and Serge Demeyer[3]

[1] Software Composition Group, University of Bern, Switzerland.
[2] Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance, Université de Savoie, France.
[3] Lab On REengineering, University of Antwerp, Belgium.

Abstract. Successful software systems must be prepared to evolve or they will die. Although object-oriented software systems are built to last, over time they degrade as much as any legacy software system. As a consequence, one must invest in reengineering efforts to keep further development costs down. Even though software systems and their business contexts may differ in countless ways, the techniques one uses to understand, analyze and transform these systems tend to be very similar. As a consequence, one may identify various *reengineering patterns* that capture best practice in reverse- and re-engineering object-oriented legacy systems. We present a brief outline of a large collection of these patterns that have been mined over several years of experience with object-oriented legacy systems, and we indicate how some of these patterns can be supported by appropriate tools.

1 Introduction

A *legacy software system* is a system that you have *inherited* and is *valuable* to you. Successful (*i.e.*, valuable) software systems typically evolve over a number of years as requirements evolve and business needs change. This leads to the well-documented phenomenon that such systems become more *complex* over time, and become progressively harder to maintain, unless special measures are taken to simplify their architecture and design [13].

Numerous problems manifest themselves as a legacy system begins to turn into a burden. First of all, *knowledge* about the system deteriorates. Documentation is often missing or obsolete. The original developers or users may have left the project. As a consequence, inside knowledge about the system may be missing. Automated tests that document how the system functions are rarely available.

Second, the *process* for implementing changes ceases to be effective. Simple changes take too long. A continuous stream of bug fixes is common. Maintenance dependencies make it difficult to implement changes or to separate products.

[⋆] ©Springer Verlag, 2005. Invited paper, Proceedings of GPCE 2005, Michael Lowry Robert Glück (Ed.), LNCS 3676, 2005, pp. 1-9.

Finally, the *code* itself will exhibit various disagreeable symptoms. Large amounts of duplicated code are common, as are other "code smells" such as violations of encapsulation, large, procedural classes, and explicit type checks.

Concretely, the code will manifest *architectural problems* such as improper layering and lack of modularity, as well as *design problems* such as misuse of inheritance, missing inheritance and misplaced operations. Excessive build times are also a common sign of architectural decay.

Since the bulk of a (successful) software system's life cycle is known to reside in maintenance, and "maintenance" is known to consist largely in the introduction of new functionality [14], identifying and resolving these problems becomes critical for the survival of legacy systems.

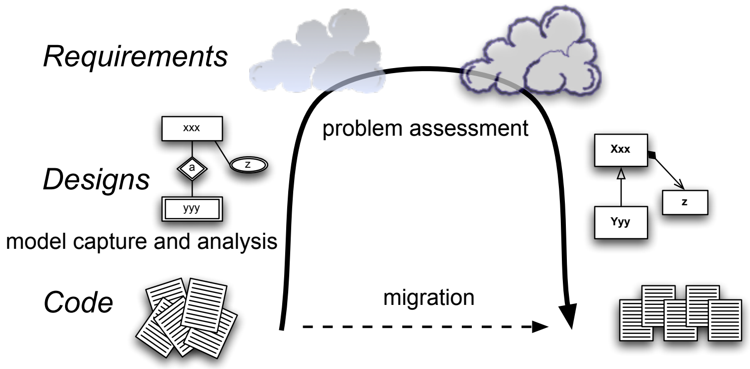

Fig. 1. The Reengineering life cycle.

To this end, it is useful to distinguish *reverse engineering* from *reengineering* of software systems [2]. By "reverse engineering", we mean the process of analyzing a software system in order to expose its structure and design at a higher level of abstraction, *i.e.*, the process of extracting various *models* from the concrete software system. By "reengineering" we refer to the process of transforming the system to a new one that implements essentially the same functional requirements, but also enables further development.

The process of reverse- and re-engineering consists of numerous activities, including architecture and design recovery, test generation, problem detection, and various high and low-level refactorings. In Figure 1 we see an ideal depiction of the reverse- and re-engineering life cycle [3, 10].

Although the motivations for reengineering a legacy system may vary considerably according to the business needs of the organization, the actual technical steps taken tend to be very similar. As a consequence, it is possible to identify a number of generally useful *process patterns* that one may apply while reverse- and re-engineering a legacy system. We provide a brief overview of these patterns in Section 2. By the same token, there exist various tools that can help support the reengineering process. In Section 3 we present a brief outline of some of the tools we have developed and applied to various legacy systems.

2 Reengineering Patterns

The term "pattern" used in the context of software usually evokes the notion of "design patterns" — recurring solutions to design problems. *Reengineering patterns* are not design patterns, but rather *process patterns* — recurring solutions to problems that arise during the process of reverse- and re-engineering.

We distinguish patterns from "rules" or "guidelines" because each pattern must be interpreted in a given context. Patterns are not applied blindly, but entail tradeoffs. Just as one would never deliberately implement a software system applying all of the GOF patterns [7], one should not blindly apply reengineering patterns without considering all the consequences.

We were able to mine a large number of reengineering patterns during the course of FAMOOS, a European project[4] whose goal was to support the evolution of first-generation object-oriented software towards object-oriented frameworks. FAMOOS focussed on methods and tools to analyse and detect design problems in object-oriented legacy systems, and to migrate these systems towards more flexible architectures. The main results of FAMOOS are summarized in the FAMOOS Handbook [4] and in the book "Object-Oriented Reengineering Patterns" [3].

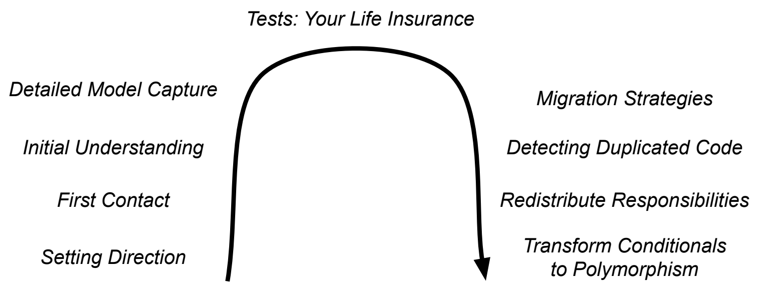
Tests: Your Life Insurance
Detailed Model Capture
Initial Understanding
First Contact
Setting Direction
Migration Strategies
Detecting Duplicated Code
Redistribute Responsibilities
Transform Conditionals to Polymorphism

Fig. 2. Reengineering pattern clusters.

In Figure 2 we see how various clusters of reengineering patterns can be mapped to our ideal reengineering life cycle. Each name represents a collection of process patterns that can be applied at a particular stage during the reengineering of a legacy system.

[4] ESPRIT Project 21975: "Framework-based Approach for Mastering Object-Oriented Software Evolution". `www.iam.unibe.ch/~scg/Archive/famoos`

Setting Direction contains several patterns to help you determine where to focus your re- engineering 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 diagrams. *Detailed Model Capture* helps you to develop a more detailed model of a particular component of the system. *Tests: Your Life Insurance* focusses 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.

Since a detailed description of the patterns is clearly out of the scope of a short paper, let us just briefly consider a single pattern cluster. *First Contact* consists of patterns that can be useful when first encountering a legacy system. There are various *forces* at play, which one must be conscious of. In particular, legacy systems tend to be *large* and complex, so it will be difficult to get an overview of the system. *Time is short*, so it is important to gather quality information quickly. Furthermore, *first impressions are dangerous*, so it is important not to rely on a single source of information.

One has various resources at hand: the source code, the running system, the users, the maintainers, documentation, the source code repository, the changes log, the list of bug requests, the test cases, and so on. Even if some of these are missing or unreliable, one must take care to not reject anything out of hand.

In Figure 3 we see a map of the patterns in this cluster, and how they relate to each other. As with each pattern cluster, patterns support each other to resolve the forces at play. The *First Contact* cluster resolves the forces by balancing what you learn from the users and maintainers with what you learn from the source code.

In Figure 4 we see a capsule summary of one of the better-known patterns of this cluster. The *name* is typically an action to be performed, that expresses the key idea of the pattern. Not every pattern is always relevant in every context, so one must be clear about the *intent* of each pattern, the *problem* it solves, the key idea of the *solution*, and the *tradeoffs* entailed. In this particular pattern, the context of a demo is used as a device to help the user to focus on concrete rather than abstract qualities of the application, while communicating typical use cases and scenarios to the engineer. Each pattern may also include *hints*, *variants*, *examples*, *rationale*, *related patterns*, and an indication of *what to do next*. *Known uses* are very important, since only established best practices can truly be considered “patterns”.

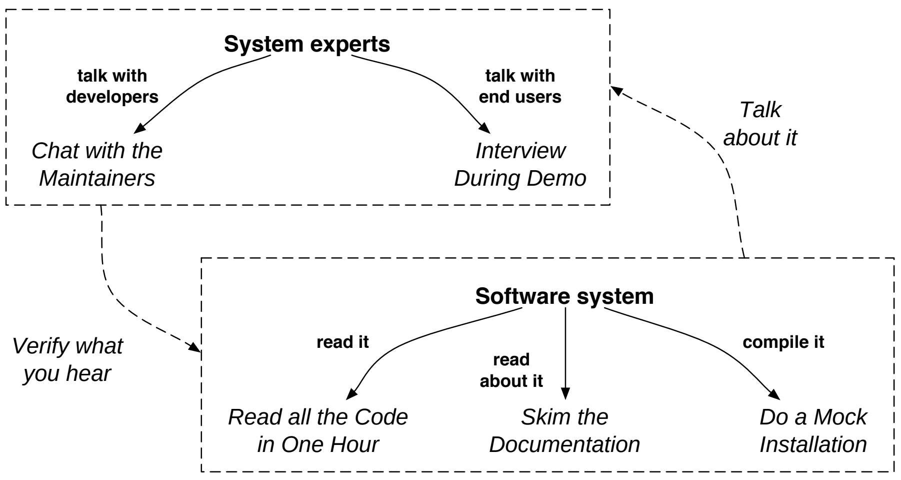

Fig. 3. First Contact.

Name	*Interview During Demo*
Intent	Obtain an initial feeling for the appreciated functionality of a software system by seeing a demo and interviewing the person giving the demo.
Problem	How can you get an idea of the typical usage scenarios and the main features of a software system?
Solution	Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Note that the interviewing part is at least as enlightening as the demo.
Hints	The user who is giving the demo is crucial to the outcome of this pattern so take care when selecting the person. Therefore, do the demonstration several times with different persons giving the demo.
Tradeoffs	*Pro:* Focuses on valued features. *Con:* Provides anecdotal evidence only. *Difficulties:* Requires interviewing experience.
Example	*(Description of a typical interview ...)*
Rationale	Because users must start from a working system, they will adopt a positive attitude in explaining what works. The interviewer can ask precise questions, get precise answers, thus digging out the expert knowledge about the system's usage.
Known Uses	Commonly used for evaluating user-interfaces.
Related Patterns	See *Customer Interaction Patterns* [17]
What Next	Carry out several attempts of *Interview During Demo* with different kinds of stakeholders. Perform these attempts before, after or interwoven with *Read all the Code in One Hour* and *Skim the Documentation.* Afterwards, consider to *Chat with the Maintainers* to verify some of your findings.

Fig. 4. A pattern in a nutshell.

3 Reengineering Tools and Techniques

It is easy to put too much faith into tools. For this reason the reengineering patterns put more emphasis on process than tools. (As a popular saying puts it: "A fool with a tool is still a fool.")

Nevertheless, certain activities can be streamlined with the help of carefully chosen tools. In particular, the process of reverse engineering can be aided by tools that build models from source code. Note that it is *not* a question of generating UML diagrams from source code. (10'000 class diagrams do not necessarily aid program comprehension more than 1'000'000 lines of source code.)

One the other hand, during *Initial Understanding*, a key pattern is *Study the Exceptional Entities*. Very often it is the software entities that are very large, very small, most tightly coupled, inherit the most, inherit the least, *etc.*, that tell one the most about how a software system works. It may be that these outliers are indicative of design problems, but this need not be the case.

CodeCrawler is a tool that presents simple visualizations of software entities based on direct metrics [12]. A polymetric view, is a two-dimensional visualization of nodes (as entities) and edges (as relationships) that maps various metric values to attributes of the nodes and edges. For example, different metrics can be mapped to the size, position and color of a node, or to the thickness and color of the edge.

Polymetric views can be generated for different purposes: coarse-grained views to assess global system properties, fine-grained views to assess properties of individual software artifacts, and evolutionary views to assess properties over time.

Figure 5 shows a System Complexity View which is coarse grained view [11]. The figure shows the hierarchies of CodeCrawler itself. Each node represents a class, and each edge represents an inheritance relationship. The height of a node represents the number of methods, the width represents the number of attributes and the (greyscale) color represents the number of lines of code. A System Complexity View can help one to quickly identify many kinds of outliers. For example, tall, isolated, dark nodes have many methods, many lines of code, and few attributes, and they may be signs of procedural classes with long, algorithmic methods.

CodeCrawler is built on top of Moose, a reengineering environment that offers a common infrastructure for various reverse- and re-engineering tools [5, 15]. At the core of Moose is a common meta-model for representing software systems in a language-independent way. Around this core are provided various services that are available to the different tools. These services include metrics evaluation and visualization, a repository for storing multiple models, a meta-meta model for tailoring the Moose meta-model, and a generic GUI for browsing, querying and grouping.

Some other tools that have been developed either in the context of Famoos, or subsequently as clients of Moose, include:

- Duploc— detects duplicated code in large software systems in a language-independent way [6, 16].

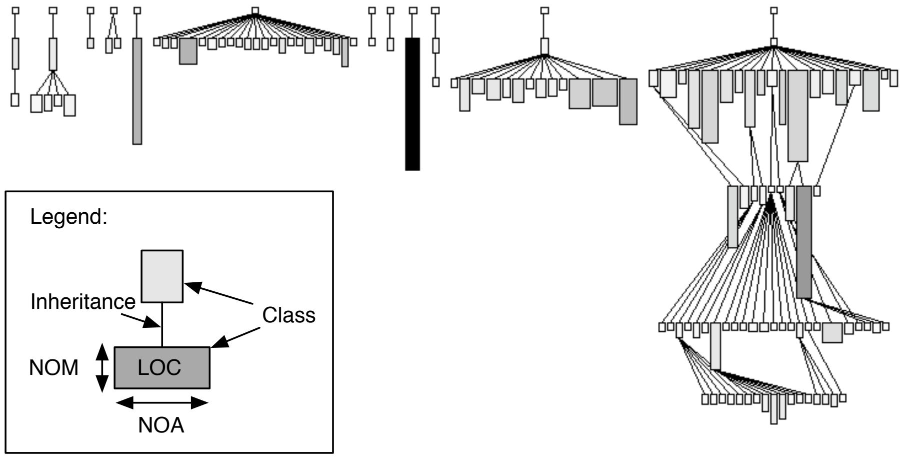

Fig. 5. A System Complexity view of CodeCrawler.

- ConAn— applies formal concept analysis to detect implicit contracts in object-oriented software [1].
- Van— analyzes version histories of software systems to uncover trends [8].
- TraceScraper— analyzes run-time traces of instrumented software to correlate features with software artifacts [9].

4 Conclusions

Given the premise that “the only constant is change”, any interesting software system must evolve to stay interesting. As a consequence, however, we must invest in reengineering if the architecture and design of the system is to stay abreast of the changing requirements. Even though every system is different, we can identify various useful *reengineering patterns* that ease the process of understanding a complex legacy system, identifying its problems, and transforming it to a more flexible design.

The patterns we have documented include only those for which we have personally witnessed success. The Famoos reengineering patterns therefore represent only a starting point, and not a definitive work. What is important is that each pattern document best practice as experienced by experts in the field, as opposed to new research ideas that have not yet been proven in industrial contexts. There is clearly much research that can be done to investigate, for example, the synergy between tools and reengineering patterns, but one must not confuse the two.

We hope that the value of reengineering patterns, and more generally process patterns, will increasingly be recognized and encouraged as an effective means to improve the state of the art and disseminate best practice.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006). Thanks are due to Laura Ponisio for suggesting several improvements in the text.

References

1. Gabriela Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis.* PhD thesis, University of Berne, January 2005.
2. Elliot J. Chikofsky and James H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.
3. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.
4. Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook.* University of Bern, October 1999.
5. Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.
6. Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance: Research and Practice*, 2005. To appear.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.
8. Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.
9. Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society Press, 2005.
10. R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.
11. Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.
12. Michele Lanza and Stéphane Ducasse. Codecrawler — an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005.
13. Manny M. Lehman and Les Belady. *Program Evolution – Processes of Software Change.* London Academic Press, 1985.

14. Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison Wesley, Boston, MA, 1980.
15. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Girba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*. LNCS, 2005. Invited paper. To appear.
16. Matthias Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Berne, June 2005.
17. Linda Rising. Customer interaction patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 585–609. Addison Wesley, 2000.