An Iterative Reengineering Process Applying Test-Driven Development and Reverse Engineering Patterns

Vinícius H. S. Durelli¹, Rosângela A. D. Penteado², Simone de Sousa Borges², Matheus C. Viana²

¹Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (ICMC-USP)
13560-970 – São Carlos – SP – Brasil

²Departamento de Computação
Universidade Federal de São Carlos (DC-UFSCar)
13565-905 – São Carlos – SP – Brasil

durelli@icmc.usp.br, {rosangel,simone_borges,matheus_viana}@dc.ufscar.br

Abstract. Nowadays, software technology is evolving quickly and therefore software systems which have been built upon some technologies are deprecated even before being released and used. Thus, software systems are in constant evolution in order to adapt themselves to the current technologies as well as users’ needs. An approach to revitalize software systems that have already been released is reengineering. In this paper, we propose an iterative reengineering process that uses reverse engineering patterns and test-driven development to cope with issues involved in migrating from the legacy system to an equivalent software system implemented in more recent technologies. As a preliminary evaluation of the proposed process, a large (more than 29 KLOC) legacy system was reengineered from Smalltalk to Java.

1. Introduction

Software systems undergo many modifications during their life cycle. The act of either improving or modifying an existing software system without introducing problems is quite a challenge. Reengineering is aimed at revitalizing software systems through fixing existing or perceived problems. However, unlike forward engineering that is supported by a plenty of processes, such as the spiral and waterfall models of software development, no established process for reengineering is available [Mens and Tourwé 2004].

Due to the absence of an established reengineering process, patterns and some techniques which have been in existence for a long time and are recognized and generally accepted can be combined and used within a reengineering context. Moreover, agile practices have been widely applied in many forward engineering efforts since their introduction, thus there is interest in determining the applicability of some agile practices to reengineering projects.

The novelty of our process is that reverse engineering activities use particular reverse engineering patterns (drawn from the literature) and forward engineering activities are performed applying test-driven development. Furthermore, in the context of our process, reverse and forward engineering activities are iteratively and incrementally carried
out. To assess the effectiveness of the process, it was experimented on a legacy system; the resulting software system confirms its effectiveness.

In order to describe our approach, the remainder of this paper is structured as follows. Section 2 and 3 presents background on the main concepts and techniques involved in the proposed reengineering process: reverse engineering patterns and test-driven development, respectively. Section 4 describes our iterative reengineering process and Section 5 presents a study case which briefly describes a legacy framework reengineering applying the proposed process. Section 6 concludes the paper with some remarks, limitations of the process, and future directions.

2. Reverse Engineering Patterns

Patterns describe a solution to recurring problems [Gamma et al. 1995]. They were at first adopted by the software community as a way of documenting solutions to problems that occur in many phases of the software systems life cycle. Usually, they are documented as a literary form which introduces the problem to the reader, describes the context within which it generally occurs, and presents a solution to the underlying problem.

Reengineering efforts deal with some typical problems and there is no tool or technique that is able to overcome all those problems. In addition, the process of reengineering is, like any other process, one in which many techniques have emerged, each of which entails many trade-offs. Reengineering patterns are well suited to describing and discussing these techniques; they help in diagnosing problems and identifying weaknesses that may hinder further development of the system, and they aid in finding more appropriate solutions to problems typically faced by developers [Stevens and Pooley 1998].

According to Arnold [1993], reverse engineering can be regarded as the initial phase in the process of software reengineering. Thus, reverse engineering patterns aim at building higher-level software models and acquiring more abstract information from the source code. Significant research has been done to record patterns that occur in reengineering and other contexts [Demeyer et al. 1999; 2002]. The reverse engineering patterns used in our process are described in the Section 4.1. The next Section outlines the agile practice called test-driven development.

3. Test-Driven Development

Test-Driven Development (TDD) is one of the core practices that have been introduced by the Extreme Programming discipline [Beck and Andres 2004]. TDD is also known as test-driven design and test-first design [Janzen and Saiedian 2005].

Essentially, applying TDD requires writing automated tests before producing functional code. Using TDD, the implementation of each new functionality starts with the developer writing a test case which specifies how the program should invoke that functionality and what its result should be. The recently implemented test fails, thus the developer implements just enough code to make the test pass. Finally, unless a prior test is not still passing, the developer reviews the code as it now stands, improving the code by means of a practice called refactoring.

Refactorings are behavior-preserving program modifications that improve a software system design and underlying source code [Fowler et al. 1999]. Refactoring as a
practice consists in restructuring software systems by applying a series of refactorings without altering their observable behavior. In the context of a TDD cycle, refactorings are carried out in order to make the introduction of new functionalities easier, during this process all of the previously written tests act as regression tests to make sure that the changes have not had any unexpected side effects.

Applying TDD, working software is available at every step and tests validate that each feature works as expected. The software system being developed using TDD is built and improved feature by feature, and the tests ensure that it is still working before the developer move on to next features. Thus, although its name implies that TDD is a testing technique, it is an analysis and design practice [Beck 2001]. It is considered an analysis technique because, during the creation of the tests, the developer selects what is going to be implemented, defining hence the functionality scope. Moreover, it is regarded a design technique because, while each test is implemented, the developer makes decisions related to the application programming interface (API) of the software system (e.g., classes and methods names, number of parameters, return type, and exceptions that are thrown).

The next Section describes our reengineering process and the fundamental role that reverse engineering patterns and TDD play in the underlying process.

4. Iterative Reengineering Process

Usually, legacy software systems are complex. Thus, to overcome this complexity, our reengineering process approaches parcels of legacy software systems. The system being reengineered is split into parcels coarse grained, such as layers and packages, or fine grained such as classes. After subjectively split the legacy system up into parcels, two types of activities are iteratively done for each parcel of this legacy system. The first type approaches the recovery of the existing design – reverse engineering – and the second one is related to implementing and improving – forward engineering and restructuration – the extracted design.

In order to accomplish reverse engineering activities, some of the patterns proposed by Demeyer et al. [2002] are applied. During the forward engineering activities, TDD is applied to implement the information related to the parcel which was reverse engineered. An overview of the process is shown in Figure 1. The patterns applied on this iterative process consider the available documentation as well as the source code of the legacy system being reengineered. The reverse engineering activities as well as the patterns used are described in the next Subsection.

4.1. Reverse Engineering Patterns Used in the Process

In this process, while performing reverse engineering activities, both the documentation and the source code are iteratively consulted in order to understand and to validate information obtained on a software system parcel. The patterns used in this case are: Read All the Code in One Hour, Skim the Documentation, and Speculate About Design [Demeyer et al. 1999; 2002]. These patterns have been chosen because they are well documented and have produced adequate results in our previous studies. Moreover, although these patterns are presented in the context of a major reengineering effort, according to their authors, they can also be applied when the reengineering is done in small iterations.
The main goal of Read All the Code in One Hour pattern is to assess the source code quality and complexity. This assessment is done by means of brief but intensive code review. There is an important difference between traditional code reviews and the ones performed during this process. The former is mainly meant to detect errors, while the latter is meant to get a first impression of the quality of the code and to recover information on how the functionality is implemented.

Figure 1. An overview of our iterative reengineering process.

This pattern originally suggests that all the source code should be read in an hour. However, in our process, only the source code related to the parcel being reengineered is examined. Therefore, there is a reduction in the amount of relationships among classes that must be comprehended in each iteration. According to Demeyer et al. [2002] a drawback of applying Read All the Code in One Hour is that the obtained information needs to be complemented with other more abstract representations. Thus, to complement those information another more abstract representations of the legacy system as, for instance, class and sequence diagrams must be consulted if available.

Skim the Documentation is applied in order to evaluate the relevance of the available documentation, and it is applied either before or after Read All the Code in One Hour. In the context of this iterative reengineering process, it is applied to select the sections of the documentation which contain more relevant information on the parcel being reengineered. This information is used to validate and to complement the low level information acquired by the use of Read all the Code in One Hour pattern. The Figure 2, by means of an activity diagram, represents the activities which have to be carried out during the reverse engineering of each parcel.

The mentioned patterns are repeatedly applied for each parcel of the legacy system being reengineered. All acquired information is verified and summarized since the documentation may not correspond to the implementation. This process, because of its inherently iterative nature, enables the developer to decide when the obtained information is enough to start carrying out reverse engineering activities.
Some parcels are harder to be comprehended since their documentation does not contain class diagrams or any other abstract representation of the functionality implemented by such parcels. The Subsection 4.2 describes how class diagrams which depict the design of the most complex parcels of the legacy system can be produced.

![Diagram](image)

**Figure 2.** Reverse engineering activities.

### 4.2. Constructing Class Diagrams of Complex Parcels

Class diagrams assist in understanding some parcels of a legacy system. Speculate About Design pattern can be applied in order to produce these class diagrams [Demeyer et al. 2002]. This pattern suggests the creation of a hypothetical class diagram based on suppositions about how the functionality of those parcels has been implemented. This hypothetical diagram, initially abstract and without any implementation details, is gradually refined and classes, methods and attributes may be added to it. Hence, the hypothetical diagram becomes closer to what is implemented by the parcel being considered. If the legacy system has already class diagrams, this pattern can also be applied, but those class diagrams have to be verified in order to check their consistency with the implementation. The pattern steps used to construct and refine the class diagram were adapted to become more flexible in the context of this iterative process. The modified steps which are being used are:

1. Developer creates a class diagram based on his/her understanding of the legacy system parcel being reengineered. Such class diagram serves as an initial hypothesis of what to expect in the source code.
2. The names of classes, methods and attributes are enumerated according to the likelihood that they would appear in the source code. After that, the developer searches the parcel code for those names.
3. Developer keeps in the class diagram classes, methods, and attributes that were found in the parcel code. In this step, it is also possible to rearrange the number assigned to the likelihood of each element.
4. Developer renames classes, methods, and attributes which have names that do not match with the names chosen in the parcel code.
5. The hypothetical class diagram is re-modeled when it does not correspond with the parcel code representation. For instance, a method may be turned into a class.
6. The hypothetical class diagram is extended when there are elements in the parcel code which do not appear in the class diagram.

The steps 2 to 6 are repeated until the class diagram adequately represents the functionality implemented by the parcel being reengineered. In this process, class diagrams do not need to be very detailed because during forward engineering implementation details are approached using TDD, hence the effort needed to generate such class diagrams is reduced. The Subsection 4.3 describes this process forward engineering step.

4.3. Forward Engineering Applying TDD and Refactoring

The information previously obtained is used to implement a parcel which is equivalent to the existing one. In this step of the reengineering process, the implementation of the resulting parcel may contain some improvements in relation to the legacy system equivalent parcel, due to an improvement in the functionality or some technological difference. Furthermore, some modifications may be necessary during the integration of the recently implemented parcel with the already reengineered parcels. If these modifications were not addressed, problems would be inadvertently introduced. In this process TDD is adopted in order to attenuate problems caused by the necessary modifications.

A list of test cases is created based on the information that was obtained during the reverse engineering activities. After the creation of the list of test cases, the steps of TDD cycle are performed and the parcel being addressed and a set of automated tests are implemented. These automated tests can be used as regression tests and therefore used to verify if problems were introduced in the source code due to modifications done during the integration of the parcels. If it is necessary, some refactorings can be done to assist in improving the parcel code which facilitates the parcel integration. The forward engineering activities are depicted in Figure 3.

Since parcels are implemented applying TDD, the creation of an upfront project may be unnecessary. This way, developers do not need to concern about how some features will be re-implemented when a new programming language is chosen.

In order to evaluate the effectiveness of the aforementioned process and the seamlessly integration of the techniques included in it, it was used to reengineer a large (more than 29 KLOC) legacy system. The following Section briefly describes how the reengineering process was applied.

5. Study Case: GREN Framework

This study case describes how the GREN framework [Braga and Masiero 2002] was reengineered from Smalltalk to Java – the new version of this framework is called GRENJ
GREN was built based on a pattern language called Business Resource Management (GRN) [Braga et al. 1999]; a pattern language that contains fifteen patterns which belong to the business resource management domain. The GREN framework architecture consists of three layers: persistence, business (model) and graphical user interface (GUI). Both the persistence and the business layers were reengineered; the business layer comprises the implementation of the GRN patterns. During the reengineering, each pattern implemented in this layer was considered as a parcel and iteratively approached.

Reverse engineering activities involved reading both the available documentation and the legacy framework source code. The major available documentation related to the framework is its “cookbook”, which conveys information on how the GRN patterns were implemented. GRN pattern language was also consulted, since it contains analysis-level class diagrams for each of its patterns. However, the main information source used was the GREN framework source code.

The reengineering of each parcel of the GREN framework was started by applying the Read All the Code in One Hour pattern to evaluate the quality of the parcel source code and to identify problems. After that, Skim the Documentation has been applied, thereby relevant documentation parts were obtained. Usually, these patterns were applied more than once. Read the Documentation pattern is applied until the developer obtains a textual description of the functionality implemented, and Skim the Documentation is applied to ensure that the documentation is synchronized with the current parcel implementation.

Two parcels were more complex to be comprehended only by consulting the source code and the available analysis-level class diagram, i.e., Rent the Resource and Maintain the Resource [Braga et al. 1999]. In order to overcome that complexity, the pattern Speculate About Design was applied. It is important to mention that no tools were used during the creation of the diagrams, thus, it took almost four hours to iteratively build each class diagram.

The mentioned patterns were applied until a coherent functionality description was obtained. However, in our process, the developer must decide when the information, gathered by means of those reverse engineering activities, is enough to start forward engineering activities.

For each parcel which has been reengineered a list of test cases was created based on the information obtained during reverse engineering activities. The textual functionality description of each parcel is transformed into a list of test cases. This list of test cases drove the implementation of the resulting system, i.e., GRENJ. During forward engineering activities, occasionally, additional tests were added to the list of test cases and some tests were split up into two or more test cases. It is important to emphasize that a parcel is considered reengineered only after implementing all test cases related to it as well as the respective functionality which fulfills these test cases.

6. Concluding Remarks

In this paper, we have presented an iterative reengineering process that consists of selecting a decomposition of the original system in terms of parcels and reengineering each of the individual parcels using TDD and reverse engineering patterns. Reverse engineering activities are carried out using patterns, thus there is reuse of knowledge. During forward
engineering, the automated tests created applying TDD provide the developer with feedback about analysis, design, and implementation decisions. Furthermore, these test cases can also be used as regression tests when future modifications are necessary. Nonetheless, a drawback of this practice is that the developer has to maintain both the functional code and the automated tests.

To present evidence of the efficiency of our approach, we have described a study case where a legacy system was reengineered from Smalltalk to Java. Although we have tested our approach to reengineer the aforementioned system, with more than 29 KLOC, we believe that our process does not scale well for larger applications, i.e., more than 35 KLOC. In order to apply this process in larger legacy systems, either reverse engineering or program comprehension tools must be used instead of reverse engineering patterns; since applying patterns to accomplish reverse engineering activities is time-consuming.

Several relevant points are not precisely addressed by our process: (i) the criteria to be used to define the parcels to be submitted to each iteration of the process and (ii) how the documentation, extracted by reverse engineering, can be used to define test cases in the forward engineering step. We are currently working on solutions for these limitations.

References
Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). Object-Oriented Reengineering Patterns. Morgan Kaufmann.