Applying Test-Driven Development and Reverse Engineering Patterns to Reengineer Legacy Software Systems

Vinícius H. S. Durelli[1], Simone S. Borges[2], and Rosângela A. D. Penteado[2]

[1] Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (ICMC-USP)
13560-970 – São Carlos – SP – Brasil
{durelli}@icmc.usp.br,
[2] Departamento de Computação
Universidade Federal de São Carlos (DC-UFSCar)
13565-905 – São Carlos – SP – Brasil
{simone_borges,rosangel}@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 approach that uses reverse engineering patterns and test-driven development to cope with issues involved in migrating from a legacy system to an equivalent software system implemented in more recent technologies. As a preliminary evaluation of the proposed approach, we contrasted it with an *ad-hoc* approach during the reengineering of a legacy system from Smalltalk to Java.

Key words: Reengineering; Test-Driven Development; Refactoring; Reengineering Patterns.

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 [1].

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 approach 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 approach, reverse and forward engineering activities are iteratively and incrementally carried out. To assess the effectiveness of the approach, it was experimented on a legacy system; the results indicates its effectiveness in terms of quality of the produced code.

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 approach: reverse engineering patterns and test-driven development, respectively. Section 4 describes our iterative reengineering approach and Section 5 presents a case study that describes a legacy framework reengineered applying the proposed approach. Section 6 concludes the paper with some remarks, limitations of the approach, and future directions.

2 Reverse Engineering Patterns

Patterns describe a solution to recurring problems. Usually, they are documented in 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.

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 [2,3]. The reverse engineering patterns used in our approach 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. 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 [1]. 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 [4]. 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 fundamental role that reverse engineering patterns and TDD play in the underlying approach is presented in the following section.

4 Iterative Reengineering Approach

Usually, legacy software systems are complex. Thus, to overcome this complexity, our reengineering approach deals with 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 [3] 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 approach is shown in Figure 1. The patterns applied on this iterative approach 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 Approach

In our approach, 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 [2,3]. 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 in the context of our approach. 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.

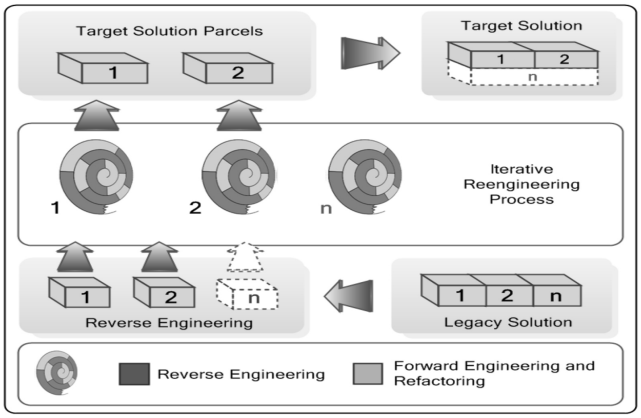

Fig. 1. An overview of our iterative reengineering approach.

This pattern originally suggests that all the source code should be read in an hour. However, in our approach, 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. 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 [3]. 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 approach, 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. 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. The proposed approach, 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.

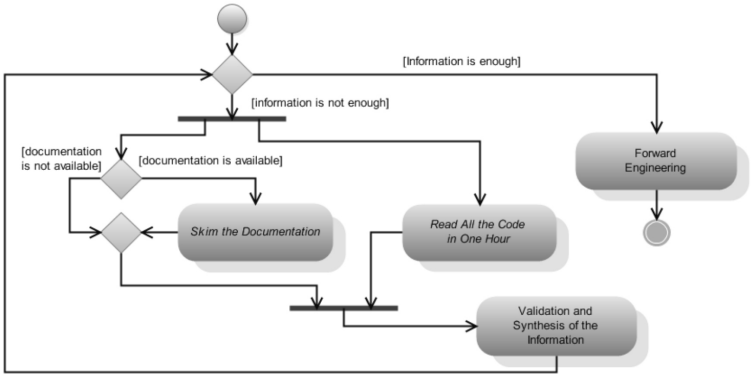

Fig. 2. Reverse engineering activities.

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 [3]. 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.

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 approach, 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 approach 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.

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

In order to evaluate the effectiveness of the aforementioned approach and the seamlessly integration of the techniques included in it, it was used to reengineer a legacy system which has more than 29 KLOC. The next section outlines how the reengineering has been conducted using our approach.

5 Case Study: GREN Framework

This case study describes how the GREN framework [5] was reengineered from Smalltalk to Java. GREN was built based upon a pattern language called Business Resource Management (GRN) [6]; a pattern language that contains fifteen patterns which belong to the business resource management domain. The framework architecture consists of three layers: persistence, business (model) and graphical user interface (GUI). The business layer comprises the implementation of the GRN patterns, and only code related to this layer was taken into consideration during the case study. In addition, it is worth to note that only one of the authors participated in the case study, and his knowledge of the languages involved can be classified as: advanced and intermediate, regarding Java and Smalltalk, respectively.

In order to provide evidence of the efficiency of our approach, the first three GRN patterns implemented in the framework GREN have been reengineered using an *ad-hoc* approach to perform reverse engineering activities; and the information that was obtained during these reverse engineering activities was implemented using a test-last approach. In contrast to it, other three patterns have been reengineered applying our proposed approach, thereby using reverse engineering patterns to support reverse engineering activities and forward reengineering activities applying TDD. The Table 1 shows information related to the pattern implementations which have been reengineered during the case study.

Table 1. Patterns implemented in the framework GREN that have been reengineered.

Name	Number of Classes	KLOC (Smalltalk)
Identify the Resources	7	0,652
Quantify the Resources	4	0,339
Rent the Resource	4	0,781
Trade the Resource	5	0,783
Quote the Trade	3	0,350
Check Resource Delivery	2	0,256

Throughout the case study we were interested in investigating whether our approach produces higher-quality code and reduces the amount of time spent on reengineering. Nevertheless, in the context of the proposed case study, quality is simply defined in terms of defect rates from the perspective of the software developer conducting the reengineering. Thus, the measures we have used to draw conclusions are: defect density (i.e., the number of defects per line of code) and the time spent to reengineer each pattern. The effect we expect our approach to have is formalized into hypotheses as follows.

Null Hypothesis, H_0: this hypothesis states that there is no real advantage in applying our approach, i.e., using TDD does not result in lower defect rates and the accuracy of the information retrieved by undertaking reverse engineering applying patterns is not cost-effective.

Alternative Hypothesis, H_1: according to this hypothesis, carrying out forward engineering activities using TDD improves the code quality, i.e., the resulting code has lower defect rates than code generated by a test-last approach. Moreover, the accuracy of the information drawn from the source code applying reverse engineering patterns is cost-effective.

After implementing all the functional code, the unit tests were created using the framework JUnit. The boundaries of each activity performed to reengineer the first three patterns are distinguishable (e.g., reverse engineering, forward engineering, testing, and debugging activities). Thus, we were able to stipulate the time that would be spent in each activity, these estimation as well as the real time spent reengineering each pattern are presented in Table 2. The results, gathered during the reengineering of these first patterns, are shown in Table 3.

Table 2. Estimation and real time in minutes spent reengineering the first three patterns

	Pattern#1		Pattern#2		Pattern#3	
Minutes	**Estimated**	**Spent**	**Estimated**	**Spent**	**Estimated**	**Spent**
Reverse Engineering	240	192	120	116	240	348
Forward Engineering	480	698	300	345	480	557
Testing	120	158	60	45	120	96
Debugging	120	47	60	143	120	98
Total	960	1095	540	649	960	1099

As it can be noticed in Table 2, more time than estimated had to be spent on the reengineering of all patterns. Moreover, tests have revealed that defect density of the first pattern code was considerable low (Table 3). However, as the first pattern's code had to be changed in order to be integrated with the other patterns' code, an increase in defect density of Pattern#2 and Pattern#3 appeared and, therefore, in the amount of time spent in debugging activities.

Table 3. Defect density and lines of Java code of each pattern (1 through 3).

Pattern#1		Pattern#2		Pattern#3	
Defect Density	**KLOC**	**Defect Density**	**KLOC**	**Defect Density**	**KLOC**
4,47	1,567	5,67	0,882	6,24	1,443

The other patterns have been reengineered using our approach. Thus, the time devoted to forward engineering and testing were joined up into one activity, namely TDD. The Table 4 shows the stipulated times for each activity, as well as the time spent in carrying them out. The results obtained after reengineering the remaining patterns are very satisfactory, Table 5. Although the time spent applying reverse engineering patterns was always more than previously stipulated, the information retrieved by using those reverse engineering patterns has been shown more accurate than the information obtained from an *ad-hoc* approach. In addition to it, forward engineering activities using TDD have shown to be well suited to deal with the issues involved in translating constructions of a language into similar constructions of another language. The defect density presented was very encouraging for almost all patterns. The number of defects found in the Pattern#6 was very close to the other patterns, namely ranging from 2 to 5. Nevertheless, its defect density was higher than the other pattern implementations since it has a reduced number of lines of code. The information gathered during the case study is summarized in the graphs of Figures 3 and 4, and by means of such information we were able to reject the null hypothesis.

Table 4. Estimation and real time in minutes spent reengineering the last three patterns

	Pattern#4		Pattern#5		Pattern#6	
Minutes	**Estimated**	**Spent**	**Estimated**	**Spent**	**Estimated**	**Spent**
Reverse Engineering	240	312	120	255	120	179
TDD (Forward Engineering)	545	509	360	499	300	405
Debugging	120	21	60	23	60	47
Total	905	842	540	777	480	631

Table 5. Defect density and lines of Java code of each pattern (4 through 6).

Pattern#4		Pattern#5		Pattern#6	
Defect Density	**KLOC**	**Defect Density**	**KLOC**	**Defect Density**	**KLOC**
2,68	1,498	3,90	0,771	9,13	0,438

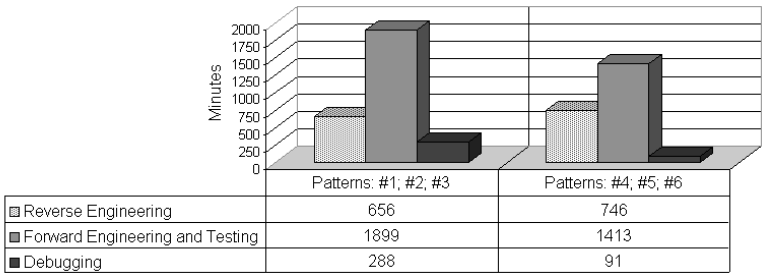

Fig. 3. Real time spent in each activity during the case study.

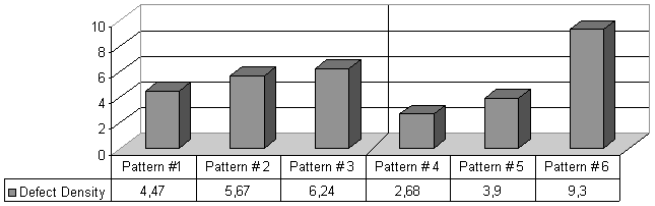

Fig. 4. Defect density rates presented by each pattern.

6 Concluding Remarks

In this paper, we have presented an iterative reengineering approach 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 case study 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 approach does not scale well for larger applications, i.e., more than 35 KLOC. The evaluation of the time necessary to perform reverse engineering using the proposed patterns pointed out that: in order to apply this approach 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.

According to the results obtained during the presented case study, TDD seems to be an effective practice to deal with the issues related to incrementally reengineering legacy systems. In the context of this case study, creating the tests before implementing the functional code has helped to address the translation of the up-front design, created by means of the information drawn from reverse engineering activities, into a suitable design which conforms to the features of the Java language. Moreover, the assessment of the results indicates that employing our approach implies in trading productivity for quality, since more time was generally spent using TDD and reverse engineering patterns than test-last and an *ad-hoc* approach.

Several relevant points are not precisely addressed by our approach: *(i)* the criteria to be used to define the parcels to be submitted to each iteration of the approach 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 and we aim at evaluating the results from different people or teams conducting reengineering efforts as the described in the case study, since the described one was conducted by just one developer.

References

1. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering **30**(2)
2. Demeyer, S., Ducasse, S., Nierstrasz, O.: A Pattern Language for Reverse Engineering. Proceedings of EuroPLoP 1999 (1999) 189–208
3. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann (2002)
4. Beck, K.: Aim, fire. IEEE Software (5)
5. Braga, R.T.V., Masiero, P.: A Process for Framework Construction Based on a Pattern Language. Computer Software and Applications Conference (2002)
6. Braga, R.T.V., Germano, F.R., Masiero, P.C.: A Pattern Language for Business Resource Management. Conference on Pattern Languages of Programs **6** (1999)