A SOFTWARE REFACTORING PROCESS AND THE SUPPORTED TOOLS

A. Stoyanova-Doycheva

Faculty of Mathematics and Informatics, Dept. of Computer Systems, University of Plovdiv, 236 Bulgaria Blvd., 4000 Plovdiv, Bulgaria

asya.stoyanova.doycheva@ecl.pu.acad.bg

Abstract: This paper describes the software refactoring process and the supported software tools which are under development as part of the project "Software Engineering: Education and Research Cooperation" (SEERC). All steps of this refactoring process are characterized, as well as the main functionality of the supported tools for the management of this process.

Key words: refactoring, reengineering, software process

1 Introduction

A large number of systems exist and, having been in use for years on end, have proved their utility. Nowadays, however, the requirements for these systems have changed with respect to the systems' functionality, technology or system platform. These changes may be urgent. In most cases the team that has to make the changes is not the same people that developed the system. For that reason, the new team has to understand the existing system in order to improve it.

Refactoring is the technique of changing a software system in such a way that it does not alter the external behavior of the code, and yet improves its internal structure. It is a disciplined way of cleaning up a code, which minimizes the chances of introducing bugs. In essence, when we refactor, we improve the design of the code after it has been written [1].

In today's practice the more and more frequent use of refactoring presents a good reason and motivation to define one complete refactoring process. The main task of this process is to make one systematic and simple approach to the improvement of legacy systems, which will ease management of this process and will give us the possibility of using some automatic tools.

2 Definition of the software refactoring process

Based on the definitions for the software development process [2, 3] and refactoring [1], the software refactoring process can be defined in the following manner:

The software refactoring process is an integral process of setting a task, planning, change of the internal structure and evaluation of an inherited software and hardware application, including the additional resources and methods used, as well as the nec-

essary staff to improve its internal structure, readability and comprehensiveness without altering its apparent behaviour.

3 General description of the process

The proposed refactoring process consists of the following steps:

- Analysis
- Investigation of the "smell" code
- Applying methods of refactoring
- Testing
- Measuring and assessment
- Statistics

Figure 1 presents these steps and the relationships between them in more detail. We start with analysis of the legacy system, and then continue with an investigation of the "smell" code and applying methods of refactoring. Testing is an activity which covers all these steps, and during each one of them we make different kinds of tests. The remaining steps of the process are measuring and assessment and statistics collection.

This paper describes each step of the software refactoring process.

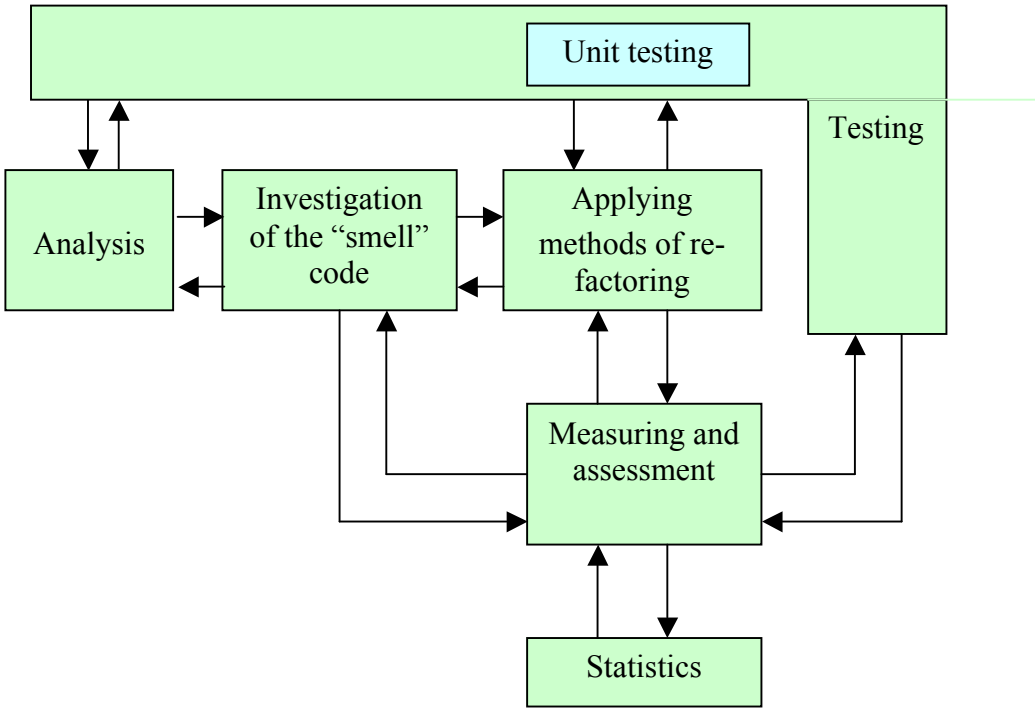

Figure 1: The software process of refactoring

3.1 Analysis

This step compares to the "Analysis" phase of the software development process. The participants in this phase are users of the legacy system, customers and developers.

Developers have to be introduced to the problem by the customers in order to define the tasks. They have to estimate if the problem can be solved with refactoring. The legacy system has to be in line with the criteria of refactoring, one of which is:

The legacy system works correctly [1]

For this purpose we have to implement "Global tests".

As an end product of this step we will create a document called "Requirements Specification", which will contain description of the problem, test cases for the global tests and a decision about our future work.

If the legacy system is not suitable for refactoring, the software development team can suggest some other solution appropriate for this problem: a new development from scratch, the purchase of software, which can be modified for this problem or development of the system using a component-based approach. If the legacy system is suitable for refactoring, we can continue with the next step of our software refactoring process.

3.2 Investigation of the "smell" code

After we have created the "Requirements Specification" and have decided to apply refactoring to the legacy system, we need to determine all places, where a method of refactoring [1, 5] has to be applied. In this step developers have to extract the current architecture of the system and define all the above-mentioned places.

In many cases they will be able to employ a method of refactoring in one place or another. They will have to decide which one to use, as they apply some metrics [6], or they can use their own experience.

Another problem which can appear is that "one refactoring leads to another refactoring". A developer has to decide if that would be good for the system.

In this step the target is a document including all the needed refactorings, the places, where they will be applied, and an account of the reasons for their use.

This step will help developers to take the right decisions, and it is the right decisions that guarantee the success of the project [2].

3.3 Applying methods of refactoring

Here are implemented all the described refactorings in the previous step.

Before a developer applies a method of refactoring, they have to take a suite of test cases for a current piece of the code (system). After applying the refactoring method they have to take the same suite of test cases. This will guarantee that the behavior of this current piece of code is correct. These tests are known as unit tests [1].

At this point M. Fowler suggests a self-testing code – The JUnit Testing Framework (CppUnit Testing Framework) – "Classes should contain their own tests" [1]

The applying of different methods of refactoring has to be conformable to a concrete programming language. We can use a list of Fowler's methods of refactorings [1, 5], or we can specify our methods for a concrete programming language. We have some experience in this activity with the legacy system XCTL [7]. We have defined some additional refactorings for the C++ code.

In this step developers have to create a document with the test cases for changed code pieces, and the current condition of the changed code in the system.

3.4 Testing

Inevitably, testing is a step of each process. It accompanies the whole software refactoring process (all steps) – analysis, applying methods of refactoring and measurement and assessment. Figure 2 describes this step.

Testing starts with global tests, and as a result we receive test cases and results from them "A". After that we continue with testing of the concrete pieces of code, where we create concrete test cases and receive concrete results for each piece of code "a". The next step is applying refactoring methods. As a result of this we have a new changed code. Over this changed code we apply test cases "a". Result "b" is compared with result "a". If the comparison is not correct, we return to the state where the code is changed to look for an error. If the comparison is correct, we continue with global or partial tests. These tests are necessary to make sure the system works correctly after each change of the code. We can use all test cases "A" from the global testing, or only these, which are suitable for the concrete functionality. The results from them "B" should be compared with results "A" from the global tests. If the comparison is not correct, we return to the state where we have made the last change to look for an error. If the comparison is correct, we can go to the finishing steps. First we have to make a full global test with test cases "A". The result "C" is compared with result "A". If the comparison is correct, we receive a refactored system with documentation. If the comparison is not correct, we can conclude that the partial test cases are not complete.

Consequently, we can divide the test step into the following sub steps:

1. Testing of the whole system before changes are made (refactoring) (Tests in the step Analysis). We create test cases.
2. Testing of the concrete piece of code that will be changed. We create test cases for this piece of the code. (Tests in the step Applying methods of refactoring)
3. Testing of the concrete piece of code after the changes (refactoring). For this purpose we use the same test cases as before the changes. (Tests in the step Applying methods of refactoring)

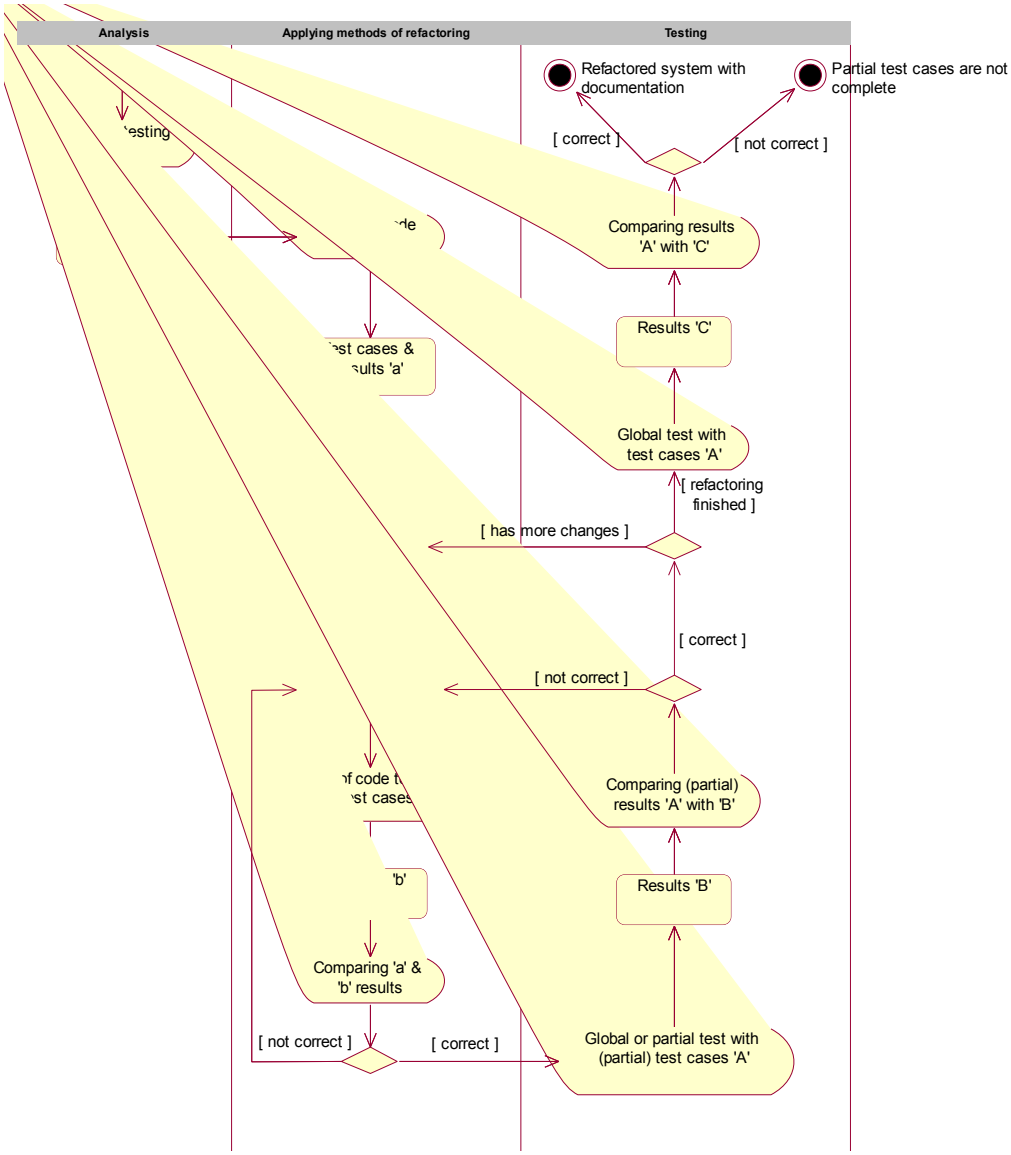

Figure 2: Activity diagram of the test step in the software refactoring process

4. Testing of the whole system after each refactoring if that is necessary. We can implement a partial system test, only on the part that we have changed. We use the test cases created in the Analysis.
5. After we complete the refactoring of the whole system, we have to apply test cases from the Analysis. This gives us a guarantee that the system behavior is not changed.

The substeps 2 and 3 form unit tests [1].

3.5 Measuring and assessment

Except for measures, determining the specific refactoring method to be used in a certain place in the source code [6], we can apply different metrics to assess the possible improvement of the design system. These metrics have to be chosen in conformity with the concrete refactoring, or with the concrete criterion of the quality, which we want to make better. This can be not only metrics which measure refactoring, but also ones that measure the design quality of the code.

We can explain why the design of the system after refactoring is better than the original design of the legacy system. Some of the reasons are:

- less code, easier to maintain – LOC metric;
- less complexity, which makes the code more understandable and easier to change – metrics for cyclomatic complexity;
- division of the system in clear parts – data representation, interface and control representations;
- etc.

For these measurements we can use some automated tools as JMetric [8], or others, which we develop based on the needed metrics.

This step can be used as a quality control. If the measures are bad we can find out why. This is of primary importance for us, because we can go back to previous steps in time – before we lose money or time.

3.6 Statistics

We can collect some statistics. It will be useful for the future work of the team. Thus they can avoid many mistakes in their future projects.

We can collect statistics for:

- used resources;
- used methods of refactoring;
- some information about the cases when one refactoring leads to another;
- some information about the cases when a developer has been able to apply one or another method of refactoring.

4 Management of the software refactoring process and supported tools

The most important part of each process is its management. The process of management comprises the generic activities and tasks, which may be employed by any party to manage their respective process. The manager is responsible for the product management, project management and task management of the applicable process.

The list of activities relevant to this process of management is marked as [4]:

- Initiation and scope definition;
- Planning;

- Execution and control;
- Review and evaluation;
- Closure.

For this purpose it will be good to use an automated tool. It will help the project manager to watch the progress of the process through its different steps. The automated tool should have the following main features:

- Tracing the artifacts and source code through the steps of the refactoring process;
- Distributing of the tasks and recourses according to characteristics of the project – time and money;
- Collecting of different project statistics, which can be used to determine some critical points;
- Redistributing the tasks and recourses according to the critical points.

With this tool the project manager will always know the current project status, where some problems may arise and when it is needed to rearrange the work. This tool can include other tools for the automated part of the process. That could be tools for automated product testing (ATOS), for measuring (JMetric [8]) and automatic refactoring (RefactorIT [9]).

5 Conclusions

Now work is being done on the first version of such a tool for management of the refactoring process, which was described above, and a tool for automatic refactoring. They are part of the project "Software Engineering: Education and Research Cooperation" (SEERC). The participants in the project are Humboldt University of Berlin, University of Novi Sad, University of Skopje, University of Belgrad, University of Plovdiv and others. All our experience of this refactoring process is based on the legacy system XCTL [7], which is the main topic in this project.

6 References

1. Fowler Martin, Refactoring. Improving the design of existing code, Addison Wesley, 2000
2. Pressman Roger, Software Engineering. A practitioner's Approach European Adaptation, McGraw-Hill Publishing Company, 2000
3. Jacobson Ivar, Grady Booch, James Rumbaugh, The unified software development process, Addison Wesley, 1999
4. ISO/IEC 12207, Information technology – Software life cycle processes, First edition 1995-08-01
5. List of refactorings in alphabetic order. Available at http://www.refactoring.com/catalog/

6. Simon Frank, Frank Steinbrückner, Claus Lewerentz, Metrics Based Refactoring, Software Systems Engineering Research Group, Technical University Cottbus, Germany
7. http://www.informatik.hu-berlin.de/Institut/struktur/softwaretechnikII/intkoop/se/XCTL-Man-Adj.htm - XCTL web site.
8. JMetric - http://www.jmetric.com
9. RefactorIT - http://www.refactorit.com/