Abstract
We will study the economical aspect of refactoring by investigating what effect refactoring has on the costs of adding new functionality to a system and by looking at the costs of the refactoring activity itself. We expect that the costs of adding functionalities will be lower after the system was refactored. Indeed it is generally accepted that refactoring has a positive influence on the maintainability of a system. However this is purely based on anecdotal evidence and still needs to be reinforced by a thorough scientific and systematic investigation. In our research we will be conducting a series of case studies on some “real-life” systems. On these cases we will determine where and when they were refactored and what influence these refactorings had on the development costs of the systems.

Keywords: refactoring, economics, refactoring detection, software evolution, mining software repositories

1. Introduction
The lion’s share of any software engineering project is the so called maintenance. Indeed the 20-80 rule states that 20% of the time is spent on the development of a system and 80% of the time on maintenance [20]. Moreover, the largest part of this maintenance (again 80%) is not spent on fixing bugs, but on other maintenance activities such as adding new functionalities [1]. However, as Lehman’s laws teach us, adding new functionalities will become more and more difficult as the system evolves, since a system will gradually become more and more complex (“Law of Increasing Complexity”) [14].

Nowadays refactoring is generally accepted as part of the solution for the long term maintainability of a software system (cf. the “Software Engineering Body of Knowledge” [1]). Refactoring is defined as “Transformations of the source code, without changing the observable behavior, to make the source code easier to understand and reuse” [19]. (i.e., refactoring lowers the complexity of the code) It is generally agreed upon that refactoring improves the maintainability of a software system, however this is based purely on anecdotal evidence and has yet to be empirically validated [16].

We will therefore study the refactoring economics in a software development process. We ask ourselves the following questions:

a. What are the costs of the individual refactoring steps?
b. What effect does refactoring have on the maintenance costs?
c. When will refactoring pay off its investment?

Small and Medium Enterprises (SME’s) will gain advantages from this research, as they will be better able to keep an eye on the costs and benefits of software maintenance which will give them a competitive advantage on much larger and more cumbersome companies. This research will also increase our knowledge on the costs and benefits of refactoring, which will give us a clearer image on when, where and how we need to apply refactorings. If we can show with this project that refactorings pay off their investment, then this can be used to convince software companies who do not refactor that refactoring is a useful and cost saving activity in the development of their software. Which
will allow them to spend more resources on more innovating applications. If on the other hand we find that refactoring does not pay off its investment, then management will be able to do a better cost benefit analysis by taking the loss caused by refactoring into account when deciding if they will refactor.

2. Goals

We have set ourselves two goals to examine the economical aspect of refactoring in a software development process:

- to empirically validate whether refactoring has a positive effect on the maintainability of a software system. (i.e., Will the manpower needed to add new functionalities decrease?)
- to examine if and when refactoring will pay off its investment.

The first goal can be achieved by showing that the cost to add new features after the system was refactored ($After$) is substantially lower than the cost of adding new features to the system before it was refactored ($Before$). In short, we will show that:

$$After \ll Before$$

For the second goal we will need to look at the cost of the refactoring activities themselves ($Refactoring$). We will know that refactoring pays back its investment if we can show that:

$$Refactoring + AddingFeatures \leq AddingFeatures'$$

with $Refactoring + AddingFeatures$ the cost to add a set of new features to a system after it was refactored and $AddingFeatures'$ the cost of adding that same set of features to the system without first refactoring the system.

To realize aforementioned goals we will need to lay some groundwork. We need to ascertain how we can determine where and when the software was refactored. In the past 10 years there has been a lot of research into techniques to do refactoring detection. However this was always done independently of one another. That is why we start our work with a comparative study of existing refactoring detection techniques, to determine which technique is most suitable for our purposes.

3. Related Work

Some work has already been done towards understanding the economics of refactoring. Leitch and Stroulia for instance have proposed a framework to predict the return on investment for a refactoring using cost-benefit analysis [15]. Sullivan et al. have shown that options thinking can be used to value software restructurings [23]. Bahsoon and Emmerich use the flexibility of a system as a way to valuate the payoff of refactorings [3].

There have been many studies on the effect refactoring has on a Software System. Du Bois for instance has studied the effect on the comprehensibility and understandability of a system [9] [8]. He also studied the effect refactoring have on the coupling and cohesion of a software system [7]. Other noteworthy work is that of Geppert et al., who studied the effect of refactoring on changeability [10], that of Moser et al., who investigated the effect on reusability [18], that of Kataoka et al., who did some research involving the effect of refactoring on the maintainability of a system [12] and that of Higo et al., who analyzed the effect of refactoring on some complexity metrics [11]. Some more general studies on the effect of refactoring on software quality include the ones by Alshayeb, Wilking et al, Stroggylos and Spinellis and Moser et al. [2] [27] [22] [17]

4. Proposed Solution

This research is split into two parts. The first one is the preparatory work and mainly consists of a comparative study of algorithms that search refactorings in the history of a system (i.e., refactoring detection techniques). The result of this study can then be used in the second phase of the project, which consists of a series of case studies on refactoring economics.
We will do our measurements on a few different cases (both open source and industrial projects). These cases need to be software systems where refactoring was actually applied.

We will primarily be looking at projects written in Java, since this is a very popular programming language with a very good support for automated refactoring (JRefractory, Eclipse ID, Netbeans IDE, ...). Later on we will also be looking at C and C++ programs to see if our conclusions can be made language independent.

Our experiments will be done in a few iterations, increasing the difficulty with each iteration. This way we will be able to identify possible problems in the used techniques and tools early in the project.

1. In the first iteration we will perform our experiments and measurements on small to medium sized projects (i.e. projects ranging from 10 to 100 KLOC), that are written in Java and have known refactoring periods.
2. In the second iteration we will look at systems of the same size as in the first iteration, but this time we will look at projects written in C, C++ or C# and with unknown as well as known refactoring periods.
3. In the final iteration we will look at larger systems (i.e., projects ranging from 100 to 1000 KLOC).

To insure the quality of our research we intend to follow the guidelines for empirical research set by Kitchenham et al. as well as the guidelines on conducting and reporting case studies set by Runeson and Höst [13] [21].

4.1. Refactoring Detection

Several refactoring detection techniques that have lead to good results have already been developed. Weißgerber and Diehl for instance presented a technique that uses signature-based analysis to identify possible refactorings and clone detection to rank the likelihood of it being a refactoring [26]. Danny Dig et al. used a combination of syntactical and semantical analysis to identify refactorings [4][5][24]. Demeyer et al. have developed a set of heuristics based on a combination of change metrics to identify refactorings. Van Rysselbergh and Demeyer have developed a way that uses clone detection [25]. And Xing and Stroulia developed the UMLDiff algorithm, which is capable of detecting some basic structural changes in a system [28].

We will compare these refactoring detection techniques based on a few criteria, that we have subdivided into primary and secondary criteria.

Primary Criteria

- **Precision** is a criterion that measures to which extent the techniques are correct, in other words: “Are the detected refactorings truly refactorings?”
- **Recall** is a criterion that measures to which extent the techniques are complete, in other words: “How many of the actual refactorings were found?”
- **Performance** is a criterion that measures the speed with which the techniques detect refactorings.

Secondary Criteria

- **Scalability** is a criterion that measures to which extent the technique is applicable on larger projects. Scalability and performance are important criteria to us, since we also want to detect refactorings on medium sized to large projects and projects with a long history.
- **Scope**: This means that we will determine whether the algorithm is language independent or specific and we will also look at which refactorings it is capable of finding.

4.2. Cost of a new functionality

With this experiment we will empirically study the effect refactoring has on the costs of adding functionality to a system. To do this we shall determine what the costs are of adding a new feature before and after a refactoring period. The experiment will be performed on a few cases and consists of a few steps:

1. Using some refactoring detection technique, we can analyze the case’s version control system to find a version that has clearly just been preceded by a refactoring period. (Call this version $V_R$)
2. We analyze the versions of the system before $V_R$ to find features that were added before $V_R$, but after a possible previous refactoring period. For each of these features we determine the cost. This way we can determine the average cost of a new feature per version.

3. We do the same for the versions after $V_R$, but before a possible subsequent refactoring period.

4. Then we can project how the average cost of adding a new feature changes over time (i.e., how this average cost changes with each version of the system).

In this last step (step 4) we expect to see results like the ones presented in the sketch in figure 1. We expect to see the average cost of new features increase before the refactoring period. The average cost of new features in the versions following the refactoring period will be significantly lower than in the versions preceding the refactoring period. However the cost of new features will again increase over time.

![Figure 1: Cost of implementing a new feature should increase with time. After a refactoring period the cost should decrease.](image)

4.3. PayOff of Refactoring

In this part of the research we will address the following research questions:

- Will refactoring eventually pay off its investment?
- When does this payoff become apparent? e.g., How many new features need to be added before there is a profit?

In this experiment we will look at a series of features that were implemented after the system was refactored and we will compare their cost to what they would have cost if the system had not been refactored. Since the latter cost can not be measured we will need to use a cost estimation model. The cost of the features as is, can be measured from the implementation, however in order to have comparable data it is best to also use the cost estimation model to determine this cost. The cost estimation model that we use, will be the one(s) that we concluded from the preparatory work to be the best for our purposes.

To answer our first research question we need to measure (or estimate) the cost of the refactoring itself (call this $S_{Refactoring}$). We also need to estimate what the costs are of adding a series of new features after the refactoring (call this $S_{AddingFeatures}$). And finally we need to estimate what the costs are of adding this same series of features to the system before the refactoring (call this $S'_{AddingFeatures}$).

The refactoring will then produce a profit only if $S_{Refactoring} + S_{AddingFeatures} \leq S'_{AddingFeatures}$.

If we find that the refactoring produces a profit, we can project how $S_{Refactoring} + S_{AddingFeatures}$ changes by increasing $S_{AddingFeatures}$ with one feature at a time. This way we can find at which point in time (after how many features) the refactoring starts to return on its investment.
5. Summary

In short we propose to investigate the economical aspect of refactorings by a series of case studies on some real-life systems. We will study the costs of refactoring and the influence refactoring has on the costs of adding functionality to an existing system. We expect to see that the cost of adding functionality is lower after the system has been refactored.

After that we can also determine whether the refactoring activity is a profitable one.


