Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?

Bart Du Bois and Serge Demeyer
Lab On ReEngineering
Universiteit Antwerpen
{bart.dubois, serge.demeyer}@ua.ac.be

Jan Verelst
Dept. of Management Information Systems
Universiteit Antwerpen
jan.verelst@ua.ac.be

Abstract

Program comprehension is a fundamental requirement for all but the most trivial maintenance activities. Previous research has demonstrated key principles for improving comprehension. Among others, these consist of the introduction of beacons as indexes into knowledge, and the chunking of low-level structures into higher-level abstractions. These principles are naturally reflected in the reverse engineering pattern Refactor to Understand, which uses incremental renaming and extracting of program elements as the means to decipher cryptic code. In this paper, we discuss a controlled experiment to explore differences in program comprehension between the application of Refactor to Understand and the traditional Read to Understand pattern. Our results support added value of Refactor to Understand regarding specific aspects of program comprehension and specific types of source code. These findings illustrate the need for further experiments to provide clear guidelines on the application of refactorings for improving program comprehension.

1 Introduction

Program comprehension can be defined as the recognition of the overall program function, an understanding of intermediate-level processes including program organization, and comprehension of the purpose of each statement [Sheiderman, 1976]. Next to comprehension of the maintenance requirement itself, program comprehension is one of the first objectives in any maintenance activity, and therefore has a major impact on the effort to resolve a maintenance request. Some early works have identified that up to half of the maintenance time is spent on understanding the code [Parikh and Zvegintzov, 1983, Corbi, 1989]. Being an important and difficult activity, program comprehension should be supported by validated techniques.

Empirical research has demonstrated significant improvements in comprehensibility by changing the source code representation concerning indentation [Miara et al., 1983], typographic style [Kernighan and Plauger, 1982, Oman and Cook, 1990], source folding [Cockburn and Smith, 2003], variable names [Mynatt, 1990, Gellenbeck and Cook, 1991] etc. These works have therefore shown that code adaptation can be beneficial for future comprehension activities. However, can an adaptation process itself – physically modifying source code – improve program comprehension?

In this work, we test a comprehension technique that focusses on such physical modifications to steer understanding, as we are convinced that traditional reading activities are less efficient. Therefore, we analyze answers of final year computer science students to open program comprehension questions in order to compare the reverse engineering pattern Refactor to Understand to the traditional Read to Understand with respect to response accuracy and time. The former pattern is suggested as a way to expose the design of cryptic code by testing hypotheses concerning the code [Demeyer et al., 2002].

We argue that higher response accuracy and lower response times are good indicators for improved comprehension. Therefore, we set up a controlled experiment in which we test the following null hypotheses:

- $H_{0,accuracy}$ – Refactor to Understand does not increase the accuracy of answers to program comprehension questions.
- $H_{0,time}$ – Refactor to Understand does not shorten the time of answers to program comprehension questions.

Rejecting $H_{0,accuracy}$ would lead to the acceptance of the alternative hypothesis, which states that Refactor to Understand increases comprehension accuracy. Furthermore, the rejection of both null hypotheses would demonstrate that the improved comprehension would be available as recall from memory, and therefore, that the achieved program

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

comprehension would be readily available for comprehension related tasks.

The exploratory nature of this experiment does not allow us to focus on statistical significance, and therefore this work is bound to remain inconclusive. However, as a first test of the proposed hypotheses, this work provides a first indication of the practical value of the process of refactoring for the purpose of understanding software system.

This paper is organized as follows. Section 2 relates program comprehension to Refactor to Understand. In section 3, we describe the experimental set-up. An analysis of the gathered data is presented in section 4, and discussed in section 5. The threats to validity are discussed in section 6. Finally, we conclude in section 7 and elaborate on future work in section 8.

2 Supporting Comprehension Principles

2.1 Program Comprehension

Research on program comprehension has led to a number of comprehension models, which can be categorized into bottom up, top down or integrated (hybrid models). A bottom up model develops mental representations, top down models decompose code into typical elements, and an integrated model combines both top-down and bottom-up understanding. While it has been shown that programmers switch between all models , top-down understanding is typically applied for familiar code [von Mayrhauser and Vans, 1994]. In this work, we focus on unfamiliar code, and therefore on the aspects of bottom-up understanding, as investigated by Pennington [Pennington, 1987].

Pennington's mental model originates from the theory of text comprehension. This model poses that bottom-up comprehension develops two distinct mental representations: a program model and a situation (or domain) model. The representation of the program model is a control-flow program abstraction, and classifies two categories of knowledge about the program: control flow and operations. The situation model (aka domain model) representation creates a dataflow/functional abstraction, and therefore classifies data flow and functional knowledge. Next to these four categories, a fifth category – state – has also been identified, and is shared between the two representations. Pennington's mental model therefore categorizes comprehension as knowledge on Control Flow, Operations, Data Flow, Function and State. Typical comprehension questions for these categories of knowledge are:

- **Function.** What is the overall functionality? Such knowledge is important for localization in maintenance activities.
- **Control Flow.** What is the sequence of execution? Control flow knowledge allows to distinguish and recognize different scenarios.
- **Operations.** What does the code compute? This knowledge incorporates the semantics of statements.
- **Data Flow.** Where does a data object get updated? Data flow knowledge allows to recognize dependencies between data and behavior.
- **State.** What is the content of a data object? This knowledge requires both operations knowledge (part of the program model) and dataflow knowledge (part of the situation model).

Therefore, to validate the usefulness of a particular cognition technique, one can evaluate its contribution to these five aspects of comprehension. We select Pennington's mental model for its practical applicability and its relation to common software engineering knowledge.

2.2 Refactor to Understand

Refactor to Understand is a typical reverse engineering pattern in that it does not intend to improve the code base itself, but the maintainers understanding. Consequently, less emphasis is put on regression testing, and more on the composition and verification of hypotheses concerning the code.

The iterative process of Refactor to Understand, as applied in this experiment, can be described as follows:

1. Read the code
2. Evaluate names on their correspondence to the true semantics of the variable/method/class. Rename if necessary.
3. Evaluate groups of statements on their semantical coherence. Extract in a method if necessary. Name the method in correspondence to its true semantics.

These actions are used to test hypotheses concerning the code. The formulation of hypotheses, checking whether they are true or false, and the consequential revision is known as code cognition [von Mayrhauser and Vans, 1995]. Therefore, we consider Refactor to Understand foremost a cognition technique.

Key comprehension principles stimulated in these actions are the introduction of a) beacons; and b) chunks:

- **Beacons** are recognition points that index into knowledge [von Mayrhauser and Vans, 1995]. Mostly, beacons are names that are semantically associated with a specific programming concept. Current literature on beacons has focused on their recognition

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

[Wiedenbeck and Scholtz, 1989, Crosby et al., 2002], and therefore, uses beacons for top-down comprehension. In a refactoring context, we focus on their introduction in the code as to stimulate their recognition by future code readers, and therefore, in our context, we use the introduction of beacons to stimulate bottom-up comprehension during the actual refactoring.

- **Chunking** is the composition of lower level structures into higher level abstraction structures. Such a composition strategy allows the reader to trade off when to zoom in, based on the names of the chunks that already serve as beacons. For the person performing the actual composition, chunking is a means to introduce abstraction, for example by recognizing that a specific sequence of mathematical operations signifies the calculation of the tax amount for a specific product price and tax rate. Therefore, extracting statements into methods supports bottom-up comprehension.

Summarizing, there is a clear link between program comprehension and Refactor to Understand, in that the latter provides a means to introduce principles demonstrated to be useful for constructing mental models of the source code.

2.3 Testing Refactor to Understand

A comprehension technique should be most supportive for code fragments which are hard to comprehend. We assume that one of the factors which influence comprehensibility is coupling, as it is an indicator for the extent to which the functionality is distributed. Such distribution can lead to localization - as in encapsulation - but also, to delocalization. It has been shown that code readers have difficulty in comprehending such delocalized implementations [Letovsky and Soloway, 1986]. Thus, it is important to test a program comprehension technique for both localized and delocalized code fragments. We will therefore test Refactor to Understand under representative degrees of coupling.

3 Experimental Set-Up

In April 2004, 12 final year Computer Science students participated in a controlled experiment in the context of a course on Object-Oriented Reengineering. This course is optional in the final (fourth) year of the Master level Computer Science curriculum at the University of Antwerp. As final year students, they had substantial experience with Object-Oriented development. At the time the experiment took place, students were familiar with the Object-Oriented Reengineering Patterns defined in [Demeyer et al., 2002] during one introductory and three practical sessions.

Students were motivated to participate in the experiment for two reasons: a) it would introduce them to the software system targeted in the associated course project; and b) it would introduce them to issues specific to understanding fragments of a large unknown software system.

One week prior to the experiment, the students participated in the so called Refactoring Lab Session. During this session, a real-life scenario is simulated in which a development team has implemented initial requirements for a system, received requests for extra functionality, and decides to call in refactoring expertise to prepare the task of adding the new functionality. A reusable course package has been composed, which can be downloaded freely [Demeyer and Du Bois, 2004]. This session familiarized all students with the environment used, and the techniques applied in the experiment.

As an experimental design, we used a crossed factorial design with two factors (depicted in Table 1). As a between-subjects factor, the program comprehension technique has two levels, being Refactor to Understand and Read to Understand. As a within-subjects factor, the degree of coupling of the program under study has three levels, being low, moderate and high. Thus, each of the six subjects in any group was exposed to all degrees of coupling, and applied the same comprehension technique during each run.

Table 1. Design of the experiment.

	Comprehension Technique	
Degree of Coupling	Refactor to Understand	Read to Understand
None	Group 1 - run 1	Group 2 - run 1
Moderate	Group 1 - run 2	Group 2 - run 2
High	Group 1 - run 3	Group 2 - run 3

The 12 students were divided into two groups: a treatment and a control group. Students in the treatment group applied Refactor to Understand. In the control group, students applied Read to Understand - the traditional comprehension technique.

3.1 Subjects

As a countermeasure for controlling extraneous factors in the performance of the two groups, we chose to form blocks of students with similar estimated performance. As the estimation of performance is a research on its own, we relied on conclusions from a previous observation experiment.

As a procedure for matching, we formed blocks of subjects exhibiting most similar grades on the observed course.

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

Each block consisted of two students, which were randomly distributed over the two groups. As a result, for each subject in the treatment group (Refactor to Understand), a matching subject is present in the control group (Read to Understand).

Therefore, we feel assured that a) the overall performance of the groups can be expected to be equal; and b) each student matches up to its counterpart in the other group.

3.2 Experimental Procedure

The experiment was performed over three runs. Each run consisted of a fixed time slot during which the subjects were allowed to use the assigned comprehension technique, and a fixed time slot during which they were asked to answer questions concerning the code. These open questions were variants of a generic question for each category of knowledge, which were mentioned in the section 2.1.

The concrete questions for each experimental run, together with the rest of the experimental material, can be found on the authors website (http://www.lore.ua.ac.be/Research/Artefacts/ refactor2Understand/).

At the beginning of the experiment, an introduction to the experimental procedure was given. At the end of the experiment, a post-experiment questionnaire was filled in by all subjects. Guidelines were provided for the comprehension technique to be applied by subjects in the treatment and control group. These guidelines contained as a common part the explanations of the five categories of comprehension being target of inquiry at the end of each task. For the treatment group, guidelines for the application of the Refactor to Understand pattern were specified:

1. Spend your time wisely, no beautification.
2. Iteratively rename attributes/methods/classes to introduce meaningful names).
3. Extract methods or local variables to make the code structure reflect what the system is actually doing.

These guidelines were adapted from [Demeyer et al., 2002], and are practical directions for stimulating comprehension principles. More specifically, they can be translated in program comprehension terminology as a) introduce beacons; and b) introduce chunks. Both renaming and extracting methods are automated as refactorings and required only the typing of the associated key-binding and the new name of the program entity.

Task sheets were provided at the beginning of each run. The sheets indicate two items: a) the duration of the time slots for comprehension and for answering the open questions; and b) the boundaries of the piece of code to be studied.

The software system under study was the Eclipse Platform (version 2.1.3), which is mostly known as an Integrated Development Environment for Java. More specific, the selected pieces of code originated from the `org.eclipse.jdt.internal` package, which contains a total of 721 classes over more than 140 thousands lines of code.

3.3 Experimental Variables

Our two independent variables are a) the programming comprehension technique applied (either Read to Understand or Refactor to Understand); and b) the degree of coupling of the code part to be comprehended (no, moderate or high coupling).

As dependent variables, we measure the response accuracy and response time of answers on comprehension questions. For each run, one open question for each comprehension category was asked. These questions allowed us to test the recall of subjects without the danger of providing partial answers, which would exist with closed questions.

3.3.1 Measurement of the dependent variables

As we are interested in finding clear, objective differences between accuracy, we choose to rate accuracy in four distinct categories:

- **Wrong answer**: Assignment not understood, no answer given or plain wrong answer.
- **Right idea**: The response contains the essence of a good solution, but is not worked out properly.
- **Almost correct**: The response is lacking minor details, or these minor details are wrong.
- **Correct**: The response is flawless.

It is obvious to see that accuracy rated this way is at least ordinal as the ratings can be ranked. However, there is also a difference in distance among the answers. Therefore, we rated a wrong answer as 0, an answer which contains the right idea with 5, answer which is almost correct with 8, and a correct answer as a 10. Such a rating is commonly applied in education. We did not distinguish between responses in the same accuracy category, as such differences would be quite subjective.

Therefore, the accuracy measurement is interval scale, which allows us to calculate means and standard deviations on these measurements, as well as apply common statistical techniques. Moreover, we also performed an analysis on the ordinal data, which distinguished the same significant differences.

Subjects themselves wrote timestamps on the task sheets when starting to read a question, and when finishing it.

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

Therefore, the time measurement incorporates time for reading the question, time for solving the question mentally and time for writing down the question. As the questions and associated answers were quite short (answers took less than 3 lines), the number of seconds it took to provide the answer (absolute scale) is a good indicator for the time to solve the question.

3.4 Experimental Runs

To study the effect of Refactoring to Understand under different degrees of coupling, source code fragments were chosen to vary from no coupling in the first run, over moderate coupling in the second run, to high coupling in the last run. We set out to select code fragments which varied regarding coupling, yet were about similar w.r.t. other properties. As illustrated in Table 2, perfect similarity is unfeasible in practice. The table depicts the cumulated Lines of Code (LOC), Number of Methods (NOM), Number of Local Variables (NLV) and Cyclomatic Complexity (CC) values, which also vary but not to the extent as the coupling properties Message Passing Coupling (MPC) and Coupling Between Object Classes (CBO).

Table 2. Properties of each of the code fragments denoted as Not Coupled, Moderately Coupled and High Coupled.

	NC	MC	HC
LOC	109	102	76
NOM	1	1	4
NLV	18	24	38
CC	27	5	14
MPC	0	42	61
CBO	0	10	30

As can be read from the MPC and CBO values, the localized code fragment (NC) was fully self-contained, and did not depend on other classes. The moderately coupled code fragment (MC) has about the same length, yet has a simpler control flow and collaborates with some other classes. Finally, the delocalized code fragment (HC) implements its functionality by collaboration with many different classes.

3.5 Experimental Nature

This experiment was set up for exploration. Therefore, our focus is more on observing interesting phenomena than on statistical significance. An experiment with a focus on statistical significance would require at least sixty subjects. The resources for such experiments are scarce, and should be spent carefully. Therefore, it is advisable to first perform exploratory experiments to refine hypotheses in order to optimize resource usage.

Consequently, we set the significance criterion α to 0.1, which is common for exploratory experiments. In other words, we can only accept group differences when the probability of a difference as extreme as the one observed is less than 10% ($p < 0.10$).

4 Data Analysis

Data was collected from task sheets, source code, cvs log data and a post-experiment questionnaire.

On the task sheets, the subjects noted down time stamps before and after each question. Together with their answer on the open question, this allowed us to assess the response accuracy and calculate the response time regarding all five comprehension categories.

In the post-experiment questionnaire, closed questions were asked about their performance, motivation, adherence to the guidelines, perceived difficulty of the tasks and reasons for not being able to answer all the questions. This questionnaire was provided at the end of the experiment.

4.1 Applications of Refactorings

Table 3 displays the number of refactorings applied by each of the subjects in the refactoring group. This data is relevant as it allows to characterize the specific treatment as applied by the members of the treatment group.

For the tasks on the code fragments with no and moderate coupling, one subject did not apply any refactorings. However, searching for refactoring opportunities still steered their comprehension, and therefore, we still categorize them in the treatment group.

Over the tasks, there was a difference in the applications of refactorings. The relative frequency of applications of the refactorings rename were respectively 17% for the code with no coupling; 32% for the moderately coupled code; and 63% for the highly coupled code.

The table indicates that the Refactor to Understand pattern was indeed applied. However, it was not applied extensively, as can be observed from the limited number of refactorings. Therefore, it is reasonable to assume that more extensive application of this reverse engineering pattern will lead to more extreme group differences.

4.2 Research hypotheses

Our null hypotheses are that Refactor to Understand does not increase the accuracy nor the response time of answers to questions regarding program comprehension. As we have reasons to expect that the effect of a program comprehension technique is dependent on the degree of coupling of

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

Table 3. Number of refactorings applied by each subjects in the refactoring group.

coupling	subj	#Rename	#Extract Method
NC	s1	0	2
	s2	1	1
	s3	0	0
	s4	0	1
	s5	1	2
	s6	0	2
	avg	*0.33*	*1.33*
MC	s1	0	0
	s2	5	1
	s3	0	4
	s4	0	2
	s5	1	3
	s6	1	1
	avg	*1.17*	*1.83*
HC	s1	4	1
	s2	1	0
	s3	0	2
	s4	2	0
	s5	1	3
	s6	3	0
	avg	*1.83*	*1*

the code fragment to be understood, we specialize these null hypotheses for each degree of coupling:

- $H_{0,accuracy,X,Y}$ – Refactor to Understand does not increase the accuracy of answers to questions regarding program comprehension category X for code fragments which exhibit a degree of coupling Y.
- $H_{0,time,X,Y}$ – Refactor to Understand does not reduce the response time of answers to to questions regarding program comprehension category X for code fragments which exhibit a degree of coupling Y.

As we are testing whether Refactor to Understand improves comprehension, our alternative hypothesis are one-sided. Our statistical tests therefore are one-tailed tests, which verify whether the responses of the Refactor to Understand group exhibit higher accuracy or shorter response times. Consequential, the resulting p-value can only be used to indicate the probability of a one-sided difference as extreme as the one observed under the null hypothesis.

It is common to apply Analysis of Variance (ANOVA) to test for main effects for either factor comprehension technique or degree of coupling, or for an interaction effect between both. However, the nature of this exploratory experiment is not suitable for such analysis techniques. Therefore, we rely on descriptive statistics.

4.3 $H_{0,accuracy}$ – Differences in response accuracy

Table 4 depicts the average response accuracy for both groups, in the format Refactor to Understand group – Read to Understand group. The p-value for the associated one-sided paired t-test is also provided. To ease the discussion, we will refer to the accuracy of the responses regarding a particular comprehension category X as the X accuracy (e.g. function accuracy).

We will discuss the observed differences for each of the levels of coupling separately.

- **No coupling.** The Refactor to Understand group demonstrated higher average function and operations accuracy. However, control and data flow accuracy, as well as state accuracy were lower.
- **Moderate coupling.** On average, the Refactor to Understand group responses were more accurate for the function, control flow, operations and data flow questions. State accuracy was approximately equal for both groups.
- **High coupling.** Significantly higher function and control flow accuracy was observed for the Refactor to Understand group. Data flow and state accuracy were also higher. Conversely, the Refactor to Understand group demonstrated lower operations accuracy.

However, we must remark that there was data loss. Only 3 matched observations were available for data flow comprehension of the moderately coupled code fragment, 3 for the state comprehension of the fragments with no and moderate coupling, and 5 for state comprehension of the highly coupled code.

Table 4. Average response accuracy.

	NC	MC	HC
Function	7.7 – 6.3	6.7 – 4.7	8.5 – 4.7
	p= .2598	p= .2314	p= .0560
Control Flow	4.7 – 5.5	6.2 – 5.2	6.8 – 4.0
	p= .8184	p= .6686	p= .0878
Operations	6.5 – 4.7	6.0 – 2.5	6.3 – 6.8
	p= .2551	p= .1261	p= .5778
Data Flow	4.7 – 5.8	3.3 – 1.7	5.5 – 4.7
	p= .8898	p= .6291	p= .2484
State	5.0 – 6.3	6.7 – 6.7	9.0 – 6.4
	p= .7887	p= .6291	p= .1418

Based on these observations, we can reject $H_{0,accuracy,Function,HC}$ and $H_{0,accuracy,ControlFlow,HC}$.

We can associate an effect size γ with significant group differences by standardizing the difference between group

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

means by the standard deviation. A calculation of this effect size leads to γ = 1.20 for function accuracy, and γ = 0.69 for control flow accuracy for the third experimental run. Some general guidelines apply on effect sizes, being that starting from 0.2, effect sizes are small, starting from 0.5 they are considered medium, and effect sizes larger than 0.8 are considered large [Cohen, 1988]. However, these guidelines should only be used if no other information for categorizing effect sizes is available, and such is the case in this domain.

Summarizing, improved function and control flow accuracy was observed in the responses of the Refactor to Understand group in two out of three experimental runs. This group difference was most extreme in the code fragment with high coupling.

4.4 $H_{0,time}$ – Differences in response time

Group differences regarding the time for answering program comprehension questions are reported in Table 5.

- **No coupling.** Response times for the Refactor to Understand group were higher for function, control flow, operations and data flow comprehension. Conversely, the state question was answered faster by the Refactor to Understand group.

- **Moderate coupling.** Shorter response times were observed for the questions on all but the operations comprehension category. Answers on the question for the latter category were equal in time.

- **High coupling.** Significant shorter response times were demonstrated for the function comprehension category. The questions on operations and data flow comprehension were also responded in less time. Responses on control flow and state, however, took more time for the Refactor to Understand group.

Table 5. Average response time.

	NC	MC	HC
Function	172 – 168	182 – 190	109 – 201
	p= .5383	p= .4587	p= .0908
Control Flow	402 – 362	366 – 409	322 – 321
	p= .7042	p= .3066	p= .5047
Operations	236 – 205	113 – 113	128 – 165
	p= .6655	p= .5036	p= .2171
Data Flow	132 – 111	81 – 101	123 – 151
	p= .8779	p= .2443	p= .1118
State	165 – 216	76 – 79	200 – 158
	p= .3031	p= .4459	p= .9742

We remark that some observations on data flow and state comprehension were missing, and refer to the discussion of accuracy results for more details on this loss.

Based on these results, we can reject $H_{0,time,Function,HC}$. The associated effect size γ equals -0.98. Therefore, for the question on the function of the highly coupled code, we observed a large effect of the comprehension technique.

Summarizing, we observed shorter response times for questions on the functionality of the highly coupled code fragment. Comprehension questions on the fragment with no coupling were answered slower for all but state comprehension. For the moderately coupled code fragment, slightly shorter response times were noted.

4.5 Post-experiment questionnaire

After the experiment, a questionnaire was handed out, which was an adaptation from [Briand et al., 2001]. Questions were asked about motivation, adherence to guidelines, understanding of requirements, approach to answering the questions, accuracy estimation, reasons for not completing, code quality estimate and difficulty of questions on the three tasks.

We observed no problems with motivation nor clarity of questions. On a scale from 1 (barely comprehensible) to 10 (easily comprehensible), the quality of the code was rated an average of 7.1 by the Refactor to Understand group and 6.3 for the Read to Understand group.

There was a notable regarding the difficulty of the tasks. The questions on the localized code fragment were considered more difficult than those on the moderately coupled fragment (p=0.0609), which on their turn were found more difficult than those on the delocalized code fragment (p=0.1412). This is rather peculiar as accuracy was highest for HC.

5 Discussion of Results

To discuss the results of this exploratory experiment, we describe our findings for the two factors under study: a) the comprehension techniques Read to Understand and Refactor to Understand; and b) the degree of coupling of the code fragments to be understood.

5.1 Comprehension Technique as a Factor

In this experiment, we verified whether the reverse engineering pattern Refactor to Understand can help improve program comprehension. In two out of three experimental runs, the Refactor to Understand group demonstrated higher

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

accuracy for function, control flow, data flow and state comprehension. Table 6 depict the average comprehension accuracy over the three runs.

Table 6. Average comprehension accuracy.

	Read to Understand	Refactor to Understand
Function	5.2	7.6
Control Flow	4.9	5.9
Operations	4.7	6.3
Data Flow	4.5	4.5
State	6.4	7.0

As can be read from the table, the overall largest group differences were observed regarding function, control flow and operations knowledge. Differences on data flow and state comprehension were smaller. Thus, on average, the Refactor to Understand group outperformed the Read to Understand group.

To verify whether this higher comprehension accuracy was achieved in lower response times, we compared groups on their time measurements. In two out of three experimental runs, the Refactor to Understand group answered faster. Therefore, we can state that the higher accuracy achieved by the Refactor to Understand group was not due to restarting comprehension. These results thus demonstrate better recall for the Refactor to Understand group.

Summarizing, our results support the hypothesis that Refactoring to Understand improves comprehension. Concretely, two out of three experimental runs demonstrated higher accuracy of most comprehension categories.

5.2 Degree of Coupling as a Factor

Delocalization has been shown to affect comprehensibility [Letovsky and Soloway, 1986]. As the degree of coupling is an indicator of this delocalization, we argued that it should be incorporated as a factor. Our results support the assumption that the effect of Refactor to Understand is dependent on the degree of coupling in two ways: a) the usage of refactorings varied over the degrees of coupling; and b) the effect of Refactor to Understand differed over the degrees of coupling.

The degree of coupling affects the application of Refactor to Understand in the relative usage of refactorings. To comprehend the localized code fragment, mostly extract method was applied, while for the delocalized code fragment, mostly renames were applied (Table 3). When comparing the properties of the code fragments (see Table 2), an intuitive explanation would be that the delocalized fragment contained more local variables. This difference in properties can itself be explained by the difference in degree of coupling, as a delocalized implementation would rely on the collaboration of many instances, which would be reflected in more local variables, and therefore, in more rename opportunities.

Between the code fragments, a variation in the group differences was observed. The application of Refactor to Understand led to improved program comprehension of the moderately and highly coupled code, and to inferior program comprehension of the code fragment with no coupling. As these code fragments were selected to be good representatives of localization, semi-delocalization and delocalization, it is probable that this interaction between the effect of Refactor to Understand and the degree of coupling is representative too.

Therefore, while we cannot provide conclusive evidence that the degree of coupling affects the effect of Refactor to Understand on program comprehension, we have found good support for incorporating coupling as a factor. This allows us to refine our hypothesis. Correspondingly, we hypothesize that there is an interaction effect between the application of Refactor to Understand and the degree of coupling, and that more specifically, Refactor to Understand will be mostly beneficial for delocalized code fragments.

6 Threats to Validity

This work was set up as an exploratory experiment. Therefore, there are severe limitations to generalizability of its results. Moreover, practical restrictions did not permit the ideal setup. For clarity, we specify the main threats to validity.

6.1 Internal Validity

Internal validity is the degree to which the experimental setup allows to accurately attribute an observation to a specific cause rather than alternative causes. Specific setup deficiencies make it difficult to infer clear conclusions. The threats we have identified are:

- Maturation effects can occur due to processes within subjects which act as a function of the passage of time. As the tasks were similar over the runs, and the run order could not be randomized, all subjects performed the task on the localized fragment first, and the task on the delocalized fragment last. Moreover, the order of the comprehension questions was also similar. Therefore, order effects such as the learning and fatigue effect cannot be ruled out. However, as the instructions for the tasks were clearly given at the beginning of the experiment, it is unlikely that extra knowledge was gained during the execution of the first task.

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

- An instrumentation effect can occur when any instrument used varied over the experimental runs. We paid attention to find code fragments which are as similar as possible while varying in degree of coupling. The comprehension tasks were identical, and the open questions for each comprehension category were based on templates. However, even while applying all these precautions, we could observe in the results of the post-experiment questionnaire that there was a significant group difference regarding subjective evaluation of the difficulty of the tasks. More specifically, the tasks on the delocalized fragment were found significantly easier, and those on the localized fragment were found significantly harder. As we are unable to prove that this difference in subjective difficulty of the tasks was due to the degree of coupling, we are inconclusive about the extent of this threat.

6.2 External Validity

External validity is the degree to which research results can be generalized outside the experimental setting or to the population under study. As an exploratory study, this work was more designed towards investigating interesting phenomena then to be externally valid.

- Evaluation of the internal properties assure that the code fragments are representative for each of the three degrees of coupling. However, their size is clearly not representative. Therefore, we cannot generalize our conclusions for larger code fragments.

- Our subjects were last year computer science students. The experiment took place in the second semester, in an optional course on Object Oriented Reengineering. It is likely that these students are not representative for the population of last year computer science students, as their choice for this course might be an indicator of special interests and earlier expertise. Neither can we generalize our conclusions to the population of professional software engineers. Surely, there is little difference between these students and junior software engineers. However, more experienced software engineers might have developed optimal reading techniques and therefore would provide a totally different comparison.

7 Conclusion

These findings provide the first empirical support for the general claim that the process of refactoring can improve the understanding of software systems [Fowler, 1999], illustrated in higher accuracy of both the program and the situation model as mental representations of the source code. We have found some indications that the degree of coupling might affect this effect of Refactor to Understand, yet remain inconclusive on this subject.

Concretely, these results confirm that the introduction of beacons – recognition points that index into knowledge – and chunks – higher level abstractions of lower level structures – through renaming program elements and extracting methods can make the extraction of information from the source code easier. Especially, information on the functionality of the code, and its sequence of execution can be extracted more easily through the application of these refactorings.

We focus the applicability of these results as the input to confirmatory experiments, that test the refined hypothesis resulting from this work. More specifically, our refined insight into the effect of Refactor to Understand has led to the hypothesis that this reverse engineering pattern is most useful for highly coupled code. This hypothesis is still to be confirmed in further experiments, in which Refactor to Understand is applied on multiple code fragments representing a wide spectrum of coupling.

Concluding, this work provides the first exploration of the effect of Refactor to Understand on external quality attributes. Our results confirmed hypotheses regarding improvements in program comprehension, as predicted from the link between its key activities and known comprehension principles.

8 Future Work

In this work, we explored the application of two program comprehension techniques on small parts of a realistic software system (a mail client) with regard to program comprehension. We observed improved function and control flow comprehension of the subjects applying the Refactor to Understand pattern in two out of three experimental runs. Coincidentally, the parts of the software system used in these runs exhibited moderate and high coupling. In the experimental run on the code part with no coupling, no substantial difference could be observed. These results can merely be interpreted as a demonstration that the Refactor to Understand pattern can indeed improve program comprehension. The specific conditions under which such improved comprehension can be expected remain, however, unclear.

Therefore, an elaboration of the empirical comparison of these two comprehension techniques should be continued with an exploration of those properties of software systems which have the highest possibility of interacting with the comprehension technique. As it is natural to assume that program complexity interacts with the applied comprehension technique, sources of program complexity are good candidates for such properties. Among others, these sources include size, control flow complexity, degree of coupling,

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**

etc. Once a model of such hypothesized interaction factors is composed, a research plan can be composed in order to test the various combinations of the levels of these factors as efficiently as possible.

One other open issue is the assessment of open questions for each category of knowledge from Pennington's model as a test for program comprehension. Currently, while this model has been used in the comparison of comprehension of program paradigms [Wiedenbeck and Ramalingam, 1999, Khazaei and Jackson, 2002], we have no means to verify that high rankings on these questions are indeed an indicator for deep program comprehension. We suggest to verify this pragmatically by correlating these ratings with the correctness of specific tasks that require program comprehension.

9 Acknowledgments

This work has been sponsored by the Belgian National Fund for Scientific Research (FWO) under grants 'Foundations of Software Evolution' and 'A Formal Foundation for Software Refactoring'. Other sponsoring was provided by the European Science Foundation by means of the project 'Research Links to Explore and Advance Software Evolution (RELEASE)'. We also would like to thank the reviewers for contributing their valuable comments.

References

[Briand et al., 2001] Briand, L. C., Bunse, C., and Daly, J. W. (2001). A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Engineering*, 27(6):513–530.

[Cockburn and Smith, 2003] Cockburn, A. and Smith, M. (2003). Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Computers*, 15(3):387–407.

[Cohen, 1988] Cohen, J. (1988). *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates.

[Corbi, 1989] Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306.

[Crosby et al., 2002] Crosby, M. E., Scholtz, J., and Wiedenbeck, S. (2002). The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*.

[Demeyer and Du Bois, 2004] Demeyer, S. and Du Bois, B. (2004). Refactoring lab session. http://www.lore.ua.ac.be/Research/Artefacts.

[Demeyer et al., 2002] Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Gellenbeck and Cook, 1991] Gellenbeck, E. M. and Cook, C. R. (1991). An investigation of procedure and variable names as beacons during program comprehension. In *Empirical Studies of Programmers: Fourth Workshop*, pages 65–81.

[Kernighan and Plauger, 1982] Kernighan, B. W. and Plauger, P. J. (1982). *The Elements of Programming Style*. McGraw-Hill, Inc.

[Khazaei and Jackson, 2002] Khazaei, B. and Jackson, M. (2002). Is there any difference in novice comprehension of a small program written in the event-driven and object-oriented styles? In *Proc. IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 19–26. IEEE Computer Society Press.

[Letovsky and Soloway, 1986] Letovsky, S. and Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, pages 41–40.

[Miara et al., 1983] Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B. (1983). Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867.

[Mynatt, 1990] Mynatt, B. T. (1990). Why program comprehension is (or is not) affected by surface features. In *Proceedings of the IFIP TC13 Third Interantional Conference on Human-Computer Interaction*, pages 945–950. North-Holland.

[Oman and Cook, 1990] Oman, P. W. and Cook, C. R. (1990). Typographic style is more than cosmetic. *Commun. ACM*, 33(5):506–520.

[Parikh and Zvegintzov, 1983] Parikh, G. and Zvegintzov, N. (1983). *Tutorial on software maintenance*. IEEE Computer Society Press.

[Pennington, 1987] Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341.

[Sheiderman, 1976] Sheiderman, B. (1976). Exploratory experiments into programmer behavior. *International Journal of Computer and Information Sciences*, (5):123–143.

[von Mayrhauser and Vans, 1994] von Mayrhauser, A. and Vans, A. M. (1994). Comprehension processes during large scale maintenance. In *Proceedings of the 16th international conference on Software engineering*, pages 39–48. IEEE Computer Society Press.

[von Mayrhauser and Vans, 1995] von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55.

[Wiedenbeck and Ramalingam, 1999] Wiedenbeck, S. and Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Hum.-Comput. Stud.*, 51(1):71–87.

[Wiedenbeck and Scholtz, 1989] Wiedenbeck, S. and Scholtz, J. (1989). Beacons an initial program comprehension. *SIGCHI Bull.*, 21(1):90–91.

Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)
1534-5351/05 $20.00 © 2005 **IEEE**