CURBING DEPENDENCIES IN SOFTWARE EVOLUTION OF OBJECT-ORIENTED SYSTEMS

Mats Skoglund

Stockholm University
Department of Computer and Systems Sciences

Stockholm 2006
Abstract

Relationships between classes and objects in object-oriented software are necessary in order for the parts of the systems to provide dynamic behavior. These inherent relationships also create dependencies which can give rise to problems for software evolution of object-oriented software systems. Dependencies in software make systems difficult to understand, reuse, change and verify.

This thesis presents analytical and empirical investigations of dependency-related problems in software evolution of object-oriented software and on how such problems can be handled with dependency focused techniques, methods and processes. The research presented in this thesis includes: Development of a programming language construct for controlling dependencies; formal experiments on code inspection techniques; exploring change strategies’ effects on test suites; an industrial case study of regression test selection techniques for object-oriented software; proving the efficiency and defect detection capabilities of a novel regression test selection technique.

The thesis contributes to increased knowledge on the role of dependencies in software evolution of object-oriented software. Specific contributions are a programming language construct that can control access to dependencies in software. Other main contributions are insights on the efficiency of dependency focused code inspection techniques and contribution to the knowledge on dependency-based regression test selection techniques for large scale software. Another contribution is a novel change-based regression test selection technique.
ACKNOWLEDGEMENTS

Calm down, it’s only ones and zeros – Kathy Mar

First and foremost, I would like to express my deepest gratitude to my advisor Per, both for accepting to be my advisor and for excellent guidance through this work. Without your help I would never have finished.

I also give special thanks to my family. My Mom for being an excellent chef and for always giving a helping hand. My brother Anders for being a good listener and for long and interesting conversations about everything. My son Pontus for being Pontus, and for showing me what life is all about. I’m sorry Pontus for being this busy during the final writing of the thesis. My parents-in-law for nice dinners and good times.

I thank colleagues and friends at DSV. Harald for inspiring me to focus, for many interesting conversations, for hilarious lunches and for your Percy Nilegård imitation. Tobias for great conference trips, for all discussions the first years and for the \texttt{LaTeX} magic. Henrik for being a life-raft all the times when research and teaching collided. Terttu for believing in me, and Louise for all your help during the years after Terttu. Beatrice, Kjell, Fredrik, Janis and other people at the department who have made this journey easier and more fun. All colleagues that helped me with bureaucracy, computers, software, copiers and other things.

A great thanks also to my friends and family forcing me to stay in touch with the real world at times.

Other people I acknowledge are the staff at Swedbank and Swedbank IT who helped me when conducting case studies on their software. The master students who helped me by doing hard work and by providing interesting ideas, and the staff at the Forum library for always being nice and very helpful, no matter what obscure article or book I was out hunting for.

Thank you, all other who have contributed to my work with material, data, knowledge, viewpoints, advice, comments, suggestions, ideas and other things, you know who you are, thanks.

Last, but definitely not least, a very, very special thanks to my wife Mia for your endless support, encouragement and understanding, I cannot express my gratitude enough.

To my late Dad: I know you would have thought this was cool!
# Contents

## I Prologue

1 Introduction .................................................................................. 3
   1.1 Motivation .............................................................................. 3
   1.2 Outline ................................................................................... 4
   1.3 Research Questions .................................................................. 4
   1.4 Publications .......................................................................... 8

2 Research Methodology .................................................................. 11
   2.1 Research Strategies ................................................................. 11
   2.2 Research Design ...................................................................... 14

3 Background ................................................................................... 17
   3.1 Software Evolution ................................................................. 17
      3.1.1 Software Comprehension ...................................................... 19
      3.1.2 Change Impact Analysis ...................................................... 20
      3.1.3 Change Propagation ............................................................ 20
      3.1.4 Software Inspection ............................................................. 21
      3.1.5 Software Testing ................................................................. 22
      3.1.6 Unit Testing ....................................................................... 23
      3.1.7 System Functional Testing ................................................. 24
      3.1.8 Regression Testing .............................................................. 25
   3.2 Object-Orientation ................................................................. 26
   3.3 Relationships in Object-oriented Software .................................. 28
      3.3.1 Generalization ................................................................. 30
      3.3.2 Aggregation ..................................................................... 30
      3.3.3 Use or Association ............................................................ 34

4 Research Results .......................................................................... 35
   4.1 Research Evolution ................................................................. 35
   4.2 Contributions .......................................................................... 37
II  Publications  

**Paper I**  41  
**Paper II**  57  
**Paper III**  69  
**Paper IV**  87  
**Paper V**  103  
**Paper VI**  119  

III  Appendencies  155  

A  Proof of the Generalised Read-only Theorem  157  
B  The Questionnaire  169  
C  Summary Frequency Statistics  175
PART I

PROLOGUE
Chapter 1

Introduction

1.1 Motivation

Software constitutes important parts in many products and services we use each day, such as microwave ovens, cars, trains, buildings, mobile phones, bank services and other various services on the Internet. As the software is becoming more and more vital to us, the demand for high quality software increases. At the same time, the software products are growing larger and becoming more and more complex, leading to more opportunities for defects to sneak in during software development and maintenance. Software development organizations spend lots of resources on building software systems, and are thus interested in having their software functioning and providing values for as long time as possible. Thus, due to new conditions, both in the environment and in the user requirements, it is important that the software is maintainable, easy to adapt and change. However, large complex software are difficult to understand and reason about and this lead to considerable challenges for people developing and maintaining software.

Structured programming was introduced in the 60:s to help software developers focus on their current task at hand and letting them abstract away from irrelevant details [44, 40]. This later evolved into further concepts, such as modules, abstract data types and object-oriented classes, supporting similar ideas of abstracting away details to help software engineers [128, 21, 112]. This has been taken a step even further by the introduction of COTS (Components Of The Shelf) technologies where components may be created by one provider and perhaps used by many clients in many different types of systems [166]. The software engineers assembling components to larger systems may have little or no knowledge about the implementation details of the components they use in their possibly business critical application systems.

Even though the actual implementation details may be hidden in different types of abstractions, some kind of connection to the abstraction must still exist to be able to access and use the abstraction. When such a connection is estab-
lished, a *relationship* between the user of the abstraction and the abstraction itself is created, and relationships may lead to *dependencies* according to the following definition stated by Booch et al. [24]:

"A *dependency* is a using relationship that states that a change in specification of one thing (for example, class *Event*) may affect another thing that uses it (for example, class *Window*), but not necessarily the reverse."

For example, a component, *A*, that uses another component, *B*, must have some relationship to *B* in order to communicate. The relationship between the components may thus be a dependency if *A* is affected by changes to e.g. *B*’s location, ID, name or the communication channel’s protocol. Changing a component with dependencies may lead to ripple effects and cause change propagation where other components, apart from the changed, must also be changed [105, 134, 99]. Dependencies are ubiquitous in software, but they may also lead to problems such as ripple effects and understandability problems [41, 56, 88]. This is since the information needed to understand one line of code may reside at several other locations in the system [47].

### 1.2 Outline

The thesis is divided into three parts. The first part consists of an introduction in Chapter 1, which itself is structured as follows: In Section 1.3 the research questions are stated, followed by a description of publications included to this thesis in Section 1.4. Chapter 2 describes the research methodology used in this research. In Chapter 3 a background is presented, followed by Chapter 4 that summarizes the research results and presents the contributions.

The second part of the thesis contains six research publications that are the main contributions to this thesis. The third part contains appendencies.

### 1.3 Research Questions

The research question that drives the research presented in this thesis is stated as follows:

*In software evolution, how can object-oriented software be achieved efficiently and effectively, by focusing on dependencies?*

This research question is relevant since, as mentioned above in Section 1.1, when software is used in more and more vital functions in organizations and in society, the availability, reliability, usability and efficiency demands on software
increases. Furthermore, software is continuously evolving due to an ever changing environment and must thus be changed and adapted to new requirements, standards, laws and so forth. Hence, there is a need for tools, methods and techniques that aid in producing software effectively, not only when the software is initially developed, but also when it evolves over time. Effective is defined in the Oxford English Dictionary as "producing a desired or intendent result" [160]. However, this definition says nothing about resource limitations that are a reality in most software projects. Thus, not only should the software be achieved effectively but should also be achieved efficiently which, according to the Oxford English Dictionary is "working productively with minimum of wasted effort or expense" [160]. Achieving the "desired or intendent result" implies creating high quality software according to the IEEE definition of quality [83]:

1. "The degree to which a system, component, or process meets specified requirements."

2. "The degree to which a system, component, or process meets customer or user needs or expectation."

Quality is a broad concept that involves several aspects of the software which may all contribute to the overall quality of the software. For example, ISO describes six different major characteristics involved in the software quality concept: functionality, reliability, usability, efficiency, maintainability and portability [84]. Thus, both external (the customer's viewpoint) as well as internal (the project's viewpoint) attributes are involved. Each of the major characteristics are then further divided into several subcharacteristics that also can be used to describe the broad software quality concept.

The focal point of the research question is dependencies. Dependencies are inherent in software to allow for interaction between different parts of a software. However, as mentioned in Section 1.1, the existence of dependencies also lead to various problems, such as ripple effects, change propagation, understandability problems and problems with reasoning about programs. It is thus relevant to have dependencies as the gist in the enquiry for achieving software efficiently and effectively.

Software evolution involves a number of steps, such as specification, design, implementation, validation and verification and documentation. Each step could gain from improvements with respect to dependencies, such as for example:

- Requirements elicitation to produce better requirements specifications for change requests that take into account the effects of dependencies. A good knowledge of how change propagation affect different parts of the software could lead to better time and effort estimations for change requests.

- Implementation techniques and methods for implementing change requests at optimal locations and in optimal ways without causing unnecessary amounts of change propagation in the system.
• Validation and verification methods and techniques to reduce the number of defects efficiently, taking into account ripple effects and understandability problems caused by dependencies.

• Documentation methods and techniques could also be improved with respect to dependencies. This could give documentation where parts are identified that are sensitive to changes due to ripple effects. Having such information available when changing a system can lead to consequences of proposed changes perhaps can be anticipated earlier in the software evolution process.

Two of the areas listed above, namely implementation techniques and validation and verification, were selected to be studied in this research. The decision to select these particular areas are based partly on the author’s own background, knowledge and interests as a software engineer, and partly on problems described in the literature, e.g. [78, 15, 42, 123, 96, 158, 106].

The research presented in this thesis addresses the main research question above and is conducted by performing a number of studies, analytical as well as empirical. The studies are all directed toward the main research question by addressing the following six sub questions:

RQ1 How can problems with dependencies be reduced by controlling access to relationships between objects in object-oriented software?

RQ2 How are relationships between objects in object-oriented software used in practice?

RQ3 How useful and effective would controlling access to relationships be for coping with problems with dependencies?

RQ4 How effective and useful are dependency focused reading techniques for software inspections?

RQ5 How does dependency focused software change strategies affect maintenance of test cases?

RQ6 How effective are dependency-based regression test selection techniques?

RQ1 concerns the inherent relationships that exist in software developed in object-oriented programming languages having references, e.g. Java [89]. An object, A, referencing another object, B, through a reference can use B in any way permitted by B’s protocol (its public methods and variables). This can in theory lead to problems in software evolution if existing references to B are used in ways not anticipated when the software was originally written [122]. In Paper I a programming language construct is presented that can control access to references and thus control relationships between objects.

RQ2 and RQ3 concerns the practical effects of the theoretical problem described in RQ1. Even though a problem exists in theory, it may or may not be a
problem in the real world. Thus, it is relevant to investigate the practical effects of current programming languages’ weakness regarding references. It is also relevant to evaluate a theoretical solution’s practical effects. In Paper II the results from a questionnaire survey are presented that describes how experienced software engineers use encapsulation and their opinion on different tools and technologies for controlling access to relationships.

RQ4 involves understandability and reasoning of software with dependencies. As mentioned above, the presence of dependencies in software makes software harder to understand and reason about [125]. Software inspections benefit when the people performing the inspections have a high degree of understanding on what they inspect [48, 138, 169]. Investigating whether software inspection techniques focusing on dependencies are effective is thus motivated. Furthermore, a software inspection technique that is perceived to be useful should have a higher chance of being regularly used compared to a technique not being perceived as useful. Investigating whether a technique is useful in addition to being effective is thus also relevant. In Paper III, a dependency focused reading technique for software inspections is compared to a non-dependency focused approach in a series of experiment.

RQ5 concerns dependencies in software by addressing the software changing process, as opposed to addressing the programming languages as in RQ1. Unit test cases created with tools such as JUnit [13] are written in the same programming language as the system they test. Since such unit test cases call methods in the tested system directly, the test cases are inherently dependent on the parts of the system they test. For example, when a tested method’s interface changes, its associated test case must also be changed in order to function together with the changed method. Thus it is relevant to investigate change strategies that take into account how ripple effects impact test cases. In Paper IV, a study is presented where a dependency focused change strategy is explored to investigate its effect on test suites.

RQ6 concerns using dependencies as the focal point for selecting test cases for regression testing. When re-testing a system after a change, not only the changed part should be retested, but also parts that have a risk of being affected by the change [110]. One technique that has been proposed for selecting parts that should be retested based on dependencies is the Class firewall regression test selection technique [82]. This technique had, to the author’s knowledge, not been evaluated on a large-scale industrial system prior to this study. It was thus motivated to investigate its effectiveness in a real-life, large-scale, complex industrial context. In Paper V, a study is presented where the Class firewall regression test selection technique is evaluated on a large scale software system. In Paper VI the technique is analytically compared to a change-based technique with respect to effectiveness and other properties.

The six research questions are dealt with through a series of studies, reported on in six publications.
1.4 PUBLICATIONS

Included in this thesis are the six publications listed below. For each publication it is describes which research question it addresses and the author’s individual contribution.


The following publications are related but not included in the thesis:

1.4. PUBLICATIONS


The publications in this thesis are included in the chronological order and this order also reflects the research evolution as presented in Section 4.1. However, a suggested reading order is the reverse chronological order, i.e. to start with the most recent work and finish with the least recent.
Chapter 2

Research Methodology

2.1 Research Strategies

When the purpose of the research is to understand a poorly investigated phenomenon, one possible research approach is to conduct an explorative study [139, p. 80]. An explorative study may be conducted using a qualitative approach with a flexible design to generate new ideas and ask more and new questions regarding the poorly investigated phenomenon at hand. In qualitative research, real objects are studied in their natural settings and environments and the answers are not quantifiable in terms of numbers but are rather expressed in words [139, p. 5]. With a flexible design the research is not fixed to a predefined specification of how the investigation should progress during the enquiry, but may instead adopt to new insights gained during the study [139, p. 5]. When an overall insight has been gained and new questions and ideas arise, perhaps a more focused enquiry is undertaken, narrowing down on a specific issue found promising in the explorative study. Then, a quantitative research approach may be taken, collecting quantitative data that can be compared statistically. With a quantitative approach, the phenomenon or relationships in the phenomenon are compared or quantified, e.g. to identify cause-and-effect relationships [179].

Both the qualitative and the quantitative approaches may be categorized as empirical research strategies. Within empirical research a specific study may be categorized further with respect to its purpose. According to Babbie, a study may be explorative, explanatory or descriptive [5]:

- **Explorative.** As mentioned above, an explorative study may be conducted to gain insights and new views of the phenomenon. For example, by using an open-ended questionnaire to get more information before a more thorough investigation is performed.

- **Explanatory.** The purpose of an explanatory study is to explain a relationship or a concept in the studied population. For example, to explain why some programmers prefer one programming language before another.
• *Descriptive.* The purpose of descriptive studies is to present the distributions of attributes about the phenomenon or to classify observations [179, p. 7].

As mentioned above, empirical research may be qualitative or quantitative and explorative, explanatory or descriptive. Depending on purposes and type of results expected, explorative research can be said to be more qualitative in its nature, while explanatory and descriptive research are more related to quantitative approaches.

![Research Strategy Diagram](image)

**Figure 2.1: Research strategies and methods.**

Software engineering research, as other research, involves the use of research methods. By applying known research methods it is possible to make use of previously collected knowledge, experience and techniques to answer the research questions effectively. Depending on the research question to be addressed, different methods are more or less applicable. Methods for conducting empirical research, both qualitative and quantitative, are surveys, experiments and case studies, see Figure 2.1.

**Survey.** Survey research "is a comprehensive system for collecting information to describe, compare or explain knowledge, attitudes and behavior" [130]. A survey may be conducted as interviews or with questionnaires, see Figure 2.1. A survey is conducted on a sample that is drawn from the population being studied. Then, the results are analyzed and general conclusions are drawn for the whole population.

**Experiment.** Another empirical research method is a *formal experiment*, see Figure 2.1. An experiment is a formal test to study a phenomenon, preferably
in a laboratory environment [179, p. 9]. A formal experiment may give a higher degree of control over the studied objects compared to e.g. a case study performed in a real-world organization [90]. An experiment may be conducted to try to quantify the degree of effect one factor has on another factor [117, p. 1].

![Diagram of an experiment](image)

Figure 2.2: Example of an experiment. A random sample is drawn from the population and randomly divided into experimental groups. Then a factor variable is varied between the groups and results are observed. Finally conclusions are drawn and generalized to the population.

An example of an experiment setup may be described as follows, see also Figure 2.2: First is a random sample drawn from the population of interest. The individual objects in the sample is then randomly assigned into one of two groups. Then the value of the independent factor variable is varied between the two groups while other factor variables are fixed or controlled. By measuring the effects on the dependent factor variable in the two groups, a conclusion may be drawn about the degree of cause-and-effect relationship between the dependent and the independent variables. When the sample is random and the assignment to groups is also random, the experiment is a randomized controlled experiment [139, p. 123]. In some cases it may be difficult to achieve a pure randomized experiment, then a quasi-experiment may be conducted instead. In a quasi-experiment, the researcher should handle the issues introduced by the non-randomness of the experiment by analyzing properties of the sample groups, this is thoroughly discussed in [39].

**Case study.** A case study may be conducted to study a phenomenon in its real setting and environment. A case study is a useful research method when it is difficult to isolate the phenomenon from its environment, and when the environment interacts with the phenomenon [184, p. 13]. In a case study, one or more small or larger cases are studied and data is collected. Depending on the
purpose of the research, data collected may be quantitative or qualitative. With quantitative data the material may be treated statistically and with qualitative data it may be treated with e.g. protocol analysis or other analysis techniques [126]. A case study may be conducted in a laboratory or in the industry, see Figure 2.1. An industrial case study must sometimes be performed to be able to study a phenomenon in its real environment, e.g. studying the introduction of a test process in a software development company where the company’s staff, development environment, testing environment and software products are part of the study subject. A laboratory case study may be conducted when the case study environment may be replicated, e.g. to study the distribution of defects in an open-source software where the software may be installed and executed in the researcher’s own laboratory.

In addition to empirical research approaches, an analytical approach may also be used, see Figure 2.1. The analytical research may be described as to “propose a formal theory or set of axioms, develop a theory, derive results and if possible compare with empirical observations” [2]. This method may be used when for example building a mathematical model of a software. By manipulating the model, some properties of the software may then be shown mathematically [8].

2.2 RESEARCH DESIGN

In this research, analytical as well as empirical studies have been conducted. This section describes the research approaches, research methods and data collection techniques used for the studies presented in this thesis, see Table 2.1. In this section, the word "Study" is used to denote the studies presented in each of the research papers included in this thesis. One research paper corresponds to one study performed, e.g. PAPER I corresponds to Study 1.

<table>
<thead>
<tr>
<th>Study</th>
<th>Strategy</th>
<th>Approach</th>
<th>Purpose</th>
<th>Method</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analytical</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Empirical</td>
<td>Quantitative</td>
<td>Descriptive</td>
<td>Survey</td>
<td>Questionnaire</td>
</tr>
<tr>
<td>3</td>
<td>Empirical</td>
<td>Quantitative</td>
<td>Exploratory</td>
<td>Experiment</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Empirical</td>
<td>Qualitative</td>
<td>Exploratory</td>
<td>Case study</td>
<td>Laboratory</td>
</tr>
<tr>
<td>5</td>
<td>Empirical</td>
<td>Quantitative</td>
<td>Exploratory</td>
<td>Case study</td>
<td>Industrial</td>
</tr>
<tr>
<td>6</td>
<td>Analytical</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Overview of research.

Study 1. Study 1 is an analytical study where a new construct for an object-oriented programming language is constructed. The construct introduces features in the programming language that can be used to control the access to some relationships in a program. These properties of the construct are shown mathematically using type systems and operational semantics.
2.2. RESEARCH DESIGN

Study 2. Study 2 is a follow-up study to Study 1 where the programming language feature introduced is evaluated using an empirical approach. The programming language construct is evaluated with respect to usefulness and to alternative solutions. Study 2 is also a follow-up study to another study where software engineers were interviewed on the same subject. Knowledge gained from the previous interview study was used when designing the questionnaire study. The study is mainly a quantitative descriptive study, although with some explanatory parts. The data collection was conducted using an electronic questionnaire and the questions asked were similar to the questions asked in the previous interview study, although adjusted to the purpose of this study. New questions considered relevant were also added. These new questions were based on indications resulting from the previous interview study that was interesting to pursue further. The subjects in the study were software engineers reading software engineering newsgroups on the Internet and thus the sample was not a pure randomized sample but includes those who read the newsgroups and were motivated to answer the questionnaire. This is further discussed in Paper II.

Study 3. Study 3 is an empirical, quantitative, explanatory study conducted as a formal experiment with replication. The purpose was to evaluate a dependency based code inspection reading technique for object-oriented source code. The target population was software engineers, both inexperienced and experienced. The experiment comprises 157 university students on two different software engineering courses. The sample was a convenience sample in that the courses selected were courses the author was involved in at the time. Thus, the experiment may be viewed as a quasi-experiment [39]. However, the sample comprised all students on both of the courses and the courses were software engineering courses and thus appropriate to perform such experiments on. The first course is compulsory for all students on a computer and systems sciences programme and is given on the second year of the programme. This implies that the students are probably not very experienced in programming or code comprehension and probably they are not all motivated to conduct code inspections. The second course was a voluntary course given to third and fourth year students in academia and in computer industry which probably imply more motivated and experienced students with respect to software engineering, software quality, programming and code inspections.

Using students as subjects in software engineering experiments is debated, see e.g. [146, 80, 73, 79, 154]. One issue is for example whether students are representative for the professional software engineering community when evaluating programming languages, development tools and software inspections techniques. This issue is further discussed in the context of the conducted experiments in Paper III.
Study 4. Study 4 is an empirical, qualitative, explorative case study conducted in a laboratory. The purpose was to explore dependency-based strategies for conducting code changes and their effect on unit test and functional system test suites. The case study was performed by changing a software using change strategies and explore how the changes propagated to the test suites. The case studied was a software written by other for different purposes than this study and had been used in various case studies by other researchers. The change strategies and the change request were defined by others for their purposes and the unit test cases were constructed by the software creators for testing purposes. The system test cases were constructed by the author, based on a functional specification created by others. The case study was performed as a laboratory cases study since it did not require any real-life organisation and the environment for running the software was possible to setup in a laboratory.

Study 5. Study 5 is an empirical, quantitative, descriptive and explanatory case study conducted in the industry. The purpose was to study a dependency based regression test selection technique for functional system test cases on a large, industrial system. Since the purpose’s scope involves a real-world environment, a industrial case study was motivated. The case that was studied was an industrial, large, object-oriented distributed system. The functional system test cases and the different versions of the system used in the case study were created by the organization to support their every-day business.

An alternative to performing an industrial case study could would be to perform a laboratory case study. By conducting a laboratory case study some parameters may be easier to control compared to an industrial study and thus the internal validity of the study may increase. For example, in this study it was not possible to restrict valid users to access the system. Thus, when test cases were executed and source code traces were collected this could have been affected if other than the test engineers used the system during the data collection process. However, since the scope of this study was to investigate the real-world, a toy system using fictional test cases could not suffice and thus it was motivated to conduct an industrial case study. Furthermore, replicating the industrial system and its operational environment was not realistic due to the system’s complexity and size.

Study 6. Study 6 is an analytical study that is a follow-up study to Study 5. Observations in Study 5 lead to a hypothesis regarding the effectiveness of the dependency based regression test selection technique used in the previous study. The purpose of the study was to show how the dependency based regression test selection technique can be made more effective by focusing on dynamic dependencies as opposed to static dependencies used by the selection technique. To prove the hypothesis, a theorem on how object-oriented programs may behave during execution of test cases is stated and proved.
Chapter 3

Background

3.1 Software Evolution

Successful software may live a long time after its first release. Since customer demands and operating environments change over time, successful software must evolve to meet these new demands and continue to give value to its users. Software evolution is the set of processes, procedures and activities intended to create new versions of released software [15].

The following five-stage model for the software life-cycle is presented by Rajlich and Bennet [135]:

1. Initial development. The first version of the system is developed.
2. Evolution. The system evolves and new features are added and other are changed.
3. Servicing. Only small functional changes and defects are fixed.
4. Phase-out. No more servicing of the system. The system is about to be retired.
5. Closedown. The system is retired, perhaps replaced by a new system, if one exists.

In the initial development stage the system is originally developed to its first version. The overall architecture is defined and its goals are determined. Goals of the architecture may be of performance, maintainability, testability, understandability, reusability or reliability. Anticipated changes may be regarded in the original design while unforeseen changes are harder to predict and thus harder to design the architecture for.

In the evolution stage, new features requested from customers are added, defects are repaired (corrective maintenance) and changes due to changed operating environment are conducted (adaptive maintenance). Also the software
may be changed to enhance performance or other properties in the system (perfective maintenance).

Servicing stage is entered when evolution of the system has slowed down. The organization withdraws staff from the software project and only small modifications to the software are made.

Phase-out and closedown stages regard the retirement of the software. The organization focuses on either replacing the system or manage without it. Only absolutely necessary changes are made to the software to have it working during the phase-out and closedown stages.

The initial development stage is somewhat similar to the evolution stage [129]: Analyze requirements, evaluate the system and program design, write and review code, test changes and update documentation.

There are however differences between development and evolution. The people working with evolving a system may be different from those who originally developed it, thus the maintainers may not have the same understanding and knowledge of the system as the developers [168, 174]. Instead, the maintainers may have to gain knowledge of the system by working with it, studying the documentation and source code and, if possible, communicating with the original developers. Thus, there is a risk that maintenance tasks take longer than expected, are performed in ways not anticipated by the original developers or that the results are in conflict with the goals of the original system design. Moreover, software evolution is performed in the context of an existing system and is thus bound to conditions defined perhaps several years ago. The system may also have users relying on features in the system for their everyday work. Thus, defects introduced in those existing features during software evolution may perhaps be more critical to the organization, compared to defects in a new system people have not become dependent on.

Changes are made to the software in all stages after the initial development stage. The change process is described as follows by Yau et al. [183]:

1. **Request for change.** Change requests may be of several types and originate from different sources. For example, the users may request changes to functionality, maintenance staff may request changes to architecture due to new technology, operational staff due to changed server hardware or software, management due to policy or legislation changes, etc. A change request is evaluated from different perspectives, e.g. economical and technical, and is either rejected or accepted.

2. **Planning**

   - **Software comprehension.** How and where the change should be made must be determined. This requires for software comprehension where the system must be understood in order to determine the optimal way of performing the change.
   - **Change impact analysis.** To aid in deciding where a change should be implemented, a consequence analysis should be performed to determine the effects of possible implementation alternatives.
3. **Change implementation.**

- **Restructuring for change.** Software changes degenerate the structure of the software and the architecture must thus be maintained to avoid decay [14]. A part of the change process may be to restructure the system to restore the architecture, but also to isolate the concepts where the change should be made to prepare for an effective implementation [16, 68].

- **Change propagation.** When possible restructuring and change request implementation is performed, the secondary changes required due to the original changes should be handled.

4. **Validation and verification.** Verification and validation may be performed by inspections and testing, as for the initial development.

5. **Re-documentation.** Re-documentation is the last phase of the change cycle. In this phase the documentation should be updated to reflect the changes made in the software. When this phase is done the change process is finished.

The scope of this thesis is the change implementation and validation and verification stages of the change process in the evolution steps of the software life cycle. However, software comprehension and change impact analysis presented in the planning step are also relevant in other steps in the change process. Software comprehension is relevant also for re-documentation to properly document the changes made, and in validation and validation to e.g. perform code inspections and create test cases [47, 97]. Change impact analysis can also be useful in steps after planning to e.g. aid in determining change propagation and selecting which parts need retesting. Steps in the change process related to the content in this thesis are further described below.

### 3.1.1 Software Comprehension

Knowledge and understanding of code is needed in several software evolution tasks, e.g. in change cost prediction, location of elements to change, incorporation of new code into existing, code inspections and documentation [168, 47, 97, 99].

The task of locating the concepts in the code where changes should be performed may consume as much as half of the maintenance resources for an organization [16].

Some models for software comprehension are the top-down model and the bottom-up model and the combination of the two [173]. The top-down model is normally used during understanding of the software if the programmer is familiar with the software domain. Starting from the domain knowledge, the programmer conceptually decomposes the system into its elements that should exist based on the knowledge of the domain. These elements can then be further decomposed to find relevant details needed for the task at hand [173].
A bottom up model called the program model is based on the control flow of the program [172]. The programmer reads the comments and the code and follows the control-flow of the program to understand the relationships in the software. Then the data structures and variables are examined and the combined information is used to build a knowledge base of the software. Following the control-flow in a program is especially difficult in object-oriented software since the implementations of features may be spread over several interacting classes and methods in the system [175, 177].

3.1.2 Change Impact Analysis

Change impact analysis is the process of determining the consequences of a change. This can be performed manually but may also in some cases be automated. During planning, software change impact analysis can be performed to estimate the consequences of a proposed change. This information can be useful for scheduling, resource planning and cost estimation.

Risks of failing resource estimates may be reduced by identifying the consequences of a change before resources are spent on its implementation, e.g. to avoid a situation where it is found late in the change implementation phase that major unanticipated alterations in a large complex module are required to complete the change request.

Since a change may be performed in more than one way, the most appropriate way according to some criteria should be selected. This may be supported by performing change impact analyses on the available alternatives. For example, if the criterion is test effort, by choosing the alternative leading to the least number of test cases needing to be re-executed then test time may be saved.

Change impact analysis may also be used to identify parts in the software that are more sensitive to changes than other parts, e.g. by simulating changes to individual components and measuring the impact on other components. This information may be used to guide maintainers and help them to decide which parts perhaps should be avoided to change.

Change impact analysis may be useful in determining the scope of testing. By performing change impact analysis after the change request has been implemented, components that are possibly affected by the changes may be identified. This information can be used to determine which parts need re-testing.

3.1.3 Change Propagation

A change to a component affects other components due to dependencies within a system. One possible effect is that some other components related to the changed component must be adapted to work together with the changed component again. This is sometimes referred to as ripple effects or change propagation [134, 182]. Change impact analysis can aid in determining the components that may be affected by a change and thus require to be examined after a change. When the programmer examines the possibly affected components
it may be decided that follow-up changes are required in these components in order to reach a consistent system. For example, a syntactic change, such as renaming a method in Java, requires all calls to this method to be updated to use the new method name after the change. However, not all changes require such follow-up changes. For example, changing the name of the default file-name in the argument to a method from "Noname" to "noname" may not require any further changes in the called method. When a programmer has performed a follow-up change, perhaps a new change impact analysis is performed to determine a new set of components possibly affected by the follow-up change and the process is repeated.

In object-oriented software, the change propagation is more difficult to predict by the programmers compared to in procedural languages due to the dynamic behavior caused by object-oriented features such as polymorphism, inheritance and dynamic binding [152]. When all affected components have been changed and the system has reached a consistent state, the changes should be verified and validated.

3.1.4 Software Inspection

Software inspection is a process for finding defects in software project artifacts. The software inspection process is based on humans reading and critically reviewing documents related to the software, e.g. source code, design diagrams or requirements specifications. With knowledge of the application domain and of typical defects in the type of documents inspected the reviewers search for and point out potential defects in the documents.

Inspections complement testing in several ways. For example, testing is performed on an executable program while inspections may be performed on other static artefacts representing the software. Thus, with inspections it is possible to find defects before any executable program even exists. Software inspections may also detect defects not related to the software's behavior, defects testing cannot find, e.g. conformance with standards or organizational policies. On the other hand, testing may find behavioral defects difficult to find with inspections, e.g. performance or usability issues. Inspections may also be used for knowledge transfer since the reviewers will get some understanding of the software they are inspecting. This can be useful for e.g. newcomers in the team. In addition to finding defects related to the execution of the program, software inspections can also aid in making the code easier to understand and change [153]. Inspections are a useful method for finding defects in software in both a software evolution context as well as in initial development context [123].

Software inspections may be performed in different ways. Fagan, the father of software inspections, describe software inspections as a six step process [55]:

1. Planning. People are assigned to the inspection team. Time and location of the inspection is decided and scheduled.

2. Overview. The artifact that should be inspected is presented to the rest of the team by a team member, e.g. the author of the document.
3. **Individual preparation.** The team members review the artifact individually. They learn about the artifact and search for defects.

4. **Group meeting.** The team walks through the artifact in a meeting and the members bring up the issues found during the individual preparations for discussion.

5. **Edit.** The artifact is reworked and the defects are repaired. Depending on the outcome from the inspection a new inspection of the artifact may be necessary.

6. **Follow-up.** It is verified that the defects are repaired accurately.

The inspection process may vary from the original process described above. For example, the number of members in the inspection team may be varied, from one single reviewer to several reviewers [131, 127]. The degree of structure may also vary, e.g. formal inspection meetings using reading techniques, or just performing a walk-through to get feedback from colleagues.

The steps in the inspection process may be supported to enhance the effectiveness of the whole process. By using specific reading techniques, the individual preparation phase is supported. One reading technique is to use a checklist containing typical defects to search for [31]. A checklist for code inspections may for example contain entries such as "does the code conform to the coding standard?" or "are all variables initialized before they are used?". A checklist for software inspections should be updated regularly, not be to general, and not exceed a single page [31]. Another reading technique is the **stepwise abstraction** reading technique. With stepwise abstraction, the reviewers read the source code and then construct their own specification based on their understanding of the code [9]. These specifications are then compared to the original specification and any deviations are examined. A reading technique not giving any support to the reviewer is the **ad-hoc** reading technique. With this technique the reviewer must use its intuition and knowledge to find defects in the code [102].

When performing software inspections the reviewers should be familiar with the artifact that is reviewed. This requires software comprehension and knowledge of the software, its structure and behavior to be effective. Thus, inspecting code that is a part of a large complex software system is difficult. Dependencies require the inspectors to trace the control flow of the software through possibly many components and modules in order to understand the collected behavior of the software to determine whether it is correct or not [114].

This is especially difficult in object-oriented software since rule of thumbs say that methods should be short and the number of relationships in object-oriented sub-systems should be high [113].

### 3.1.5 Software Testing

By performing software inspections, defects in the software may be found. However, since software inspections focus on the static artifacts of a system’s representations it should be complemented with testing [101, 148]. Software testing
regards the execution of software under well defined conditions to observe and evaluate its dynamic behavior.

In *specification based testing* the specification of the software is used to define the conditions for the tests and for evaluating the results. Specification based testing is sometimes called *black-box* testing since the software is regarded as a black box that responds to some input by producing some output, the result [63]. How the result is produced, i.e. the implementation, is irrelevant during specification based testing. By using the specification, test cases are constructed that contain information regarding the input, execution context and the expected result [83]. Using the test cases, the software is executed in the specified execution context using specified input and produces some result [33, pp. 21-22]. The result produced is called the actual result and it is compared with the expected result specified in the test case. Any deviations between the actual result and the expected result are the manifestation of a defect and should be handled.

A piece of software may be tested on different levels, e.g. unit, integration or system level [100]. Unit level tests test small parts of the software in isolation, e.g. a method or a class. Integration tests test larger subsystems where the smaller parts have been combined and interact with each other, e.g. a cluster of related classes. System tests are used when all parts of the software have been integrated to test a complete system [88].

### 3.1.6 Unit Testing

Testing smaller parts of a system in isolation is referred to as unit or module testing. The smallest testable part in object-oriented programming is the class. A class may be tested by calling its methods in an execution context and evaluating the results produced. A single class may however be difficult to test in isolation due to dependencies to other classes [100]. Thus, to test an individual class, some minimal integration may be necessary to create a cluster of related classes [18]. A cluster of classes is more complex than a single class and it may thus be difficult to locate in which of the classes in the cluster a failed test case originates from. To reduce the complexity of using class clusters, stubs may be used to serve as stand-ins for server classes needed by the class under test [18]. A stub is a piece of software that simulates a class that is needed to test another class [29].

However, a class cluster where class A is dependent on class B, that is dependent on class C, that is dependent on the class A, is a cluster having dependency cycles, see Figure 3.1. In clusters with dependency cycles it must be determined in which order the classes should be tested and thus which classes should be replaced by stubs [100].

A system may contain many classes and one single class may require many test cases to be thoroughly tested. Automating unit testing may aid in organizing and executing unit test cases. One way of automating test cases is to implement the test cases in the same programming language as the software itself is written in. By creating test drivers that calls methods in the class under test and then implement code for checking the results, tests may be executed
and evaluated automatically. There are frameworks available that can aid in creating such tests, one example is the JUnit framework for Java\(^1\). Since automated tests are pieces of software, they are exposed to similar problems as other evolving software. The class under test is dependent on its stubs and thus when the class the stub simulates is changed, the stub must also be changed to reach a consistent test system again. Similarly, both test drivers and test cases created with framework such as JUnit are dependent on the classes they test, and thus they must also evolve to be consistent with the system they test.

### 3.1.7 System Functional Testing

When all units are tested in isolation and when all subsystems or clusters of classes are tested, the complete system containing all subsystems may be assembled and tested [109]. This is referred to as system testing and is aimed at verifying that the requirements are fulfilled [33, p. 163].

System tests may be of different types, e.g. functional tests, performance tests, security tests, stress tests, recovery tests, etc. The goal of functional system tests is to validate that the functional requirements are fulfilled and that the customer gets what is expected.

Functional system tests are in its nature black-box tests since it is the external view of the system that is tested [33].

According to the literature, functional system tests is usually based on use cases [65, 162, 33, 30, 132, 19, 72]. Use cases may be used as the basis for both object-oriented systems as well as traditional systems since it is the use of the software that is modeled by use cases, not the implementation [24, p. 225].

A use case is a scenario describing the typical use of a software and thus this kind of testing may also be referred to as scenario testing [19]. However, a use case may not always contain all information required in a test cases, such as detailed information regarding input and expected results [19]. Instead, a

---

\(^1\)http://www.junit.org
3.1. SOFTWARE EVOLUTION

A test case scenario begins with a user initiating a transaction or a series of events the system should respond to. Then the actual results produced by the system may be evaluated against the expected results specified in the test case describing the scenario [81].

3.1.8 REGRESSION TESTING

When software is changed, the software should be re-tested again to verify its behavior. This may be performed by running unit tests, integration level and system levels test as for the initial development of the software. If test cases were constructed during the initial development they can be reused and re-executed on the changed version of the software. This is sometimes referred to as regression testing and is aimed at verifying that features that worked in the previous version of the software still works after a change [83]. Regression testing may also be expressed more formally as: For a baseline version of a software, $P$, and a successor version of the same software, $P'$, a baseline set of tests, $T$, used to test $P$, regression testing attempts to use $T$ to verify $P'$ [141].

Regression testing performed by re-executing the full test suite is costly, especially when the tests are executed manually. Thus, by selecting those test cases with a high probability to detect new defects due to the changes, a subset of the full test suite may instead be executed. This should be more cost effective compared to executing all test cases. Using a selection strategy, the regression testing process may be described as [110]:

1. Identify the changes in $P$.
2. Select those test cases, $T' \subseteq T$ that should be re-executed on $P'$.
3. Execute the test cases in $T'$ on $P'$ to verify $P'$ with respect to $T'$.
4. Evaluate the results of the tests.
5. Identify and repair the found defects.

Different criteria may be used to select the test cases that should be re-executed on a changed software. One criterion can be that the selected test cases should be those test cases that may reveal a defect in the changed software. Several regression test selection strategies have been proposed, e.g. [3, 143, 10, 20, 37, 69, 104, 107, 144, 142, 141, 109, 74].

When choosing a regression test selection strategy the cost of applying it should be considered. In order to have an effective test selection technique, the cost of selecting and executing test cases should not exceed the cost of re-executing all test cases in the test suite [108].
### 3.2 Object-Orientation

In this section basic object-oriented principles are described and defined. Different types of dependencies between classes and objects in object-oriented software are also described. A reader already familiar with basic object-orientation and relationships in object-oriented software may skip this section.

Object-oriented programming started in late 60’s with the programming language Simula [21]. Simula introduced the concept of an object class where data and operations that manipulates the data are encapsulated together into a single unit [120]. Software developed using object-orientation is said to be easier to reuse and change compared to software developed with procedural languages such as Pascal and C [120, 1, 178, 23].

In object-oriented programming, the data and the operations that manipulate the data are kept together in a **class**. The class serves as a blueprint for creating **objects** that are instances of classes. A class can have **member variables** and **methods**. The member variables hold the data for the class and the methods realizes the operations that can be performed on the data. The class **interface** lists the operations that may be performed on the class. In addition to classes and objects, some other features associated with object-oriented programming are: **inheritance**, **polymorphism** and **dynamic binding** and **encapsulation** and **information hiding** [50, 45].

**Inheritance.** Classes may reuse from other classes by the use of inheritance that let classes share behavior and data with other classes. A child class that inherits from a parent class is called a **subclass**, see Figure 3.2, and the parent class is called a **superclass**. The subclass inherits member variables and methods from its superclass. By defining new variables and methods, the subclass may also extend its inherited behavior. By redefining variables and methods the subclass may change its inherited behavior.

For example, in Figure 3.2, class `B` inherits from class `A`, i.e. `A` is the superclass to `B` that is the subclass. Since `B` is a subclass to `A`, `variable_1` and `method_1` are inherited from `A` and can be used in `B`. Variable `variable_2` and method `method_2` are also inherited but redefined in `B`. The inherited behavior from `A` is extended in `B` by the new variable `variable_3` and method `method_3` defined in `B`.

There may also exist **abstract classes** that specifies operations but does not provide any implementations for them. Objects may not be created from abstract classes but they are instead used expressly for creating class hierarchies.

**Polymorphism and dynamic binding.** "A polymorphic function is one that can be applied uniformly to a variety of objects" [120]. In object-oriented programming the same method identifier may be defined in more than one class. For example, a system for geometric figures with the classes `Rectangle` and `Triangle` that both define a method named `rotate`, see Figure 3.3. The system may rotate objects of both the class `Rectangle` and class `Triangle` since
they both define the same operation. The implementation of the operation may perhaps be different in Rectangle compared to Triangle. Polymorphism and inheritance are closely related in object-oriented programming. A requirement for using the same polymorphic operation in different classes is that the classes have a common superclass. In the example, both Rectangle and Triangle are subclasses to the abstract class Shape.

By *dynamic binding* a variable declared of a certain class may be assigned objects of subclasses to that class. Then it is determined dynamically at runtime which class the variable is actually assigned to. For example, in Figure 3.4, an object of class Rectangle or Triangle may be assigned to a variable.
declared of class Shape. Then it is determined in run-time whether the rotate method Rectangle or Triangle should be used.

```java
Shape figure;
// Due to some condition either a Triangle
// or a Rectangle will be assigned to the
// figure variable
if(condition is True)
    figure = new Triangle();
else
    figure = new Rectangle();
...
figure.rotate(); // When compiling, the class of
// figure is not known.
// It is determined in run-time
```

Figure 3.4: A variable of type Shape may be assigned objects of subtypes to Shape.

**Encapsulation and information hiding.** Encapsulation and information hiding are often used interchangeable to refer to the same concept. Another view is that encapsulation is a technical programming language feature implementing the design principle of information hiding [140]. Encapsulation regards the packaging of data and operations that manipulates the data, i.e. the member variables and methods, into a class. This should increase the independence of the module since the class is self contained and is reusable without modifications, and may be changed independent of other classes [159].

Information hiding regards hiding the details of the internal structure and the implementation of a class. By publishing only the interface to a class, the internal implementation can be changed without affecting the users of the class. For example, a PhoneList class with AddEntry and GetPhoneNumber as operations may use a simple file structure to store the entries in the phone list, see left part of Figure 3.5.

When the number of entries grows, the designer decides to change the implementation to store and retrieve entries from a relational database instead of using files. This implementation change should not affect the users of the class since the class’ interface is kept intact, see right part of Figure 3.5.

### 3.3 Relationships in Object-oriented Software

Object-orientation is said to facilitate development, reuse and maintenance of software. As mentioned above, it should be possible to change the implementation of the PhoneList class without affecting the users of the class, since the implementation is hidden from the users. However, objects and classes need
3.3. RELATIONSHIPS IN OBJECT-ORIENTED SOFTWARE

Figure 3.5: A PhoneList class where the interface specify the methods AddEntry and GetEntry. The implementations of the methods are changed but the interface remain unchanged (the methods’ interfaces highlighted with boxes) and thus users of the PhoneList class should not be affected.

to work together and communicate with each other to achieve a dynamic system. When the classes collaborate, they need to exchange information, for example send parameters and return values from method calls. These collaborations create relationships between classes and sometimes these relationships are also dependencies which sometimes create problems when working with object-oriented software.

By measuring the number of relationships, the quality of the design of software may be evaluated. When measuring the relationships, the metrics used is called coupling and can be defined as "the measure of the strength of association established by a connection of one module to another" [163]. Coupling often refers to the same concept as relationships but also adds an extension to the relationship concept since it includes a valuation of the strength of the relationship. This gives the possibility to compare and measure the degree of relationship between modules. Coupling may however be measured in different ways by defining different properties of the relationships to be measured. Briand et al. report that they found more than 30 different measures on coupling when searching the literature [27].

This section describes different types of relationships and problems related to relationships. In this thesis, we differ between three types of relationships: generalization, aggregation and association. In the framework for coupling measures presented by Eder et al., these relationships are referred to as inheritance coupling, component coupling and interaction coupling respectively [52].
3.3.1 Generalization

The relationship between something more specific to something more general is referred to as a generalization relationship. Two classes where one is a subclass of the other have a generalization relationship [24]. As mentioned in Section 3.2, inheritance facilitates reuse since a new subclass that is created may reuse some of its behavior from its superclass. However, this also creates a dependency between the subclass and its superclass. When the superclass’ behavior or interface changes, this may break all of its subclasses. Some collections of classes, e.g. Java Software Development Kit (SDK), are frequently subclassed by many programmers in many software products and thus a small change in a frequently used superclass in such a framework have a great impact. This is sometimes referred to as the fragile base class problem [116].

The fragile base class problem can be described as either semantic or syntactic [166, pp. 102–104]. This is also referred as implementation modification and signature modification [52]. Semantic fragile base class problem regards the implementation of the superclass. When the superclass’ implementation is changed and this leads to changes in the class’ behavior, all subclasses’ behaviors will also change. When the implementation is changed, the subclass behavior will change, perhaps without any notification of this to the developer. This can lead to problems, if for example, the subclass’ users have a version of the superclass which is different from the one used when the subclass was developed.

Syntactic fragile base class problem is when the superclass interface is changed. When syntactic changes are made to the superclass, the subclasses may also need to be changed and recompiled. For example, in Java, if the name of a member variable that is declared in the superclass and used in the subclass is changed, the subclass must change all references to this superclass variable. Another example is when the superclass name itself is changed, then all subclasses must also change.

Not all syntactic changes to a superclass require recompilation of its subclasses. For example, in left part of Figure 3.6, a class C, that is a subclass to class B, that is a subclass to class A, have inherited a method, m1, that is defined in A and redefined in B. When the m1 method is called on instances of the class C the call will dynamically bind to the method defined in C’s superclass B.

In the next version of B, see right part of Figure 3.6, the method’s name is changed to m1b, but the calls to the method are not changed. All calls to m1 on C will now bind to the implementation of m1 in the topmost superclass A and not to the method named m1b in B which perhaps was the intention by the developer. This change will not require recompilation of C and thus changing the required calls to m1b in C in order to get a consistent program may be missed.

3.3.2 Aggregation

Aggregation is a structural relationship between two classes where one class is conceptually a part of the other class. For example, a Wall object may aggregate a Door object. In UML notation the aggregation relationship is drawn as a line
3.3. RELATIONSHIPS IN OBJECT-ORIENTED SOFTWARE

Figure 3.6: An example of the syntactic fragile base class problem. When the method name \texttt{m1} is changed in class \texttt{B}, all calls to \texttt{m1} in \texttt{C} will bind to \texttt{m1} in \texttt{A} instead of to the change method name \texttt{m1b} in \texttt{B}.

between two classes and annotated with a diamond, see Figure 3.7.

Figure 3.7: Aggregation relationship.

In programming, aggregation can be realized differently depending on the programming language and its features. At least three different types of semantics can be used to represent aggregations in programming, value, reference and pointer semantics.

**Value semantics.** With value semantics, items can be physically stored in the memory within other items. An example of this is Java’s primitive data types and C#’s \texttt{struct}.

Figure 3.8 shows an example of using the C#’s \texttt{struct} for implementing the relationship that a wall aggregates a door and the door aggregates the properties \texttt{height} and \texttt{width}. With value semantics, data are stored directly in the \texttt{struct} and no dynamic allocation of memory is required, see right part of Figure 3.8. This enhances performance compared to dynamic allocation of memory, especially when many items are created [76]. When using value semantics such as C#’s \texttt{struct}, items are copied when assigned to variables, passed as arguments or as return values. This means that there cannot exist any alias to a \texttt{struct} which may lead to side effects. This is illustrated in the example in Figure 3.9.
CHAPTER 3. BACKGROUND

Figure 3.8: Aggregation with value semantics.

Copying an item is slower compared to copying a reference or a pointer to the

Wall w = new Wall(); // A wall and a door is created. The door
// is physically stored in the door.
w.d.height = 2000; // The height of the door is 2000.
Door theDoor = w.d; // A copy of the door is created and
// assigned to theDoor.
theDoor.height = 2500; // The copy’s height is assigned 2500 but
// the door in the wall is still 2000.

Figure 3.9: A wall and a door created using value semantics.

item and thus using value semantics may have negative effect on performance if items are passed or assigned frequently in a program [76].

**Pointer and reference semantics.** With pointer and reference semantics the variable for an object is stored in one location in memory while the actual object is stored in another location. The variable is merely a handle to the object [171, pp. 137–140]. Pointer and reference semantics are similar in that both pointers and references contain memory addresses to where objects are stored. A difference lies in their possible use for the programmers. In for example C++, pointers are memory addresses that can be manipulated directly by the programmer. The value of a pointer variable may be changed to point to different memory locations using standard arithmetic operations such as addition and subtraction [111, pp. 88–93]. Similar to a pointer, a reference also con-
tains the memory address to where the object is stored, but the programmer is not allowed to manipulate the memory address directly. When the programmer uses the reference, it is dereferenced automatically and the actual object is manipulated instead of the memory address.

In this thesis references are used for discussing issues with handles to objects in general, and thus the term reference will be used for the rest in this thesis to denote this concept. However, since pointers are similar to references, much of the content of this thesis is applicable to both references and pointers. When issues are discussed that are applicable only to pointers this will be stated explicitly.

Objects in e.g. Java and C# are dynamically allocated and stored on the heap of its virtual machine [171, 76]. Member variables are stored within their objects but refer to objects stored in other locations in the memory, see Figure 3.10.

Hence, aggregating object by using references does not physically store objects within other objects as when using value semantics. Thus the example above, gives somewhat different results with references. In the example in Figure 3.11, the assignment Door theDoor = w.d creates an alias to the door in the wall. Both the variables theDoor and w.d can be used to manipulate the same Door object.

There is a risk of side effects using references. When references to an object are passed around, aliases to the referenced object are created and any of the aliases can be used to manipulate the object. This is a potential risk of side effects if the programmer is not aware of the consequences of using a reference.
Wall w = new Wall(); // A wall and a door is created
w.d = new Door(); // and stored in different locations.
w.d.height = 2000; // The height of the door is 2000.
Door theDoor = w.d; // The reference to the door in the wall
// is stored in theDoor.
theDoor.height = 2500; // The height of both theDoor and the door
// in the wall is now 2500.

Figure 3.11: The wall and the door using reference semantics.

Reference semantics require dereferencing each time a reference is used and this is slower compared to use value semantics. However, passing references as parameters or return values are faster since potentially large memory chunks do not need to be copied.

3.3.3 Use or Association

Use or association relationships are more general than aggregation relationship. Aggregation denotes that one object is a part of some other object, e.g. walls and doors. Association denotes any use of one class from another. Thus, aggregation is a specialization of the association relationship. However, in this thesis the association and aggregation are different with respect to how they are realized in programming. Aggregation relationship is implemented as a class with one or more member variables of other classes, e.g. a Wall class with a Door member variable. Association is realized as the use of a class which purpose is not inheritance or aggregation. For example, the use of a local variable declared in a method, parameters to methods, global variables, class members or return values.

The same techniques for transferring objects can be used for associations as for aggregation, i.e. value, pointer or reference semantics and thus the same issues also apply.

In run-time, association and aggregation may be viewed somewhat differently. Association may be seen as a short-term relationship where the values in local variables in methods and method parameters live only as long as the method is executing. Aliases created when e.g. references are passed as arguments to methods, cease to exist when the method returns. In contrast, aggregation relationships probably last longer in general, e.g. from the object is created together with the objects in member variables until the program terminates where all objects are destroyed.
Chapter 4

Research Results

4.1 Research Evolution

My research has evolved, from the initial research area in computer science with focus on programming languages, to software engineering research, focusing on methods and techniques such as software evolution and validation and verification, see Figure 4.1.

<table>
<thead>
<tr>
<th>CS: Programming languages</th>
<th>SE: Software evolution validation and verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytical: Programming language semantics</td>
<td>Empirical: Survey, case study experiment</td>
</tr>
<tr>
<td>Language theory, type systems, OO, discrete mathematics</td>
<td>Software development validation and verification, statistics</td>
</tr>
</tbody>
</table>

Figure 4.1: The research focus has evolved from the area of computer science to software engineering, and as a natural consequence of this, methods and literature used also shifted along with the research focus.

Programming languages and software engineering are related and have several connections. The work of implementation, testing and evolution all require programming languages in their activities of realizing their goals. As mentioned by Ryder et al.: "Software engineering and the design of programming languages enjoy a synergistic relationship, each influencing each other" [151].
When the research focus evolved, the research approaches used shifted as a natural consequence, from an analytical approach to empirical approaches. The same applies to the literature, which evolved from programming, programming language theory, type systems and discrete mathematics to software development, validation and verification, software evolution and statistics literature.

Dependencies has been a focal point throughout the research but has shifted somewhat in angle of approach depending on the purpose of each study. In the beginning (Studies 1-2), dependencies to an object’s internal representation or state was in center, focusing on the problems dependencies infer in software. Then, in the following study (Study 3), the number of dependencies in a program was measured to form the base for determining inspection order for classes and methods. Here, a view of dependencies inferring problems is also taken, but the focus is on problems with understanding and inspecting software having dependencies. In the next study (Study 4), dependencies between software test cases and the software they test were in focus, exploring the change impact on test suites. In this study the viewpoint is that dependencies cause problems since test cases are required to change when the software is changed, thus a more process oriented view. In the final studies (Studies 5-6), inherent dependencies between software parts were used to evaluate regression test selection techniques with the view that defects inferred by having dependencies should be found. In these studies, the view is that dependencies lead to defects in seemingly unrelated parts of the software and thus the set of test cases that should be selected to test the software can be optimized by focusing on these dependencies.

Software evolution is regarded as a research context throughout this work. However, its position varied depending on inherent properties of the studies and on the purpose of each study. A specific programming language may be used both for the initial development as well as during software evolution. Thus, a programming language concept is typically in itself not only applicable in a software evolution or an initial development context only, but may very well be used in both contexts. In the first studies (Studies 1-2), evolution was the context in that the programming language construct created should be used by the programmer during the initial development, to facilitate the evolution work later. Software inspections, as was studied in Study 3, may be used during both the initial development as well as in software evolution. However, software inspections rely on software comprehension which is a more important concept in evolution than in initial development to facilitate for conducting changes in code produced by others. The studies of test suite maintenance and regression test selection techniques (Studies 4-6), are focused solely on software evolution since maintenance of tests and regression testing are activities that normally occur in evolution after the initial development in the software’s life cycle, see Chapter 3.
4.2 CONTRIBUTIONS

The main contributions of this research are:

- A formal system for imposing access rights on relationships between objects in object-orientation programming. This system is shown to, in theory, reduce the effects of having dependencies between objects. The programming language construct could be useful in software production to control dependencies in object-oriented programs. If the programming construct is used when the software is initially developed it can be checked statically that references held in read-only variables will not be used to change the inner representation of the object pointed to by the reference. This relates to RQ1. [PAPER I].

- Knowledge of how relationships are used and their effects in software. Together with the theoretical contributions above, this gives a broader view of the effects of controlling dependencies in software. Situations where external objects are wrongfully dependent on the inner representation of other objects are not unusual to find in a software evolution context according to the software engineers asked. However, even though the situations rarely lead to defects in the software, tools and techniques for handling dependencies in object-oriented programs were still considered interesting. This is related to the research questions RQ2 and RQ3. [PAPER II].

- Contributions to the knowledge on effectiveness and usefulness of dependency focused reading techniques for software inspections. The dependency focused reading technique studied seems not to be more effective than reading techniques based on checklists. This is similar results as achieved in other studies on the subject. Dependencies in object-oriented software make the software harder to understand and reason about. However, this is not reflected in the defect detection process even though inspections are said to rely on understandability and comprehension of the software under inspection. Research question RQ4 regards the effectiveness and usefulness for reading techniques. [PAPER III].

- Contributions to the understanding of how dependency centered change strategies affect test cases during software evolution. A unit test case is strongly dependent on the unit it tests, and thus a change to the unit leads to ripple effects for the unit test suite. However, by using a change strategy that is focused on not changing existing code, but instead only add new code, the ripple effects can be reduced considerably without losing any functionality in the changed software. This relates to RQ5. [PAPER IV].

- Contributions to the body of empirical knowledge of regression test selection techniques. The Class firewall regression test selection technique is sensitive to the location of changes in the software under test. It also
selects more test cases than a more light-weighted, not dependency centered approach when used on a large complex software using scenario based system test cases. This relates to RQ6. [PAPER V AND VI].

- A change-based regression test selection technique that is less expensive to use and that selects fewer, or in worst case the same number of test cases, and find the same defects, as the Class firewall regression test selection technique. This is related to RQ6. [PAPER V AND VI].

Dependencies are inherent in object-oriented software but may also lead to problems. Some of the negative effects of having dependencies may be reduced by using programming language constructs for controlling relationships in software. This may lead to that software is achieved more effectively since some defects may be avoided. However, the number of defects related to dependencies caused by references that are exported from composite objects to external objects does not seem to be high, and thus the effects of controlling these dependencies may be limited.

Software may be achieved effectively by reducing the secondary work required from performing some task in software evolution. By using a dependency centered change strategy with the focus on not changing existing code, the secondary work of changing test cases due to system changes may be reduced, or even eliminated. However, a risk of using such a strategy is that the amount of other secondary work may be raised, such as the work of creating new test cases instead of changing the existing.

By effectively and efficiently identify defects in a changed system, software may be achieved more effective and more efficient. Defects in a changed system may be found by performing software inspections or by regression testing. The dependency focused reading technique for object-oriented source code studied in this research could not be shown to be more efficient or effective compared to the traditional checklist based technique. By using a selection technique for regression tests the number of test that is executed on a software may be reduced, leading to a more efficient software evolution process. By focusing on changed classes and classes dependent on the changed classes test cases may be selected. However, the dependency focused regression test selection technique studied in this research selected a larger test case set and thus was less efficient, compared to a technique focusing only on changed classes and their dynamic relationships to other classes.

By focusing on dependencies in object-oriented software different methods and techniques supporting software evolution have been studied to illuminate the research questions from different angles.
Bibliography


[146] Runeson, P. Using students as experiment subjects–an analysis on graduate and freshmen student data. In EASE’03: 7th International Conference on Evaluation and Assessment in Software Engineering (April 2003), Keele University, Staffordshire, UK.


Ph.D. theses

No 91-004  Olsson, Jan
An Architecture for Diagnostic Reasoning Based on Causal Models

No 93-008  Orci, Terttu
Temporal Reasoning and Data Bases

No 93-009  Eriksson, Lars-Henrik
Finitary Partial Definitions and General Logic

No 93-010  Johannesson, Paul
Schema Integration Schema Translation, and Interoperability in Federated Information Systems

No 93-018  Wangler, Benkt
Contributions to Functional Requirements Modelling

No 93-019  Boman, Magnus
A Logical Specification for Federated Information Systems

No 93-024  Rayner, Manny
Abductive Equivalential Translation and its Application to Natural-Language Database Interfacing

No 93-025  Idestam-Almquist, Peter
Generalization of Clauses

No 93-026  Aronsson, Martin
GCLA: The Design, Use, and Implementation of a Program Development

No 93-029  Boström, Henrik
Explanation-Based Transformation of Logic programs

No 94-001  Samuelsson, Christer
Fast Natural Language Parsing Using Explanation-Based Learning

No 94-003  Ekenberg, Love
Decision Support in Numerically Imprecise Domains

No 94-004  Kowalski, Stewart
IT Insecurity: A Multi-disciplinary Inquiry

No 94-007  Asker, Lars
Partial Explanations as a Basis for Learning

No 94-009  Kjellin, Harald
A Method for Acquiring and Refining Knowledge in Weak Theory Domains

No 94-011  Britts, Stefan
Object Database Design
No 94-014  Kilander, Fredrik  
Incremental Conceptual Clustering in an On-Line Application

No 95-019  Song, Wei  
Schema Integration: Principles, Methods and Applications

No 95-050  Johansson, Anna-Lena  
Logic Program Synthesis Using Schema Instantiation in an Interactive Environment

No 95-054  Stensmo, Magnus  
Adaptive Automated Diagnosis

No 96-004  Wærn, Annika  
Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction

No 96-006  Orsvärn, Klas  
Knowledge Modelling with Libraries of Task Decomposition Methods

No 96-008  Dalianis, Hercules  
Concise Natural Language Generation from Formal Specifications

No 96-009  Holm, Peter  
On the Design and Usage of Information Technology and the Structuring of Communication and Work

No 96-018  Höök, Kristina  
A Glass Box Approach to Adaptive Hypermedia

No 96-021  Yngström, Louise  
A Systemic-Holistic Approach to Academic Programmes in IT Security

No 97-005  Wohed, Rolf  
A Language for Enterprise and Information System Modelling

No 97-008  Gambäck, Björn  
Processing Swedish Sentences: A Unification-Based Grammar and Some Applications

No 97-010  Kapidzic Cicovic, Nada  
Extended Certificate Management System: Design and Protocols

No 97-011  Danielson, Mats  
Computational Decision Analysis

No 97-012  Wijkman, Pierre  
Contributions to Evolutionary Computation

No 97-017  Zhang, Ying  
Multi-Temporal Database Management with a Visual Query Interface

No 98-001  Essler, Ulf  
Analyzing Groupware Adoption: A Framework and Three Case Studies in Lotus Notes Deployment
No 98-008  Koistinen, Jari  
Contributions in Distributed Object Systems Engineering

No 99-009  Hakkarainen, Sari  
Dynamic Aspects and Semantic Enrichment in Schema Comparison

No 99-015  Magnusson, Christer  
Hedging Shareholder Value in an IT dependent Business society—the Framework BRITS

No 00-004  Verhagen, Henricus  
Norm Autonomous Agents

No 00-006  Wohed, Petia  
Schema Quality, Schema Enrichment, and Reuse in Information Systems Analysis

No 01-001  Hökenhammar, Peter  
Integrerad Beställningsprocess vid Datasystemutveckling

No 01-008  von Schéele, Fabian  
Controlling Time and Communication in Service Economy

No 01-015  Kajko-Mattsson, Mira  
Corrective Maintenance Maturity Model: Problem Management

No 01-019  Stirna, Janis  
The Influence of Intentional and Situational Factors on Enterprise Modelling Tool Acquisition in Organisations

No 01-020  Persson, Anne  
Enterprise Modelling in Practice: Situational Factors and their Influence on Adopting a Participative Approach

No 02-003  Sneiders, Eriks  
Automated Question Answering: Template-Based Approach

No 02-005  Eineborg, Martin  
Inductive Logic Programming for Part-of-Speech Tagging

No 02-006  Bider, Ilia  
State-Oriented Business Process Modelling: Principles, Theory and Practice

No 02-007  Malmberg, Åke  
Notations Supporting Knowledge Acquisition from Multiple Sources

No 02-012  Männikkö-Barbutiu, Sirkku  
SENIOR CYBORGS—About Appropriation of Personal Computers Among Some Swedish Elderly People

No 02-028  Brash, Danny  
Reuse in Information Systems Development: A Qualitative Inquiry

No 03-001  Svensson, Martin  
Designing, Defining and Evaluating Social Navigation
No 03-002  Espinoza, Fredrik
Individual Service Provisioning

No 03-004  Eriksson-Granskog, Agneta
General Metarules for Interactive Modular Construction of Natural Deduction Proofs

No 03-005  De Zoysa, T. Nandika Kasun
A Model of Security Architecture for Multi-Party Transactions

No 03-008  Tholander, Jakob
Constructing to Learn, Learning to Construct—Studies on Computational Tools for Learning

No 03-009  Karlgren, Klas
Mastering the Use of Gobbledygook—Studies on the Development of Expertise Through Exposure to Experienced Practitioners’ Deliberation on Authentic Problems

No 03-014  Kjellman, Arne
Constructive Systems Science—The Only Remaining Alternative?

No 03-015  Rydberg Fåhræus, Eva
A Triple Helix of Learning Processes—How to cultivate learning, communication and collaboration among distance-education learners

No 03-016  Zemke, Stefan
Data Mining for Prediction—Financial Series Case

No 04-002  Hulth, Anette
Combining Machine Learning and Natural Language Processing for Automatic Keyword Extraction

No 04-011  Jayaweera, Prasad M.
A Unified Framework for e-Commerce Systems Development: Business Process Patterns Perspective

No 04-013  Söderström, Eva
B2B Standards Implementation: Issues and Solutions

No 04-014  Backlund, Per
Development Process Knowledge Transfer through Method Adaptation, Implementation, and Use

No 05-003  Davies, Guy
Mapping and Integration of Schema Representations of Component Specifications

No 05-004  Jansson, Eva
Working Together when Being Apart—An Analysis of Distributed Collaborative Work through ICT from an Organizational and Psychosocial Perspective

No 05-007  Cöster, Rickard
Algorithms and Representations for Personalised Information Access
No 05-009  Ciobanu Morogan, Matei
Security System for Ad-hoc Wireless Networks based on Generic Secure Objects

No 05-010  Björck, Fredrik
Discovering Information Security Management

No 05-012  Brouwers, Lisa
Microsimulation Models for Disaster Policy Making

No 05-014  Näckros, Kjell
Visualising Security through Computer Games Investigating Game-Based Instruction in ICT Security: an Experimental approach

No 05-015  Bylund, Markus
A Design Rationale for Pervasive Computing

No 05-016  Strand, Mattias
External Data Incorporation into Data Warehouses

No 05-020  Casmir, Respickius
A Dynamic and Adaptive Information Security Awareness (DAISA) approach

No 05-021  Svensson, Harald
Developing Support for Agile and Plan-Driven Methods

No 05-022  Rudström, Åsa
Co-Construction of Hybrid Spaces

No 06-005  Lindgren, Tony
Methods of Solving Conflicts among Induced Rules

No 06-009  Wrigstad, Tobias
Owner-Based Alias Management