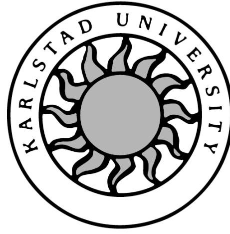

Department of Computer Science

Hanna Karlsson and Thomas Karlsson

Semantic Errors in ArgoUML – A Case Study of Semantic Integrity

Master's Thesis

2004:05

Semantic Errors in ArgoUML – A Case Study of Semantic Integrity

Hanna Karlsson and Thomas Karlsson

© 2004 The authors and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis that is not our own work has been identified and no material is included for which a degree has previously been conferred.

Hanna Karlsson

Thomas Karlsson

Approved, 2004-06-17

Opponent: Hans Hedbom

Advisor: Eivind J. Nordby

Examiner: Donald F. Ross

Abstract

This dissertation investigates the semantic integrity of an open source project called ArgoUML. In order to achieve this, a number of files are randomly selected and searched for methods that we believe may cause semantic errors. For each of the found methods a contract was constructed using reverse engineering. To evaluate the semantic integrity of ArgoUML, each client of said methods were examined in order to determine if the constructed contract was respected. The number of clients not respecting the contracts is the number of semantic errors found. If extrapolating the results of the case study to the whole of ArgoUML, the system would contain relatively few suppliers whose clients cause semantic errors. This result indicates that either ArgoUML contains few semantic errors or our method of investigation is unsatisfactory for detecting semantic errors.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Purpose 2

1.3 Goals 2

1.4 Delimitation 3

1.5 Disposition 3

2 Terminology **5**

2.1 The Concept of Semantics 6

2.2 The Basic Idea of a Contract 7

2.3 Design by Contracts 9

2.3.1 Semla – a software design method 10

2.4 The Concepts of Strong and Weak Contracts 11

2.4.1 The relation between contracts and errors 12

2.4.2 The relation between strong and weak contracts 12

2.4.3 Demanding and tolerant contracts 16

2.5 Definitions of semantic and logical errors 16

2.5.1 Semantic errors 17

2.5.2 Logical errors 18

2.6 Semantic Specification Levels 19

2.6.1 No semantics . . . 19
2.6.2 Intuitive semantics . . . 19
2.6.3 Structured semantics . . . 20
2.6.4 Executable semantics . . . 21
2.6.5 Formal semantics . . . 22

3 Discussions of Issues Related to the Dissertation 23
3.1 Similarity between a Complier and a Contract . . . 23
3.2 Defensive Programming . . . 24

4 ArgoUML 25
4.1 About ArgoUML . . . 25
4.2 Issues of Problem and Technical Domains . . . 26

5 Planning the Case Study 31
5.1 Outline of Case Study . . . 32
5.2 Feasibility Study . . . 33
5.3 Trivial and Non-trivial Methods . . . 33
5.4 Selecting Methods . . . 34
5.5 Constructing Contracts . . . 34
5.6 Evaluation of Contracts . . . 35

6 Results of the Case Study 37
6.1 Methods Used in the Case Study . . . 38
6.2 Comparison of Preconditions . . . 38
6.2.1 Methods with tolerant contracts . . . 39
6.2.2 Methods with demanding contracts . . . 39
6.3 Comparison of Postconditions . . . 40
6.3.1 Accessor-like methods . . . 40

6.3.2 Unstable methods 41
6.3.3 Methods deviating from intended use 42
6.4 Contract Violations 43
6.5 General Observations 45
6.6 Summary of Case Study Results 46

7 Conclusions **47**
7.1 Case Study Analysis 47
7.2 Problems 49
7.2.1 Problems with finding suitable methods 49
7.2.2 Problems concerning the construction of contracts 49
7.3 Conclusions of the Case Study 50
7.4 Future work 51

References **53**

A Acronyms **55**

B Additional Results from the Case Study **57**
B.1 Logical Errors in ArgoUML 58
B.1.1 ModuleLoader.activateModule() 58
B.1.2 ModuleLoader.loadModules() 59
B.1.3 OCLUtil.getInnerMostEnclosingNamespace() 59
B.1.4 ProjectBrowser.open() 60
B.1.5 ToDoItem.stillValid() 60
B.1.6 Translator.loadImageBindings() 60
B.2 Inconsistencies in Existing Documentation 61
B.3 Dubious Exception Handling 62
B.4 Further Development of ArgoUML 63

C Excerpts of ArgoUML Source Code **65**

C.1 CrWrongDepEnds.computeOffenders() 65

C.2 GoSummaryToInheritance.getChildren() 68

C.3 ModuleLoader.activateModule() 70

C.4 ModuleLaoder.loadModules() 70

C.5 OCLUtil.getInnerMostEnclosingNamespace() 72

C.6 ProjectBrowser.open() 73

C.7 TableModelNodeByProps.rowObjectsFor() 73

C.8 ToDoItem.stillValid() 74

C.9 Translator.loadImageBindings() 75

List of Figures

2.1 Relation Between Client, Supplier and Contract 8
2.2 Relation between Strong (A) and Weak (B) Contracts 13
2.3 Example of a Strong Contract 14
2.4 Example of a Weak Contract 15
2.5 Example of Intuitive Semantics 20
2.6 Example of Structured Semantics 21
2.7 Example of Executable Semantics 22

4.1 Problems with Different Domains 27
4.2 The Difficulty of Expressing Contracts 28

List of Tables

6.1 Examined Non-trivial Methods, and their Documentation Levels 38
6.2 Methods with Tolerant Contracts 39
6.3 Methods with Demanding Contracts 40
6.4 Postconditions of the Accessor-like Methods 41
6.5 Postconditions of the Unstable Methods 42
6.6 Methods with Deviations between Semantics and Implementation 43
6.7 Results from Contract Evaluation of Non-trivial Methods and their Clients 44
6.8 Semantic Specification Levels of Classes and Methods in the Case Study - Values 45
6.9 Semantic Specification Levels of Classes and Methods in the Case Study - Percentages 45

Chapter 1

Introduction

Quality is an important factor in today's software engineering projects. There are many ways to improve quality by controlling the development process. Examples of processes used today are eXtreme Programming (XP), Rational Unified Process (RUP), and Test-Driven Development (TDD) among many others. Since there are many factors that affect good software, processes differ in how they approach the problem of ensuring quality. Some processes focus entirely on automated tests and other on documenting semantics.

1.1 Background

The members of the research group SERG (Software Engineering Research Group) at the University of Karlstad study semantics, contracts and semantic integrity. The research group handles the problem of describing semantics with contracts. An important question is whether contracts enhance understanding significantly enough to be motivated in commercial software development projects. This case study tries to give an indication of this by examining the semantic integrity of the code in a real-world, not contract-based, development project.

In order to give the dissertation's results more impact, another similar case study has

been made by another group. The sister project's case study is essentially a replication of the methods this dissertation uses, but on another software project.

1.2 Purpose

As a step towards learning whether or not contracts would improve the level of quality in software, a case study of an open source project is presented in this dissertation.

The case study examines the code for a CASE (Computer Aided Software Engineering) tool called ArgoUML, see Section 4.1. ArgoUML is an open source program to make UML (Unified Modelling Language) diagrams during the analysis and design phases in software development. Automated tests are used in ArgoUML, instead of contracts, to ensure a certain degree of correctness. There is no particular reason why the case study is based on ArgoUML. It seemed to be a suitable open source project, which we came across.

The goal of this dissertation is to determine if the semantic integrity is preserved within ArgoUML's code. The answer gives a small indication if contracts should be used or if the quality of the software is good enough as it is. If the overall quality is good then contracts may not need to be used at all, due to the extra cost of writing the contracts.

1.3 Goals

The goal of this dissertation is to measure the number of semantic errors that exist in the latest stable version of ArgoUML. These errors would indicate breaches in the semantic integrity of the project. The errors are believed to stem from seemingly legal operations within the code, resulting in abnormal program states that are currently not detected by the automated testing used in ArgoUML. The errors are a result of inconsistent semantics. As further explained in Chapter 2, a reason for this may be that client modules rely directly on their suppliers' implementations and not the suppliers' semantics.

1.4 Delimitation

This case study has the semantic integrity, expressed by contracts, as its focus, and other issues are less relevant. The people working with ArgoUML have had many people commenting upon the design of the program and that some parts can be restructured to be more adaptable. Therefore, the case study does not attempt to change or comment on the design. The semantic errors are found by constructing contracts and evaluating them. Other issues, such as logical errors or documentational inconsistencies, are excluded from the main result.

1.5 Disposition

In Chapter 2, the key phrases and concepts encountered in the dissertation are explained. The most important concept is semantics and contracts so the chapter presents these in detail. Other supporting concepts also included are Design by Contract, the concept of weak and strong contracts, semantic integrity, semantic errors, logical errors and semantic specification levels. Most of the reasoning in the dissertation is based on these subjects, which is why they are presented as early as possible.

The next chapter includes discussions about concepts related to this dissertation, although these subjects are not in focus in this dissertation. The point of using contract in software is often questioned. The first discussion sketches an analogy between using compilers for detecting syntactic errors and using contracts for detecting semantic ones. Defensive programming is briefly mentioned since it is a commonly known programming style, and is also used in ArgoUML.

Chapter 4 presents information about ArgoUML and its history, giving some information about the software project used as a base for the case study. The chapter ends with a discussion about a problem encountered in ArgoUML. The problem arises when the design, implementation, and programming language denote different meaning to the same

terminology.

Chapter 5 focuses on how to carry out the case study. The chapter starts by presenting a feasibility study for the case study. The rest of Chapter 5 contains details on how to conduct the case study. This includes how to find suitable methods, how to construct contracts, and how to evaluate said contracts.

Chapter 6 presents the results of the case study. Apart from comparing and discussing the contracts of the methods examined in the case study, and presenting the results, some general observations about the case study are also included.

Chapter 7 presents the discussions and conclusions of the results presented in Chapter 6. Also included are a few problems encountered during the case study, and a section presenting a few ideas of future work.

Appendix A is a list of the acronyms used in the dissertation. Appendix B contains information about findings during the case study that, although interesting, are beyond the focus of the case study. The final appendix contains the source code of the methods thoroughly examined in the case study. Also included are the constructed contracts for each method. The statistical data gathered during the case study is available on the CD accompanying the dissertation. Also included on the CD is the random sample of ArgoUML source files, used in the case study.

Chapter 2

Terminology

The focus of this chapter is semantics. The reason is that semantics are an important issue in software development. Since this dissertation is an examination of the potential semantic errors in a software system, semantics are an important subject in this dissertation as well. A vital issue concerning semantics is documentation, since documenting semantics helps programmers avoid simple software errors. The discussion of the importance of documenting semantics leads to a number of descriptions on how to document it. The approaches of documenting semantics described in this dissertation are centred on the concept of contracts. Thus, contracts are first described briefly, before two approaches of documenting and handling semantics are described. These two approaches are called Design by Contract [8] and Semla [3]. Design by Contracts is presented since it is the most commonly known approach of handling semantics and since it is a predecessor of Semla. Semla is presented since this is the method used to handle semantics in the case study of the dissertation. Strong and weak contracts, semantic integrity, semantic and logical errors, as well as a description of different semantic specification levels are then introduced as a preparation for discussions in later chapters.

2.1 The Concept of Semantics

The concept of semantics has different definitions in computer science depending on its function in a particular area. A broad definition is that semantic is the meaning, as opposed to syntax which is the appearance, of a piece of code. A more formal description of semantics is to define it as the meaning, purpose or intention behind code. A software system contains several levels of abstraction, and therefore contains several levels of semantics. The lowest level of semantics lies within the programming language itself. This level is outside the scope of this dissertation, which instead is concerned with the semantic levels in the application software. The focus is on the semantics of classes and their methods, i.e. the meaning of classes and methods.

The main reason for arguing that documenting semantics is an important issue is that if the semantics of code is not clearly documented, part of the semantics could be lost. The loss of semantics becomes a problem when the knowledge is needed again, such as in the case of reuse, correction or extension. The documentation should take place at the time of creation of the code because that is when the programmer has a complete understanding of its meaning and purpose.

When working with another programmer's code within a system, it is important to use it the way its creator intended. Only then can the semantic consistency within the system be preserved. However, when the semantic information is missing, the meaning of the code is ambiguous. Any programmer unfamiliar with the code must make assumptions about its intention based on the code alone. Those assumptions can be different from the intentions of the original author. Since the views of what the code means differ between programmers, changes made to the code may introduce inconsistencies in the software system. These inconsistencies could result in faults that can be hard to track down and correct once they have been introduced.

A possible solution to the problem of capturing the semantics information in documentation has been advocated by, among others, Bertrand Meyer [8]. According to his

solution, the documentation of the code should be kept in the code either as comments or as executable contracts. By keeping the documentation close to the code, it is easier for the programmer to keep the documentation updated since he or she only has to update a single source of information. A good example of this is the tool for documentation used in Sun's programming language Java, the JavaDoc [6]. When using JavaDoc, programmers write the documentation as comments in the code itself. The documentation can later automatically be extracted in a HTML-version.

Semantic information should also be part of the close-to-code documentation according to Meyer, and is captured in contracts and invariants. The contracts are an abstraction of the meaning of the code and thus provide part of the documentation. This abstraction focuses on the semantics and facilitates understanding of a program. The reason being, that programmers or other personal do not have to read the actual code. Methods for forming contracts and other related issues are discussed in the following sections.

2.2 The Basic Idea of a Contract

As mentioned previously, contracts are used to describe the meaning of the code, i.e. the meaning of an implementation. When programmers do not use an abstraction, like contracts, to document semantics, software modules may depend directly on each other's specific implementation and not on the semantics of the implementation. Software modules using other modules will subsequently be referred to as *clients*, and software modules being used will be referred to as *suppliers*. When depending directly on the supplier's implementation, the client is tightly coupled with that implementation. Consequently, when the supplier's implementation changes the client may be forced to change its implementation as well. This means that what first may appear as relatively small changes in a few supplier modules can affect additional parts of the software system. The worst situation would be for changes in a supplier to cause faults in clients of that supplier. The tightly

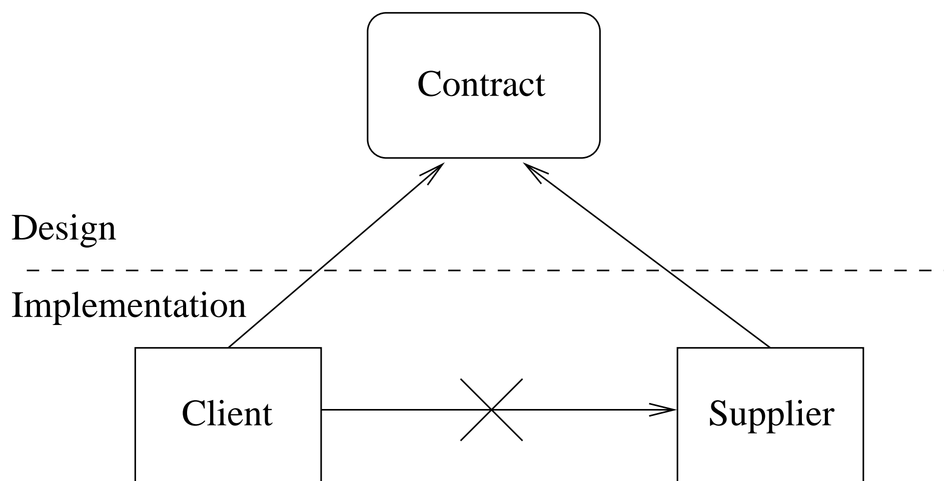

Figure 2.1: Relation Between Client, Supplier and Contract

coupled relation is in Figure 2.1 shown as the arrow directly connecting the client to the supplier.

To resolve the problem of the tight coupling between a supplier and its clients, an intermediary abstraction, called a *contract*, can be used. When this is used, both the client and the supplier must conform to the terms of the contract. This is shown in Figure 2.1 as the arrow from the client to the contract, and the arrow from the supplier to the contract. As a result, the client is unaffected by changes in the supplier's implementation, as long as these changes are within the contract. Only changes in the contract forces the client to change. The use of contracts helps decouple the semantics from the implementation. Constructing the contracts also makes programmers more aware of the design and the semantics of their code instead of only looking at the design at code level. The different levels of semantic awareness is distinguished by the dotted line in Figure 2.1. This way contracts can give better documentation of semantics and the intentions behind the code

and design. Contracts are further explored in the remaining sections of this chapter.

This dissertation attempts to form contracts by reverse engineering parts of the source code of ArgoUML, described in Chapter 5. As a result of this backward approach, these contracts are directly dependant on the code.

2.3 Design by Contracts

In his book Object-Oriented Software Construction [8], Bertrand Meyer presents a method for designing and implementing object-oriented software called Design by Contract (DbC). Included in this method is his proposal on how to handle the semantics within software. Basically, Meyer identifies the caller of a method as the client and the method being called the supplier. He proposes that there should be a contract between the two. The supplier can set up a number of prerequisites, called preconditions, which the client must fulfil before calling the supplier. The supplier also assures that a number of conditions, called postconditions, are guaranteed to be met if and only if the client fulfils the stated prerequisites expressed by the preconditions. If the client does not meet the preconditions, the supplier has no obligation to meet the postconditions, since the contract is already broken.

A contract clearly states the responsibilities of each partaker. As long as each partaker upholds their end of the contract, they can benefit from the guarantees of the contract. The clear definition in the contract of who is supposed to do what can result in reduced testing. The reason for this is that the client need only check the parameters or other circumstances that are part of the suppliers precondition once. The supplier always assumes that everything is in accordance with its contract, i.e. that its preconditions are fulfilled. Likewise testing is reduced for the client, who assumes that the postconditions of the supplier holds true. In other words, no values returned from the supplier need to be tested as long as the supplier's prerequisites are fulfilled. Meyer states that DbC ensures greater

correctness in software by less testing by decreasing the number of tests and the overall complexity of the code.

Pre- and postconditions are examples of a more general concept called assertions. Assertions are used to describe properties of classes. Pre- and postconditions are used primarily for defining contracts for methods, but there is also another type of assertion that is used with classes, for example invariants. An invariant states the conditions for the properties of a class that must hold true at all stable times. The invariants are established at the creation of an object and then maintained by the executing methods. During the execution of a method, a method may violate invariants of the class if this is necessary, if the method re-establishes the invariant before the end of the method or before another method call is made. This implies that invariants of a class are part of the pre- and postconditions of all methods of that class.

DbC states that the checks of all contracts should be executable so that the compiler can automatically test the correctness of each contract and invariant. These automatic tests are then used during the development process to immediately pinpoint the exact location of a broken contract or invariant by halting the execution of the program if an error occurs. This option, which may be controversial, is further discussed in Section 3.1. It allows the broken contract to be corrected during the development phase instead of during the test phase of a project. Meyer argues that automatically testable conditions are more valuable than those that are not, since the testable ones help detect faulty code. There are other views on this subject, which is presented in the following section.

2.3.1 Semla – a software design method

Semla[3] is a design method, developed by SERG at Karlstad University, with a view of contracts that is an adaptation of DbC. The main differences are that Semla does not assume automated testing and executable semantic formalism. The developers of Semla advocate that using contracts to improve software quality is useful and powerful, since it

ensures greater knowledge of the semantics of the code. Semla gathers a number of good and useful design principles and its focus is on semantics and contracts. The reason for developing Semla was to compose established academic rules of design into a format that can easily be absorbed by the software industry. A drawback that hinders the spread of DbC is that DbC is developed for the programming language Eiffel. Because of this, it is difficult to use DbC unless you are using Eiffel as well. The reason is that the principle of using assertions in DbC, which is essential according to Meyer, is a built in feature of Eiffel.

Semla is an adaptation of DbC that can be used with any programming language. A difference is that Semla does not assume automated testing during development. Instead, Semla relies on the programmer's increased semantic awareness that comes both from forming the contracts and from the documenting effect of the contracts. When Meyer argues that each contract should be testable, Semla states that a contract or invariant expressed in plain text can be equally valuable for providing documentation of the semantics of a method or class. Using contracts in the way Semla advocates can easily be added to any coding conventions and thus integrated in the development process in any language.

2.4 The Concepts of Strong and Weak Contracts

The concept of strong and weak contracts has been developed in the research group SERG at Karlstad University [9]. The purpose of introducing the concept of strong and weak contracts here is to serve as the basis for defining terminology for a broad categorisation of the contracts used in the case study. This dissertation uses the terms demanding and tolerant to refer to two cases of contracts conforming to the concept of strong and weak contracts. The terms are used for easy categorising of the types of contracts encountered in the case study. First internal and external errors are described to emphasize the difference between strong and weak contracts and their uses. This is followed by a subsection that

gives the formal definition of strong and weak contracts illustrated with examples. The section ends with a clarification of the terms demanding and tolerant contracts, which are used later in the dissertation.

2.4.1 The relation between contracts and errors

When designing a software system it is important to make a distinction between external and internal errors in order to determine how to handle respective error type. External errors arise from sources outside the system and internal errors arise from the development team during the development of the system.

For a system to be robust and easy to use, it should have a way of dealing with external errors. The reason for making the system deal with this type of errors is that it is hard to prevent the errors because external agents impose them on the software. Typical examples of external errors are incorrect input from human end users, hardware and network failures or arithmetic overflow interrupts from the operating system.

As opposed to external errors, programmers should not handle internal errors; instead, contracts should deal with them. Contract should do this by, for example, demanding correct input. A method to calculate the square root may for example demand only non-negative numbers. The reason for handling internal errors with contracts is that internal errors may cause the system to fail even when using it correctly. Typical examples of causes for internal errors are design and programming errors. Such errors are possibly caused by ambiguous semantics as previously mentioned in Section 2.1. Obviously, it is much better to build software in which internal errors, if they are introduced, are easy to detect and remove.

2.4.2 The relation between strong and weak contracts

The formal definition of the relation between strong and weak contracts [9] is:

> A redefined contract is weaker than the original one if its precondition is equal to or weaker than the original precondition and its postcondition is equal to or stronger than the original postcondition in the domain of the original contract.

This relation is illustrated in Figure 2.2. There the contract of A is redefined as the contract of B. The new contract for B is, as stated in the definition, weaker than the original one.

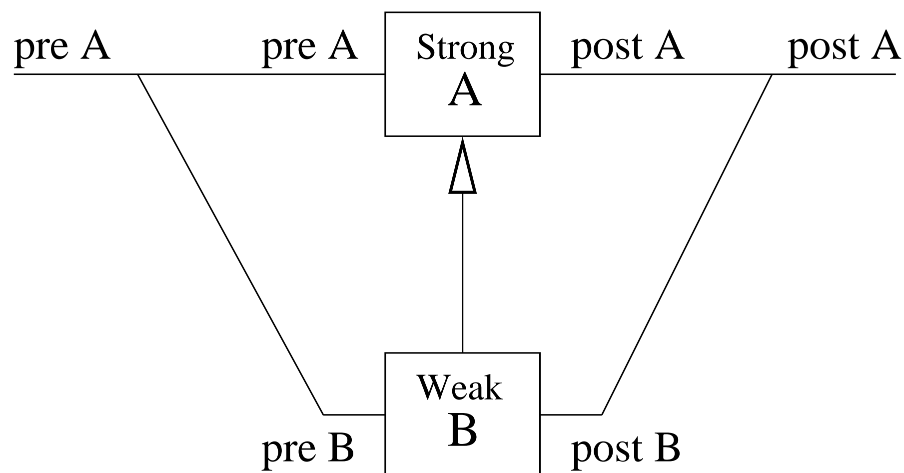

Figure 2.2: Relation between Strong (A) and Weak (B) Contracts

Using the distinction between external and internal errors, introduced in the previous section, it is possible to device a contract suitable for dealing with each type of error. The internal errors that occur during the development of a software system are conveniently handled with the more demanding approach of strong contracts. This type of contract has been the one in mind in all the previous discussions in the dissertation, since it is what the term contract usually means. The precondition states all the vital requirements for legal use, and the postcondition states the result for when the preconditions are met, as

```
/** Description: The method returns the
* non-negative square root of parameter
* x.
* Pre: x >= 0
* Post: result == the non-negative
* square root of x */
public Double squareroot(Double x)
{ ... }
```

Figure 2.3: Example of a Strong Contract

shown by the example in Figure 2.3. It shows a method that computes the square root of a number sent to the method. It is only useable for non-negative input values.

With a strong contract, a client programmer cannot make the client code work unless he or she meets the preconditions in the contract. In the example in Figure 2.3, clients should have no problem fulfilling the precondition. Violations of the precondition, i.e. internal errors, can be found by examining all calls to the supplier. The client programmer could then easily remove the errors by making sure that all the calls fulfil the precondition of the supplier. This consequence of the demanding nature of strong contracts could be an efficient way of decreasing internal errors in software. Therefore, it is generally recommended to use the strong type of contract at all possible times. One exception, however, would be for handling external errors.

As stated in subsection 2.4.1, software systems need to handle external errors in order to be robust. Since this is the opposite of the purpose of strong contracts another type of contracts are needed. A more suitable solution is to use a more tolerant kind of contract, a weaker contract.

```
/** Description: The method returns the
* non-negative square root of parameter
* x or an error code.
* Pre: True
* Post: if x >= 0 result == the non-
* negative square root of x else
* result == -1 */
public Double squareroot(Double x)
{...}
```

Figure 2.4: Example of a Weak Contract

Although it is possible to define a strong contract for dealing with an external error source, the causes of external errors lie beyond the control of a software system. This means that programmers cannot ensure that such a precondition is not violated. Thereby, clients cannot be expected to meet such a precondition, and therefore the contract is useless. The solution, as mentioned, is a weaker contract.

The example in Figure 2.3 should work fine with any part of a software system that must respect its contracts. However, if a human user used it directly, this user, even though aware of the precondition, could enter an illegal value. If common mistakes like that caused a failure, the software would not be considered user friendly, but instead hard to use. A solution is to weaken the original strong contract into the more tolerant variant shown in Figure 2.4.

The precondition in the example in Figure 2.4 demands less, but the client must also devise a way to deal with the additional case of a returned error code. If the weak contract is used for user input, the error code could be used to notify the user of the erroneous input and ask the user to try again. A suitable use of tolerant contracts is to deal with external errors. External errors can be handled by either using tolerant contracts or by weakening

existing demanding contracts. The weakening of a contract is invisible to clients already conforming to a strong contract. New clients can choose to conform to either the original contract or the weaker contract, which gives them more freedom.

2.4.3 Demanding and tolerant contracts

Although the concepts of strong and weak contracts have been defined [9], they have not yet spread to the general discussion of contracts in the world of Computer Science. When discussing contracts, people tend to mean strong contracts. Weaker contracts are usually not considered contracts. However, even if a contract is weak, the method bound by the contract is no freer to do as it pleases than a method bound by a strong. A method must always behave according to its contract. Since the terms strong and weak expresses a relationship between two contracts, this dissertation uses other terms to describe different cases of the concept of strong and weak contracts. The term *demanding* will from now on refer to the strong type of contracts. That is, a contract that makes demands on the client, i.e. the contract contains one or more preconditions. Moreover, the term *tolerant* will from now on refer to the weakest type of contracts. The weakest type of contracts generally has no preconditions i.e. it makes no demands, this can be written as **true**, but is usually omitted.

2.5 Definitions of semantic and logical errors

As stated in section 1.3, the goal of this dissertation is to find a measurement of the semantic errors in ArgoUML's source code. The term semantic error is therefore defined here, together with semantic integrity. Apart from the semantic errors sought, another type of error is identified in the case study. This is called logical errors in this dissertation. Since the logical errors are treated differently than the semantic errors in the dissertation, different terms are necessary in order to distinguish them. The reason that they are treated

differently is that the semantic errors are part of the main goal of the dissertation whereas the logical errors are part of a sidetrack discussed in appendix B.

2.5.1 Semantic errors

In order to explain semantic errors the term semantic integrity needs to be defined. The term semantic integrity is defined in [4] as:

> The *semantic integrity* of a software system is the degree to which its semantic properties are preserved. The term can be explained by stating that each part of the system should respect the intended purpose of all other parts.

Semantic errors arise when the semantic integrity of code is violated. Invariants and contracts express semantic integrity constraints. In other words, *semantic errors* arise when programmers violate invariants and pre– or postconditions. The violations that are easily detected are the ones concerning invariants and preconditions. Fulfilling the postcondition is, as stated previously, the responsibility of the supplier, but the client depends on it. Depending on the type of contract binding the supplier, a tolerant or a demanding contract, the client may have to deal with multiple results from the supplier.

Section 2.4.2 briefly mentions the issue of clients handling error codes from contracts with multiple clauses. The present section is an elaboration of the subject. In order to maintain the semantic integrity of the system, clients need to do more than simply meet a supplier's precondition. A client must also handle the result from the supplier, i.e. all clauses in the supplier's postcondition, in a meaningful way. That is, the continuing execution of the client must be based on the result of the call to the supplier, whatever the result is. This is especially important when the postconditions have multiple clauses.

To the authors' experience, most demanding contracts have only one clause in their postcondition. The reason is that the demanding precondition contracts narrows down the other possible outcome clauses. Many tolerant contracts, which have no precondition

instead, end up with multiple result clauses. When a client uses a supplier with a single result clause, assuming the precondition, the client only has to deal with that one result. This means that the client only needs to build on that single result in order to preserve the semantic integrity of the contract. However, when a client uses a supplier with multiple clauses in the postcondition, the client must be prepared to deal with each clause in a meaningful way. This means that the client must build a meaningful continuation on each result in order to preserve the semantic integrity of the contract. Anything else will result in a semantic error.

There is no clear definition of what a meaningful way is, since this depends on the context of the client. Therefore, when working within an unfamiliar system, even determining a meaningful way of handling a single result, may be difficult, especially when determining the implications of error codes or exceptions.

2.5.2 Logical errors

Logical errors are usually programming errors. They have different appearances, but generally, logical errors can be described as code snippets that seem to be in the wrong place, or seems not to do what they are meant to. The reason for their existence may be a simple programming error. Examples of this is a programmer accidentally writing **true** instead of **false**, forgetting to add break conditions, and similar. It is also a possibility that these logical errors are residues from restructuring of the code, leftovers accidentally remaining within the code. The logical errors are small and cannot typically be detected by a compiler since the code, despite containing small logical errors, still adheres to the syntax of the programming language. The errors may not be apparent until the semantics, the intended use, of the method containing the error is analysed.

2.6 Semantic Specification Levels

This section presents the different levels of semantic formalism for semantic specification that is used in this case study. The distinctions between different levels of semantic formalism serve as the base of the classification system developed in the case study described in Chapter 5. A literature survey [2] revealed five distinct levels. The naming of the levels changed slightly in a later publication [4] and this dissertation uses the latest names. The levels, in increasing order of formalism, are "no semantics", "intuitive semantics", "structured semantics", "executable semantics", and "formal semantics". For each of the levels, a short description is given in the following subsections.

2.6.1 No semantics

No semantics correspond to specifications that hold no explicit semantic information at all. The reason for this is that either no documentation at all is present or that it is purely syntactic in nature. The only help a client programmer receives is from the method and parameter names. Everything else concerning the purpose of the method is up to the programmer to guess. How well the programmer guesses depends on his or her experience and intuition when interpreting code.

2.6.2 Intuitive semantics

When a section of code is described with unstructured plain text the semantics is considered to be intuitive. It is stated in [4], that a large portion of the code from non-critical software projects falls into this category of documentation. In other words, the semantics is mentioned, but only in an intuitive way without apparent structure or consistency. This is illustrated in Figure 2.5.

Because of the lack of structure in the documentation, the information is scattered and the programmer has to read the full description. A problem with this approach is

```
/** Desc: This method removes the top
* element from the stack-object if the
* stack is not empty. If the stack is
* empty, a StackException will be
* thrown. The size of the the stack
* is decreased by one. */
public void pop() throws StackException
{ ... }
```

Figure 2.5: Example of Intuitive Semantics

the difficulty to distinguish the conditions for using the method. The precondition in the description above is not apparent at first.

2.6.3 Structured semantics

The structured level of semantic formalism increases the readability by introducing structure in the description. The descriptions that fall into this category have structure, but they follow no particular syntax or formalism. This means that they may be written in plain text. The code description in Figure 2.6 emphasizes this point.

Structured descriptions are more readable than intuitive ones. In an intuitive documentation, the programmer needs to read the whole description to understand the contract for a particular method. With a structured documentation, on the other hand, there is no need to read all of the information, because it is divided into easily recognisable blocks. The plain text in contracts of this type still has to be reinterpreted into code before they can be used. An example of this is for checking whether or not a precondition holds before a method call.

One possible drawback of using this level of semantics is that the plain text in the

```
/** Desc: This method removes the top
* element from the stack.
* Pre: The stack is not empty.
* Post: The top element has been
* removed. The size of the stack has
* decreased by one. */
public void pop()
{ ... }
```

Figure 2.6: Example of Structured Semantics

contract may need to be re-interpreted into code before it can be used. For example to check whether or not a precondition holds true before a method call. The next section presents a solution to this issue.

2.6.4 Executable semantics

In executable semantics, the pre- and postconditions are written in code. The code used in these circumstances may differ slightly from the target language used. These constructs are mainly used to express conditions and values that may have changed. An example is a value that existed during the start of a method and is then used as a reference when describing the method's postcondition. This will be used and explained in the example in Figure 2.7.

Contracts written in code make the pre- and postcondition more easily checkable during run time. The contracts advocated in Meyer's DbC [8] use executable semantics. Contracts conforming to Semla may also fall into this category. The example from the previous sections is duplicated below, but is documented with this increased level of formalism. Observe in Figure 2.7 the construct "size() @ pre" which means the return value of the

```
/** Desc: This method removes the top
* element of the stack.
* Pre: !isEmpty()
* Post: size() == size()@pre - 1
public void pop()
{ ... }
```

Figure 2.7: Example of Executable Semantics

method size() before the call to pop(). These constructs are from the Object Constraint Language (OCL) [12].

The method isEmpty() used in Figure 2.7 should be used to check if the precondition holds before the use of the method pop(). This means that isEmpty() must be a usable predicate in the class for this precondition to be valid. Similarly, size() must be a predicate in the class for the postcondition to be valid.

2.6.5 Formal semantics

This section briefly mentions what formal semantics is, even though it is outside the scope of this dissertation. None of the class and method descriptions that are examined in this dissertation uses this level of formalism.

Descriptions in formal semantics use mathematics to prove the consistency of methods, usually by denotational, operational, or axiomatic semantics [13]. It is exhausting to prove that the semantics of a method is correct; therefore, not many programmers choose to do so.

Chapter 3

Discussions of Issues Related to the Dissertation

This chapter contains discussions of subjects related to the dissertation. The first section discusses the usefulness of the similar behaviour of a compiler and demanding contracts. The following section discusses the concept of defensive programming, since this style is used in ArgoUML and therefore has influenced the work of the case study.

3.1 Similarity between a Complier and a Contract

Before the use of advanced programming language compilers, programmers had to check the syntax of their source code by hand. Nowadays, the compiler does this. If a compiler detects syntactical errors or type checking errors in the source code, the compiler will not create an executable version of the program until the errors are fixed. The programmer has no choice but to correct the errors detected by the compiler. This ensures that when the source code is finally compiled, it is free from syntactic errors and checked type errors.

Languages, compilers, and programs have all grown more complex with the evolution of computer science. As stated previously in section 2.1, when the complexity of a system

increases so does the importance of semantics. Compilers can detect and thereby ensure removal of syntactic and type related errors, but there is no similar method for semantics. A possible solution could be contracts.

Since compilers will not allow any syntactic or type related errors in the code, contracts should not allow any semantic errors in the code. By halting the compilation of the source code upon the detection of errors, compilers force programmers to remove said errors. Similarly, contracts could halt the execution of the executable program upon detection of semantic errors to force programmers to remove those errors as well. This would be a very demanding use of contracts. Meticulous compilers are a great help to find syntactic errors. Having the same help with semantic errors would be as useful for programmers. The idea is the same as the one behind Design by Contract, discussed in section 2.3.

3.2 Defensive Programming

Defensive programming, or defensive development [5], is an approach to programming that focuses on robustness. To achieve robustness, defensive programming places all responsibilities for ensuring correctness in the supplier's hands. Defensive programming also uses assertions, in other words preconditions, postconditions and invariants. The big difference to DbC though, is that defensive programming only uses tolerant contracts. When something happens in the supplier that it cannot handle, the supplier notifies the client by using exceptions. DbC usually do not use exceptions, because it assumes that the (demanding) preconditions are met before method execution.

Chapter 4

ArgoUML

This chapter briefly describes ArgoUML and matters concerning it. The first section presents some information about ArgoUML. The second, and final, section discusses a problem encountered during the examination of ArgoUML's source code. This problem relates to using different vocabulary domains that exist within software projects.

4.1 About ArgoUML

ArgoUML [1, 10] is a CASE tool for use in Object-Oriented Analysis and Design (OOAD). It helps designers by having a diagrammatic representation of their designs in a graphical user interface. ArgoUML uses, as the name implies, UML for the representation of the diagrams. ArgoUML also fully incorporates the UML 1.3 specification [11] and its meta-model. ArgoUML is written in Java and is supported by Java 1.3 [6] or later versions. The program is currently developed as an open source project.

The unique factor of ArgoUML is its critics. The critics are a cognitive system that monitors the design and the diagrams. When the critics find an error or something that could be done in a better way, it comments on this. The comments can be found as todo-items in the program. This makes it easy for the designers to find and possibly correct

them later. The system of critics is the heart of ArgoUML and was initially developed by Jason Robbins [10]. He designed and wrote almost everything in the program until version 0.7. This corresponds to the first 100,000 lines of code. ArgoUML has been distributed since July 1998. The current stable version is 0.14.1. It is this version that is under scrutiny in this case study.

4.2 Issues of Problem and Technical Domains

This section handles a problem that arises when denoting different aspects with the same term. To explain this problem an example follows:

In an accounting system the difference between the terms account and Account-class is quite obvious. An account is something customers use in the accounting system. An Account-class is an implementation of the account concept in the accounting system. This means that the Account-class models the behaviour of an account. That the Account-class has the same name as account is not a coincidence. This is standard general practice in OOAD.

ArgoUML works in a similar way. ArgoUML uses UML to model its designs, but at the same time uses an implementation of UML. ArgoUML uses a naming convention so the programmers do not confuse the terminology from UML and the implementation of UML. All the terms in the implementation have the same name as in the UML specification [11], but is prefixed by a capital m. For example, the programmers have named the UML term Class MClass in the implementation.

So far, the analogy with the accounting system holds for ArgoUML, but there is another aspect to consider in the ArgoUML case. UML uses terminology like class, object, and inheritance. OOP uses identical terms, but with a slight difference in meaning. When a programmer writes about inheritance, does he or she mean inheritance in UML or in the programming language? The discussion about the above problem continues further down

in this section. First, two terms need to be presented to make the discussion easier to follow. The terms are problem domain and technical domain.

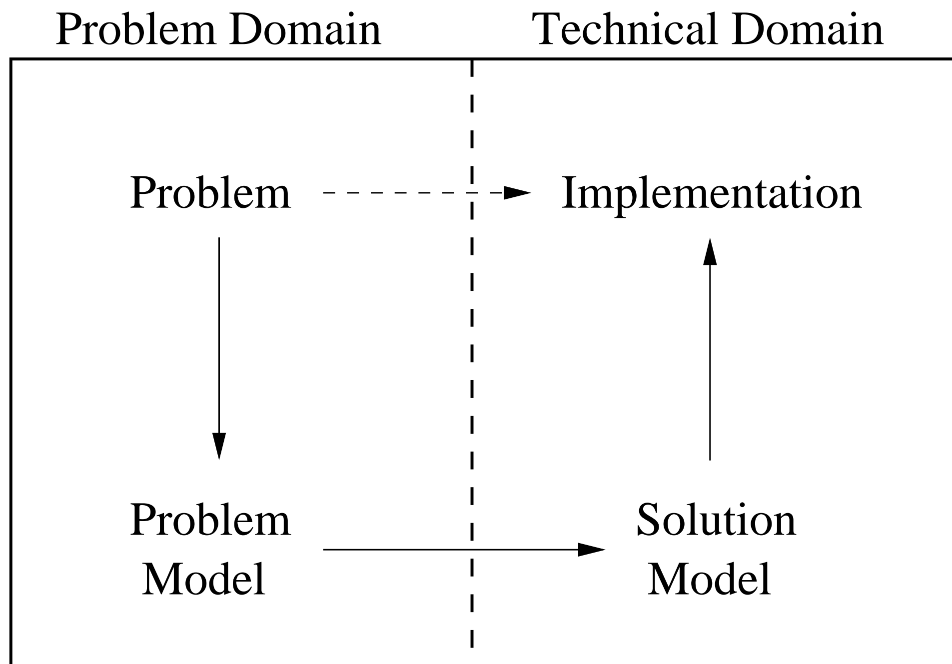

Figure 4.1: Problems with Different Domains

Designers use terminology related to a specific problem when identifying a problem and creating a problem model. This is shown on the left side of Figure 4.1. The problem domain uses terminology associated with the problem, and the problem model. The terminology used is usually part of a common metaphor. Dialogs and discussions about a problem become easier when using a metaphor. The reason is that the designers and the stakeholders have a common vocabulary and similar ideas associated with the metaphor. When creating a banking system, an ordinary bank is a good metaphor to use.

The right side of Figure 4.1 shows the technical domain. It contains the terminology used in the solution model and in the programming language. As previously mentioned, it is a general guideline to use the names from the problem model in the solution model. In other words, the technical domain usually contains the terminology used in the problem model

and new terms used in the solution. In addition to this, it also contains the programming language terminology.

```
public Enumeration gen(Object o)
{
  Vector res = new Vector();
  if(!(o instanceof MGeneralizableElement))
    return res.elements();
  MGeneralizableElement cls =
    (MGeneralizableElement) o;
  Collection gen = cls.getGeneralizations();
  if(gens == null)
    return res.elements();
  accumulateAncestors(cls, res);
  return res.elements();
}
```

Figure 4.2: The Difficulty of Expressing Contracts

An example of the difficulty of expressing semantics, because of the terminology used in the problem and technical domains, is shown in Figure 4.2. It shows code from one of ArgoUML's classes. The method extracts a list of all the classes that Object o inherit (generalizations in UML), either directly or indirectly. The Object o should be a class or other classifier from an UML design in the program. One possible postcondition to this method is:

```
If Object o is an instance of MGeneralizableElement, an Enumeration with
the generalizations of o has been returned, else an empty Enumeration has
been returned.
```

The potential problem with this postcondition is that programmers can misunderstand it. A programmer might think that the method returns the predecessors of the class MGeneralizableElement (from the technical domain, the implementation). The client of this method could try to use this erroneous result for Java Reflection [6] for example.

A GeneralizableElement is a UML entity meaning classes, interfaces and a few other constructs that can inherit another GeneralizableElement. This means that the method is meant to return the classes or interfaces that Object o has inherited (problem domain). The distinction between the two cases may be a bit confusing, but that is exactly the reason why this section was written.

It is standard practice to use the same naming in the problem domain as in the technical domain. Even though programmers sometimes misunderstand documentation because of this, it is not really a problem. ArgoUML increases the chance of misunderstandings because the programming language uses some of the terms otherwise associated with UML. The previous sample code should give a general idea of the problem. In this dissertation, the problem manifests itself in the semantic documentation. Before discovering this issue, a few contracts were easy to misinterpret in the case study.

Chapter 5

Planning the Case Study

This chapter outlines the approach of the case study. The first section presents an outline that elaborates on how to execute the case study in practice. The next section presents a feasibility study. It is conducted in order to gain knowledge about how to practically execute the case study. The feasibility study shows that approximately 100 files from the ArgoUML code is appropriate to search for semantic errors. Next, there is a section outlining some case study specific terminology. The terms presented are trivial and non-trivial. The dissertation uses these terms extensively from this point, which is why this chapter presents them. The chapter continues by explaining how to select the methods to check for semantic errors. This is needed because we cannot examine every method from 100 files in the time allotted for the dissertation. In addition, if the selection is not random, the statistics is biased.

In order to evaluate whether the clients are making semantic errors, every supplier need to have a contract. With a contract, we can check to see if the clients to the supplier meet the contract or not. The two last sections in this chapter explain how we deal with this. They discuss how to document the semantics for the methods chosen. The first of the two sections present how to construct the contracts. The last section discusses how to evaluate whether the suppliers and clients in the case study conform to the contracts we create for

the suppliers.

5.1 Outline of Case Study

As stated in the introductory chapter, the goal of this case study is to find a measure to the amount of semantic errors that exists in the latest stable version of the open source project ArgoUML. In order to do so the following steps are necessary:

- Do a feasibility study.
- Explain any specific terminology to the case study.
- Give an approach to find files, classes and methods random enough to be statistically correct. Otherwise, there is a possible risk of compromising the results.
- Define how to construct the contracts from the ArgoUML code.
- Define a way for evaluating said contracts.
- Carry out the case study.
  - Choose the methods to evaluate.
  - Create contracts for the chosen methods.
  - Check if the clients and suppliers respect the contracts.
- Process and analyse the data from the experiment execution.
- Present the results and conclusions.

This chapter has a structure very similar to the items in the list above. The next section covers the feasibility study and the rest of the chapter almost follows the list. The exceptions are that the dissertation do not report the actual practical work of executing the case study, only how to do it. Also, the results and conclusions are not reported in this chapter, but in Chapter 6 and 7.

5.2 Feasibility Study

We conducted a feasibility study by finding ten non-trivial methods from ArgoUML's code. A point of this was to estimate if it was reasonable to make a second selection based on subjectively dividing methods into the categories trivial and non-trivial.

The ten methods are not automatically included into the case study, since the case study is based on the methods from randomly selected files. We examined the ten methods for semantic errors as a preparation for the case study. The examination identified a number of attributes as interesting and possibly important. These form the base for the collection of data used in the case study.

The result of the feasibility study is that we search the methods from 100 ArgoUML source files for semantic errors. We chose the number 100 because it is large enough to give a good statistical overview of ArgoUML. The number constitutes approximately ten percent of the files in ArgoUML. In addition, 100 files seemed to be a good estimated workload when considering the time allotted for the dissertation. Since we had no trouble finding ten non-trivial methods, we decided to use the distinction of trivial and non-trivial methods. The difference is further developed in the next section.

5.3 Trivial and Non-trivial Methods

This dissertation uses the terms *trivial* and *non-trivial* to classify methods in an intuitive way. The origin of the terms trivial and non-trivial stems from the estimation that the 100 randomly selected files would probably contain too many methods for examining all methods thoroughly. In addition, the methods deemed as trivial are believed not to cause semantic errors.

The trivial methods are those that we find to be easy to understand. An example of trivial methods is accessor methods like get– and set-methods, commonly known as getters and setters. These methods either return or set a private member of a class. Methods

other then accessors, which are either very short or easy to understand, are also referred to as trivial. Many of the methods in ArgoUML are trivial. Mainly because the developers' cookbook [7] say that complex methods should be broken up into several smaller methods.

The non-trivial methods are the methods that we feel are difficult to understand. The longer the method, the more time and energy programmers need to read and understand the code. Therefore, long methods are more likely to be non-trivial and therefore more interesting in the case study.

5.4 Selecting Methods

It is necessary to find a randomised way to select the material to use in such a way that the case study is not biased. In order to achieve this, the one hundred files to search are randomised from the 1132 source files in the ArgoUML code. The feasibility study concluded the number of files to examine.

In order to reduce the number of methods from the 100 files, we made another selection. This is because each source file may contain one or more classes or interfaces with a varying number of methods, and not every method is worth examining. We have made the decision to search the non-trivial methods for semantic errors. The reason being, that we believe that programmers probably would use the trivial methods correctly. As methods grow more complex so does the importance of good semantic documentation. We believe that complex code has more complex semantics and could therefore be more likely to cause programmers to make semantic errors.

5.5 Constructing Contracts

Once we have chosen a method, its purpose and function must be determined in order to create a correct contract for it. The contracts are created in accordance with the existing

source code in the manner mentioned in Section 2.3.1.

This case study uses the existing code in the methods to create the contracts. This is done by reverse engineering the code, not considering the current documentation. The documentation is not considered because the main goal is to evaluate the actual code and not the documentation of ArgoUML. By constructing the contracts, without in any way changing the code, we can examine if the clients meet the contract and not make a semantic error. How this is done is further discussed in the next section.

5.6 Evaluation of Contracts

The process of evaluating contracts consists of examining whether the contracts meet a number of conditions. For all non-trivial methods chosen, all clients must be found. Additionally, the clients and the suppliers must conform to the rules below:

- The client must meet the precondition.
- The supplier must ensure the postcondition, assuming the precondition.
- The client must handle the result in a meaningful way, based on the postcondition. This depends on the logical meaning of the called method and the surrounding context.

The programming style used in ArgoUML is defensive, described in section 3.2. Therefore, the contracts created from the source code are likely to be tolerant. Because of this, the first criterion in the list is most likely met by most of the clients. The second criterion in the list always met, since the contracts are reverse engineered from the code.

The third item may need further clarification. If the supplier has a tolerant contract with multiple clauses, the client must deal with each clause. For example, many methods in the ArgoUML either return a result or **null** if an error occurs. If a client always expects a supplier to return a specific result and receives **null** instead, an exception may be thrown.

Therefore, a client to such a method must be able to handle both results, i.e. handle all the clauses in the supplier's postcondition.

Chapter 6

Results of the Case Study

This chapter presents the results of the examinations in the case study. First the non-trivial methods found in the case study are presented. The contracts and source code of the nine non-trivial methods can be found in Appendix C. Then the methods are grouped and discussed according to the type of their preconditions. The different groups are then compared to each other. The next logical step is discussing the postconditions of the non-trivial methods. To alleviate the discussion, the methods with similar postconditions, or behaviour, are grouped together. Since this concerns postconditions, this grouping is done without consideration of how the methods were grouped previously.

After having discussed the contracts of the non-trivial methods, follows the results of the evaluation of how the suppliers' contracts are respected by clients. Some general observations about the semantic specification level of the material used in the case study are then presented before summarising the results of the case study in the final section.

During the case study some additional issues concerning ArgoUML was discovered. The issues are; logical errors in ArgoUML's code, inconsistencies in the existing documentation, and dubious exception handling. These issues do not fall within the boundaries of the examination of the goal of this dissertation. However, they were considered important issues when considering the future development of ArgoUML. Therefore, these issues are

Class Name	*Method Name*	*Documentation*
CrWrongDepEnds	copmuteOffenders()	Intuitive
GoSummaryToInheritance	getChildren()	None
ModuleLoader	activateModule()	Intuitive
ModuleLoader	loadModules()	None
OCLUtil	getInnerMostEnclosingNamespace()	Intuitive
ProjectBrowser	open()	Intuitive
TableModelNodeByProps	rowObjectsFor()	None
ToDoItem	stillValid()	Intuitive
Translator	loadImageBindings()	Intuitive

Table 6.1: Examined Non-trivial Methods, and their Documentation Levels

further discussed in appendix B.

6.1 Methods Used in the Case Study

A total of nine non-trivial methods were found during the examination of ArgoUML. Using reverse engineering, all the methods were given contracts based directly on their implementation. In order to detect violations against the contracts, the clients of the selected methods were identified and the call sequences examined. Given that the project uses OOP and Java, with features like polymorphic calls and reflection, finding the actual call sequences proved to be quite a challenge. The nine methods are presented in Table 6.1. The contracts and source code for the methods, along with the original documentation, are available in appendix C.

6.2 Comparison of Preconditions

The preconditions of the nine examined methods are compared and the contrasts are discussed. First the six methods with tolerant contracts are treated and then the remaining three with demanding contracts.

Method Name	*Precondition*
getChildren(Object **parent**)	True
activateModule(ArgoModule **module**)	True
getInnerMostEnclosingNamespace(MModelElement **me**)	True
open(Object **element**)	True
rowObjectsFor(Object **t**)	True
stillValid(Designer **d**)	True

Table 6.2: Methods with Tolerant Contracts

6.2.1 Methods with tolerant contracts

It was expected that the defensive programming style used in ArgoUML would cause all or most of the contracts to become tolerant. This turned out to be true. As shown in Table 6.2 six of the nine methods, or 66%, have no preconditions, since they have tolerant contracts. The methods specified in the table will not show their parameter list in any other table save this, due to lack of space. A few methods exhibit contracts with preconditions, which make the contracts demanding. Some are even more demanding than they may first appear.

6.2.2 Methods with demanding contracts

The methods with demanding contracts are shown in Table 6.3. The methods specified in the table will not show their parameter list in any other table save this, due to lack of space. If the clients of method computeOffenders() do not meet the precondition, the Java run-time system throws a NullPointerException. Thus violating this precondition could cause the program to terminate.

The methods loadModules() and loadImageBindings() are constructed in a way that makes their preconditions dependant of sources of external errors. Both methods contain I/O operations that, if they cause an IOException, will terminate the program. An example of this would be if the file being read, in either method, were compromised. For the methods to complete their tasks successfully, the methods assume that no such IOExcep-

Method Name	*Precondition*
computeOffenders(UMLDeploymentDiagram **dd**)	**dd** != **null**
loadModules(InputStream **is**, String **filename**)	**is** is a file stream initiated with the file specified by **filename**. The file is readable.
loadImageBindings(String **file**)	Member **propertiesFile** contains the name of an existing, readable file. **file** contains the name of the file specified by **propertiesFile**.

Table 6.3: Methods with Demanding Contracts

tion will happen. However, as discussed in section 2.4.2, programmers can never ensure preconditions controlling external error sources. This means that the clients of loadModules() and loadImageBindings() can never meet the preconditions. In turn, this means that clients will always violate the contracts of loadModules() and loadImageBindings(). If reconstructed, the methods could be given tolerant contracts and avoid contract violations. Together, the two methods have four clients violating the contracts. The clients of the other examined methods all meet the preconditions of their suppliers. Overall, there were eleven clients to the nine examined methods.

6.3 Comparison of Postconditions

The postconditions of the nine examined methods are compared and the contrasts are discussed. Similar postconditions are grouped together. First accessor-like methods are discussed, then unstable methods are discussed, and finally methods that deviate in intended use and actual behaviour are treated.

6.3.1 Accessor-like methods

The four methods shown in Table 6.4 are all operations that resemble accessor methods. Accessor methods are methods that are used to get data about an object. This is also the purpose of these four methods. The methods show similar behaviour, but there are

Method Name	*Postcondition*
computeOffenders()	result == a VectorSet containing the offending objects or result == **null** if no offenders were found.
getChildren()	If **parent** is an instance of InheritanceNode, result == a Collection with all children of textbfparent, else result == **null**.
getInnerMostEnclosingNamespace()	result == the innermost enclosing namespace of **me** or result == **null** if **me** has no enclosing namespace.
rowObjectsFor()	result == a Vector containing the nodes of **t**.

Table 6.4: Postconditions of the Accessor-like Methods

important differences in their postconditions. The three methods computeOffenders(), getChildren(), and getInnerMostEnclosingNamespace() all return either the desired result or **null**. **null** means that either no result was found or an error occurred. The possibility of returning two different answers may provide a way of informing clients whether the operation was successful. But this also means that the clients must be able to handle both clauses in the postcondition. The method rowObjectsFor() only has one return clause. The answer looks the same whether a result is found, an error occurs or no result is found. All the methods perform similar tasks, but in different ways. This means that even though the operations are similar, client programmers must adapt to several different types of postconditions.

The clients of computeOffenders(), getChildren(), and rowObjectsFor() have no problems handling the postconditions. However, the client of getInnerMostEnclosingNamespace() does not account for the result == **null** clause in the postcondition.

6.3.2 Unstable methods

As discussed in section 6.2.2, the methods shown in Table 6.5 have preconditions that cannot be met by clients. This is why we call these methods unstable, they may not return at all if an error occurs. The postconditions only describe the result when the method

Method Name	*Postcondition*
loadModules()	result == **true**
loadImageBindings()	result == the Properties found in the file specified by member **propertiesFile**.

Table 6.5: Postconditions of the Unstable Methods

actually returns according to the law of total correctness [8]. As previously stated, the methods could be reconstructed with tolerant contracts, in order to avoid termination of the program.

6.3.3 Methods deviating from intended use

The three methods shown in Table 6.6 all appear to have deviations between their intended uses described by their documentation, and their actual behaviour described by their implementation. The reason we group these as methods deviating from intended use is that they do not seem to have anything else in common. The fact that they all seem to deviate from what we believe to be their intended use is why we examine them here. An example of this is the method activateModule(). According to its documentation the method returns either **true** or **false**, but according to its implementation only returns **false**. The description of method open() does not describe the intended use, but under which circumstances it is to be called. Thus, the intended use is not documented and its actual behaviour is confusing, since it does not do anything. The method stillValid() also exhibits some deviation between intended use and actual behaviour. All such deviations make the methods more difficult to use since they make the intention behind the methods unclear.

Method Name	*Postcondition*
activateModule()	result == **false**
open()	Nothing has happened.
stillValid()	if this ToDoItem has no poster, or if this ToDoItem's wizard is running, then result == **true** else result == the poster's view of this item's validity.

Table 6.6: Methods with Deviations between Semantics and Implementation

6.4 Contract Violations

As stated previously, violations to contracts or invariants represent semantic errors. The results from examining the calls from clients are presented in Table 6.7.

The table's first column contains the class and method names of the supplier. The second column contains the class and method names of the client. The three remaining columns describe the result of the examination. The first of the remaining columns, labelled *Meets Pre*, reveal whether the client met the supplier's precondition. The second column, labelled *Builds on Main Clause*, reveals whether the clients builds its further execution on the main result of the supplier. The last column, labelled *Handles Additional Clauses*, reveal whether the client handles all of the remaining result clauses correctly, if the supplier has more than one result clause. **Yes** signifies that everything is in order, **No** signifies a semantic error and - indicates that there are no multiple clauses to handle. Since the contracts are made by reengineering the supplier's code, the supplier always builds on its precondition an always fulfils its postcondition. Therefore, the table describing the examination of the calls from clients have no column for showing this. Two of the non-trivial methods, activateModule() and open(), have no clients, i.e. there are no calls to these methods, in the table this is represented by empty cells.

Supplier	*Client*	*Meets Pre*	*Builds on Main Clause*	*Handles Additional Clauses*
CrWrongDep-Ends.compute-Offenders()	CrWrongDep-Ends.predicate2()	Yes	Yes	Yes
CrWrongDep-Ends.compute-Offenders()	CrWrongDep-.toDoItem()	Yes	Yes	-
CrWrongDep-Ends.compute-Offenders()	CrWrongDep-.stillValid()	Yes	Yes	Yes
GoSummaryTo-Inheritance-.getChildren()	Navperspective-.getHelperIndex()	Yes	Yes	-
ModuleLoader-.activateModule()				
ModuleLoader-.loadModules()	ModuleLoader.-loadModulesFrom-PredifinedLists()	No	Yes	-
ModuleLoader-.loadModules()	ModuleLoader.-loadInternal-Modules()	No	Yes	-
ModuleLoader-.loadModules()	ModuleLoader.-loadModules-FromFile()	No	Yes	-
OCLUtil.getInner-MostEnclosing-Namespace()	OCLUtil.get-ContextString()	Yes	Yes	No
ProjectBrowser-.open()				
TableModelNode-ByProps.row-ObjectsFor()	TableModel-Composite-.setTarget()	Yes	Yes	-
ToDoItem-.stillValid()	ToDoList.force-ValidityCheck()	Yes	Yes	Yes
Translator.load-ImageBindings()	Translator.get-ImageBinding()	No	Yes	-

Table 6.7: Results from Contract Evaluation of Non-trivial Methods and their Clients

Semantic Specification Level	*None*	*Intuitive*	*Structured*	*Executable*	*Formal*
Classes	52	59	0	0	0
Methods	348	427	0	0	0

Table 6.8: Semantic Specification Levels of Classes and Methods in the Case Study - Values

Semantic Specification Level	*None*	*Intuitive*	*Structured*	*Executable*	*Formal*
Classes	47%	53%	0%	0%	0%
Methods	45%	55%	0%	0%	0%

Table 6.9: Semantic Specification Levels of Classes and Methods in the Case Study - Percentages

6.5 General Observations

The ArgoUML system contains 1329 classes and interfaces, and these contain 9380 methods. The 100 examined files of source code in the case study contained 107 classes, including abstract classes, and four interfaces. The random sample constituting the case study was, prior to the study, estimated to be approximately ten percent of ArgoUML's total source code. But instead the case study constitutes a little more than eight percent of ArgoUML's source code.

During the examination of the semantic level in the documentation in ArgoUML, the documentation is graded using the semantic specification levels presented in section 2.6. The results are summarised in the Tables 6.8 and 6.9, which present the actual values and the percent rates respectively. The results show that the level of documentation in the source code does not follow what is stated in the coding standards [7] for the project. If the code documentation standards are applied, all code should at least have intuitive semantic descriptions.

6.6 Summary of Case Study Results

From the case study the following results can be summarized. Six of the nine non-trivial methods have intuitive descriptions. Three of the nine non-trivial methods actually have demanding contracts.

Altogether, the nine methods have eleven clients. According to the results, presented in Table 6.7, from examining these clients, the calls from the clients cause five contract violations. The contract violations are divided between three suppliers. Four of these violations are caused by preconditions that are impossible to meet. The fifth violation is caused by a client ignoring an additional result clause. Although the client handles the result from legal use correctly, it fails to handle the additional clause. This means that 33% of the suppliers have clients that cause semantic errors. The five calls causing contract violations constitutes 45% of all the calls from clients.

Chapter 7

Conclusions

This chapter contains our discussions and conclusions of the case study based on the results presented in Chapter 6. The discussion also considers the additional findings presented in appendix B. Besides a discussion of the implications of the result of the case study, problems concerning the approach used in the case study are brought up. The chapter ends with a few suggestions for future work.

7.1 Case Study Analysis

The case study contained nine non-trivial methods. If assuming that the proportion of non-trivial methods found in the case study is the same as in ArgoUML, ArgoUML would contain approximately 109 non-trivial methods. This approximation is not entirely correct since the selection of non-trivial methods was subjective. The nine non-trivial methods had eleven calls from clients. Five of these calls caused semantic errors. Thus counting the semantic errors in the case study itself was easy. However, estimating the number of semantic errors in the whole of ArgoUML, by estimating the number of clients causing semantic errors, is extremely difficult. Instead, we divide the number of semantic errors among the suppliers, and then estimate the number of non-trivial suppliers with clients

that cause semantic errors.

Of the nine methods found in the case study, three had clients that caused semantic errors by violating the contracts. If the proportion of contract violations among non-trivial methods is the same in the case is the same as in ArgoUML, ArgoUML would have 36 non-trivial methods whose contracts are violated by clients. Since ArgoUML contains a total of 9380 methods, and having approximately 36 non-trivial methods with erroneous clients, 0.4% of the methods in ArgoUML have contract violations.

The result of 36 non-trivial methods with contract violation in the whole of ArgoUML seems to be relatively small. Having a small result set of non-trivial methods in the case study gives the results a relatively small impact on the whole of ArgoUML. The reason for this is that the result may be too biased by the subjective trivial and non-trivial distinction to give an accurate picture of the actual number of semantic errors in ArgoUML. In order to have a greater impact, more non-trivial methods would have to be found. Taking into account the incomplete documentation, the logical errors, the inconsistencies in existing documentation, and the dubious exception handling indicates that we should have been able to find more semantic errors in the case study than we did.

During the case study we examined the source code of the randomly selected files. By keeping track of which methods and classes that were documented and which were not, we discovered that approximately 50% of ArgoUML's source code is documented. We also discovered that some of the documentation contained slight inconsistencies. Nonetheless having some documentation, even slightly inconsistent, was a great help in understanding the intentions of the code, compared to having no documentation. Therefore, it would be helpful if all the source code was documented.

All the existing documentation in ArgoUML is intuitive. From our own experience, structured documentation is even more helpful, since it clearly states the intentions of the code. Therefore, we believe that one should strive to use structured documentation when documenting semantics, i.e. contracts and invariants. We feel that this could be a useful

tool for handling semantics in projects such as ArgoUML. Hopefully, structured semantics could, to some extent, remove the previously mentioned issues in ArgoUML.

7.2 Problems

This section separately discusses the problem of finding suitable methods to use in the case study and the problem of construction contracts for those methods. Finding only nine methods to examine more thoroughly weakens the impact of the case study. Also, not being able to construct the appropriate contracts for the methods actually found may compromise the result of the case study in other ways.

7.2.1 Problems with finding suitable methods

Using a random sample of files from ArgoUML is a suitable base for a case study. However, the approach of finding methods should perhaps have been more random as well. Using the distinction of trivial and non-trivial methods resulted in a small number of methods that were more thoroughly examined. Taking a random sample of methods from the whole of ArgoUML or thoroughly examining every method in a smaller random sample of files could perhaps have given a better view of the system.

A reason for finding few non-trivial methods is that the coding standard states that programmers should break up complex or long methods into several smaller and simpler ones. Also, the general guideline in OOP is also to use as short methods as possible to make the code easy to understand. Since ArgoUML is programmed in Java [6], and therefore uses OOP, the result is many trivial methods.

7.2.2 Problems concerning the construction of contracts

The number of semantic errors discovered in the case study was relatively small. Due to the other discovered issues, we believe that the explanation for finding such a small

number of semantic errors is the approaches used in the case study. We believe that the issue causing the most impact on the case study is how the contracts for the methods were created. Creating contracts that mirror the intended use might have been better than using reverse engineering. Using reverse engineering tightly couples the contracts to the code. Thereby, the contracts are not the abstraction of intended use that they are meant to be. Creating contracts that describe the intended use could have resulted in finding more semantic errors. The reason being, that some of the additional issues discovered could then have been classified as errors.

Creating contract from someone else's code, trying to understand the intentions, would be difficult even if the code is documented. Given that nearly half of ArgoUML's code is not documented, reverse engineering seemed like a better option at first. However, after concluding the examination we believe that creating the more abstract type of contracts would have given a better view of the semantic integrity in ArgoUML. In addition, using reverse engineering makes small logical errors part of the contracts, which may result in contracts differing from the intended use they are meant to represent.

7.3 Conclusions of the Case Study

After examining the non-trivial methods found in the randomly chosen files in the case study, we found a total of five semantic errors. The errors are caused by clients violating the contracts of their suppliers. Either ArgoUML has very few semantic errors or our method of investigation is incapable of detecting the errors. As mentioned in section 7.2, we have two distinct objections to our method of investigation. If the issues with the method were resolved, it might prove more effective for detecting semantic errors.

7.4 Future work

Here we present a few suggestions for future projects that perhaps can help give a better view of semantics in software design and development.

The first project idea is to mimic this case study, but with a better approach of constructing contracts and selecting methods. As the section 7.2 points out, contracts based on intended use could give a better indication of the number of semantic errors in ArgoUML and other similar software development projects.

The assumption that contracts could have decreased the number of errors in the development phase can be the base of future dissertations. Every project has its fair share of bugs. If programmers use contracts, the number of bugs may decrease. This is an interesting investigation that can be further explored in a future work.

Examining the exception handling of existing programs could reveal several issues of semantic importance. One example of this is the non-trivial methods, loadImageBindings() and loadModules(), discovered in the case study which have preconditions that cannot be met by clients. This could provide an insight on how to use exceptions and contracts together in a more semantically correct way.

The last prospective project is a more active investigation of contracts. Other dissertation can be based on conducting experiments using contracts in an ongoing project or on a part of a project, much like ArgoUML. For example, demanding contracts could be created for a part of a system. This could perhaps more clearly show whether contracts help remove inconsistencies and errors, or not.

References

[1] ArgoUML, http://argouml.tigris.org, 2004-02-18.

[2] Martin Blom, Eivind Nordby. *Semantic Integrity in Component Based Development.* Internal report on CBSE, Mälardalen University, Västerås, Sweden, 2000.

[3] Martin Blom, Eivind J. Nordby, Anna Brunström. *Method Description for Semla – A Software Design Method with a Focus on Semantics.* Karlstad University, Karlstad, Sweden, 2000.

[4] Ivica Crnkovic, Magnus Larsson (editors), *Building Reliable Component-Based Software Systems*, Artech House Publishers, 2002.

[5] Donald G. Firesmith, *A Comparison of Defensive Development and Design by Contract*, p 258-267 in Proceedings from TOOLS USA '99.

[6] Java Technology. http://java.sun.com, 2004-03-10.

[7] Marcus Klink, Linus Tolke, *Cookbook for Developers of ArgoUML*, http://argouml.tigris.org/files/documents/4/2924/cookbook.pdf, 2004-05-14.

[8] Bertrand Meyer. *Object Oriented Software Construction*, 2nd Edition, Prentice Hall PRT, 2000.

[9] Eivind J. Nordby, Martin Blom, Anna Brunström. *On the Relation between Design Contracts and Errors: A Software Development Strategy.* Karlstad University, Karlstad, Sweden, 2002.

[10] Jason Robbins, *Cognitive Support Features for Software Development Tools*, http://argouml.tigris.org/docs/robbins_dissertation/diss.pdf, 2004-02-18.

[11] UML 1.3 specification. http://www.omg.org/cgi-bin/doc?formal/00-03-01, 2004-02-28.

[12] J. Warmer, A. Kleppe, *The Object Constraint Language, Precise Modeling with UML*, Addison Wesley, 1999.

[13] Wikipedia. http://en.wikipedia.org/wiki/Formal_semantics, 2004-03-17.

Appendix A

Acronyms

CASE Computer Aided Software Engineering

DbC Design by Contract

OCL the Object Constraint Language

OO Object-Oriented

OOAD Object-Oriented Analysis and Design

OOP Object Oriented Programming

RUP Rational Unified Process

TDD Test-Driven Development

SERG Software Engineering Research Group

UML Unified Modelling Language

XP eXtreme Programming

Appendix B

Additional Results from the Case Study

While searching the files used in the case study for suitable non-trivial methods a few other important observations about the code in general were made. These observations are not part of the goal of case study but the authors still feel that they are important issues that concern the semantic quality of ArgoUML. The first issue discussed is the existence of logical errors in the code, where each logical error found is described. The logical errors are regarded both in the view of the current system, and in the risk that they pose for further development and change of ArgoUML. Following this is a section discussing the semantic inconsistencies in the existing documentation of ArgoUML's methods and the implementation of the methods. The chapter ends with a discussion of the frequent exception handling found in the code, and some issues concerning this, and a section pointing out the danger in unresolved issues for the future development of ArgoUML.

B.1 Logical Errors in ArgoUML

During the examination of the methods in the files used in the case study, a number of possible logical errors were observed. Logical errors were defined in section 2.5.2. These logical errors made the source code more confusing. In fact, some of the methods were considered non-trivial because they contain them. Thereby, the logical errors affected the selection of non-trivial methods, which is why they are included in this appendix. Another reason to discuss this issue is that, since the logical errors are part of the code of the non-trivial methods, these errors become part of the contracts obtained through reversed engineering. Then the logical errors have not only affected the selection of non-trivial methods, but also the result of the contract evaluation.

A total of seven doubtful code snippets have been observed among the non-trivial methods chosen for further examination in the case study. Each method, along with the reasons that we find them doubtful, is described in its own section. Most of the contradictions in these methods stem from contradictions between their documentation, which should specify the intended use, and implementation, the actual behaviour. At present these errors may not be detected since some of the methods are not in use in the current version of ArgoUML.

B.1.1 ModuleLoader.activateModule()

This method does not actually contain a logical error. Instead some doubts about what it does and what it is supposed to do exists. The documentation of the method suggests that the method is fully implemented. However, its implementation looks like the dummy-implementations that exist in the ArgoUML source code. Instead of making use of the parameter, the method simply answers every call by returning **false**. At present the method has no clients so this should not pose any problems in the current version of ArgoUML. However, during further development a possible inconsistency between intended

use, described by the documentation, and actual behaviour could cause problems. The source code of this method is found in appendix C.3.

B.1.2 ModuleLoader.loadModules()

This method seems to have three possible ways to end at first glance, **true**, **false** of termination of program on account of an exception. However, if the implementation is further examined, it appears that the return statement returning **false** can never be reached. The reason is that the while loop has no break statement. Therefore the method can only return **true** or terminate the program. This logical error has affected the contract of this method since it is no longer possible for the method to return **false**. However, this does not appear to affect the method's clients. Further examination of the call sequences to the clients, and the clients of the clients might show other effects of this logical error. The cause of this logical error is most likely that a break statement in the while loop simply has been forgotten. The source code of this method is found in appendix C.4.

B.1.3 OCLUtil.getInnerMostEnclosingNamespace()

This method has an implementation that may cause null to be returned. One of the break conditions in the loop is that the parameter 'me' is not **null**. But at return 'me' is cast to a MNamespace regardless of what its value is. Since this is what is implemented this is what ends up in the method's contract. This doubtful implementation has affected the outcome of the case study since it caused the only contract violation found among the clients of the nine non-trivial methods. The client shows no sign of expecting null as return value and thereby violates the contract. The source code of this method is found in appendix C.5.

B.1.4 ProjectBrowser.open()

This method is another example of contradiction between intended use and actual behaviour. The documentation states that the method is called as a response to a request. Therefore, it seems reasonable that something should happen in response to the request, not that the request should be ignored. At present the method has no clients so this implementation poses no problem in the current version of ArgoUML. However, since its documentation suggests that it is fully implemented, this could be a problem in further development. The source code of this methods is found in appendix C.6.

B.1.5 ToDoItem.stillValid()

According to the documentation of this method it is to return **false** if its poster has been deactivated. But the implementation returns **true** if the reference to the poster is **null**. To the best of our knowledge, the reference to the poster being **null** seems like a good indication that the poster has been deactivated. It seems that the method should instead return **false** when the poster is **null**, i.e. deactivated. However, we have been unable to find the statement that changes the reference to the poster to **null**, and are therefore unsure whether the poster ever actually is **null**. This implies that the poster is not likely to be **null**. The method seems to be working fine though, since it also relies on additional factors to determine the validity of the current ToDoItem. The source code of this method is found in appendix C.8.

B.1.6 Translator.loadImageBindings()

This method has a documentation and a syntax that makes it appear far more generalized than it actually is. The documentation states that the method will return the Properties found in the file specified by the parameter file. But instead the method returns the Properties in a file specified by a private instance variable. At present this causes no

problems since the only client of this method uses that specific private member in the call. However, should an attempt be made to use the method in the more general way, as intended by the documentation, errors would arise. The source code of this method is found in appendix C.9.

B.2 Inconsistencies in Existing Documentation

During the examination of the 100 files used in the case study it was determined that approximately 55% of all the methods had intuitive documentation, see section 6.5 for more details. But ever so often the documentation contains semantic inconsistencies. By this we mean that the documentation is not consistent with the implementation. This can cause trouble during further development of ArgoUML since the documentation is the memory aid for the intended use of the implementation, the semantics of the code, as discussed in section 2.1. Because of this, we decided to count the number of inconsistencies to get an estimation of how much of the documentation that contains such deviations.

Among the 427 examined methods that have documentation, 27 inconsistencies were found. That is to say approximately 6% of the existing method documentation was faulty in one way or another. The simplest documentation inconsistencies concerns documenting parameters that no longer exist. Another variant of this is not documenting parameters that do exist. Although the inconsistencies may not cause trouble today, they make the methods more difficult to understand and thereby more difficult to use correctly in the future. The documentation is an important issue when regarding the future development of the ArgoUML project. Documentation should be written, but it also should be kept up to date in order to fulfil its purpose. Also as mentioned previously, despite the fact that the ArgoUML cookbook [7] states that all methods should be documented; nearly half of them are not.

B.3 Dubious Exception Handling

During the examination of the files used in the case study a rather frequent use of try-catch clauses was noted. This is not unusual in Java, but it seems that a number of the catch clauses contain statements which abruptly terminate the program without notifying the end user why. Two such methods, loadImageBindings() and loadModules(), are included among the non-trivial methods examined. The ArgoUML cookbook [7] states that when exceptions occur, a message describing the problem should be written to a log file. Thus exceptions should not be reported to the user. There are several different types of log entries used to differentiate the severity of the issue causing the exception. However, at several points in the examined files, this practice of logging is not followed.

Among the 100 files used in the case study, 14 files containing a total of 43 try-catch clauses were found. Since these occur rather frequently within the files used in the case study, and sometimes cause the program to terminate without warning this issue was considered important. Having a lot of statements that abruptly terminate the program can make the program unstable. Unstable software is difficult and tedious for the end user to use.

Defensive programming is a programming style that advocates exception handling, which may be one of the reasons that there much exception handling in ArgoUML. Since there are guidelines for how the exceptions are to be handled these were decided to represent correct usage. Any deviation from that practice was considered dubious. This means that exception handling incorporating logging, or ignoring an exception but motivating the reason, is considered acceptable exception handling. Consequently, ignoring an exception without a motivation, simply printing a stack trace or an error message, or just terminating the program is considered dubious exception handling. This kind of exception handling can make the program both hard to understand for programmers and difficult to use for end users.

Using these criteria, 30 of the 43 try-catch clauses found handled the exceptions in

accordance with the guidelines in the cookbook. And 13 of the 43 try-catch clauses found did not. This means that approximately 30% of all the exception handling examined was considered dubious and does not appear to follow the guidelines in the ArgoUML cookbook. Faculty members of the Department of Computer Science at Karlstad University have found ArgoUML to be difficult to use due to sudden, unexpected program termination. This may indicate that the exception handling may be causing problems in ArgoUML.

This sidetrack from the main focus of the dissertation was discovered late in the work on the dissertation, which is why it has not been further explored. Studying the issue of dubious exception handling may be a possible future work, see section 7.4.

B.4 Further Development of ArgoUML

ArgoUML is still under development, and since the beginning of the work on this dissertation, several new beta versions have been released. There seem to be a lot of issues; logical errors, lack of documentation, inconsistencies in existing documentation, and dubious exception handling. Should these issues remain unresolved, they pose a risk of causing errors and confusion in the future development of ArgoUML.

Appendix C

Excerpts of ArgoUML Source Code

This appendix contains the source code of the non-trivial methods examined in the case study. Each method is presented along with its contract, which was created through reengineering. The methods are sorted alphabetically, first by class name and then by method name.

C.1 CrWrongDepEnds.computeOffenders()

In class CrWrongDepEnds, method VectorSet computeOffenders(UMLDeploymentDiagram dd) was considered non-trivial. The following contract was constructed for this method:

Pre: dd != null
Post: result == a VectorSet containing the offending objects or result == **null** if no offenders were found.

```
/**
 * If there are deps that are going from inside a FigComponent to
 * inside a FigComponentInstance the returned vector-set is not
 * null.  Then in the vector-set are the UMLDeploymentDiagram and
```

```
 * all FigDependencies with this characteristic and their
 * FigObjects described over the supplier and client.
 **/
public VectorSet computeOffenders(UMLDeploymentDiagram dd) {
    Vector figs = dd.getLayer().getContents();
    VectorSet offs = null;
    int size = figs.size();
    for (int i = 0; i < size; i++) {
        Object obj = figs.elementAt(i);
        if (!(obj instanceof FigDependency)) continue;
        FigDependency fd = (FigDependency) obj;
        if (!(fd.getOwner() instanceof MDependency)) continue;
        MDependency dep = (MDependency) fd.getOwner();
        Collection suppliers = dep.getSuppliers();
        int count = 0;
        if (suppliers != null && (suppliers.size() > 0)) {
            Iterator it = suppliers.iterator();
            while (it.hasNext()) {
                MModelElement moe = (MModelElement) it.next();
                if (moe instanceof MObject) {
                    MObject obj_sup = (MObject) moe;
                    if (obj_sup.getElementResidences() != null
                        && (obj_sup.getElementResidences().size() > 0))
                        count = count + 2;
                    if (obj_sup.getComponentInstance() != null)
                        count = count + 1;
                }
```

```
            }
        }
        Collection clients = dep.getClients();
        if (clients != null && (clients.size() > 0)) {
            Iterator it = clients.iterator();
            while (it.hasNext()) {
                MModelElement moe = (MModelElement) it.next();
                if (moe instanceof MObject) {
                    MObject obj_cli = (MObject) moe;
                    if (obj_cli.getElementResidences() != null
                        && (obj_cli.getElementResidences().size() > 0))
                        count = count + 2;
                    if (obj_cli.getComponentInstance() != null)
                        count = count + 1;
                }
            }
        }
        if (count == 3) {
            if (offs == null) {
                offs = new VectorSet();
                offs.addElement(dd);
            }
            offs.addElement(fd);
            offs.addElement(fd.getSourcePortFig());
            offs.addElement(fd.getDestPortFig());
        }
    }
```

```
        return offs;
    }
```

C.2 GoSummaryToInheritance.getChildren()

In class GoSummaryToInheritance, method Collection getChildren(Object parent) was considered non-trivial. The following contract was constructed for this method:

Post: If parent is an instance of InheritanceNode, result == Collection with all children of parent, else result == **null**.

```
    public Collection getChildren(Object parent) {
        if ( parent instanceof InheritanceNode) {

            List list = new ArrayList();

            Iterator it =
                ModelFacade.getSupplierDependencies(
                                                            ((InheritanceNode) parent)
                                                              .getParent()).iterator();

            while (it.hasNext()) {

                Object next = it.next();
                if (ModelFacade.isAAbstraction(next))
                list.add(next);
            }

            it =
```

```
            ModelFacade.getClientDependencies(((InheritanceNode) parent)
                                                  .getParent())
            .iterator();

        while (it.hasNext()) {

            Object next = it.next();
            if (ModelFacade.isAAbstraction(next))
                list.add(next);
        }

        Iterator generalizationsIt =
            ModelFacade.getGeneralizations(((InheritanceNode) parent)
                                                 .getParent());
        Iterator specializationsIt =
            ModelFacade.getSpecializations(((InheritanceNode) parent)
                                                 .getParent());

        while (generalizationsIt.hasNext())
            list.add(generalizationsIt.next());

        while (specializationsIt.hasNext())
            list.add(specializationsIt.next());

        return list;
    }
    return null;
```

```
}
```

C.3 ModuleLoader.activateModule()

In class ModuleLoader, method boolean activateModule(ArgoModule module). The following contract was constructed for this method:

Post: result == **false**

```
/** Activate a loaded module.
 *  @return true if the module was activated,
 *          false if not or if it was already active.
 */
public boolean activateModule(ArgoModule module) {
    return false;
}
```

C.4 ModuleLaoder.loadModules()

In class ModuleLoader, method boolean loadModules(InputStream is, String filename). The following contract was constructed for this method:

Pre: **is** is a file stream initiated with the file specified by **filename**. The file must be readable.

Post: result == **true**

```
public boolean loadModules(InputStream is, String filename) {
    try {
        LineNumberReader lnr =
        new LineNumberReader(new InputStreamReader(is));
```

```
while (true) {
    String realLine = lnr.readLine();
    if (realLine == null) return true;
    String line = realLine.trim();
    if (line.length() == 0) continue;
    if (line.charAt(0) == '#') continue;
    if (line.charAt(0) == '!') continue;
    String sKey = "";
    String sClassName = "";
    try {
        int equalPos = line.indexOf("=");
        sKey = line.substring(0, equalPos).trim();
        sClassName = line.substring(equalPos + 1).trim();
    }
    catch (Exception e) {
        System.err.println ("Unable to process " + filename +
            " at line " + lnr.getLineNumber() +
            " data = '" + realLine + "'");
        continue;
    }
    try {
        if (sKey.startsWith("module.")) {
        loadClassFromLoader(getClass().getClassLoader(),
                            sKey,
                            sClassName,
                            true);
        }
```

```
                } catch (Exception e) {
                    Argo.log.warn("Could not load Module: " + sKey);
                    ArgoModule.cat.debug("Could not load Module: " + sKey, e);
                }
                sKey = "";
            }
        }
        catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        return false;
    }
```

C.5 OCLUtil.getInnerMostEnclosingNamespace()

In class OCLUtil, method MNamespace getInnerMostEnclosingNamespace(MModelElement me). The following contract was constructed for this method:

Post: result == the innermost enclosing namespace of **me** or result == **null** if **me** has no enclosing namespace.

```
    /**
     * Get the inner-most enclosing namespace for the model element.
     */
    public static MNamespace getInnerMostEnclosingNamespace (MModelElement me) {
        while ((me != null) &&
               (!(me instanceof MNamespace))) {
            me = me.getModelElementContainer();
```

```
        }
        return (MNamespace) me;
    }
```

C.6 ProjectBrowser.open()

In class ProjectBrowser, method void open(Object element). The following contract was constructed for this method:

Post: Nothing has happened.

```
    /**   Called by a user interface element when a request to
     *   open a model element in a new window has been recieved.
     */
    public void open(Object element) {
    }
```

C.7 TableModelNodeByProps.rowObjectsFor()

In class TableModelNodeByProps, method Vector rowObjectsFor(Object t). The following contract was constructed for this method:

Post: result == a Vector containing the nodes of **t**.

```
    public Vector rowObjectsFor(Object t) {
        if (!(t instanceof UMLDeploymentDiagram)) return new Vector();
        UMLDeploymentDiagram d = (UMLDeploymentDiagram) t;
        Vector nodes = d.getNodes();
        Vector res = new Vector();
        int size = nodes.size();
```

```
    for (int i = 0; i < size; i++) {
        Object node = nodes.elementAt(i);
        if (node instanceof MNode) res.addElement(node);
    }
    return res;
}
```

C.8 ToDoItem.stillValid()

In class ToDoItem, method boolean stillValid(Designer d). The following contract was constructed for this method:

Post:if this ToDoItem has no poster, or if this ToDoItem's wizard is running,
then result == **true**
else result == the poster's view of this item's validity.

```
/** Reply true iff this ToDoItem should be kept on the Designer's
 *  ToDoList. This should return false if the poster has been
 *  deactivated, or if it can be determined that the problem that
 *  raised this issue is no longer present. */
public boolean stillValid(Designer d) {
    if (_poster == null) return true;
    if (_wizard != null && _wizard.isStarted() && !_wizard.isFinished())
        return true;
    return _poster.stillValid(this, d);
}
```

C.9 Translator.loadImageBindings()

In class Translator, method Properties loadImageBindings(String file). The following contract was constructed for this method:

> **Pre:** Member **propertiesFile** contains the name of an existing, readable file. **file** must contain the name of the file specified by **propertiesFile**.
>
> **Post:** result == the Properties found in the file specified by member **propertiesFile**.

```
/**
 * Loads image bindings from a File.
 * @param file the properties file
 * @return the properties in file
 */
private static Properties loadImageBindings (String file) {
    InputStream inputStream = null;
    Properties properties = new Properties();
    try {
        inputStream = Translator.class.getResourceAsStream(propertiesFile);
        properties.load(inputStream);
        inputStream.close();
    } catch (IOException ex) {
        cat.fatal("Unable to load properties from file: " + file, ex);
        System.exit(1);
    }
    return properties;
}
```