Construct by Contract: Construct by Contract: An Approach for Developing Reliable Software

Juan Jose Mendoza Santana

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment
of the requirements for the
Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley

Supervisor: Dr. Rosemary Monahan

July 1st, 2013

Declaration

I hereby certify that this material, which I now submit for assessment of the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of other save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:_________________________ Date:__________________

Acknowledgments

I want to thank to Dr. Rosemary Monahan for all her support, help, patience and contributions made to this project. I extend this thank to all the professors that have shared their knowledge with myself throughout the different lectures I have had at the National University of Ireland (NUIM), and to my classmates whose support has been immensely helpful for the culmination of this research.

I want to thank as well to the Erasmus Mundus Program for funding my studies and thus the development of this project. I also acknowledge to Tony Hoare, Bertrand Meyer and Joseph Kiniry whose ideas inspired this project, in conjunction with the Erasmus Mundus Master of Science in Dependable Software Systems. I thank to Eva Darulova, Dino Distefano, Vladimir Klebanov and Reiner Hähnle who kindly helped me with doubts about the verification tools they've been involved.

At last but not the least I want to thank God, my family and my friends because they are always sharing my dreams, supporting and helping me no matter what the circumstances are.

Abstract

This research introduces "*Construct by Contract*" as a proposal for a general methodology to develop dependable software systems. It describes an ideal process to construct systems by propagating requirements as contracts from the client's desires to the correctness proof in verification stage, especially in everyday-used software like web applications, mobile applications and desktop application. Such methodology can be converted in a single integrated workspace as standalone tool to develop software. To achieve the already mentioned goal, this methodology puts together a collection of software engineering tools and techniques used throughout the software's lifecycle, from requirements gathering to the testing phase, in order to ensure a contract-based flow. *Construct by Contract* is inclusive, regarding the roles of the people involved in the software construction process, including for instance customers, users, project managers, designers, developers and testers, all of them interacting in one common software development environment, sharing information in an understandable presentation according to each stage. It is worth to mention that we focus on the verification phase, as the key to achieve the reliability sought. Although at this point, we only completed the definition and the specification of this methodology, we evaluate the implementation by analysing, measuring and comparing different existing tools that could fit at any of the stages of software's lifecycle, and that could be applied into a piece of commercial software. These insights are provided in a proof of concept case study, involving a productive Java Web application using struts framework.

Categories

D. Software
D.2 Software Engineering
D.2.4 Software/Program Verification [1]

General terms

Management, Documentation, Design, Reliability, Standardization, Verification.

Keywords

Software Engineering, *Construct by Contract*, Design by Contract, Software Verification, Dependable Software, Software Engineering, Dependable Software Systems, Rigorous Software Development, Java Reliable Software.

Table of Contents

Chapter 1. Introduction ... 7
1.1 Research Overview ... 7
1.2 Problem Statement ... 7
1.3 Motivation ... 8
1.4 Aims and Objectives ... 9
1.5 About this Document ... 9
1.6 Chapter Summary ... 9
Chapter 2. Related Work ... 10
2.1 Requirements Engineering ... 10
2.2 Software Design ... 11
2.2.1 UML and OCL ... 12
2.2.2 BON ... 13
2.3 Rigorous Software Development ... 14
2.3.1 Hoare Logic ... 14
2.3.2 Design by Contract ... 15
2.3.3 JML ... 15
2.3.4 Verified Software Initiative ... 16
2.3.5 Software Verification ... 17
2.3.5.1 Formal Verification ... 18
2.3.5.2 Static Verification ... 18
2.3.5.3 Dynamic Verification ... 22
2.3.6 Software Validation ... 23
2.4 State of the Art ... 23
2.5 Chapter Summary ... 24
Chapter 3. Construct by Contract: Solution Proposed ... 25
3.1 Principles of CbC ... 25
3.2 Software Development Lifecycle for CbC ... 26
3.3 Roles involved in CbC ... 27
3.4 Specification Languages according to CbC ... 28
3.5 Conceptual Structure of CbC ... 28
3.6 Tool Support for CbC ... 29
3.7 Functionality per Phase ... 30
3.7.1 Gathering Requirement Phase for CbC ... 30
3.7.2 Design Phase for CbC ... 31
3.7.3 Specification and Coding Phase for CbC ... 32
3.7.4 Verification Phase for CbC ... 33
3.7.4.1 Static and Formal Verification ... 33
3.7.4.2 Dynamic Verification ... 33
3.8 Chapter Summary ... 34

Chapter 4. Construct by Contract Tool Design 35
4.1 Use Cases for CbC Tool 36
4.2 Sequence Diagram to develop Software with CbC Tool 37
4.3 Architecture Diagram for CbC Tool 37
4.4 Components Selection for CbC Tool 38
4.4.1 Workspace 38
4.4.2 Requirements 38
4.4.3 Design 39
4.4.4 Coding 39
4.4.5 Verification 39
4.4.6 Testing 41
4.5 Components Diagram for CbC Tool 41
4.6 Chapter Summary 41
Chapter 5. Proof of Concept 42
5.1 Industrial Case Study 42
5.2 Gathering Requirements Phase 43
5.3 Design Phase 45
5.4 Specification and Coding Phase 48
5.5 Verification Phase 50
5.5.1 Static and Formal Verification Phase 50
5.5.2 Dynamic Verification 51
5.6 Chapter Summary 52
Chapter 6. Evaluations 53
6.1 Evaluation of the Proposed Solution 53
6.2 Evaluation of the CbC Tool Design 54
6.3 Evaluation of the Proof of Concept 55
6.3.1 Gathering Requirements Phase 55
6.3.2 Design Phase 56
6.3.3 Specification and Coding Phase 56
6.3.4 Verification Phase 56
6.3.5 Summary of the Section 57
6.4 Evaluation of the Existing Work 57
6.5 Chapter Summary 57
Chapter 7. Conclusions 59
7.1 Summary 59
7.2 Problems Found 59
7.3 Future Work 60
7.4 General conclusion 61
References 62

Chapter 1. Introduction

Throughout this chapter, we will present and overview of the research work, including the problem we will try to solve, the motivation to solve that problem, the clear objectives of this research and overview of this document.

1.1 Research Overview

Reliability in software is a feature that would make clients the happiest people in the world, and even programmers, software architects, project managers and users. Unfortunately in most of the existing systems this feature is almost inexistent [2], thus software is trapped into a costly cycle of bug discovery and fixing [3]. Formal methods and static verification are some techniques that can be used to assert software reliability, in a structured fashion these tools are put together in a methodology called Design by Contract (DbC). This methodology is based on the formal specification of functional requirements, through the implementation of contracts; such contracts enumerate the responsibilities of clients and providers and is enhanced by the use Theorem Provers in order to evaluate Software Correctness.

This research seeks to explode and expand the Design by Contract methodology to a new level, where contracts are not only formal specifications for methods within classes in an Object Oriented Software, but also they are real human desires, and functional specifications of the solution for a problem. In order to build such methodology we will propose a path for software development based on the software's lifecycle, and we will zoom in every step of this path in order to analyse what tools or techniques can be integrated to achieve reliability in the final result.

Once we have described the methodology, we will evaluate it in a proof of concept, based on the state of the art of the different blocks that have been joint in *Construct by Contract*. We will build this proof in the Java Software Domain, by taking a bit of Industrial Software already working, and showing how the methodology can be applied in that case. It is worth to mention that the case is based in a Java Web Applications found on the Struts Framework, being this considered as an everyday-used application.

1.2 Problem Statement

Software is developed by programmers; programmers are humans; humans make mistakes, and therefore software can be incorrect. Some mistakes can be detected by compilers, some cannot, some mistakes are not a big deal, but some can cost human lives, some mistakes are cheap to correct, but some mean large financial losses.

Verified software is crucial in safety critical systems, such as transportation, health, banking and avionics systems [4], but correctness should be a feature present in any kind of software, after all, isn't it what clients expect to get? Yes, it is, but nowadays it might be expensive, and time consuming, besides the fact that programmers are not used to deal with formal correctness, they prefer the "try and fail method", usually called testing. Sadly, testing is not a way to proof 100% correctness, either if we use boundary testing, or equivalent partitions, or coverage testing or any other kind of testing; at the end, testing only increases confidence in the software, but it does not give any warranty.

The concrete problem is that commercial everyday-used software is not proved to be correct by formal methods [5], because it requires an additional effort [6] [7], and

because developers are not prepared to deal with it, especially in the vast range of application domains, tools, programming languages, design patterns and development methodologies existing in the "software engineering world". It is always economic, and time costly to learn a wide range of tools to achieve such correctness, and nowadays there is not any ultimate unique tool that can include correctness in current development skills. Even though it is true that there are several tools to achieve correctness, they are neither seamless integrated in one single workspace, nor are they complete, nor are they automated and nor are they easy to use.

Hence, we come with the following questions: Is there any "easy" way to introduce correctness in daily software? If so, what would it be? What do developers should do to enhance their software reliability? Is it responsibility only of developers?

1.3 Motivation

In my previous industrial experience as software engineer, I have had to deal with all the stages involved in the software's lifecycle, either gathering requirements, or designing, or coding and/or testing. I have done it based on the principle of functionality, and quickness, after all clients always want something working in the least time possible; but now, after the master course in Rigorous Software Development, and the literature review in software verification, I have realized that it would be even better have software correctness as driving principle for commercial everyday-used software, and not only for critical systems.

The Verified Software Imitative (VSI), which will be covered in more detail in section 2.3.4, has proposed an international project aiming to bring verification in the software industry, in its manifesto has requested the contribution of industrial partners, researcher and developers to achieve a world where programmers make less mistakes almost as any other professional person, and where software is absolutely reliable in an industry that adopts verification the theory and practice throughout the software life-cycle, including requirement, specification, model and design, coding, testing, and quality assurance, all these integrated in one tool [5].

This is the motivation for this research, to be a small contribution to the VSI, by proposing one alternative that, if being implemented, can accomplish pretty much the aims of this manifesto. By proposing *Construct by Contract*, we can promote software correctness as desirable and reachable in everyday-used software, by showing how this feature can be conceived from the basic idea of Design by Contract integrated in the Software's Lifecycle, in such a way that can be "easily" adopted by developers, either individual freelancers preferring Agile Development, or software factories with well-structured procedures following the standard waterfall model.

A more personal motivation, is to achieve the degree of Master in Science in Dependable Software Systems, by presenting a thesis related with, of course, dependable software, but stop seeing it as something that only happen in very specific domains, and promote the acquisition of dependable software in the common day. There is also a motive coming from my desire of make my life as developer easier, and have happier clients.

1.4 Aims and Objectives

The ultimate goal behind this project is to generate an ideal methodology that can be followed first by myself in my everyday job, and then shared with others, bringing formal verification and software correctness into everyday-used software, enhancing its reliability, by propagation contract in each stage of the software development lifecycle.

The specific objectives of this research are:

- Identify basic stages in the software development lifecycle, and the actors/roles involved.
- Define a methodology to expand Design by Contract through the software development lifecycle.
- Define how contracts must be introduced and presented in each phase.
- Propose tools that can be adopted to achieve the goal of using contracts in each stage
- Show how the methodology might work in a specific application with a specific domain (Proof of concept)
- Show a comparison of exiting tools to verify software and choose the one that fits the best the methodology on the specific application designated
- Identify and summarize the challenges in building a seamless unified workspace that supports the methodology.

These objectives will be covered in Chapter 3, Chapter 4 and Chapter 5 and then will be evaluated in Chapter 6.

1.5 About this Document

The Chapter 1 presents an introduction for this research project, summarizing what will be discussed along the following chapters and the motivations for this work. In the Chapter 2 of this document, we will explain the state of the art in the domain of dependable software systems and its relationship with software engineering; we will introduce the different existing bricks that later will be used to build our solution for the identified problem. In Chapter 3, the solution proposed with the name of *Construct by Contract* will be explained with all of its components and their definitions, including the presentation of the software development lifecycle used. In Chapter 5 the realization of the solution will be shown through a case study based on a piece of software used in the industry. In 0 we will summarize what has been done in this project, then, both the solution proposed and the proof of concept will be evaluated, besides we will evaluate the existing work against our proposed solution; we will also present some threats to validity, a the future work and a general conclusion of the project.

1.6 Chapter Summary

Along this chapter, we have presented the problem we will face with our solution, and the motivation for this project, we have also established the objectives that will be covered and evaluated in further chapters.

Chapter 2. Related Work

It is essential, to mention briefly what is the essence of this work, after all computers have become so common nowadays that we rarely stop to think what they are. The Oxford Dictionary defines a computer as “a person who makes calculations, especially with a calculating machine” [8]. Probably this definition might have suited perfectly from the 17th century till the end of 19th century, but after that, computers stopped being humans to be defined as machines so that now we can define computation as “a sequence of simple, well-defined steps that lead to the solution of a problem” [9].

What is that thing that has given real value to computers in daily life, this is in one hand the ubiquity of machines, bringing concepts like mobility, wearable computing, cloud computing, etc. in the other hand, and it has been due to the software that computers have become indispensable in everyday life. In this work, we will focus on the second one: “the software”, and to be precise in a way to develop dependable software.

Regardless the definition of software engineering, as any engineering, it is a discipline that seeks to apply scientific knowledge into practice to develop techniques needed to generate new knowledge, improvement of society and economic growth. Hence, software engineering pursuits to dictate the best practices and techniques to develop software, considering all its aspects such as functionality, reliability, usability, efficiency, maintainability and portability among others [10], [11].

Once we have established that the concerning topic is related with software engineering, we will indicate what specific section of this wide domain is covered. To be precise, we will focus on how to develop dependable software, defining dependability as a feature of software that ensures that it does what it is expected to do an anything else. In order to achieve such dependability, we will explore some specific areas and techniques that are part of software engineering discipline. This exploration is done in the following sections.

2.1 Requirements Engineering

It encapsulates a collection of standardized techniques that can help to understand the definition of the functionality expected from a specific software, it means, to dive into clients and user’s needs, to figure out what the software must do. For instance, if we go back to the definition of computer as a person who performs calculations, and we ask that person to calculate the sum of one and three, we expect to get four as the result; in the same way if we write a program to reproduce such calculation, we should expect the same result. Thus, the main reason for the existence of software is to solve a problem, but we cannot solve a problem that we do not know, or that we do not understand.

Requirements are the inception point for software, they are the reason of its existence and its ultimate goal. Nevertheless, it can happen that clients themselves do not know what they want, in this case there is not much that can be done, and it might be like trying to solve an inexistent problem. It can also happen that clients actually know what they want, but the project manager, or whoever the person in charge for gathering requirements is, do not understand these requirements properly, and then, this misunderstanding can provoke software to failure. According to the Standish Group Report Chaos the United States spend more than $250 billion each year on IT application development for approximately 175,000 projects. From this total 31.1% of projects will be cancelled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. It also shows that clear statement of

requirements is key in the success of such projects while incomplete requirements and specifications are the second most common reason for project failures [12].

Nowadays there exist some well-known techniques used to gather requirements, and keep track on them. Some of these techniques include interviews with clients and users, storyboard with scenarios for the use of the software, quick prototyping, long contract-style documents, UML use case diagrams, etc. [13]. Some of them are embedded in defined frameworks, and some of them have a specific way to represent requirements, especially implementing visual representations like diagrams, graphs, lists, pictures, figures, etc.

Regardless the existing techniques, we will always have to deal with the way clients express their wishes, it might be tough to ask the client to learn any specific language or tool, just to express what they want, it could even annoy them; therefore, we must approach to them in a seamless way, It means the most natural way we can, and that way is for sure the Natural Language. By using this language, clients can express themselves easily, but then, the Project Manager, should be able to gather requirements and to map them in a way that later can be transformed into software, and understood by the technical people like software architects, developers or testers. This is the actual real challenge, how to preserve requirements throughout the whole software development lifecycle.

We have also said that in order to consider a piece software as dependable, we must ensure it is correct; thus we must ensure that the software does what is expected to do, and i.e. the implementation of the software accomplishes the specification. Of course if I am developer, I will claim that what I've done is what I've been asked to do, but it can happen that I've made any mistake. It raises new questions: How can we demonstrate that the code accomplishes its specifications? Is there any way to prove that what the client required is exactly what he is getting? The answer is yes, but if we say that this proof is based on my speech, it would not be a convincing argument, would it? Therefore, we need an indisputable way to prove it, based on any universal truth, thus that we appeal to mathematics as the unfailing proof that our software is correct. How can we prove mathematically something based on natural language? The answer is we cannot, since symbols is the language of maths we have to achieve some symbolic representation of our client's specifications, in order to be able to manipulate them and prove the correctness of our software. This process will be explained in detail in Chapter 3.

2.2 Software Design

In the previous section, we have discussed the importance of understanding the client's requirements since they will define software's path. Once we have gathered them, now we can proceed to think how software will be constructed, i.e. we can design our software. As it happens with requirements engineering, software design is also a branch of software engineering, and it explains a set of techniques used to represent the structure of the software.

Software design defines a system in terms of small components with specific functionality for each of them. It also explains how these components are interconnected and how the information flows within each of them. The most used way to do this representation is by using graphic diagrams where each of the components can be drawn and detailed, and where the relations can be seen like lines or arrows between

blocks. Some of these diagrams are flowcharts, UML class diagrams, UML sequence diagrams, architecture diagrams, entity-relationship diagrams, etc. [14]

2.2.1 UML and OCL

So far we have mention UML in both the present and the previous section; thus we will introduce briefly what UML is. The Unified Modelling Language (UML) is a set of language-like rules, used to describe software and its components in a standardized way. By itself, it is only a language, but its features make it easy to represent the same information throughout visual diagrams [15]. Some of its advantages are:

- Visualization
- Validation
- Clear communication [16]

Another reason why UML is being widely used is because of the support it has had in the IT industry as it is a standard, and since it has been taken by IBM, it has also been accepted by the IT development community. All of this acceptance make easier to adopt UML as the favourite way to model software, besides the set of existing tools that support the development of UML diagrams, like ArgoUML [17], Rational Software Architect [18], UMLet [19], etc. [20].

In the previous section, we mentioned briefly contract-style based documents, even though it is not a commonly used technique to specify requirements, for it is based only on a written speech in natural language, without any decoration nor specific visual representation. Nevertheless it might be the closest way to approach software correctness as it can be used as a checklist of requirements, it can also has some legal/financial use because it can acts as a legal contract between the clients and developers, and for sure it provides a high level description, especially useful in large systems. But again, the problem resides in how to convert these statements in natural language into something that can be used to prove software correctness. So far there is not an absolute answer to that question, that is why *Construct by Contract* will explore the possible choices and will propose a methodology to achieve this goal, but we will retake this topic in Chapter 3. The following is an example of UML class diagram:

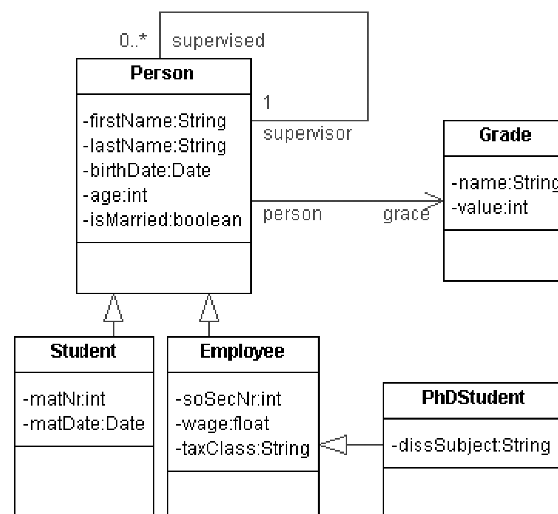

Figure 1 Example of UML class diagram [21]

Now we will explore one of the existing tools that allow us to write some contract-like restrictions in our software design, this tool is the Object Constraint Language (OCL), this, similar to UML, is a system of syntactical rules that allow us to express such restrictions in a formal way. This tool works together with UML class diagrams, discussed in previous paragraphs, in order to impose constraint to our classes, their methods and their fields. This language uses the Design by Contract approach that will be discussed in the following section, in order to write preconditions, postconditions and class invariants, by providing a set of formal expressions that allow us to model software functionality without the ambiguities of natural language and without tough and complex mathematics [22].

```
context Person
inv tudOclInv1: self.supervisor.grade.value > self.grade.value

context Student
inv tudOclInv2: self.supervisor.grade.value > self.grade.value

context Employee
inv tudOclInv3: ((self.grade.name = 'diploma') implies (self.taxClass = 'tc1'))
and ((self.grade.name = 'doctor') implies (self.taxClass = 'tc2'))
and ((self.grade.name = 'professor') implies (self.taxClass = 'tc3'))
```

Figure 2 Example of OCL contracts [21]

From here, we can infer that UML in addition with OCL, can give us a complete suite to design software, but we also need to keep in mind that it does not care about requirements gathering by client side, nor about program verification. So despite all their advantages and convenient features, there is still a gap that these two tools cannot fill by themselves because finally Unified Modelling Language is different to Natural Language, which means that a client, who does not know UML and OCL, could not express its requirements in such languages. Therefore, it is necessary to perform some additional tasks to translate these languages, one into another. This is what will be covered in section 3.7.2 of this thesis.

2.2.2 BON

So far we have explored the functionality of two tools to design software, they are UML and OCL. We have also found that there is no direct connection between them and the original specification of requirements in natural language. Lucky those are not the only tools to model software, so here we introduce another tool that seems to fit in the goal of transforming from natural language requirements to some software design representation, this is the Business Object Notation (BON) [23].

BON is a method for specifying object oriented software, in a structure similar to natural language, like a high level approach, which also includes a graphical notation. It claims to minimize the gap between software's lifecycle states, like design and implementation by using the same semantic and conceptual base for the notation in each stage, which increases seamlessness, reversibility, all these in a contract-based way [24]. The reader can see an example of BON below.

CLASS	*ELEVATOR*		Part: 1/1
TYPE OF OBJECT Models an elevator which is being pulled by a motor.		INDEXING **cluster:** *ELEVATOR_CONTROL* **created:** 1997-03-29 kw **revised:** 1997-04-08 kw	
Queries	Current floor, Pending floor requests, Is elevator moving? Are doors open? Is elevator traveling up? Is elevator traveling down? Is elevator idle?		
Commands	Open doors. Close doors. Process floor requests.		
Constraints	Cannot be traveling both up and down. If stopped and no more requests then elevator is idle. Cannot move when doors are open. An idle elevator is not moving.		

Figure 3 BON informal class chart example [25]

From here that BON seems to be a terrific option for of pursuit on correctness, it seems to include the features that UML and OCL together are missing to achieve the goal of keep requirements as contracts easily throughout the software's lifecycle [26]. So why do not just use BON and be happy? Well, the answer is that although BON uses a high-level definition, it is still not natural language, so either the client learns BON, or the Project Manager does, in order translate clients requirements from natural language to BON, in any case, there is still a gap, so we cannot consider this as a complete solution, but this is only one limitation, another limitation is the small number of tools available to deal with BON, especially compared with a huge amount of tools to implement UML and OCL, and the final limitation is that BON is conceived to work particularly with the Eiffel programming language and ins this thesis we expect to work on Java Software; thus we cannot consider BON as the ultimate solution for our concerns.

2.3 Rigorous Software Development

What we have called here as Rigorous Software Development is a set of tools and techniques that allow us to develop dependable software systems by means of formal prove of the correctness of software applying logical and mathematical operations to software's code and specifications [27]. In the following sections, each of these techniques and tools will be briefly explained.

2.3.1 Hoare Logic

It is a logic system, developed by C.A.R. Hoare that allows us to prove software correctness by means of the so called Hoare Triple {P}S{Q}. It establishes that given a safety state for execution {P} called precondition, some statements S are executed, and the result of such execution must satisfy the postcondition {Q}. This system logic includes rules to reason about statements like skip, assignments, composition, conditionals, and while loops for both partial and total correctness [28]. Below it is shown one example of Hoare logic for assignment rule.

Hoare Logic: Rule for Assignment

$$\frac{}{\{Q[E/id]\}\ id=E;\ \{Q\}}$$

Example:
{y+7>42} **x=y+7;** {x>42}

Figure 4 Hoare Logic Example for Assignment Rule

This logic system is used by many tools, frameworks and models to prove software correctness, and thus to develop dependable software, but here we must emphasize that Hoare Logic is not a tool by itself, it is just a collection of reasoning rules. This can be considered as the core of the maths underneath reliable software.

2.3.2 Design by Contract

It is a design strategy, also known as Programming by Contract, originally proposed by Bertrand Meyer as part of the Eiffel programming language. It retakes the principle of Hoare Logic and extends it in order to impose some restrictions in software design than actually help to develop dependable software. It defines contracts as specifications where clients and providers must observe certain obligations in order to obtain certain benefits, these contracts are expressed as preconditions, postconditions and invariants. To achieve that, such contracts must be known by the calling client, so he has the responsibility to execute the provider's method in a safety state [29].

We have talked about clients and providers in the previous paragraph, but, it is worth to clarify that those two concepts do not have the same meaning that in the previous sections. Here, both clients are providers are classes belonging to an object oriented software. It differs from the *Construct by Contract* definition where client can be understood as the software's sponsor instead, but these definitions will be properly established in the section 3.3. Because, in Design by Contract, the client is not a person who express their desires, but a class who calls a method in another class, we clearly see a gap between the two previous phases. Where is the relationship between these class contracts, and the human-client requirements, originally expressed in natural language, and then mapped into any design representation either with UML+OCL or BON? So far there is not any well-defined procedure to link these concepts, and this is what *Construct by Contract* is all about, and this is what will be defined in Chapter 3.

2.3.3 JML

We have talked about how to gathering requirements from our clients, and then how to represent them in the design of our software. We have also talked about some techniques to achieve software correctness by using design by contract, and Hoare Logic, from here we have introduced the term contract, which we have described as the specification of a program indicating what it expects to receive, and what it is expected to produce whenever the input conditions are satisfied. Now we need to think in how those contracts are implemented in real life, or at least how they should be implemented since our concerns are with the Java programming language and the Java framework, we need to introduce a new concept that allow us to write such contracts for our Java software, and this is exactly the role of the Java Modelling Language (JML).

JML [30] is, just as English and UML and OCL, a language; it means that is a way to express something based on some syntactic and semantic rules. It focuses on the specification of the behaviour of a program, ideally nut not necessary by annotating Java interfaces. It combines Design by Contract [29] with a Model-Based Specification Approach [31] and with some refinement calculus [32].

Why JML?, first because it is designed to model Java programs, and this is our domain, secondly because it allows to write specifications in terms of preconditions, postconditions and class invariants as Design by Contract proposes [33], thus at some point we can think that these contracts may have some similarities with the client's

requirements, thirdly because it uses annotations to specify Java-code files, so the specification does not affect the functionality of the software, and can coexist in a same place, so it can be used as documentation as well, hence we get the seamless feature sought. Below the reader can find an example of JML specifications.

```
public class BankingExample
 {

    public static final int MAX_BALANCE = 1000;
    private /*@ spec_public @*/ int balance;
    private /*@ spec_public @*/ boolean isLocked = false;

    //@ public invariant balance >= 0 && balance <= MAX_BALANCE;

    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample()
    {
        this.balance = 0;
    }

    //@ requires 0 < amount && amount + balance < MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == \old(balance) + amount;
    public void credit(final int amount)
    {
        this.balance += amount;
    }
}
```

Figure 5 JML Banking Example

We've said that we can gather client's requirements in natural language, then we can design our software either in UML+OCL or EBON, then we can use JML to write down the specifications of the software in the way Design by Contract requires. All these in order to develop dependable software systems, but we are still missing connection point, the first one is how all these previous concepts and tools interact within each other, the second one is how JML annotations ensure our software correctness. The first point will be discussed as the core of the solution proposed in Chapter 3. The second one will be exposed in the following section.

2.3.4 Verified Software Initiative

The Verified Software Initiative (VSI) is an international project for fifteen years that focus on the scientific challenges of large-scale software verification; its main goal is the construction of error-free software systems, by constructing a comprehensive programming theory that covers the features needed to build practical and reliable programs, and by providing a coherent toolset that automates such theory and scales up to the analysis of industrial software. It also includes a collection of realistic verified programs that could replace unverified programs in current service and continue to evolve in a verified state [5].

Under this initiative, researchers collaboratively will study the scientific foundations of software verification covering theoretical aspects, interoperable tools, and comprehensive verification experiments. They will focus on languages and logic for software specification, abstract models, and methods in order to design systematically, analyse, and build software, integrating these methods with programming languages, efficient verification algorithms, and theory unification, in a seamless, powerful, and versatile toolset that supports all aspects of verified software development. The software industry will adapt and build on the VSI toolset, incorporating verification technology into commercial tools. Industrial partners will contribute software artefacts as experimental material they will benefit both from the VSI toolset and the experimental verification of

substantial and smaller-scale artefacts. This initiative will also prove a new source of employment where researchers and graduates in verification technology can find a place in the industry to apply, adapt, and extend the ideas and tools from VSI to specific needs and application domains [5].

We can mention different perspectives to think that this initiative is worth: as client I can be happy that my software can be error-free so I will not lose money, as user I will have a better experience, as developer my job will be easier and as researcher I will have and active area to work. So we can clearly see how the initiative represents an ideal world, and once we have introduced our solution in Chapter 3 we'll see how *Construct by Contract* plays a role in this initiative.

2.3.5 Software Verification

To build software means to write a program that can solve a problem for this construction, developers are key since they are who possess the knowledge to write such programs using programming languages. A programming language is a collection of basic standardized blocks of instruction used to build more complex instructions by following some syntactic rules; at the end it is only a way to express such instructions after they are expressed we require somebody, or something that can understand such instructions and execute them. The same happen with human beings, we can express something, but those expressions would mean nothing if they were not attended by somebody else, even more, we can express some specific desires or commands, and we expect somebody else to perform such activities, but what if we do not express our desires or commands correctly, it would lead us to unexpected or even worse undesirable results. Thus, we have to be sure that what we are expressing is what we mean to. It is pretty much the same with programs, somehow we have to ensure that what we are expressing is well expressed, and that is what we certainly want to express.

Nowadays is pretty easy to ensure that what we are programming is correctly expressed, thanks to compilers, since they can parse our source code and thus determine if the code we have written satisfies the syntactic rules of the specific programming language if so then our program can be built and executed. Even though, compilers can help us only to evaluate how we write our instructions, they cannot evaluate the meaning of the instructions; for that reason is that we need to verify and validate our programs.

Verifying a program means to assure that it does what it is expected to do, it means that all requirements are fulfilled completely within the actual software [14]. With this definition about what verification is, we can easily note a relationship between verification and Design by Contract discussed in previous sections. In fact, we could say that Design by Contract is a way to perform software verification, and indeed it is, because we have both, our requirements and our actual code, thus we can "compare" them.

But how do we actually perform software verification?, well, we probably could do it manually, something like have a group of people trying to follow the code and some rules, perhaps Hoare Logic rules, to prove that the program matches its requirements. But it would rise more problems, like humans can make mistakes, it would require a huge additional human-effort to achieve this goal, and therefore it would mean an increase in the price and time required to build software. For this reason, and since we have computers that allow us to automate some repetitive tasks, we can draw upon

automatic tools that help us to verify software. Such tools can be categorized according to the way the perform verification, in the following sections we will explore some of the main categories.

2.3.5.1 Formal Verification

Throughout the following sections, we will talk about the different options we possess to execute the formal verification of our software. Even though they are different in essence, they can actually work together by seeking the same goal; therefore, the following sections are not about an exclusive decision, but about a structural composition.

SAT/SMT Solvers

A SAT solver is an automatic procedure used to determine if, by assigning any boolean value true or false to each of the variables in a given formula, the whole formula can be evaluated as true; this combination of values is known as interpretation. The SAT solver must decide if there is any interpretation that satisfies the formula. These tools are not powerful enough because the problem to decide if a formula can be satisfied belongs to the NP-Complete, and it also requires some additional help over the formulae, like a transformation into a Normal Form, to improve the solver performance [34].

Satisfiability Modulo Theories (SMT) solvers can be considered as an extension to SAT solvers, because they use the same principle but extend the functionality by adding some theories as first order propositions than can be specific for different domains, and that use decision procedures to interact with SAT [35].

Some SMT solvers are Z3 [36], Yices [37], Simplify [38], CVC3 [39]. We know they exist, and we know they can help us to prove “somehow” software correctness. We have said before that the proof for corrects relies in the demonstration that the actual implementation of a given software matches its specification. Sadly we cannot take the piece of code and its specifications, just as they are, in an SMT solver, and prove its correctness. As we have said, these solvers require some well-defined logic formulae. We can say now that we have, for example, a piece of Java code with its respective JML notations, but we can’t feed any SMT with this information, therefore we need some additional tools that can be used as kind of translators, so we can transform our programs and our specifications into logical formulae that the SMT can reason about. We won’t go any further in the discussion of each specific SMT solver, because it is not the main domain of the work to go to that low level of detail, nevertheless we have provided the references where the reader can find more information about each of them.

2.3.5.2 Static Verification

In the previous section, we have defined an important logical-mathematical tool that can be used to reason and to prove things, in our case, software correctness. The problem is that such tools deal with logic formulae instead of programs (code and specifications). Thus we need some other tools that allow us to deal with code by itself, such tools are static checkers.

Static checkers take the code, including comments, of a program and perform an analysis on it, based on the syntactic and semantic rules of the language, without needing to execute the software. This analysis can help us to find potential software

bugs, and even more to prove the relation between specifications and implementations of a program [40].

Because our main concern is related with the verification based on Design by Contract, we will focus on the tools that help us to achieve this goal within Java domain. These tools are explained in the following section.

Krakatoa

Krakatoa is a verification tool for Java programs that works as front-end of the Why platform for deductive program verification, it deals with programs annotated in a variant of the Java Modelling Language known as KML [41]. The basic functionality of Krakatoa is depicted in the following diagram.

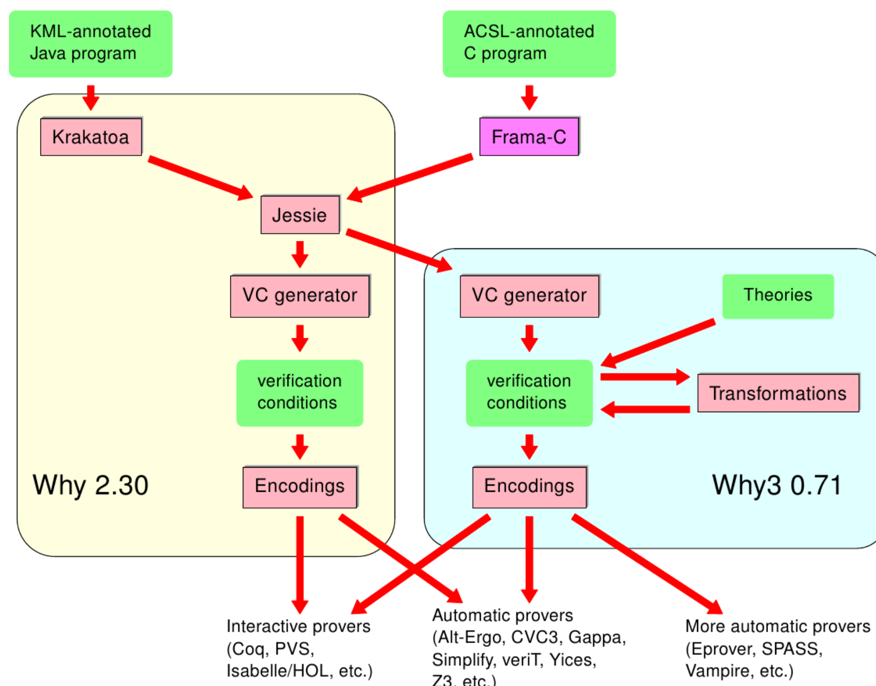

Figure 6 Architecture of Krakatoa [42]

ESC/Java2

This is the second version of the Extended Static Checker for Java. This tool attempts to identify common run-time errors Java programs annotated with JML contract. This tool performs a static analysis of the code and its formal specifications. The number and kinds of checking that ESC/Java2 performs can be customized by adding specific annotations called pragmas to the programs. [43].

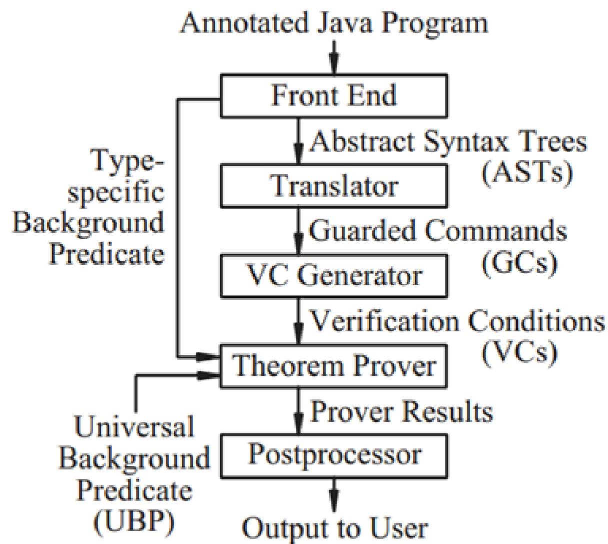

Figure 7 Architecture of ESC/JAVA2 [44]

Modern Jass

Modern Jass uses Java 5 annotation to specify contracts, the Pluggable Annotation Processing API to validate contracts, and the Bytecode Instrumentation API to enforce contracts. It provides seamless integration into every java IDE and build process, Modern Jass validates contract while the Java compiler (javac) works because it uses the annotation processing facilities. In case a contract cannot be validated successfully, a compile error occurs. Such compiler error is created by Modern Jass and presented in the same way a javac compiler error is presented. For instance, when the pre-condition refers to an unknown variable, a contract compile error occurs. Contracts are enforced at runtime by using bytecode instrumentation, so that the program terminates with an AssertionError if an assertion is not met, it is enough to execute one command to run the program in "contract-protected" mode [45]. The following diagram shows the functionality of Modern Jass.

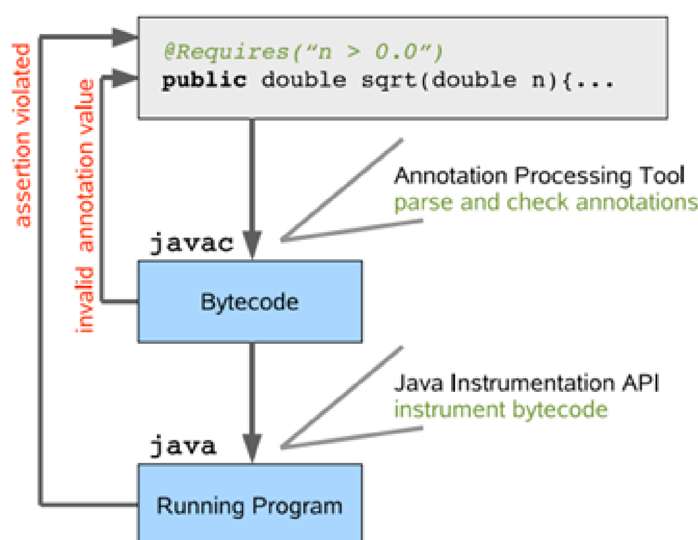

Figure 8 Architecture of Modern Jass [45]

JStar

jStar is a verification tool based on separation logic to automatically verify object-oriented programs written in the Java programming Language. It evaluates if a given program, or piece of program accomplish the method's preconditions and postconditions. It also computes automatically loop invariants by means of abstract interpretation. The main components of this tool are:

- A theorem prover for separation logic that embeds an abstraction module for defining abstract interpretations.
- A symbolic execution module for separation logic.

jStar relies on coreStar, which is a generic language intended for building verification tools based on separation logic. [46]

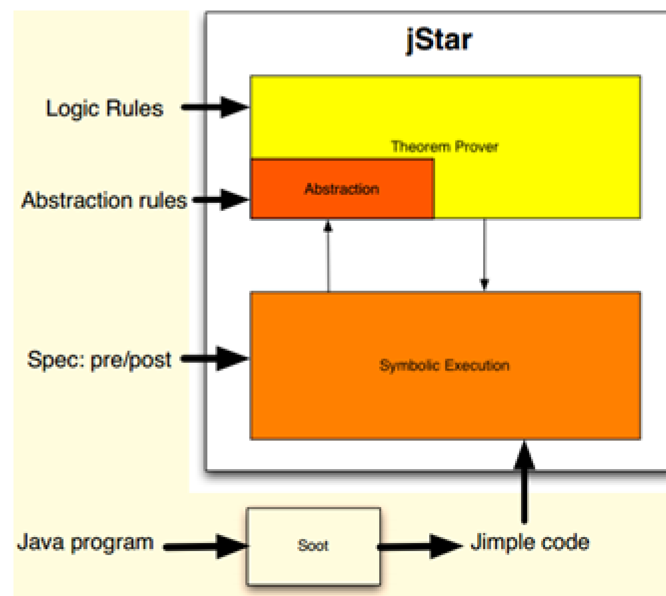

Figure 9 Architecture of JStar [46]

KeY

KeY tool claims to integrate design, implementation, formal specification and verification of software coded with the object-oriented methodology in a seamless way. The core of the tool is a theorem prover for first-order Dynamic Logic for Java with a user-friendly graphical interface [47]. The following figure represents the architecture of the KeY tool.

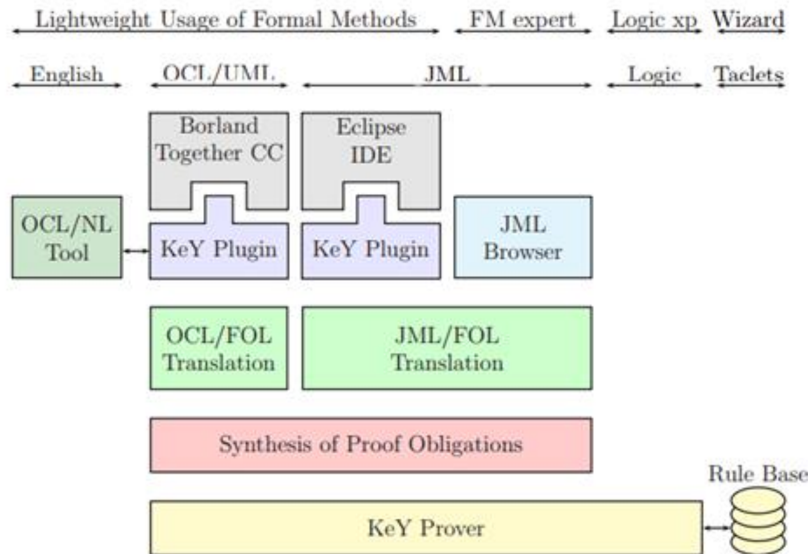

Figure 10 Architecture of KeY [48]

2.3.5.3 Dynamic Verification

As we have seen, formal verification is done by using symbolic representations of our program with some decision solvers, while static verification is done over the code by itself, these two kind of verifications can be used together in order to achieve an integral verification, but so far we haven't actually seen that the software works correctly, we can assume it, or infer it from the previous verifications, nevertheless there are some things that cannot be verified by them, like decimal representations, at least not without some modifications to the code, and here is where dynamic verification takes place. It basically consists in executing the software in a progressive way to ensure that the results it throws are the expected results under certain circumstances. This process is commonly called testing.

Throughout this chapter we have talked about client requirements that can be used as contracts to specify our software, and that can be verified with some static and formal verification tools, now we will explain the relationship of requirements and testing. We can test as our imagination or our impulses tell us to do it, sure, but then it would not be engineering at all, for testing we need to have a control flow about what to test, and this control flow is exactly the use cases that we mentioned in section 2.2. Each use case is related to one specific set of requirements that together compound a specific task, this task can be proven by executing any sequence of steps to achieve the goal off the task. Since we have previously listed our requirements, we can use that list as a checklist for our use cases, and thus for our test. It's worth to mention that this is the most used way to verify software, usually in software projects, there exist a group of people that is in charge of executing tests [49].

The main advantage of dynamic verification over static and formal verification, is that it can be done by simply running the program, while for the other two sorts of verification some additional knowledge and tools are required, for these reasons and the fact that formal and static verification tools are not mature enough, is why software is still tested, and probably will be tested for a long time more. Nevertheless there are some disadvantages in testing, and the clearest evidence is that no matter how hard a software is tested, it is always possible to find any kind of bugs (defects) on it [50].

Automatic Testing

Although testing seems to be the easiest way to verify software, it is actually expensive in economic and timing terms, this is why the testing research area has been growing up. Nowadays we can count on tools that allow us to deal with testing in an easier way. It’s enough to imagine one test case, coming from a use case; that has to be repeated hundred times by a person, to realize that this is not optimal, therefore we automate such task. In Java domain the, the most used tool, because of its ease, is JUnit, which allow us to easily write repeatable tests. The problem with this tool, and many other is that it is still responsibility of the tester to write the test cases, therefore there is still some work to do [51]. Fortunately, there exist some sophisticated tools that allow us to generate automatic test cases, some examples are Jargete [52] and JMLUnitNG [53], which allow us to generate unit tests cases for Java classes specified in JML.

2.3.6 Software Validation

We are arriving to the last part of the software’s development lifecycle, from the inception to almost the release as a productive dependable software, gradually passing from specification to design, to development and to verification, the last step coming through is the validation. The key concept while validating software is to ensure that it looks exactly as the client expected to. It can be very easy, Validation can be very easy if we have our contract-like requirements list, because if the verification passes we know that those requirements work correctly, and because the list was generated by the client, we know that it is that he was expecting to get. And this is how we can say that our software is ready to be released.

2.4 State of the Art

So far we have reviewed different existing tools and methodologies that help us during the whole software development lifecycle to seek reliability in our software as a desirable property. The reality is that those tools and techniques exist independently, they require a high learning curve, and the integration is tough; all these issues represents a major challenge.

Formal verification, for example, is often used only in critical safety systems [54] like avionics [4], healthcare, automotive, transportation, but what happen with everyday systems, like web or mobile applications, of course the a human life is invaluable, and it justifies that critical safety systems must care about reliability, but it does mean that common-used applications don’t deserve to be dependable. We can also think that there exist a huge economic difference for each sector, and this is another reason why common-used applications only use dynamic verification (testing) as a parameter to determine its “reliability”, because the budget for this is fair enough to develop it, not to research in verification techniques, and formal methods, neither to spend time learning how to write formal specifications, or how to write logic rules for proving, or configuring complex frameworks to prove correctness, besides the time it requires. So, even though software is ubiquitous, dependability is not, at least not today. After all developers are there to produce software, apparently as fast as possible, today’s trend from clients is to require working systems in as less time as possible, and of course it requires some sacrifices, and the most often feature scarified is formal proof of correctness, but we also know that at long time it is more expensive to maintain and patch software, than to develop it only once, but correct.

Since Java, and its related tools have grown in such an open community, it is also hard to achieve some standardization, or integration, especially because there exist several small groups working in developing tools, but in general, they don't have any industrial support, or higher institutional support, some projects are started as master or PHD thesis, some tools are developed, but then they are abandoned, Some projects are good in their own domain, but the can hardly be integrated with other domains, all these problems clearly constitute another challenge.

There are also some software development methodologies, such as waterfall, prototyping, incremental development, spiral, rapid application development, agile development, among others. But no matter how different they, somehow they all seek to develop the software optimally, and they all, somehow are concern with requirements, design, develop and test, in greater or less degree. But there isn't any ultimate solution that can assure dependable software developed quickly and easily. And that is precisely what *Construct by Contract* intents to be, and this is what we will talk about in the next chapter.

In present time there isn't any IDE, or development tool that integrates all of the previously mentioned features in one seamless workspace, if it is true that such tools exist, it is also true that they work completely independent, which makes it harder, especially for developers and for IT industry in general, their adoption.

2.5 Chapter Summary

In this chapter we have introduced some of the existing tools that somehow help to improve reliability in software, tools that are good in their own domain. These tools will aid our solution because they can be chained together to achieve a better result, so far we have just looked at them individually, but their integration will be discussed in the following chapter.

Chapter 3. Construct by Contract: Solution Proposed

The solution, as already mentioned in the title of this thesis, is called "*Construct by Contract*" (CbC), during the following sections all the components of this solution will be described, in terms of some of the concepts we have already discussed in Chapter 2. We will show how each individual brick interacts with each other in order to create this methodology that seeks to develop dependable software systems. First we have to define what our solution is, we can describe it as follows:

> **Construct by Contract:** abbreviated as CbC by its English acronym. It is a methodology to develop everyday-used[1] software, which pursues reliability as main objective, by gathering different software engineering tools and techniques, starting from customer requirements shaped in the form of contracts, and propagating them throughout the entire software development lifecycle, in order to ensure correctness, and thus dependability.

Definition 1 Construct by Contract

It takes as principle the concept of *contract* defined by Bertrand Meyer in his book *Object-Oriented Software Construction* for the Eiffel programming language [29]. As we have already mentioned in the section Design by Contract in Chapter 2, the nature of Meyer's contracts establishes a direct connection between two classes; thus it is not possible to implement them in that way for CbC since we expect firstly to get contracts from the client's requirements, therefore a redefinition of contract that suits better to this methodology is needed. This definition must be able to deal with all stages of the software lifecycle, and must be acceptable for all the people involved in the mentioned process, such as customers, users, project managers, designers, programmers and testers. This enforce contracts to be part of the documentation in each stage. Now we will continue to describe contracts as required for our solution.

> **Contract:** is a collection of meaningful sentences that describe a singular task, its rules, its inputs and its outputs, in any given language according to the stage of the software's lifecycle where it is looked.

Definition 2 Contract

Through the following sections, we will explain how contracts are propagated from stage to stage of the software's lifecycle, being translated from one language to another. It is necessary to keep in mind that no matter what phase we are in, contracts should always express the same thing.

3.1 Principles of CbC

The main principles desirable in CbC are related with the seamless integration, which means that there should exist a natural flow from stage to stage of the software's lifecycle. Such features are defined as follow:

> **Propagation:** Is the property to generate a contract based on the client's requirements and map that contract to lower phases in the software development lifecycle.

Definition 3 Propagation

> **Reversibility:** Is the property to perform backtracking of contracts, in such a way that we can navigate from lower to upper phases in the software's lifecycle.

Definition 4 Reversibility

Persistency: Is the property of a contract to express exactly the same meaning for a task, no matter the phase of the software's lifecycle where it is observed.

Definition 5 Persistency

3.2 Software Development Lifecycle for CbC

So far we have talked about the different stages or phases in software development lifecycle, but we haven't formally defined them. The following diagram shows the structure of the section D.2 Software Engineering of the *1998 ACM Computing Classification System* [1].

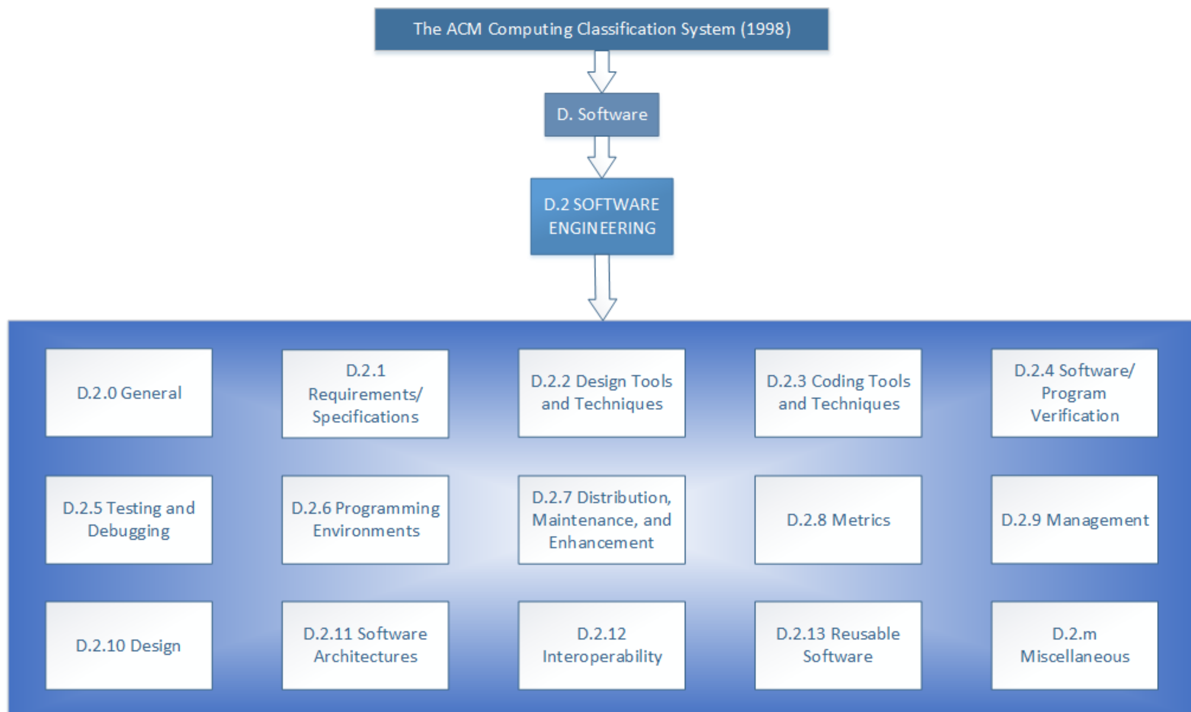

Figure 11 The ACM Computer Classification System (1998)

From the above classification, we will consider only in the following five categories, as part of software development lifecycle used in the solution proposed:

- D.2.1 Requirements/Specifications
- D.2.2 Design Tools and Techniques
- D.2.3 Coding Tools and Techniques
- D.2.4 Software/Program Verification
- D.2.5 Testing and Debugging

In the State of the Art section of the Related Work we mentioned the software development methodologies, we can easily notice that the sections chosen from the ACM classification, correspond somehow to the waterfall model [55]; not only that, but also these sections are recurrently present in all other methodologies in greater or less degree, thus by choosing these sections we are trying to propose a methodology that can serve to developers who used to work with the other development methodologies, no matter if it is incremental, spiral, rapid or agile, they all are related with these stages.

So finally we can define our stages for the software development lifecycle as follows:

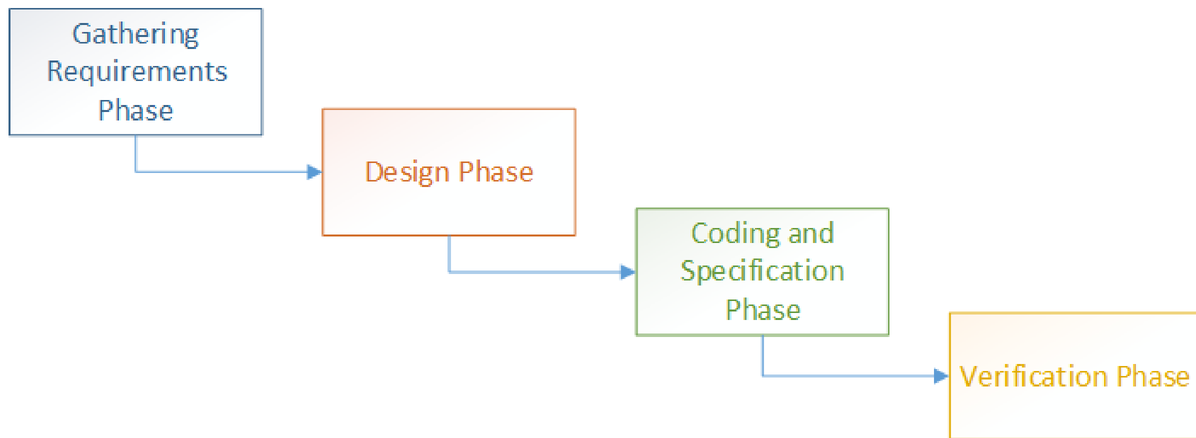

Figure 12 CbC Software Development Lifecycle

Even though this lifecycle seems to be lineal, it does not mean that it is inflexible, in conformance with agile manifesto [56], changes in requirements are allowed, as long as they are expressed in the right way, which means as contracts.

3.3 Roles involved in CbC

We have said that CbC is thought to be used whenever developing software considered as part of the daily life. Whether they are desktop, web applications or mobile applications, such applications are usually sponsored by a business that can be either a single person, like somebody offering his professional services, or a large company selling any product. Such kind of projects is developed by a wide range of entities, like freelance developers, IT consultancy companies, or software factories of any size. In this type of process, we can identify the following roles involved.

Client: Is the final owner of the software, and the one who pays for the project. He can establish the requirements partially or totally. He represents a company, community, or any kind of business.

Definition 6 Client

User: Is any person, who will use the system once it is completed. He can express requirements as well, partially or totally, according to the business.

Definition 7 User

Project Manager: Is the person who gather the requirements, therefore, serve as a bridge between the business and the IT people. His main responsibility is to coordinate people involved in the project, and to keep track of its progress and budget, as well as collect the documentation and deliver the final product.

Definition 8 Project Manager

Software Architect: Is the person who receives and analyse the requirements brought by the project manager and capture them in the design of the software. Usually a person with wide knowledge in software development, software engineering, pattern designs, etc. Its responsibility is to generate the design documentation, including diagrams, notes, etc.

Definition 9 Software Architect

Developer: Is the person who writes the code of the software based on the design detailed by the software architect. In general his responsibilities also include the unit testing and the integration testing.

Definition 10 Developer

Tester: Is the person who perform the system test based on the requirements and generates testing reports to validate these results.

Definition 11 Tester

Those roles are usually observed in the teams that develop the so called everyday-used software, it does not mean that there must be one specific person per role, but it just refers to the activities according to each phase of the software lifecycle. Thus, such roles can be performed by a group of people if we are talking about and IT factory, or by one individual if we are talking about a freelancer.

3.4 Specification Languages according to CbC

Once that we have defined our software development lifecycle, we can proceed to explain how contracts will be expressed in accordance with the principles defined in the first section of the present chapter.

For each phase, we can use a specific language to express the same contract, conserving the principle of persistency defined previously. The table of Languages mapped to phases is as follow.

Phase	Language
Gathering Requirement Phase	Natural Language (e.g. English or Spanish)
Design Phase	Design Language (e.g. UML+OCL or BON)
Specification and Coding Phase	Specification and Programming Language (e.g. JML+Java or Spec#+C#)
Static and formal verification Phase	Logic Formulae (e.g. Propositional Logic, Dynamic Logic, Hoare Logic, Temporal Logic)

Table 1 Specification Languages per Phase

3.5 Conceptual Structure of CbC

In the previous sections, we have provided some definitions for the Software Development Lifecycle for CbC, the Roles involved in CbC and the different Specification Languages according to CbC that will be used on it, but we haven't given any concrete relationship within such concepts. The backbone of CbC is the software development lifecycle and then everything is related to that as it can be seen in the following diagram.

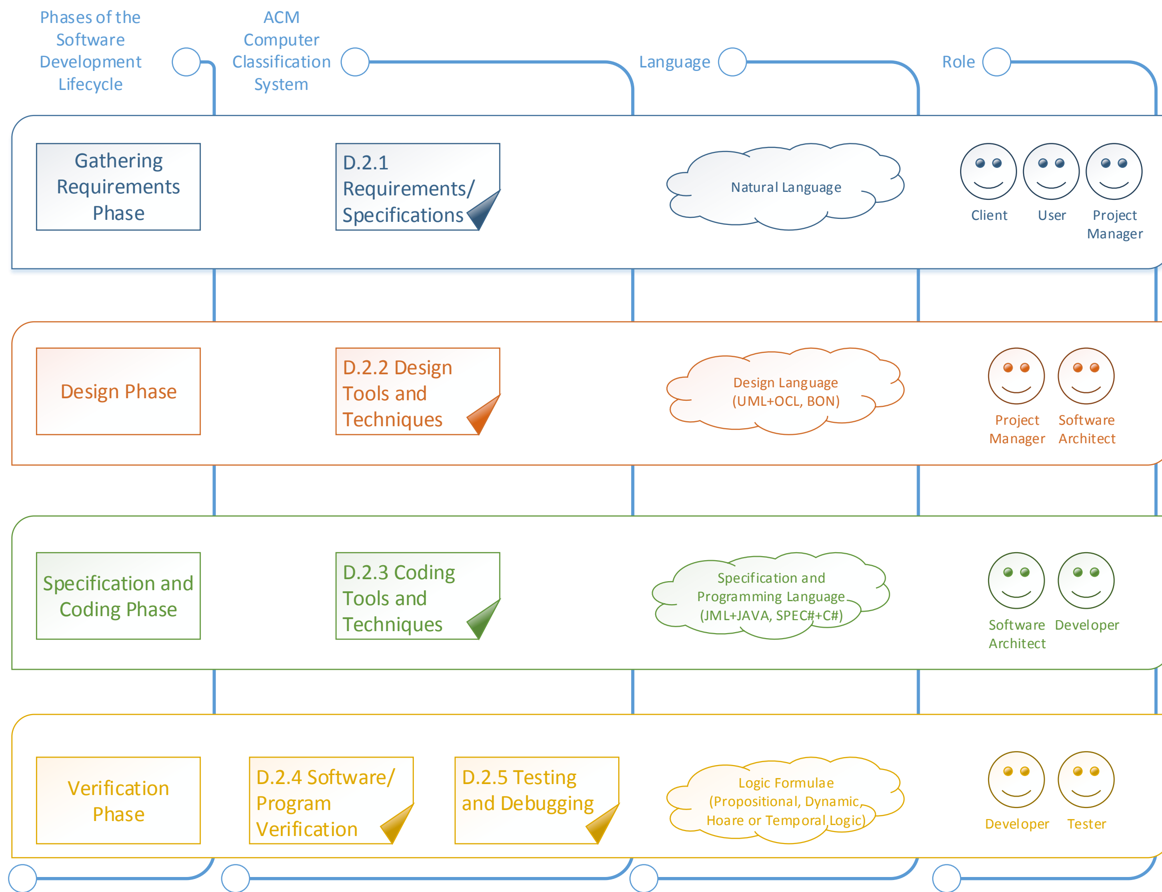

Figure 13 Conceptual Structure of Construct by Contract

This diagram shows a general overview of each phase in the software development lifecycle of CbC; it is worth to mention that this is a high level definition for each phase, the specific definitions for them will be explained in the following section.

3.6 Tool Support for CbC

The ultimate goal of CbC, besides being a methodology to develop dependable software systems is to provide all the features already described in one single tool, and generate one single and seamless integrated workspace where all the software development lifecycle can be reached and automated, where contracts are persisted and reversed, where code can be written and verified, where all this happen in one single place in a semiautomatic way; a workspace where clients, project manager, software architects, developers and testers can interact, and review the information they require about the software according to their needs and their understanding level, one centralized workspace where client and project managers can review contracts in natural language while software architect and developers can review the OCL and UML diagrams, where developers and testers can explore the code and its specifications. Only one tool, one workspace to construct reliable software due to contracts accomplishment making life easy because it can automate tasks such like contract translations, diagrams generation, test cases generations, tests execution, progress tracking, bugs predictions, contracts verification, and automatic generation of documentation. Because of the time available

this tool won't be built as part of this research, but its design and implementation will be explored in Chapter 4, after proving our concept in Chapter 5.

3.7 Functionality per Phase

During the following sections, we will explain the conceptual functionality of our solution in each phase of software development lifecycle, detailing the meaning of the Figure 13 Conceptual Structure of Construct by Contract.

3.7.1 Gathering Requirement Phase for CbC

The gathering of requirements is the starting point where the base contracts must be generated since requirements establish the overall functionality of the application. In Design by Contract, Meyer suggest that a contract establishes obligations and guarantees for a piece of code in relation with another one. In Construct by Contract, contracts are seen as client's expectations for a system; hence these contracts are made within people. Thus, the main actors involved in this phase are client, user and project manager.

Because this approach seeks to serve as documentation through the whole project's lifecycle, the requirements must be written down in the way of contract that can be understood by both actors at this phase, client and project manager.

Construct by Contract suggest that the client should be free to express whatever he wants, he also should be able to communicate his need to the project manager as clear as possible, and the simplest way to achieve this goal is by using common sentences in natural language.

Because the goal of Construct by Contract is to persists the contracts in every stage of the software, it is necessary to have a structure that can be propagated to the lower levels of the lifecycle, this is the reason why an additional effort is required from the client because he is expected to deliver his requirements, as far as possible, in terms of simple sentences. The definition of simple sentence is as follow.

> Simple Sentences: Are those sentences that consist in only one action (verb) that is performed on subjects (nouns) by actors (nouns) e.g. "The student reads a book" or "the system computes the taxes".

Definition 12 Simple Sentence

Specifications are not limited only to one simple sentence per method; in fact, the client does not need to know about methods, he just need to express his needs in a list of simple sentences. These sentences can be linked within each other, it means that one sentence can produce, or be produced by another sentence. E.g. "Every time the dog barks, the parrots fly to the tree" in this sentence we can easily identify two sentences: "the dog barks" (sentence A) and "the parrots fly to the tree" (sentence B) in this case the relation is that the sentence A is the cause of sentence B to happen, or sentence B happens because of Sentence A. In this way, the list of requirements can be done by adding simple sentences and relationships among them.

Many of the software projects fail in meeting their times and costs because of poor specifications [12], hence before try to prove our software is correct, we need to know what the software is expected to do as clear as we can. It is also true that very often

clients don't know what they truly want, but the "simple sentences technique" explained before, may help clients to understand their own needs as they have to think how to explain them in simple sentences, after all it was Albert Einstein himself who said "if you can't explain it simply, you don't understand it well enough".

Trivial example

One trivial example of contract-like requirement expression might be the specification for a calculator. The requirements for such method are as follows:

The software is a calculator
The software must add two numbers
The software must subtract two numbers
The software must multiply two numbers
The software must divide two numbers
The numbers must be integers
The result of any operation must be an integer

In this way, we have specified our software in natural language using simple sentences. A real example of this can be found in the Proof of Concept shown in Chapter 5.

3.7.2 Design Phase for CbC

In the section 3.2 we proposed the Software Development Lifecycle for CbC, after gathering the requirements, it is needed to design how software is going to be built, and how its components will interact within each other, this is part of the best practices of a software engineer, and thus is necessary to develop dependable software systems. There are many tools and techniques to design software out there, we have design patterns, model-driven designs, functional-based design, embedded systems design, and a bit less know and less used Design by Contract, just to mention some.

We have talked about Design by Contract (section 2.3.2), because is the base used to develop the Construct by Contract methodology, at some point, the second one, might be seen as an extension of the first one. Before going any further, it is necessary to explain a bit more what Design by Contract is, and how it is related to our methodology.

Design by Contract takes the concepts of class and its relationship with its clients as a formal agreement, expressing each party's rights and obligations [29]. These obligations are based, for methods, in Hoare Logic, from where we know that given a true safe state {P} we can achieve the state {Q} by executing a statement S. The safe state {P} is known as the precondition of a method and indicates the obligations that a client calling the method should observe before calling it, in order to get the expected result established in the postcondition {Q}, i.e. If a client accomplishes {P} then the supplier must ensure {Q}. Meyer also provides the concept of class invariant, they are restrictions imposed on class's fields, and establish rules that have to be true for any object of the class [29].

Once that we have rescued the basic concepts of Design by Contract, we will explain how they fit together within Construct by Contract. For Meyer, contracts are established between classes, but so far, we haven't talked about classes at all; thus the significant next step is to map somehow the requirements gathered from the human client in the

first stage, to classes and methods specifications, in the form of preconditions, post conditions and class invariants.

When we are talking about designing systems we can come to Object Oriented Programming because its nature allow us to work with analogies between the real world and the software, it is decidedly common/easy to think in objects and classes from the real world, and then integrate them in a system; also, Design by Contract is thought to work with Object Oriented paradigm as Construct by Contract is. We have already defined the perspective that will be used to design and construct our software, now we can come to see the tools and techniques that will help us to shape the design. UML [16] has been the most common technique to represent Object Oriented Software, because it helps to draw and explain in a simple way how classes interact within each other.

Ideally in Construct by Contract, there must exist a direct translation from the simple sentences-based contract specification in natural language to the OCL constrains annotated in a UML class diagram, as a result of this translation the Software Architect can work with the design of the software and keep the requirements as contracts according to the principles of propagation, reversibility and persistency.

3.7.3 Specification and Coding Phase for CbC

The coding should not be different to any normal coding process since contracts must not have side effects in the functionality of the program. At this point, contract should only be seen as "code comments", thus is easy to imagine contracts not only as a tool to prove the correctness of a system, but also as a way to document code, and to explain explicitly and in a standardized way, what a method is intended to do whether for the same programmer to remember what was he doing, or to new programmers getting involved in the project.

We have already mentioned that contracts should be written in code as code comments, but it doesn't mean that we can write them as any programmer want to; instead we will opt for any standard and formal technique depending of the software that we are developing, any programming language can have their own specification language, like in the case of Java we can talk about of JML and in the case of C# we can talk about Spec#. At the end, the decision of that specification language use will be close related with the programming language we are using to build the application. Since Construct by Contract is thought as a generic methodology to develop dependable software, we won't limit the definition to any determined specification language, but in the Chapter 5 and Chapter 4 we will focus in a JML as a specific tool for a concrete case.

CbC seeks to ensures that all requirements are covered, then, we can evaluate if there exist any contract that is expressed in the requirements phase, and that does not have a correspondence with the coding phase, it would mean that we are missing some functionality. If all contracts listed in the requirements phase are present in our code and contract annotations then we are accomplishing the CbC principles, and we can be happy that we are doing everything that has required by the client.

Nevertheless it is possible to add additional contracts in the code since the requirements phase contracts only express client requirements, we need to take particular considerations in the software architecture, and coding, especially when considering best practices for programming, like the use of pattern designs, or a layered model, and the pursuit of reutilization or abstraction or encapsulation, usually while coding software,

for one requirement only we can produce many different classes and methods; therefore, all these classes and methods must be specified independently to the general contract that specifies the client requirement.

3.7.4 Verification Phase for CbC

In this section, we will explain the main features sought by CbC in the overall Verification Phase. This phase will be divided into two smaller phases, the static and formal verification and the dynamic verification, the first one is responsible for the proof of correctness of our software while the second one is responsible for the testing of our program.

3.7.4.1 Static and Formal Verification

During the previous sections we have translated our natural language requirements into OCL conditions attached to a UML class diagram, then we have realized such specifications in any defined programming and specification language according to the required platform so we could say that the software is finally constructed, by using contracts. It is not enough, our next task is to prove that actually such contracts are really accomplished with the actual implementation. This is the main goal of this phase, static and formal verification.

Ideally, during this stage neither programmer, nor anybody else should perform any other activity, but to see the results of the verification. It is a proposition of CbC that once we have written our code and our contracts we can automatically prove that the resulting software is correct with the minimum effort. Therefore, what should happen here is that the programmer should see if is there any place where the implementation of the code does not suit its contracts if the answer is yes, then the programmer should analyse his implementation and fix the issue, if the answer is not, it means that the software Is correct respect to its specifications, which means that all contracts are accomplished, which means that all requirements are fulfilled.

During this process, we must pay particular attention to the contracts of the interfaces that match the client's requirements, and to the additional contracts that belong to the actual implementation, even though they don't belong directly to any specific requirement. So far we can ensure that our program actually does what it is expected to do in normal conditions, sadly, there are always some circumstances that we cannot control, so we can find some abnormal circumstances, the control of them must be implemented by the programmer in the coding and specification phase.

3.7.4.2 Dynamic Verification

In the previous section, we discussed how to verify that the software does what it is expected to do, at least under normal circumstances, but now we need to think in the additional cases, and the other cases that cannot be covered with the formal and the static verification this is where dynamic verification takes place.

Here, CbC offers different alternatives, the first one is to generate automatically test cases from the contract annotations, the other one is to generate test cases based on potential bug sources. Ideally CbC must be able to identify and generate automatically such tests cases.

3.8 Chapter Summary

The present chapter has purely defined our generic solution, and the concepts it involves, the following chapters will look at its implementation in more detail, by showing the design of a tool to support it, and by developing a proof of concept based in an industrial piece of software.

Chapter 4. Construct by Contract Tool Design

In the previous chapter, we have defined what Construct by Contract is, and how it should be used to develop any system based on contracts; thus it can be seen like a generic methodology to develop dependable software systems. In the section 3.6 we mentioned that this methodology is expected to have the support of a tool that can realize the methodology, nowadays such a tool does not exist, and because of the time we dispose it is not possible to build it as part of this project, but we will provide the design of the tool. Because we are talking about something more concrete in this tool, we will have to decide for any given programming language, to be considered the main kind of software to be developed with such implementation. Because our experience is greater with Java technology, and because of the wide collection of tools available, we will focus the design of the tool to be built on the Java platform, and to produce Java Software. It does not mean that the methodology is limited; indeed, the methodology allow us to develop the supporting tool for any programming language, such Python, C#, PHP, Java, etc. The decisive point will be the tools and knowledge available about each concept used in the definition of the solution.

The Construct by Contract tool must integrate an interface to gather requirements from clients and users in natural language using the simple sentence strategy, and then by using artificial intelligence and natural language processing identify design elements like classes, methods and fields in order to generate automatically, at least, UML class diagrams with OCL annotations, this diagrams can be used by the software architect to integrate the complete design of the system. Later this diagrams and annotations will be translated to java interfaces with JML specifications, as preconditions, postconditions and class invariants, and will be presented to the development who will later have to implement such interfaces with the actual design of the software, and as the code is written, the tool will be able to evaluate how much the implementation match the contracts of the specification in real time, so the developer can be sure that its implementation is correct. Once all the implementation is done and verified in real time the tester can access to the same tool and request the automatic generation of test cases, based on the contracts, JML annotations and use cases coming from contracts, as well the tool will provide one interface to generate additional test cases manually to ensure all possible cases are covered. Finally, the tool should be able to pack the software in the proper distribution format, either .ear, or .jar o .apk depending on the kind of application is being developed, and it will also be able to generate a suite of printable documentation for the software.

As we can notice, the goal of this tool is to integrate all the basic Software Engineering principles in one single place, and mould them to work around contracts. Once again, even though this tool proposes initially a workflow based on the waterfall model, we can clearly notice how this flow can be easily applied to any other software development methodology.

Finally, we must recognize that in order to build this tool it is necessary either to polish existing tools for each specific tasks, in order to integrate them in one common environment, or to generate a complete new suite of tools that can natively interact

within each other. This task require a considerable amount of time and knowledge to de done, so the construction of such tool is not done as part of this project, but it can be seen like a PhD proposal. This discussion will be continued deeply in the Future Work section.

4.1 Use Cases for CbC Tool

This section presents in the form of UML use cases, the functionality that must be expected from the CBC tool.

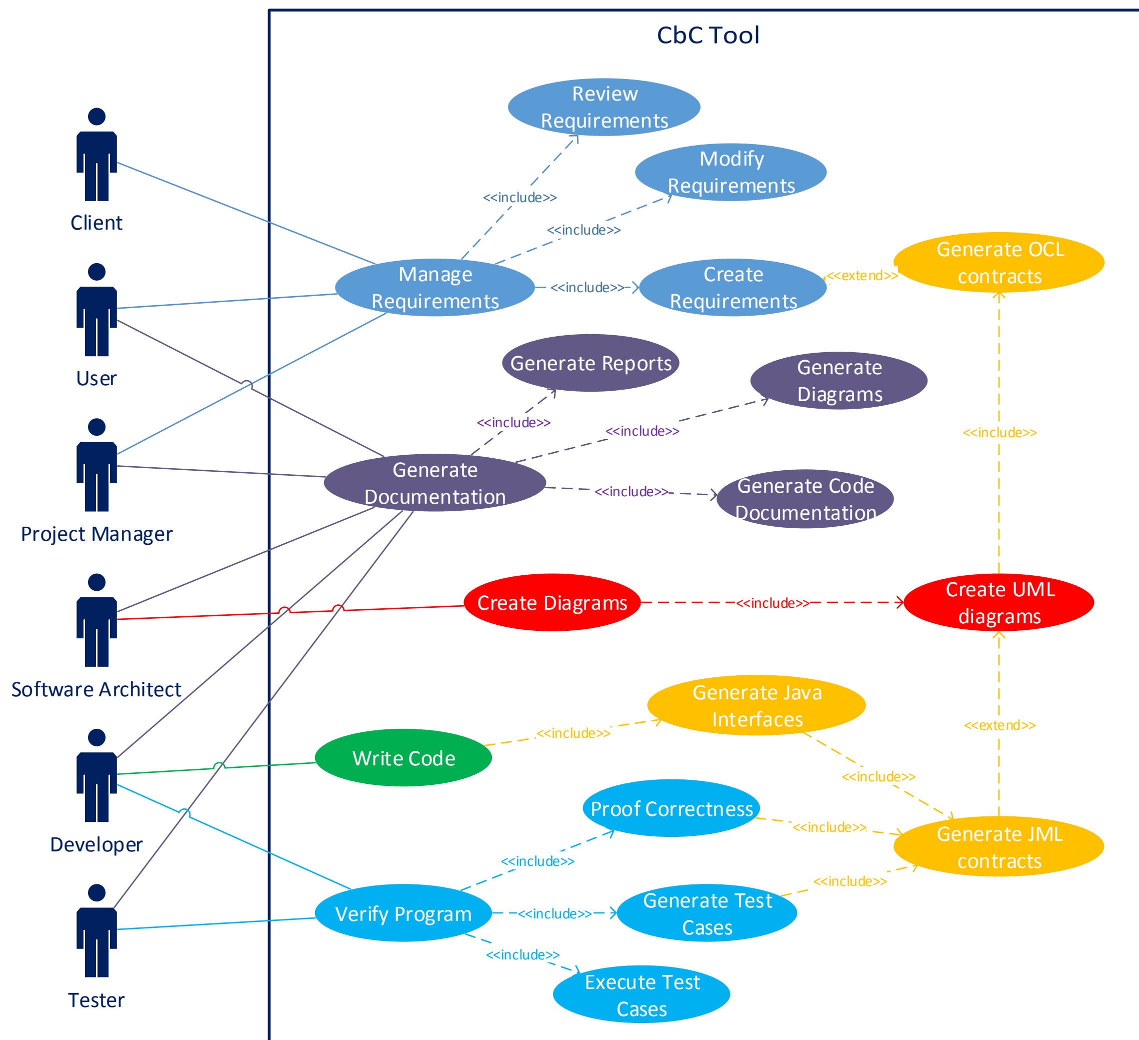

Figure 14 UML Use Cases for CbC Tool

In the previous diagram, we can easily notice how the CbC tool provides different facilities to each of the characters defined in the section 3.3. We can see how the client, user and project manager are directly linked with the requirements specification process, and how the tool provides an interface to deal with this situation. Then the tool automatically should be able to generate all the translations form Natural Language, to UML+OCL as models of the software that can be improved manually by the Software Architect, and how this model is translated into JML annotations accompanying Java interfaces, that later will be implemented by the developer; finally the tester can access in the same tool to all the resources previously collected in order to deal with the testing phase. An important feature of this tool is that, as seen in the diagram above, all the

characters are able to generate documentation from the tool, which is necessary to understand, and maintain the software.

4.2 Sequence Diagram to develop Software with CbC Tool

The following diagram explains the most general process to generate dependable software, supported by the tool. Due to its sequence structure, we can easily visualize the phases of the software development lifecycle explained in section 3.2. In this diagram, we explain how each of the actors defined in our methodology perform any specific activity (input) to any of the elements of the tool. We also see how each element generates a response message either to another element or to an actor. We can also appreciate an iterative process done by the programmers and testers, this process will be broken once we can prove that our software is correct, and once it has passed all the tests. We can see how developers interact with the code editor, and how they get feedback from the verification tool, in order to validate both the JML specifications and the Java code, thus we can prove the correctness in our software in real-time and automatically. We can also appreciate how testers make use of the JML annotations to generate test cases, and how they provide their own test cases throughout the code editor, then they execute such tests cases, complementing the correctness proof done by the developer and the verification tool. Testers also give feedback to developers in order to correct any bug, and how once all tests are passed the tester reports it to the project manager. At this moment, the developer can release the software, by giving it to the Project Manager who will close the legal/financial contract with the client.

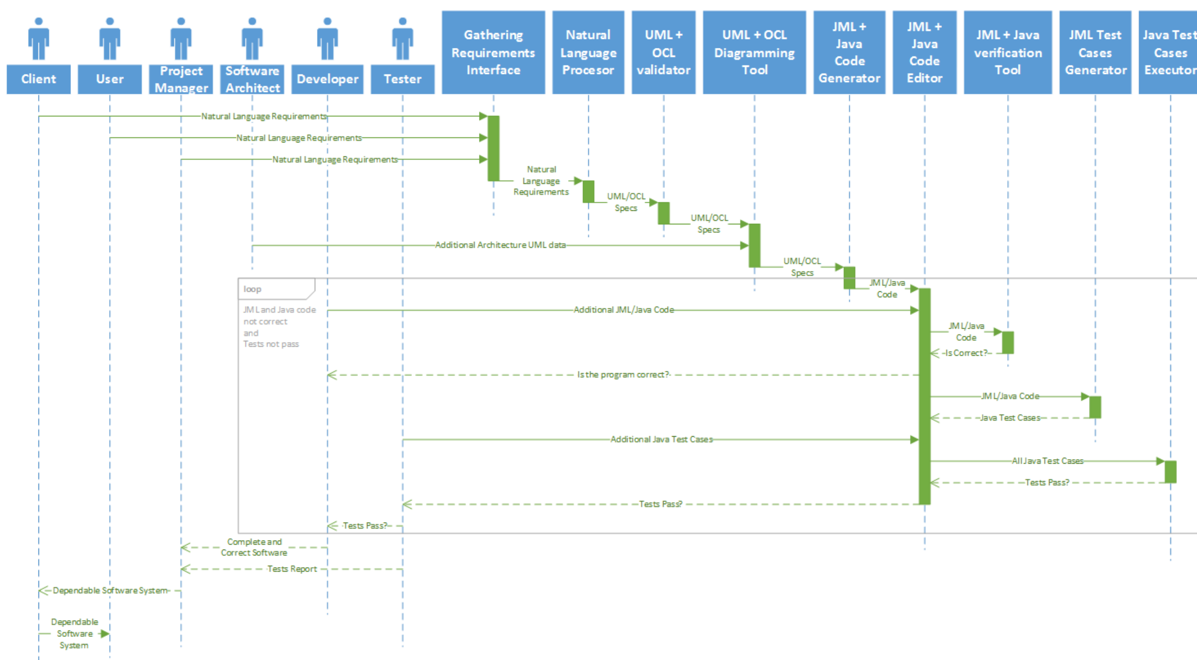

Figure 15 Sequence diagram to develop dependable software systems with the CbC tool

4.3 Architecture Diagram for CbC Tool

The following architecture diagram, is indeed, an adaptation of the general architecture diagram of the solution found in section 3.5. In this case, we clearly specify the scope of our proposed tool to be develop for and in Java platform.

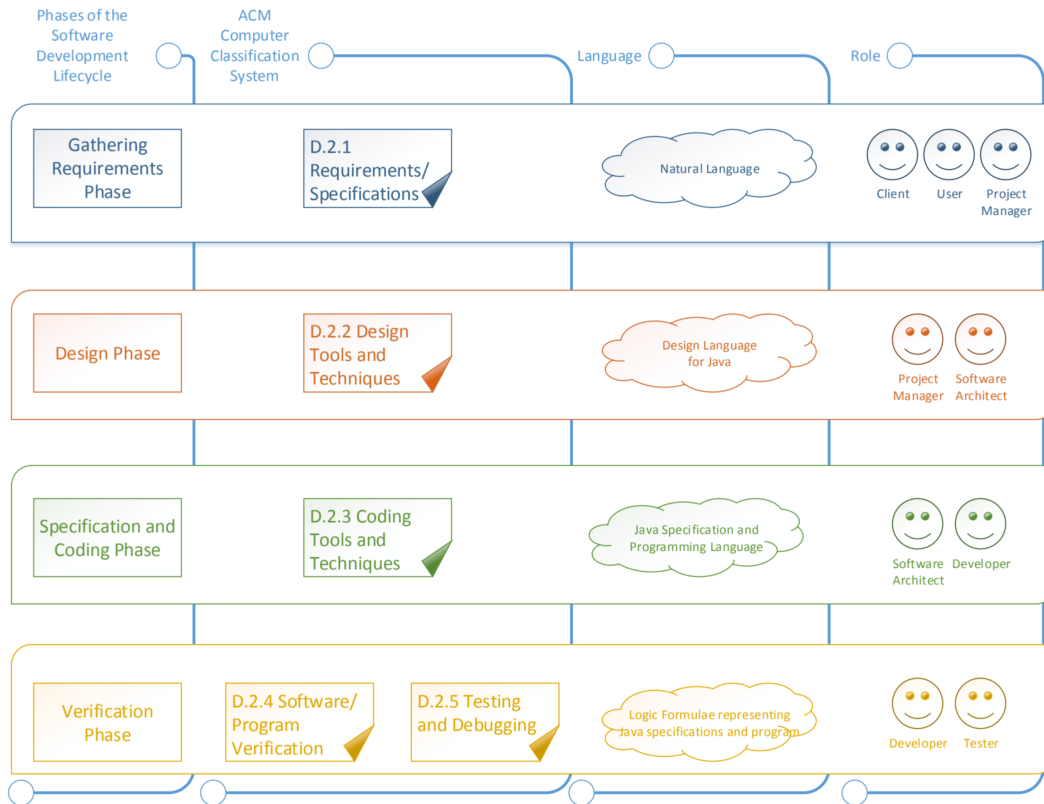

Figure 16 Architecture Diagram for CbC Tool

4.4 Components Selection for CbC Tool

In this section we will justify the components chosen to be part of the CbC tool, each component is related with a given specific task and has been chosen because of its compatibility with the CbC methodology. It is worth to mention that the selection of components is made in the Java context; thus we have explored individual elements that allow us to develop Java dependable software systems. Nevertheless, it does not mean that the same methodology is limited, but just that the time available for this project was not enough to go any further.

4.4.1 Workspace

We have decided to work in the Eclipse IDE as base workspace where all tools are expected to be integrated. This is because Eclipse has a vast support from the Open Source community and because it is pretty easy to deal with the additions to the base IDE by means of plugins, additionally we count on the Eclipse Marketplace, which is a repository where different tools can be added to the IDE. Thus, Eclipse is a strong option to achieve integration and compatibility, despite the fact that is an excellent platform to develop Java software.

4.4.2 Requirements

For this phase the CbC tool should provide not only the interface to write the specifications, but ideally a Multilanguage support, with options like customized dictionaries to include new words, especially those specific to the software domain, it

should also provide some drag and drop aid to build the simple sentences in terms of noun and verb blocks, additional to a free-text space where client can just write the requirements as a normal text document.

Because we seek to achieve seamless integration in one workspace, we have opt to use the Eclipse default text editor, in order to list our requirements in our simple sentences proposed in section 3.7.1. The advantage of this is that we can keep our requirements document exactly in the same workspace, and we can actually open it, and modify it in the same IDE. As part of the CbC tool, the text editor might be extended to a requirements editor, including the features mentioned in the previous paragraph.

4.4.3 Design

In this phase, the CbC tool should be able to generate java code from the UML class diagram, and JML annotations from the OCL constrains, additionally any change in the Java code or the JML annotations should be reflected in the UML diagram and the OCL constrains.

Based on the principle of integration, we searched for a design tool that can be compatible with the Eclipse IDE, in the marketplace we found many options, but most of them are not free. Based on the cost factor we decided to work with UMLet, a free design software that can be run either standalone or integrated in same IDE. Additionally this tool has the advantage of generating class diagrams from the java code, this property supports the reversibility principle sought by CbC.

4.4.4 Coding

We have already talked about the Eclipse IDE as a platform for integration, but it also provides a lot of advantages in the programming stage, like the code generation, code navigation, code versioning, code managing, and code revision in real time. Eclipse also provides tools to manage web servers and databases, to design user interfaces and web interfaces. For this reason, Eclipse has been chosen as the coding tool. A final comment is related to the programming language, which, as already mention, is Java.

4.4.5 Verification

Now we need to decide what existing tool will be used to perform static and formal verification, in order to write JML specifications on the code, and then verify the relationship, all in real time. The list of possible tools is as follow:

- Krakatoa
- ESC/JAVA2
- ModernJass
- JStar
- Key

Many other tools were found, but most of them are just started projects, or just code repositories without enough documentation to be considered for this methodology. Once we have found the candidate tools we will evaluate each of them, in order to identify which one suits better for the purposes of CbC, this evaluation considers only parameters that are relevant to our methodology. The results of such evaluation are shown in the following table.

	Krakatoa	ESC/JAVA2	ModernJass	JStar	KeY
Active Developed	No	No	No	No	Yes
IDE integration	Own	Eclipse and Own (Mobius)	Half Eclipse	Eclipse	Eclipse and Own
Platform	Linux	Linux, Mac OS X, Windows	Linux, Mac OS X, Windows	Linux	Linux, Mac OS X, Windows
Java version	1.6+	1.4	1.5+	1.6+	1.5+
Specification	KML	BON, JML	Java Notations	JML	JML
Cost per licence	Free	Free	Free	Free	Free
Documentation	Papers, Tutorial, Installation Manual	Papers, Tutorial, Installation Manual, Videos	QuickTour	Papers, Tutorial, Installation Manual, Videos	Papers, Tutorial, Installation Manual, Book
SMT support	Alt-Ergo, CVC3, E-prover, Gappa, Simplify, SPASS, Vampire, veriT, Yices, Z3	CVC3, Simplify, Z3	none	general theorem prover	Simplify, KeYmaera
Checking	Compile Time	Compile and Real Time	Runtime	Compile and Real Time	Compile and Real Time

Table 2 Comparison of Verification Tools

Based on the previous evaluation we can firstly dismiss ESC/JAVA2 because the supported java version is 1.4 which is a terribly old version of Java. Then we can dismiss Modern Jass, because it is not based in JML notations as CbC establishes and because the checking is executed at runtime. Krakatoa and JStart will be dismissed because based on the personal experience the installation process is complicated enough to consume time, from configuring the appropriate Linux environment to gather all required dependencies in the right version and compile the source to generate binaries, besides that, Krakatoa does not use pure JML, it uses KML instead, and JStar requires the specifications to be thought in terms of separation logic, which requires an additional learning curve. After all this evaluations, the remaining tool is KeY, which is actually the most supported tool, and the only one who is active developed because it has support from the Karlsruhe Institute of Technology, the Chalmers University of Technology and the Technische Universität Darmstadt, KeY also provides JML specifications validation in real time, and integration with the Eclipse IDE.

4.4.6 Testing

For the testing task we have considered two branches, the first one is the manual test generation using JUnit for its execution, this decision was made because JUnit is a native tool of the Eclipse IDE, so no further installation or configuration is required. For the automatic process of generating test cases from the JML specification, the options were almost null, and the only one that was useful enough to be used as part of the CbC tool was JMLUnitNG, this tool allow us to generate java test cases directly from the JML annotations, it requires only two .jar files to generate such cases, the we can normally compile and execute the cases, so we can again integrate them into the Eclipse IDE.

4.5 Components Diagram for CbC Tool

The following diagram explains how all the components described in the previous section, will interact within each other to achieve the construction of java dependable software.

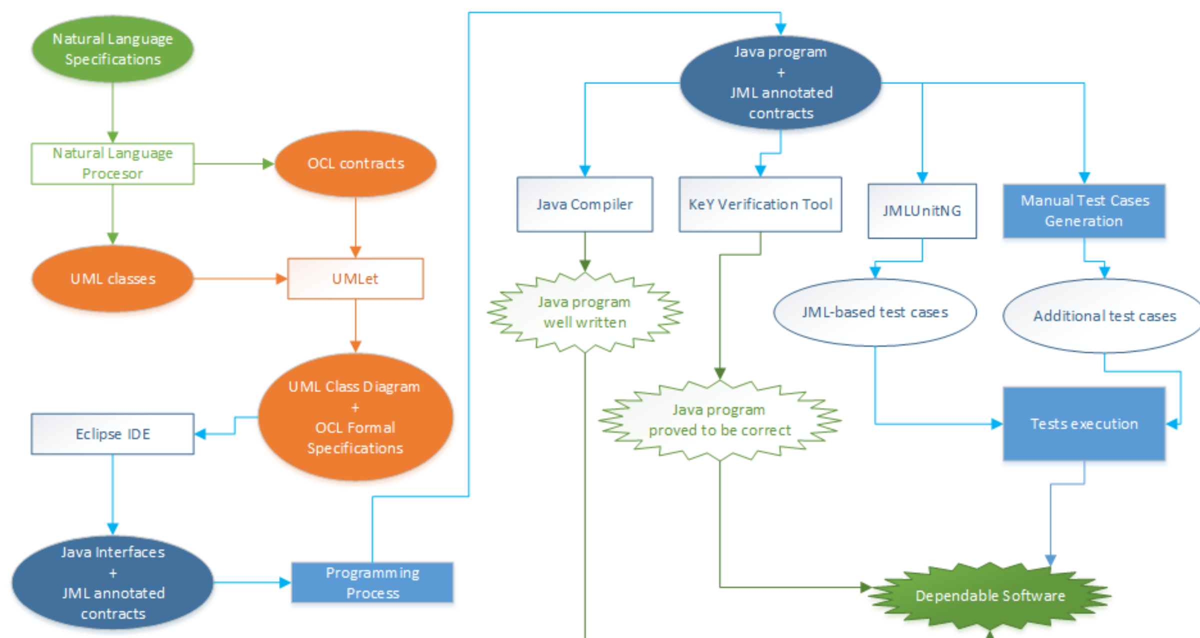

Figure 17 Components Diagram for CbC tool

4.6 Chapter Summary

In this chapter we have presented the design of the tool that will support the Construct by Contract methodology, we explained what functionality should be included in the tool, and what would be the normal flow to achieve dependable software systems. We have also listed the components of such implementation in the domain of Java software, and the interaction of this component along the software development lifecycle defined in the section 3.2. In the following chapter, we will show how this methodology can be applied, without the tool, and by means of the existing individual tools, in an industrial piece of software.

Chapter 5. Proof of Concept

In this chapter, we will implement the Construct by Contract methodology described in Chapter 3 in an every-used application, of the type of Web Applications. The goal of this proof is to show hoe the methodology is expected to work when putting together the existing tools.

5.1 Industrial Case Study

Because of the time available to develop this dissertation, the solution will be bounded to a dynamic web application developed in Java, using JSPs, servlets, and the MVC model implemented within Struts framework. For instance, the definitions and specifications of CbC, have been defined in general terms, but the proof of concept will show only how the methodology can be applied on this specific application.

The web application already exists by itself as a result of a project done by myself as a freelancer, the name of the application is BluenetsWeb and belongs to Bluenets, a Mexican company established in Toluca, State of Mexico. The company own a building for renting offices and meeting rooms, and provide a wide range of services for SMEs, such as virtual offices, mail addressing, calls reception, corporate image, and so on.

The web application BluenetsWeb contains the main explanatory information about the company and the services it offers. It also contains an access to social contents like Facebook and Twitter, and provide a contact form for users, it also allows to contact the support centre via live chat.

The main website of the application can be accessed from the following URL: http://www.bluenets.com.mx. The actual homepage looks as follows:

Figure 18 BluenetsWeb main page

One of the main features of the web application is the ability to calculate prices for renting either offices or meeting rooms, based on a set of allowed configurations according to the properties of rooms available, such as type of room, capacity and services included. This functionally is the one that will be explored within the CbC methodology as proof of concept, we'll focus on the process for specifying, designing, implementing, verifying and testing requirements for calculating the price for renting either an office or a meeting room.

In the proof of concept we will not deal with the specification of the web interface, avoiding html code and graphic elements such buttons, labels, text fields, option lists, etc. Instead we will look at the business logic behind the computing of prices.

5.2 Gathering Requirements Phase

The original requirements were exposed in a Microsoft Excel spreadsheet as follows:

	A	B	C	D	E	F
1	**RENTA SALAS DE JUNTAS**					
2						
3	**Descripción**	**Alternativas**	**Precio**	**Descuento**	**Descuento**	**Descuento 3**
4				**5-10 Horas**	**11-19 horas**	**20 en adelante**
5				**30%**	**40%**	**50%**
6	Espacio	Sala de Juntas 20P	500.00			
7	Horario hábil /no hábil	No hábil	750.00			
8	Formación	Colaboración				
9	Capacidad	20				
10	Personas	20				
11	Horas	2				
12	Días	1				
13	Total horas	2	1,500.00	1,050.00	900.00	750.00
14	Precio por hora		750.00	525.00	450.00	375.00
15	Fechas					
16	Descuento Cliente con contrato?	SI	150.00	105.00	90.00	75.00
17	**Total**		**######**	**945.00**	**810.00**	**675.00**
18	**Depósito**		**675.00**	**472.50**	**405.00**	**337.50**
19	**Diferencia**		**675.00**	**472.50**	**405.00**	**337.50**
20						
21	**Hora adicional horario hábil**		**975.00**	**682.50**	**585.00**	**487.50**
22	**Hora adicional horario no hábil**		**######**	**787.50**	**675.00**	**562.50**
23	**Promociones:**					
24	**10% de descuento a clientes con contrato**					
25	**30% al contratar de 10 a 19 horas**					
26	**40% de descuento al contratar de 20-29 horas**					
27	**50% de descuento al contratar de 30 en adelante horas**					

Figure 19 Original Requirements Specification

As proof of concept, our task will be to show how to specify real requirements for the web application detailed in the previous section, using the simple sentences methodology presented in the section 3.7.1 for the Gathering Requirement Phase for CbC. Thus, we will translate the functionality depicted in Figure 19 Original

Requirements Specification into simple sentences requirements. The result of this translation is as follows

- Every room has a class from a set of classes available
- Every room has a unique code composed by literals and numbers
- Every room has a location
- Every room has a maximum capacity in number of persons
- Every room has a name
- Every room has a type
- Every room has a description
- Every room has a unit
- Every room has a service
- Every room has a status
- Every room has a unitary price with two decimals
- Every room has a modality
- Every room has a billing concept
- Meeting rooms are rented for hours per day
- Meeting rooms are rented for number of days
- Meeting room is a kind of room
- Pricing for meeting room depends on the number of persons required
- Pricing for meeting room depends on the number of hours per day required
- Pricing for meeting room depends on the number of days required
- Internet service is free for meeting rooms
- Internet service is free for meeting rooms
- Whiteboard is included for meeting rooms
- Furniture is included for meeting rooms
- TV-Screen is an additional cost for meeting rooms
- Projector is an additional cost for meeting rooms
- Coffee service is included for meeting rooms
- Coffee service is offered every two hours
- The pricing of an additional hour for meeting room will be 30% of the original price per hour
- If the meeting room is rented from 5 to 10 hours, the price will get a 30% discount on the final price.
- If the meeting room is rented from 11 to 19 hours, the price will get a 40% discount on the final price.
- If the meeting room is rented for more than 20 hours, the price will get a 50% discount on the final price.

The simple sentences mentioned above as requirements, will be the base for our contracts, so before keep going on, it is essential to establish where and how this sentences should be built in this ideal framework. CbC should provide a tool suite that allows clients to build these simple sentences either in an aided drag and drop tool, where user can create dictionaries with domain-specific nouns, and actions, and then link each of them in an interactive way, or in a free text editor, where client can write down his expectations of the system, based as well in simple sentences. Here, the challenge exists in having a powerful-enough Natural Language Processor that can deal with the structure of the simple sentences.

The requirements related with the pricing process, and those that will be used in our proof of the concept, are listed above. So now we are ready to proceed with the next stage: the design.

5.3 Design Phase

The first step to map requirements to class-based design is to identify all the nouns appearing in the requirements of the previous section. Initially every noun can be thought as a potential class; thus we need to list all the nouns found in the requirements as we'll see, some nouns are repeated, so it is necessary to have a control on the frequency of appearance for each noun, the main idea underneath is that the highest frequency of appearance for a noun, the highest possibility to identify such as a class. To keep tracking the proof of concept, we will list the nouns appearing in the section 5.2.

- Meeting rooms are rented for hours per day
- Meeting rooms are rented for number of days
- Meeting room is a kind of room
- Pricing for meeting room depends on the number of persons required
- Pricing for meeting room depends on the number of hours per day required
- Pricing for meeting room depends on the number of days required
- Internet service is free for meeting rooms
- Internet service is free for meeting rooms
- Whiteboard is included for meeting rooms
- Furniture is included for meeting rooms
- TV-Screen is an additional cost for meeting rooms
- Projector is an additional cost for meeting rooms
- Coffee service is included for meeting rooms
- Coffee service is offered every two hours
- If the meeting room is rented from 5 to 10 hours, the price will get a 30% discount on the final price.
- If the meeting room is rented from 11 to 19 hours, the price will get a 40% discount on the final price.
- If the meeting room is rented for more than 20 hours, the price will get a 50% discount on the final price.

As it can be seen above, not all nouns are straightforward, some of them are complex nouns, or compound nouns as "final price", and even some of them could be highly specific to the business domain; thus it brings a new challenge here, to automatically map requirements, even in simple sentences to a model of the system. In this particular case, we will have to trust in the criteria of the Software Designer, based on the information he has gotten after the Project Manager has gathered client's requirements.

This is a desirable goal of CbC tool suite that once all requirements have being collected through the requirements interface, they can be processed automatically into classes, and perhaps into design diagrams in an automated way. This of course is another of the future challenges for CbC implementation, depending still in Natural Language Processing tools that can understand a set of requirements as properties (classes, relationships, fields, inheritance, composition, aggregation) of a design, for instance a UML design.

Once that our classes have been identified, we can proceed with the design of the system. As mentioned in the previous section, this web application is based on the Model-View-Controller (MVC) [57] software architecture pattern. This pattern is widely used especially in, but not limited to, Web Applications. The basic pattern design for MVC is illustrated in the diagram below

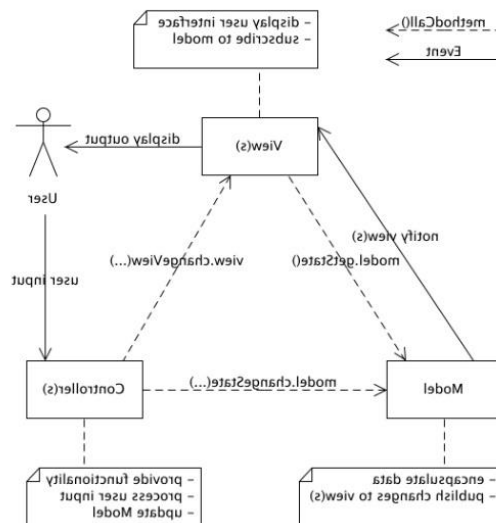

Figure 20 MVC Pattern [57]

The reason to use MVC patter is that this patter allow us clearly to identify each component based on its functionality, either if it is a web view, an object of the model or of the problem domain, or a business logic component. In Construct by Contract our contract are focused in the business logic rules, so with MVC we can easily focus only in the needed section, after all we haven't designed CbC to specify user interfaces, but to specify business rules and model objects.

Because we don't possess the appropriate tool to perform this task automatically, we will have to do it manually as it is usually done. The results of this manual process will be shown in the class diagram below, the only classes considered in this sections are those used in the business logic; hence we will omit the information coming from the Web Interface section.

Below is shown the class diagram involving the model and the controller tiers, but also the contracts. In order to map the contracts we need first to identify the type of each requirement, to see which can be treated as contract. The UML class diagram was done in the Eclipse IDE with the help of the UMLet tool mentioned in the section 4.4.3. This tool is not particularly powerful in terms of style, but at least it provides integration with the IDE.

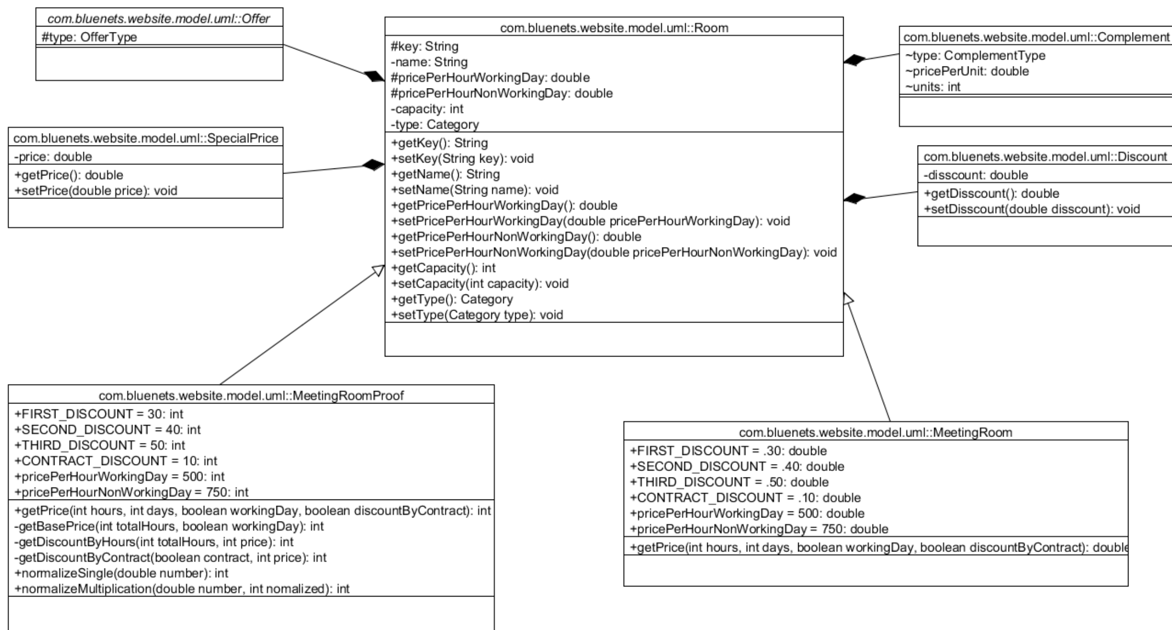

Figure 21 UML class diagram for BluenetsWeb

Now that we have the class diagram, we can freely add contracts to it, and its classes, thus the next step is to define the right format to express our contracts in the best suitable way to this stage. The requirements phase is an interaction between clients and project managers, the design stage should represent and interaction between project managers and software architects. Thus, the representation of contract can be a bit more technical, using some specific rules for notation. In this stage, we will use the OCL language to write the contracts related to our class diagram. Ideally we would add OCL constraints directly into the diagram, but because of the limitations of UMLet, we will write the OCL specifications below.

```
context MeetingRoom:getPrice(hours: int, days: int, workingDay: boolean,
discountByContract: boolean): double
pre validHours: hours>0

context MeetingRoom:getPrice(hours: int, days: int, workingDay: boolean,
discountByContract: boolean): double
pre validDays: days>0

context MeetingRoom:getPrice(hours: int, days: int, workingDay: boolean,
discountByContract: boolean): double
post : (hours*days>=5 && hours*days<=10 && workingDay &&
discountByContract)
==>
        (\result == (
        ((hours*days)*pricePerHourWorkingDay)-
        (((hours*days)*pricePerHourWorkingDay)*FIRST_DISCOUNT)-
        ((((hours*days)*pricePerHourWorkingDay)-
(((hours*days)*pricePerHourWorkingDay)*FIRST_DISCOUNT))*CONTRACT_DISCOUNT
)))
```

Figure 22 OCL constrains for BluenetsWeb

5.4 Specification and Coding Phase

So now that we have decided to work with the KeY tool before continue with the coding stage, it was necessary to prepare the development environment, even though CbC tool expects to provide its own workspace, and unified development framework, we don't possess that tool now, so we worked with the existing tools. One of the most used Integrated Development Environment for Java Software is the Eclipse IDE, since we are working with a Dynamic Web Application we will use the J2EE developer version, and because of compatibility with the KeY tool we will use the Indigo release.

After installing the workspace with the Web Server add-ons, the SNV controls and the KeY verification plugins with the Simplify SMT, our workspace is ready to be used. The following image shows the basic configuration of the Eclipse IDE with the features mentioned.

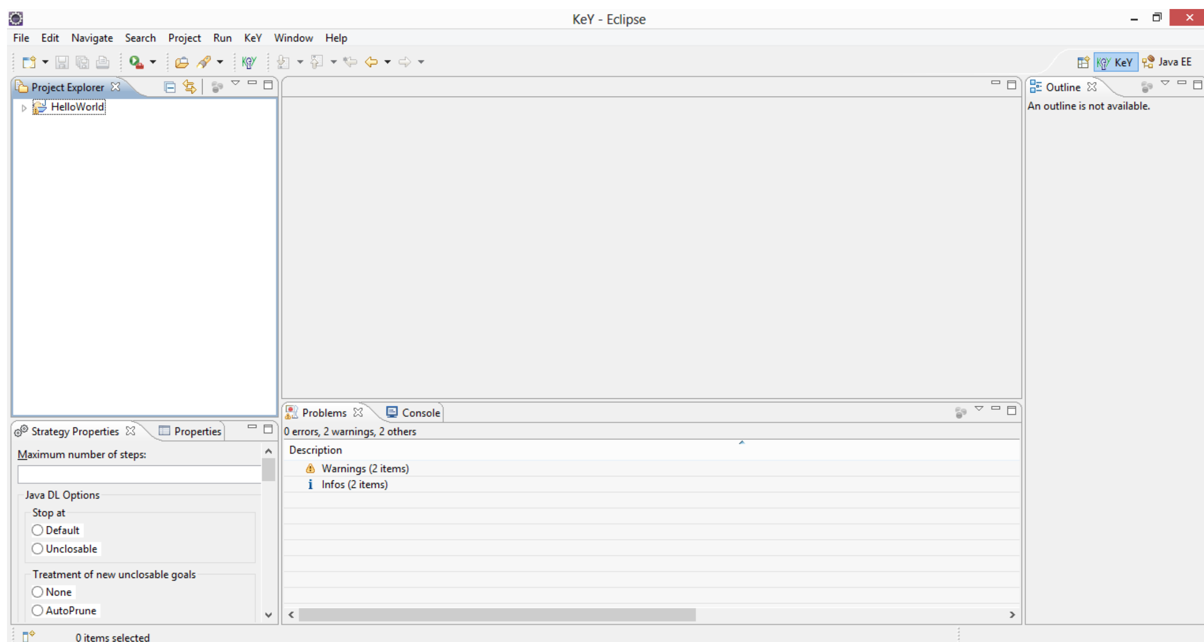

Figure 23 The Eclipse workspace for KeY

As we can see the in the Figure 1, the workspace includes the KeY feature, it provides one of the advantages sought in CbC that is real time feedback and unity integration in a seamless way.

Now we proceed to write down the code of the application, we won't show all the code, but only snippets of the sections related with the method used in this proof of concept.

```
public double getPrice(boolean workingDay, int hours, int days, boolean
discountByContract){
		double price=0.0;
		double totalHours=hours*days;

		if(workingDay){
			price=totalHours*pricePerHourWorkingDay;
		}else{
			price=totalHours*pricePerHourNonWorkingDay;
		}

		if(hours*days>=5 && hours*days<=10){
			price=price-(price*FIRST_DISCOUNT);
		}else if(hours*days>=11 && hours*days<=19){
			price=price-(price*SECOND_DISCOUNT);
		}else if(hours*days>=20){
			price=price-(price*THIRD_DISCOUNT);
		}

		if(discountByContract){
			price=price-(price*CONTRACT_DISCOUNT);
		}

		return price;
	}
```

Figure 24 Java code for the pricing method

The code above shows the actual functionality of the method in the working version of the BluenetsWeb. Since the implementation of the requirements already exists, for this proof of concept we will write the contracts directly on the code, having as result the following segment of code. Note that only a small part of the JML specification is shown here for spatial reasons.

```
/*@
	@public public normal_behavior
	@requires hours>0;
	@ensures (workingDay && hours*days>=5 && hours*days<=10 &&
discountByContract)==>(\result==((hours*days)*pricePerHourWorkingDay)-
((hours*days*pricePerHourWorkingDay)*FIRST_DISCOUNT)-
(((hours*days*pricePerHourWorkingDay)*FIRST_DISCOUNT)-CONTRACT_DISCOUNT));
	@ensures (workingDay && hours*days>=5 && hours*days<=10 &&
!discountByContract)==>(\result=(hours*days*pricePerHourWorkingDay)-
((hours*days*pricePerHourWorkingDay)*FIRST_DISCOUNT));
	@ensures (workingDay && hours*days>=11 && hours*days<=19 &&
. . .
	@*/
	public double getPrice(boolean workingDay, int hours, int days,
boolean discountByContract){
		double price=0.0;
		double totalHours=hours*days;

		if(workingDay){
			price=totalHours*pricePerHourWorkingDay;
		}else{
			price=totalHours*pricePerHourNonWorkingDay;
		}

	. . .
	}
```

Figure 25 JML specification for pricing method

5.5 Verification Phase

5.5.1 Static and Formal Verification Phase

So far we have defined the preconditions and postconditions for our method, now we can see in real time the results of the verification process performed by KeY.

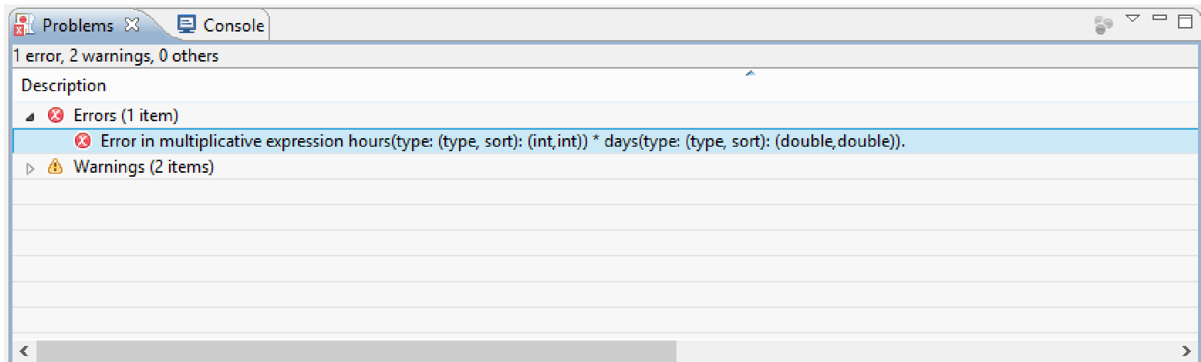

Figure 26 KeY verification results

As we can see here, the Key verification tools show us one error, it is because in the JML specification, we are trying to reduce the discount by using some floating point numbers, they Key Tool does not have support for such numbers, and that's the reason for the error we are getting.

In order to solve this issue we have had to adapt the method by dividing it three sub method that can be easily verified, this division can be seen in the class diagram of the Figure 21 UML class diagram for BluenetsWeb, specifically in the class MeetingRoomProof where we have created the following methods with their respective JML specifications:

```
/*@
        @public normal_behavior
        @requires totalHours>0;
        @ensures (workingDay==true) ==>
(\result==totalHours*pricePerHourWorkingDay*100);
        @ensures (workingDay==false) ==>
(\result==totalHours*pricePerHourNonWorkingDay*100);
        @*/
       private int getBasePrice(int totalHours, boolean workingDay){
               int basePrice=0;
               if(workingDay){
                       basePrice=totalHours*pricePerHourWorkingDay*100;
               }else{
                       basePrice=totalHours*pricePerHourNonWorkingDay*100;
               }
               return basePrice;
       }
```

Figure 27 Java code and JML specification for getBasePrice method

```
/*@
        @public normal_behavior
        @requires totalHours>0;
        @requires price>0;
        @ensures (totalHours<5) ==> (\result==0);
        @ensures (totalHours>=5 && totalHours<=10) ==>
(\result==price*FIRST_DISCOUNT/100);
        @ensures (totalHours>=11 && totalHours<=19) ==>
(\result==price*SECOND_DISCOUNT/100);
        @ensures (totalHours>=20) ==> (\result==price*THIRD_DISCOUNT/100);
        @*/
       private int getDiscountByHours(int totalHours, int price){
               int discount=0;
               if(totalHours>=5 && totalHours<=10){
                       discount=price*FIRST_DISCOUNT/100;
               }else if(totalHours>=11 && totalHours<=19){
                       discount=price*SECOND_DISCOUNT/100;
               }else if(totalHours>=20){
                       discount=price*THIRD_DISCOUNT/100;
               }
               return discount;
       }
```

Figure 28 Java code and JML specification for getDiscountByHours method

```
/*@
        @public normal_behavior
        @requires price>0;
        @ensures (contract==false) ==> (\result==0);
        @ensures (contract==true) ==> (\result==price*CONTRACT_DISCOUNT/100);
        @*/
       private int getDiscountByContract(boolean contract, int price){
               int discount = 0;
               if(contract){
                       discount=price*CONTRACT_DISCOUNT/100;
               }
               return discount;
       }
```

Figure 29 Java code and JML specification for getDiscountByContract method

By doing this division we can verify with key the three individual methods getting as result the following acceptations for the verification.

```
Proof closed: /BluenetsWebsite/Proofs/com_bluenets_website_model_integration_MeetingRoomProof[com_bluenets_website_model_integration_MeetingRoomProof_getBasePrice(int,boolean)]_JML_normal_behavior_operation_contr
Proof closed: /BluenetsWebsite/Proofs/com_bluenets_website_model_integration_MeetingRoomProof[com_bluenets_website_model_integration_MeetingRoomProof_getDiscountByContract(boolean,int)]_JML_normal_behavior_opera
Proof closed: /BluenetsWebsite/Proofs/com_bluenets_website_model_integration_MeetingRoomProof[com_bluenets_website_model_integration_MeetingRoomProof_getDiscountByHours(int,int)]_JML_normal_behavior_operation_co
```

Figure 30 Proof of correctness for pricing method

Now we have proof that at least these methods are correct, it means that their implementation matches its specification.

5.5.2 Dynamic Verification

This proof of concept is small enough to be completed with the proof of correctness, Nevertheless, in order to complete the methodology we will generate some test cases based on the JML annotations by using the JMLUnitNG tool. To do so, we need to run

the JMLUnitNG jar by passing the JML annotated classes as parameters, once it is done we will get the following java files containing the test cases generated.

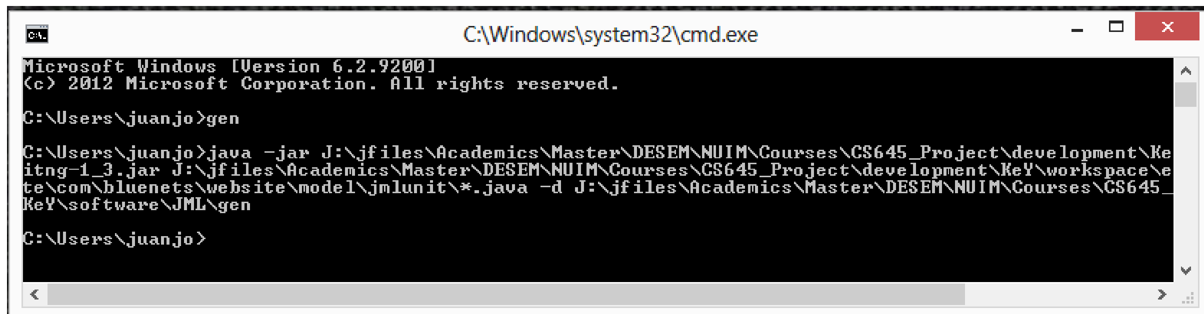

Figure 31 Execution of JMLUnitNG

```
public class
getPrice__int_hours__int_days__boolean_workingDay__boolean_discountByContract__0__d
ays
  extends ClassStrategy_int {
  /**
   * @return local-scope values for parameter
   *  "int days".
   */
  public RepeatedAccessIterator<?> localValues() {
        return new ObjectArrayIterator<Object>
        (new Object[]
         { /* add local-scope int values or generators here */ });
  }
}
```

Figure 32 Test Case generated by JMLUnitng

The last steps consist only in in compile and execute the test cases generated, every class contains a main method so each class can be executed individually as a test case.

5.6 Chapter Summary

At this point, we have shown how we can implement the Construct by Contract methodology in an industrial piece of software, by chaining a collection of individual tools, even though we don't have the CbC tool implemented, which in general will automate many of the manual processes done in this proof of concept.

Chapter 6. Evaluations

In this chapter, we will offer a collection of evaluations, including the evaluation of the proposed solution, the design of the tool to support the solution, the proof of concept and the existing work. Such evaluations will show a critical analysis of the results obtained.

6.1 Evaluation of the Proposed Solution

In this section, we will evaluate our proposed solution Construct by Contract. Such evaluation will be done by considering the coverage of the Aims and Objectives listed in the section 1.4. In order to proceed with this evaluation, we will take each bullet of the already mentioned section,1.4 and we will question if every bullet was accomplished.

Did we generate an ideal methodology that brings formal verification and software correctness into everyday-used software?
Yes, we did. We proposed Construct by Contract as an ideal methodology which application means directly to introduce formal verification and proof of correctness in any kind of software. As we have explained in Chapter 3 this methodology is ideal because it assumes the integration of tools, phases and characters can be seamless done in one single workspace. We also built our solution as a generic methodology, which means that can be applied to any programming language, and to any kind of application.

Does this ideal methodology can be used by myself and other people?
Yes, it does. After the description we have provided, we have presented the methodology to some colleagues that work with different domains (e.g. Programming languages), they could understand what was the methodology about, and its benefits, so they agree that such methodology could be used if it provides any tools support that requires no additional effort in the normal development lifecycle.

Does this methodology propagate contracts in each stage of the software development lifecycle?
Yes, it does. In section 3.5, we have presented a conceptual diagram that shows how contract must be carried from one to another phase in the software development lifecycle.

Did we identify basic stages in the software development lifecycle, and the actors/roles involved?
Yes, we did. In section 3.2, we presented the Software Development Lifecycle that has been considered for our methodology while in section 3.3 we introduced the roles that can be played for the different characters involved in such lifecycle. In section 3.5, we showed the interactions of each role within each phase. We also proposed that these phases and roles don't need to be followed strictly, they can be flexible enough to be introduced in any software development methodology, such iterative or agile development. We also clarified that each role does not represent one specific person since only one person can cover different roles, according to the extension of the project and the working team.

Did we define a methodology to expand Design by Contract through the software development lifecycle?
Yes, we did. As we mentioned in section 3.7.2 we took the base concept of Design by Contract [29] and we extended it to become more than just a design strategy, so it can

be transformed in a methodology that goes from the gathering requirements phase to the verification phase.

Did we define how contracts must be introduced and presented in each phase?
Yes, we did. In section 3.4, we presented the different languages that can be used in order to present and propagate contracts in each phase of the development lifecycle.

Did we propose tools that can be adopted to achieve the goal of using contracts in each stage?
Yes, we did. In Chapter 2, we discussed the existing tools that, independently, can help us to implement our methodology the building software process. We presented this information in general topics, not mapped in the specific phases of the CbC software development lifecycle, but the relation is clear once we have introduced such lifecycle in the section 3.2.

Did we show how the methodology might work in a specific application with a specific domain?
Yes, we did. In the Chapter 5 we showed how we applied our methodology to an industrial case study as proof of concept, such demonstration has the support of the company Bluenets, and the implementation of the different tools discussed in Chapter 2, to achieve most of the features proposed in Chapter 3. When we say most of the features, we mean that even though Construct by Contract claims to perform automatically the translation of contracts to each different specification language, so far it was not possible to do this, thus the translation of contracts had to be done manually.

Did we show a comparison of exiting tools to verify software and choose the one that fits the best the methodology on the specific application designated?
Yes, we did, in Chapter 2 we listed existing tools and their respective features, and in concrete, in section 4.4 we explained the reason to choose each of the components for the design of the tool that can support our methodology.

Did we identify and summarize the challenges in building a seamless unified workspace that supports the methodology?
Yes, we did, but these challenges will be presented in the following section where we will talk about the evaluation of the design of the tool to support CbC.

6.2 Evaluation of the CbC Tool Design

In this section, we will evaluate the design of the tool that will support CbC against the definition of the methodology as an ideal solution. We will consider the tools mentioned in the Chapter 2 and how they can be put together into the construction of such tool. In order to complete this evaluation, we will response a set of questions that compare the solution with the tool designed.

Does the tool design covers all phases of the software development lifecycle described as part of the methodology in section 3.2?
The answer is yes; indeed we can directly compare the Conceptual Structure Diagram of section 3.5 labelled as Figure 13 Conceptual Structure of Construct by Contract with the Figure 16 Architecture Diagram for CbC Tool. So we can see that the architecture of the tool perfectly matches the structure of the methodology.

Does the tool design support all the roles described by the methodology in the section 3.3?

Yes, it does. In the Figure 14 UML Use Cases for CbC Tool, we can see how each of the roles can perform different activities in the same tool, according to their specific tasks defined in the solution. Additionally in Figure 15 Sequence diagram to develop dependable software systems with the CbC tool we can see how each of the characters interact with the components of the tool in order to achieve the construction of dependable software by means of the tool.

Does the tool design consider all the principles and features of Construct by Contract?

Sadly we can't answer yes to this question; the reason is that although the tool provides the means to perform propagation, reversibility and persistency of contracts, this design is not a generic design since it is built only to develop Java software. This limitation is due to the tools considered especially in the Specification and Coding Phase and in the Verification Phase, in order to achieve a concrete design of this tool, we had to choose the underlying tools we expect to work with; thus we bound it only to the Java domain, scarifying the generality of the methodology. Nevertheless, this design can be reproduced to any other platform (programming language) just by using the correct tools.

Is the construction of the designed tool viable?

In the design of the tool, we have presented the components that should integrate such tool, these tools already exists; thus the effort should be paid in three stages to achieve the construction of such tool. The first stage is the integration stage, in the design we established Eclipse as the base IDE for the integration, so all tools listed should be able to be plugged to this IDE; thus some additions are required. Once we have achieved the integration we can make an effort in the automation because now we have all of the tools in one single space, then we need to make them interact by themselves, in order to achieve the properties of CbC: propagation, reversibility and persistency. If we have achieved integration and automation, the last effort must be done in the competition of the tools, we are aware that the existing tools are defined by a specific domain and that they are not thought to be part of the CbC methodology since it is new, thus we need to add the missing parts to complete the ultimate goal of CbC. In conclusion, the construction of the tool is viable if the needed effort is paid.

6.3 Evaluation of the Proof of Concept

In this section, we will evaluate the proof of concept developed against the solution we have proposed. In order to produce this evaluation, we will explore the results phase by phase of the software development lifecycle.

6.3.1 Gathering Requirements Phase

During the Gathering Requirements Phase, we proposed to express all requirements in terms of simple sentences, during the elaboration of the proof of concept we needed to convert the original requirements from the excel document to simple sentences. The process implied that we must first interpret the original file, which actually was in Spanish, then we had to understand it in English, then we were able to express the requirements in terms of simple sentences, just as CbC requires, so we can say that our proof of concept accomplishes the general functionality for this phase.

Even though we achieve the goal of the methodology for this phase, we have identified some challenges. We have to deal with the expressive power of the simple sentences, which should be able to capture any requirement, despite the Natural Language they are expressed (English, Spanish, Chinese, etc.). Here is where we can justify the need of a tool like the one proposed in the section 4.4.2, with an interface that can provide support for Multilanguage specifications.

6.3.2 Design Phase

For our proof of concept, we were able to generate manually the class diagram of the BluenetsWeb application in the UMLet tool, and generate a document with the OCL constraints for such diagram so we could achieve the goal of the phase according to the software development lifecycle proposed by CbC in section 3.7.2. All this as a result of understanding the simple sentences gathered from the previous phase.

Because eclipse is an open community some partial options could be found, but the most complete ones required a payment, and the less complete were only diagramming tools (UMLet), without validation nor automatic code generation, such tools also have a limited interface, making it more difficult to produce such diagrams.

Even though reaching the goal is good enough, we must highlight that both the translation process and the diagramming process was done manually, and then it requires time and is susceptible to errors. Had we had the CbC tool described in section 4.4.3, we would be able to generate such diagram and constraints automatically.

6.3.3 Specification and Coding Phase

CbC proposed to translate JML into OCL specifications in only one automated environment, but because OCL and UML class diagrams had to be done manually with external tools, it was not possible to achieve such automation and integration; therefore, JML specifications were done manually directly into the code written in the Eclipse IDE.

One notable observation took place here, the method to calculate the cost of a meeting room could not be correctly specified because whenever we talk about prices, we must consider such prices as floating point numbers, floats in the case of Java, and KeY as the chosen tool to perform the verification does not support operations with floating-point numbers. For this reason we had to find a workaround for the implementation of such method, this workaround was described in section 4.4.5.

Once again, the goal of the phase prescribed by the methodology in the section 3.7.3 was achieved, but the process is neither propagative, nor reversible and thus not persistent, then we are missing the properties of CbC. In order to accomplish with such properties we may address the issue and solve it in the CbC tool as described in the previous cheaper.

6.3.4 Verification Phase

We already mention one of the problems risen form the specification phase, in relation with the limitations of KeY, despite that, we can say that KeY suits perfectly the CbC methodology because it can be easily integrated in the Eclipse IDE as a plugin, and it can perform verification in real time, so as we write our code from our JML specification we can see that it actually is correct. For it, we can say that Key is a suitable tool to be

integrated into the CbC development tool. For the case of floating point number three options can be considered, either generate a new verification tool or extend KeY or use dynamic verification in such methods.

Finally we can talk a little about the tool we finally used, Key, which is quite simple to use, either with the Eclipse plugin or with the standalone installation, the interface is remarkably clear, and it requires a small learning curve, its documentation is vast and complete, and the project is actively developed and has some strong support, overall it seems to be the perfect too, expect that it happens that it has some limitations, like when dealing with floating-point numbers, which are not supported by KeY. Money is an extremely common example from real life, and it is always represented with the use of floating-point numbers; therefore, again, such limitation represents a serious challenge of the roadmap.

6.3.5 Summary of the Section

We have proof that we can implement the CbC methodology in an industrial piece of software, which is enough, but we have also highlighted the problems of implementing the methodology with the existing tools, and this is the main reason to assure that the CbC tool is required to complete the CbC methodology and bring it into the software development process.

6.4 Evaluation of the Existing Work

There have been some other intents to approach this problem, one of them is the Mobius Program Verification Environment [58], which is a proposal that integrates an extension of the BON language discussed in section 2.2.2 in its own IDE, nevertheless such intent has been abandoned and the last delta release is dated as November 28th 2008, the main differences is that this tool as followed the use of BON instead of UML+OCL notations, and it does not support Natural Language Specification as CbC does, therefore one step of the requirements phase is still missing.

In contrast, KeY, is actively developed and supported, and in his roadmap it has included some additional features as the automation of the OCL-JML translation, feature that suits perfectly into the CbC methodology. Despite that, KeY will require the Natural Language Processing and the UML automatic generation from it, which means that KeY would work together with CbC to achieve the ultimate goal, additional work is required in KeY to fulfil CbC specification, in terms of support for floating-point numbers and automatic test generation.

So we can say that even though there exist, and have existed, some similar approaches, none of them has been completed, and none of them covers entirely what CbC proposes, thus can define a roadmap for the future, in order to complete the CbC tool.

6.5 Chapter Summary

Construct by Contract, indeed has identified basic stages in the software's lifecycle, and the actors involved, we have also explained how these roles and phases can be adapted in different software development methodologies, from waterfall model to agile development. Our methodology has successfully given an extended definition of contracts, basing on the principle of Design by Contract and adapting it to every phase

of the software development lifecycle. And we have covered all the objectives proposed in the section 1.4.

We have evaluated the design of the CbC tool, which should include all phases of the software development lifecycle and all the roles proposed by our solution, it should also integrate the existing tools in one common workspace and should be able to automate some translating tasks. We have evaluated this tool in terms of how it suits the solution proposal, and how viable it can be, and we have established that by having enough resources, this tool cab be constructed to support Construct by Contract.

We have also evaluated the proof of concept against CbC, and we have demonstrated that it is possible to adopt the methodology, but in the actual condition, without any specific tool it can be really tough, so our proof of concept can be seen as well as a motivation to support the CbC tool.

Chapter 7. Conclusions

7.1 Summary

Along this thesis, we have introduced the problem, when dependability and formal verification, in the present, is not a “native” feature of the software, especially in everyday-used software.

Then we researched over the different tools and methodologies that somehow bring reliability into software systems, and that might play a role in the solution of the problem.

After, we proposed Construct by Contract as a generic methodology to develop dependable software systems, by putting contracts in the backbone of the well-defined software development lifecycle, listing the roles and techniques involved in the process.

Then, we proposed the design of a tool to support our methodology based on the research done over the existing tools and techniques individually applied in software engineering to enhance reliability, and how they can be integrated in an unified development workspace, we also showed how this tool is related to each of the phases defined in the solution proposal, and how each of the characters can interact with such tool, all this by means of use case diagram, sequence diagram, architecture diagram and components diagram.

We also presented a proof of concept based on an industrial case study to show how the methodology can be nowadays applied, at least, but not limited to the Java Web Applications domains. Such proof of concept provides an illustration of the use of the independent tools and the implementation of the CbC methodology.

Finally, we evaluated our solution proposal from Chapter 3, the design of the tool that can be developed to support the CbC methodology from Chapter 4, the proof of concept from Chapter 5 and the existing work.

In summary this is what this work is all about, all this with only one goal, to develop dependable software systems.

7.2 Problems Found

In this section, we will summarize the problems found when dealing with the proof of concept. Because Construct by Contract is an ideal methodology that should be supported by a tool, it is important to highlight that it is because these problems exist that it is worth to dedicate some effort and resources, like economical and timing resources to the solution proposed.

Regarding the gathering requirements phase, in CbC we expect the client to be able to express his requirements in terms of simple sentences, but it is initially a huge challenge for non-trivial, and for more complex systems, naturally the client must perform an additional effort to express himself in terms so small and simple blocks of sentences, it is true though that it may help him to understand better what he truly wants, but, it is still an additional effort, we have not evaluated in this proposal if is there any set of requirements that cannot be expressed in terms of simple sentences, perhaps this proposition can be evaluated as some of the historical computational problems proposed by the Hilbert’s Program; thus we can open a new whole research area on how clients,

and in general human being can express their need without any additional effort in a computable way.

Assuming that we can bypass the previous challenges, and we can automatically generate our UML+OCL designs, then life would be easier since actually there are some active developments that search to translate OCL constrains to JML notations, and we are already capable of generating code from UML class diagrams, and vice versa. Then the step from Design phase to implementation phase would not represent a significant challenge, and due to we already have tools that can verify JML contracts within Java code, and generate test cases automatically, it would not mean a major obstacle to complete the software's lifecycle as CbC propose it.

Not everything in the garden is rosy, and it happens that the integration of existing tools into one common environment represents a huge challenge as well. First because the rhythm of development is very different due to the open nature of Java, many tools are created, and many tools are abandoned very often, and the community is so open, and has so few support that it is genuinely tough to achieve a state where all the tools can converge into one common goal. In any case, it would be desirable to unify effort from the different fragmented groups into one common goal, and I strongly think that CbC might be a strong incentive to join forces.

Some specific difficulties found along the way are related with the verification tools, It was time consuming to browse websites trying to find the adequate information, I had to use the trial and error method to discover many things, first during the installation process, where it happened that installing Krakatoa can only be done in some specific distributions, due to the dependencies, and when trying to configure a virtual machine a lot of compatibility problems arose, and after some time and effort I decided to give up, because there was not any kind of external support like forums or any other kind. Something similar happen when trying to work with JStar. With this second, it also happens that in order to specify the program, it requires a sizeable learning curve because the structure of the specification and the proves are based in separation logic and the idea is to make life easy for developers, not more complicated, especially when they have time restrictions.

Regarding to the requirements phase, we find another challenge in terms of Natural Language Processing and Artificial Intelligence, it is true though that there are algorithms that allow us to recognise verbs and nouns, and to disambiguate sentences, but even further, how can he use these AI techniques to generate automatically a UML class diagram annotates with OCL contracts, we need to bring together understanding of real world and understanding of computational and programming world, which is obviously another serious challenge.

7.3 Future Work

We can see that some parts of the CbC methodology are already advanced, like the pursuit of integration, some of them are still rising like the natural language specification processing, But once again construct by contract offers an opportunity to join efforts and bring formal verification and reliability to everyday software.

Specifically we can divide the future work in two branches, the first one follows the path of integration of existing tools, which mean reuse, collaboration and support, but also

means dependency. The second one is referred to the inception of the CbC tool from scratch, ignoring the existing tools and developing the CbC workspace from the knowledge acquired. Finally, I consider that the smartest option is actually a hybrid, starting from scratch might mean waste, and integrating might mean difficulties and incompatibilities.

Once we have mentioned our options, We consider that having enough time and economic resources, it might be possible to build the CbC workspace, by enhancing the core functionality of KeY, extending the translation capabilities, including some AI to process natural language, include the automatic generation of UML diagrams, aggregating automatic generation of documentation and including support for floating point numbers among other features.

7.4 General conclusion

Reliability is a desirable feature for each software system, despite the domain, functionality or platform they are related. Nevertheless this feature is generally sacrificed in favour of time and financial savings, remaining only as mandatory for critical safety systems. The reason because reliability is expensive is because the tools that allow to reach this property are not mature enough to be easily integrated in the common processes to build software.

We have realized the vast number of tools and techniques that can help us to improve software reliability, but since this is a “fresh” research area, these tools and techniques are not mature enough. They present problems such like integration, automation, or completeness, besides the fact that most of them are abandoned because of the low support they receive either from the industry or from the research founding groups.

Our solution proposed is a methodology called Construct by Contract that claims achieve the development of dependable software systems, by seamlessly integrating the existing tools and techniques related to reliability in one generic multifunctional development workspace, by extending Design by Contract and focusing the developing lifecycle in contracts gotten from the requirements speciation and persisted till the verification phase.

Our solution can be implemented in non-trivial and industrial software by chaining the existing independent tools according to the definitions of the methodology. This implementation process can be improved significantly by supporting our methodology with one tool that can seamless integrate and automate our development process.

Both the adoption of the methodology and the development of the tool are desirable steps for the future of the development of dependable software systems, and they can be seen as a partner for the Verified Software Initiative.

References

[1] Association for Computing Machinery, “The 1998 ACM Computing Classification System,” 1988. [Online]. Available: http://www.acm.org/about/class/ccs98-html.

[2] D. Jackson and E. Kang, “Separation of concerns for dependable software design,” in *FoSER '10 Proceedings of the FSE/SDP workshop on Future of software engineering research*, New York, NY, USA, 2010.

[3] B. Littlewood, “Theories of Software Reliability: How Good Are They and How Can They Be Improved?,” *IEEE Transactions on Software Engineering,* vol. 6, no. 5, pp. 489-500, 1980.

[4] G. Gigante and D. Pascarella, “Formal methods in avionic software certification: the DO-178C perspective,” in *ISoLA'12 Proceedings of the 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation: applications and case studies*, Berlin, Heidelberg, 2012.

[5] C. Hoare, J. Misra, G. T. Leavens and N. Shankar, “The verified software initiative: A manifesto,” *ACM Computing Surveys (CSUR),* vol. 41, no. 4, 2009.

[6] J. Woodcock, P. Gorm Larsen, J. Biscarregui and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM Computing Surveys (CSUR),* vol. 41, no. 4, 2009.

[7] J. P. Bowen and M. G. Hinchey, “Seven More Myths of Formal Methods,” *IEEE Software,* vol. 7, no. 5, pp. 11-19, 1995.

[8] Oxford University Press, “computer,” 06 June 2013. [Online]. Available: http://oxforddictionaries.com/definition/english/computer?q=computer.

[9] J. S. Conery, Explorations in Computing: An Introduction to Computer Science, Eugene, USA: CRC Press, 2010.

[10] BSI, *BS ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models,* British Standard, 2011.

[11] G. Philipson, “A Short History of Software,” Routledge, 2004.

[12] “THE STANDISH GROUP REPORT CHAOS,” The Standish Group, 1995.

[13] I. S. Gerald Kotonya, Requirements Engineering: Processes and Techniques, John Wiley & Sons, 1988.

[14] I. Sommerville, SOFTWARE ENGINEERING, 9th ed., Pearson, 2010.

[15] OMG, “UML Semantics,” in *OMG-Unified Modeling Language, v1.4*, OMG, 2001.

[16] IBM, “Unified Modeling Language (UML),” 21 June 2013. [Online]. Available: http://www-01.ibm.com/software/rational/uml/.

[17] M. v. d. Wulp, “ArgoUML User Manual,” 2010. [Online]. Available: http://argouml-stats.tigris.org/documentation/manual-0.32/.

[18] IBM, “Rational Software Architect,” [Online]. Available: http://www-03.ibm.com/software/products/us/en/ratisoftarch.

[19] UMLet, “UMLet 12.0 (beta) Free UML Tool for Fast UML Diagrams,” [Online]. Available: http://www.umlet.com/.

[20] Wikipedia, “List of Unified Modeling Language tools,” October 2011. [Online]. Available: http://en.wikipedia.org/wiki/List_of_UML_tools.

[21] Dresden, “1. The university model - An UML/OCL example,” [Online]. Available: http://dresden-ocl.sourceforge.net/usage/ocl22sql/modelexplanation.html. [Accessed 28 June 2013].

[22] Object Management Group (OMG), “Object Constraint Language Specification,” in

OMG Unified Modeling Language Specification, 1st ed., 2000.

[23] E. D. S. Kim Waldén, "Business Object Notation (BON)," in *Handbook of Object Technology*, S. Zamir, Ed., CRC Press, 1998.

[24] J.-M. N. Kim Waldén, Seamless Object-Oriented Software Architecture — Analysis and Design of Reliable Systems, Prentice Hall, 1994.

[25] S. Zamir, Ed., Handbook of Object Technology, CRC Press, 1998.

[26] R. F. P. a. J. S. Ostroff, "A Comparison of the Business Object Notation and the Unified Modeling Language," in *«UML»'99 — The Unified Modeling Language. Beyond the Standard Second International Conference Fort Collins*, CO, USA, 1999.

[27] J. F. M. P. J. M. d. S. S. Almeida, Rigorous Software Development, Springer, 2011.

[28] C. A. R. HOARE, "An Axiomatic Basis for Computer Programming," *Communications of the ACM,* vol. 12, pp. 576-580, 10 October 1969.

[29] B. Meyer, "Object-Oriented Software Construction Second Edition," in *Object-Oriented Software Construction*, Santa Barbara (California), ISE Inc., p. 333.

[30] A. L. B. a. C. R. Gary T. Leavens, "JML: A Notation for Detailed Design," in *Behavioral Specifications for Businesses and Systems*, Kluwer Academic Publishers, 1999, pp. 175-188.

[31] J. V. Guttag, J. J. Horning and J. M. Wing, "The Larch Family of Specification Langaages," *IEEE Software,* 1985.

[32] R.-J. Back, "On the Correctness of Refinement Steps in Program Development," Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978.

[33] G. T. Leavens and Y. Cheon, "Design by Contract with JML," 2006.

[34] "A modular integration of SAT/SMT solvers to coq through proof witnesses," in *CPP'11 Proceedings of the First international conference on Certified Programs and Proofs*, Berlin, Heidelberg, 2011.

[35] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM,* pp. 69-77, September 2011.

[36] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, Berlin, Heidelberg, 2008.

[37] B. Dutertre and d. M. Leonardo, "The YICES SMT Solver," 2006.

[38] D. Detlefs, G. Nelson and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM (JACM),* vol. 52, no. 3, pp. 365-473, 2005.

[39] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, Berlin, Germany, 2007.

[40] B. Livshits, "IMPROVING SOFTWARE SECURITY WITH PRECISE STATIC AND RUNTIME ANALYSIS," 2006.

[41] J. C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *CAV'07 Proceedings of the 19th international conference on Computer aided verification*, Berlin, Heidelberg , 2007.

[42] "Krakatoa and Jessie: verification tools for Java and C programs," 20 April 2013. [Online]. Available: http://krakatoa.lri.fr/. [Accessed 22 June 2013].

[43] J. R. Kiniry, "The Logics and Calculi of ESC/Java2," 2004.

[44] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, "Extended static checking for Java," in *PLDI '02 Proceedings of the ACM SIGPLAN*

2002 Conference on Programming language design and implementation, NY, USA, 2002.

[45] J. Rieken, “Design By Contract for Java - Revised,” 2007.

[46] D. Distefano, “jStar: towards practical verification for java,” in *OOPSLA '08 Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, New York, NY, USA, 2008.

[47] B. Beckert, R. Hähnle and P. H. Schmitt, Verification of Object-Oriented Software:, Springer-Verlag, 2007.

[48] B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager and P. H. Schmitt, “The KeY system 1.0 (Deduction Component),” in *CADE-21 Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, Berlin, Heidelberg, 2007.

[49] S. Brown, J. Timoney, T. Lysaght and D. Ye, Software Testing Principles and Practice, China Machine Press, 2011.

[50] Y. Nishi, “Quality-Adaptive Testing: A Strategy for Testing with Focusing on Where Bugs Have Been Detected,” in *ECSQ '02 Proceedings of the 7th International Conference on Software Quality*, London, UK, 2002.

[51] H. Do, G. Rothermel and A. Kinneer, “Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis,” *Empirical Software Engineering,* vol. 11, no. 1, pp. 33-70, 2006.

[52] C. Oriat, “Jartege: a tool for random generation of unit tests for java classes,” in *QoSA'05 Proceedings of the First international conference on Quality of Software Architectures and Software Quality, and Proceedings of the Second International conference on Software Quality*, Berlin, Heidelberg, 2005.

[53] Applied Formal Methods Group, “JMLUnitNG Usage,” Institute of Technology, University of Washington Tacoma, 01 January 2012. [Online]. Available: http://formalmethods.insttech.washington.edu/software/jmlunitng/usage.html. [Accessed 28 June 2013].

[54] Proceedings of the 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation: applications and case studies, Berlin, Heidelberg: Springer-Verlag, 2012.

[55] T. Gilb, “Evolutionary Delivery versus the "waterfall model",” *ACM SIGSOFT Software Engineering Notes,* vol. 10, no. 3, July 1985.

[56] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn and W. Cunningham, “Manifesto for Agile Software Development,” 11 November 2001. [Online]. Available: http://agilemanifesto.org/. [Accessed 26 June 2013].

[57] S.-q. Huang and H.-m. Zhang, “Research on Improved MVC Design Pattern Based on Struts and XSL,” in *ISISE '08 Proceedings of the 2008 International Symposium on Information Science and Engieering*, Washington, DC, USA, 2008.

[58] Kind Software, “Mobius Program Verification Environment,” Kind Software, 28 November 2008. [Online]. Available: http://kindsoftware.com/products/opensource/Mobius/. [Accessed 25 June 2013].

[59] Gliffy, “UML Example,” Gliffy, [Online]. Available: http://www.gliffy.com/examples/uml/. [Accessed 28 June 2013].

Table of figures

Figure 1 Example of UML class diagram [21] 12
Figure 2 Example of OCL contracts [21] 13
Figure 3 BON informal class chart example [25] 14
Figure 4 Hoare Logic Example for Assignment Rule 14
Figure 5 JML Banking Example 16
Figure 6 Architecture of Krakatoa [42] 19
Figure 7 Architecture of ESC/JAVA2 [44] 20
Figure 8 Architecture of Modern Jass [45] 20
Figure 9 Architecture of JStar [46] 21
Figure 10 Architecture of KeY [48] 22
Figure 11 The ACM Computer Classification System (1998) 26
Figure 12 CbC Software Development Lifecycle 27
Figure 13 Conceptual Structure of Construct by Contract 29
Figure 14 UML Use Cases for CbC Tool 36
Figure 15 Sequence diagram to develop dependable software systems with the CbC tool 37
Figure 16 Architecture Diagram for CbC Tool 38
Figure 17 Components Diagram for CbC tool 41
Figure 18 BluenetsWeb main page 42
Figure 19 Original Requirements Specification 43
Figure 20 MVC Pattern [57] 46
Figure 21 UML class diagram for BluenetsWeb 47
Figure 22 OCL constrains for BluenetsWeb 47
Figure 23 The Eclipse workspace for KeY 48
Figure 24 Java code for the pricing method 49
Figure 25 JML specification for pricing method 49
Figure 26 KeY verification results 50
Figure 27 Java code and JML specification for getBasePrice method 50
Figure 28 Java code and JML specification for getDiscountByHours method 51
Figure 29 Java code and JML specification for getDiscountByContract method 51
Figure 30 Proof of correctness for pricing method 51
Figure 31 Execution of JMLUnitNG 52
Figure 32 Test Case generated by JMLUnitng 52

Table of definitions

Definition 1 Construct by Contract 25
Definition 2 Contract 25
Definition 3 Propagation 25
Definition 4 Reversibility 25
Definition 5 Persistency 26
Definition 6 Client 27
Definition 7 User 27
Definition 8 Project Manager 27
Definition 9 Software Architect 27
Definition 10 Developer 28
Definition 11 Tester 28
Definition 12 Simple Sentence 30