



OULUN YLIOPISTO
UNIVERSITY of OULU

An Experimental Evaluation of Java Design-by-Contract Extensions

University of Oulu
Faculty of Information Technology and
Electrical Engineering
Degree Programme in Information
Processing Science
Master's Thesis
Majid Aghaei
28.11.2018

Abstract

Design by Contract (DbC), also referred as Programming by Contract is a programming paradigm for software verification proposed by Bertrand Meyer. The idea is to put obligations for code elements such as methods, interfaces and classes to satisfy the specification of the source code. Indeed, DbC enforces a piece of code to satisfy some conditions before execution (Precondition), and to ensure some conditions after execution (Postcondition) with holding some conditions unchanged (Invariant). This settlement is called Contract which must be valid for that piece of code. According to Meyer's paradigm, a program can work correctly if it fulfills DbC principles for each method. To empower programmers with DbC, various libraries are made to make DbC possible in coding phase each of which is applicable in a specific programming language and has some features and functionalities. However, choosing the most suitable tool for coding upon a particular purpose is considerably important for development teams with software validation deployment. This thesis aimed to experimentally evaluate and compare DbC instrumentors specially for Java and figure out that which tool had better performance. In order to accomplish such a task, a simple model system had to be designed and implemented with regard to using constraining principles of mentioned tools. The scrutiny of the extensions revealed that Open JML as a powerful framework has generated better results rather than other tools. However, the results of this research is viable for small projects deploying constraining tools.

Keywords

Design by Contract, Formal Software Verification, Java pre-processors

Supervisor

University Lecturer, Antti Siirtola

Abbreviations

DbC – Design by Contract

C4J – Contracts for Java

Cofoja – Contracts for Java

JML – Java Modelling Language

Contents

Abstract	2
Abbreviation.....	3
Contents	4
1. Introduction	5
1.1 Goal of Study.....	7
1.2 Research Question	7
1.3 Research Methodology.....	8
1.4 Research Contribution.....	8
2. Prior Research and Related Work	9
3. Contract Programming	12
3.1 What Is a Contract?	12
3.2 Design by Contract VS Defensive Programming.....	13
3.3 Design by Contract VS Static Code Analysis	13
3.4 Design by Contract for Java	14
3.4.1 Why Java?	14
3.4.2 Java Built-in Assertion Support	15
3.5 Java Implementations and Specification Languages.....	15
3.5.1 Bean Validation API	16
3.5.2 Contracts for Java (C4J).....	17
3.5.3 Contracts for Java (Cofoja)	18
3.5.4 ezContract.....	19
3.5.5 iContract	20
3.5.6 jContractor.....	20
3.5.7 Modern Jass	21
3.5.8 OpenJML.....	21
3.5.9 Oval with AspectJ.....	22
4. Study Design	24
4.1 Model System.....	24
4.2 Implementation.....	26
4.2.1 Bean Validation	29
4.2.2 C4J.....	35
4.2.3 Cofoja	40
4.2.4 ezContract.....	42
4.2.5 iContract	45
4.2.6 jContractor.....	47
4.2.7 Modern Jass	49
4.2.8 Open JML.....	50
4.2.9 Oval with AspectJ.....	53
5. Findings and Analysis	57
5.1 Test Station Specification.....	57
5.2 Evaluation Criteria	57
5.3 Functionality Assessment.....	58
5.4 Performance Analysis.....	60
5.5 Answering Research Questions.....	61
6. Discussion and Conclusion	63
6.1 Discussion	63
6.2 Limitations.....	64
6.3 Conclusion.....	64
References	65

1. Introduction

Software correctness is the minimum requirement of any software (Ehmer Khan, 2010). It means that without correctness, software might not be viable under any circumstances and it is fundamentally essential from the perspective of both software producers and end users at the same time. In addition, correctness criticality gets higher degrees in safety-critical software in which a trivial error can have serious effects and puts human lives in danger such as failure in fight systems or avionics (German, 2003). In this case, one could think that it is better to test every possible permutation of code and create as many test cases as possible, which is not an easy process at all (Myers, 2004). Furthermore, software testing is useful to find the defects but it does not guarantee that a piece of code will not generate any error for all possible inputs.

Based on theories in computer science, correctness is the compliance of code with its specification (Manna & Pnueli, 1974). In other words, the prime property of a program is whether the user's intention is fulfilled or not (Hoare, 1969); hence, code is correct if it conforms thoroughly to its presumed specification. This ideal is not only a gratifyingly academic theory, e.g. an erroneous piece of code in a software module caused the failure and crash of the European Ariane 5 launcher (Jézéquel & Meyer, 1997). It is mentioned that the reason for such an expensive disaster arose from the lack of accurate specification of a reusable module making an unprecedented error in converting a 64-bit floating-point number to a 16-bit signed integer which is a technical flaw. Although mission document stated that the value should be fit in a 16-bit integer, this requirement had not been specified in the code. Jézéquel & Meyer (1997) highlighted that to have effective reusability, it is critical to demand precise specification in the reusable code.

To distinguish the extent of conformance, formal verification is able to prove or reject the correctness of a program through considering formal specifications and deploying mathematical methods. Some of formal methods tap into pre- and postconditions for each code fragment. Preconditions promise to be true before the code execution and postconditions guarantee to be valid after execution. Hoare (1969) described that user intentions can be shown through some assertions before and after execution of code to ensure its validity in accordance to what it is expected to do. Design by Contract (DbC) paradigm originated from Hoare's logic (Crocker, 2004) and addresses the aim of reaching a correct software with respect to certain specifications expressed by mentioned assertions (Arnout & Simon, 2001). In essence, this discipline defines meticulous checkable specifications for elements of a programming language, e.g. types, methods, and objects to ensure that the real execution does not violate its properties (Plessel, 1998).

A contract is briefly an agreement between two parties in which both parties demand benefits from the contract and both are obliged to fulfil some conditions (Meyer, 1992). For instance, the client provides some data (client's obligation) to gain a value (profitability aspect of contract for client) and the contractor is responsible for processing the given data and satisfy the customer's need (contractor's obligation), thus, he is paid for his service (profitability aspect of contract for supplier). In software development, a contract is an arrangement between a method and its caller in a sense that if it is invoked on state S_1 with the satisfaction of its precondition P , it will return in state S_2 satisfying its postcondition R (Crocker, 2004). For instance, consider the following Java class:

```

public class MyClass {
    ...
    public int[] sort(int[] array)
    {
        int n = array.length;
        for (int i = 0; i < n; i++)
            for (int j = 1; j < (n-i) ; j++)
            {
                int temp = 0;
                if(array[j-1] > array[j])
                {
                    temp = array[j-1];
                    array[j-1] = array[j];
                    array[j] = temp;
                }
            }
        return array;
    }
    ...
}

```

Before the execution of *sort* the input array is unsorted (state S_1 : having the unprocessed array) but after *sort* invocation it is expected that the returned array is sorted (state S_2 : having the sorted array). Therefore, the precondition (P) is that the array should be a non-null array and the postcondition (R) is that the returned array should be non-null and sorted ascendingly. Despite holding the validity of conditions before and after the running, there are some permanently unchangeable conditions throughout the method called *invariants* (consistency constraints). For instance, in the above code *i* is always a positive integer. To characterise a contract, it is important to add the pre- and postconditions as part of the code (Meyer, 1992) which, as a result, modifies our example to be like:

```

public class MyClass {
    ...
    //requires array should not be null
    //ensures array will not be null and will be sorted ascendingly
    public int[] sort(int[] array)
    {
        int n = array.length;
        for (int i = 0; i < n; i++)
            for (int j = 1; j < (n-i) ; j++)
            {
                int temp = 0;
                if(array[j-1] > array[j])
                {
                    temp = array[j-1];
                    array[j-1] = array[j];
                    array[j] = temp;
                }
            }
        return array;
    }
    ...
}

```

Of course, in this example the contract is reflected in comment format, and it does not affect the compiling process and it is only for clarification. As a matter of fact, specifications are being written by using annotations (mostly in form of *@requires* or *@ensures*) influencing the compilation of a program and add some runtime assertions to

check the validity of execution. Nowadays, a variety of DbC implementations have been emerged in different languages like Smalltalk, Ada, C and C++, C#, Java, Perl, Python, so forth each of which providing pre- and postconditions, and invariants in distinctive annotation principles for developers to describe their software specification. For Java, there have been created various extensions with varied purposes and functionalities and the aim of this thesis is to evaluate the quality of existing DbC pre-compilers, particularly made for Java, to find the state of the art innovations.

1.1 Goal of Study

The key objective of this study is to find the best DbC extensions tailored to Java programming language to entice those developers who are willing to use internal assertions such as DbC annotations in their design to reduce software flaws as much as possible by selecting the most appropriate and beneficial tool. In order to choose a set of existing tools, some criteria have primarily been taken into account, e.g. live and active tools, updated continuously yet recently, extensions with provided documentation and user guidance, and those tools for whom the highest number of posts and comments are created in famous development forums like StackOverflow.

By listing the promising artefacts, then it is possible to accomplish the defined purpose by comparing current libraries, pre-compilers, and pre-processors through analysing their structure and performance. The analysis is exerted in two distinguishable parts; first part is responsible for finding all Java DbC instantiations provided so far for deployment and subsequently prioritising them according to the adopted criteria to nominate the most befitting instances for further research and examination. The second part is focusing on tools' qualifications and performance particularly when user intermixes specifications with code and deploys the tool by simulating a real execution of a simple program during writing the code, writing the specification, compilation, and runtime assessment. Finally, the results will be discussed and the extent of reliability and usability of tools will be determined.

1.2 Research Question

The main research question of this attempt is: *What is the state of the art in Java contract programming?* In other words, what is the best implementation of DbC for Java? To answer this question, it is necessary to consider different viewpoints for each extension to be better/best. Thus, the research question would be shrunk to the following subquestions:

1. Which tools can create better results from the viewpoint of functionality?
2. Which tools are better from the viewpoint of performance?

To ease the research process, first it is important to identify DbC and illustrate how it is possible to create contracts for classes, and add specifications in the code. Then it would be nice to find the existing tools, no matter commercial or free of use, that support DbC and then select the most mature instances, and finally evaluate them to catch out the best extension. The evaluation effort definitely needs defining a measurement approach and quality metrics, criteria for assessment, writing a simple program, and test the performance of tools.

1.3 Research Methodology

There are two famous and scientific research approaches: qualitative and quantitative research. Muijs (2004) argued that qualitative research is a rather subjective methodology, being rather means it is not an absolute fact, and it does not go for numerical data analysis while quantitative paradigm concentrates on using statistics and mathematical models to tackle a problem in a rather objective approach. In fact, it explains a phenomenon by using numerical data gathered from examining variables and analysing them by employing mathematical methods (Yilmaz, 2013). The primary intention of quantitative research is theory or phenomenon explanation in a mathematical context, though a quantitative research can be experimental. According to Muijs (2004) an experimental quantitative research (known as scientific method) is a type of study based on testing a phenomenon on a set of controlled conditions to prove a truth or examine the validity of a claim. Considering such theory, this research aims to evaluate some artefacts under controlled conditions and according to experimental measurements and then compares the collected data to find the most promising solution in a group of applications. The measurement consists of some items that should be scored at the examination of tools and also after the experience the researcher gains at working with the tools. The experience of the author can be digitised by letting him/her to score an aspect in a range of points, e.g. from 1 to 5 for each item. For systematic qualities, the environment of testing can generate data automatically. This kind of research cannot be considered as design science research, although coding is required. The main reason is that the coding will not make a new artefact that resolves an existing problem because the coding phase alleviates the comparing of tools by making the comparison a structured and similar to all extensions.

1.4 Research Contribution

Although there is versatile research done with regard to DbC and most of them tried to either introduce new frameworks or propose analysis on existing instantiations of constraint-based programming approach, there is a scarcity of research on technology evaluation in this area. In essence, none of studies has done a systematic assessment on the prolonged tools mentioned earlier. This study not only describes all relevant concepts of DbC, which is certainly informative for developers to employ DbC in their design, but it also focuses on an experimental evaluation of provided instrumentations to avail Java developers to have a clear understanding of current technologies embedded in DbC extensions. As a result, they will be able to figure out that which tool can bring them better productivity.

2. Prior Research and Related Work

In this chapter prior research and related work done since the emergence of DbC will be articulated and it is simultaneously tried to have a chronological perspective towards the research advancement in this field. In addition, the main concern is to give a clear understanding of efforts thrive to reach a new tool or evaluate the impact of DbC in real life situations and connected outcomes.

DbC originated from the work of Bertrand Meyer, the president of Interactive Software Engineering Inc., which subsequently combined in Eiffel language. Eiffel was the progenitor of such implementations, in which using contracts was supposed to be fruitful for early design phases of software development (Meyer, 1988). Indeed, these phases concentrate on object-oriented programming and considerations of reliability, reusability, extendibility, and modularity. Meyer (1992) focused on reliability and robustness of object-oriented technology in terms of methodological principles and systematic approach. In fact, reliability is implementable through considering software elements where they are precisely meant and actualised in order to satisfy well-perceived specifications when the theory of contract programming becomes necessary.

To follow up contract theory, many research studies and scholars significantly attempted to integrate specification provisions in their work and innovate new extensions with higher acceptance and capabilities for different programming languages. Carrillo-Castellon, Garcia-Molina, Pimentel, & Repiso (1996) stated a crucial drawback of Smalltalk in its core where the object-oriented paradigm was not seen as a basis, then proposed an extension of DbC for Smalltalk to increase the reliability of software, in particular, empowering Smalltalk to cover correctness and robustness. In essence, not only was the capability of specifying assertions embedded, and the behaviour of the code was identified in case of any assertion violation, but also maintaining execution control after the violation was provided. Likewise, Plösch (1997) integrated DbC into Python, which similar to Smalltalk is a dynamically typed programming language, and argued that the main purpose of adding DbC was due to the need of supporting assertions in analysis and design phases of development. Moreover, this implementation would help to perform object-oriented domain knowledge modeling, domain model semantic specification through formal methods, and prototyping the domain model with software architecture.

For other languages like C++ and C#, DbC was implemented as well. Guerreiro (2001) introduced a light assertion mechanism for C++ with a simple on and off toggle for enabling and disabling DbC even in the middle of execution. It is frankly mentioned that although the proposed tool is not fully functional, having minor issues with inheritance and recursion, it is quite applicable to insert DbC principles into C++ codes. In addition, Arnout & Simon (2001) explained that ISE Contract Wizard enables .NET assemblies to be contracted by using Eiffel as an intermediate language. Indeed, the Eiffel compiler is used to get the associated information from metadata and once the contracts are added to the code, Eiffel compiler subsequently generates a proxy to the original assembly with a similar interface but contracted as a result.

For Java a wide variety of tools have been made and released even more than other languages. According to Kramer (1998), iContract was the first Java attempt designed to pre-process Java source code and preserve readability of instrumented code. Moreover, it has performance configurability to avoid unacceptable overhead. Leavens, Baker, & Ruby (1998) focused on JML and its ultimate goal and contended that JML is able to

support quantified assertions (pose a constraint on group of elements), reuse contracts, provide checkable redundancy and a useful set of pure types (e.g. `JMLInteger`, `JMLObjectMap`, etc.). In another effort, Karaorman, Hölzle, & Bruno (1999) introduced a library-based extension called `jContractor` designated to support DbC and enable the developer to incorporate contracts either directly in Java classes or separately in associated contract classes. Moreover, it accommodates the coder with practice of conventions without any specific requirement or demand of other systems. By using `jContractor` programmer can express pre-, post- conditions, and class invariants, check the results of methods at runtime, and access the old values of attributes and variables in order to handle the possible exceptions through defined contracts. Additionally, the composition of classes including contract patterns would be achievable during the class loading or object instantiation and the runtime contract enforcement is possible as well.

The continuation of tool instrumentation for Java can be seen in the work of Leavens, Baker, & Ruby (2003); Leavens & Cheon (2006) presented *Java Modelling Language* (JML) as a behavioural interface specification language, which supports DbC and extends Java expressions, and described its architecture, concepts, technical requirements, usage and tool support in detail. According to their research, the aim of JML was to record software behaviour then provide understandable notations and rigorous formal semantics to capacitate static analysis and testing. Furthermore, it was important to actualise a specification compiler to create prototypes from specification systematically. Besides, the main approach of JML was to provide specification-only usage of model variables expressing the abstract behaviour and value of attributes and conceal the mathematical notation behind a class by using pure methods without any side-effects.

In another study, Wampler (2006) pinpointed `Contract4J` as an open source DbC extension defining a design pattern to attach contracts to classes with two different syntax forms and concurrently concerning less coupling between aspects and components. Arnout & Simon (2001) in addition to outlining some tools with support of contracts for programmers in languages other than Eiffel mentioned *HandShake* which enabled the programmer to add contracts to the class definition without accessing to it directly. Chen, Cheng, & Hsieh (2008) classified the Java tools according to a specification-based scheme with affirming the lack of defined quality attributes in those attempts. Furthermore, a survey of tools was presented in their study categorised in five groups and compared with the specified criteria. Their main contribution was the innovation of a new tool called *ezContract* averring to cover all quality measures and be fully functional. The advantage of *ezContract* over other tools has been stated as using markers, which are dummy methods and attributes implementing keywords such as `require`, `ensure`, etc. in a contract structure, and *ezContract* utilises them to delineate the contracts.

Minh Lê (2011) described contract programming and its applicability, and claimed that since Java originally supports only simple assertions, many alternatives had been introduced yet a very robust aspect-oriented framework with outstanding features was required. Therefore, *Cofoja* was introduced to provide concise contract specification with the least dependencies. In the meantime, it provides type checking and compile-time syntax, compilation and integration of contracts without replacing Java compiler, and adding contracts by modifying bytecode online or offline. In addition, some other efforts such as *C4J*, *JASS*, and *Modern Jass* done by different research teams were cited.

Despite the creation and introduction of new tools, some researchers concentrated on studying the impact of DbC in other areas. For example, Feldman (2003) showed that DbC has a prestigious effect on quality of code, as Bolstad (2004) also highlighted, and embodies synergy with agile methodologies, e.g. XP. Besides, it facilitates vivid design,

detects common bugs as early as possible, and makes the simplification of test case generation attainable. Hakonen, Hyrynsalmi, & Järvi (2011) advocated the ease of unit testing by reducing the number of test cases when the contracting approach comes in. Some research scholars investigated the impact of harnessing DbC in software robustness. Crooker (2004) focused on issues of safety-critical software sectors and their reluctance toward entering the object-oriented utilisers' community. The research proposed a framework made of a design technique empowered by formal verification to increase the robustness of safety-critical software. The framework employed modern automated reasoning methods to provide a more cost-efficient platform for developers and used formal specification to reduce code flaws and make the auto code generation possible for higher productivity purposes. The essence of DbC was to tame polymorphism with dynamic binding in object-oriented technology that had been causing uncertainties for safety-critical community.

In a same fashion, contract-based design can be fruitful in reducing bugs in car industry related software, which to some extent can be taken into account as another safety-critical field. Zhou, Pelliccione, Haraldsson, & Islam (2017) studied the effect of contract-based programming in robustness of AUTOSAR (Automotive Open System Architecture) software components. The aim of AUTOSAR is to get a universal standard for automotive electrical/electronic (E/E) architecture to reach a good level of modularity, scalability, and reusability for its associated vehicle industry embedded software systems. The results of their scrutiny unveiled that DbC is arguably able to escalate the robustness of the software, enhance code readability and understandability, alleviate the refactoring phase of manually coded modules, and reduce redundancy remarkably.

Not only is DbC important from the perspective of technicality, but it also can be considered from the view of evaluation. Programming by contract can be assessed through non-functional aspects of programming and software development particularly on developers' skills. For instance, Chen, Cheng, & Hsieh (2008) claimed that the extent to which a developer's programming activities gets affected by the engagement of contracts in their work stays at the highest level of importance for evaluating the pertinent DbC extensions.

In addition to mentioned research attempts, there are several works and studies considering different aspects of DbC and its influence in various fields of software validation and verification, however, they are out of the scope of this study.

3. Contract Programming

In this chapter programming by contract and all related concepts will be explained and exemplified as simple as possible. Subsequently, the essence of using contracts in programming will be justified theoretically. Moreover, the DbC for Java will be expressed and the most popular DbC extensions will be introduced.

3.1 What is a contract?

In Wikipedia (<https://en.wikipedia.org/wiki/Contract>, 2018 October) contract is defined as:

“A contract is a promise or set of promises that are legally enforceable and, if violated, allow the injured party access to legal remedies. Contract law recognises and governs the rights and duties arising from agreements.”

Intuitively, the definition illustrates the nature of contract as an agreement, either voluntary or forceful, and pinpoints a very important concept explicitly, i.e. both parties have rights and duties during the agreement. In real life, contract is an agreement which forces the parties to undertake some obligations in return of some benefits. As a quite familiar example to the academic environment, the admission and right of study at a university is an agreement between university as a legal person and a student as a natural person in which the university pledges to teach and give the student savvy and the student promises to attend the lectures and study hard to pass the exams. It can be intuitively inferred that one party's obligation is the other one's benefit (Meyer, 1992). For example, Table 1 shows the obligations and benefits for university and student.

Table 1. An Example of a Simple Contract between University and Student

	Obligation	Benefit
University	Teach and develop student's knowledge and skills	No need to give certificate to students who fail the assessments
Student	Attend lectures, study and pass the exams	Gain knowledge and expertise

It might be questionable at first glance that the benefit for the university doesn't make sense! But it does not because the university is an institution aiming to develop people's capabilities and is obliged to give proper candidates associated certificates of achievement. In fact, it is reasonably beneficial for a university to not issue a diploma for an unqualified student.

When it comes to programming, the metaphor of contracting has an identical meaning to the definition above. In fact, a contract can be viewed as an interaction between two operations; sender operation that sends a message as a client or customer and the receiver operation as a supplier that implements the message delineated in terms of assertions (Firesmith, 1999). It should be noted that an assertion is a boolean-valued expression

representing a condition and is able to result in true or false situation (Plessel, 1998). Any violation of the defined assertion is considered as a defect. The reason for such a tight obligation is to provide correctness and robustness of software that are two components of software quality. According to Plessel (1998), correctness is

“The ability of software to perform their exact tasks as defined by their specification.”

And robustness is

“The ability of software systems to react appropriately to abnormal conditions.”

To ensure the correctness and robustness of software, Hoare (1969) stated that the values sent to the receiver operation (called method) definitely affects the results of the program before it is initiated. In other words, there is a dependency between the result and the inputs. The validity of initialisations, that should be true before entering the call, is defined as precondition, and the guarantee to result from the correct inputs after the execution is defined as postcondition (Firesmith, 1999). The depiction of the dependency of inputs and outputs and their connection to program can be showed in a triple format of

$$P \{Q\} R$$

In which P is precondition, Q is the executing program, and R is the postcondition and describes the relation as “if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion.” (Hoare, 1969). In essence, Hoare triple conveys the definition of contract implicitly but theoretically and mathematically proven using axioms and rules which help to place a reliability on outputs.

3.2 Design by Contract Vs Defensive Programming

Defensive programming is a design and implementation methodology at the class-specification level destined to reduce the bugs or misuse of an abstraction (Firesmith, 1999). Defensive development focuses on correctness and robustness of software as DbC does with some distinctions. In consequence, defending approach aims to shield every module by putting as many checks as possible (Le Traon, Baudry, & Jézéquel, 2006). Although defensive development is noticeably fruitful because it prevents even trivial errors might cause bigger faults during the production phase, it incurs extra overhead both at runtime and for the coder at programming (Liskov & Guttag, 1986). However, DbC and Defensive programming are not the same approaches, even if they have many similarities. Firesmith (1999) mentioned that both methods use assertions and amazingly the same assertions, in both approaches the receiver has to take care of postconditions and invariants, and both has to provide alternative execution upon rising exceptions. However, Defensive programming poses redundant checking as there is a lack of systematic framework for addressing reliability issues, hence, there would be an imposed performance penalty (Meyer, 1997).

3.3 Design by Contract Vs Static Code Analysis

Delev & Gjorgjevikj (2017) described static analysis as the process of evaluation of source code without real execution of it, which can find severe errors before they happen and is useful to heighten the quality of code, code security, and its robustness. This early detection of serious errors can diminish product development cost, save a huge amount of time for development teams, and increase the overall quality (Novikov, Ivutin,

Troshina, & Vasiliev, 2017). In fact, static analysis concerns the correctness of a software module before its real implementation, an implementation leading to execution, and it focuses on finding and eliminating possible errors at early stages of the coding when the cost of correction is trivial. Contract programming is influential at the implementation and execution time at which each element is checked to satisfy the declared contract. For static analysis, specialized tools have been created to analyse a piece of code statically and make a thorough and consistent report of code assessment (Delev & Gjorgjevikj, 2017). German (2003) outlined some of the techniques proposed under the term of static analysis such as control flow analysis, data flow analysis, information flow analysis, semantic analysis, formal verification and so forth. However, although design by contract can be categorised in formal techniques the existence of defects is examined at runtime rather than at compiling or instrumenting bytecodes.

3.4 Design by Contract for Java

Before going any further, it is indispensable to specify frequently-used terms in DbC, i.e. precondition, postcondition, invariant, and inheritance, which will be recruited regularly in the following sections.

Precondition: is an assumption that should be true before the execution of a method and it is exactly what the caller of the method has to ensure (Thüm, Schaefer, Kuhlemann, Apel, & Saake, 2012). For example, an array argument entering a method should not be null because a null array is completely useless at any circumstances. As another example yet more semantic, let's suppose that a loop count, say n , is going to be passed to a method. In this case, n should not be a negative integer and only a positive value is meaningful.

Postcondition: is an assertion that should guarantee to be true after the method's termination (Leavens & Cheon, 2006). As an example let's assume we are writing a method responsible for making a withdrawal from a bank account. Once the transaction is made and the method is terminated successfully, the balance should be reduced; its amount has to be less than the former balance before the withdrawal. It ensures that some amount of money has been taken out.

Invariant: an invariant is valid before, during, and after the termination of the execution or throughout the class. From the perspective of object-oriented programming, it evaluates the conditions that must be true whenever the state of an object is stable and observable (Oliveira e Silva & Francisco, 2014); (Meyer, 1992). For instance in the previous example, the balance should always hold a non-negative value ($\text{balance} \geq 0$) even before or after the transaction.

Inheritance: as conceived in object-oriented approach, new classes can be defined by combining previously defined classes (including all their features) and besides, they can have their own features (Meyer, 1992). The same concept, also called subcontracting, can be applied to contracts in programming by contract as well with a pivotal rule; According to Plösch (1997), subcontracting requires keeping or weakening preconditions and keeping or strengthening the postconditions of the inherited method.

3.4.1 Why Java?

There are overwhelmingly various number of programming languages worldwide making software engineers boggle at choosing an appropriate language for long term deployment. Kumar & Dahiya (2017) has done a survey on existing programming languages'

popularity and employed different international indices to advocate their findings; their study shows that although Java and C had about 6 percent of downward compared to other languages since 2016, they are still on top of the list for experts, of course Java is the first choice, and this has been unexceptionally true for over 15 years. Even from the perspective of teaching and learning programming languages, Java has been the mostly used language for introductory courses with regard to pedagogical benefits and industrial relevance in UK and Australia (Simon, Mason, Crick, Davenport, & Murphy, 2018). Apart from the thesis writer's own preference, which is absolutely Java, research studies are supporting the idea that Java is the most popular language. Thus, the impact of engaging DbC in coding with Java may profit developers more.

3.4.2 Java Built-in Assertion Support

If the contracting is possible to be placed directly by a programming language, using assertions can be considered as a built-in support, which is checked by compiler and can be enabled or disabled (Plösch, 2002). According to Sharan (2017), Java provides assertion by assert statement in two ways:

1. `assert booleanAssertionExpression;`
2. `assert booleanAssertionExpression : errorMessageExpression;`

`booleanAssertionExpression` is the condition to be evaluated and the `errorMessageExpression` is the custom error message shown in case of any occurrence of error. For example, in the following snippet of code:

```
int i = 2;

int j = 5;

int z = i * j;

assert z == 10; //asserts if z is equal to 10 using the first style

assert z == 10 : "z is " + z; //in case of an error shows a message
```

Since deploying assertions at runtime will dictate performance penalty, assert statement is disabled by default in Java but using `-enableassertions` and `-enablesystemassertions`, it is easy to enable them. In a same fashion, disabling is manually possible using the equivalent commands only by replacing “enable” to “disable” in each one. It should be noted that assertion is quite different from exception, i.e. assertions are used to find the developer's programming flaws, it is highly critical to find and resolve the bug, while exception handling is useful to give an alternative execution path in case of end user's errors, and the recovery is optional (Sharan, 2017).

3.5 Java Implementations and Specification Languages

In this section the most popular DbC extensions for Java are introduced in an alphabetical order along with a brief explanation of their structures, usage, and principles. Since for evaluation of existing tools it is important to write a model program practically with its associated contracts, using annotation principles of each tool separately is inevitable, hence, it is worth to be familiar with syntactical rules and actualise them by easy to

understand examples. Indeed, the intention of this section is merely to let the reader envisage that how it is possible to write a code under each implementation and deploy the tool at runtime.

3.5.1 Bean Validation API

According to Morling (2017), JavaBean Validation is a Java API for Java EE 6 and later and Java SE done as part of JSR 380, JSR 349, and JSR 303 under the Java Community Process Program that provides a validation platform for Java. Defining constraints on objects, validation of parameters and return values, and reporting the possible violations can be done through Bean Validation. In fact, the validation mechanism placed in the domain model prevents redundant checking on different layers of software, from presentation to persistence, and creates metadata model declaratively simplifying the coding and its maintenance. Since the release of version 1.1 it supports DbC but not merely bundled to it meaning that it is generally designed for constraining the code elements.

Bean Validation specification provides a framework to define and declare constraints on fields, properties, container elements, and classes (Ferentschik, Morling, & Smet, 2018). Class constraints can be inherited in such a way that when a class extends another class or implements an interface, all the constraints defined for super-type will be valid for the new type. According to Morling (2017), constraints can be generic or cross-parameter; generic constraint targets:

1. Field (an attribute)
2. Method (getters and returned values)
3. Constructor (constructor return values)
4. Parameter
5. Type
6. ANNOTATION_TYPE (creating constraints on other constraints)
7. TYPE_USE (confining container elements)

While cross-parameter constraint targets:

1. Method
2. Constructor
3. ANNOTATION_TYPE

In both cases there should be a constraint validator for the annotation. Hence, the validator is responsible to validate the declared constraint on the given type. For clarity concerns, Table 2 presents each type of constraint by a simple example.

Table 2. Four Types of Contracts in Bean Validation Specification (According to Morling, 2017)

Type	Example
Field	<code>@NotNull private String name;</code>
Property	<code>@NotNull public String getName() { return name; }</code>
Container Element	<code>private Set<@ValidCourse String> Courses = new HashSet<>(); or private List<@ValidCourse String> Courses = new ArrayList<>();</code>
Class	<code>@ValidPrerequisiteCourse public class Student { private String name; private String StudentNo; Private List completedCourses; //... }</code>

To make everything easier for developers, some commonly used built-in constraints have been defined in `javax.validation.constraints` package. It should be noted that all of these annotations are only applicable on field or property level and there is no class-level predefined constraints. The most popular built-in annotations in Bean Validation are:

- `@Null`: To ensure that the element is null
- `@NotNull`: To ensure that the element is not null
- `@Min`: To ensure that the element is higher than or equal to a minimum amount
- `@Max`: To ensure that the element is less than or equal to a maximum amount
- `@Positive`: To ensure that the element is positive and not zero
- `@Negative`: To ensure that the element is negative and not zero
- `@AssertFalse`: To ensure that the element is false
- `@AssertTrue`: To ensure that the element is true
- `@Email`: To ensure that the element is a valid email address

3.5.2 Contracts for Java (C4J)

According to Bergström (n.d.), C4J is another framework which enables the coder to deploy easy to use but powerful features to implement DbC in the code. The minimum Java required for using C4J is Java 1.6. In C4J, full contract inheritance is supported and writing contracts in separate files as external contracts can leverage the refactoring process of legacy code. In essence, annotating programing elements makes it possible to assess the behaviour of the program in real situation not only for a limited number of test cases, which some of them might never be asserted at all!

There are two forms of contracting in C4J: internal and external that have some differences in definition and implementation. But the general format of contracting is straightforward; preconditions, postconditions, and class invariants are defined like this:

- Precondition:

```
if (preCondition()) {
    assert statement1;
    assert statement2;
    ...
    assert statement;
}
```

- Postcondition:

```
if (postCondition()) {
    assert statement1;
    assert statement2;
    ...
    assert statementn;
}
```

- Class Invariant:

```
@ClassInvariant

public void invariant() {

    assert statement1;

    assert statement2;

    ...

    assert statementn;

}
```

Each contract class can extend the contracted class (a class to which the contract may apply) and put precondition and postcondition on its methods and declare an invariant method by using `@ClassInvariant`. The whole concepts are viable for interfaces by the same mechanism.

3.5.3 Contracts for Java (Cofoja)

Cofoja introduced by Minh Lê (2011) is another third-party alternative framework for contract programming paradigm with a set of aspects and features. The tool is not a static analysis tool but it enables runtime checking through bytecode instrumentation and annotation assessment. Cofoja has a similar approach in processing annotations, utilising compile tools and instrumentation standard with a purpose of improvement in core techniques and emphasis on usability through supporting a comprehensive language (Minh Lê, 2011).

In a same manner to other extensions, Cofoja uses annotations to create contracts and bind them to code elements; types and methods. In essence, these annotations can be applied to classes and interfaces. Three main annotations defined by Cofoja are concise and self-describing:

- Precondition:

```
@Requires("an expression")
```

- Postcondition:

```
@Ensures("an expression")
```

- Invariant:

```
@Invariant("an expression")
```

For example, `@Requires("i > 0")` states that `i` should be a positive number. Or `@Ensures("y > 100")` guarantees that `y` will be greater than 100. In fact, pre and postconditions will be assessed in the context of the methods for which those pre and postconditions are defined. For postconditions, the annotation can have access to the result of the target method by keyword `result`, e.g. `@Ensures("result > 0")`, and can use the old value of a parameter by keyword `old`, e.g. `old(x)` considers the value of `x` on entry of the method call. Furthermore, Cofaja supports exception handling in case of incidents happened within contracted methods. In this case `@ThrowEnsures` is provided to catch the exception and prevent changing the state of an object for an exceptional execution of a method.

3.5.4 ezContract

ezContract as an open-source extension came to fill the gap of previous works by considering their drawbacks and deficiencies. Cheng, Chen, & Hsieh (2007) highlighted two main issues of other tools with regard to DbC programming and integration; the first issue is breaking source compatibility which occurs when new keywords are introduced but the manipulation of the code with proprietary compilers is not welcomed in most of the software companies. The second problem is related to the solutions suggested to resolve the first issue when naming conventions, adding meta-data to annotations or separating the contract files from source code files are proposed. In this case, any change of source code will not propagate to the contracts. Apart from the novel approach ezContract proposes, preconditions, postconditions, and invariants have a specific format:

- Precondition: each precondition starts with `Require.begin()` and ends with `Require.end()`. Within these lines any assertion can be added. For example, the following code shows how a precondition works:

```
Require.begin();
    assert i != 0: "the loop counter cannot be zero!";
Require.end();
```

- Postcondition: In a same manner, each postcondition starts with `Ensure.begin()` and ends with `Ensure.end()`. The following code snippet exemplifies defining and using postconditions:

```
Ensure.begin();
    assert result >= 0: "Square of an integer is a positive number!";
Ensure.end();
```

- Invariant: defining a class invariant requires a class to implement *ezcontract* interface which imposes the implementation of *classInvariant* method. For instance a class invariant method might look like this:

```
public class Test implements ezcontract{
    public void classInvariant{
        assert k != null: "size of array should not be null!"
    }
}
```

3.5.5 iContract

iContract developed by Reto Kramer is a Java preprocessor that lets the coder to specify custom assertions to instrument classes. According to Kramer (1998), after annotating the source code a repository of contract associated with classes, methods, and interfaces is created to support contract inheritance. The new files will be compiled to enforce precondition, postcondition, and invariant checks automatically. This process promises to keep the compliance of source files with Java standards at any point. However, to produce these checks, iContract has a specific pattern:

- **Precondition:** `@pre` works well to specify a precondition in code. For example to check that a guy's weight is a valid input the code can be like:

```
/**
 * @pre weight > 0
 * @pre weight < 200
 */
void setWeight(int weight);
```

- **Postcondition:** `@post` instruments postcondition on a class method. For instance, to get somebody's weight from a method, returned value should always be greater than zero:

```
/**
 * @post return > 0
 */
int getWeight();
```

- **Invariant:** `@invariant` defines a class or interface invariant to which all instances of a class should conform. For example, for a registered student the current semester is active and he has to have at least one course in his course list:

```
/**
 * @invariant isRegistered != false
 *             implies
 *             courses.size() >= 1
 */
public class Student{
    String name;
    String[] courses;
    Boolean isRegistered;
}
```

3.5.6 jContractor

jContractor is a Java library that eases the practice of DbC without reliance on any compiler, preprocessor, runtime system, or virtual machine. According to Karaorman, Hölzle, & Bruno(1999), jContractor analyses meta-level information of Java files to discover the encoded contracts. In particular, all the principles of DbC are provided in jContractor and one can create contracts either in the source classes or as separate contract classes. Contracting pattern in this tool is described in next few lines:

- **Precondition:** for each class method it is possible to simply write a precondition method by adding `_PreCondition` to the target method name with identical arguments but the return value of the constraint method should always be a protected boolean, that is normally the result of the required assertion:

```

object method1(<arglist>){
...
}
protected boolean method1_PreCondition(<arglist>){
    return (a boolean expression);
}

```

- **Postcondition:** with a similar scenario to precondition, a postcondition constraint of a method can be defined by adding `_PostCondition` to the target method name with boolean type of return value:

```

object method2(<arglist>){
...
}
protected boolean method2_PostCondition(<arglist>){
    return (a boolean expression);
}

```

- **Invariant:** to specify an invariant for a class, it is essential to add `_ClassInvariant` to the class name and write a new method that does not accept any arguments and returns a boolean:

```

class MyClass{
...
}
protected boolean MyClass_ClassInvariant(){
    return (a boolean expression);
}

```

3.5.7 ModernJASS

Rieken (2007) introduced ModernJASS as a DbC tool with a rich set of features that can be integrated with most of IDEs in a seamless build process. It employs Java 5 annotation to define a program's behaviour in terms of contracts. Specifying contract principles are quite easy and straightforward:

- **Precondition:** `@pre` expresses a precondition for a method. For example, `@pre("x % 2 == 0")` enforces that `x` should be an even number.
- **Postcondition:** `@post` expresses a postcondition for a method. For instance, `@post("@Result != null")` enforces a non-null return value of the target method.
- **Invariant:** to express an invariant, `@Invariant` is used to put an obligation on a class. For example:

```

@Invariant("@ForAll(String s: courses; !s.equals(""))")
public class student{
...
    String[] courses;
...
}

```

3.5.8 OpenJML

OpenJML is a verification tool that checks the specification of a Java program in which contracts are annotated by JML statements. Both static checking and runtime checking is

supported by OpenJML and parsing, type-checking and even editing is available as well. In addition, OpenJML is capable of generating test cases and documentation with JML information (“Introduction to OpenJML”, 2018). To use OpenJML, it is critical to know the basic principles of JML annotation style. According to Leavens & Cheon (2006), writing JML consistent annotations to specify pre-, postcondition, and invariant enforces the programmer to put statements in `/*@...*/` or start with `//@`:

- **Precondition:** `requires` clause defines precondition obligations to a method. For example `//@ requires x != 0;`
- **Postcondition:** `ensures` clause specifies the state of postcondition to the target method. For example:

```
/*@
  @ ensures \result == x * y;
  */
public int multiply(int x, int y){
    return x*y;
}
```

- **Invariant:** to specify an invariant for a class, using invariant clause will be necessary. For instance for the following class student’s ID is always non-empty string which is defined as an invariant:

```
public class Student{
    String studentID;
    //@ invariant !studentID.equals("");
}
```

3.5.9 Oval with AspectJ

OVal is a Java validation framework that allows developers to specify and configure constraints on any kind of Java objects using annotations. Although OVal is not fully fledged DbC implementation, when it is combined with AspectJ, which is a seamless aspect-oriented Java extension, DbC features are available (OVal - the object..., n.d.). apart from all the build-in assertions that OVal provides for programmers, its constraining format is similar to other tools:

- **Precondition:** `@Pre` expresses a conditional constraints on method parameters using scripting language. A sample precondition is like the following code fragment:

```
public class bankAccount{
    private BigDecimal balance;
    @Pre(expr = "_this.balance >= withdrawAmount", lang = "groovy")
    public void withdraw(BigDecimal withdrawAmount)
    {
        balance = balance - withdrawAmount;
    }
    ...
}
```

- **Postcondition:** in a similar way, `@Post` defines a postcondition. For example:

```
public class bankAccount{
    private BigDecimal balance;
    @Post(expr = "_this.balance < _old", lang = "groovy")
    public void withdraw(BigDecimal withdrawAmount)
    {
        balance = balance - withdrawAmount;
    }
    ...
}
```

- **Invariant:** by default if `@Guarded` is set on a class, all class invariants are checked before and after any non-private method. In case of annotating an invariant for a specific method, `@PreValidateThis` and `@PostValidateThis` will enforce object validation before or after method call respectively:

```
@Guarded
public class bankAccount{
    private BigDecimal balance;
    @PreValidateThis
    @PostValidateThis
    public void method1(BigDecimal withdrawAmount)
    {
        //do something here
    }
    ...
}
```

4. Study Design

In this chapter a simple Java program will be contracted and for each extension introduced in 3.3.3, it will be annotated according to their principles. The aim of scripting is to implement pre- and postcondition, invariant and inheritance in order to examine each tool on ease of introduction, ease of use, dependencies and external tools, compilation time, memory consumption, execution overhead, support of inheritance, and so forth. The candidate Integration Development Environment for testing the extensions is Eclipse Java EE in which all the tools will be deployed.

4.1 Model System

The model system chosen to deploy DbC is a simple car rental system with which a client can rent some cars and pay associated fee for a certain time of rent. Although this system can be amazingly broad, for sake of research concerns it should be considered as a subsystem with fewer components and classes. Figure 1 shows the main classes with their fields.

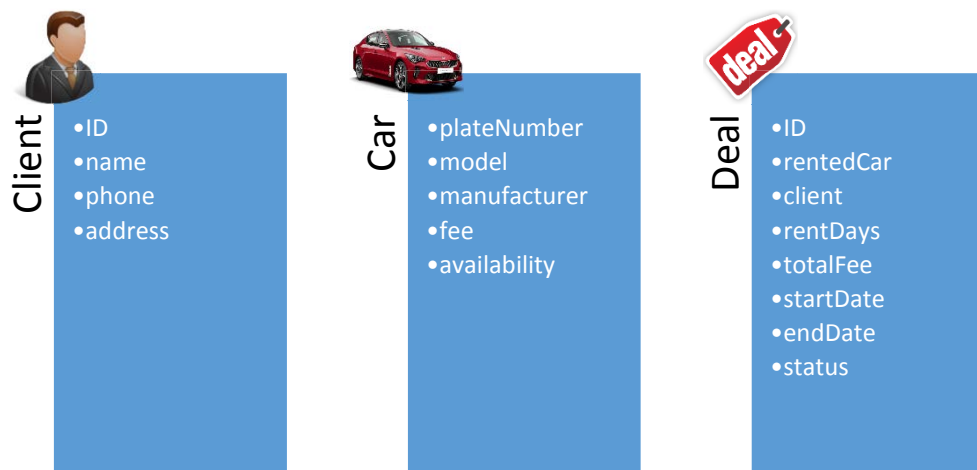


Figure 1. Car Rental System Classes

The model introduces three main classes:

- **Client:** a real/unreal customer who is registered and has been given a client ID. The client is allowed to rent one car at a time and he/she has to pay the fee to reserve an available car for a certain time. The client's id, name, phone number, and address are the most important fields.
- **Car:** a vehicle which is registered in the list of cars and can be rented periodically. A car can be rented by only one customer at a time. The class pertaining to car entity should hold plate number, it's model and manufacturer, rent fee, and its availability.
- **Deal:** a deal is a signed agreement that holds the information of the contract between the customer and the agency. Additionally, a deal keeps the information of chosen car, and the start/end time of the rental. A deal is reachable using its ID for further processes. The defined class has to keep the information of both car

and client, rent days, total rent fee (which is simply fee multiplied by rent days), start and end date of contract, and the contract status, which can be open or closed.

To be able to rent a car, a handler class is needed to act as a car agency. CarRental class is responsible to keep a list of clients, a list of cars, and a list of deals and has to administer searching an available car by `searchAvailableCar` method, leasing a car by `rentCar` method, and terminating the deals when the car is returned by `terminateDeal` method. This class makes it possible for the client to sign a deal with which the reserved car can be delivered to the client for a determined time. The customer can decide on the start and end date/time. For simplicity and in order to clarify the DbC design in this model, the main field constraints are shown in Table 3 and a precondition, a postcondition, an invariant, and an inherited constraint are defined in Table 4 that should be implemented.

Table 3. Constraints on fields or properties of client, car, and deal

Client	<ul style="list-style-type: none"> • ID should not be an empty string or null • name should not be an empty string or null • phone should always be a valid phone number • address should not be an empty string or null
Car	<ul style="list-style-type: none"> • plateNumber should not be an empty string or null • model should not be an empty string or null • manufacturer should not be an empty string or null • fee should always be a positive integer • availability should always be true or false
Deal	<ul style="list-style-type: none"> • ID should not be an empty string or null • rentedCar should not be null • client should not be null • rentDays should always be a positive integer • totalFee should always be a valid number • startDate should always be a valid date and time • endDate should always be a valid date and time • status should always be either Open or Closed

Table 4. Precondition, Postcondition, Invariant, and Inherited constraint

Precondition for rentCar	Before renting a car, its availability should be true and the car must not be in another lease. An available car can be searched and seen by clients. Therefore, as a precondition to rentCar method, a car object's availability should be true when it is passed as a parameter.
Postcondition for rentCar	After renting a car, its availability should be set to false in order to prevent CarRentalSystem to offer it to other clients. rentCar method returns a deal and the deal keeps the rented car as a field. The method should guarantee that the rented car is not available as long as the deal is open.
Invariant in Deal class	When a deal is created the end date should always be later than start date and this is true for all deals on all instances of Deal objects or its extensions.
Constraint inheritance	A new class called InsuredDeal extends Deal and has an extra field called maximumInsuranceCover to keep the amount of insurance coverage. Since the new class keeps start date and end date for a deal, the same invariant for InsuredDeal should be inherited from Deal class.

Pre and post conditioning can be done by constraining rentCar method and the class invariant is applicable on Deal in which a proper period of contract has to be guaranteed. For contract inheritance, another class is required to realise the inheritance and check the contracting on sub class. For this reason, InsuredDeal class will be defined that represents a deal that supports insurance on its contract. This class extends Deal and has one extra field for storing maximum insurance coverage. Thus, by having some invalid inputs of super class the impact of validation on sub class can be evaluated.

The examination of each tool requires implementation of one field constraint from Table 3 and all constraints of Table 4. In this case, setting the plate number of a car as a null input value is designated to represent field constraint. Although coding can be done separately without using any particular tool, an IDE can make everything so smooth and integrate all required components more efficiently. In this case, Eclipse Neon 4.6.3 in combination with Maven will be used to program the mentioned system. The reason that why Maven is used is that Maven alleviates the automation of build and manages the dependencies in a more structured way.

4.2 Implementation

The implementation of tools in Eclipse requires the Maven plugin because the default project should be a Maven project with org.apache.maven.archetypes:maven-archetype-quickstart. The new created project looks like Figure 2.

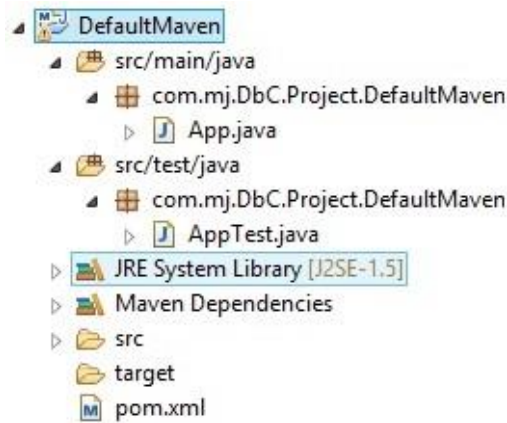


Figure 2. Default Maven Project for CarRentalSystem

The project includes src/main/Java for user defined classes and src/test/Java for defining user test classes. Each Maven project has a pom.xml file that holds the information of dependencies and build configuration. In order to ease the process of writing test classes, Maven adds jUnit dependency to the pom.xml automatically. However, to add the jUnit dependency manually the following dependency should be added:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Furthermore, maven compiler plugin should be added to the project by inserting some extra lines to pom.xml which is shown in Figure 3.

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.0.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-install-plugin</artifactId>
        <version>2.5.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-deploy-plugin</artifactId>
        <version>2.8.2</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

Figure 3. Maven compiler plugin requirements

Now for each entity introduced in 4.1 a Java class should be defined. Each class has its own fields, one constructor, getters and setters, and other methods when needed. To follow object-oriented programming concepts all fields are defined as private and other objects can have access to other objects' fields only by calling getters and setters. The appendix A shows all the classes.

On a platform such as Eclipse, testing the classes and DbC performance needs some unit test cases upon which fallibility of constraints can be scrutinised. Thus, CarRentalTest class implements eleven methods for testing different principles:

1. `plateNumberIsNotNull`: tries to instantiate a car with valid parameters as a positive test case.
2. `plateNumberIsNull`: tries to instantiate a car with a null plate number that violates field constraint
3. `carIsAvailable`: Trying to rent a car while it is available as a positive test case
4. `carIsNotAvailable`: Trying to rent a car while it is not available violates rentCar precondition
5. `rentedCarIsNotAvailable`: On quitting rentCar, the availability of a rented car is false and tested as a positive test case.
6. `rentedCarIsAvailable`: On quitting rentCar, the availability of a rented car registered for a specific deal should be false
7. `dealIsValid`: Tries to create a deal in which end date is after start date (normal period of time) as a positive test case.
8. `dealIsNotValid`: Tries to create a deal in which start date is after end date (abnormal period of time)
9. `insuredDealHasValidFields`: InsuredDeal extends Deal and thus has to call super with corresponding parameters. With having all parameters valid an insuredDeal is tested as a positive test case for inheritance.
10. `insuredDealHasInvalidFields`: InsuredDeal extends Deal and thus has to call super with corresponding parameters. Without constraining InsuredDeal's fields, this method attempts to pass an invalid parameter to sub class to test the inheritability of guards.
11. `aLongTestIncludingAllOtherCases`: to mimic the behaviour of a big code, a loop repeating for 10,000 times in which all positive test cases are tested. The reason for iterating the loop for 10,000 times is that having an unusual iteration like 12,365 is a burdensome to memorise rather than 10,000. Besides, more iterations, e.g. 100,000 or 1,000,000, seem to be infeasible for testing purposes.

Since the constraining of classes under each DbC extension is done according to the target tool's principles the implementation of DbC for each tool will be explained briefly.

4.2.1 Bean Validation

Bean Validation supports Eclipse and it is straightforward to set it up for development purposes. Although at first glance Bean Validation seems to be strenuous to be harnessed, getting accustomed with it is really simple. It is very powerful with plenty of features and full documentation support. The good point about this framework is that it provides a step by step user guide in a well-documented format, which helps to give a broad understanding about the API's principles.

Moreover, it lets to create custom constraints which is dramatically critical to developers who are willing to use it in real projects and need a variety of constraints. This flexibility makes Bean Validation a practically beneficial framework for software validation. Besides, modification of previously scripted constraints is quite simple without any interruption or interference to other coding elements. Changing annotation will affect quickly. Bean Validation provides an excellent documentation for user guidance purposes and API introduction which is a great aspect.

Initialisation

Using Bean Validation in a Maven project needs three dependencies:

1. Hibernate validator core, which transitively pulls the Bean Validation API into project.
2. Unified Expression Language (Unified EL), which evaluates the dynamic expressions when a constraint violation happens and its messages are created.
3. Contexts and Dependency Injection for Java (CDI), which adds integration points of Bean Validation with CDI, JSR 346.

Applying Constraints

Field constraining: constraining the fields was quite simple using Bean Validation because it provides a variety of built-in constraints. `@NotNull` can guard a string from being null conveniently. `@Positive` can guard an integer to be always a positive number. Therefore, by only adding the annotation to parameters of Car constructor, the constraining is done:

```
public Car(@NotNull String plateNo, @NotNull String model, @NotNull
String manufacturer, @Positive int fee) {
    ...
}
```

The same approach can be taken for setters as well:

```
public void setPlateNumber(@NotNull String plateNumber) {
    this.plateNumber = plateNumber;
}
```

Client, Deal, InsuredDeal, and CarRental have the same field constraining using this tool.

Preconditioning: Bean Validation lets the coder to define custom annotations and define a validator for the new annotation. To check that if a car is available before renting it, `@Available` is defined. In fact, any annotation consists of two parts; annotation code and

validation code, in this case, for @Available the associated code is shown in Figure 4 and Figure 5.

```
@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE, TYPE_USE })
@Retention(RUNTIME)
@Constraint(validatedBy = AvailableValidator.class)
@Documented

public @interface Available {

    String message() default
        "Car must be available.";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };

    @Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Available[] value();
    }
}
```

Figure 4. Annotation code for @Available

```
public class AvailableValidator implements ConstraintValidator<Available, Car>{

    @Override
    public void initialize(Available constraintAnnotation) {
    }

    @Override
    public boolean isValid(Car car, ConstraintValidatorContext constraintContext) {
        if ( car.isAvailable() == true ) {
            return true;
        }
        else
            return false;
    }
}
```

Figure 5. Validation code for @Available

This constraint is used in rentCar method as an independent custom constraint:

```
public Deal rentCar(@NotNull Client client, @NotNull @Available Car
car, @Positive @Max(10) int rentDays, Date start, Date end) {
    ...
}
```

If the parameter car is not available at calling rentCar, a constraint violation will be thrown.

Postconditioning: To check if the rented car is available when rentCar method is terminated, a new constraint called @RentedCarNotAvailable is defined and used:

```

@RentedCarNotAvailable

public Deal rentCar(@NotNull Client client, @NotNull @Available Car car,
@Positive int rentDays, Date start, Date end) {

...

}

```

In fact, this annotation checks if `Deal.getRentedCar().isAvailable()` is false or true and respectively acts normal or throws and error.

Setting Invariant: to define the invariant in Table 4 another annotation called `@ProperDate` is defined, which checks if end date is after start date always. The same as `@Available`, two separated parts defining constraint annotation and constraint validation has to be declared. The relevant code for `@ProperDate` is show in Figure 6 and Figure 7.

```

@Target({ TYPE, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = ProperDateValidator.class)
@Documented

public @interface ProperDate {

    String message() default
        "End date must be after start date.";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default {};

    @Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Available[] value();
    }
}

```

Figure 6. Annotation code for `@ProperDate`

```

public class ProperDateValidator implements ConstraintValidator<ProperDate, Deal>{

    @Override
    public void initialize(ProperDate constraintAnnotation) {
    }

    @Override
    public boolean isValid(Deal deal, ConstraintValidatorContext constraintContext) {
        if (deal.getEndDate().after(deal.getStartDate())) {
            return true;
        }
        else
            return false;
    }
}

```

Figure 7. Validation code for `@ProperDate`

Since this constraint is a class level invariant it should be before `Deal` class definition:

```
@ProperDate
public class Deal {

    ...

}
```

Constraint Inheritance: since InsuredDeal extends Deal it has to call super([params]) in its constructor and any constraint defined for Deal should work on InsuredDeal as well:

```
public InsuredDeal(String id, int days, int totalFee, Car rentedCar,
Client client, Date start, Date end, @Positive int maxInsCover) {
    super(id, days, totalFee, rentedCar, client, start, end);
    this.maximumInsuranceCover = maxInsCover;
}
```

Except maxInsCover that should be annotated manually, other fields don't have to be constrained due to contract inheritance. However, validating the inheritance is done at runtime and if any parameters are not valid an appropriate violation will be created.

Test Cases: According to the testing scenario described earlier, eleven test methods have to be implemented and validated. The cause in the negative test case making the code improper, is set as the correct code in positive test cases which works without errors. Bean Validation provides two types of validators; for non-executables validator and for executables validator. The former is suitable when the validation is performed on an object and the latter is useful when a method is going to be validated. The following code illustrates how to set up both validators in CarRentalTest class:

```
private static ExecutableValidator forExecutablesValidator;
private static Validator forNonExecutablesValidator;

@BeforeClass
public static void setUpValidator() {
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    forExecutablesValidator = factory.getValidator().forExecutables();
    forNonExecutablesValidator = factory.getValidator();
}
```

When a validator validates an element, it returns a set of possible constraint violations. Therefore, by asserting the size of the set, it is easy to monitor the validation and generate appropriate error messages. Thus, the first method that checks field constraining looks like:

```
@Test
public void plateNumberIsNull() {
    Car car = new Car(null, "508", "Peugeot", 60);
    Set<ConstraintViolation<Car>> violations =
    forNonExecutablesValidator.validate(car);
    assertEquals(1, violations.size());
}
```

Since the first parameter is null (implies that plate number is improper), the size of violations would be one after the validation of car. In the corresponding positive case with almost the same code only the plate number is not null so everything works correctly with no fault.

For testing the precondition the test case should be:

```
@Test
public void carIsNotAvailable() throws NoSuchMethodException {
    Car car = new Car( "DD-CE-456", "X6", "BMW", 120 );
    car.setAvailability(false);
    Client client = new Client("cl-2018-1003", "James", "1438-
Vertongen st.", "46324835");
    CarRental rental = new CarRental();
    Method method = CarRental.class.getMethod("rentCar",
Client.class, Car.class, int.class, Date.class, Date.class);
    Object[] params = {client, car, 5, new Date(), new Date()};
    Set<ConstraintViolation<CarRental>> violations =
forExecutablesValidator.validateParameters(rental, method, params);
    assertEquals(1, violations.size());
}
```

In this case, a method should be validated, therefore, a method with test parameters should be delivered to the validator. However, setting the car's availability to false before validating the rentCar causes one violation. The positive test case with same code does not violate the precondition so it works without any error. For testing postcondition of this method, a similar validator works well notifying that this time after creating a deal, the rented car's availability is set to true which causes a violation:

```
@Test
public void rentedCarIsAvailable() throws NoSuchMethodException {
    Date start = new Date();
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(start);
    calendar.roll(Calendar.DAY_OF_YEAR, 3);
    Date end = calendar.getTime();
    Client client = new Client("CL-2018-1001", "George", "1851-
Willington st.", "46322311");
    Car car = new Car("DD-CE-123", "XC60", "Volvo", 50);
    Method method = CarRental.class.getMethod("rentCar",
Client.class, Car.class, int.class, Date.class, Date.class);
    Deal returnedDeal = new Deal("DL-" + client.getID() + "-" +
car.getPlateNumber(), 3, 65, car, client, end, start);
    returnedDeal.getRentedCar().setAvailability(true);
    Set<ConstraintViolation<CarRental>> violations =
forExecutablesValidator.validateReturnValue(new
CarRental(), method, returnedDeal);
    assertEquals(1, violations.size());
}
```

Invariant validation requires to make an incorrect date sequence in a deal to disrupt the defined invariant. This can be done by swapping end date and start date, as a result, the validator detects the problem and makes a violation:

```

@Test
public void dealIsNotValid() {
    Date start = new Date();
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(start);
    calendar.roll(Calendar.DAY_OF_YEAR, 15);
    Date end = calendar.getTime();
    Client client = new Client("CL-2018-1001", "George", "1851-
Willington st.", "46322311");
    Car car = new Car("DD-CE-123", "XC60", "Volvo", 50);
    Deal deal = new Deal("DL-"+ client.getID() + "-" +
car.getPlateNumber(), 5, 100, car, client, end, start);
    Set<ConstraintViolation<Deal>> violations =
forNonExecutablesValidator.validate(deal);
    assertEquals( 1, violations.size() );
}

```

The positive test case for testing invariant does not swap the dates, thus the code is not generating any violations. Constraint inheritance can be tested by using an invalid input to test if propagated guards work consistently or not. In this case, delivering a negate integer for rent days to InsuredDeal is imaginable that forces the validator to result in a violation:

```

@Test
public void insuredDealHasInvalidFields() {
    Date start = new Date();
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(start);
    calendar.roll(Calendar.DAY_OF_YEAR, 5);
    Date end = calendar.getTime();
    Client client = new Client("CL-2018-1001", null, "1851-
Willington st.", "46322311");
    Car car = new Car("DD-CE-123", "XC60", "Volvo", 85);
    InsuredDeal deal = new InsuredDeal("DL-"+ client.getID() + "-"
+ car.getPlateNumber(), -5, 650, car, client, start, end, 30000);
    Set<ConstraintViolation<InsuredDeal>> violations =
forNonExecutablesValidator.validate(deal);
    assertEquals( 1, violations.size() );
}

```

The positive test case for inheritance has the code but with valid inputs, so the code is working with no errors. All other tools' test cases have the logic with a slightly different implementation. Therefore, there is no need to describe test cases each time for each framework and having the result of build and execution suffices.

Build and Execution

Compiling the project requires to right click on project and go to Run As then click on Maven Clean, which is ready by default. The result of building the project with Maven clean is shown in Figure 8.

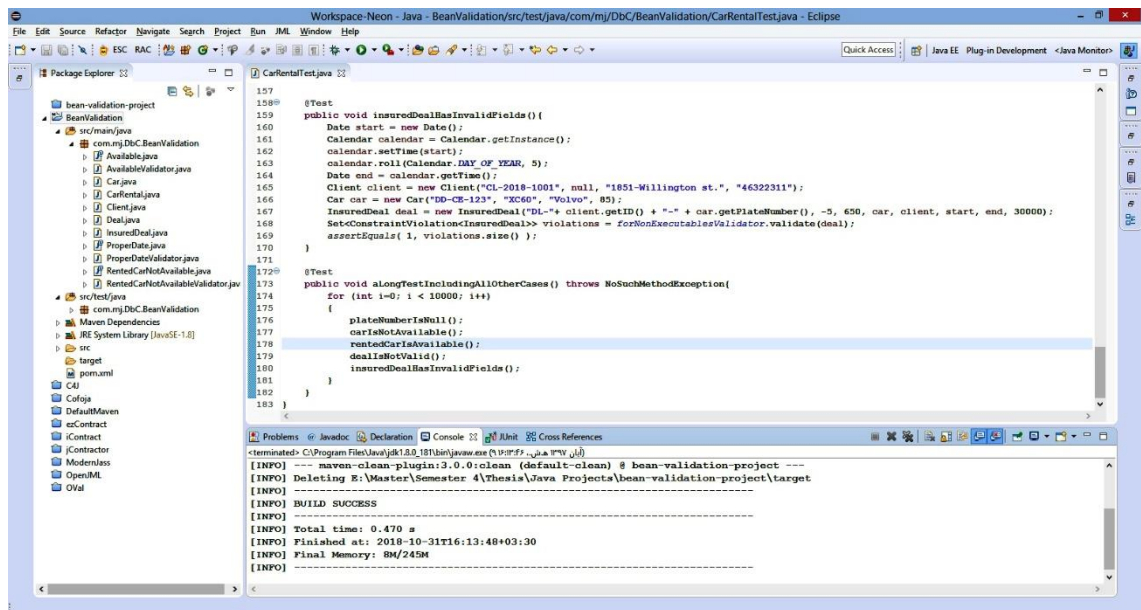


Figure 8. Building the project using Bean Validation and Maven clean goal

As it is presented by Eclipse compiler the total time is 0.470 s and memory consumption is 8 MB. Moreover, testing the project means that all test cases of CarRentalTest are executed using Maven test goal. The result is shown in Figure 9.

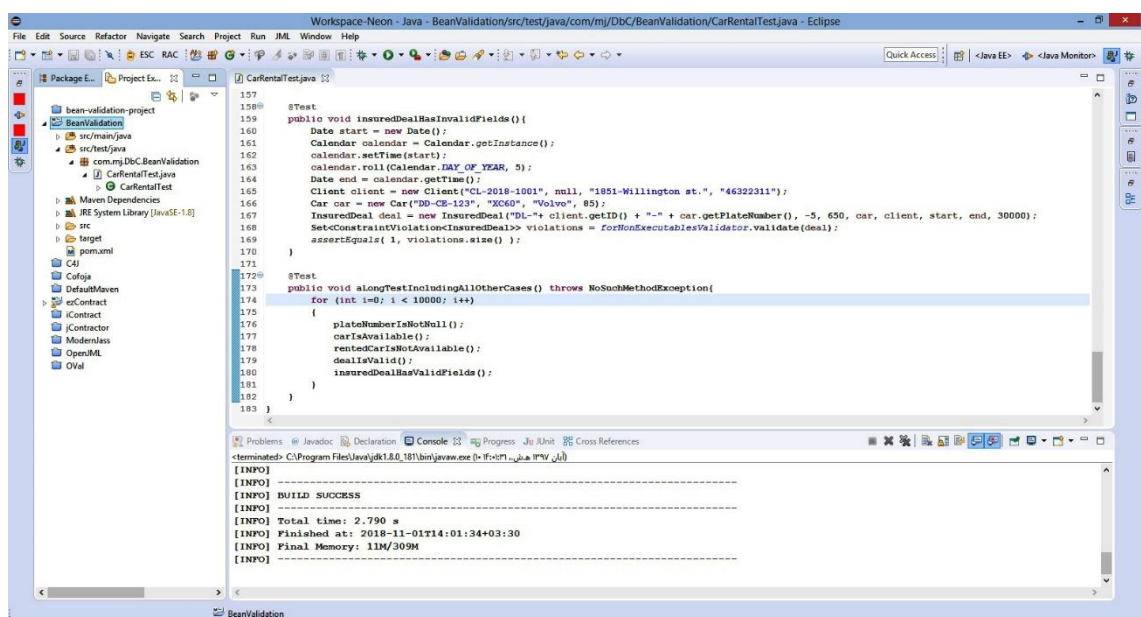


Figure 9. Testing the project using Bean Validation and Maven clean test goals

This time the execution time is 2.790 s and memory consumption is 11 MB.

4.2.2 C4J

With an easy to use approach C4J has fewer rules and complexity that facilitates the constraining dramatically. On one hand, adding constraints to a class definition is supported by default meaning the code looks concise and guards are reachable at the point where the logic is bound. On the other hand, being able to separate classes from contracts brings a higher modular programming in which one knows to find contract body and focus on its logic in an organised way without bearing the source code full of annotations. In

addition, modifying a contract does not disrupt the source code script. Moreover, C4J lets to specialise each assertion with a custom message without need of extra validation or violation handling at runtime; it just creates the contracts and generates error messages and it is done without concerning further violation management. However, adding more sophisticated constraints as custom annotations is not supported in C4J.

Initialisation

C4J can be used from the command line and also in Eclipse. To use it from command line the following line works well:

```
Java -ea -Javaagent:${c4j_loc}\c4j-6.0.0.jar [ClassName].Java
```

in which `${c4j_loc}` is the absolute file system path of C4J library, `[ClassName]` is the name of any defined class with having in mind that all the required libraries should be included in classpath. Moreover, it provides a plugin for Eclipse to ease the development process. However, running C4J under Eclipse needs to modify default VM arguments by deploying `-Javaagent` switch.

Additionally, using C4J in a Maven project needs three libraries:

1. `c4j-6.0.0.jar`, which is the core library for contracting in C4J
2. `Javaassist-3.16.1.jar`, which simplifies the manipulation of Java bytecode
3. `log4j-1.2.16.jar`, which is an Apache logging library for Java making it possible to login at runtime without modifying the application binary and with less performance penalty

Enabling a project to run C4J needs to add `-ea -Javaagent:${c4j_loc}\c4j-6.0.0.jar` to the VM arguments of the installed JRE in which `${c4j_loc}` is the absolute file system path of the core library.

Applying Constraints

For each class a contract class is defined to realise the contracting in practice. Each contract includes pre and postcondition and class invariant designated for the respective method. For example, for Car class a contract class called `CarContract` is defined and it overrides all methods of `Car`. In a same fashion, `CarContract`'s constructor acts as the overridden constructor of `Car`. `CarContract` class with its constraints is shown in Figure 10.

```

@Contract
public class CarContract extends Car {

    @Target
    private Car target;

    public CarContract(String plateNo, String model, String manufacturer, int fee) throws Exception{
        super(plateNo, model, manufacturer, fee);
        if (preCondition()) {
            assert plateNo != null: "Plate number must not be null.";
            assert model != null: "Model must not be null.";
            assert manufacturer != null: "Manufacturer must not be null.";
            assert fee > 0: "Rent fee must be greater than 0.";
        }
        if (postCondition()) {
            assert target.isAvailable() == true: "A new car must be available.";
        }
    }

    @Override
    public void setPlateNumber(String plateNumber) {
        if (preCondition()) {
            assert plateNumber != null: "Plate number must not be null.";
        }
    }

    public void setModel(String model) {
        if (preCondition()) {
            assert model != null: "Car model must not be null.";
        }
    }

    @Override
    public void setManufacturer(String manufacturer) {
        if (preCondition()) {
            assert manufacturer != null: "Car manufacturer must not be null.";
        }
    }

    @Override
    public void setFee(int fee) {
        if (preCondition()) {
            assert fee > 0: "Rent fee must be greater than 0.";
        }
    }
}

```

Figure 10. A contract class defined and bound to Car class

Field constraining: C4J only supports defining contracts including pre and postconditions and class invariants. Therefore, it doesn't offer distinctive field constraints. Instead, it supports defining separated contract files for each Java class. But in CarRentalSystem constraining a field of a class has the same way of preconditioning. Indeed, the precondition of CarContract's constructor checks if a field is valid or not.

Preconditioning: Since CarRental class owns method rentCar, the associated contract class overrides it setting the relevant precondition for Car's availability. Related code looks like the following lines:

```

@Override
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    if (preCondition()) {
        ...
        assert car.isAvailable(): "Car must be available to be
rented.";
        ...
    }
    ...
}

```

Postconditioning: To specify the postcondition for rentCar method, the overridden rentCar method in CarRentalContract should define the proper postcondition:

```

@Override
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    ...
    if (postCondition()) {
        Deal result = result();
        assert !result.getRentedCar().isAvailable(): "Rented
car must not be available as long as it is rented by another client.";
    }
    ...
}

```

Setting Invariant: specifying Invariant follows the same rule as pre and post condition described above. In this case, for DealContract a method defining the invariant is declared. It should be noted that C4J detects the invariant by @ClassInvariant annotation used before the method:

```

@Contract
public class DealContract extends Deal{

    @Target
    private Deal target;
    ...
    @ClassInvariant
    public void invariant() {
        assert target.getEndDate().after(target.getStartDate()): "End
date must be after start date.";
    }
    ...
}

```

the `target` variable stands for a reference to the guarded object.

Constraint Inheritance: Presumably any constraint defined on Deal should be propagated to InsuredDeal that is testable at runtime. Since InsuredDeal has one field that should always be a positive integer, InsuredDealContract is defined to guard `maximumInsuranceCover` field. However, calling the `super([params])` in the constructor causes the inheritance propagation and if any parameters are not valid at runtime an appropriate violation will be created.

Test Cases: C4J alleviates the validation of constraints meaning that by defining a contract there is no need to add extra validators in test methods. The reason is using `assert` command in defining a contract throws an exception if any constraint is violated. For example, the following code throws an exception because one parameter is against the precondition:

```

@Test
public void plateNumberIsNull() {
    Car car = new Car(null, "508", "Peugeot", 60);
}

```

Other test methods follow the same rule.

Build and Execution

As described in 4.3.1.3 compiling and running a project is quite similar to what has been done using Bean Validation. The result of building the project with Maven annotated by C4J is shown in Figure 11.

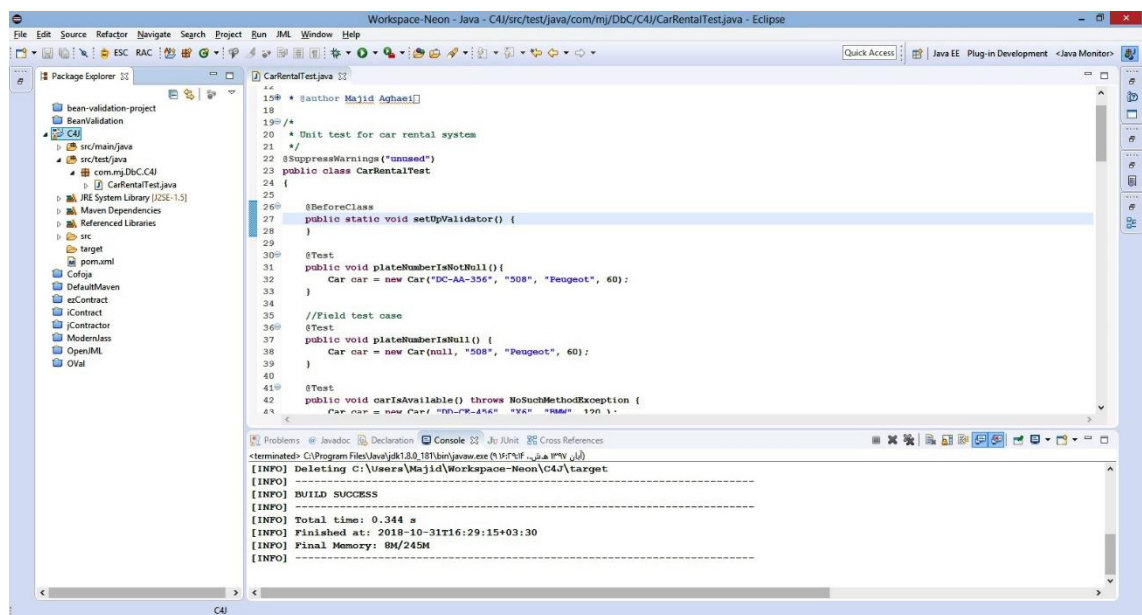


Figure 11. Building the project using C4J and Maven clean goal

The total build time is 0.344 s with using 8 MB of memory. The results of testing the application using Maven test goal is shown in Figure 12.

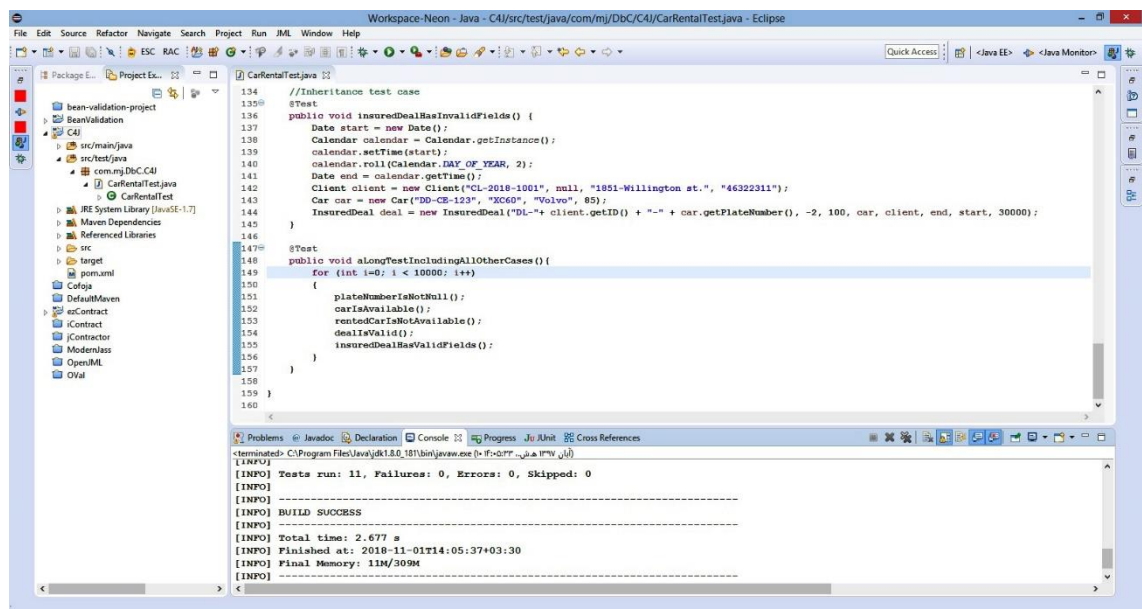


Figure 12. Testing the project using C4J and Maven clean test goals

As it is exhibited, the total testing time is 2.677 s and it claimed 11 MB of memory.

4.2.3 Cofoja

Using Cofoja simplifies contracting as well. It concentrates on contracting with fewer lines of code and less complexity. It is possible to mix several expressions in one line and set it as a precondition but it forsakes customisability. Since preconditions are made of string expressions, a tiny mistake in typing an expression may generate ambiguous situation and complicated diagnostic with a long time error recovery. Furthermore, initialisation of Cofoja is a little bit confusing and needs higher understanding of Eclipse. Indeed, it does not configure its own prerequisites automatically and puts the duty to the coder! Any mistake in initialisation will lead to plenty of compile or runtime errors.

Initialisation

Deploying Cofoja requires two libraries:

1. cofoja-1.3-20160207.jar, which is the core library for Cofoja
2. ASM 5.x or higher for bytecode instrumentation

Empowering a Java project to be able to utilise Cofoja needs to do some configurations. By enabling project specific setting in Java compiler->annotation processing, one should add three processor options:

1. Key= com.google.java.contract.classoutput, value= %PROJECT.DIR%/bin
2. Key= com.google.java.contract.classpath, value = %PROJECT.DIR%/lib/cofoja.asm1.320160207.jar
3. Key= com.google.java.contract.sourcepath, value= %PROJECT.DIR%/src

Furthermore, in Java compiler->annotation processing->factory path, cofoja library should be added to the plugins, and the `-ea -Javaagent:${cofoja_loc}/cofoja-1.3-20160207.jar`, in which `${cofoja_loc}` is the absolute file system path of the core library, has to be included in VM arguments when the project is going to be compiled or tested.

Applying Constraints

Field constraining: Cofoja the same as C4J, doesn't support field annotating separately but it is possible to constrain fields through preconditions of constructors or getters and setters. For example, constraining the `plateNumber` field of `Car` is possible through defining precondition for the constructor and the associated setter of `plateNumber`:

```
@Requires({"plateNo != null", "model != null", "manufacturer != null",
"fee > 0"})
public Car(String plateNo, String model, String manufacturer, int fee)
{
    this.plateNumber = plateNo;
    this.model = model;
    this.manufacturer = manufacturer;
    this.fee = fee;
    this.availability = true;
}
```

It prevents the entered string from being null when an object is instantiated. And for the setter:

```

@Requires("plateNumber != null")
public void setPlateNumber(String plateNumber) {
    this.plateNumber = plateNumber;
}

```

So far, the field constraining is done indirectly.

Preconditioning:

Using `@Requires` it is possible to add a precondition prior to a method. In our scenario, a car should be available before calling the `rentCar` method. Therefore, the related code should be according to Cofoja style:

```

@Requires("car.isAvailable()")
...
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    ...
}

```

However, if Cofoja detects that the availability of parameter `car` is false, an error will be thrown at runtime. For other methods scripting a precondition and its validation follows the same rule.

Postconditioning: When method `rentCar` is executed successfully, the created deal's rented car must not be available anymore:

```

@Ensures("!result.getRentedCar().isAvailable()")
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    ...
}

```

Setting Invariant: specifying an invariant is quite easy in Cofoja. For our scenario, to say Cofoja that end date must be after start date of a deal, an invariant should be defined before class definition of `Deal`:

```

@Invariant("endDate.after(startDate)")
public class Deal {
    ...
    private Date startDate;
    private Date endDate;
    ...
}

```

Constraint Inheritance: Like C4J, in Cofoja all the constraints are inherited when a class extends another class. Therefore, apart from `InsuredDeal`'s own fields' constraints no special guard needs to be added.

Build and Execution

The result of building the project with Maven annotated by Cofoja is shown in Figure 13.

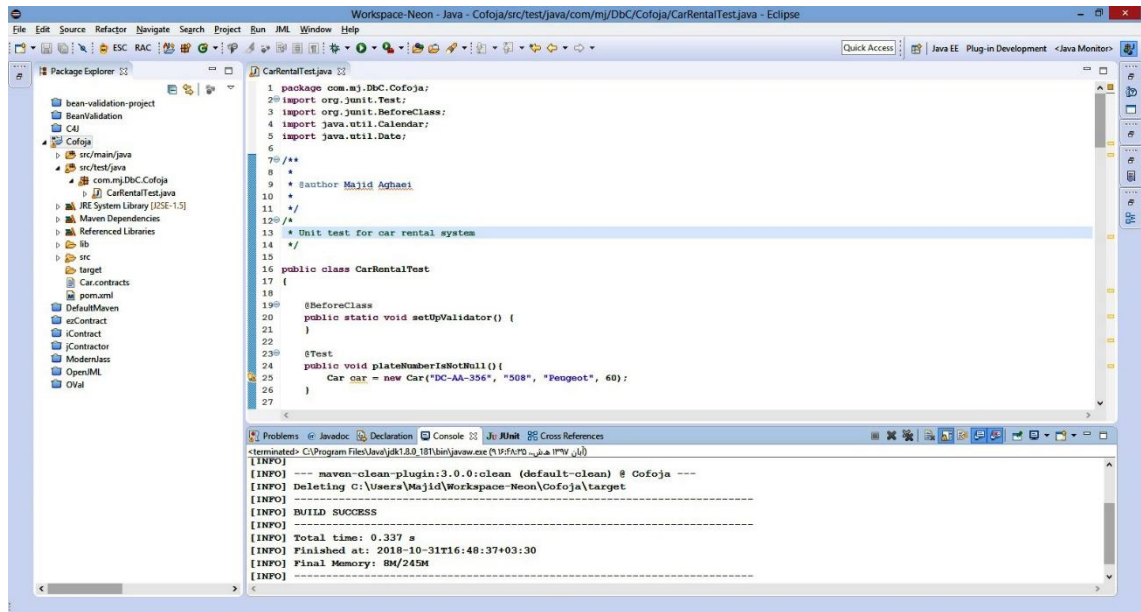


Figure 13. Building the project using Cofoja and Maven clean goal

Total compilation time is 0.337 s and the total memory consumption is 8 MB. Running the application under test cases generated another result, which is shown in Figure 14.

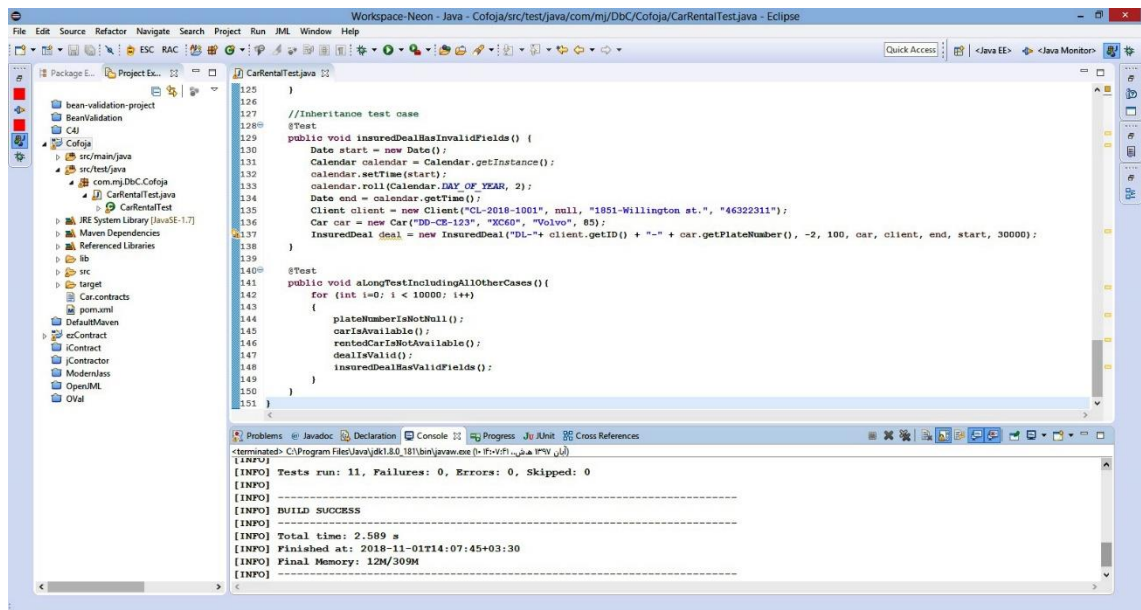


Figure 14. Testing the project using Cofoja and Maven clean test goals

As it is generated by Eclipse, the test time is 2.589 s and the project consumed 12 MB of memory.

4.2.4 ezContract

ezContract can be considered as an easy to use tool. It proposes a smooth way of contracting by making a constraint packed in a {begin, end} clause. Like C4J, it is not

difficult to define and specify violation reporting messages in ezContract due to directly use of assertions in a clause. Thus, testing the code and annotations do not require any further validation or monitor at runtime. One of this framework's pros is that a constraint' logic can be distinguished from other elements quickly without so much effort and modifying it is straightforward as well without unsettling the source code.

Initialisation

Utilising ezContract needs to add some external libraries to project:

1. ezContract-0.01a.jar, which is the core library for ezContract
2. Javassist-ctchen.jar, which is an old version of Javaassist for bytecode manipulation
3. commons-0.01a.jar, which is an assisting library for processing strings and working with files

a noticeable drawback of ezContract is that it does not support Eclipse by default and all operations should be done through command line, which makes the development obscure. But with spending some time it is possible to code in Eclipse environment and at least compile a project deploying ezContract.

Applying Constraints

Field constraining: field constraining is not directly plausible in ezContract but it can be done using preconditions and constraining getters and setters the same as describe for Cofaja. For example, since the fields of Car are not directly accessible by other objects, constraining the fields can be done using preconditioning of Car's constructor:

```
public Car(String plateNo, String model, String manufacturer, int fee)
{
    Require.begin();
        assert plateNo != null: "Plate number must not be null.";
        assert model != null: "Model must not be null.";
        assert manufacturer != null: "Manufacturer must not be null.";
        assert fee > 0: "Fee must be greater than 0.";
    Require.end();
    this.plateNumber = plateNo;
    this.model = model;
    this.manufacturer = manufacturer;
    this.fee = fee;
    this.availability = true;
}
```

Preconditioning:

In ezContract checking the status of Car's availability before renting it is done using the following code snippet:

```

public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    Require.begin();
        assert car.isAvailable(): "Car must be available to be
rented.";
    Require.end();
    ...
}

```

Postconditioning: to guarantee that the return of rentCar method ensures that the rented car is not accessible by other clients while it is in another deal, an ensure marker clause should be added:

```

public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    Ensure.begin();
        Deal target = (Deal) Result.value;
        assert !target.getRentedCar().isAvailable(): "A rented car
must not be available.";
    Ensure.end();
    ...
}

```

target is a keyword known to ezContract as a reference to the object that should be guarded by the written assertion.

Setting Invariant: if the coder needs to declare an invariant for a class, ezContract proposes a different method to other tools. The target class should implement IContract interface, which forces the coder to implement classInvariant method in class body. In our scenario, the agreed invariant for Deal is scripted as:

```

public class Deal implements IContract{
    ...

    public void classInvariant() {
        assert endDate.after(startDate): "End date must be after start
date.";
    }
    ...
}

```

Constraint Inheritance: Like other tools, it is assumed that transferring a constraint to a sub class influenced that manner of that class at runtime. Therefore, except new fields defined in sub class that need new constraints fields from the super class are guarded in advance. However, the code does not differ from other tools.

Test Cases: test cases for ezContract are the same as previous extensions that do not need to validate a method or object separately, e.g. C4J. it should be noted that in test classes there is no need to add extra assertions in test methods' body since ezContract uses assert command in declaring contract elements.

Build and Execution

Compiling and running ezContract needs the following steps:

1. Compiling the source files as:

```
Java -cp %CLASSPATH%; ezContract-0.01a.jar; -sourcepath MyClass.Java
```
2. Instrumenting the class file generated from the previous step as:

```
Java -cp %CLASSPATH%;ezContract-0.01a.jar;Javassist-ctchen.jar;commons-0.01a.jar; ezcontract.core.instrument.Instrumentor MyClass
```
3. Running contracted files as:

```
Java -ea -cp %CLASSPATH%;ezContract-0.01a.jar;Javassist-ctchen.jar;commons-0.01a.jar; ezcontract.core.instrument.Instrumentor MyTestClass
```

4.2.5 iContract

iContract simplifies the contracting very much and the annotations are short and concise. It uses the commenting format for adding constraints to a project which is both acceptable and risky. Its advantage is that it is easy to find in the code due to being a comment and it is easy to understand but its weakness is if the programmer writes a parameter or a field's name with some typos, it might be very tedious to catch syntax errors because Java compiler ignores them and all the lines and comments should be checked character by character and manually. Furthermore, it does not support Eclipse making the usage of the framework difficult because all the steps have to be done by command line.

Initialisation

Initialising a project with iContract in Eclipse is not automated by the framework inventor but it is possible to add its core library and code in an IDE such as Eclipse. However, a project using iContract requires to have icontract2.jar as a referenced library and add it to the class path when it is time to compile the project. In fact, although using iContract's constraining approach is easy to catch and straightforward, working with iContract does not seem to be simple and easy to go!

Applying Constraints

Field constraining: in iContract, there is no annotation mark because it only deals with comments like constraints. Therefore, to restrict a field using encapsulations rules and preconditioning a class constructor seem to be effective. For instance, limitation of Car fields is done using the following code:

```
/*
 * @pre plateNo != null
 * @pre model != null
 * @pre manufacture != null
 * @pre fee > 0
 */
public Car(String plateNo, String model, String manufacturer, int fee)
{
    ...
}
```

Preconditioning: to make the agreed precondition for `rentCar`, adding a few lines suffices to monitor a car's availability before renting it:

```
/*
 * @pre car.isAvailable() == true
 */
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    ...
}
```

Postconditioning: on a same method taken to impose the precondition, the described postcondition can be set as:

```
/*
 * @post @return.getRentedCar().isAvailable == false
 */
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
    ...
}
```

`@return` refers to the return value of the target method, which here is the deal created by `rentCar`.

Setting Invariant: implementing the invariant for `Deal` is before the class declaration:

```
/*
 * @inv endDate.after(startDate)
 */
public class Deal {
    ...
}
```

Constraint Inheritance: In `iContract` all constraints of a father class will be applied to children. Thus the `InsuredDeal` class must satisfy the constraints of `Deal`.

Test Cases: test classes for `iContract` do not need particular validation means, therefore test methods used for last tools works well for `iContract` as well.

Build and Execution

Compiling and building a project in `iContract` needs some steps which is not viable in Eclipse. However, the steps are:

1. Compiling classes without contracts: it needs the source classes and other dependencies to be added to class path.
2. Generating configuration files, instrumented and repository sources: it needs `iContract2.jar`, `log4j.jar`, other regular dependencies, config classes and repository classes of contract-enabled dependencies to be added to class path.
3. Compiling instrumented files to contracted classes: it needs config classes of contract-enables dependencies and other regular dependencies to be added to the class path

4. Compiling repository resources to classes: it needs repository classes of contract-enabled dependencies along with other regular dependencies to be added to the class path.

Since the mentioned steps should be done manually, building and testing the project using iContract in Eclipse has not been performed.

4.2.6 jContractor

Using jContractor is simple without complexity and it provides a separation of contracts and source code, which is a way to make the code more understandable. Indeed, for each class there would be a contract class that extends the source class. This tool does not support annotating style and only focuses on contracting through using methods for each element of DbC, e.g. defining a method for precondition of a source method in the source class and define the expressions that should be guarded. This style of constraining is not so clear because one has to find the correspondent contract method for the original source method and catch the logic of the constraint by comparing both methods, which is a time consuming process. Since there is no annotation in jContractor, constraints are not reusable.

Although jContractor is straightforward with less complexity of deploying in code, it does not support Eclipse and the compiling and instrumenting should be done step by step and manually.

Initialisation

Utilising jContractor requires only its own core library that should be added as a regular dependency to class path for contract-enabled classes.

Applying Constraints

Field constraining: as illustrated above, no field constraining is adopted by jContractor and guarding a field in only applicable by constraining constructors, getters and setters. For example, restriction of plateNumber in Car is implemented using the following lines:

```
public class Car_CONTRACT extends Car {

    public Car_CONTRACT(String plateNo, String model, String manufacturer,
        int fee) throws Exception{
        super(plateNo, model, manufacturer, fee);
    }
    protected boolean Car_CONTRACT_Precondition(String plateNo, String
        model, String manufacturer, int fee){
        return (plateNo != null) && (model != null) && (manufacturer
        != null) && (fee > 0);
    }
    protected boolean setPlateNumber_Precondition(String plateNumber) {
        return plateNumber != null;
    }

    ...
}
```

Guarding `plateNumber` to be a valid string has been checked in `Car_CONTRACT_Precondition` method which controls the act of the constructor. In addition, the constraint method of the `plateNumber`'s setter is guarded in a same manner as well.

Preconditioning: Setting the designed precondition needs to add the contract method in `CarRental_CONTRACT` class:

```
public class CarRental_CONTRACT extends CarRental{
protected boolean rentCar_Precondition(Client client, Car car, int
rentDays, Date start, Date end) {
    return car.isAvailable();
}
...
}
```

Postconditioning: to check that if the new deal has a car with false availability the constraint looks like:

```
public class CarRental_CONTRACT extends CarRental{
...
protected boolean rentCar_Postcondition(Client client, Car car, int
rentDays, Date start, Date end, Deal RESULT){
    return !RESULT.getRentedCar().isAvailable();
}
...
}
```

`RESULT` refers to the returned deal of `rentCar` method.

Setting Invariant: defining the invariant for `Deal` requires to declare a method with name "`_Invariant`", which does not accept any arguments, in a correspondent contract class as:

```
protected boolean _Invariant(){
    return getEndDate().after(getStartDate());
}
```

Constraint Inheritance: Inheritance of contracts does not need any extra effort because all the contracts are inherited automatically.

Test Cases: test cases for `jContractor` are the same as previous extensions that do not need to validate a method or object separately, e.g. C4J. it should be noted that in test classes there is no need to add extra assertions in test methods' body since `jContractor` throws an error for each violation and reports an informative message.

Build and Execution

`jContractor`'s build and execution differs from other tools because it needs to run `jContractor` passing the target class to it, for example:

```
Java jContractor [options] Car.class
```

In addition, adding the contract code to the class files is possible using jInstrument program as:

Java jInstrument Car.class

since jContractor is not working in Eclipse, there is no build and test experience to be exhibited.

4.2.7 Modern Jass

Modern Jass provides a set of predefined constraints as annotations and this makes everything better and quicker due to less coding and manually constraining. Using Modern Jass is simple with no complexity. Adding the library to the project and using it will help to guard methods and fields and implement DbC wherever is required. The framework lets to define custom error messages where the constraints is going to be declared that leads to more flexible and user friendly validation tool. Unfortunately, it only supports Eclipse in Mac OS and its official website has not yet provided any plug in for windows. Therefore, compiling and running a program is merely possible for Mac users. A noticeable drawback of this tool, alike Cofaja, is using string expressions in constraints. In this case, any user flaw in scripting the expression may impede the coding and compiling with ambiguous errors. Additionally, the inventors have not articulated all the required information needed for the user to work with the framework. In fact, the documentation is not well-formed and there is no details for a programmer to deeply understand the principles.

Initialisation

Coding can be started with no particular initialisation and adding the core library as a dependency in class path suffices to start a project deployed by Modern Jass. However, using it needs to work with command line and repeat a task many times for building and running a project.

Applying Constraints

Field constraining: Thanks to providing some built-in constraints it is possible to constrain a field using an annotation. Car's plateNumber can be restricted using @NonNull in the class definition:

```
private @NonNull String plateNumber;
```

Other fields for which there is no proper built-in constraint restricting constructor's parameters and getters and setters will work correctly, as described for previous tools.

Preconditioning: Enforcing that a car must be available is done as the following code states:

```
@Pre("car.isAvailable()")
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
...
}
```

Postconditioning: right after defining the precondition in `rentCar` the postcondition comes to specify the unavailability of a rented car:

```
@Post("!!@Result.getRentedCar().isAvailable()")
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
...
}
```

`@Result` refers to the new deal instantiated by the method and returned for further processes.

Setting Invariant: the invariant has to be defined before class definition:

```
@Invariant(value = "getEndDate().after(getStartDate())", msg = "End
date must be after start date.")
public class Deal {
...
}
```

The expression is assigned to value field and the custom error message is assigned to msg field of the annotation.

Constraint Inheritance: as explained for previous frameworks, all the constraints are inherited automatically and the tool is responsible to check the inherited constraints at runtime.

Test Cases: test cases for Modern Jass are the same as for previous tools with having in mind that there is no need to do extra validation at the execution time.

Build and Execution

Compiling a Java class is done using the Javac and Modern Jass core library in class path:

```
Javac -cp ModernJass.jar CarRenta.Java
```

Modern Jass will create `CarRental.class` contracted, which can be run by Java and using Modern Jass as the Javaagent:

```
Java -Javaagent:lib/ModernJass.jar CarRental.class
```

4.2.8 Open JML

Open JML can be considered as a sophisticated and powerful extension including a variety of features each of which is very important for validation process. This tool provides static checking, runtime assertion checking, test generators, etc. and supports Eclipse particularly in Mac OS. Open JML has established a graphical plugin for Eclipse that shows warnings and errors of constraining the source code automatically when user saves Java files or presses Ctrl+S. Utilising Open JML is easy specially when integrated with Eclipse. Moreover, this tool provides a user guide, of course it is not complete and most of the parts are missing, that explains main concepts and related principles of Open JML. Like some other tools, Open JML uses commented constraints in its own way,

which can be problematic if user makes a trivial mistake. But using the GUI version all errors are alerted in red colour and the warnings are notified perfectly.

Initialisation

For enforcing the constraints in command line only imposes having Java 8 and being careful about setting the class path correctly and doing the compiling flawlessly. Furthermore, Eclipse plugin can be installed using update site link (<http://jmlspecs.sourceforge.net/openjml-updatesite>) according to plugin installation rules in Eclipse. After adding the plugin, a menu item (JML) and a toolbar item (ESC for static checking and RAC for runtime assertion checking) will be added to the development environment. This will result in having the red and yellow markers at coding time for alerting errors or warnings. However Open JML is dependent on three external libraries:

1. Openjml.jar, which is the core library
2. jmlspecs.jar, which helps to add JML specifications in a Java class
3. jmlruntime.jar, which will make the runtime assertion checking possible

in addition, for developers who need static checking capability, a SMT solver should be added to ESC part of the Open JML. A solver performs the proofing obligations of the specifications and behaviour of the scripted program. However, using the static checking is out of the scope of this research.

Applying Constraints

Field constraining: field constraining is applicable through using parameter and getter and setter constraining in JML style. To constrain Car's plateNumber, the following code snippet will work fine:

```
/*@
  @ requires plateNo != null;
  @ requires model != null;
  @ requires manufacturer != null;
  @ requires fee > 0;
  */
public Car(String plateNo, String model, String manufacturer, int fee)
{
  ...
}
/*@ requires plateNumber != null;
public void setPlateNumber(String plateNumber) {
    this.plateNumber = plateNumber;
}
```

Preconditioning: to specify the precondition according to Open JML format:

```
/*@
  @ requires car.isAvailable();
  */
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
  ...
}
```

Postconditioning: to specify the postcondition according to Open JML format:

```

/*@
  @ ensures !\result.getRentedCar().isAvailable();
  @*/
public Deal rentCar(Client client, Car car, int rentDays, Date start,
Date end) {
  ...
}

```

Setting Invariant: adding the invariant into Deal class is done inside the definition of the class:

```

public class Deal {
  //@ public invariant getEndDate().after(getStartDate());
  ...
}

```

Constraint Inheritance: the same as explained for other tools, all the constraints are inherited automatically and the tool will check them at runtime.

Test Cases: test cases for Open JML are the same as for previous tools with having in mind that there is no need to do extra validation at the execution time.

Build and Execution

Compiling and running a program written under Open JML constraints can be performed either in command line or in Eclipse. Compiling in command line requires to add openjml.jar as the core library to build the contracted class files and running the code requires to put jmlruntime.jar into class path and check the results. Building a Maven project deploying this tool needs to add all the dependencies in class path or add the plugin as mentioned earlier. Compiling the project claims the similar steps to other tools using a clean goal which is shown in Figure 17.

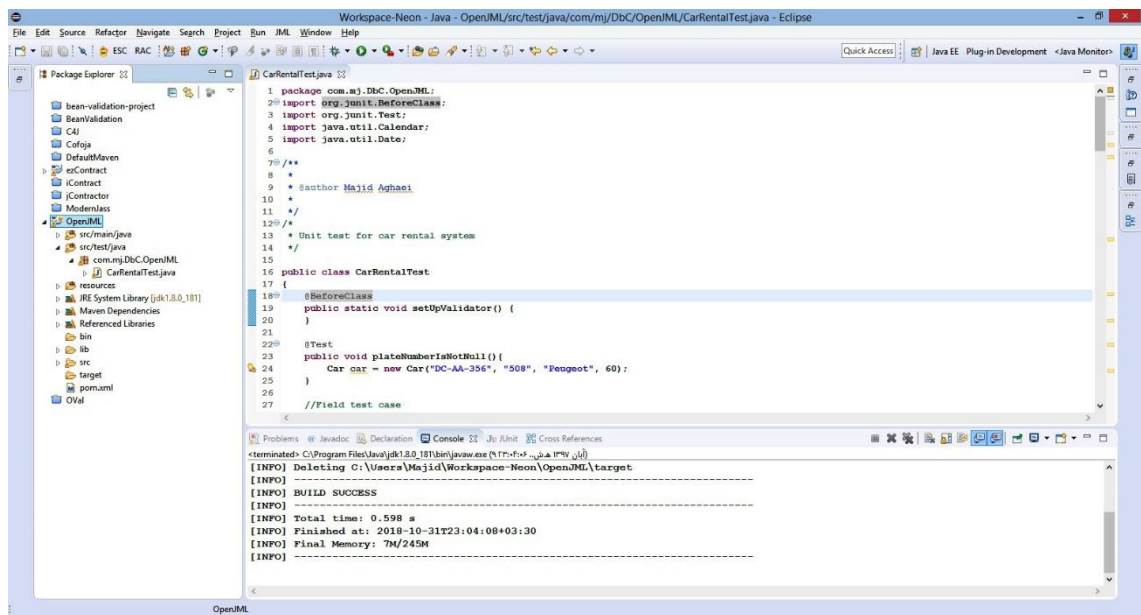


Figure 16. Building the project using Open JML and Maven clean goal

As Eclipse console reports the compilation time is 0.598 s and the project used 7 MB of memory. The execution of the project to run test methods created different result which is shown in Figure 17.

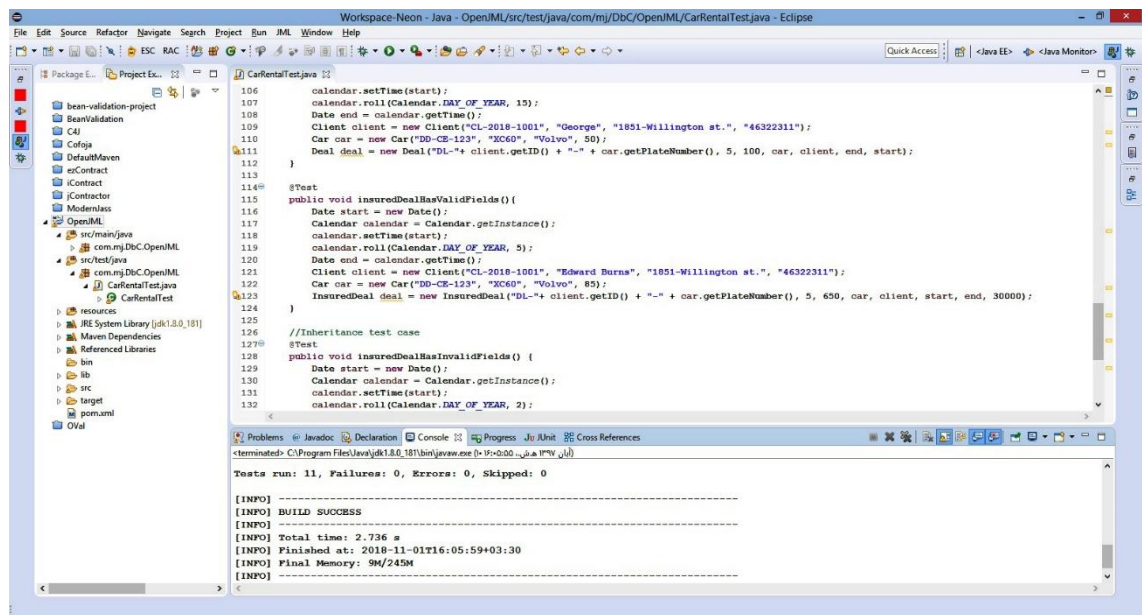


Figure 17. Testing the project using Open JML and Maven clean test goals

Running the test cases took 2.736 s and it consumed 9 MB of memory.

4.2.9 Oval with AspectJ

The final tool is a powerful extension and is somewhat similar to Bean Validation but also with remarkable differences. When Oval is used along with AspectJ it realises the DbC. AspectJ utilises the aspect-oriented programming paradigm in which defining an aspect one could modularise the concerns about a program at certain points and write some code snippets to be executed when those points are reached. However, Oval lets to define custom constraints the same as described for Bean Validation. Wherever a constraint is used in source code, a pointcut should be defined in the corresponding aspect. In this way, when Oval reaches to that pointcuts checks the validation of the field or parameter. The AspectJ plugin can be installed in Eclipse and easily taken to be paralleled with Oval. Thus, using Oval is straightforward with smooth coding and validation. However, declaring aspects is complicated and needs more dominance on Aspect-oriented programming.

Initialisation

Setting up a project in Eclipse and converting it to an AspectJ project is the easiest way to start using Oval. In addition, oval-1.87.jar is critical as a referenced library that has to be in class path. Finishing these steps one can code and add Oval constraints to the classes.

Applying Constraints

To guard a class and enable runtime checking with AspectJ, before any class declaration `@guarded` is required. Furthermore, for each constraint, a pointcut should be set in a related aspect.

Field constraining: Oval is able to add and check annotated constraints to fields quite the same as Bean Validation. Therefore, exact annotations defined in Bean Validation Project can be imported and utilised in this framework with trivial changes. For guarding Car's `plateNumber` `@NotNull` matches thoroughly:

```
private @NotNull String plateNumber;
```

to enable the constraint CarAspect should be:

```
public aspect CarAspect extends GuardAspect{
    pointcut callConstructor(String plateNo, String model, String
        manufacturer, int fee):
        call(void Car.Car()) && args (plateNo, model, manufacturer,
        fee);
    pointcut callSetPlateNumber(String plateNo):
        call(void Car.setPlateNumber()) && args (plateNo);
    ...
}
```

Other fields have the same situation and the implementation does not differ from `plateNumber`.

Preconditioning: in a similar implementation described for Bean Validation, precondition designated for `rentCar` can be realised by adding `@Available`:

```
public Deal rentCar(Client client, @Available Car car ,int rentDays,
    Date start, Date end) {
    ...
}
```

Definition and validation of `@Available` has been demonstrated in Figure 4 and Figure 5.

However, the associated aspect is like:

```
public aspect CarRentalAspect extends GuardAspect{
    pointcut callRentCar(Client client, Car car ,int rentDays, Date start,
        Date end):
        call(Deal CarRental.rentCar()) && args (client, car, rentDays,
        start, end);
}
```

Postconditioning: `@RentedCarNotAvailable` enforces `rentCar` to be ensured of the status of the `rentedCar` to be false:

```
@RentedCarNotAvailable
public Deal rentCar(Client client, @Available Car car ,int rentDays,
    Date start, Date end) {
    ...
}
```

The pointcut defined for precondition will work for the postcondition as well.

Setting Invariant: the invariant can be checked by adding `@ProperDate` before class definition:

```
@Guarded
@ProperDate
public class Deal {

    ...

}
```

Definition and validation of `@ProperDate` has been demonstrated in Figure 6 and Figure 7.

Constraint Inheritance: as explained for previous frameworks, all the constraints are inherited automatically and the tool is responsible to check the inherited constraints at runtime.

Test Cases: test cases for Oval are the same as for previous tools with having in mind that there is no need to do extra validation at the execution time.

Build and Execution

Compiling Oval with Maven only needs to run the project under Maven clean goal which is shown in Figure 18.

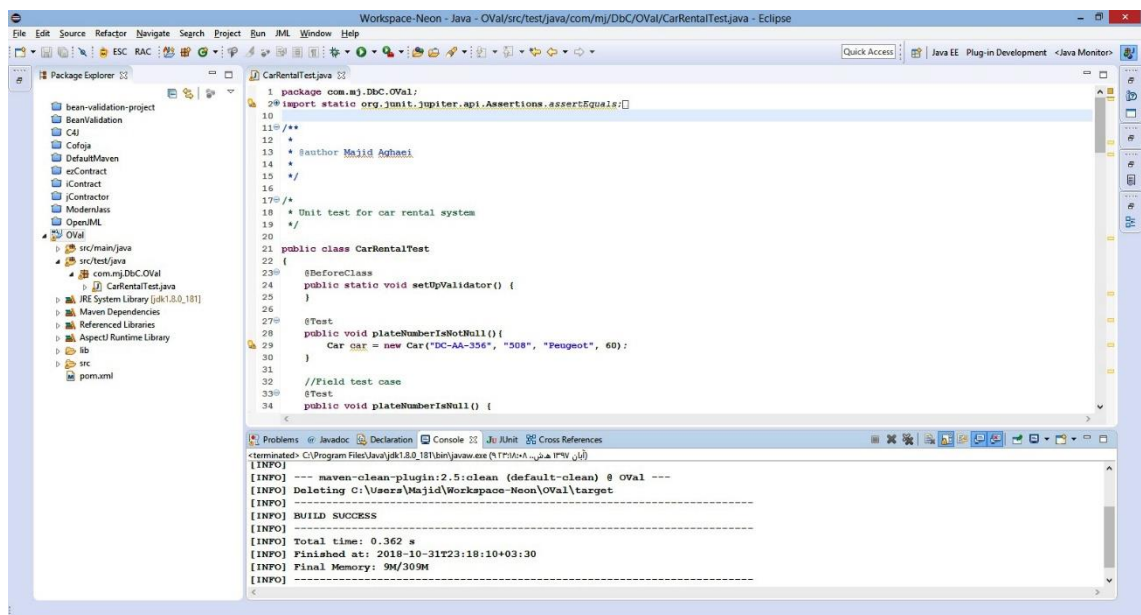


Figure 18. Building the project using Oval and Maven clean goal

Building elapsed for 0.362 s and the environment claimed 9 MB of memory in this phase. The results of testing the project under unit testing with Maven clean test goals is displayed in Figure 19.

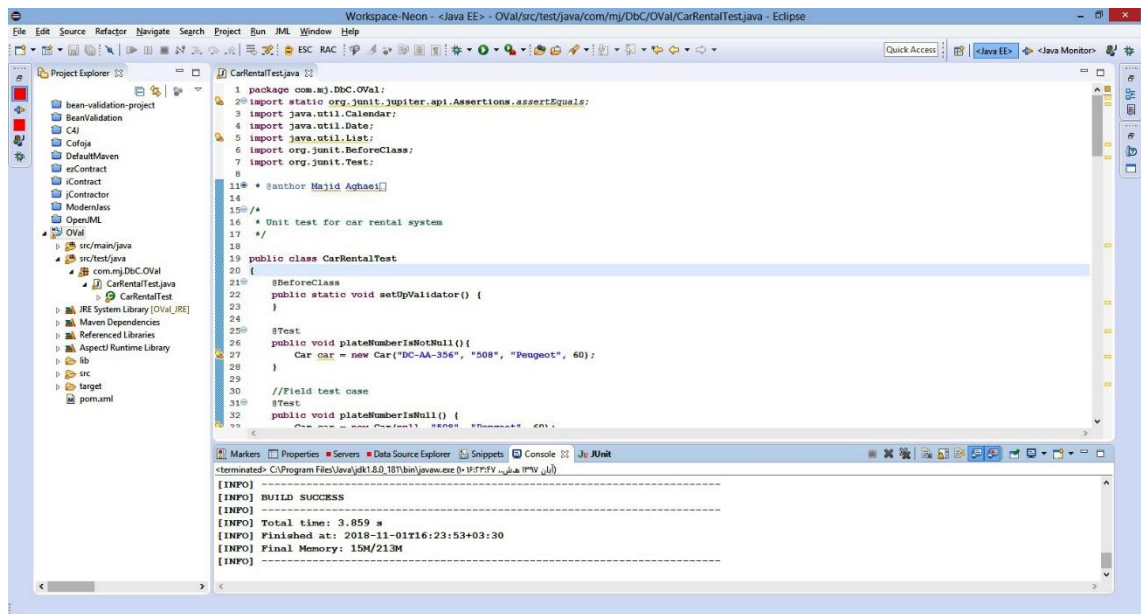


Figure 19. Testing the project using Oval and Maven clean test goals

Execution time of test methods lasted for 3.859 s and the project consumed 15 MB of memory at runtime.

5. Findings and Analysis

In this chapter gained experience of working with different frameworks and the numerical results from building and compiling the implemented projects will be analysed to find the most promising extension. To perform this analysis, the criteria explained in 5.2 is deployed. The items mentioned in Table 4 is measured for each tool and according to its situation, functional aspects are analysed. Additionally, the compilation and execution time and memory consumption of all tools are listed in a table to give a better understanding of tools' performance.

5.1 Test Station Specifications

This study is conducted on a typical system with the minimum requirements for coding, compiling, and running a program on Windows 8.1. However, a more detailed specification of the hardware is:

- Processor: Intel Core i7-4700 MQ 2.4 GHz
- Installed Memory: 16 GB
- Graphics: GeForce NVIDIA 4 GB
- System Type: 64-bit Windows 8.1 Operating System

5.2 Evaluation Criteria

To have a realistic yet a simple evaluation of tools, this research concentrates on functional and executional aspects of the extensions in a stable environment and under the same conditions. Some of the most critical aspects are initialization, the number of dependencies, compilation/build time, execution time, test time, memory consumption, etc. Eclipse as a very common and powerful development tool provides some of the numerical data required for this purpose and for other parts it is necessary to have a non-numerical analysis. To have practical and more reasonable criteria, a scoring approach is taken to score each tool according to its capabilities upon each factor with accounting a simple scenario shown in Table 4.

Table 4. Framework Scoring Template

Feature	Yes	No
Tool supports Eclipse	+5	0
Tool provides a plugin	+5	0
Tool supports custom constraining	+5	0
Tool depends on an external library(for each dependency)	-1	0
Tool supports full documentation and user guide	+1~+5	0
Ease of use	+1~+5	0

The reason for including the first row in Table 4 is that one of the most popular IDEs for Java, not being unreasonable if say the most popular one, is Eclipse, although there are a variety of IDEs such as IntelliJ IDE, NetBeans, JDeveloper, BlueJ, etc. Therefore, due to gaining this popularity and since Eclipse is non-proprietary development tool, this study perceives supporting it as a critical aspect for a DbC instrumentor. Besides, focusing on one IDE makes the research more focused since most of the tools are not so mature to support a plethora of different IDEs.

Documentation support focuses on a detailed documentation and user guidance with providing code examples wherever is necessary with a good style of writing and formatting the manual neatly. Moreover, ease of use concentrates on how ease the tool is accessible in coding with regard to constraining and how the tool can simply create and constraint with less complexity for the developer.

In addition to mentioned features and their points, coding and executional aspects can be compared accordingly:

- Less compilation/build time can show better performance
- Less memory consumption can show better performance

The final score can claim which tool is better to be deployed.

5.3 Functionality Assessment

In this section, for tools introduced in this research the scoring scenario is applied and the final scores will be compared. The comparison is partially deterministic because both qualitative and quantitative aspects of a software in an experimental analysis have been taken into account. However, the performance results will help to have a meticulous outlook of the frameworks. The results of functionality assessment are shown in Table 5. The assessment is based on tools' official documentation and the author's experience. Some items can be seen as yes or no features and for each feature there is 5 points. It should be noted that external dependencies are considered as negative aspects and for each dependency a -1 point is measured for the corresponding tool. For example, if a tool depends on three external libraries -3 points are set for this item. Additionally, since documentation support and ease of use are qualitative aspects, the pointing should be taken from a range of points, which is 1 to 5 in this criterion.

Table 5. Java DbC Tools Functionality Assessment Scores

Tools	Feature						Total Score
	Eclipse Support	Plugin Support	Custom Constraining	External Dependencies	Documentation and User Guide	Ease of Use	
Bean Validation	5	0	5	-3	5	3	15
C4J	5	0	0	-3	3	4	9
Cofaja	5	0	0	-2	3	2	8
ezContract	0	0	0	-3	3	4	4
iContract	0	0	0	-1	2	4	5
JContractor	0	0	0	-1	3	4	6
Modern Jass	0	0	0	-1	3	4	6
Open JML	5	5	0	-3	4	3	14
OVal	5	0	5	-2	3	2	13

- ❖ For each item a tool gets 5 points if:
 - It supports an IDE
 - It provides a plugin
 - It supports custom constraining
- ❖ For each external dependency a tool gets -1
- ❖ For documentation and user guide a tool can get a score from 1 to 5-the higher the better
- ❖ For ease of use a tool can get a score from 1 to 5-the higher the better

As Table 5 reveals, Bean Validation with 15 points is the most promising tool in IDE support, customisability, documentation and user guidance, ease of use and other functional aspects. At next level, Open JML and Oval with 14 and 13 points respectively have got the second and third place in the scoring table. Although they might have huge differences, one can see them in almost a same level for development and validation. C4J and Cofaja with 9 and 8 point respectively are at next levels. C4J has suppressed Cofaja in the ranking due to its more simplicity of usage. jContractor and Modern Jass with 6 points showed less benefitability from the viewpoint of tool's capabilities. iContract with 5 and ezContract with 4 points got the lowest points in the ranking.

5.4 Performance Analysis

From the perspective of building and execution the tools have generated different outcomes. The data captured from deploying those frameworks that supported Eclipse environment at compile and runtime is shown in Table 6. Results are separated as two categories one for build phase and one for test phase.

Table 6. Java DbC Extensions Compile and Execution Results

Tool Name	Build		Test	
	Compile Time(s)	Memory Consumption (MB)	Compile Time(s)	Memory Consumption (MB)
Bean Validation	0.470	8	2.790	11
C4J	0.344	8	2.677	11
Cofaja	0.337	8	2.589	12
ezContract	-	-	-	-
iContract	-	-	-	-
jContractor	-	-	-	-
Modern Jass	-	-	-	-
Open JML	0.598	7	2.736	9
OVal with AspectJ	0.362	9	3.859	15

The best compiling time is for Cofaja with 0.337 s and the worst is for Open JML with 0.598 s. After Cofaja, C4J and Oval with 0.344 s and 0.362 respectively showed better compiling rather than other tools. Bean Validation with 0.470 s has the second worst compile performance among all extensions. In fact, although Bean Validation has proven the best tool from the viewpoint of functionality and features, this tool does not show a good result of compiling. Indeed, Bean Validation may claim long ages for compiling a project particularly if the project is a big with thousands of lines of code. In a similar compile time, simpler tools compiled the projects quicker than other tools. This means

that simpler tools with less functionality might be quicker in build and compile time. Unfortunately, ezContract, iContract, jContractor and Modern Jass do not support Eclipse, thus, the building and execution time are missing for them. Even measuring the associated times in command line for them will disrupt the equality of development conditions because it definitely will affect build and test time when the environment changes.

From the perspective of test time, Cofoja with 2.589 s owns the fastest running and OVal with 3.859 s has the worst. C4J, Open JML, and Bean Validation with 2.677 s, 2.736 s, and 2.790 s respectively, generated test time at a lower level. Moreover, it should be noted that OVal may take a long time for testing a big project with a plethora of lines of code. As mentioned above, ezContract, iContract, jContractor and Modern Jass have not been comparable to other tools due to lack of Eclipse support. Not only the build and test time can be critical for the frameworks, memory consumption is also important for an evaluation. Thanks to Eclipse that provides memory usage as a supplementary information in its console, memory usage of the extensions, which is listed in Table 6.

For build phase, Open JML allocated the least amount of memory with 7 MB, which is a very good result. Bean Validation, C4J, and Cofoja with 8 MB have equally used memory less efficiently. However, the worst tool in memory usage was OVal 9 MB which is a bit more in comparison to other tools. It could be inferred that OVal may involve a lot of memory during the compile of a real big project. In test phase, Open JML proved to be the best framework with 9 MB. The second best tools are Bean Validation and C4J with equally 11 MB. Cofoja with 12 MB, quite close to Bean Validation and C4J, showed a bit lower performance. However, OVal with 15 MB of memory consumption had the worst memory allocation. It means that using OVal in big projects may be drastically terrible due to high memory consumption both at build and run.

5.5 Answering Research Questions

The main research question of this study was:

RQ: What is the state of the art in Java contract programming?

To get rid of generality included in the question it was necessary to divide it to two minor questions focusing on two different areas. In order to address the new questions an experimental study has been conducted and corresponding results are collected. According to the criteria concerning functionality and performance analysis the sub questions of this investigation are answered.

RQ1: Which tools can create better results from the viewpoint of functionality?

Bean Validation with 15 points, as listed in Table 5, is the most promising framework from the viewpoint of functionality. Bean Validation supports an IDE, provides customisability of constraints, and proposes a very well-formed documentation both in its official website as an html introduction and a PDF version with a detailed information about the API and the tool's principles. Bean Validation reduces coding with regard to constraining and is very powerful in validation of executable and non-executable elements. In Addition, it has updated the API regularly meaning that the inventors are improving the extension.

RQ2: Which tools are better from the viewpoint of performance?

As demonstrated in Table 6, C4J has the best compile and test time with medium memory allocation. On the other hand, Open JML has the best memory management with low time management. Considering both functionality and performance concurrently, Open JML with 14 points, showing moderate results in build and test time and being very efficient in assigning memory can be considered as the most promising tool for validation with integration in Eclipse. Although Bean Validation has got the highest point in the ranking, it did not show good results in build and test time and memory management. In fact, Bean Validation stayed at the middle of the ranking from the perspective of performance. In other words, if the coder is not concerned about hardware resources, means that they have got powerful computers, Bean Validation can work superbly. But since the resources are inevitably critical in programming and it is generally admitted that reserving fewer resources can show better performance, therefore, Open JML is reliable both in having great features and performing well in real situations. However, Open JML and Bean Validation can be chosen interchangeably according to a project's configuration and resources.

6. Discussion and Conclusion

6.1 Discussion

Contract programming as a useful programming technique has proven to be a practical validation method for most of programming languages. This paradigm helps to reach a better situation when the coder is concerned about fewer coding flaws and runtime errors. In fact, DbC detects and reduces unexpected errors by providing a method to check preconditions, postconditions on which the code specification should be satisfied. DbC enforces a software to implement some checks on each element to diminish programming faults and concurrently makes the code more readable. Although DbC involvement in a project can be time consuming and a lot of efforts and impose more cost to a project team, it is worth employing its theory because it helps to save time at test and deploy phase where real users should work with the product and give their feedback with reporting possible errors. This process, i.e. error detection and continuous resolving, is costly and very time consuming. Thus, having a powerful DbC tool can be definitely beneficial for a project. Reasons like these will justify the comparison of previously made tools to differentiate current endeavours leading to have a list of promising tool with respective applicability.

Comparing currently available Java extensions for software validation can be seen from different lenses. First, working with different tools through this study can give a useful sight to developers who are not familiar with DbC and if they know the theory of DbC, they might not have heard about these tools and their features and capabilities. Additionally, a reader can get a concise overview of the tools, their constraining principles, their practical contracting, and their validation methods. Indeed, a quick understanding of what each tool can do and how it adds a contract to a class can be delivered with notifying that it is possible to rapidly get the gist of contract programming and its basic rules. Next, for a coder with a minimum perception of DbC and knowing some tools, this research can help to choose better tools for their needs according to their usefulness and applicability in a certain project frame. A project highly dependent on a specific IDE, e.g. Eclipse, needs a tool for which there is a support of that IDE using one specific extension. Even for more detailed requirements, e.g. having a tool with a plug-in support, a development team can choose a precise tool for a precise purpose rather than only using a tool randomly. In addition, a tool with a complete documentation and better user guidance can fulfil a novice developer's needs effectively with fewer self-tutorial courses. Next, an evaluation of performance, i.e. build/run time and memory allocation, is sufficiently influential on the decision making of working with a particular framework according to a project time, project goals, and a product quality and precision. For instance, a simple project not emphasising on test phase and memory management can deploy a tool working better only in compilation no matter how much memory it allocates for its validation process. But for a project with thousands of lines of code, in which build time and memory allocation is highly influential on the product and its regularly intangible deliverables, choosing a fast enough tool with a professional memory management is critical. Besides, although tools are different in features and performance, this study can claim that each of tools are useful for a specific purpose and all of them have positive effects on programming with paying attention to software validation.

Considering the evaluation and comparison of tools this study can be extended by adding more items to the criteria or deploying a more sophisticated comparison method to get more accurate results. For example, adding other possible items to the criteria and

dividing each item to some sub items then giving a score to each of them can generate more precise points. For instance, one can consider testability as another item in which legibility, and traceability can be taken as sub items and for each of them a similar or even different scoring template can elaborate the study and the findings. But these new criteria may require different test environments and test systems due to having many more aspects. Even by taking into consideration of software quality aspects and examine each framework under one aspect, other researchers can evaluate the tools in a different manner. Although this thesis only concentrated on scrutiny of DbC tools, while future studies can try to compare DbC tools with tools from other validation and verification methods to even compare tools from a higher level of evaluation. Moreover, in a more sophisticated evaluation a study can attempt to examine the extensions by asking a group of real developers and conducting some interview with the participants as the human analysers to gather their experiences and at the same time, collecting related data of those users' interaction with the tools while they are coding based on a defined scenario.

6.2 Limitations

This treatise does not aim to find a silver bullet for specification language support for any kind of programming language but it seeks to catch out a highly beneficial and proper extension from a group of solutions merely generated for Java. However, the reason why Java is selected as the leading language will be discussed later. Another limitation is that this study chooses candidates only from available tools meaning that there might be dissimilar Java libraries or tools with special features but once they do not support DbC or are not alive anymore, they will not be dwelled on as potential candidates. In addition, since the improvement of current tools is a continuous process and most of them are likely to improve in near future, the results of this research might be viable and restricted to the time of examination and it would be reasonable if there might be a great change in the results after a while.

6.3 Conclusion

This research aimed to find a better solution from a variety of solutions for programming by contract in Java language by an experimental evaluation of current DbC instrumentors according to a novel scrutiny and comparison criteria. This examination required to implement a semi real software system as the model coded system on which the tools should have been deployed for an analysis. Working with each framework and test it on Java code indicated that tools showed different results from the viewpoint of features and performance. Some tools have better or more features, some tools provide better performance and some tools are in moderate situation. This study shows that a tool with outstanding features cannot guarantee to have noticeable performance. Bean validation with the highest functionality score can be considered as a really professional and flexible tool with excellent features when the resources are not restricting the validation. C4J as a simple tool is the fastest tool when it comes to build and execution time and Open JML as more complicated tool, not as complicated as Bean Validation is, with a lot of amazing features is the best tool when it comes to memory management.

References

- Arnout, K., & Simon, R. (2001). The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel. *In 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 14-23. *IEEE*
- Bergström, J. (2018). C4J Design By Contract for Java. Retrieved July 8, 2018 from <http://c4j.sourceforge.net/>
- Bolstad, M. (2003). Design by Contract: A Simple Technique for Improving the Quality of Software. *In Proceedings of the 2004 Users Group Conference*, pages 319-323. *IEEE*
- Carrillo-Castellon, M., Garcia-Molina, J., Pimentel, E., & Repiso, I. (1996). Design by Contract in Smalltalk. *Journal of Object-Oriented Programming 1996*, pages 23–28
- Chen, C., Cheng, Y., & Hsieh, C. (2008). Contract specification in Java: Classification, characterization, and a new marker method. *IEICE Transactions on Information and Systems 2008, VOL.E91-D, Pages 2685-2692*
- Cheng, Y., Chen, C., & Hsieh, C. (2007). ezContract: Using Marker Library and Bytecode Instrumentation to Support Design by Contract in Java. *In Proceedings of the 14th Asia-Pacific Software Engineering 2007*, pages 502-509. *IEEE*
- Crocker D. (2004) Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm. *Practical elements of safety: in proceedings of the twelfth Safety-Critical Systems Symposium 2004*, pages 19-41. *Springer*
- Delev, T., & Gjorgjevikj, D. (2017). Static analysis of source code written by novice programmers. *In 2017 Global Engineering Education Conference (EDUCON)*, pages 824-830. *IEEE*
- Ehmer Khan, M. (2010). Different Forms of Software Testing Techniques for Finding Errors. *IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 3*, pages 11-16
- Feldman, Y. A. (2003). Extreme Design by Contract. *In Extreme Programming and Agile Processes in Software Engineering (XP 2003) 4th International Conference*, pages 261–270. *Springer*
- Ferentschik, H., Morling, G., & Smet, G. (2018). Hibernate Validator 6.0.10.Final -JSR 380 Reference Implementation: Reference Guide. Retrieved July 5, 2018 from <http://beanvalidation.org/>
- Firesmith, D. (1999). A Comparison of Defensive Development and Design by Contract. *In 30th International Conference on Technology of Object-Oriented Languages and Systems, Delivering Quality Software - The Way Ahead (TOOLS 1999)*, pages 258-267. *IEEE*
- German, A. (2003). Software Static Code Analysis Lessons Learned. *CrossTalk the journal of defense software engineering 2003, Vol. 16, No.11*, pages 13-17

- Guerreiro, P. (2001). Simple Support for Design by Contract in C++. In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, pages 24–34. IEEE
- Hakonen, H., Hyrynsalmi, S., & Järvi, A. (2011). Reducing the Number of Unit Tests with Design by Contract. In *Proceedings of the 12th International Conference on Computer Systems and Technologies 2011*, pages 161-163. ACM
- Hoare, C. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM 1969, Volume 12 Issue 10*, pages 576-580
- Introduction to OpenJML (2018). Retrieved July 9, 2018 from <http://www.openjml.org/documentation/onlinemanual.shtml>
- Jézéquel, J. M., & Meyer, B. (1997). Design by Contract: The Lessons of Ariane. *IEEE Computer 1997, Vol. 30 Issue 1*, pages 129-130
- Karaorman, M., Hölzle, U., & Bruno, J. (1999). jContractor: A Reflective Java Library to Support Design By Contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection 1999*, pages 175-196. Springer
- Kramer, R. (1998). iContract - The Java™ Design by Contract™ Tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems 1998*, pages 295-307. IEEE
- Kumar, K., & Dahiya, S. (2017). Programming Languages: A Survey. *International Journal on Recent and Innovation Trends in Computing and Communication 2017, Vol. 5 Issue 5*, pages 307-313
- Leavens, G. T., Baker, A. L., & Ruby, C. (1998). JML: A Java Modeling Language. In *Formal Underpinnings of Java Workshop at OOPSLA'98*
- Leavens, G. T., Baker, A. L., & Ruby, C. (2003). Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes 2006, Vol. 31 Issue 3*. ACM
- Leavens, G. T., & Cheon, Y. (2006). Design by Contract with JML. Retrieved April 13, 2018 from www.jmlspecs.org
- Le Traon, Y., Baudry, B., & Jézéquel, J. M. (2006). Design by Contract to improve Software Vigilance. *IEEE Transactions on Software Engineering 2006, Vol. 32 Issue 8*, pages 571–586
- Liskov, B., & Guttag, J. (1986). Abstraction and Specification in Program Development. MIT Press/Mc Graw Hill
- Manna, Z. & Pnueli, A. (1974). Axiomatic Approach to Total Correctness of Programs. *Acta Informatica Vol. 3 Issue 3*, pages 243-263. Springer
- Meyer, B. (1988). Eiffel: A Language and Environment for Software Engineering. *The Journal of Systems and Software 1988 Vol. 8 Issue 3*, pages 199-246
- Meyer, B. (1992). Applying “Design by Contract”. *IEEE Computer 1992*, pages 40-51
- Meyer, B. (1997). Object-Oriented Software Construction Second Edition. Prentice Hall

- Minh Lê, N., (2011). Contracts for Java: A Practical Framework for Contract Programming. *Grenoble INP – Ensimag Technical Report, Google Switzerland GMBH*
- Morling, G. (2017). Bean Validation specification. *Java Community Process Program*. Retrieved July 5, 2018 from <http://beanvalidation.org/>
- Muijs, D. (2004). Doing Quantitative Research in Education with SPSS. *SAGE Publications Ltd*
- Myers, G. J. (2004). The Art of Software Testing, Second Edition. *John Wiley & Sons, Inc.*
- Novikov, A. S., Ivutin, A. N., Troshina, A. G., & Vasiliev, S. N. (2017). The Approach to Finding Errors in Program Code Based on Static Analysis Methodology. In *6th Mediterranean Conference on Embedded Computing (MECO) 2017*, pages 437-440. *IEEE*
- Oliveira e Silva, M., & Francisco, P. G. (2014). Contract-Java: Design by Contract in Java with Safe Error Handling. In *3rd Symposium on Languages, Applications and Technologies 2014*, pages 111-126. *SLATE*
- OVal - the object validation framework for Java™ 5 or later (2018). Retrieved July 10, 2018 from <http://oval.sourceforge.net/userguide.html>
- Plessel, T. (1998). Design By Contract: A Missing Link In The Quest For Quality Software. Retrieved April 8, 2018 from <https://www.eiffel.org/documentation>
- Plösch, R. (1997). Design by Contract for Python. In *Proceedings of the Joint Asia Pacific Software Engineering Conference 1997*, pages 213–219. *IEEE*
- Plösch, R. (2002). Evaluation of Assertion Support for the Java Programming Language. *Journal of Object Technology 2002 Vol. 1 No. 3*, pages 5-17
- Rieken, Y. (2007). Design by Contract for Java-Revised. *Master thesis, Universität Oldenburg*
- Sharan, K. (2017). Beginning Java 9 fundamentals: Arrays, objects, modules, JShell, and regular expressions. *Apress*
- Simon, Mason, R., Crick, T., Davenport, J. H., & Murphy, E. (2018). Language Choice in Introductory Programming Courses at Australasian and UK Universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, Pages 852-857. *ACM*
- Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., & Saake, G. (2012). Applying Design by Contract to Feature-Oriented Programming. In *Fundamental Approaches to Software Engineering 2012, Lecture Notes in Computer Science, Vol 7212*, pages 255-269. *Springer*
- Wampler, D. (2006). Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software 2006*, pages 27–30. *Bonn University*

Yilmaz, K. (2013). Comparison o-f Quantitative and Qualitative Research Traditions: epistemological, theoretical, and methodological differences. *European Journal of Education* 2013, Vol. 48 No. 2, pages 311-325

Zhou, Y., Pelliccione, P., Haraldsson, J., & Islam, M. (2017). Improving Robustness of AUTOSAR Software Components with Design by Contract: A Study Within Volvo AB. *Software Engineering for Resilient Systems. SERENE 2017. Lecture Notes in Computer Science, Vol 10479, pages 151-168. Springer*

Appendix A. Classes Before Deploying Constraints

```
public class Car {

    private String plateNumber;
    private String model;
    private String manufacturer;
    private int fee;
    private boolean availability;

    public Car(String plateNo, String model, String manufacturer,
int fee) {
        this.plateNumber = plateNo;
        this.model = model;
        this.manufacturer = manufacturer;
        this.fee = fee;
        this.availability = true;
    }

    public String getPlateNumber() {
        return plateNumber;
    }

    public void setPlateNumber(String plateNumber) {
        this.plateNumber = plateNumber;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    public int getFee() {
        return fee;
    }

    public void setFee(int fee) {
        this.fee = fee;
    }

    public boolean isAvailable() {
        return availability;
    }

    public void setAvailability(boolean availability) {
        this.availability = availability;
    }
}
```

```

public class Client {

    private String ID;
    private String name;
    private String address;
    private String phone;

    public Client(String id, String name, String address, String
phone)
    {
        this.ID = id;
        this.name = name;
        this.address = address;
        this.phone = phone;
    }

    public String getID() {
        return ID;
    }

    public void setID(String id) {
        this.ID = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

```

public class Deal {

    private String ID;
    private Car rentedCar;
    private Client client;
    private int rentDays;
    private int totalFee;
    private Date startDate;
    private Date endDate;
    private DealStatus status;

    public Deal(String id, int days, int totalFee, Car rentedCar,
Client client, Date start, Date end) {
        this.ID = id;
    }
}

```

```

        this.rentDays = days;
        this.rentedCar = rentedCar;
        this.client = client;
        this.totalFee = totalFee;
        this.startDate = start;
        this.endDate = end;
        this.status = DealStatus.Open;
    }

    public String getID() {
        return ID;
    }

    public void setID(String id) {
        this.ID = id;
    }

    public Car getRentedCar() {
        return rentedCar;
    }

    public void setRentedCar(Car rentedCar) {
        this.rentedCar = rentedCar;
    }

    public Client getClient() {
        return client;
    }

    public void setClient(Client client) {
        this.client = client;
    }

    public int getRentDays() {
        return rentDays;
    }

    public void setRentDays(int rentDays) {
        this.rentDays = rentDays;
    }

    public int getTotalFee() {
        return totalFee;
    }

    public void setTotalFee(int totalFee) {
        this.totalFee = totalFee;
    }

    public Date getStartDate() {
        return startDate;
    }

    public void setStartDate(Date startDate) {
        this.startDate = startDate;
    }

    public Date getEndDate() {
        return endDate;
    }

    public void setEndDate(Date endDate) {
        this.endDate = endDate;
    }

```

```

    public DealStatus getStatus() {
        return status;
    }

    public void setStatus(DealStatus status) {
        this.status = status;
    }

    public enum DealStatus{Open, Closed}
}

public class InsuredDeal extends Deal{

    private int maximumInsuranceCover;

    public InsuredDeal(String id, int days, int totalFee, Car
rentedCar, Client client, Date start, Date end, int maxInsCover) {
        super(id, days, totalFee, rentedCar, client, start,
end);
        this.maximumInsuranceCover = maxInsCover;
    }

    public int getMaximumInsuranceCover() {
        return maximumInsuranceCover;
    }

    public void setMaximumInsuranceCover(int
maximumInsuranceCover) {
        this.maximumInsuranceCover = maximumInsuranceCover;
    }
}

public class CarRental {

    public static ArrayList<Client> clients = new
ArrayList<Client>();
    public static ArrayList<Car> cars = new ArrayList<Car>();
    public static ArrayList<Deal> deals = new ArrayList<Deal>();

    public CarRental() {
    }

    public static void main(String[] args) throws
NoSuchMethodException {

        Client customer1 = new Client("CL-2018-1001",
"George", "1851-Willington st.", "46322311");
        Client customer2 = new Client("CL-2018-1002",
"Michael", "2310-Hamilton st.", "46322412");
        Client customer3 = new Client("CL-2018-1003", "James",
"1438-Vertongen st.", "46324835");

        clients.add(customer1);
        clients.add(customer2);
        clients.add(customer3);

        Car car1 = new Car("DD-CE-123", "XC60", "Volvo", 50);
        Car car2 = new Car("DD-BC-456", "RAV4", "Toyota", 65);
        Car car3 = new Car("DD-BC-456", "508", "Peugeot", 40);
        Car car4 = new Car("DD-AC-234", "S8", "Audi", 60);
        Car car5 = new Car("DD-EF-567", "Omega", "Opel", 45);
    }
}

```

```

        cars.add(car1);
        cars.add(car2);
        cars.add(car3);
        cars.add(car4);
        cars.add(car5);
    }

    public void addNewClient(String id, String name, String
address, String phone) {
        Client client = new Client(id, name, address, phone);
        clients.add(client);
    }

    public void addNewCar(String plateNo, String model, String
manufacturer, int fee) {
        Car car = new Car(plateNo, model, manufacturer, fee);
        cars.add(car);
    }

    public Car searchAvailableCar(int fee) {
        Car car = null;
        for(Car c: cars) {
            if (c.getFee() == fee && c.isAvailable()) {
                car = c;
                break;
            }
        }
        return car;
    }

    public Deal rentCar(Client client, Car car, int rentDays, Date
start, Date end) {
        Deal deal = new Deal("DL-" + client.getID() + "-" +
car.getPlateNumber(), rentDays, car.getFee() * rentDays, car, client,
start, end);

        deal.getRentedCar().setAvailability(false);
        deals.add(deal);
        return deal;
    }

    public void terminateDeal(Deal deal) {
        deal.setStatus(Deal.DealStatus.Closed);
        deal.getRentedCar().setAvailability(true);
    }
}

```