Design By Contract Implementation For ExtendJ

Martin Lindström
Department of Computer Science, Lund University, Sweden
dat15mli@student.lu.se

Jakob Hök
Department of Computer Science, Lund University, Sweden
dat15jh1@student.lu.se

Abstract

The paper demonstrates how *Design By Contract* (DBC) was integrated into the aspect-oriented Java compiler *ExtendJ* by using annotations. The scope of the implementation was limited to only include preconditions, i.e., only method-input validation. Adding DBC into Java, is a handy way of validating execution and also to improve readability. Therefore, DBC is a desirable feature to implement. *ExtendJ* makes use of aspect-oriented programming to compile Java code. Extending the compiler was therefore convenient and allowed the implementation to conform to the *open-closed principle*. The preconditions' byte code was inserted before the method's or constructor's byte code. Hence, the precondition validation is run before the method's code. An exception is thrown if any precondition is not met. In the end, the extension supports preconditions *NotNull*, *MinValue*, *MaxValue*, *Not* and arbitrary preconditions with the annotation *Requires*.

1 Introduction

Contracts, as most often referred to, is an agreement between two parties; the contract is commonly formed between an employer and an employee. Each party has to live up to the agreed requirements of the contract. This is very useful because there is a mutual benefit of knowing what to expect from one another.

Design By Contract (DBC) is this concept applied to software. The term was invented by Bertrand Meyer in 1992 through his own designed programming language Eiffel [5]. A contract within DBC is instead formed between a caller and callee of a method or a function. The usefulness of these contracts shows itself when a contract is broken. Either the caller has provided invalid input to the method, or the callee has returned invalid output from the method. The contracts enable a decoupling of these types of runtime checks while also making it easier to know where in the program such an error occurred. Meyer separates a contract into three pieces, namely being:

- *Preconditions* : The requirements for the *caller* to fulfill
- *Postconditions* : The requirements for the *callee* to fulfill
- *Invariants* : A condition for an object's state that needs to be true at all times.

The routine when executing any method is to first check all preconditions from the parameters. Then, the method itself is run, with the knowledge of every parameter being valid. Lastly, before returning to the caller, the postconditions is validated. This ensures that both the caller and the callee receive proper data from one another. If no exception is thrown in the process, both the caller and callee meet the conditions of the contract.

If instead any condition of the contract is not met, it is an easy task to tell whether the error originates from the caller or callee.

The extent of this paper will be limited to implementing preconditions into a Java compiler named *ExtendJ* [6]. Specifically, the preconditions targeted are *NotNull*, *MinValue*, *MaxValue*, *Not* and *Requires*. The first four represent predefined and reusable preconditions; the last one, *Requires*, is instead implemented to enable arbitrary preconditions for the programmer to define.

To declare preconditions in the Java code, annotations will be used. Any precondition annotation will serve as additional data to the compiler. During compilation, this data can be used to generate byte code for the annotation's contained precondition. Any precondition byte code will be assured to execute before its respective method.

ExtendJ is an aspect-oriented compiler, which has full support for the Java language. Therefore, the project can focus only on implementing the precondition functionality while using *ExtendJ's* existent implementation.

2 Motivating example

As a Java programmer, one has most certainly stumbled upon scenarios where methods become bloated of extensive input parameter checking. Since the Java language is rather verbose in its nature, methods that are intended to be a couple of lines of code can easily become twice the size. Consider the simple piece of code in Listing 1; the method `sumDice` takes two die rolls as input parameters and returns the sum as output where $a, b \in \{1, ..., 6\}$.

Course paper, EDAN70, Lund University, Sweden
January 28, 2019.

Listing 1. Method that returns the sum of two dice rolls.

```
public int sumDice(int a, int b) {
  if (a > 6 || a < 1 || b > 6 || b < 1) {
    throw new Exception();
  }
  return a + b;
}
```

The if-block within `sumDice` is not part of the method's actual functionality and can make it harder to interpret, not only the implementation, but also the purpose of the method. One way to remove the undesired if-block is to apply DBC. This will decouple the parameter validation from the method body to instead express it as a contract carrying preconditions. The modified example shown in Listing 2 displays this by using Java's native annotations, which is attached to the parameters of the function.

Listing 2. Same functionality as the method in Listing 1 but without internal if-check.

```
public int sumDice(
    @MinValue(1) @MaxValue(6) int a,
    @MinValue(1) @MaxValue(6) int b) {
  return a + b;
}
```

3 Dealing with inheritance

As Meyer states regarding DBC [5], an issue emerges when contracts are going to be applied to subtypes which intend to override the original method's declaration. Imagine B is a subtype of A, and S is a module that calls A's method $m()$. When S calls $A.m()$ there is no way for S to tell if A actually is an instance of B, because of the fundamental rules of object-oriented programming. Therefore, if B modifies the precondition contracts of $m()$, S will still expect its input to $m()$ to be according to the rules of A, not B, which may lead to unintended behavior.

Meyer proposed a solution for this, which is to use the logical OR-operator across all redeclarations of a precondition. In practice, this means that a precondition for a method can not become more strict when that method is overridden. The rules to obey for the caller can only be loosened, not strengthened. This ensures that a caller does not have to know about which subtype its callee is and may use the precondition in the expected type. *Cofoja* and JML are two separate examples of DBC-featured Java compilers that have this functionality implemented [2] [3].

The compiler of this project handles inheritance in a simpler manner by letting preconditions for a method be stated only once and not be loosened by subtypes. The following three rules are introduced to handle subtypes:

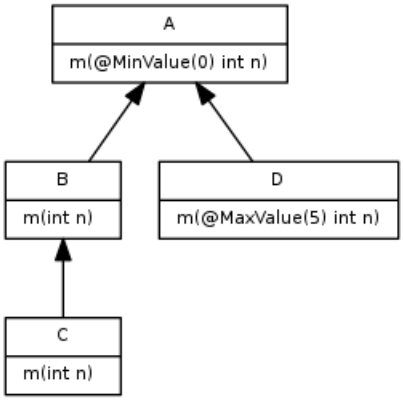

Figure 1. A UML diagram where all classes will pass the compilation except D.

1. A precondition can not be redeclared or modified from the original.
2. A precondition must be declared in the class where the method is first introduced.
3. Any subclass inherits the preconditions for a method, no matter whether the inherited method is abstract or not.

To illustrate these rules with an example, Figure 1 shows a situation of four classes and their relation to each other in terms of inheritance. By looking at each class individually, one can conclude that:

- A passes the set rules, because it introduces $m()$ together with its precondition `@MinValue`.
- B and C also pass the rules. Additionally, they both inherit the precondition defined in A, according to rule 3.
- D does break rule 1 and 2 because it tries to modify the preconditions of $m()$, while not being the class that introduces $m()$.

To fix the conflict that occurs in D, the most natural approach would be to lift up the `@MaxValue` onto A so that the signature becomes `m(@MinValue(0) @MaxValue(5) int n)`.

3.1 Shared method signature across ancestors

Since the compiler forces preconditions to be defined only in the highest ancestor, a new problem emerges. That is when a method in a subclass is inherited from multiple ancestors. A simple example, shown in Figure 2, would be to have two separate interfaces A and B, with a common abstract instance method $m(int\ n)$. Let C be a concrete class that implements the two interfaces, and thus implements m.

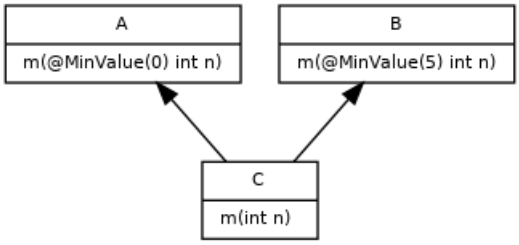

Figure 2. This situation causes a compile time error, because the precondition of $C.m(int\ n)$ is ambiguous.

Both interfaces are allowed to define their respective preconditions since they both introduce the method m. But if A and B have defined their preconditions for m differently, there will be a collision in C, because a caller of e.g., $A.m(int\ n)$ would not know anything about $B.m(int\ n)$. The standard Java compiler '*javac*' would not throw any compiler errors, so the extended compiler of this project has to define a solution for this scenario.

The chosen solution to this problem is to throw a compile error if any of the highest ancestors states any precondition. In the example, it means that the only way $C.m()$ will pass compilation, is if A and B removes their preconditions for m.

4 Arbitrary preconditions

The preconditions earlier presentented has a clear meaning and works well for primitive types; however, in a Java program there are often methods taking in more complex types as parameters. Therefore an annotation `@Requires` has been added. The annotation is attached to an entire method takes in a string array as its own parameter, like Listing 3 is showing.

Listing 3. *divisible* acts as a precondition for *divide* to ensure dividing whole numbers has no remainder.

```
@Requires({"divisible"})
int divide(int numerator, @Not(0) int
    denominator) {
  return numerator / denominator;
}

boolean divisible(int numerator, int denominator)
    {
  return numerator % denominator == 0;
}
```

The string array parameter to *Requires* is pointing to the names of other methods, like `divisible` in the example. The requirement of such a method is that the parameters to it need to match the ones of the *Requires*-annotated method and also return a boolean type. The compiled program is then forced to run the boolean methods referenced to through the annotation, before running the method itself. Therefore, there is a possibility of expressing arbitrary preconditions.

4.1 Overriding the boolean methods

In order to make these arbitrary preconditions as useful as possible, there is an exception regarding the strict rule of not allowing precondition overriding. All boolean methods `Requires` points to, are allowed to be overridden in subclasses; however, the `Requires`-signature itself must not be changed, i.e., the string parameters in the annotation will remain the same by following the rules explained in the last section.

The reason for this decision is to be able to incorporate attributes that the superclass does not have. The added functionality a subclass usually comes with will in some cases require a modification to checking internal state. Therefore, subclasses can choose their own way of checking their instance attributes by overriding the boolean precondition methods.

The task for the programmer is consequently to verify that these boolean methods do *not* possess any stateful behaviour; also, if overridden, the boolean methods should *not* be made stricter towards the annotated method's caller.

5 Implementation of the preconditions

ExtendJ is in its entirety built with a metacompiler called *JastAdd* which makes it the main tool for achieving the DBC extension. *Jastadd* works by generating Java code for the compiler [6]. The generated code can then be executed to compile any Java file. In addition to *JastAdd* code, several lines of native Java code was written which do nothing more than defining the annotations used for the preconditions. Other than that, *JastAdd* was used exclusively to write to extension.

5.1 Generating the byte code

The fundamental rule across all precondition checks is that they need to be executed before the method itself. The idea is therefore to add a hidden code snippet before the method's actual code, exactly like Listing 1 shows. Like regular Java compilation, *ExtendJ* works by generating byte code. The DBC-extension generates additional byte code for every method- and constructor declaration, which handles the contract-based code. As an example, the parameter precondition `@NotNull` will produce the Java code snippet displayed in Listing 4 when attached to an `Object obj`. The produced Java code is then what generates the extra byte code when completing the compilation.

Listing 4. Code snippet generated at compile time when @NotNull is attached to Object obj.

```
if (obj == null) {
  throw new PreconditionViolationException(
     "Argument 'obj' must not be null");
}
```

Designing this functionality into *ExtendJ* was achieved by making use of the `refined` keyword in *JastAdd*. Refining a method simply means to modify its implementation while still having the option to call the original method from the modified version. One can see it as overriding a method in Java. When implementing the DBC compiler, this was taken advantage of by refining the byte-code generation for constructors and methods. This enabled the compiler's code to conform to the *open-closed principle*, i.e., "open for extension, but closed for modification" [4]. There was no need to modify existing *ExtendJ* code, i.e., not be forced to introduce dependencies to the DBC extension. *ExtendJ* was rather used as an API to call existing functions and refine methods in the compiler to modify them to match DBC-featured behaviour. This coincides well with what Öqvist explains regarding the extensibility of *ExtendJ* [6].

6 Evaluation

Performing a thorough evaluation of the compiler can be achieved by having solid test coverage. During the working process, unit tests were written and verified for each new addition of functionality. The unit tests covers both compilations expected to fail, as well as runtime tests verifying correct behaviour of preconditions.

In addition to unit tests, example code taken from *Cofoja's* repository were used. The example code were rewritten to use this project's precondition. This was done with the goal in mind order to make a comparison between an existing DBC-featured compiler.

The time scope of this project was 400 man-hours split between two individuals. If more time were given, the project could be extended to a larger degree by implementing the remaining parts of DBC. As this point, the number of code lines is measured to be 415 for *JastAdd*, and 39 for Java. The measurements were retrieved by using the terminal-based program `cloc`. Since `cloc` has no support for *JastAdd* code, the files had to be temporarily renamed to the have the .java-extension. The program then evaluates the files as being Java code, which makes no difference since the syntax is similar.

The main time consumer for the project's progress was to get used to *ExtendJ*, i.e., using the existing functionality correctly. For someone who is more familiar to *ExtendJ*, the implementation would go considerably quicker. Aside from that, this project had a solid workflow and reasonable difficulty level.

7 Related work

Not too surprisingly, a number of third party DBC implementations for Java already exists. *Cofoja*, as an example, was developed by Nhat Minh Lê during his internship att Google in 2010 [2]. Syntactically this is similar to the implementation of this project, but *Cofoja* has more thorough support for arbitrary preconditions by letting a boolean expression be expressed as a string in their own predefined annotation `@Requires`. For instance, in a stack data structure, the `peek()` can be provided with `@Requires("size() > 0")` as a precondition. Implementing this feature into the compiler of this project was considered, but dropped because the scope require time beyond what was planned.

An example of another approach regarding design is *Java Modeling Language* (JML) [3]. Instead of using Java annotations, JML parses the contract-related code from regular code comments, meaning that a different Java compiler would process the code without even parsing the preconditions as part of the code. Therefore the code has the potential advantage of not directly being dependent on the JML compiler.

Aside from another choice of syntax, the DBC implementation is similar to the compiler of this project because both build a type checked abstract syntax tree (AST). The AST is then attached to the rest of the code's tree. Finally, the new AST is responsible for generating the byte code, just like this project's compiler.

Both JML and *Cofoja* uses their own compiler for their intentions. A project named *jContractor* has achieved DBC for *Java* without writing a compiler. Instead, they load the class files with the DBC extension, and then execute the program just like any other program. [1].

Instead of using annotations, *jContractor* uses method names as key, e.g., for a method named `m`, the precondition would be named `m_Precondition`. A program called `main` can then be run in the command line by simply typing `java jContractor main arg1 arg2 arg3`. *jContractor* has also support for running the program without the DBC features for lower execution time.

However, *jContractor* was written around the time when its report were published, in 2002. It does not seem to be a recent version of the program. Most likely, *JContractor* was discontinued and not compatible with modern *Java* versions.

8 Conclusion and Future Work

This project includes a partial implementation of the DBC-concept by extending the Java compiler *ExtendJ*. The usage of annotations as preconditions not only increases the readability of the code, but also removes any if-checks required to verify valid input. As a result, the code becomes less bloated. The programmer, both as a writer and a reader, can focus on the actual functionality of the method or constructor.

Annotations enables additional possibilities. Since the Java language already supports syntax for annotations, declaring any preconditions in the code will not break the compilation when using another compiler, e.g., *javac*. Naturally, any other compiler would completely ignore any preconditions by not generating the byte code.

As the evaluation clearly displays, extending an aspect-oriented compiler with preconditions can be done by using the existent code base to a high degree; the newly introduced lines of code is therefore of a comprehensible amount. The current implementation of `@Requires` is limited to referencing a method name. A future improvement can be to implement usage of arbitrary boolean expressions, stated directly in the annotation itself, e.g., `Requires("x > 0")`, just like *Cofoja* has achieved [2]. If one is ambitious, future work is to develop the other features of DBC, i.e., postconditions and invariants.

As a final addition to this DBC extension, it would be beneficial to be able to compile code without precondition checking. This applies if the programmer is using preconditions only for debugging, and not catching the precondition errors in the production code. An addition like this would be useful when doing a stable release of a program, in order to achieve a slight performance increase.

Acknowledgments

This computer-science project was part of a school assignment at *Lunds Tekniska Högskola* (LTH). As we are students who have recently been introduced to the design and implementation of compilers, this project was a very educative and entertaining way of getting deeper knowledge within the area.

We want to thank Görel Hedin for being an instructive supervisor by helping us drive the project in the right direction. The stepwise, weekly feedback we received was valuable to us to make the compiler work as intended.

The first two weeks of the project was demanding because we had to get accustomed to the *ExtendJ* compiler. We thank Jesper Öqvist, PhD student at LTH, who helped us to get a basic understanding of *ExtendJ*. The strategies he proposed for solving our issues spared us hours of potential frustration.

References

[1] Parker Abercrombie and Murat Karaorman. 2002. jContractor: Bytecode instrumentation techniques for implementing design by contract in Java. *Electronic Notes in Theoretical Computer Science* 70, 4 (2002), 55–79.

[2] Nhat Minh Lê. 2011. *Contracts for java: A practical framework for contract programming.* Technical Report. Google Switzerland GmbH.

[3] Gary T Leavens and Yoonsik Cheon. 2006. Design by Contract with JML. (2006).

[4] Robert C Martin. 1996. The open-closed principle. *More C++ gems* 19, 96 (1996), 9.

[5] Bertrand Meyer. 1992. Applying'design by contract'. *Computer* 25, 10 (1992), 40–51.

[6] Jesper Öqvist. 2018. ExtendJ: extensible Java compiler. In *2nd International Conference on Art, Science, and Engineering of Programming, Programming 2018*. Association for Computing Machinery (ACM), 234–235.