Danish University Colleges

Contract-Based Software Development: Class Design by Contract

Kongshøj, Simon

Publication date: 2016

Document Version
Other version

Link to publication

Citation for published version (APA):
Summary

This note introduces the foundations of contract-based development in an object-oriented setting. We introduce the six fundamental principles using motivating examples in JML (Java Modeling Language), and introduce the concepts of type invariants and representation invariants.

Introduction

We are about to begin the study of the programming technique of Design by Contract, a systematic and semi-formal approach to discovering and dealing with defects in software during development. It is systematic in the sense that it proceeds according to well-defined principles, and it is semi-formal in the sense that it has enough formality to be automatically verifiable by an appropriate programming language runtime, but not in the sense of mandating mathematical proof technique during the development process.

All nontrivial programs develop a variety of defects during development, and a large portion of any software development effort is dedicated to discovering and repairing or removing defects. Many defects — particularly those that involve a faulty composition of multiple components — arise because programmers make a variety of assumptions about their code; assumptions that turn out to be false.

One way to mitigate this problem is for programmers to deliberately make as few assumptions as possible. This leads to the technique of defensive programming, in which all program components are liberally sprinkled with tests for various problematic conditions. For example, the expression $42 / x$ tacitly assumes that the variable $x$ will never be zero, so a defensive programmer might wrap it in an appropriate `if`, taking
some appropriate action if the assumption turns out to be false. When calling out to methods from libraries, defensive programmers attempt to handle all possible error states that may arise from an unexpected result — in a sense, defensive programming deliberately avoids assuming that any method works “as advertised”.

Unfortunately, defensive programming has two major problems. The large amount of error handling code obscures the primary functionality of the code, and since all these tests are executed at runtime (where the vast majority of them will turn out to be false, for a correct and working program), they also introduce a performance penalty. When this performance penalty is unacceptable, defensive programmers will then re-introduce some assumptions (perhaps documenting them in comments) and subsequently omit some of the error handling code.

*Design by Contract* (DbC) provides an alternative approach to detecting program defects during development. Rather than avoiding assumptions as in defensive programming, DbC makes them explicit parts of the interface of methods (or functions) and enforces them. Returning to the example of $42 / x$, rather than placing a test inline, a DbC programmer will state as part of the interface to the surrounding method that $x \neq 0$, and a DbC-enabled language runtime will then take an appropriate action (typically crashing the program with a helpful contract violation report) if the assumption fails. This error testing can be disabled using a compile-time flag (which is usually done once development is finished), so it can be made arbitrarily complex without affecting performance in the final program deployment.

It is important to note that DbC is not an alternative to exception-handling mechanisms. DbC is specifically concerned with discovering *bugs*, whereas exception handling can manage any exceptional condition, including ones that aren’t indicative of a bug. For example, if a user tries to access a nonexistent file, then that isn’t necessarily indicative of a bug; it could just be the user mistyping a filename. Such situations are best handled using exceptions. As a general rule, contracts are a *development time* mechanism, and no user should ever see a contract violation, even if the user makes a mistake while using the program.
Core Concepts

A contract is a collection of assertions, consisting of three different kinds of constructs: **preconditions, postconditions** and **invariants**. They correspond closely to the related constructs from the Q language.

A **precondition** is a condition that a method assumes to be true. It is the responsibility of any code that calls the method to make sure that it is true — for example, if a method specifies as a precondition that its argument `x` must be nonzero, then it is illegal to call it with zero in that argument. A precondition that is violated means that the program has a bug, and most DbC tools will crash the program and notify that there has been a precondition violation.

A **postcondition**, in contrast, is a condition that a method guarantees to be true. It is the responsibility of that method to make sure that once the method returns, it is true. For example, a method could specify as a postcondition that a particular field in an object is non-null, and that makes it illegal for the method to return when that field is null. A postcondition violation is treated just like a precondition violation, and also means that the program has a bug.

Finally, an **invariant** is a condition that all methods guarantee to be true — even the constructor. Invariants can be viewed as a kind of shared postcondition used by all methods.

Another view of the three core concepts is that preconditions specify the **circumstances** under which it is valid to call a method, postconditions specify the **effect** of calling the method, and invariants specify **unchanging properties** of a class. Most DbC tools check conditions at runtime, although some can also do some static analysis.

Contracts serve two purposes: They are part of the **specification** of a program component, and as such serve as useful documentation to programmers. Unlike comments, they’re documentation that is **guaranteed to be correct**, because code that doesn’t respect them will result in a well-defined error. They therefore also comprise a suite of **correctness checks**. In most modern realizations, contracts can be applied both to interfaces and to implementations. When used in interfaces, they serve to make sure that any implementation of the interface correctly implements it, when used in implementations they serve to make sure that an implementation doesn’t suffer from internal inconsistencies.
Tools for Contract-Based Development

Historically, the first programming environment that supported design by contract was Bertrand Meyer’s programming language Eiffel, an early object-oriented language that was specifically designed for DbC\(^1\). Eiffel integrated DbC-related keywords directly into the language, and the runtime supported configurable strictness for contract violation checks. A relatively large amount of DbC literature uses Eiffel for code examples.

In this course, we shall use two DbC tools: JML and CodeContract. One is for Java and one for the .NET platform, and they represent two very different approaches to tool support for DbC. All code examples in these lecture notes and in the slides will be in Java using JML; for some of your exercises you will need to translate them to CodeContract.

JML

The Java Modeling Language (JML) is an academic research project that seeks to bring Design by Contract to Java. There are two distinct branches of the project: The original JML project, which is no longer actively maintained, and only works with Java 1.4 or earlier. OpenJML supports Java 1.7, but work is currently in progress to support Java 1.8.

JML is based on a preprocessor, in which contract code is written as Java comments. This means that any JML program is also a valid Java program, and JML programs can be compiled with a normal Java bytecode compiler (yielding a program with no contract checks). The JML preprocessor reads specially formatted comments, and emits a Java program instrumented with contract checks that is then passed to the Java compiler. The preprocessor-based approach gives considerable flexibility, but at the cost of having a slightly awkward surface syntax. The next page has an example of a class specified and implemented using JML.

---

\(^1\)In fact, Bertrand Meyer is also the inventor of the Design by Contract methodology.
public class Person{
    private /*@ spec_public non_null @*/
        String name;
    private /*@ spec_public @*/
        int weight;

    /*@ public invariant !name.equals("")
        @
        && weight >= 0;
        @ also
        @ ensures \result != null; @*/
    public String toString(){
        return "Person(" +name+"," +weight+")";
    }

    //@ ensures \result == weight;
    public /*@ pure @*/ int getWeight(){
        return weight;
    }

   /*@ requires kgs >= 0;
        @ requires weight + kgs >= 0;
        @ ensures weight == \old(weight + kgs);
        @*/
    public void addKgs(int kgs){
        weight= weight+kgs;
    }

   /*@ requires n != null && !n.equals("");
        @ ensures n.equals(name)
        @ && weight == 0; @*/
    public Person(String n){
        name= n;
        weight= 0;
    }
}
}
This example demonstrates the format used for JML. The keyword requires is used to declare preconditions, the keyword ensures is used to declare postconditions, and invariant declares invariants. The pure keyword tells the preprocessor that a method has no side effects, and the also keyword is used to override inherited contracts — in the example, it is used to override contracts given by toString which is inherited from Object. The constructs \texttt{old} and \texttt{result} are very useful for contract checking; \texttt{old} can be used to refer back to the states of variables before a method is run, and \texttt{result} refers to the return value.

It is generally considered good design practice to separate interface from implementation, and consequently JML allows contracts to be used in both interfaces and classes. Here is an example of a JML-based interface:

```java
public interface PersonIF {
    //@ public invariant getWeight() >= 0; @/

    //@ also
    //@ ensures \texttt{result} != null;
    public /*@ pure non_null @*/ String toString();

    //@ ensures \texttt{result} >= 0;
    public /*@ pure @*/ int getWeight();

    //@ requires kgs >= 0;
    @ ensures getWeight() == (\texttt{old}(getWeight()) + kgs);
    //@
    public void addKgs(int kgs);
}
```

The JML preprocessor will emit Java code that ensures that any code that implements or uses the aforementioned interface will detect contract violations. With this approach, JML code placed in interfaces refer to contracts for any correct implementation of the interface, whereas JML code placed in classes refers to contracts for specific implementations.
CodeContract

CodeContract is a .NET tool to integrate DbC with C#, and is an official Microsoft project bundled with Visual Studio. Unlike JML, CodeContract doesn't use a preprocessor, but is simply a *library*, using normal method calls to perform checks.

class Person{
    private String name;
    private int weight;
    public int Weight { get; private set; }
    [ContractInvariantMethod]
    private void ObjectInvariant(){
        Contract.Invariant(!name.Equals("") && weight >= 0);
    }
    [Pure]
    public override String ToString(){
        Contract.Ensures(Contract.Result <string>() != null);
        return "Person(" + name + "," + weight + ")";
    }
    [Pure]
    public int GetWeight(){
        Contract.Ensures(Contract.Result<int>() == weight);
        return weight;
    }
    public void AddKgs(int kgs){
        Contract.Requires(kgs >= 0);
        Contract.Requires(weight+kgs >= 0);
        Contract.Ensures(weight == Contract.OldValue(weight) + kgs);
        weight = weight + kgs;
    }
    public Person(String n){
        Contract.Requires(n != null && !n.Equals(""));
        Contract.Ensures(n.Equals(name) && weight == 0);
        name = n;
        weight = 0;
    }
}


Just like the first JML example, this example doesn’t separate interface and implementation. Unfortunately, the library approach used by CodeContract makes it impossible to have CodeContract declarations in interface — they’re method calls, and we can’t have method calls in interfaces! The solution is to use a rather ugly hack using an internal abstract class with a dummy implementation:

```csharp
[ContractClass(typeof(IPersonContract))]
interface IPerson{
    [Pure]
    String ToString();
    int GetWeight();
    void AddKgs(int kgs);
}

[ContractClassFor(typeof(IPerson))]
internal abstract class IPersonContract : IPerson{
    [ContractInvariantMethod]
    private void ObjectInvariant(){
        Contract.Invariant(GetWeight() >= 0);
    }

    public override String ToString(){
        Contract.Ensures(Contract.Result<string>() != null);
        return default(string);
    }

    public int GetWeight(){
        Contract.Ensures(Contract.Result<int>() >= 0);
        return default(int);
    }

    public void AddKgs(int kgs){
        Contract.Requires(kgs >= 0);
        Contract.Requires(GetWeight()+kgs >= 0);
        Contract.Ensures(GetWeight()==Contract.OldValue(GetWeight())+kgs);
    }
}
```
CodeContract also has some problems with inherited contracts, as we shall see in a later session. This series of lecture notes will use JML for most examples; some exercises will involve translating them to CodeContract.

The Six Fundamental Principles

In an object-oriented context, Design by Contract usually proceeds according to six basic design principles. These tend to yield programs that are particularly well-suited for contract-based development, and good DbC design usually follows them closely. We shall consider each in the context of a simple stack class.

A classic object-oriented Stack object (presented here in pseudo-Java) typically has the following basic operations:

```java
Stack<T> {
    Stack();  // constructor
    int count();  // number of items on the stack
    boolean isEmpty();  // self-explanatory
    void push(Object) // push an item onto the stack
    Object pop()  // pop an item from the stack, returning it
}
```

1. Separate commands and queries

In DbC, it is often very useful to make an explicit distinction between methods that change an object, and methods that merely return information about an object. A query is a method that has a return value, but does not have any side effects\(^2\). In contrast, a command is a method that has a side effect, but doesn’t have a return value. This distinction is important because all queries can then freely be used to write preconditions, postconditions and invariants for our classes.

The stack example has two straightforward examples of queries (count() and isEmpty()), and two straightforward examples of commands (push() and the constructor). But what about pop() — is that a query, or is it a command? In fact it is neither, and that makes it problematic from a DbC perspective.

Imagine that you’re writing some application which uses or extends the stack class. You need to write a contract which specifies some property about what is on top of the stack. Is it safe to write this in terms of pop()?

\(^2\)Recall that a side effect is simply a state change upon an object.
In fact it is not, and it should be obvious why. Because pop() not only returns the top value of the stack but also transparently removes the top value, using it in a pre- or postcondition is a terrible mistake. If it is used in a precondition of some method, then the top element of the stack will be removed before the method is even executed, potentially leading to utterly bizarre errors. Likewise, using it as a postcondition of a method will remove the top element of the stack after the method has run, potentially turning a push operation into a null operation — and is certainly not the desired behaviour. Furthermore, recall that the dynamic checks in DbC tools can be disabled if necessary. This implies that a program with contract checking should be semantically equivalent to the same program without contract checking, which in turn means that no contract may have any observable side effects.

A DbC stack object would therefore get rid of pop(), separating it into two methods: A query named top() and a command named remove().

2. Separate basic queries from derived queries

Just like methods can be decomposed into queries and commands, queries can be decomposed further, depending on their data access mechanism.

A basic query is a query which simply accesses some variable in an object directly. They are precisely equivalent to what some object-oriented programming environments call "getters". In the stack example, count() is a basic query. The newly introduced top() is also a basic query.

In contrast, a derived query is a query which performs some computation in order to return a result. They are often written in terms of basic queries. In the stack example, isEmpty() is a derived query: It can be written in terms of checking whether count() == 0.

Another perspective is to consider basic queries to be "fundamental", in the sense that they can only be written in terms of direct access, whereas derived queries can use other queries in an object. As another example, if a Person class has a representation of a person’s height and weight, then get_height() and get_weight() would be basic queries (because you can’t compute them from any other information in the object), but get_BMI() would be a derived query (because you can write it in terms of get_height() and get_weight()).
3. For each derived query, give postconditions in terms of basic queries

Because derived queries are derived from basic queries, their results should be understandable entirely in terms of basic queries. In DbC terms, this means that they should have a postcondition which expresses their result entirely in terms of basic queries and (perhaps) constants. For example, an appropriate JML postcondition for `isEmpty()` could be:

```jml
/*@ ensures \result == (count() == 0); */
```

This states that it is an error for `isEmpty()` to return anything that is not equivalent to the Boolean expression `count() == 0`, which is exactly how we intuitively would understand a method like `isEmpty()`. In the postcondition, we don’t care how `isEmpty()` arrives at that result, merely the value of the result. Perhaps `isEmpty()` simply calls `count()`, or perhaps it checks whether its internal stack representation is `NULL`. What makes it a derived expression is that it can be written in terms of `count()`.

In some cases, this might mean that the postcondition is identical to the actual implementation of the query. In the person example, that would almost certainly be the case for the `get_BMI()` method. Although this will raise most programmers’ hackles because it smells of code duplication, it is still considered good practice in DbC, for two reasons. First, the practice of DbC often involves viewing code from two different perspectives — the contract perspective and the implementation perspective. When considering the contract perspective, the relevant question is “what is the code supposed to do?”, and in the implementation perspective, the question is “how does the code work?”. Restating the code from both perspective can sometimes reveal unfounded assumptions or inappropriate implementation-level thinking. Second, if the implementation is later rewritten to work in a completely different way, the contract will still express our expectations for the code (and raise errors if the new implementation is buggy).

4. For each command, give postconditions in terms of every basic query

Since commands change the state of an object, they should have postconditions that specify the expected result state of the object. This means that every command should specify the expected results of every basic query. In the stack example, this means that `push()` should specify the expected results of `top()` and `count()` as postconditions. In JML:

```jml
/*@ ensures count()==\old(count()+1); */
```
public void push(Object e);

This states — precisely as we intuitively understand a stack — that pushing an object onto the stack increases its size by one, and that the new top element is the object we gave as a parameter to push().

But this postcondition isn’t complete! It says nothing about what has happened to the rest of the stack. A pathological implementation could replace the rest of the stack with rubbish, and this contract would still accept it, as long as it yields a stack of the appropriate size and with the last-pushed object intact. Later, we shall see how we can express contracts that work on entire data structures. For now, it’ll suffice to note that our current stack interface would be very awkward to use for that, because top() can only be used to peek at the top stack value.

It is possible to write an itemAt() query to work around this deficiency — this would allow peeking into the stack to an arbitrary depth. Some programmers balk at this, feeling that it violates the LIFO principle of stacks. However, because it only allows inspection to arbitrary depth, without modification at arbitrary depth, it does no such thing. In fact, it is trivially easy for a client of Stack to construct it (inefficiently), using only count(), top(), remove() and push(). Having such a method would make it much easier to write a more bulletproof contract that specifies that older elements of the stack mustn’t change as a result of a push() or a remove(), so DbC data structure implementations typically have such a method. Note that having itemAt() would make top() a derived query.

The principle says to give postconditions for every basic query. In the person example, there might be a command addKgsToWeight(). Should such a command have postconditions that refer to the height, name or age of a person? Yes, it should: An addKgsToWeight() implementation which modifies the name, height or age of a person is wrong, and the contract should make this explicit.

5. For every query and command, give suitable preconditions

Whereas the earlier principles are rather “mechanical” in the sense that they can be applied without much reflection once the concepts of commands and basic vs. derived queries have been understood, it isn’t possible to give any formal definition of which

---

3To maintain realism, removeKgsFromWeight() is left out, and can only be implemented as a more complicated compound procedure.
preconditions should be considered "suitable". Like much programming, this aspect of DbC is an art form rather than a scientific discipline.

A good guideline is to carefully think about which assumptions you, as a programmer, are making when you want to use a given method. For example, if we have implemented the `itemAt()` query, then we might make the assumption that we never try to reach beyond the top or bottom of the stack. In JML:

```java
/*@ requires 0<=i && i<count(); */
public /*@ pure @*/ Object itemAt(int i);
```

Note that it is very important here to differentiate between error situations that are bugs and error situations that are user errors. If we have a stack implementation with the above contract, and users may input values that are used as the i parameter, then we must check for that and raise an appropriate exception before the call to `itemAt()`. On the other hand, if some other part of the code tries to read beyond the end of the stack for any other reason, then that’s a sure sign that something is wrong.

Some methods may not have any preconditions. For example, there is no useful precondition for `isEmpty()` — it should work on any stack object.

The relationship between pre- and postconditions can be said to form a contract, which is where the name of DbC is derived from. They express the contract that if all the preconditions are met, then the method will guarantee that it does precisely what its postconditions state. If the preconditions are not met, then the method gives no guarantees at all — it could return any arbitrary value, crash the program, format the hard drive or launch the missiles. Because this situation means that there is certainly a bug in the program under development (and buggy programs shouldn’t be permitted to run, because they might do terrible things), DbC tools will then crash the program with an appropriate error message.

6. Specify invariant properties in the class invariant

In the vast majority of cases, there are assumptions that we always make about all objects within a given class — indeed, if those assumptions don’t hold, then an object can be considered in an invalid state, and the method that left it in that state clearly has a bug. Such assumptions should be expressed as a class invariant. If they are assumptions that any client of the class must be able to make, then they’re referred to as a type invariant, whereas if they’re assumptions for the internal implementation of the class, then they’re referred to as a representation invariant.
For example, a type invariant of stacks could be that they always have a non-negative size — a quite common assumption, since a stack of negative size is nonsensical, and any method that creates a stack of negative size clearly has a bug in it. In JML, this would be expressed in the interface of the stack:

```java
/*@ public invariant count()>=0; @*/
```

Representation invariants are, by definition, implementation-specific. For example, in a stack implementation based on linked lists, a JML representation invariant could state (in the implementation class):

```java
/*@ private invariant ((top==null)==(count==0)); @*/
```

Assuming that `top` is an internal variable that holds the node on top of the stack and `count` is an internal variable that holds the current stack size, then this expresses that `top` is `null` if and only if the stack is empty.

Class invariants are pre- and postconditions of every method in the class, and thus hold across the entire lifecycle of the object. In particular, they should be considered to be the postcondition of the constructor. This affects the design of the constructor: In the person example, if an invariant states that the person's name must not be `null`, then the constructor must either take a name as a parameter or set a dummy name.

It is important to note that methods may break the invariant for some period of time before they return — as long as the invariant is maintained when they return. In other words, a `clearStack` method might briefly break the representation invariant above by first setting `count` to 0 and then setting `top` to `null` (or vice versa), but this is all right as long as it does both before returning.

**Literature**

Mitchell & McKim: *Design by Contract, by Example*, Addison-Wesley 2002