

A Methodology for Writing Class Contracts

Nele Smeets
Eric Steegmans

Report CW 399, January 2005



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Methodology for Writing Class Contracts

Nele Smeets
Eric Steegmans

Report CW 399, January 2005

Department of Computer Science, K.U.Leuven

Abstract

One of the principles of Design by Contract is that contracts for software components must be written in a declarative way, using a formal, mathematically founded notation. When we apply the Design by Contract methodology in a naive and straightforward way, we risk ending up with unwanted duplication. In this paper, we describe a methodology for writing class contracts that avoids specification duplication and that gives rise to uniform class specifications with a clear and fixed structure.

A Methodology for Writing Class Contracts

Nele Smeets*, Eric Steegmans

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan
200A, 3001 Leuven, Belgium

{Nele.Smeets, Eric.Steegmans}@cs.kuleuven.ac.be

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium)

Abstract. One of the principles of Design by Contract is that contracts for software components must be written in a declarative way, using a formal, mathematically founded notation. When we apply the Design by Contract methodology in a naive and straightforward way, we risk ending up with unwanted duplication. In this paper, we describe a methodology for writing class contracts that avoids specification duplication and that gives rise to uniform class specifications with a clear and fixed structure.

1 Introduction

The concept of Design by Contract (DBC) was introduced by Bertrand Meyer [14] as a way to increase the reliability of object-oriented programs. The basic idea behind DBC is that there is a contract between the supplier and the client of a class, specifying rights and obligations for both parties. The basic ingredients of this contract are preconditions, postconditions and invariants. *Preconditions* of a method specify conditions that must be fulfilled each time a client invokes the method. They are obligations for the client and benefits for the supplier. *Postconditions* of a method define the conditions that must be satisfied upon returning from each invocation. They must only be satisfied if the method has been called under conditions satisfying its preconditions. Postconditions are obligations for the supplier and benefits for the client. *Invariants* are assertions that must be satisfied at all stable times, i.e. after the creation of an object, and before and after each invocation of a public method. Among the major benefits of DBC, we mention improved reliability, clearer and consistent documentation, extended opportunities for reuse and better support for debugging.

One of the principles of DBC is that contracts for software components must be written in a declarative way, using a formal, mathematically founded notation. When we apply the DBC methodology in a naive and straightforward way, we risk ending up with unwanted duplication. In particular, specifications of constraints imposed on the state of objects tend to get spread all over the definition of a class. This duplication has a negative impact on the adaptability of software and it will sooner or later result in an inconsistent class definition. This problem

statement is discussed in more detail in Sect. 2, and illustrated with a typical example.

In this paper, we describe a methodology for writing class contracts. Our methodology avoids specification duplication and gives rise to uniform class specifications with a clear and fixed structure. We describe our methodology using the Java programming language, but it is also applicable to other object-oriented languages.

The remainder of this article is structured in the following way: in Sect. 2, the problem statement is illustrated with an example. Sects. 3, 4 and 5 present the methodology in detail. Sects. 6 and 7 describe how invariants and set methods are defined in our methodology. Sect. 8 discusses the main advantages and disadvantages of the methodology. Finally, Sect. 9 describes related work and Sect. 10 contains a short conclusion and summarises future work.

2 Problem Statement

When we apply the DBC methodology in a naive and straightforward way, we risk ending up with unwanted duplication. For example, consider the Java code in Fig. 1, describing a class of bounded lists. In a bounded list, the number of list elements is bounded by the capacity of the list.

Despite its simplicity, the class of bounded lists already gives an example of unwanted specification duplication. Indeed, part of the class invariant is repeated in the precondition of the `setCapacity` method to ensure that, after application of the method, the bounded list still satisfies the class invariant. A similar duplication is needed in the specification of other methods and constructors that manipulate or initialise the capacity or the elements of a bounded list.

Duplicating constraints imposed on the characteristics of objects has a negative impact on software quality. Correctness is at stake, especially when a class contains several characteristics and when the class invariant imposes restrictions on the relation between these characteristics. In that case, there is a considerable risk of being incomplete, when writing the preconditions of a method: all invariants concerning the characteristic(s) involved in the method must be duplicated. In Fig. 1, two preconditions are needed for the `setCapacity` method. The first precondition says that the given capacity must be positive. But also the second precondition is needed, imposing a condition on the relation between the given capacity and the *current* number of elements in the list: it says that the current number of list elements should not exceed the given capacity.

Duplication of constraints also hampers adaptability. Because of changing requirements, certain constraints will need to be changed over time. When the specification of a constraint is spread over invariants and preconditions, chances are high that the constraint will not be adjusted at all places, resulting in an inconsistent class definition.

```

/**
 * A class of bounded lists.
 * @invar The list of elements is effective.
 *       | getElements() != null
 * @invar The capacity of a list is positive.
 *       | getCapacity() > 0
 * @invar The number of elements in a list is smaller than
 *       or equal to the capacity of the list.
 *       | getElements().size() <= getCapacity()
 */
public class BoundedList {
    /** The elements in this bounded list. */
    private List elements;
    /**
     * Return the list of elements.
     * @basic
     */
    public List getElements() {
        return new ArrayList(elements);
    }
    /** The capacity of this list. */
    private int capacity;
    /**
     * Return the capacity of this list.
     * @basic
     */
    public int getCapacity() {
        return capacity;
    }
    /**
     * Set the capacity of this list to the given value.
     * @param capacity The new capacity for this list.
     * @pre The given value is positive.
     *     | capacity > 0
     * @pre The number of elements of this list is smaller
     *     than or equal to the given value.
     *     | getElements().size() <= capacity
     * @post The capacity of this list is set to the given
     *     value.
     *     | getCapacity() == capacity
     */
    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
    ...
}

```

Fig. 1. An example of unwanted specification duplication.

When we take inheritance into account, things get even more complicated. When the class invariant in Fig. 1 is strengthened in a subclass, the precondition of the `setCapacity` method must be strengthened accordingly. Since the Liskov substitution principle [13] expresses that preconditions can only be weakened, we are forced to use abstract preconditions [14]. Abstract preconditions will play an important role in our methodology.

Summarizing, a naive and straightforward application of the principles underlying DBC violates a basic principle of good software engineering: each fact must be worked out in one, and only one, place. Parnas [16] was one of the first to explicate this principle. In fact, this problem not only applies to the specification of a class, but also to its implementation. Indeed, the actual code for checking constraints imposed on the state of objects is much too often worked out at several points.

3 Basic Queries

In our methodology, a first important step in writing class contracts is choosing a set of *basic queries*. This way of working is also advocated in [15] and [19]. As in [14] and [15], we separate commands from queries. Basic queries are a minimal set of queries chosen in such a way that the entire state of an object can be inspected using basic queries only. The basic queries represent the characteristics of the objects of a class. In Fig. 1, there are two basic queries, indicated by the `@basic-tag`.

The basic queries of a class can be used as the basis of the contracts that specify the features of the class: the class invariants, the result of derived (i.e. non-basic) queries, and the effect of commands and constructors can all be specified thoroughly in terms of the chosen basic queries. We also avoid the risk of writing circular assertions.

To describe the conditions imposed on the characteristics of the objects of a class, we introduce two new concepts: partial conditions (Sect. 4) and complete conditions (Sect. 5). Both are represented by Boolean queries. The concepts will be presented for a class `X` containing two basic queries `getv1()` and `getv2()` with return types `V1` and `V2`. These basic queries are declared in `X`, or inherited from supertypes. The schema used for a class with `n` basic queries is a straightforward generalisation of the schema for two basic queries.

4 Partial Conditions

To describe the conditions imposed on the characteristics of the objects of a class, we introduce two new concepts: partial conditions (Sect. 4) and complete conditions (Sect. 5). Both are represented by Boolean queries.

4.1 Short Description

A *partial condition* on a set of characteristics is a constraint involving all these characteristics. In Fig. 1, the partial condition on the capacity of a bounded list expresses that the capacity should be positive. The partial condition on the capacity and the list elements expresses that the number of elements should not exceed the capacity of the list.

The developer of a class is responsible for providing the partial conditions of the class. For each non-empty subset of characteristics, a partial condition must be introduced.

4.2 General Schema

In our methodology, the following Boolean queries are introduced to encapsulate the partial conditions on the characteristics of class X .¹

```
/**
 * @return ...
 */
public boolean partialConditionv1(V1 v1) { ... }
/**
 * @return ...
 */
public boolean partialConditionv2(V2 v2) { ... }
/**
 * @pre    | partialConditionv1(v1)
 * @pre    | partialConditionv2(v2)
 * @return ...
 */
public boolean partialConditionv1v2(V1 v1, V2 v2) { ... }
```

In general, evaluating `partialConditionv1v2Partial(v1, v2)` is only meaningful when `partialConditionv1(v1)` and `partialConditionv2(v2)` evaluate to true. This explains the two preconditions of the `partialConditionv1v2` method. An example of such a situation is given in Sect. 4.3. Notice that, by introducing Boolean queries to encapsulate the partial conditions, we do not have to explicitly duplicate the partial conditions on v_1 and v_2 in the precondition of the `partialConditionv1v2` method.

A partial condition can be deterministic or non-deterministic.

¹ To save space, only a formal specification is given. The informal specification and the implementation of the Boolean queries are omitted. The same is done in following sections.

4.3 Example

The partial conditions for the Java class described in Fig. 1 are given below.

```
/**
 * @return | elements != null
 */
public boolean partialConditionElements(List elements){...}
/**
 * @return | capacity > 0
 */
public boolean partialConditionCapacity(int capacity) {...}
/**
 * @pre    | partialConditionElements(elements)
 * @pre    | partialConditionCapacity(capacity)
 * @return | elements.size() <= capacity
 */
public boolean partialConditionElementsCapacity
                (List elements, int capacity) {...}
```

Evaluating `partialConditionElementsCapacity` is only meaningful when the given list is effective. This is guaranteed by the first precondition.

5 Complete Conditions

5.1 Short Description

A *complete condition* on a set of characteristics is the conjunction of all partial conditions concerning at least one of those characteristics. Thus, a complete condition specifies all conditions that must be satisfied by a set of characteristics. In Fig. 1, the complete condition on the capacity expresses that the capacity must be positive and that the current number of list elements must be smaller than or equal to the capacity.

This definition allows complete conditions to be automatically generated from the partial conditions. So, the developer of a class only has to provide the partial conditions.

5.2 General Schema

In our methodology, the following Boolean queries are introduced to encapsulate the complete conditions on the characteristics of class *X*. Since all complete conditions have similar semantics, the complete conditions on proper subsets of characteristics are defined in terms of the complete condition on all characteristics.

The complete condition *on all characteristics* is given by the following Boolean query:

```
/**
 * @return | if ( not { partialConditionv1(v1) &&
 *                |      partialConditionv2(v2) &&
 *                |      partialConditionv1v2(v1, v2) } )
 *                |      then
 *                |      result == false
 */
public boolean completeConditionv1v2(V1 v1, V2 v2) { ... }
```

In the specification of the complete condition, we use Java's conditional and. In this way, we guarantee that a partial condition on a set of characteristics is only evaluated when all partial conditions on proper subsets of these characteristics are true (as prescribed by the preconditions of these partial conditions).

The query is made non-deterministic to support overriding in subclasses (see below).

The complete conditions *on proper subsets of characteristics* are defined in terms of the complete condition on all characteristics `completeConditionv1v2`.

```
/**
 * @return | completeConditionv1v2(v1, getv2())
 */
public boolean completeConditionv1(V1 v1) { ... }
/**
 * @return | completeConditionv1v2(getv1(), v2)
 */
public boolean completeConditionv2(V2 v2) { ... }
```

When a new characteristic is introduced in a subclass of `X`, represented by the basic query `getv3`, the three complete conditions shown above are redefined in terms of the complete condition `completeConditionv1v2v3`. It can be shown that these redefinitions satisfy the Liskov substitution principle [13].

5.3 Example

The complete conditions for the Java class in Fig. 1 are given below.

```
/**
 * @return | completeConditionElementsCapacity(
 *                |      elements, getCapacity())
 */
public boolean completeConditionElements(List elements) {
    ...
}
```

```

}
/**
 * @return | completeConditionElementsCapacity(
 *         |         getElements(), capacity)
 */
public boolean completeConditionCapacity(int capacity) {
    ...
}
/**
 * @return | if ( not {
 *         |     partialConditionElements(elements) &&
 *         |     partialConditionCapacity(capacity) &&
 *         |     partialConditionElementsCapacity(
 *         |         elements, capacity)
 *         | } )
 *         | then
 *         |     result == false
 */
public boolean completeConditionElementsCapacity(
    List elements, int capacity) { ... }

```

6 Invariant

Using the Boolean queries proposed by our methodology, the invariant of class X can be described as follows.

```
@invar completeConditionv1v2(getv1(), getv2())
```

The invariant of class X is fixed: it is given by the complete condition on all characteristics, with the basic queries as actual arguments. This implicates that it is no longer necessary to write invariants explicitly. They can be omitted or generated by a tool from the partial conditions.

7 Set Methods

Using the Boolean queries representing the complete conditions, the set methods of a class X have a uniform structure.

```

/**
 * @pre | completeConditionv1(v1)
 */
public void setv1(V1 v1) { ... }
/**

```

```

    * @pre | completeConditionv2(v2)
    */
public void setv2(V2 v2) { ... }
/**
 * @pre | completeConditionv1v2(v1, v2)
 */
public void setv1v2(V1 v1, V2 v2) { ... }

```

The precondition of a set method is given by the corresponding complete condition. In this way, it is no longer necessary to duplicate constraints in the preconditions of a set method. Further, no redefinition of the set methods is needed in a subclass, because the precondition of a set method is implicitly strengthened. So, set methods need to be defined only once, namely in the class where the corresponding characteristic is introduced. Finally, this methodology recognizes the need to initialize multiple variables at once, for example, in constructors.

Above, a contractual specification of the set methods is given, but the specification can just as easily be worked out in a defensive way using exceptions, or in a total way by incorporating the complete condition in the postcondition. Obviously, complete conditions can be used in a similar way to write the specification of constructors and more complex commands that initialise or manipulate the characteristics of a class.

8 Advantages and Disadvantages

In this section, we describe the main advantages and disadvantages of the methodology proposed in this paper. We start by examining the influence of our methodology on the quality factors described in [14].

Our methodology has a positive impact on *correctness*, since it avoids duplication and helps in writing complete specifications.

Our methodology has a good influence on *extendibility*. First, when a condition on a certain characteristic or set of characteristics changes, only local changes are required. The reason is that this knowledge is not spread around over the whole class, but is concentrated in one single Boolean query, namely the corresponding partial condition. In general, these partial conditions are small and readable and therefore easy to adapt. Analogously, when certain conditions must be strengthened in a subclass, only local changes are required. The complete conditions are non-deterministic, thereby anticipating the strengthening of conditions in subclasses, or the introduction of new variables. These redefinitions satisfy the Liskov substitution principle. Second, the problem of duplication (as illustrated in Sect. 2) is solved. In this way, various consistency problems are avoided.

Using our methodology gives rise to well-documented classes with a transparent structure. Contracts that clearly explain what methods in library classes do and

what constraints there are on using these methods have a positive influence on *reusability* and simplify communication between developers.

When writing a class containing n characteristics, a developer must provide $2^n - 1$ partial conditions. In addition, $2^n - 1$ complete conditions should be provided or automatically generated by a tool. This exponential number of methods considerably increases the size of a class, thereby requiring more memory, so *efficiency* is negatively influenced.

Using our methodology, we abstract away from the code level, as prescribed by MDA (Model Driven Architecture) [6]. From the partial conditions, provided by the developer of a class, a proper tool can generate the complete conditions, the basic queries, the set methods and the class invariants. Since the basic structure of the class is generated, the developer can pay more attention to the other parts of the class, which has a positive influence on productivity and on the general quality of the code.

9 Related Work

The ideas underlying DBC go back to the work of pioneers in computer science such as Dijkstra [3] and Hoare [7]. They already suggested documenting routines in terms of preconditions and postconditions. DBC, as we know it today, was developed by Bertrand Meyer as a part of Eiffel [14]. Eiffel is an object-oriented programming language that has built-in support for contracts. Another object-oriented language incorporating support for DBC is Sather [18]. Most object-oriented languages, including Java, C++ and C#, lack support for DBC though. Nevertheless, there is a growing interest in DBC and several tools have been developed that provide support for DBC in Java: Biscotti [2], Contract Java [5], Handshake [4], iContract [10], Jass [8], Jcontract [9], jContractor [11]. The Java Modeling Language (JML) [12] is a behavioural interface specification language that can be used to specify the behaviour of Java modules and that is supported by a wide range of tools. The Object Constraint Language (OCL) [20] is a formal language that can be used to specify invariants and operations for UML (Unified Modeling Language) models in a formal way.

One of the principles of DBC is that contracts for software components must be written in a declarative way, using a formal, mathematically founded notation. All the systems described above use Boolean assertions to write class contracts. JML follows a model-based specification approach, while Eiffel, OCL and the tools listed above write specifications in terms of variables and queries directly applicable to the documented software component.

The methodology proposed in this paper prescribes that class contracts must be written in terms of basic queries. This way of working is advocated and described in more detail in [15] and [19]. It is also supported by JML, C# and Enterprise JavaBeans (EJB). In JML [12], model fields can be used to describe the conceptual model of a class. In C# [1], the characteristics of a class can be made

available using properties. In Enterprise JavaBeans [17], two different kinds of components are distinguished: entity beans and session beans. The characteristics of a class are separated in entity beans; these beans represent data. Session beans, on the other hand, model application logic.

10 Conclusion

In this paper, we described a methodology for writing class contracts. Our methodology avoids specification duplication and gives rise to uniform class specifications with a clear and fixed structure. Using our methodology, the basic structure of a class can be generated automatically by a tool, allowing the developer to pay more attention to the other parts of the class, which has a positive influence on productivity and on the quality of the code.

Future work comprises the development of a tool that generates the basic structure of a class when the partial conditions are provided. The details of the generated specification and implementation will vary depending on whether the characteristics of the class are mutable or immutable, whether their type is a reference type or a primitive type, and whether we are programming in a contractual, defensive or total way. Further, a better naming convention needs to be found for the partial and complete conditions.

References

1. Judith Bishop, Nigel Horspool. *C# Concisely*. Pearson Education Limited, 2004, 0 321 15418 5.
2. Cynthia Della Torre Cicalese, Shmuel Rotenstreich. Behavioral Specification of Distributed Software Component Interfaces. *IEEE Computer*, July 1999, Vol. 32, No 7, p46-53.
3. Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976, ISBN 013215871X.
4. Andrew Duncan, Urs Hölzle. Adding Contracts to Java with Handshake. <http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-32.pdf>.
5. Robert Bruce Findler, Matthias Felleisen. Contract Soundness for Object-Oriented Languages. <http://www.ccs.neu.edu/scheme/pubs/oopsla01-ff.pdf>.
6. David S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley Publishing, 2003, ISBN 0-471-31920-1.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, vol. 12, no. 10, pages 576-583, October 1969.
8. The Jass Page. Home Page. <http://semantik.informatik.uni-oldenburg.de/~jass/>.
9. Using Design by Contract[™] to Automate Java[™] Software and Component Testing. <http://www.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jcontract>.

10. Reto Kramer. iContract – The Java Design by Contract Tool. <http://www.reliable-systems.com/tools/iContract/documentation/icontract-tools98usa.pdf>.
11. Murat Karaorman, Urs Hölzle, John Bruno. jContractor: A Reflective Java Library to Support Design By Contract. <http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-31.pdf>.
12. Gary T. Leavens, Albert L. Baker, Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06x, June 1998, revised July, November 1998, January, April, June, July, August, December 1999, February, May, July, December 2000, February, April, May, August 2001, June, August, October, December 2002, April, May, September, November 2003. <ftp://ftp.cs.iastate.edu/pub/leavens/JML/prelimdesign.pdf>.
13. Barbara Liskov, Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841, November 1994.
14. Bertrand Meyer. *Object-Oriented Software Construction*, Second Edition. Prentice-Hall Inc, 1997, ISBN 0-13-629155-4.
15. Richard Mitchell, Jim McKim. *Design by Contract, by Example*. Addison-Wesley, 2002, ISBN 0-201-63460-0.
16. D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053 - 1058.
17. Ed Roman, Scott Ambler, Tyler Jewell. *Mastering Enterprise JavaBeans*, Second Edition. John Wiley & Sons, Inc., 2002, 0-471-41711-4.
18. Sather home page. <http://www.icsi.berkeley.edu/~sather/>.
19. Eric Steegmans, Jan Dockx. *Objectgericht programmeren met Java*. Acco, 2002, ISBN 90-334-4535-2.
20. Jos B. Warmer, Anneke G. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison-Wesley, 1999, ISBN 0-201-37940-6.