The Contract Pattern

Michel de Champlain[1]
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve Blvd. West, Montréal, Qc, Canada H3G 1M8
michel@ece.concordia.ca

14 October 1998

Abstract This paper describes the Contract pattern, a programming pattern (idiom) that lets you apply assertions to guarantee pre-conditions and post-conditions of methods and invariants on the state of objects. The *Design by Contract* methodology—introduced by Meyer is available in the Eiffel context, but unfortunately not for all the other object-oriented languages interested to profit from Meyer's contribution. This paper shows how to implement this idiom in Java. By using this idiom, developers can improve the reliability of their classes by precising what constraints must be satisfied by the client to guarantee correct functioning.

Intent Provide an implementation of the *Design by contract* methodology [Meyer88] for developing reliable classes and create robust objects in Java. This programming pattern lets you apply assertions to guarantee the pre-conditions and post-conditions of methods and invariants on the state of objects.

Also Known As Design by contract

Motivation Sometimes it's necessary to restrict a class interface because you don't want to provide all the possible input combinations for all collaborators. One way to achieve this goal is to make minimal and consistent methods that impose pre-conditions on input parameters. Moreover, you might want to ensure the class behavior by imposing post-conditions before methods return. You might also need to make sure that the class is always in a stable state. Such an approach can result in reliable classes to create robust objects.

1 Revised version of the paper presented at PLoP' 97.

Consider for example a bounded counter that has both a lower and upper bound. A bounded counter object is constructed by setting these bounds. The class interface provides two methods to increment the counter: *inc()* that increments the counter by one and *add(int n)* that adds the counter by *n*.

How do we ensure that the bounded counter class is reliable, i.e., incrementing or adding the counter is legal only when its value is less that its upper bound, and that a class instance is correct and robust?

One way to deal with that problem is to rely on an exception handling mechanism[2]. Such a mechanism provides an elegant way to handle errors by separating error-handling code from normal code, by throwing an exception. For example, create an *Exception* object that includes a message providing information about this particular exception and use it with a throw statement:

```
if ( count <= upper )
    throw new OutOfBoundException();
```

We can improve this situation: by using assertions to establish a contract between a class and its clients. The usage of assertions[3] eliminates (hides) the throw statement, but may require providing information on the exception via an optional parameter:

```
assert ( count <= upper , new OutOfBoundException() );
```

or

```
assert ( count <= upper ); // will throw a more generic message
```

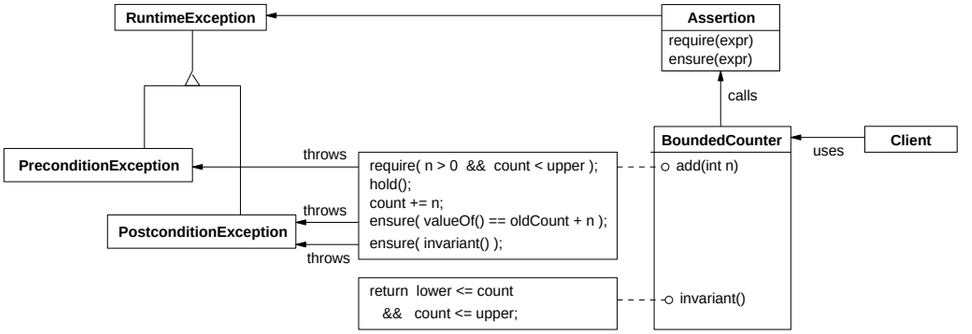
This contract[4] needs also to take into account the concept of a class invariant or property—*lower <= count <= upper*, in the case of the bounded counter class—that must be satisfied by all instances of a class in every stable state.

A solution is to provide a precise definition of what constraints (input parameters) must be satisfied by the client, and the knowledge that this behavior (functionality of each method) will be guaranteed if it agrees to those constraints. In order to support the above solution at run-time, exceptions can be thrown whenever a contract is violated, that is when either a pre-condition, post-condition or invariant is not satisfied. In this regard, the *Contract* pattern is a complement to exception handling.

This bounded counter class illustrates how assertions may be applied to characterize the possible states of an object and to guard its runtime consistency, i.e. *lower <= count <= upper*. For example, a bounded counter object is constructed by setting these bounds. The method *add()* has:

- a pre-condition (*require*) stating that adding *n* to the counter is legal only when *n is greater than zero and its value is less that its upper bound* and,
- a post-condition (*ensure*) promising the correct functionality of the method, and before returning a verification (using *ensure*) that the class invariant is satisfied.

2 Assuming that the language has one, such as Ada, C++, Java, etc.
3 Similar to macros like in C and C++.
4 A contract can be simply characterized by an assertion ensuring than its expression is true in order to resume execution.



This OMT-based class diagram illustrates how the *Contract* pattern offers a solution by providing the *PreconditionException*, *PostconditionException*, and *Assertion* classes that support the invocation of *require*, *ensure*, and *invariant* methods. The implementation of the bounded counter is robust, since it guards clients against possible misuse. The main advantage of using assertions, apart from providing checks to legal usage, is that they explicitly state the requirements imposed on the client and on the supplier. The main disadvantage is the execution overhead since assertions cannot be conditionally compiled in Java—there is no preprocessor like in C and C++.

Applicability

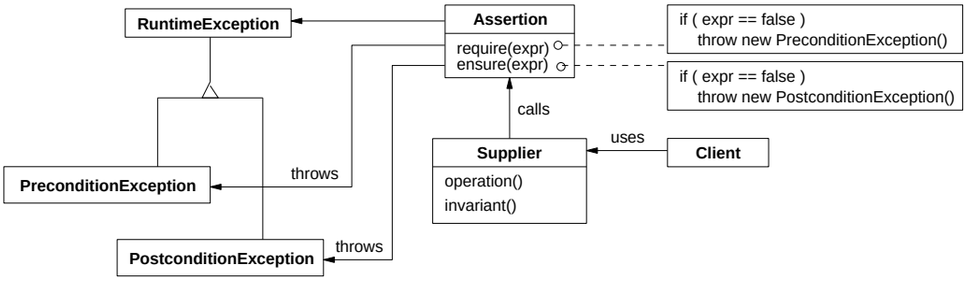
Use the Contract pattern to verify and validate the behavior of an object. The designer develops an interface where assertions are the basic elements of a more formal specification for an object. Such specification of its behavior may be given by defining a *pre-condition* and *post-condition* for each method and an invariant for each class:

- A pre-condition exception denotes a problem in the calling method: the call did not observe the required conditions, i.e. the client's side of the contract.
- A post-condition exception denotes a problem in the called method: the method is not functional, i.e. fulfill the supplier's side of the contract.
- A class invariant defines the invariant properties of the state of each instance of the class. A class invariant ensures class instance consistency.

Class invariants, pre-, and postconditions, all three work together to achieve the purpose of design by contract.

Structure

Participants

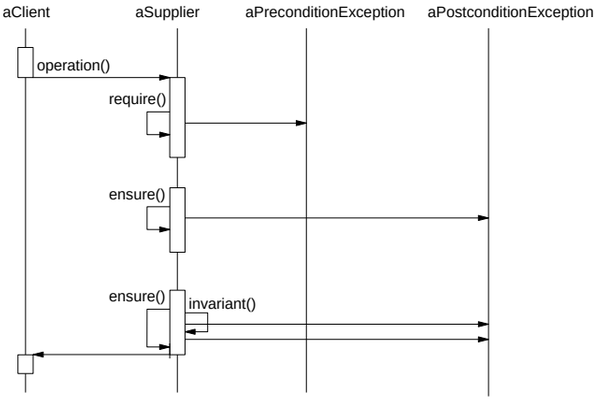


This pattern uses:

- **Supplier** (BoundedCounter)
  - creates a reliable class by using *require()* and *ensure()* operations and by defining the class *invariant()*.
- **RuntimeException**
  - defines a root of the exception hierarchy.
- **PreconditionException**
  - implements and returns a message containing the class and method names associated with the occurring exception.
- **PostconditionException**
  - implements and returns a message containing the class and method names associated with the occurring exception.
- **Assertion**
  - implements the *require(expr)* and *ensure(expr)* operations where each of them can throw the corresponding *Precondition* and *Postcondition* exceptions if the boolean expression (*expr*) is false.

Collaborations

The following interaction diagram illustrates the collaborations between a client and a supplier:



Consequence

The Contract pattern offers the following benefits:

- *Contracts may be used to precise the correct specification on entry and object state on exit*. Contractual obligations avoid to make every method responsible to every kind of possible input. They ensure that every component is characterized by a precise definition of what constraints must be satisfied by the client to guarantee correct functioning.
- *Contracts may be used to document the method interface of a class*. For example, the Java language supports special *doc comments* that can be extracted and automatically turned into HTML documentation.
- *Assertions and exception handling are two complementary mechanisms*. Assertions declare constraints and exceptions prescribe actions. The presence of assertions

contributes to a more formal specification by expressing semantic properties of classes.

- *The Contract pattern can be a simple extension* for several object-oriented languages such as C++ [Stroustrup91], Java [Gosling+96], Smalltalk [Goldberg+83] that handle exceptions but do not support assertion mechanisms. Consider for example the assert macro in C++ that does not benefit from the flexibility of its exception mechanism. It uses the same approach as ANSI C, that is, an interrupt handler invoking the *exit()* function. All these languages can control execution in the case of exceptional situations, but don't have the capability of specifying assertions. One way to solve the problem would be to define an interface that combines assertions with exception handling so that pre- and post-conditions of methods and class invariants can be defined (see implementation section). This would allow the application of the *design by contract* methodology in the context of these languages.

The Contract pattern has the following drawbacks:

- *Execution overhead*. Since Java does not have a preprocessor, all assertions cannot be compiled conditionally. In order to remove this overhead, a small Java preprocessor may comment all assertions imported by the *Assertion* package.
- *Extra methods and attributes*. Another potential overhead of this idiom can occur when class has to define private methods and attributes to support the class invariant.

Implementation

A class annotated with an invariant and pre- and post-conditions for its methods may be regarded as a *contract*, since it specifies precisely the behavior of its instances and the constraints imposed on the interactions between the object and its clients.

In terms of implementation, an assertion is a simple boolean expression that we expect to be true. In practice, assertions come in pairs: a pre-condition and a post-condition. The pre-condition requires the correct specification at the method's entry; the post-condition ensures that the method's specification and object state are correct at exit. The following illustrates how assertions are localized in a method:

```
public void aMethod() {
    Assertion.require( booleanExpression );  // pre-condition

    // statements of the method...

    Assertion.ensure( booleanExpression );  // post-condition
    Assertion.ensure(    invariant()    );  // invariant
}
```

- *How do you observe that the required (input) conditions are not respected?* By using a pre-condition to denote a problem in the client software. The pre-condition of a method specifies what restrictions the client invoking a particular method is obliged to comply with. For example, the following is a method *first()* with a pre-condition *requiring a non-empty list* before returning the reference of the first node in a circular singly-linked list:

```
public Node first() {
    Assertion.require( count != 0 );
    return last.next;
}
```

Note that a missing pre-condition is equivalent to *require(true)*.

- *How do you make sure that the method fullfills the supplier's side of the contract?* By using a post-condition to denote a problem in the called method. The post-condition of a method states what obligations the supplier object has when executing the method, provided that the client's request satisfies the method's precondition. For example, the following is a method *clear()* with a post-condition *promising an empty circular singly-linked list* before returning:

```
public void clear() {
    Node n;

    while ( (n = getFirst()) != null )
        n = null; // delete (return it to GC)

    Assertion.ensure( count == 0   &&  last == null );
}
```

Note that a missing post-condition is equivalent to *ensure(true)*.

- *How do you make sure that a class is always in a stable state?* By defining the invariant properties—via a protected method—of the state of each instance of the class and possible subclasses.

```
public void dec() {
    Assertion.require( count > 0 );
    count--;
    Assertion.ensure( invariant() );
}

protected boolean invariant() {
    return lower <= count  &&  count <= upper;
}
```

Sample Code The following code shows how to implement the Contract pattern in Java:

```
// Assertion.java

package mdec.util;

class PreconditionException extends RuntimeException {
    PreconditionException() {
        super();
    }
}

class PostconditionException extends RuntimeException {
    PostconditionException() {
        super();
    }
}

public class Assertion {
    public static void require(boolean  expr)  {
        if ( !expr ) throw new PreconditionException();
    }

    public static void ensure (boolean  expr) {
        if ( !expr ) throw new PostconditionException();
    }
}
```

The *Assertion* package defines the *PreconditionException*, *PostconditionException*, and *Assertion* classes. Extending the *RuntimeException* allows to throw pre- or post-condition exceptions[5] without declaring them in *throws* clause. These two subclasses are private to the Assertion Class. We made this implementation decision to:

- hide the exception handling mechanism, and
- catch all exceptions generated by *require()* and *ensure()*

This programming pattern has the advantages of:

- not bothering the user to handle himself his exception(s)
- not cluttering the method signature with *throws* clause
- offering a Eiffel's *look and feel* in terms of require and ensure constructs

5 In the class *Assertion* and all classes that use the *require()* and *ensure()* methods.

Classes that uses *Assertion* have the ability of defining more robust classes. For example, *BoundedCounter* class imports *Assertion* (line 8) and implements its own invariant method (lines 59-61):

```
1   // BoundedCounter.java
2
3   /**
4    * A bounded counter is a counter that has a lower and
5    * upper bound that are set when constructing the object.
6    */
7
8   import mdec.util.Assertion;
9
10  class BoundedCounter {
11      public BoundedCounter(int initialCount,
12                            int lower, int upper) {
13          Assertion.require( lower < upper );
14          count = initialCount;
15          this.lower = lower;
16          this.upper = upper;
17          Assertion.ensure( invariant() );
18      }
19
20      public BoundedCounter(int lower, int upper) {
21          count = 0;
22          this.lower = lower;
23          this.upper = upper;
24          Assertion.ensure( invariant() );
25      }
26
27      public void inc() {
28          Assertion.require( count < upper );
29          count++;
30          Assertion.ensure( invariant() );
31      }
32
33      public void dec() {
34          Assertion.require( count > lower );
35          count--;
36          Assertion.ensure( invariant() );
37      }
38
39      public void add(int n) {
40          Assertion.require( n > 0  &&  count < upper );
41          hold();
42          count += n;
43          Assertion.ensure( valueOf() == oldCount + n );
44          Assertion.ensure( invariant() );
45      }
46
47      public int valueOf() {
48          return count;
49      }
50
51      private int count, lower, upper;
52
53      // Method(s) and variable(s) related to the class invariant.
54
55      private void hold() {
56          oldCount = count;
57      }
58
59      private boolean invariant() {
60          return count >= lower  &&  count <= upper;
61      }
62
63      private int oldCount;
64  }
```

Known Uses Eiffel [Meyer92] is a pure strong-typed object-oriented language supporting the concepts of assertions and exception mechanisms. Meyer [Meyer88] has proposed the *Design by Contract* methodology in the context of the Eiffel language. That methodology aims at improving software reliability through correctness and robustness.

With several extensions in C++ [Cline+90, Eliens95], Smalltalk [Carillo-Castellon+96] and Java [deChamplain97], the contract pattern is not anymore exclusive to Eiffel. This idiom can be implemented in other object-oriented languages that supports exception handling, eg, Ada, C++ and Modula-3.

Acknowlegments

I would like to thank Liping Zhao, Doug Lea, Ken Auer, Manish Bhatt, John Brant, Jim Doble, Neil Harrison, Philippe Lalanda, Eugene Wallingford, Stuart Yeates, Wolfgang Kreutzer, Paul Ashton, and Neville Churcher for their insightful comments on this paper.

References

[Carillo-Castellon+ 96] M. Carillo-Castellon et al., *Design by Contract in Smalltalk*, JOOP, Nov-Dec 1996, pp. 23-28.

[Cline+ 90] M. Cline and Lea. D, *The Behavior of C++ Classes*, In Proc. Symp. on Object-Oriented Programming, Marist College, 1990.

[deChamplain 97] M. de Champlain, *Designing by Contract in Java*, Invited Talk, Department of Electrical and Computer Engineering, Concordia University, May 1997.

[Eliens 95] A. Eliens, *Principles of Object-Oriented Software Development*, Addison-Wesley, 1995.

[Goldberg+ 83] A. Golberg. and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Gosling+ 96] J. Gosling et al., *The Java Language Specification*, Addison-Wesley, 1996.

[Meyer 88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[Meyer 92] B. Meyer, *Eiffel: the Language*, Prentice-Hall, 1992.

[Meyer93] B. Meyer. "Systematic Concurrent Object-Oriented Programming", Communication of The ACM, Vol36(9), pp56-80, 1993.

[Stroustrup 91] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Reading, MA, Addison-Wesley, 1991.