Java Framework Implementing Design Patterns by the Use of JML and Contract4J[*]

Gergely Kovásznai
Department of Information Technology
Eszterházy Károly College, Eger, Hungary
kovasz@aries.ektf.hu

December 20, 2010

Abstract

We develop a generic Java framework that implements software design patterns. In this paper, the Decorator and State patterns are focused. Nevertheless, our current results can be considered in connection with any design pattern which employs delegation constraints. We investigate how such constraints could be expressed in Java by the help of Design-by-Contract tools like JML or Contract4J. Furthermore, we illustrate how to use our generic framework, through examples.

1 Introduction

In order to continue the work that Wolfgang Schreiner has started [3], we are trying to develop a generic Java framework that implements further object-oriented design patterns [1], and to incorporate constraint specifications by using Design-by-Contract notations, like JML or Contract4J.

In the following sections, we are investigating typically such patterns that restrict the implementation of some methods to delegate to other methods. Such design patterns are, for instance, the Proxy, Decorator, Adapter, State, and Template Method patterns. We are trying to investigate how JML or Contract4J could express delegation constraints. As it is going to turn out, they are not expressive enough for this purpose, thus we have to incorporate delegation constraints directly in the Java source code.

In Section 2, we introduce some relevant features of JML and Contract4J. In Section 3 and Section 4, we show how the Decorator pattern and the State pattern can be formalized, respectively. In the appendices, we show examples uses of both patterns.

[*]Supported by the Austrian-Hungarian Intergovernmental S&T Cooperation Program (TéT) under the contract AT-25/2008

2 Design-by-Contract Tools

At first, we would like to show how JML supports the expressing of delegation constraints. After that, let us give a general overview on Contract4J.

2.1 JML Model Programs, and Problems

JML provides the *model program* specification, which makes it possible for the developer to specify the “skeleton” of a method [4]. Such a skeleton may contain mandatory method calls, but allows the concrete implementations to “fill to gap” among such calls. Model programs sound like a prefect solution for our problem, but unfortunately, they are not. The problem is that mandatory method calls (as Java commands) have to be fixed a priori. Then, the JML checker tries to compare implementation against model program, using *exact matching* of Java commands. For our objectives, exact matching is not sufficient since some details (e.g. the form of the actual parameters) cannot be fixed a priori.

Another serious limitation of JML is that it currently does not support Java generics. Since we are developing a generic Java framework, we cannot do without Java generics. Although one can force the JML Runtime Assertion Checker (`jmlc`) not to involve Java generics in checking, by using the `--generic` switch, groundless error messages related to generics may occur, as presented in the subsequent sections.

2.2 Contract4J

Contract4J [] is a Design-by-Contract tool based on Java annotations. Since annotations were introduced in Java 5, Contract4J supports Java generics as well. Contract4J provides 4 types of annotations:

- `@Contract`: Each class or interface that declares a contract must be annotated with `@Contract`.
- `@Invar`: Invariants can be assigned to classes, fields, constructors, or methods.
- `@Pre`, `@Post`: Preconditions and postconditions can be assigned to constructors or methods.

There are some special keywords that can be used in test expressions: `$this`, `$old`, `$return`, etc.

Contract4J can be regarded as a very simple Design-by-Contract tool, as compared to JML. For example, neither program models nor side effect constraints (`@assignable`) are supported. Furthermore, no quantifiers can be used in test expressions. Nevertheless, Contract4J also supports using model variables, and, what is important for us, does not produce groundless error messages when using generics.

3 The Decorator Pattern

The Decorator pattern can be used to allow to extend the functionality of a class dynamically (at runtime). The basic idea is illustrated by the class diagram in Figure 1, by which the participants can be specified as follows:

- An interface (`IComponent`) is located at the top of the class hierarchy. It defines a method (`Operation()`), which has `public` visibility.
- An abstract class (`Decorator`) implements `IComponent`. It has an instance variable (`component`) of type `IComponent`.
- Concrete classes may also implement `IComponent`. The ones which are however not derived from `Decorator` can be regarded as elementary components (`Component`). The rest of them (i.e., which are derived from `Decorator`) can be regarded as compound components (`DecoratorA`, `DecoratorB`).

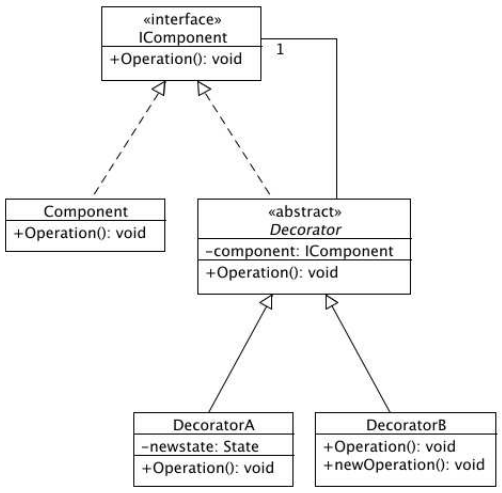

Figure 1: The Decorator pattern.

As can be seen, a recursive, linear hierarchy is formed here. Further intentions:

1. It is obligatory to define a constructor in the `Decorator` abstract class. This constructor has to receive an `IComponent` parameter, and to initialize the `component` field.
2. In the `Decorator` class or in its descendants, the `Operation` method must contain a delegation to the `component.Operation` method.

3.1 Java Implementation

However, in general, even several methods of `IComponent` could realize such delegations, we made a decision to define solely one such method (like in the Figure 1), but to make it very general by using generic parameter and result types. So, let us define the following Java interface:

```
public interface IComponent<R, A> {
  public R Operation(A arg);
}
```

The Decorator abstract class can be defined as follows:

```
public abstract class Decorator<R, A> implements IComponent<R, A> {
  protected IComponent<R, A> component;

  protected Decorator(IComponent<R, A> component) {
    this.component = component;
  }

  final public R Operation(A arg) {
    A postArg = Pre(arg);
    R preResult = component.Operation(postArg);
    return Post(postArg, preResult);
  }

  protected abstract A Pre(A arg);

  protected abstract R Post(A postArg, R preResult);
}
```

Note that the following constraints occur in the Decorator class:

- In the descending classes, the value of the instance variable `component` can be set only by calling the constructor of `Decorator`.
- The `Operation` method is `final`. Furthermore, its body contains a delegation to the `component.Operation` method, i.e., this is the place where *delegation constraints* occur.
- Thus, the behavior of descendant classes can only be specified by implementing the `Pre` and `Post` methods. The purpose of `Pre` is to customize the value of the parameter which is further passed to the `component.Operation` method. Similarly, the purpose of `Post` is to customize the return value.

In order to subclass from `Decorator`, developers must

- call the `Decorator` constructor from the constructor of the subclass (since no other constructor is defined in `Decorator`),

- and implement the methods `Pre` and `Post`.

In Appendix A, an example for such subclassing is shown.

3.2 JML Specification

Regarding to the constraints expressed in the intention of the Decorator pattern, only two simple constraints are needed additionally to our Java framework:

- The value of `component` must not be null ever.
- The value of `component` must not be modified after instantiation.

Thus, the Java implementation is to be extended in the following way:

```
public abstract class Decorator<R, A> implements IComponent<R, A> {
  protected /*@ spec_public */ IComponent<R, A> component;
  //@ constraint component == \old(component);

  //@ requires component != null;
  protected Decorator(IComponent<R, A> component) {
    this.component = component;
  }

  ...
}
```

When executing the JML Runtime Assertion Checker (`jmlc`), even together with the `--generic` switch, we get mysterious error messages. It is especially interesting that we do not get the same messages all the time. When executing the checker first time, the following error message occurs:

```
Too many parameters for type "IComponent"; required:  0
```

As can be seen, `jmlc` has not recognized that `IComponent` is a generic interface.

When executing the checker subsequently, we get error messages about assumed ambiguity of some methods. Let us show one of those dummy messages:

```
The top concrete method of the generic function is ambiguous in
this context between (at least) java.lang.Object.toString() and
                  java.lang.Object.toString()
```

Summing up, groundless error messages may be invoked by `jmlc` when the source code contains Java generics.

3.3 Contract4J Specification

Let us use Contract4J annotations instead of JML. After performing the installation and deployment steps listed in [2], we rewrite the source code of the `Decorator` class as follows:

```
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;
import org.contract4j5.contract.Pre;

@Contract
@Invar("$this.component == $old($this.component)")
public abstract class Decorator<R, A> implements IComponent<R, A> {
  protected IComponent<R, A> component;

  @Pre("component != null")
  protected Decorator(IComponent<R, A> component) {
    this.component = component;
  }

  ...
}
```

Note the following facts:

- Each class containing Contract4J annotations must be annotated with `@Contract`.
- In the case of `Decorator`, a *class invariant* is needed. Here, the form `$this.field` is necessary.

The source code in Appendix A applies Contract4J annotations.

4 The State Pattern

The State pattern allows an object to alter its behavior when its internal state changes. A class which represents such internal states is needed, as well as another class that represents the overall context and includes a reference to the current state object. The required class hierarchy can be seen in Figure 2, by which the participants can be specified as follows:

1. An interface (`State` or `IState`) is the ancestor of all the classes represent internal states. `IState` defines a method (`Handle()`), which has `public` visibility,
2. An abstract class (`Context`) that has an instance variable (`state`) of type `IState` defines a method (`Request`) that delegates to the `state.Handle` method.
3. It is optional to make the caller context instance accessible from the concrete state classes (`ConcreteStateA`, `ConcreteStateB`).

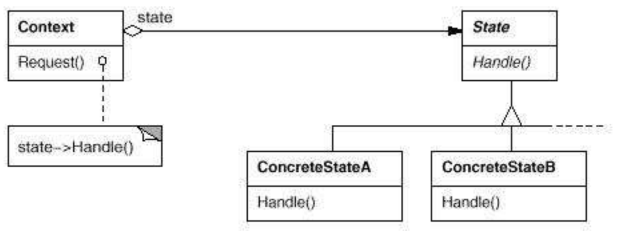

Figure 2: The State pattern.

4.1 Java Implementation

Let us define the following Java interface:

```
public interface IState<R1, A1, R2, A2> {
  public R2 Handle(A2 arg, Context<R1, A1, R2, A2> context);
}
```

As can be seen, the `Handle` method receives an argument of type `A2`, and returns a value of type `R2`. Furthermore, it can receive a context instance, to which the type parameters `A1` and `R1` related.

The Context abstract class can be defined as follows:

```
public abstract class Context<R1, A1, R2, A2> {
  protected IState<R1, A1, R2, A2> state;

  public IState<R1, A1, R2, A2> getState() {
    return state;
  }

  public void setState(IState<R1, A1, R2, A2> state) {
    this.state = state;
  }

  public Context(IState<R1, A1, R2, A2> state) {
    this.state = state;
  }

  final public R1 Request(A1 arg) {
    A2 postArg = Pre(arg);
    R2 preResult = state.Handle(postArg, this);
    return Post(postArg, preResult);
  }

  protected abstract A2 Pre(A1 arg);
  protected abstract R1 Post(A2 postArg, R2 preResult);
}
```

Note that this source code is very similar to the one for the Decorator pattern. But there is an important difference: the methods `Request` and `Handle` may differ from each other in parameter and return types. Furthermore, the current `Context` instance (i.e., `this`) is passed as a parameter to the `state.Handle` method.

In Appendix B, an example for subclassing is shown.

4.2 JML Specification

There is another difference as compared to the Decorator pattern: the value of the instance variable `state` may change, however, it must not be null ever. Thus, the Java implementation is to be extended with a single constraint:

```
public abstract class Context<R1, A1, R2, A2> {
protected /*@ spec_public */ IState<R1, A1, R2, A2> state;
//@ invariant state != null;

  ...
}
```

When executing `jmlc`, no error message is occurred. However, something mysterious happens when adding a new subclass (`ConcreteContext`) of `Context`. The following error message occurs:

```
Abstract method "Context.Pre(java.lang.Object)" in concrete class
                        "ConcreteContext"
```

So, it seems that generics cause the problem again.

4.3 Contract4J Specification

We rewrite the source code of the `Context` class as follows:

```
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;

@Contract
@Invar("$this.state != null")
public abstract class Context<R1, A1, R2, A2> {
protected IState<R1, A1, R2, A2> state;

  ...
}
```

The source code in Appendix B applies Contract4J annotations.

5 Conclusions

We have continued our investigation on building a generic Java framework for design pattern specification. Since this framework is intended to be general, we need Java generics. Unfortunately, JML, as a Design-by-Contract tool, does not support generics, as shown in the previous sections. Therefore, we have tried to find another promising Design-by-Contract tool. Although Contract4J supports generics, it provides much less annotations as JML does.

Primarily, we have tried to investigate how to express delegation constraints either in JML or in Contract4J. We have not found satisfactory solutions. Nevertheless, the framework can be implemented in such a way that the Java source code itself enforces the fulfillment of those constraints. This fact has been demonstrated by us through the Decorator and the State patterns.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1995.

[2] I. A. Krizsan, *Getting Started with Contract4J.* 2008.

[3] Wolfgang Schreiner, *A JML Specification of the Design Pattern "Proxy".* RISC, Austria, 2009.

[4] S. M. Shaner, G. T. Leavens, D. A. Naumann, *Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs.* OOPSLA 2007.

A Example Use of the Decorator Pattern

This example illustrates how to use our Decorator pattern framework. We are presenting how to develop a simplified scrollable window system.

First, let us show the content of our framework: the `IComponent` interface and the `Decorator` abstract class.

```
/**************************************************************
 * IComponent.java
 **************************************************************/
public interface IComponent<R, A> {
  public R Operation(A arg);
}


/**************************************************************
 * Decorator.java
 **************************************************************/
```

```
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;
import org.contract4j5.contract.Pre;

@Contract
@Invar("$this.component == $old($this.component)")
public abstract class Decorator<R, A> implements IComponent<R, A> {
  protected IComponent<R, A> component;

  @Pre("component != null")
  protected Decorator(IComponent<R, A> component) {
    this.component = component;
  }

  final public R Operation(A arg) {
    A postArg = Pre(arg);
    R preResult = component.Operation(postArg);
    return Post(postArg, preResult);
  }

  protected abstract A Pre(A arg);

  protected abstract R Post(A postArg, R preResult);
}
```

Let us introduce an interface for windows, which provides queriable width and height. This interface extends the `IComponent<Boolean, Point>` interface since we intend to use the `Operation` method to answer if a given point is currently visible in the given window.

```
/************************************************************
 * IWindow.java
 ************************************************************/
import java.awt.Point;

public interface IWindow extends IComponent<Boolean, Point> {
  public int getWidth();
  public int getHeight();
}
```

The following class represents a concrete, elementary window. As it has already been mentioned, the `Operation` method receives a `Point` parameter, and returns a `Boolean` value represents the visibility of the given point.

```
/************************************************************
 * WindowComponent.java
 ************************************************************/
```

```
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;
import java.awt.Point;

@Contract
public class WindowComponent implements IWindow {
  @Invar("$this.width > 0 && $this.height > 0")
  private int width, height;

  public int getWidth() {
    return width;
  }

  public int getHeight() {
    return height;
  }

  public WindowComponent(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public Boolean Operation(Point point) {
    return new Boolean(point.x >= 0 && point.x < width
      && point.y >= 0 && point.y < height);
  }
}
```

The following two classes represent a horizontally and a vertically scrollable area, respectively. E.g., the horizontally scrollable area is represented by the x coordinate of its left border and its width. Both classes have a `Scroll` method which shifts the scrollable area with the given scale.

```
/*************************************************************
 * HorizontalScrollbarDecorator.java
 ************************************************************/
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;
import java.awt.Point;

@Contract
public class HorizontalScrollbarDecorator
  extends Decorator<Boolean, Point> implements IWindow {
    @Invar("$this.left >= 0 && $this.width > 0")
    protected int left, width;

    public int getWidth() {
      return width;
    }

    public int getHeight() {
      IWindow window = (IWindow) component;
      return window.getHeight();
    }

    public HorizontalScrollbarDecorator(IWindow window, int width) {
      super(window);

      this.left = 0;
      this.width = width;
    }

    protected Point Pre(Point point) {
      return point;
    }

    protected Boolean Post(Point point, Boolean preResult) {
      return new Boolean(preResult.booleanValue() && point.x >= left
        && point.x < left + width);
    }

    protected void Scroll(int dx) {
      IWindow window = (IWindow) component;
      if (left + dx >= 0 && left + dx + width < window.getWidth())
      left += dx;
    }
}


/**************************************************************
 * VerticalScrollbarDecorator.java
 **************************************************************/
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;
import java.awt.Point;

@Contract
public class VerticalScrollbarDecorator
  extends Decorator<Boolean, Point>implements IWindow {
    @Invar("$this.top >= 0 && $this.height > 0")
    protected int top, height;

    public int getWidth() {
```

```
    IWindow window = (IWindow) component;
    return window.getWidth();
  }

  public int getHeight() {
    return height;
  }

  public VerticalScrollbarDecorator(IWindow window, int height) {
    super(window);

    this.top = 0;
    this.height = height;
  }

  protected Point Pre(Point point) {
    return point;
  }

  protected Boolean Post(Point point, Boolean preResult) {
    return new Boolean(preResult.booleanValue() && point.y >= top
      && point.y < top + height);
  }

  protected void Scroll(int dy) {
    IWindow window = (IWindow) component;
    if (top + dy >= 0 && top + dy + height < window.getHeight())
    top += dy;
  }
}
```

Finally, let us introduce a possible `main` method. It instantiates an elementary 200×200 window, and then makes it vertically scrollable, where the scrollable area has the width of 50. Then, we make this vertically scrollable area even horizontally scrollable, with the height of 100. Finally, we demonstrate the use of the `Operation` and `Scroll` methods.

```
/**************************************************************
 * Main.java
 **************************************************************/
import java.awt.Point;

public class Main {
  public static void main(String[] args) {
    WindowComponent window = new WindowComponent(200,200);

    VerticalScrollbarDecorator verticalScrollbar =
      new VerticalScrollbarDecorator(window, 50);
    HorizontalScrollbarDecorator horizontalScrollbar =
      new HorizontalScrollbarDecorator(verticalScrollbar, 100);

    System.out.println(horizontalScrollbar.Operation(new Point(10, 60)));

    verticalScrollbar.Scroll(20);
    System.out.println(horizontalScrollbar.Operation(new Point(10, 20)));

    horizontalScrollbar.Scroll(20);
    System.out.println(horizontalScrollbar.Operation(new Point(10, 20)));
  }
}
```

B Example Use of the State Pattern

Let us illustrate how to use our State pattern framework, through a less meaningful example as compared to Appendix A.

First, let us show the content of our framework: the `IState` interface and the `Context` abstract class.

```
/************************************************************
 * IState.java
 ************************************************************/
public interface IState<R1, A1, R2, A2> {
  public R2 Handle(A2 arg, Context<R1, A1, R2, A2> context);
}


/************************************************************
 * Context.java
 ************************************************************/
import org.contract4j5.contract.Contract;
import org.contract4j5.contract.Invar;

@Contract
@Invar("$this.state != null")
public abstract class Context<R1, A1, R2, A2> {
  protected IState<R1, A1, R2, A2> state;

  public IState<R1, A1, R2, A2> getState() {
    return state;
  }

  public void setState(IState<R1, A1, R2, A2> state) {
    this.state = state;
  }

  public Context(IState<R1, A1, R2, A2> state) {
    this.state = state;
  }

  final public R1 Request(A1 arg) {
    A2 postArg = Pre(arg);
    R2 preResult = state.Handle(postArg, this);
    return Post(postArg, preResult);
  }

  protected abstract A2 Pre(A1 arg);

  protected abstract R1 Post(A2 postArg, R2 preResult);
}
```

Let us derive a concrete class from `Context`. The `Operation` method of this class has a parameter and return value of `String` type, and delegates an `Integer` value to the `state.Handle` method, which returns also an `Integer` value. Therefore, the `Pre` method must receive a `String` and return an `Integer`, and the `Post` method must receive `Integer` values and return a `String`.

For illustration, let us introduce an instance variable `numOfRequests` in order to count how many times `Handle` has been invoked. We intend to access `numOfRequests` from concrete state classes later.

```
/**********************************************************
 * ConcreteContext.java
 **********************************************************/
public class ConcreteContext
  extends Context<String, String, Integer, Integer> {
    public int numOfRequests = 0; //storing some additional data

    public ConcreteContext(
      IState<String, String, Integer, Integer> state) {
        super(state);
    }

    protected Integer Pre(String arg) {
      numOfRequests++;

      if (arg == null)
        return new Integer(0);
      else
        return new Integer(arg.length());
    }
    protected String Post(Integer postArg, Integer preResult) {
      int result = postArg.intValue() + preResult.intValue();
      return "The result: " + result;
    }
}
```

The following two classes implement the `IState` interface. The `ConcreteStateB` class accesses the `numOfRequests` field of the `ConcreteContext` class.

```
/**********************************************************
 * ConcreteStateA.java
 **********************************************************/
public class ConcreteStateA
  implements IState<String, String, Integer, Integer> {
    public Integer Handle(Integer arg,
      Context<String, String, Integer, Integer> context) {
        return new Integer(arg.intValue() + 10);
    }
}


/**********************************************************
 * ConcreteStateB.java
 **********************************************************/
public class ConcreteStateB
  implements IState<String, String, Integer, Integer> {
    public Integer Handle(Integer arg,
      Context<String, String, Integer, Integer> context) {
        int inc = 0;
        if (context instanceof ConcreteContext) //using the context
          inc = ((ConcreteContext) context).numOfRequests;
        return new Integer(arg.intValue() + inc);
    }
}
```

The following `main` method creates a concrete context instance. At first, we set the state of the context to a `ConcreteStateA`, and then, to a `ConcreteStateB`, and we invoke the `Request` method in both cases. Let us recommend to compare the results of those invocations.

```
/**********************************************************
 * Main.java
 **********************************************************/
public class Main {
  public static void main(String[] args) {
    ConcreteContext c = new ConcreteContext(new ConcreteStateA());
    System.out.println(c.Request("Hello"));
    c.setState(new ConcreteStateB());
    System.out.println(c.Request("Hello"));
  }
}
```