A JML Specification of the Design Pattern "Proxy"[*]

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

April 22, 2009

Abstract

We describe a generic Java framework that implements the software design pattern "proxy" (in two variants "virtual proxy" and "remote proxy") and that is formally specified in the Java Modeling Language (JML). In addition to the information provided by a typical UML specification of the pattern, the JML specification exactly describes how a request issued to the proxy is propagated to the underlying object and how the result is forwarded from this object to the user of the proxy.

Contents

[*]Supported by the Austrian Academic Exchange Service (ÖAD) under the contract HU 14/2009.

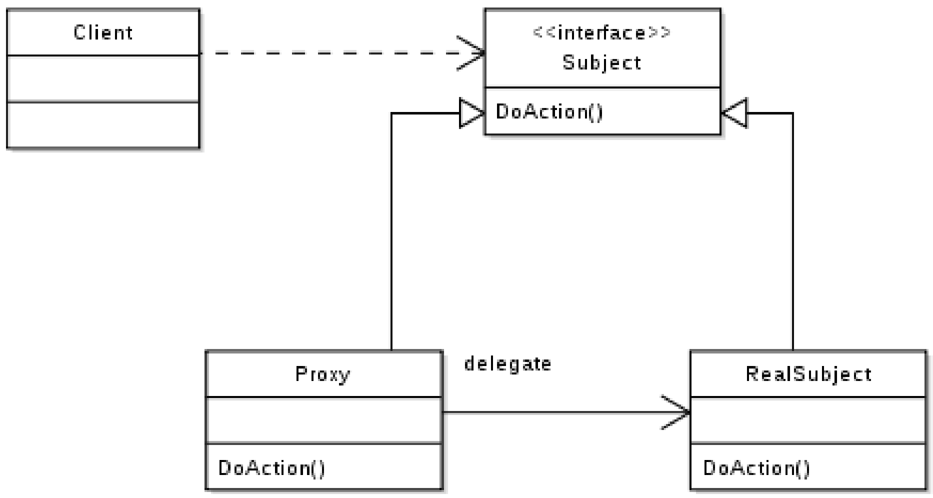

Figure 1: Proxy in UML (from Wikipedia)

1 Introduction

Like most software design patterns [1], the pattern “Proxy” is typically documented by a UML diagram (see Figure 1) accompanied by an extensive verbal description that explains the core idea, gives examples, indicates its applicability, outlines possible use cases, and so on. For example, the core description of Wikipedia on the topic “Proxy pattern” is as follows:

> A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. ...
>
> The following Java example illustrates the “virtual proxy” pattern. The `ProxyImage` class is used to delay the expensive operation of loading a file from disk until the result of that operation is actually needed. If the file is never needed, then the expensive load has been totally eliminated. ...

While in [1] the description is much more extensive and covers a dozen pages or so, the basic style is the same: UML diagrams and informal verbal descriptions aided by examples and code snippets. The only formally precise definition is represented by the UML diagrams which however typically only cover the static relationship between the classes/objects that make up the pattern. For instance, the diagram in Figure 1 says that an Object of type `Proxy` has a method `DoAction()` and owns a reference to an object of type `RealSubject` that has the same method; both `Proxy` and `RealSubject` thus implement an abstract interface `Subject` which is seen by the `Client` object. What the exact interpretation of the tag "delegate" in the diagram is (i.e. how the two methods `DoAction()` are related) is only covered by the informal verbal description.

"Proxy" is considered as a "structural" design pattern (in contrast to e.g. a "behavioral" one) i.e. it describes mainly a particular form of code organization without much interesting behavior described by the pattern itself.

Nevertheless, our goal is to make also the dynamic relationship between the methods formally precise; for this purpose we encode the pattern in a reusable Java framework which we formally specify with the Java Modeling Language (JML), a behavioral interface specification language for Java [3]. The remainder of this paper is structured as follows: In Section 2, we sketch the design of the framework. In Section 3, we discuss the JML specification of the framework. In Section 4, we summarize our experience and present our conclusions. Appendix A presents the full JML-annotated Java source code.

2 A Generic Proxy Framework in Java

Before developing a JML specification for the "Proxy" pattern, we first have to decide in which way to formulate the pattern in concrete Java code. One option is to develop a concrete example application of the pattern and to specify this application; however, then it is difficult to differentiate between those features that are fundamental to the pattern and those features that are peculiar for the particular use of the pattern. On the other side, we cannot remain completely abstract because a JML specification depends on a concrete interface for proxies and objects.

As a consequence, we made a decision to fix the action of the basic object to the concrete method

```
public R request(T arg);
```

but make it generic in the argument type T and result type R, i.e. we use a generic type interface

```
public interface Subject<T,R>
{
  public R request(T arg);
}
```

Generic types are a feature provided by Java 5 and 6, which are not generally supported by JML tools. However, the JML type checker `jml` already provides an option `-G` which allows to use also generic types in JML specifications; on the long term full JML support for generics can be expected.

As for proxies, we provide an interface

```
public interface Proxy<T,R>
{
  public R request(T arg) throws Exception;
}
```

whose `request` method is more general than the corresponding method in `Subject` in that it also allows the method to throw an exception. This extension reflects the fact that a proxy is *not* the same as the underlying object but that some mechanism is required to delegate requests from the proxy to its object. This delegation may not work, e.g. if the object cannot be created or not be located or if the propagation of arguments and results between proxy and object fails. The proxy can indicate any such situation by raising an exception, correspondingly also the user of the proxy must expect such an exception (even if the underlying object does not raise one).

Proxies can be created for various reasons, e.g. for deferring the actual allocation of the object (a *virtual proxy*) or for providing a local representative for a not directly accessible remote object (a *remote proxy*). Since these proxies have different requirements, we provide different implementations of `Proxy`.

Virtual Proxies A virtual proxy is defined as follows:

```
public class VirtualProxy<T,R,C> implements Proxy<T,R>
{
  private final SubjectCreator<T,R,C> creator;
  private final C ref;
  private Subject<T,R> vsubject;

  public VirtualProxy(SubjectCreator<T,R,C> creator, C ref)
  {
    this.creator = creator;
    this.ref = ref;
    this.vsubject = null;
  }

  public R request(T arg) throws Exception
  {
    if (vsubject == null) vsubject = creator.create(ref);
    return vsubject.request(arg);
  }
}
```

The virtual proxy encapsulates a local reference `vsubject` to the object to which the request is delegated. However, the creation of this object is deferred, thus the proxy needs a description `ref` of the object and an object creation mechanism `creator` such that `creator.create(ref)` constructs the new object. The type `C` of `ref` is generic, the type of the `creator` is determined by the generic interface

```
public interface SubjectCreator<T,R,C>
{
  public Subject<T,R> create(C ref);
}
```

Remote Proxies A remote proxy is defined as follows:

```
public class RemoteProxy<T,R,C> implements Proxy<T,R>
{
  private final SubjectReferer<T,R,C> referer;
  private final C ref

  public RemoteProxy(SubjectReferer<T,R,C> referer, C ref)
  {
    this.referer = referer;
    this.ref = ref;
  }

  public R request(T arg) throws Exception
  {
    return referer.request(ref, arg);
  }
}
```

Like a virtual proxy, also a remote proxy encapsulates an object description `ref` of generic type `C`. However, different from a proxy the object is never locally created. Rather a `referer` object is consulted every time a request is issued to the proxy to forward the request to the remote object denoted by `ref`. The interface of `referer` correspondingly is

```
public interface SubjectReferer<T,R,C>
{
  public R request(C ref, T arg) throws Exception;
}
```

Example uses of `VirtualProxy` and `RemoteProxy` are shown in class `Main` in Appendix A.4. Our goal is now to specify in JML the interfaces and classes described above.

3 A JML Specification of the Framework

The core problem of the JML specification is to describe that the proxy does not "make up" the result of a request i.e.

1. that it indeed invokes the request method of the underlying object,
2. that it forwards the received argument without change to the request,
3. that it returns the received result without change to the client.

We attempt a solution to this problem by introducing in interface `Subject` three JML model variables (specification-only mathematical variables) `calls`, `larg`, and `result` that denote the number of invocations of method `request`, the argument of the last request, and the result of the last request, respectively. The method `request` is annotated appropriately to ensure this information in the poststate of every call:

```
  public interface Subject<T,R>
  {
    /*@ public instance model int calls;
      @ public initially calls == 0;
      @*/

    /*@ public instance model T larg; @*/
    /*@ public instance model T lresult; @*/

    /*@ public normal_behavior
      @ assignable \everything;
      @ ensures calls == \old(calls)+1;
      @ ensures larg == arg && lresult == \result;
      @*/
    public R request(T arg);
  }
```

Next we introduce in interface `Proxy` a model variable `subject` that represents the object behind the proxy. The specification of the normal behavior of method `request` make sure that the object's request method is invoked exactly once with the argument provided to the proxy's `request` method and that the result is appropriately propagated from the object to the client of the proxy:

```
  public interface Proxy<T,R>
  {
    /*@ public nullable instance model Subject<T,R> subject; @*/

    /*@ public normal_behavior
      @ ensures subject != null;
      @ ensures \old(subject) != null ==> subject == \old(subject);
      @
      @ ensures \old(subject) == null ==> subject.calls == 1;
      @ ensures \old(subject) != null ==>
      @               subject.calls == \old(subject.calls)+1;
      @ ensures subject.larg == arg && subject.lresult == \result;
      @
      @ also public exceptional_behavior
      @ assignable \everything;
      @ signals(Exception e)
      @   (\old(subject) == null ==>
      @      subject == null ||
      @      subject.calls == 0 || subject.calls == 1) &&
      @   (\old(subject) != null ==>
      @     subject == \old(subject) &&
      @       (subject.calls == \old(subject.calls) ||
      @        subject.calls == \old(subject.calls)+1));
      @*/
  public R request(T arg) throws Exception;
}
```

The reason that the specification declares `subject` as `nullable` (i.e. `subject` may be null) is that it does not demand that the subject already exists in the prestate of the first request. It only demands its existence in the post-state and that, once it has been allocated, its identity does not change. Therefore various possibilities have to be considered in the normal poststate depending on the fact whether the object has already existed in the prestate or not.

Furthermore, the proxy's `request` is annotated with an exceptional behavior which is always enabled and which ensures that the object's `request` method is called *at most* once. This is because it may happen that the proxy's request fails before the object's method has been invoked or it may fail after the invocation. Furthermore, the case has to be considered that the object has not yet existed in the prestate.

Virtual Proxy The specification of class `VirtualProxy` introduces public model variables `pcreator`, `pref`, `psubject` represented by the corresponding private object variables; in the public specification of the constructor and the `request` method, the model variables serve as substitutes for the project variables which (since they are private) cannot be referenced there. Furthermore, the virtual proxy defines the representation of the model variable `subject` in specification `Proxy` by its actual object variable `vsubject`.

With these provisions, the constructor can be specified to store the arguments in the object and set its local object variable to `null`. The `request` method specializes the specification inherited from `Proxy` by indicating that on first call of the method the new object is derived from the result of the `creator` with which the proxy has been initialized:

```
public class VirtualProxy<T,R,C> implements Proxy<T,R>
{
  /*@ public model SubjectCreator<T,R,C> pcreator; @*/
  private final SubjectCreator<T,R,C> creator; /*@ in pcreator; @*/
  /*@ private represents pcreator <- creator; @*/

  /*@ public model C pref; @*/
  private final C ref; /*@ in pref; @*/
  /*@ private represents pref <- ref; @*/

  /*@ public model Subject<T,R> psubject; @*/
  /*@ nullable @*/ private Subject<T,R> vsubject; /*@ in psubject; @*/
  /*@ private represents psubject <- vsubject; @*/

  /*@ private represents subject <- vsubject; @*/

  /*@ public normal_behavior
    @ assignable pcreator, pref, psubject;
    @ ensures pcreator == creator && pref == ref && psubject == null;
    @*/
  public VirtualProxy(SubjectCreator<T,R,C> creator, C ref)
  {
    this.creator = creator;
    this.ref = ref;
    this.vsubject = null;
  }

  /*@ also public normal_behavior
    @ assignable \everything;
    @ ensures \old(psubject) == null ==>
    @           pcreator.calls == \old(pcreator.calls)+1 &&
    @           pref == pcreator.larg && psubject == pcreator.lresult;
    @*/
  public R request(T arg) throws Exception
  {
    if (vsubject == null) vsubject = creator.create(ref);
    return vsubject.request(arg);
  }
}
```

The specification of interface `SubjectCreator` is similar to that of `Subject` in that it introduces model variables to denote the number of invocations of `create` and the argument and the result of the last invocation:

```
public interface SubjectCreator<T,R,C>
{
  /*@ public instance model int calls;
    @ public initially calls == 0;
    @*/

  /*@ public instance model C larg; @*/
  /*@ public instance model C lresult; @*/

  /*@ public normal_behavior
    @ assignable \everything;
    @ ensures \result != null && \fresh(\result);
    @ ensures calls == \old(calls)+1;
    @ ensures larg == ref && lresult == \result;
    @*/
  public Subject<T,R> create(C ref);
}
```

Remote Proxy Similar to `VirtualProxy`, also `RemoteProxy` introduces public model variables for the private object variables for use in the public specifications. The specification variable `subject` is now represented by looking up among the objects known to the referrer object that with the name `(p)ref` (see the explanation below). The constructor is specified to store its arguments in the proxy; the `request` method is specified to invoke the referrer object with the stored object reference and the argument to the request:

```
  public class RemoteProxy<T,R,C> implements Proxy<T,R>
  {
```

```
  /*@ public model SubjectReferer<T,R,C> preferer; @*/
  private final SubjectReferer<T,R,C> referer; /*@ in preferer; @*/
  /*@ private represents preferer <- referer; @*/

  /*@ public model C pref; @*/
  private final C ref; /*@ in pref; @*/
  /*@ private represents pref <- ref; @*/

  /*@ public represents subject <-
    @  (Subject<T,R>)(preferer.subjects.get(pref)); @*/

  /*@ public normal_behavior
    @ assignable preferer, pref;
    @ ensures preferer == referer && pref == ref;
    @*/
  public RemoteProxy(SubjectReferer<T,R,C> referer, C ref)
  {
    this.referer = referer;
    this.ref = ref;
  }

  /*@ also public normal_behavior
    @ assignable \everything;
    @ ensures preferer.calls == \old(preferer.calls)+1;
    @ ensures pref == preferer.lref && arg == preferer.larg;
    @ ensures \result == preferer.lresult;
    @*/
  public R request(T arg) throws Exception
  {
    return referer.request(ref, arg);
  }
}
```

The interface `SubjectReferer` introduces as usual model variables for the number of invocations of `request` and for the arguments and the result of the last request, respectively. Furthermore, it introduces a model variable `subject` representing a mapping of an object name of type `C` to an object of type `Subject<T,R>`. This mapping is of raw type `HashMap` (rather than `HashMap<C,Subject<T,R>>`) because the `jml` type checker uses a non-generic definition of `HashMap`.

In the specification of the normal behavior of `request`, we introduce a local specification variable `subject` to denote the object referenced by `ref` in `subject` and demand that this object does actually exist (i.e. `subject` is not null). As usual, the number of calls and the last argument and result are preserved. More important, we demand that the `request` method of `subject` is invoked with the argument of the proxy request and that its result is returned to the client of the proxy.

```
public interface SubjectReferer<T,R,C>
{
    /*@ public instance model HashMap subjects; @*/

    /*@ public instance model int calls;
      @ public initially calls == 0;
      @*/

    /*@ public instance model C lref; @*/
    /*@ public instance model C larg; @*/
    /*@ public instance model C lresult; @*/

    /*@ public normal_behavior
      @ forall Subject<C,R> subject;
      @ requires subject == subjects.get(ref) && subject != null;
      @ assignable \everything;
      @
      @ ensures calls == \old(calls)+1;
      @ ensures lref == ref && larg == arg && lresult == \result;
      @
      @ ensures subject.calls == \old(subject.calls)+1;
      @ ensures subject.larg == arg && subject.lresult == \result;
      @
      @ also public exceptional_behavior
      @ forall Subject<C,R> subject;
      @ requires subject == subjects.get(ref) && subject != null;
      @ assignable \everything;
      @ signals (Exception e) subject.calls == \old(subject.calls) ||
      @                       subject.calls == \old(subject.calls)+1;
      @
      @ also public exceptional_behavior
      @ forall Subject<C,R> subject;
      @ requires subject == subjects.get(ref) && subject == null;
      @ assignable \nothing;
      @ signals (Exception e) true;
      @*/
  public R request(C ref, T arg) throws Exception;
}
```

The exceptional behavior of `request` may be triggered if no object named `ref` is known to the referrer; in this case the specification does not allow any state change (but this is not really a core requirement).

Caching Proxies Since the request method specified in `Subject` allows global state changes, the specification does not allow to "cache" the results of previous requests and return them without invoking the method. For such a form of proxy, one might derive from `Subject` a special interface `PureSubject` where the request denotes a pure mathematical function; correspondingly one might define an interface `CachingProxy` which drops the requirement that the object's request method is invoked on every request issued to the proxy. Since our definition of `Proxy` is not generic on the subject type, `CachingProxy` would be the origin of another proxy hierarchy with implementations of new classes `VirtualCachingProxy` and `RemoteCachingProxy` not related to the proxy classes defined above. More research, however, might reveal a better class structure that allows to make non-caching proxies subclasses of caching proxies.

4 Conclusions

The JML specification of the structural design pattern "Proxy" mainly relates a call of a "proxy" object to a call of an underlying object by ensuring that the result of the proxy is not "made up" but derived from actually invoking the underlying object's request method. For this purpose, we have introduced in the specification additional model variables that allow to ensure that a particular method is invoked and to remember the argument and result of the last invocation. Thus it becomes possible to relate the specification of one method call to the specification of another one, something which is not possible by URL class/object diagrams alone.

One can consider this technique as the special case of a "history variable" [2] that records previous states; we might e.g. use a vector to remember all previous invocations of a method (and not just the last one). In this way, it would also become possible to relate one method invocation to *sequences* of previous method invocations. It is, however, unclear whether this is really the appropriate way to formulate *temporal* properties of programs. As an alternative, e.g. a temporal logic extension of JML has been proposed [4] that operates on a higher level to describe sequences of program events (where an event is e.g. calling a particular method or returning from a particular method). However, this extension is expressed in a propositional logic framework which does not allow to refer to the variables of a previous state; thus specifications like the ones presented in this paper are not yet possible.

Our experience is currently still limited to the specification of the one design pattern presented in this paper; further work will reveal whether JML is a generally applicable framework for design pattern specification.

References

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

[2] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison Wesley, 2002.

[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06y, Department of Computer Science, Iowa State University, June 2004. See `www.jmlspecs.org`.

[4] Kerry Trentelman and Marieke Huisman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST'02), St. Gilles les Bains, Reunion Island, France, September 9–13*, volume 2422 of *Lecture Notes in Computing Series*, pages 334—348. Springer, 2002.

A The JML-annotated Java Code

A.1 Proxy

```
package patterns.proxy;

/*****************************************************************************
 * An interface to an object for which a proxy may be supplied.
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 ****************************************************************************/
public interface Subject<T,R>
{
  /** the number of calls of the request method **/
  /*@ public instance model int calls;
    @ public initially calls == 0;
    @*/

  /** the argument of the last request **/
  /*@ public instance model T larg; @*/

  /** the result of the last request **/
  /*@ public instance model T lresult; @*/

  /***************************************************************************
   * Perform a request.
   *
   * The method increases the specification's call counter and sets the
   * specification variables to remember the argument and result.
   *
   * @param arg the argument to the request
   * @return result the request result.
   **************************************************************************/
  /*@ public normal_behavior
    @ assignable \everything;                    // allow global effects
    @ ensures calls == \old(calls)+1;            // increase call counter
    @ ensures larg == arg && lresult == \result; // remember argument/result
    @*/
  public R request(T arg);
}

package patterns.proxy;

/*****************************************************************************
 * A proxy for another object of type Subject<T,R>
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 ****************************************************************************/
public interface Proxy<T,R>
{
  /***************************************************************************
   * a specification of the actual subject behind this proxy.
   **************************************************************************/
  /*@ public nullable instance model Subject<T,R> subject; @*/

  /***************************************************************************
   * the proxy request method; ensures that the subject's request method
   * is called with the proxy's request argument and that its result is
   * returned by the proxy request.
   *
   * However, the proxy request method may also throw an exception indicating
   * that it was for some reason not possible to get a result from an
   * invocation of the subject's method (either the method could not
   * be invoked or its result could not be retrieved).
   *
   * @param arg the request argument
   * @return result the request result
   * @throws Exception a exception indicating why the proxy call failed
   **************************************************************************/
  /*@ public normal_behavior
    @
    @ // side effects are allowed
    @ assignable \everything;
    @
    @ // if the call returns normally, the subject's request method was called
    @ ensures subject != null;
    @ ensures \old(subject) != null ==> subject == \old(subject);
    @ ensures \old(subject) == null ==> subject.calls == 1;
    @ ensures \old(subject) != null ==> subject.calls == \old(subject.calls)+1;
    @
    @ // the subject's request method was called with the given argument
    @ // and we return its result
    @ ensures subject.larg == arg && subject.lresult == \result;
    @
    @ // in the exceptional case we may or may have not called the method
    @ // side effects are also allowed
    @ also public exceptional_behavior
    @ assignable \everything;
    @ signals(Exception e)
    @   (\old(subject) == null ==>
    @      subject == null || subject.calls == 0 || subject.calls == 1) &&
    @   (\old(subject) != null ==>
    @     subject == \old(subject) &&
    @       (subject.calls == \old(subject.calls) ||
    @        subject.calls == \old(subject.calls)+1));
    @*/
  public R request(T arg) throws Exception;
}
```

A.2 Virtual Proxy

```
package patterns.proxy;

/******************************************************************************
 * The creator of an object for which a proxy may be supplied.
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 *****************************************************************************/
public interface SubjectCreator<T,R,C>
{
  /** the number of calls of the create method **/
  /*@ public instance model int calls;
    @ public initially calls == 0;
    @*/

  /** the argument of last call of create **/
  /*@ public instance model C larg; @*/

  /** the result of last call of create **/
  /*@ public instance model C lresult; @*/

  /***************************************************************************
   * Create a subject from a description.
   * @param ref a description of the subject.
   * @return the object (not null)
   **************************************************************************/
  /*@ public normal_behavior
    @ assignable \everything;  // global state may be changed
    @ ensures \result != null && \fresh(\result); // a new object is returned
    @ ensures calls == \old(calls)+1;              // count call
    @ ensures larg == ref && lresult == \result;  // remember effect of call
    @*/
  public Subject<T,R> create(C ref);
}

package patterns.proxy;

/******************************************************************************
 * A virtual proxy, i.e. a proxy (@see patterns.proxy.Proxy) which defers
 * creation of the object until the first invokation of an object request.
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 * @param <C> the type of the object description (used for object creation).
 *****************************************************************************/
public class VirtualProxy<T,R,C> implements Proxy<T,R>
{
  /** the creator of the subject **/
  /*@ public model SubjectCreator<T,R,C> pcreator; @*/
  private final SubjectCreator<T,R,C> creator; /*@ in pcreator; @*/
  /*@ private represents pcreator <- creator; @*/
  /** a description of the subject **/
  /*@ public model C pref; @*/
  private final C ref; /*@ in pref; @*/
  /*@ private represents pref <- ref; @*/

  /** the subject itself (null, if not yet created **/
  /*@ public model Subject<T,R> psubject; @*/
  /*@ nullable @*/ private Subject<T,R> vsubject; /*@ in psubject; @*/
  /*@ private represents psubject <- vsubject; @*/

  /***************************************************************************
   * the representation of the specification's subject
   **************************************************************************/
  /*@ private represents subject <- vsubject; @*/

  /***************************************************************************
   * Create a virtual proxy from an object creator and an object description.
   * The actual object creation is delayed until the proxy's request() method
   * is invoked.
   * @param creator of an object
   ***************************************************************************/
  /*@ public normal_behavior
    @ assignable pcreator, pref, psubject;
    @ ensures pcreator == creator && pref == ref && psubject == null;
    @*/
  public VirtualProxy(SubjectCreator<T,R,C> creator, C ref)
  {
    this.creator = creator;
    this.ref = ref;
    this.vsubject = null;
  }

  /***************************************************************************
   * An implementation of the proxy request (@see patterns.proxy.Proxy.request)
   * If the object has not yet been created, it is created now.
   * @param arg the request argument
   * @return result the request result
   ***************************************************************************/
  /*@ also public normal_behavior
    @ assignable \everything;
    @ ensures \old(psubject) == null ==>
    @           pcreator.calls == \old(pcreator.calls)+1 &&
    @           pref == pcreator.larg && psubject == pcreator.lresult;
    @*/
  public R request(T arg) throws Exception
  {
    if (vsubject == null) vsubject = creator.create(ref);
    return vsubject.request(arg);
  }
}

package patterns.proxy;

/*****************************************************************************
 * A factory for the creation of virtual proxies
 * (@see patterns.proxy.VirtualProxy).
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 * @param <C> the type of the object description (used for object creation).
 ****************************************************************************/
public class VirtualProxyFactory<T,R,C>
{
  /** the creator for this factory **/
  /*@ public model SubjectCreator<T,R,C> pcreator; @*/
  private SubjectCreator<T,R,C> creator; /*@ in pcreator; @*/
  /*@ private represents pcreator <- creator; @*/

  /***************************************************************************
   * Create a virtual proxy using the denoted object creator.
   * @param creator the creator that will be used for creating objects.
   ***************************************************************************/
  /*@ public normal_behavior
    @ assignable pcreator;
    @ ensures pcreator == creator;
    @*/
  public VirtualProxyFactory(SubjectCreator<T,R,C> creator)
  {
    this.creator = creator;
  }

  /***************************************************************************
   * Get a virtual proxy that allocate its objects with the denoted description.
   * @param ref the description of the object.
   * @return a proxy that creates its object with the denoted description.
   ***************************************************************************/
  /*@ public normal_behavior
    @ assignable \nothing;
    @ ensures \fresh(\result);
    @ ensures \result.pcreator == pcreator && \result.pref == ref;
    @*/
  public VirtualProxy<T,R,C> getProxy(C ref)
  {
    return new VirtualProxy<T,R,C>(creator, ref);
  }
}
```

A.3 Remote Proxy

```
package patterns.proxy;

/*@ model import java.util.HashMap; @*/

/****************************************************************************
 * The referer to an object for which a proxy may be supplied.
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 ****************************************************************************/
public interface SubjectReferer<T,R,C>
```

```
{
  /************************************************************************
   * a specification of the actual subjects behind this proxy.
   ************************************************************************/
  /*@ public instance model HashMap subjects; @*/ // HashMap<C,R>  not supported by JML to

  /** the number of calls of the request method **/
  /*@ public instance model int calls;
    @ public initially calls == 0;
    @*/

  /** the object reference in the last call of request **/
  /*@ public instance model C lref; @*/

  /** the argument in the last call of request **/
  /*@ public instance model C larg; @*/

  /** the result of last call of create **/
  /*@ public instance model C lresult; @*/

  /************************************************************************
   * Execute the request of an object identified by a reference.
   * @param ref the reference to the object.
   * @param arg the argument to the request.
   * @return the result of the request.
   * @throw Exception if the request could not be performed.
   ************************************************************************/
  /*@ public normal_behavior
    @ forall Subject<C,R> subject;
    @ requires subject == subjects.get(ref) && subject != null;
    @ assignable \everything;  // global state may be changed
    @
    @ // count call and remember its effects
    @ ensures calls == \old(calls)+1;
    @ ensures lref == ref && larg == arg && lresult == \result;
    @
    @ // delegate request to object
    @ ensures subject.calls == \old(subject.calls)+1;
    @ ensures subject.larg == arg && subject.lresult == \result;
    @
    @ // allow exception at any time (object may not have been called)
    @ also public exceptional_behavior
    @ forall Subject<C,R> subject;
    @ requires subject == subjects.get(ref) && subject != null;
    @ assignable \everything;
    @ signals (Exception e) subject.calls == \old(subject.calls) ||
    @                        subject.calls == \old(subject.calls)+1;
    @
    @ // also right object might not exist
    @ // (we ignore that it may also not have right type due to
    @ // lack of type parameters for HashMap)
    @ also public exceptional_behavior
    @ forall Subject<C,R> subject;
    @ requires subject == subjects.get(ref) && subject == null;
    @ assignable \nothing; // do not allow any state change then
    @ signals (Exception e) true;
    @*/
  public R request(C ref, T arg) throws Exception;
}

package patterns.proxy;

/*****************************************************************************
 * A remote proxy, i.e. a proxy (@see patterns.proxy.Proxy) which serves
 * as a local substitute for a proxy that can be only referenced by
 * an address-independent reference (i.e. a unique identifier).
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 * @param <C> the type of the object reference (used for object identification).
 ****************************************************************************/
public class RemoteProxy<T,R,C> implements Proxy<T,R>
{
  /** the mechanism by which the object is referenced **/
  /*@ public model SubjectReferer<T,R,C> preferer; @*/
  private final SubjectReferer<T,R,C> referer; /*@ in preferer; @*/
  /*@ private represents preferer <- referer; @*/

  /** the name by which the object is refererenced **/
  /*@ public model C pref; @*/
  private final C ref; /*@ in pref; @*/
  /*@ private represents pref <- ref; @*/

  // representation of the proxy object
  /*@ public represents subject <-
    @  (Subject<T,R>)(preferer.subjects.get(pref)); @*/

  /***************************************************************************
   * Creates a remote proxy.
   * @param referer the referer mechanism.
   * @param ref the unique name of the object.
   **************************************************************************/
  /*@ public normal_behavior
    @ assignable preferer, pref;
    @ ensures preferer == referer && pref == ref;
    @*/
  public RemoteProxy(SubjectReferer<T,R,C> referer, C ref)
  {
    this.referer = referer;
    this.ref = ref;
  }

  /***************************************************************************
   * An implementation of the proxy request (@see patterns.proxy.Proxy.request)
   * by invoking the referer mechanism.
   * @param arg the request argument
   * @return result the request result
   **************************************************************************/
  /*@ also public normal_behavior
    @ assignable \everything;
    @ ensures preferer.calls == \old(preferer.calls)+1;
    @ ensures pref == preferer.lref && arg == preferer.larg;
    @ ensures \result == preferer.lresult;
    @*/
  public R request(T arg) throws Exception
  {
    return referer.request(ref, arg);
  }
}

package patterns.proxy;

/***************************************************************************
 * A factory for the creation of remote proxies
 * (@see patterns.proxy.RemoteProxy).
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *
 * @param <T> the type of the argument of the object's request method.
 * @param <R> the type of the result of the object's request method.
 * @param <C> the type of the object reference (used for object lookup).
 **************************************************************************/
public class RemoteProxyFactory<T,R,C>
{
  /*@public model SubjectReferer<T,R,C> preferer; @*/
  private SubjectReferer<T,R,C> referer; /*@ in preferer; @*/
  /*@private represents preferer <- referer; @*/

  /*************************************************************************
   * Create a remote proxy using the denoted object referer
   * @param referer the reference mechanism for looking up objects.
   ************************************************************************/
  /*@ public normal_behavior
    @ assignable preferer;
    @ ensures preferer == referer;
    @*/
  public RemoteProxyFactory(SubjectReferer<T,R,C> referer)
  {
    this.referer = referer;
  }

  /*************************************************************************
   * Get a remote proxy that refers to its objects with the denoted referer.
   * @param ref the description of the object.
   * @return a proxy that refers to the object with the denoted description.
   ************************************************************************/
  /*@ public normal_behavior
    @ assignable \nothing;
    @ ensures \fresh(\result);
    @ ensures \result.preferer == preferer && \result.pref == ref;
    @*/
  public RemoteProxy<T,R,C> getProxy(C ref)
  {
    return new RemoteProxy<T,R,C>(referer, ref);
  }
}
```

A.4 Example Use

```
package patterns.proxy;
```

```
import java.util.*;

/******************************************************************************
 * A test of the implementation of the proxy pattern.
 * Passes jml type check except that there are errneous complaints of the form
 *
 *   The type of right-hand side of a represents clause, "java.lang.Integer",
 *   is not assignment-compatible to the type of left-hand side, "T" [JML]
 *
 * (i.e. type instantiation is not handled appropriately by JML).
 *
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.uni-linz.ac.at&gt;
 *****************************************************************************/
public final class Main
{
  public static void main(String[] args)
  {
    testVirtualProxy();
    testRemoteProxy();
  }

  private static void testVirtualProxy()
  {
    VirtualProxyFactory<Integer,Integer,Integer> cfactory =
      new VirtualProxyFactory<Integer,Integer,Integer>(new CounterCreator());
    System.out.println("creating proxy");
    Proxy<Integer,Integer> p = cfactory.getProxy(new Integer(3));
    System.out.println("proxy created");
    try{
    for (int i=0;i<3;i++) System.out.println(p.request(new Integer(2)));
    }
    catch(Exception e){ System.out.println(e);}
  }

  private static void testRemoteProxy()
  {
    CounterReferer creferer = new CounterReferer();
    creferer.add("a", 2);
    creferer.add("b", 3);
    RemoteProxyFactory<Integer,Integer,String> cfactory =
      new RemoteProxyFactory<Integer,Integer,String>(creferer);
    Proxy<Integer,Integer> p = cfactory.getProxy("a");
    Proxy<Integer,Integer> q = cfactory.getProxy("b");
    Proxy<Integer,Integer> r = cfactory.getProxy("c");
    try{
    for (int i=0;i<3;i++) System.out.println(p.request(new Integer(2)));
    for (int i=0;i<3;i++) System.out.println(q.request(new Integer(2)));
    for (int i=0;i<3;i++) System.out.println(r.request(new Integer(2)));
    }
    catch(Exception e){ System.out.println(e);}
  }

  private static class Counter implements Subject<Integer,Integer>
  {
    private int counter;
    // representation of the specification-only model fields
    /*@ private ghost int gcalls = 0;
      @ private represents calls <- gcalls;
      @
      @ private ghost Integer garg;
      @ private represents larg <- garg;
      @
      @ private ghost Integer gresult;
      @ private represents lresult <- gresult;
      @*/

    public Counter(int counter)
    {
      this.counter = counter;
    }
    public Integer request(Integer arg)
    {
      //@ set gcalls = gcalls+1;
      //@ set garg = arg;
      counter = counter + arg.intValue();
      Integer i = new Integer(counter);
      //@ set gresult = i;
      return i;
    }
    public int getCounter()
    {
      return counter;
    }
  }

  private static class CounterCreator implements
    SubjectCreator<Integer, Integer, Integer>
  {
    // representation of the specification-only model fields
    /*@ private ghost int gcalls = 0;
      @ private represents calls <- gcalls;
      @
      @ private ghost Integer garg;
      @ private represents larg <- garg;
      @
      @ private ghost Counter gresult;
      @ private represents lresult <- gresult;
      @*/

    public Counter create(Integer start)
    {
      //@set gcalls = gcalls+1;
      //@set garg = start;
      System.out.println("counter created");
      Counter c = new Counter(start.intValue());
      //@set gresult = c;
      return c;
    }
  }

  private static class CounterReferer implements
  SubjectReferer<Integer, Integer, String>
```

```
  {
    // HashMap<Integer,Integer> not supported by JML tools
    private static HashMap counters = new HashMap();
    /*@ private represents subjects <- counters; @*/

    // representation of the specification-only model fields
    /*@ private ghost int gcalls = 0;
      @ private represents calls <- gcalls;
      @
      @ private ghost String gref;
      @ private represents lref <- gref;
      @
      @ private ghost Integer garg;
      @ private represents larg <- garg;
      @
      @ private ghost Integer gresult;
      @ private represents lresult <- gresult;
      @*/

    public CounterReferer()
    {
    }

    public void add(String name, int init)
    {
      new CounterReferer(name, init);
    }

    private CounterReferer(String name, int init)
    {
      Counter c = new Counter(init);
      counters.put(name, c);
    }

    public Integer request(String ref, Integer arg) throws Exception
    {
      //@ set gcalls = gcalls+1;
      //@ set gref = ref;
      //@ set garg = arg;
      Counter c = (Counter)counters.get(ref);
      if (c == null) throw new Exception("no␣such␣counter:␣" + ref);
      Integer i = c.request(arg);
      //@ set gresult = i;
      return i;
    }
  }
}
```