23 Proxy

The proxy pattern I'll show you in this chapter is a structure pattern and has similarities to the facade and adapter. In all three patterns, you don't access an object directly, but through a third. Remember – the facade allowed you to deal with a complicated interface. With the adapter, the third object made it possible for two others to interact in the first place. And what will that be like with the proxy? With the proxy, you're also encapsulating access to an object. But why would you want to do that? Four reasons might be:

1. Virtual Proxy: You need a placeholder for a certain duration to create a large object.
2. Security Proxy: You want to block or at least control access to the object.
3. Smart Reference: You want to add functionality to the object; for example, you want to check whether the object is locked before actually accessing it.
4. Remote Proxy: The desired object is not in the same address space.

These four application areas are the most common; they are discussed by the GoF. However, you will find other proxies in the literature: the firewall proxy, the synchronization proxy, and many more. This chapter is one of the longer ones in the book. So, get comfortable, and let's get started.

23.1 Virtual Proxy

The Virtual Proxy should only be addressed here theoretically for the sake of completeness. Imagine you create a software with which photos can be displayed. For example, there should be 100 pictures in a certain folder. With the chosen setting 25 pictures fit on one screen page. When you open the folder, the first thing to do is to register all the images,

© The Author(s), under exclusive license to Springer Fachmedien Wiesbaden GmbH, part of Springer Nature 2023
O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_23

determine their size, create the previews and draw them. If the photos are quite large, this process may well take a while. Now it would be very unsatisfactory for the user if your program freezes. It is much more comfortable if the preview is generated only from the pictures that are currently displayed on the screen. At first the screen shows 25 frames as placeholders for the later previews. Gradually, these placeholders, the proxies, are replaced by the actual preview images. You proceed accordingly with the following screens. You create frames as placeholders and when the user scrolls the frames into the visible area of the screen, the previews are created.

23.2 Security Proxy

It can be useful to prevent write access to an object. As an example, consider the `unmodifiableList()` method of the `Collections` class in the Java class library. You pass a List object to the method and get back a read-only list. The method wraps your list in an object of the `UnmodifiableList` inner class. The UnmodifiableList extends the `UnmodifiableCollection` class, which is also an inner class. Now, when you want to access the actual database, you don't access the original list, you access the UnmodifiableList that surrounds it. The UnmodifiableList forwards requests to the original database – write accesses excluded. Take a look at the Unmodifiable sample project; there I extracted the relevant code portions of the Collections class from the class library into the file unmodifiable.java. This file is by no means complete in this excerpt and will therefore not compile. But that's not what it's intended for, it's just a reference for you. You can of course alternatively look directly in the source code of the class library, if you have also downloaded or linked it.

```
class UnmodifiableList<E>
                        extends UnmodifiableCollection<E>
    implements List<E> {
        final List<? extends E> list;
        UnmodifiableList(List<? extends E> list) {
            super(list);
            this.list = list;
        }

        @Override
        public E get(int index) {
            return list.get(index);
        }

        @Override
```

```
            public E set(int index, E element) {
                throw new UnsupportedOperationException();
            }
        // … abridged
    }
```

For the test, a client creates an ArrayList with various strings. Then he lets the `Collections` class return a read-only list. This list is then no longer an instance of the `ArrayList` class. Rather, it is an object of the `UnmodifiableRandomAccessList` class. As expected, an `UnsupportedOperationException` is thrown when trying to add another string.

```
List<String> text = new ArrayList<>();
System.out.println(text.getClass().getName());
text.add("alpha");
text.add("bravo");
text.add("charly");
text.add("delta");
text.add("echo");
text = Collections.unmodifiableList(text);
System.out.println(text.getClass().getName());
Thread.sleep(2);
text.add("foxtrot");
```

The console outputs:

```
java.util.ArrayList
java.util.Collections$UnmodifiableRandomAccessList
Exception in thread "main"
                java.lang.UnsupportedOperationException
at java.base/java.util.Collections$
        UnmodifiableCollection.add(Collections.java:1060)
at Test.main(Test.java:43)
 C:\...\run.xml:111: The following error occurred while executing
this line:
C:\...\run.xml:68: Java returned: 1
```

The UnmodifiableRandomAccessList is a subclass of the UnmodifiableList that is used to improve performance. The documentation in the Collections library says: “Many of the List algorithms have two implementations, one of which is appropriate for RandomAccess lists, the other for “sequential”. Often, the random-access variant yields better performance on small sequential access lists.” However, the principle does not change: The client only accesses the actual target object via the proxy.

23.3 Smart Reference

The Smart Reference adds more functionality to the represented class, for example to control it. In the first step, I'll show you a version that has nothing to do with the proxy (yet). It should only introduce the project to you.

23.3.1 The Basic Version

The example in this section goes back to the Adapter Pattern example. You can find it in the sample project Proxy_1: There is a third-party library that provides an algorithm that is quite unique. The super-strict-secret algorithm sorts a list in a highly efficient way.

```
public final class SorterExternalProduct {
    List<Integer> sort(List<Integer> numberList) {
        // … abridged
    }
}
```

However, your client internally works with an int array. To allow the client to work with the foreign library, an intermediate object provides a method to convert an int array to an integer list. The modified database can be passed to the foreign library for sorting. Finally, the foreign library provides a method to convert the sorted list back to an array. The interface Sorter declares these methods.

```
public interface Sorter {
    List<Integer> convertToList(int[] numbers);
    int[] convertToArray(List<Integer> numberList);
    List<Integer> sort(List<Integer> numberList);
}
```

A helper class implements these three methods.

```
public class SorterHelper implements Sorter {
    private final SorterOtherProduct otherProduct =
                        new SorterForeignProduct();

    @Override
    public List<Integer> convertToList(int... numbers) {
        // … abridged
    }

    @Override
    public int[] convertToArray(List<Integer> numberList) {
        // … abridged
    }

    @Override
    public List<Integer> sort(List<Integer> numberList) {
        // … abridged
    }
}
```

The client defines a certain number of unsorted random numbers with the method `createArray()`. Then it creates an instance of the class `SorterHelper`, lets it convert the array with the database into a list, sorts this list, converts it back into an array and processes the array further.

```
public static void main(String[] args) {
    var number = 100;
    var numbers = new int[number];
    numbers = createArray(numbers);
    var sorter = new SorterHelper();
    var convertedList = sorter.convertToList(numbers);
    var sortedList = sorter.sort(convertedList);
    var sortedArray = sorter.convertToArray(sortedList);

    for (var i : sortedArray)
        System.out.println(i);
}
```

The user asks whether it is worthwhile to convert the client software so that it works internally with a list. This would eliminate the need for conversion. You want to find out if the time it takes to convert the data is significant enough to make refactoring worthwhile. To do this, you measure time for conversions.

23.3.2 Introduction of a Proxy

To perform timing, develop a class that implements the same methods as the original `SorterHelper` class. The proxy can reference an object of type Sorter. The sort() method is used as an example to describe the procedure. The method first stores the current timestamp, executes the actual method on the sorter instance, and then measures the time again. The difference between the two timestamps is determined and output formatted on the console. You can find this code in the sample project Proxy_2.

```
public class SorterTimeProxy implements Sorter {
    private final sorter sorter;
    private final SimpleDateFormat dateFormat =
                  new SimpleDateFormat("HH:mm:ss:SSS");

    SorterTimeProxy(Sorter sorter) {
        this.sorter = sorter;
    }

    @Override
    public List<Integer> convertToList(int... numbers) {
        // … abridged
    }

    @Override
   public int[] convertToArray(List<Integer> numberList){
        // … abridged
    }

    @Override
    public List<Integer> sort(List<Integer> numberList) {
        var start = Instant.now();
        numberList = sorter.sort(numberList);
        var end = Instant.now();
        var duration = Duration.between(start, end);
        var time = duration.toMillis();
        System.out.println("sort: " +
                                dateFormat.format(time));
        return numberList;
    }
}
```

The sort method can be divided into four steps:

1. Code before the call – Pre-Invoke
2. Calling the actual method – Invoke
3. Code after the call – Post-Invoke
4. Return of the result

As in the previous project, the client creates an array of unsorted random numbers. It then creates an instance of the classes `SorterHelper` and `SorterTimeProxy`. You pass the instance of the SorterHelper class to the constructor of the `SorterProxy` class. You pass the commands for converting and sorting the database to the proxy.

```
public static void main(String[] args) {
    var number = 5000000;
    var numbers = new int[number];
    numbers = createArray(numbers);
    var sorter = new SorterHelper();
    var sorterTimeProxy = new SorterTimeProxy(sorter);
    var convertedList =
                sorterTimeProxy.convertToList(numbers);
    var sortedList = sorterTimeProxy.sort(convertedList);
    var sortedArray =
            sorterTimeProxy.convertToArray(sortedList);
}
```

Each method that is called executes its own behavior on the one hand, but also triggers the desired behavior of the target object on the other hand. Through the upstream proxy you measure the time and get the sorted database. The following text is output on the console.

```
convertToList: 01:00:00:137
sort: 01:00:01:979
convertToArray: 01:00:00:087
```

In the next section, we extend this approach even further.

23.3.3 Introducing a Second Proxy

Your client now wishes to be able to log the algorithm. So, you need to write a second proxy. And now it becomes clear why it makes sense to have the proxy implement the same interface as the SorterHelper?

You already sense that the new proxy, which you find in the sample project Proxy_3, also has to take the role Sorter. And just like the old proxy, it also holds a reference to another sorter object. Using the `sort()` method as an example, we will show that the proxy first performs its own task – logging – and then calls the desired method of the target object.

```
public class SorterLogProxy implements Sorter {
    private final sorter sorter;

    SorterLogProxy(Sorter sorter) {
        this.sorter = sorter;
    }

    @Override
    public List<Integer> convertToList(int... numbers) {
        // … abridged
    }

    @Override
   public int[] convertToArray(List<Integer> numberList){
        // … abridged
    }

    @Override
    public List<Integer> sort(List<Integer> numberList) {
        // Pre-Invoke
        var strNumber = Long.toString(numberList.size());
        print("Sorts a list of " + strNumber +
                                       " numbers");
        // Invoke and return
        return sorter.sort(numberList);
    }

    private void print(String message) {
        System.out.println("\tLog-Level INFO: " +
                                           message);
    }
}
```

The client determines which class is passed to the constructor – this can either be an instance of the class SorterHelper or an instance of the class SorterTimeProxy. So, the client determines whether it needs one, none or multiple proxies.

```
public static void main(String[] args) {
    var number = 1000000;
    var numbers = new int[number];
    numbers = createArray(numbers);
    var sorter = new SorterHelper();
    var sorterLogProxy = new SorterLogProxy(sorter);
    var sorterTimeProxy =
                    new SorterTimeProxy(sorterLogProxy);
    var convertedList =
                sorterTimeProxy.convertToList(numbers);
    var sortedList = sorterTimeProxy.sort(convertedList);
    var sortedArray =
            sorterTimeProxy.convertToArray(sortedList);
}
```

When you run the code, the following text is output to the console:

```
   Log-Level INFO: Convert an array with 1000000 numbers.
convertToList: 00:00:00:033
   Log-Level INFO: Sort a list with 1000000 numbers.
sort: 00:00:00:368
   Log-Level INFO: Convert a list with 1000000 numbers.
convertToArray: 00:00:00:024
```

The client can switch the proxy before the actual call of the target object and thus control or direct the access.

23.3.4 Dynamic Proxy

When you analyze the code of the proxy classes, you notice that the pre-invoke and post-invoke in each method are the same or very similar. Code duplicates are usually a sign of poor design. There is another drawback to this: The proxy classes fully implement the interfaces of the target classes. If you insert a new method in the interface Sorter, all proxy classes must be adapted and implement this new method.

The goal of this section is to use Reflection to isolate common pieces of code and have the proxy classes automatically get generated. In other words, extract the behavior and let it generate the proxy classes at runtime. Sounds crazy? It is! Let's take it one step at a time.

23.3.4.1 The Reflection API

I can't give a comprehensive introduction to the Reflection API here. My notes on this are limited to what is needed for Dynamic Proxy.

As a programmer, one often has an interest in examining objects and classes at runtime. The Java language includes the `instanceof` operator, which checks whether an object is (also) of the type of the designated class. The call `new Student() instanceof human` returns `true` if the class Student inherits from human.

The other tools of the Reflection API are found in the `java.lang` and `java.lang.reflect` packages, primarily in the `Class`, `Method`, and `Field` classes. With the call `Class.forName()`, you load the class at runtime whose fully qualified class name you pass to the method as a parameter. You get a Class object when you call the `getClass()` method on an object. The Class object describes the class to which it refers from a meta view. For example, you can use it to query the declared methods. In the main method of the client in the sample project ProxyDynamic, I have excerpted the following code:

```
var sorter = new SorterHelper();
var sorterClass = sorter.getClass();
var methods = sorterClass.getMethods();
for (var tempMethod : methods)
    System.out.println(tempMethod.getName());
```

All methods are queried from the Class object and their names are output to the console:

```
sort
convertToArray
convertToList
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
```

The class `SorterHelper` has no superclass, so it inherits automatically from the class `Object`, in which the last new methods are declared public, each with different parameters. The first three methods are prescribed in the interface `Sorter` and implemented by `SorterHelper.` The call `sorterClass.getClassLoader()` returns an object of type `ClassLoader`. The class loader is responsible for loading a class into memory.

23.3.4.2 The InvocationHandler

For the Dynamic Proxy you need two components: the InvocationHandler and the Proxy. The InvocationHandler describes the behavior of all methods; the Proxy class is created dynamically and has the same task as the Proxy from the previous project. This section discusses the InvocationHandler; it describes in general terms how the methods should behave. To generate the InvocationHandler, a class must implement the `InvocationHandler` interface and define the `invoke()` method. The proxy object, the method to invoke, and an array of parameters to the method to invoke are passed to this method. In addition to `invoke(),` the InvocationHandler can define its own behavior and data fields. For example, the InvocationHandler in the sample ProxyDynamic project holds a reference to the Sorter object whose methods are invoked. In addition, the InvocationHandler stores how many times the foreign library has been called. Probably the most important call within the method is the `invoke()` method on the `Method method` parameter. You pass this method the target object on which you want the method to run. You also pass the parameters to the method. In the example, you can simply pass the parameters. You also simply pass on the return value of the method of the target object.

```
public class TimeHandler implements InvocationHandler {
    // … abridged

    private final Object object;
```

```
        private static int calls = 0;

        // … abridged

        public static int getCalls() {
            return calls;
        }

        @Override
        public Object invoke(Object proxy, Method method,
                             Object[] args) throws Throwable {
            var start = Instant.now();
            var result = method.invoke(object, args);
            var end = Instant.now();
            var duration = Duration.between(start, end);
            var millis = duration.toMillis();
            var methodName = method.getName();
            System.out.println(methodName + ": " +
                                 dateFormat.format(millis));
            if (methodName.equals("sort"))
                calls++;

            return result;
        }
    }
```

What's missing now is the actual proxy class – and that's generated at runtime, which I'll show in the following section.

23.3.4.3 The Proxy Class

It is important to note that with Dynamic Proxy, the proxy class is not only **loaded** dynamically at runtime, but it is actually **defined** at runtime. In the previous section, you described the behavior of the methods. Now the actual proxy class is to be created. The code for the proxy class is defined dynamically at runtime. To do this, call the static method `newProxyInstance()` of the class Proxy. Pass it the class loader object of the target interface, an array of target interfaces, and the InvocationHandler. Cast the return value of this method to the `Sorter` interface and pass it to a variable of type `Sorter`. This variable now references an instance of the proxy class created at runtime.

```
    var sorter = new SorterHelper();
    var timeHandler = new TimeHandler(sorter);

    var targetClasses = new Class[] {Sorter.class};
```

```
var sorterProxy = (Sorter) Proxy.newProxyInstance(
            Sorter.class.getClassLoader(), targetClasses,
                                            timeHandler);
```

That's it! When you run the client's main method, your program behaves exactly as if you had generated the proxy yourself.

If you've never worked with the Reflection API before, you may find the Dynamic Proxy a bit confusing. Perhaps the best way to understand the principle is to realize that the InvocationHandler in the `invoke()` method describes the behavior of all methods in the target interface. The proxy, which is the instance of the class that is first created at runtime, determines all methods from the interface. Each method is equipped with the same behavior, namely a forwarding of the request to the InvocationHandler: `handler.invoke()`; Analyze the sample project. Be sure to also read the API documentation for the Proxy class and the InvocationHandler interface. There you will find important information about working with Dynamic Proxy.

You can also find the sample project ProxyDynamicTable. In this project the result array is displayed in a table. For this purpose, I have inserted the lines printed here, among others, in the main method of the Client class:

```
var tblModel = new TableModel() {
    @Override
    public int getRowCount() {
        // Methods of the TableModel Interface
    }
};

var handler = new TimeHandler(tblModel);
var gateway = new Class[] {
    TableModel.class
};

var tableModelProxy = (TableModel)
        Proxy.newProxyInstance(
            TableModel.class.getClassLoader(),
            gateway, handler);
tblNumbers.setModel(tableModelProxy);
```

On the console, all accesses to the methods of the `TableModel` interface are now logged and evaluated. To produce measurable results, each method call is randomly slowed down. What I want to show you with this is that the InvocationHandler can be reused. It is only bound to an interface by the dynamically created proxy class.

23.4 Remote Proxy

Each Java program is executed in its own virtual machine. In this VM, instances of all required classes are created. These objects swim around in the VM like fish in an aquarium. When an object needs information from another object or wants to use its services, it sends a message to that object. Just like a fish in an aquarium can only communicate with a fish from the same aquarium, objects only know about other objects on the same VM. With RMI, you have a way to look beyond the edge of the aquarium.

23.4.1 Structure of RMI in Principle

In the project I will present in this section, you will program a client that wants to know what value Pi has. There is a server that knows this value and announces it on request. This project is a bit simplified and does not cover all aspects of the technology behind it, RMI. But it shows the relation to the proxy pattern.

A client does not communicate directly with the server, but through a proxy, called a stub on the client side. To the client, it looks like the proxy is the server object. In fact, however, the proxy only pretends to be the server object; it packages the client's request into a data stream and sends it to the server. The server unpacks the request and sends it to the actual object. The way it works in reverse is the same. The server object sends its response to a proxy, called a skeleton on the server side. The skeleton packages the response into a data stream and sends it back to the client over the network. The data stream relies on serialization; therefore, all parameters and return values must be serializable or a primitive data type.

The following project recreates this proxy access on a small level. You can find the code in the subdirectory RMI_Test, and this time for once not as a NetBeans project. How to get the whole thing working, I explain below.

23.4.2 The RMI Server

Let us first program an RMI server. The server is to consist of two classes and one interface. The `PiImpl` class provides three methods. The `getState()` method returns that the instance is alive. The return value of the other method – method `getPi()` - is the number Pi. Finally, the `getCounter()` method tells us how many times the `getPi()`

method has been called. The `PiIF` interface declares the `getPi()` and `getCounter()` methods, which are implemented by `PiImpl`. The class `ServerStart` creates an object of the class `PiImpl`, announces it as RMIServer and queries the state of the server in an endless loop.

23.4.2.1 The PiIF Interface

It is best to start by looking at the `PiIF` interface. The interface specifies which methods can be called by an RMI client; it extends the `Remote` interface. `Remote` does not prescribe any methods by itself, but signals that all interfaces derived from it can be invoked remotely; thus, `Remote` is purely a marker interface. If the communication between client and server does not work, a RemoteException is thrown. Since this can happen in any method, each method must declare a RemoteException.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface PiIF extends Remote {
    public int getZaehler() throws RemoteException;
    public double getPi() throws RemoteException;
}
```

The class `PiImpl` implements the interface `PiIF`.

23.4.2.2 The PiImpl Server Class

The server class `PiImpl` implements `PiIF` and defines the mandatory methods `getZaehler()` and `getPi()`. In addition, the method `getState()` is defined.

```
public class PiImpl implements PiIF {
    private int counter =0;

    @Override
    public int getCounter() {
        return counter;
    }

    @Override
    public double getPi() {
        counter++;
        return 3.1415…;
    }

    public String getState() {
        return "Server is alive";
    }
}
```

Please note that only objects within the same VM can call the `getState()` method. The methods `getCounter()` and `getPi()` can also be called by objects outside their own VM – the interface Remote ensures this.

23.4.2.3 The Class ServerStart Starts the Server

The main method of the ServerStart class first creates an instance of the PiImpl class. The static methodUnicastRemoteObject.exportObject(pi) is then called so that the object can be accessed remotely at all. This instance is then registered with the RMI registry – also: “bound”. The registry is the register of the server VM where the client can request the stub. In the example, the instance is registered under the name “PiCalculator”. Instead of rebind() you could have coded bind() – the instance could then have been entered only once.

```
public static void main(String[] args) {
    var pi = new PiImpl();
    try {
        var remote = UnicastRemoteObject.exportObject(pi,
                                                       8077);
        Naming.rebind(“PiCalculator“, remote);
    } catch (RemoteException | MalformedURLException ex){
        ex.printStackTrace();
    }

    // … abridged
}
```

Finally, the main method enters an infinite loop that queries every 5 s to see if the server is still active.

```
public static void main(String[] args) {
    // … abridged
    while (true) {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ex) {
            // … abridged
        }
        var state = pi.getState();
        System.out.println(state);
    }
}
```

This already describes the server classes. Let’s take a look at what the client looks like.

23.4.3 The RMI Client

The start class creates an instance of the Calculator class and queries the number Pi there; the returned value is output to the console. Afterwards, the calculator is instructed to output to the console how often Pi has already been queried.

```
public class ClientStart {
    public static void main(String[] args) {
        var calculator = new CalculationEngine();
        System.out.println(“Pi is: “ +
                                calculator.tellMePi());
        calculator.printCounter();
    }
}
```

From the RMI point of view, the constructor of the calculator is interesting. It calls the `lookup()` method, which returns a reference to the stub object from the registry. `Naming.lookup()` is something like looking into the server's registry. Returned is the stub of the service, which is cast to the correct interface.

```
public class CalculationEngine {
    private final PiIF pi;

    CalculationEngine() {
        pi = lookup();
    }

    private PiIF lookup() {
        PiIF result = null;
        var serverName = “localhost“;
        var serviceName = “PiCalculator“;
        try {
            result =
                 (PiIF) Naming.lookup(“rmi://“ +
                        serverName + “/“ + serviceName);
        } catch (NotBoundException |
                MalformedURLException |
                RemoteException ex) {
            ex.printStackTrace();
        } finally {
            return result;
        }
    }
    // … abridged
}
```

On the stub, the client can now call the designated methods as if they were local.

```
public class CalculationEngine {
    // … abridged

    public double tellMePi() {
        double result = 0;
        try {
            result = pi.getPi();
        } catch (RemoteException ex) {
            ex.printStackTrace();
        } finally {
            return result;
        }
    }
}
```

You now know all the code – let's look at how to make the project work in the following paragraph.

23.4.4 Making the Project Work

You will find all java files in the subdirectory RMI_Test as mentioned before. Copy them locally to your computer. Open a console window and change to this directory. Translate the source files with `javac *.java.` Now call the RMI registry: `rmiregistry`. Please wait a few seconds and if necessary confirm a Windows share request dialog for network access. Now you can start the server in a new prompt with `java ServerStart.` Again, wait (and confirm another network share request if necessary) until you see the message "Server is alive" in the output, which is repeated every few seconds. If you now call the client from a third console with `java ClientStart`, the value of Pi and the number of answered requests for Pi since the last server start will be printed on the console.

If you start the server too soon after calling rmiregistry, you will most likely get an exception at first, but this will be followed by the "server is alive" messages after communication is established. On the other hand, if you start the client too early, it will abort with an exception and a value of 0.0 for Pi. You will then have to restart it and should get the correct information from the server.

Both the client and the server object communicate with their proxies – stub and skeleton – without knowing it. The proxy objects are responsible for transporting the data streams across the network to the other virtual machine.

The sequence of communication is shown in the sequence diagram in Fig. 23.1.

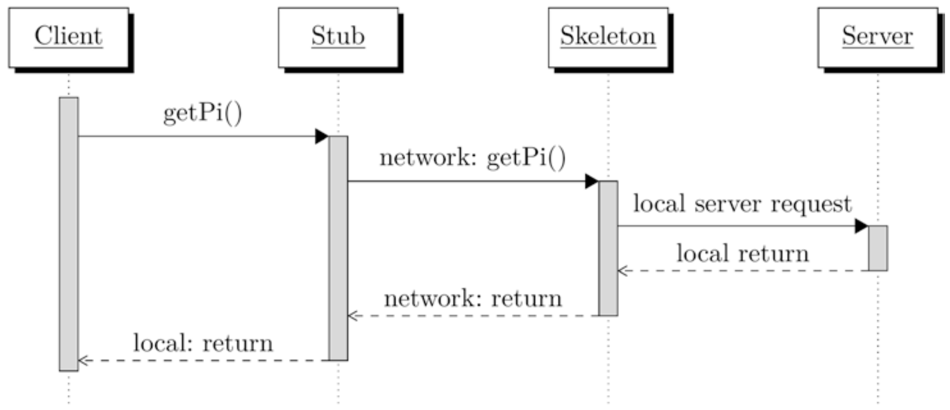


Fig. 23.1 RMI – Communication via upstream proxy

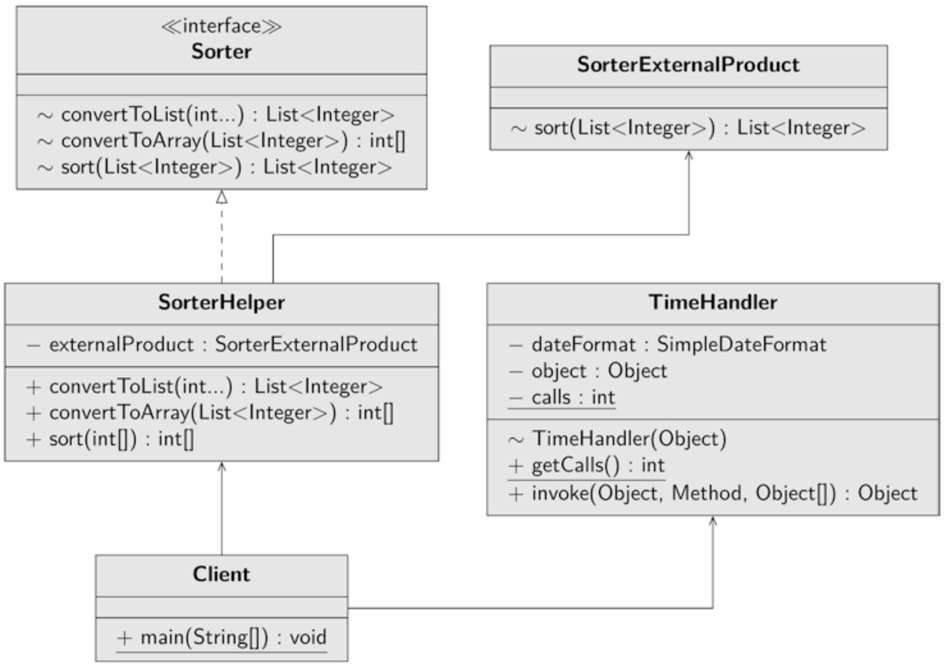
23.5 Proxy – The UML Diagram

Figure 23.2 shows the UML diagram from the sample project ProxyDynamicTable.

23.6 Summary

Go through the chapter again in key words:

- With a proxy, you do not access an object directly, but via its proxy.
- The main application areas are Remote Proxy, Security Proxy, Smart Reference and Virtual Proxy; however, there are other areas as well.
- Virtual Proxy: You do not load all images in an album, but only those that are currently to be displayed on the screen.
- Security Proxy: The actual target object is encapsulated in another object that – as in the UnmodifiableList example – prevents write access.
- Smart Reference: The function of an object is extended with the motivation to log or extend accesses.
  - The proxy implements the interface of the target object.
  - It is placed in front of the actual object.
  - The client calls the desired method.
  - The call is first processed by the proxy.
  - The call is then forwarded to the actual target object.
  - The procedure is inflexible when the interface is extended.
  - Dynamic Proxy is a relief for the programmer since Java 5:
  - The InvocationHandler defines the behavior of all methods of an interface in a general way.
  - The proxy class is created dynamically at runtime.



≪interface≫
Sorter
~ convertToList(int...) : List<Integer>
~ convertToArray(List<Integer>) : int[]
~ sort(List<Integer>) : List<Integer>
SorterExternalProduct
~ sort(List<Integer>) : List<Integer>
SorterHelper
− externalProduct : SorterExternalProduct
+ convertToList(int...) : List<Integer>
+ convertToArray(List<Integer>) : int[]
+ sort(int[]) : int[]
TimeHandler
− dateFormat : SimpleDateFormat
− object : Object
− calls : int
~ TimeHandler(Object)
+ getCalls() : int
+ invoke(Object, Method, Object[]) : Object
Client
+ main(String[]) : void

Fig. 23.2 UML diagram of the proxy pattern (example project ProxyDynamicTable)

- Remote Proxy: allows access to objects outside the own address space
  - A client wants to access a server object outside its own address space.
  - The client gets a stub from the server to which it sends its request.
  - The stub – the client proxy – forwards the request over the network to the server.
  - The server proxy – the skeleton – passes the request to the server object.
  - The server object sends the return value to the skeleton.
  - The skeleton sends the response to the stub, which sends the response to the requesting local object.
  - Neither client nor server object knows that they are sending their messages to proxies.

23.7 Description of Purpose

The Gang of Four describes the purpose of the “Proxy” pattern as follows:

> Control access to an object using an upstream proxy object.