Proxy Pattern: Defined

- The *Proxy* Pattern:
  
  *Provides a surrogate or placeholder for another object to control access to it.*

- The proxy is a go-between
  
  – In this respect, it resembles the *Adapter* pattern
  
  – However, its job is not to enable one interface to communicate with a different one
  
  – Rather, it simply is a middleman that facilitates communication between two sets of objects using the same interface

- Class diagram:

  ![Class Diagram](image)

  - *Proxy* is the client’s intermediary between it and *RealSubject*

  - The *Subject* interface allows the client to interact with *Proxy* as though it were *RealSubject*

  - The proxy’s instance variable allows it to forward requests to the real subject
Proxy Pattern: Defined (2)

• There are several types of proxy patterns:

  1. **Remote Proxy**: Provides access to remote subjects.

  2. **Virtual Proxy**: Creates and acts as a temporary stand in for a subject that is expensive to create. Once the subject is created, the proxy relinquishes control to the subject.

  3. **Firewall Proxy**: Controls access to a set of network resources, protecting the subject from bad clients.

  4. **Smart Reference Proxy**: Provides additional actions whenever a subject is referenced; e.g., counting the number of references to an object.

  5. **Caching Proxy**: Provides temporary storage for results of operations that are expensive. Can allow multiple clients to share these results to reduce computation or network latency.

  6. **Synchronization Proxy**: Provides safe access to a subject from multiple threads.

  7. **Complexity Hiding Proxy**: Hides complexity and controls access to a complex set of classes. Sometimes called *Facade Proxy* for obvious reasons. Differs from true *Facade* in that this also controls access.

  8. **Copy-On-Write Proxy**: Controls copying of an object by deferring the copying until it is required by a client. This is a variant of *Virtual Proxy*.

• We address *Remote* and *Virtual Proxys*
Proxy Pattern: Remote Proxy - Motivation

- Consider the following scenario:
  - You have existing code for performing some task WRT some set of objects
    ```java
    public class Subject {
        private type1 instance1;
        private type2 instance2;
        ...  
        public Subject (type1 x, type2 y) {
            ...
        }  
        public ... method1 (...) {
            ...
        }  
        public MyClass method2 (...) {
            ...
        }  
    }
    
    public class Client {
        private Subject subject;
        public Client (Subject subject) {
            this.subject = subject;
        }
        public ... request1 (...) {
            subject.method1();
            ...
        }
        public ... request2 (...) {
            MyClass x;
            x = subject.method2();
            ...
        }
    }
    
    public class Driver {
        public static void main (String[] args) {
            Subject testSubject = new Subject(arg1, arg2);
            Client client = new Client(testSubject);
            client.request1(...);
            client.request2(...);
        }
    }
    ```
Proxy Pattern: Remote Proxy - Motivation (2)

- A situation arises in which you need to perform this task on remote objects
  * I.e., \textit{Remote} means that objects are not in the same address space (usually not even on the same machine)
- Do not want to change existing code

- \textit{Remote Proxy} Pattern enables communication with minimal modification to existing code
  - The \textit{Remote Proxy} acts as a local representative to a remote object
  - The client call methods of the proxy
  - The proxy forwards the calls to the remote object
  - To the client, it appears as though it is communicating directly with the remote object

- Flow of control:

  ![Diagram](image.png)

- In Java, remote communication implemented using \textit{Remote Method Invocation} (RMI)
Proxy Pattern: Remote Proxy - RMI Overview

- Discussion is in terms of *client side* (local) and *server side* (remote)
- Client side consists of a client and client helper
  - The client-side helper is the proxy
  - Helper has no logic - simply packs up the request and forwards it to the server side and waits for a response
  - On receiving a response, the helper unpacks it forwards it to the client
- Server side consists of a service object and service helper
  - Helper receives requests and unpacks them, invoking them on the service object
  - Service object does actual processing
  - Helper receives results from service object, packs them up, and forwards them to client helper
- Flow of control:

  ![Diagram of RMI flow](image)

- RMI creates helpers for you
  - Client helper called a *stub* - this is the proxy
  - Service helper called a *skeleton*
Proxy Pattern: Remote Proxy - RMI Service Side Implementation

• Steps required to set up a remote service using RMI:

1. Create a remote interface
   – This step specifies which methods can be called remotely by the client
   – Implemented in terms of the stub and skeleton
   (a) Extend java.rmi.Remote
      – Remote is a marker interface - it has no methods

      import java.rmi.*

      public interface MyRemote extends Remote { ...

   (b) Declare that all accessible methods throw RemoteException exception
      – Required because network communication is prone to problems

      import java.rmi.*

      public interface MyRemote extends Remote {
        public ... myMethod(...) throws RemoteException;
        ...
      }

   (c) Make sure that arguments and return values are primitives or Serializable
      – Java built-ins are already serializable
      – Any data types defined by programmer that will be communicated
        must implement Serializable
      – Serializable found in java.io
      – If have a member in a serialized class that won’t be accessed, declare
        that member to be transient
        * This precludes serialization of that member when the transfer takes
        place

        transient MyClass class;

2. Create a Remote implementation
   – This is the service object
   – This is the actual object the client wants to communicate with (not a helper)
Proxy Pattern: Remote Proxy - RMI Service Side Implementation (2)

(a) Implement the *Remote* interface
   - I.e., the one in which the callable methods have been declared

```java
import java.rmi.*
import java.rmi.server.*

public class MyRemoteImplementation
    extends UnicastRemoteObject implements MyRemote {

    public ... myMethod(...) {
        ...
        return ...;
    }

    ...
}
```

(b) Extend *UnicastRemoteObject*  
   - Provides functionality required for communicating with remote service objects (see above code snippet)

(c) Create a constructor with no arguments that declares a remote exception  
   - This is required because the superclass (*UnicastRemoteObject*) throws this exception

```java
public MyRemoteImplementation throws RemoteException{}  
```

(d) Register the service with the RMI registry  
   - Once registered, remote services become available to the client  
   - The registry must be executing when registration takes place (See step 4)  
   - The stub is what is stored in the registry  
   - Registration performed using the *rebind()* method

```java
try {
    MyRemote service = new MyRemoteImplementation();
    Naming.rebind("myMethod", service);
} catch (Exception ex) {...}
```

- First argument is a *String* used by clients to access the service  
- Second argument is the actual service object
Proxy Pattern: Remote Proxy - RMI Service Side Implementation (3)

3. Generate stubs and skeletons
   (a) Execute `rmic` on the remote implementation class (in a terminal window)

   ```
   % rmic MyRemoteImplementation
   ```
   
   – Creates two new classes:
   i. a stub class
   ii. a skeleton class
   – Classes created in current working directory
   – Class names consist of name of remote implementation and `_Stub` or `_Skel`:
     i. `MyRemoteImplementation_Stub`
     ii. `MyRemoteImplementation_Skel`

4. Start the registry by executing `rmiregistry` (in a terminal window)
   – This program must be able to see the classes
   – The registry is an index to available proxies

5. Start the service
Proxy Pattern: Remote Proxy - RMI Service Side Implementation (4)

• Server side sample code (consolidated)

    //Remote interface

    import java.rmi.*;

    public interface MyRemote extends Remote {
        public ... myMethod(...) throws RemoteException;
        ...
    }

    //Remote service

    import java.rmi.*;
    import java.rmi.server.*;

    public class MyRemoteImplementation extends UnicastRemoteObject implements MyRemote {
        public ... myMethod(...) {
            ...
            return ...;
        }

        public MyRemoteImplementation throws RemoteException{};

        public static void main(String[] args) {

            try {
                MyRemote service = new MyRemoteImplementation();
                Naming.rebind("myMethod", service);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
Proxy Pattern: Remote Proxy - RMI Client Side Implementation

- Steps required by a client to access a remote service using RMI:
  1. Client gets the appropriate proxy by performing a registry lookup

```java
import java.rmi.*;

public class MyRemoteClient {

    public static void main(String[] args) {
        new MyRemoteClient().go();
    }

    public void go() {
        try {
            MyRemote service =
                (MyRemote) Naming.lookup("rmi://<host or IP address>/myService");

            // myService is the name that the service was registered under - see above discussion, step 2

            service.myMethod(...);
            ...
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

2. Client invokes a method on the stub returned by the lookup
   - To the client, the stub is the real object it wishes to communicate with on the service side

- Client side sample code
Proxy Pattern: RMI Server and Client Side Components

- Server side
  1. Service object
  2. Skeleton
  3. Stub
  4. Remote interface

- Client side
  1. Client object
  2. Stub
  3. Remote interface

- Note that the client never directly refers to the stub in its code
  - Rather, it communicates through the remote interface

- The server side needs the stub class because it is substituted for the real service when the service is bound to the RMI registry

- Client must have the stub and the classes for serialized objects that are to be returned by the service

- Three ways to get these:
  1. Hand deliver them (as in text’s examples)
  2. Use dynamic class downloading
  3.
Proxy Pattern: Remote Proxy - Example

• To convert existing code for use with a proxy:
  
  – Server side (changes to original code being accessed by client)
    1. Create a remote interface for use with the proxy
    2. Make any user-defined parameter types and return types *Serializable*
    3. Implement the interface in a concrete class
  
  – Client side (changes to original client code)
    1. Include rmi
    2. Reference remote interface instead of concrete remote objects
    3. Catch remote exceptions
Proxy Pattern: Remote Proxy - Example (2)

• Sample code:

```java
//Remote interface
import java.rmi.*;

public interface SubjectRemote extends Remote {
    public ... method1 (...) throws RemoteException;
    public MyClass method2 throws RemoteException;
    ...
}

//This returned user-defined class must be declared Serializable
import java.io.*; //where Serializable lives

public class MyClass extends Serializable {
    ...
}

//The remote class
import java.rmi.*;
import java.rmi.server.*;

public class Subject extends UnicastRemoteObject implements SubjectRemote {
    private type1 instance1;
    private type2 instance2;
    ...

    public Subject (type1 x, type2 y) throws RemoteException {
        ...
    }

    public ... method1 (...) {
        ...
    }

    public MyClass method2 (...) {
        ...
    }

    ...
}
```
Proxy Pattern: Remote Proxy - Example (3)

//Driver for service side
//execute rmiregistry at command line before executing this

public class ServiceRemoteDriver {

    public static void main(String[] args) {
        ...

        try {
            SubjectRemote remoteSubject = new Subject(arg1, arg2);
            Naming.rebind("MyRemoteSubject", remoteSubject); //register remote subject
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

//Client

import java.rmi.*

public class Client {

    private SubjectRemote subject; //Use interface now

    public Client(SubjectRemote subject) {
        this.subject = subject;
    }

    public ... request1 (...) {
        try {
            subject.method1(...);
        } catch (Exception e) {
            e.printStackTrace();
        }
        ...
    }

    public ... request2 (...) {
        MyClass x;
        try {
            x = subject.method2(...);
        } catch (Exception e) {
            e.printStackTrace();
        }
        ...
    }

    ...
}
Proxy Pattern: Remote Proxy - Example (4)

//Driver for local side

import java.rmi.*;

public class ClientLocalDriver {

    public static void main (String[] args) {

        ... 

        Subject testSubject = new Subject(arg1, arg2);

        try {
            SubjectRemote remoteSubject = //get proxy
                (SubjectRemote) Naming.lookup("MyRemoteSubject");
            Client client = new Client(remoteSubject);
        } catch (Exception e) {
            e.printStackTrace();
        }
        client.request1(...);
        client.request2(...);
        ...
    }
}
Proxy Pattern: Virtual Proxy - Motivation

- Suppose you need to access some object that is expensive to create (i.e., takes a long time)

- Purpose of a Virtual Proxy is to
  1. Start the creation process
  2. While the process executes, represent the object
     - Allow the system to continue to function while waiting for the object to be created
     - Represent the object during this interval (e.g., printing a suitable msg)
  3. Turn control over to the object once creation is complete
     - Proxy simply acts as a pass-through object at this time
Proxy Pattern: Virtual Proxy - Motivation (2)

- Sample code:

```java
public interface SlowObjectInterface {
    public ... method1 (...);
    public ... method2 (...);
    public ... accessSlowObject (...);
    ...
}

public class SlowObject implements SlowObjectInterface {
    ...
    public ... method1 (...) {
        ...
    }
    public ... method2 (...) {
        ...
    }
    public ... accessSlowObject (...) {
        ...
    }
    ...
}
```
public class SlowObjectProxy implements SlowObjectInterface {
    SlowObject actualObject;
    Thread startupThread;
    boolean stillCranking = false;

    public SlowObjectProxy (...) {
        ...
    }

    public ... method1 (...) {
        ...
    }

    public ... method2 (...) {
        ...
    }

    public ... accessSlowObject (...) {
        if (actualObject != null) {
            actualObject.accessSlowObject(...)
        } else {
            if (!stillCranking) {
                stillCranking = true;
                retrievalThread = new Thread(new Runnable()) {
                    public void run() {
                        try {
                            actualObject = new SlowObject(...);
                            ...
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }
                retrievalThread.start();
            }
        }
    }
}

public class SlowObjectDriver {

    public static void main (String[] args) throws Exception {
        SlowObjectDriver testDrive = new SlowObjectDriver();
    }

    public SlowObjectDriver() throws Exception {
        SlowObjectProxy proxy = new SlowObjectProxy(...);
        proxy.accessSlowObject(...);
        ...
    }
}

Proxy Pattern: Java’s Dynamic Proxy

- Java supports dynamic creation of proxys
- Supporting package is `java.lang.reflect`
- Class diagram:

  ![Class Diagram](image)

  - Requires definition of an `InvocationHandler`
    - Since Java is creating the proxy for you, you need some way of handling calls on the proxy
    - The handler contains the code that would have gone into the proxy if you had created the proxy yourself
    - A handler wraps the object that a proxy was generated for
Proxy Pattern: Java’s Dynamic Proxy (2)

• Steps in using Java’s implementation

1. Create an InvocationHandler
   – Note that the InvocationHandler interface has a single method declaration: `invoke(Object proxy, Method method, Object[] args)`
   – When a proxy is accessed, the proxy calls the `invoke()` method
     * The first parameter represents the proxy
     * The second parameter represents the method called on the proxy
     * The last parameter represents the arguments passed to the method call
   – Depending on the logic of the handler, a subsequent call may be generated
   – For example, in the call `myProxy.myMethod("ABC", 10)`
     * `myProxy` is bound to `proxy`
     * `myMethod` is bound to `method`
     * `"ABC", 10` is bound to `args`
     * Assuming `realSubject` is the object that the proxy is representing, the above may then result in the call

     ```java
     myMethod.invoke(realSubject, args)
     ```

     * The end result is equivalent to `realSubject.myMethod("ABC", 10)"
Proxy Pattern: Java’s Dynamic Proxy (3)

Sample code

```java
import java.lang.reflect.*

public class MyInvocationHandler implements InvocationHandler {

    MySubject wrappedObject;

    public MyInvocationHandler (MySubject o) {
        wrappedObject = o;
    }

    public Object invoke (Object proxy, Method method, Object[] args)
        throws IllegalAccessException {
        try {
            return method.invoke(wrappedObject, args);
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}

Note: If implementing protection proxy (as in text), can control whether
to pass on method call using
* method.getName().startsWith(String)
* method.getName().equals(String)
* etc.
```
Proxy Pattern: Java’s Dynamic Proxy (4)

2. Create proxy class and instantiate proxy object
   - This is accomplished with
     \texttt{newProxyInstance(ClassLoader, Class\[\] \langle ? \rangle, InvocationHandler)}
     \begin{itemize}
     \item \texttt{newProxyInstance} is static and creates a new proxy object
     \item The first argument is the class loader for the subject
     \item The second argument is the set of interfaces the proxy will implement
     \item The third argument is the handler for the proxy
     \end{itemize}
   - Sample code:

     ```java
     MySubject object;
     (MySubject) Proxy.newProxyInstance(object.getClass().getClassLoader(),
                                           object.getClass.getInterfaces(),
                                           new MyInvocationHandler(object));
     ```

3. Usage

   MySubject object;
   MySubject myProxy =
       (MySubject) Proxy.newProxyInstance(object.getClass().getClassLoader(),
                                           object.getClass.getInterfaces(),
                                           new MyInvocationHandler(object));
   x = myProxy.getX();
   myProxy.setY(...);
   ...