Proxy Pattern

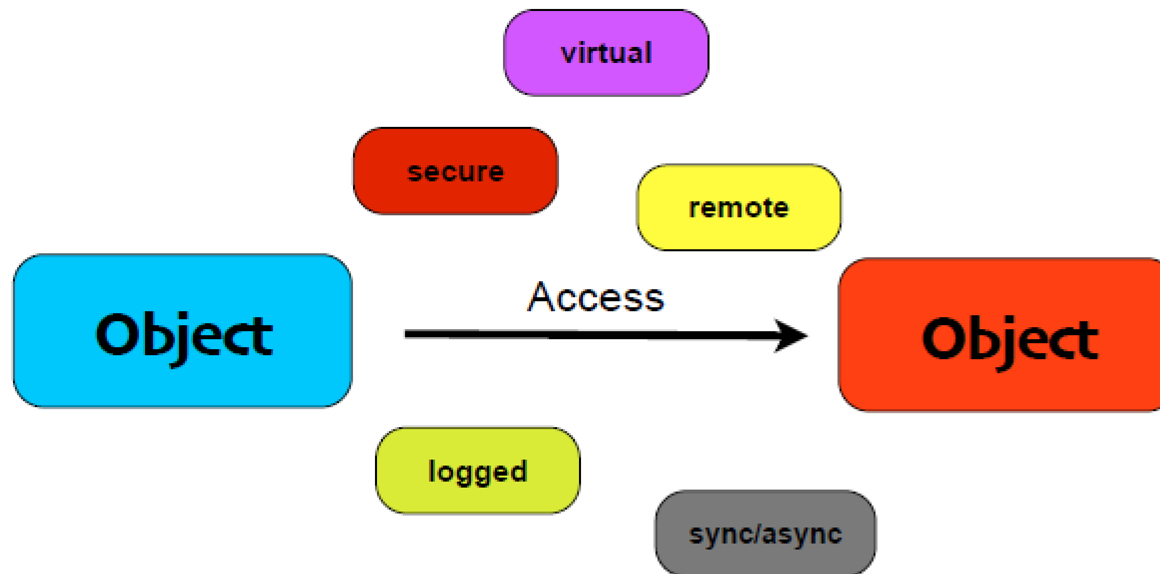
Provide a surrogate or placeholder for another object control access to the other object

A proxy in its most common form is a class functioning as an interface to something else: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

Problem

- Objects talk to each other using an interface that has been overburdened with the needs of networking, security, access coherence, or historic versions of the interface
- An object demands too much of another object

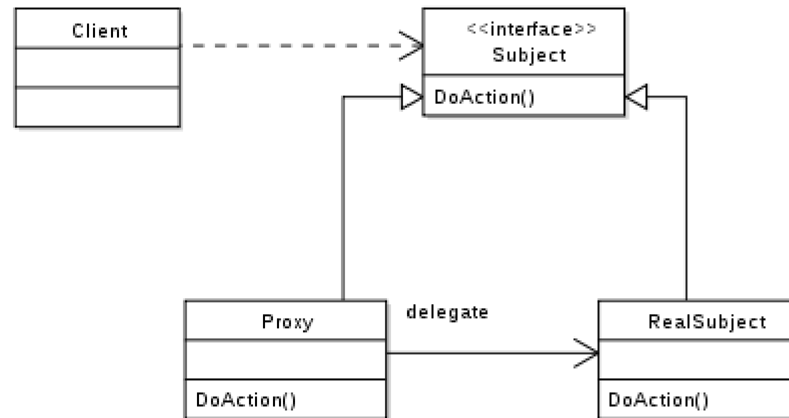
Object Access Control



Proxy Pattern: Delegation at its finest

- One object demands too much of another, so have the second (proxy) object handle some of the work and delegate the rest of the functionality to the object you want to interact with

Class Diagram



Types of Proxies: There are MANY

- Remote Proxy: Provides a reference to an object located in a different address space. HFDP example of remote gumball machine monitors each running in its own JVM
- Virtual Proxy: Allows the creation of a memory intensive object on demand. HFDP example of loading an image.
- Protection Proxy: Provides different clients with different levels of access. HFDP dating hot/not hot example
- Cache Proxy: Temporarily stores the results of expensive target operations

Types of Proxies

- Firewall Proxy: Protects targets from 'bad' clients
- Synchronization Proxy: Provides multiple clients access to target object
- Smart Reference Proxy: Tracks the number of references an object has
- Complexity Hiding Proxy: hides complexity of and **controls** access to a complex set of classes – sometimes called Facade Proxy
- Copy-On-Write Proxy: Controls the copying of an object until it is required by the client – a variant of Virtual Proxy

Proxy Pattern Similarities to Other Patterns

- Seems a little like Decorator (which adds behavior), but recall it's key job is to **control** access. Decorators may wrap an object many times, where this is seldom done with Proxy. Decorators also depend on main object existing (so they can wrap) – Proxy may be able to function without existence of object it delegates to
- Seems a little like Adapter (which changes the interface of the object it adapts), where Proxy implements the same interface. Adapter does not require that all of adapted object functionality be made visible – Protection Proxy can do the same thing

Java's Built-In Support for Proxy: Remote Proxy via Java RMI

- RMI: Remote Method Invocation
- Client/Server model
  - Client Object → Client Helper →|→Service Helper → Service Object
  - Client and server may live in different places and definitely use different memory space (separate JVM instances)
- Java RMI provides a stub on the client side that serves as the proxy

Java RMI Step by Step (this stuff is all from HFDP)

- Make a remote interface (defines the methods a client can call remotely and is what client uses as class type for the service)
- Make a remote implementation (it's what does the 'real' work) – it's the object the client wants to interact with
- Generate stubs (and possibly skeletons depending on Java version) using rmic – creates a stub .class file
- Start the RMI registry (rmiregistry) from a command prompt. This is where the client will go to get the proxy (the client stub/helper object)
- Start the remote service (java ClassNameOfService)

Interface Code

```
import java.rmi.*;

public interface MyRemote extends Remote {

   public String sayHello() throws RemoteException;

}
```

Remote/Server Implementation

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

public class MyRemoteImpl extends UnicastRemoteObject
implements MyRemote {

   public String sayHello() {
      return "Server says, 'Hey'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {
      try {
         MyRemote service = new MyRemoteImpl();
         Naming.rebind("RemoteHello", service);
      }
      catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Client Code

```
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://127.0.0.1/RemoteHello");
         String s = service.sayHello();
         System.out.println(s);
      }
      catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Virtual Proxy in Java: Use When Object you need is resource intensive

```
class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        }
        else {
            return 800;
        }
    }
```

Virtual Proxy In Java

```
public int getIconHeight() {
      if (imageIcon != null) {
         return imageIcon.getIconHeight();
      }
      else {
         return 600;
      }
   }
```

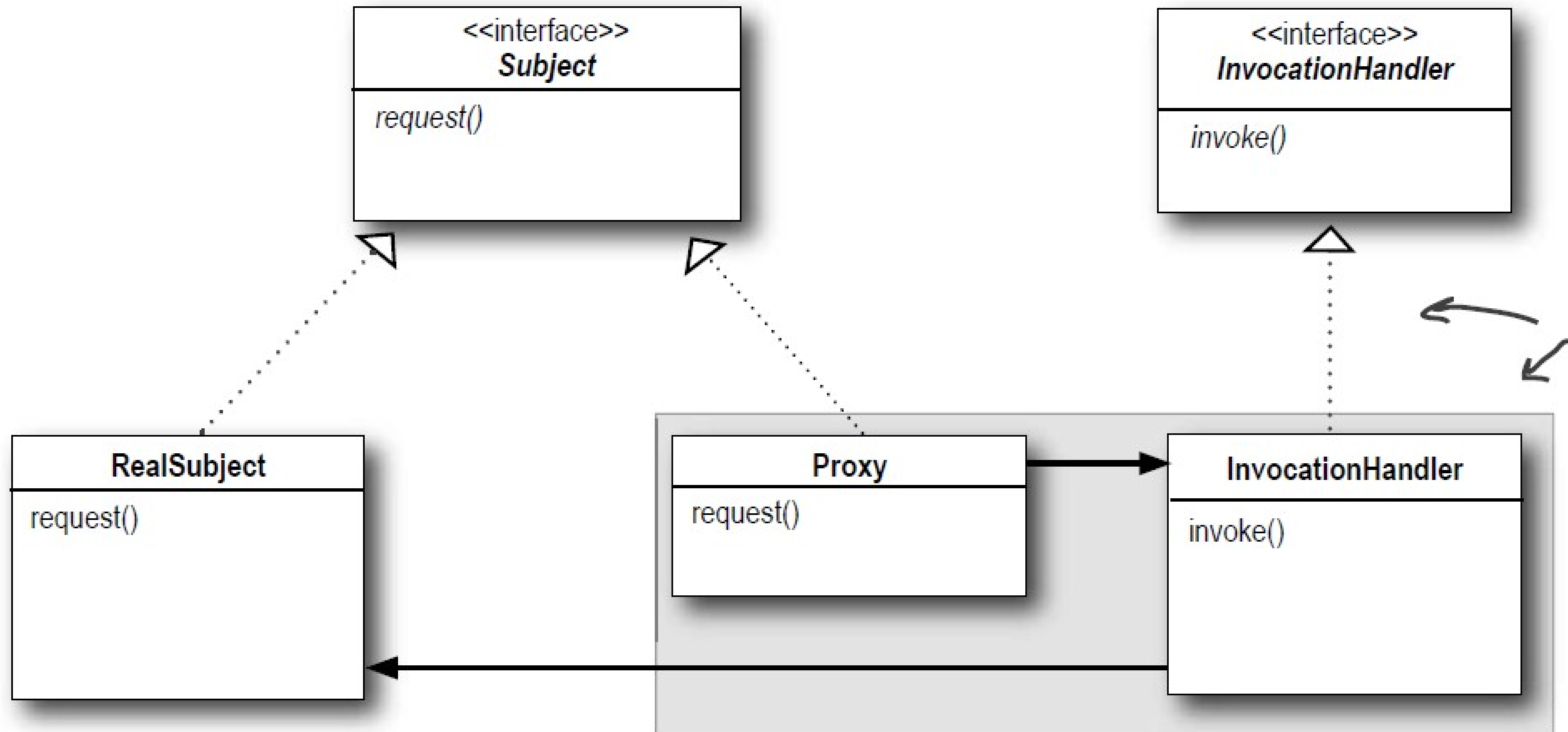
Virtual Proxy In Java

```
public void paintIcon(final Component c, Graphics g, int x, int y) {
      if (imageIcon != null) {
         imageIcon.paintIcon(c, g, x, y);
      }
      else {
         g.drawString("Loading CD cover, please wait...", x+300, y+190);
         if (!retrieving) {
            retrieving = true;
            retrievalThread = new Thread(new Runnable() {
               public void run() {
                  try {
                     imageIcon = new ImageIcon(imageURL, "CD Cover");
                     c.repaint();
                  }
                  catch (Exception e) {
                     e.printStackTrace();
                  }
               }
            });
            retrievalThread.start();
         }
      }
   }
```

Protection Proxy via Java API Proxy

- Lives in java.lang.reflect
- Lets you create a proxy class on the fly (dynamically) that imlements one or more interfaces and forwards method invocations to a class that you specify
- Requires an additional class to assist in handling invocations (InvocationHandler)
- Use when you want to restrict access to subject behaviors

Class Diagram for Dynamic Proxy



Creation Steps

- Create Subject Interface
- Create Real Subject
- Create Invocation Handlers
- Write the code that creates the dynamic proxies
- Wrap any subject object in appropriate (dynamic) proxy

Matchmaking Example: Subject Interface

```
public interface PersonBean {

    String getName();
    String getGender();
    String getInterests();
    int getHotOrNotRating();

     void setName(String name);
     void setGender(String gender);
     void setInterests(String interests);
     void setHotOrNotRating(int rating);

}
```

Subject Implementation

```
public class PersonBeanImpl implements PersonBean {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }
```

Subject Implementation

```
public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setHotOrNotRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

Owner Invocation Handler

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

public class OwnerInvocationHandler implements InvocationHandler {
   PersonBean person;

   public OwnerInvocationHandler(PersonBean person) {
      this.person = person;
   }

   public Object invoke(Object proxy, Method method, Object[] args)
         throws IllegalAccessException {

      try {
         if (method.getName().startsWith("get")) {
            return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
            throw new IllegalAccessException();
         } else if (method.getName().startsWith("set")) {
            return method.invoke(person, args);
         }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
      return null;
   }
}
```

Non Owner Invocation Handler

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

public class NonOwnerInvocationHandler implements InvocationHandler {
   PersonBean person;

   public NonOwnerInvocationHandler(PersonBean person) {
      this.person = person;
   }

   public Object invoke(Object proxy, Method method, Object[] args)
         throws IllegalAccessException {

      try {
         if (method.getName().startsWith("get")) {
            return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
            return method.invoke(person, args);
         } else if (method.getName().startsWith("set")) {
            throw new IllegalAccessException();
         }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
      return null;
   }
}
```

Matchmaking Test

```
import java.lang.reflect.*;
import java.util.*;

public class MatchMakingTestDrive {
   Hashtable datingDB = new Hashtable();

   public static void main(String[] args) {
      MatchMakingTestDrive test = new
MatchMakingTestDrive();
      test.drive();
   }

   public MatchMakingTestDrive() {
      initializeDatabase();
   }
```

Matchmaking Test

```
public void drive() {
      PersonBean joe = getPersonFromDatabase("Joe Javabean");
      PersonBean ownerProxy = getOwnerProxy(joe);
      System.out.println("Name is " + ownerProxy.getName());
      ownerProxy.setInterests("bowling, Go");
      System.out.println("Interests set from owner proxy");
      try {
         ownerProxy.setHotOrNotRating(10);
      } catch (Exception e) {
         System.out.println("Can't set rating from owner proxy");
      }
      System.out.println("Rating is " + ownerProxy.getHotOrNotRating());

      PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
      System.out.println("Name is " + nonOwnerProxy.getName());
      try {
         nonOwnerProxy.setInterests("bowling, Go");
      } catch (Exception e) {
         System.out.println("Can't set interests from non owner proxy");
      }
      nonOwnerProxy.setHotOrNotRating(3);
      System.out.println("Rating set from non owner proxy");
      System.out.println("Rating is " +
nonOwnerProxy.getHotOrNotRating());
   }
```

Matchmaking Test

```
PersonBean getOwnerProxy(PersonBean person) {

        return (PersonBean) Proxy.newProxyInstance(
               person.getClass().getClassLoader(),
               person.getClass().getInterfaces(),
                new OwnerInvocationHandler(person));
   }

   PersonBean getNonOwnerProxy(PersonBean person) {

        return (PersonBean) Proxy.newProxyInstance(
               person.getClass().getClassLoader(),
               person.getClass().getInterfaces(),
                new NonOwnerInvocationHandler(person));
   }

   PersonBean getPersonFromDatabase(String name) {
      return (PersonBean)datingDB.get(name);
   }
```

Matchmaking Test

```
void initializeDatabase() {
      PersonBean joe = new PersonBeanImpl();
      joe.setName("Joe Javabean");
      joe.setInterests("cars, computers, music");
      joe.setHotOrNotRating(7);
      datingDB.put(joe.getName(), joe);

      PersonBean kelly = new PersonBeanImpl();
      kelly.setName("Kelly Klosure");
      kelly.setInterests("ebay, movies, music");
      kelly.setHotOrNotRating(6);
      datingDB.put(kelly.getName(), kelly);
   }
}
```

Dynamic Proxy (from Java API)

- Created on demand from the set of interfaces it is passed
- InvocationHandler is not a proxy – it is what the proxy dispatches to for handling method calls

Closing Thoughts

- Proxy, like any wrapper, increases the number of classes and objects in your designs
- There are many ways a proxy can manage access to subject (as evidenced by the many different versions of proxy)
- Similar to Decorator but it controls access where Decorator adds behavior. Because of this, Proxy is considered a Structural Pattern