

Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186


 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

Proxy Design Pattern

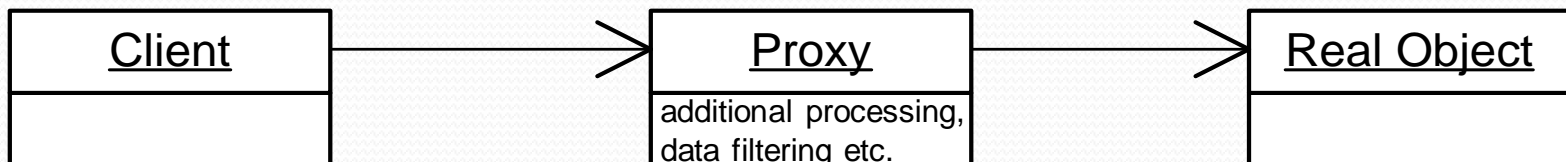
What is Proxy pattern?

Proxy is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy 	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

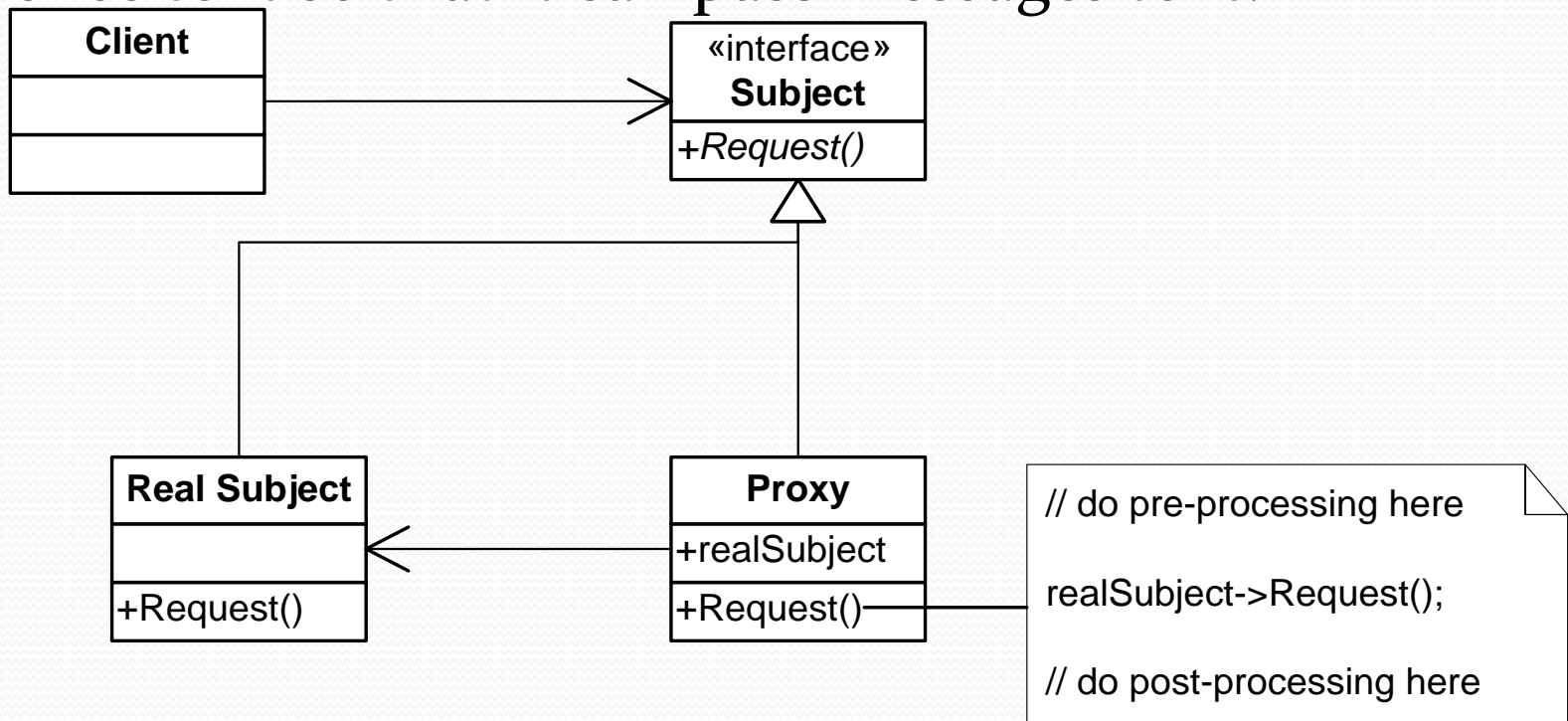
Proxy- structural pattern

- Proxy allows for object level access control by acting as a pass through entity or a placeholder object.
- **Intent:** Provides control for accessing original object.
- Also known as **Surrogate or Placeholder**.
- **Problem:** You need to control access to an object.
- **Solution:**
 - Create a Proxy object that implements the same interface as the real object
 - The Proxy object (usually) contains a reference to the real object
 - Clients are given a reference to the Proxy, not the real object
 - All client operations on the object pass through the Proxy, allowing the Proxy to perform additional processing

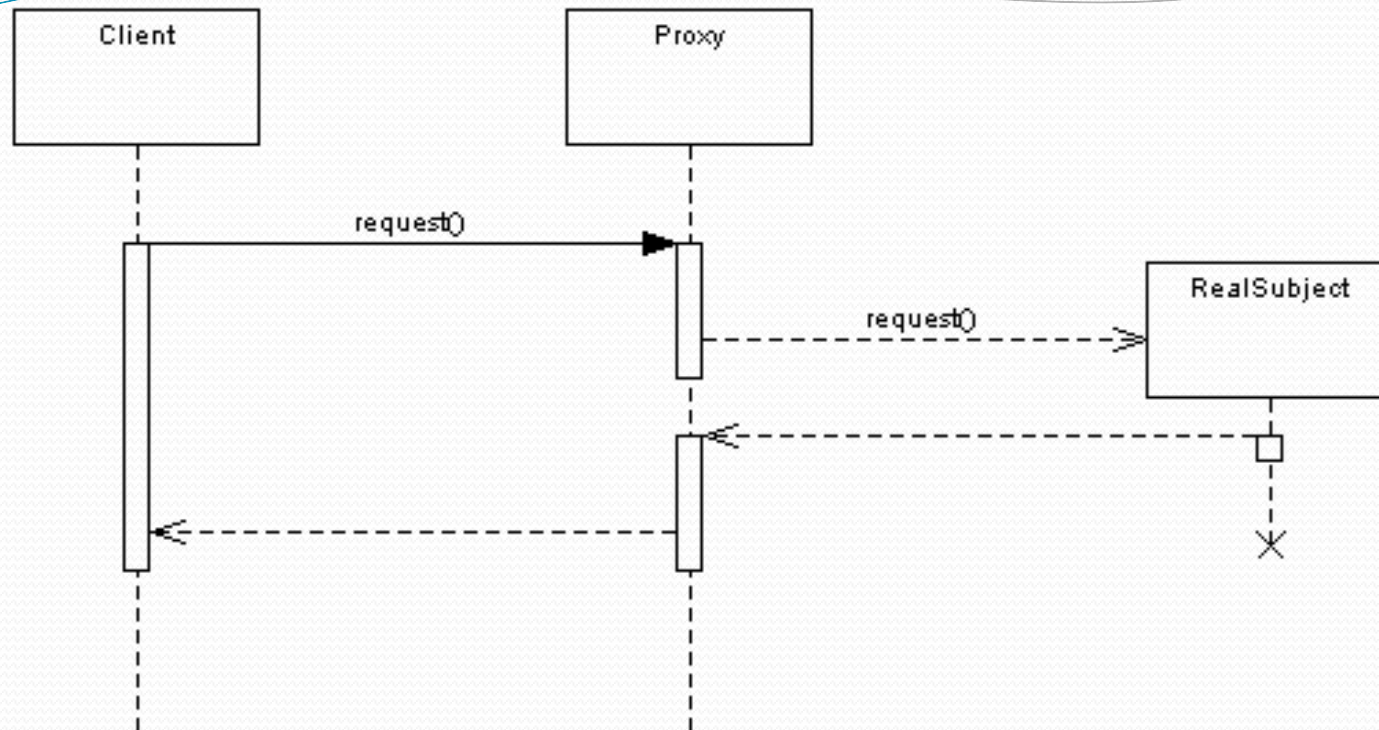


Solution

- The interface that the client knows about is the **Subject**. Both the **Proxy** and **RealSubject** objects implement the **Subject** interface, but the client may not be able to access the **RealSubject** without going through the **Proxy**. The **Proxy** would handle the creation of the **RealSubject** object, but it will have a reference to it so that it can pass messages to it.



Proxy UML Sequence & Uses



- **Would I Use This Pattern?**

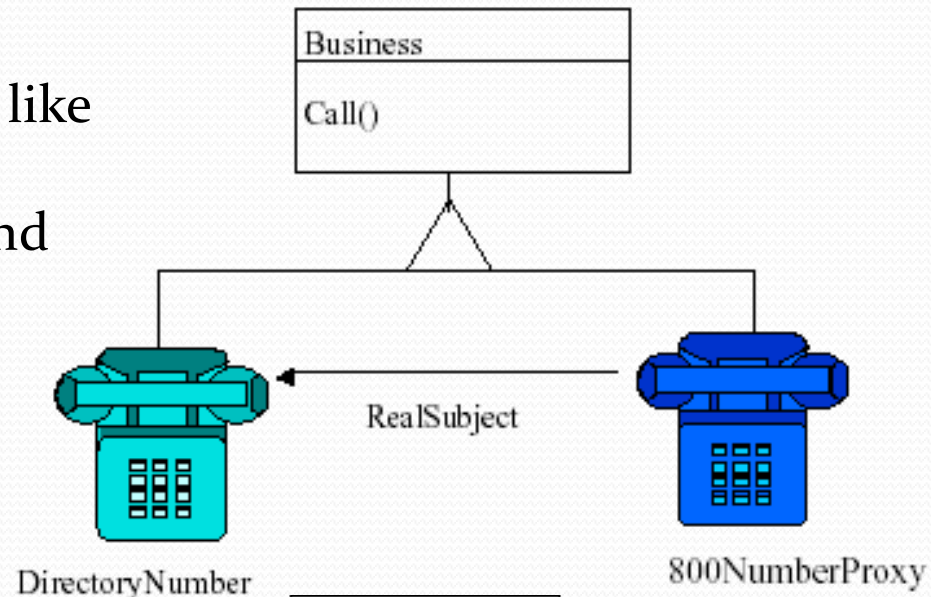
- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required
- Additional functionality is required when an object is accessed.

The *Proxy* Design Pattern

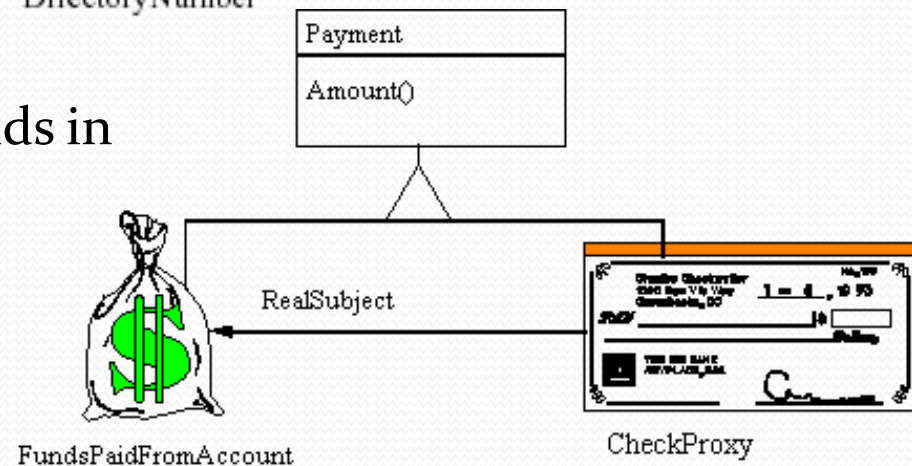
- Problem
 - **Situation I:** Wish to control access to an object because:
 - Housekeeping activities are required
 - Protection levels exist
 - Objects sit in different address spaces
 - Actual object too expensive to build immediately
 - **Situation II:** Wish to provide transparent management of services
 - The client “thinks” it is using the actual service when rather it is using a proxy for the actual service

The *Proxy* Design Pattern

- **Example1** : Toll-free numbers
- Client dials the “800” area code just like they would for a real area code
- Proxy records billing information and connects to actual area code



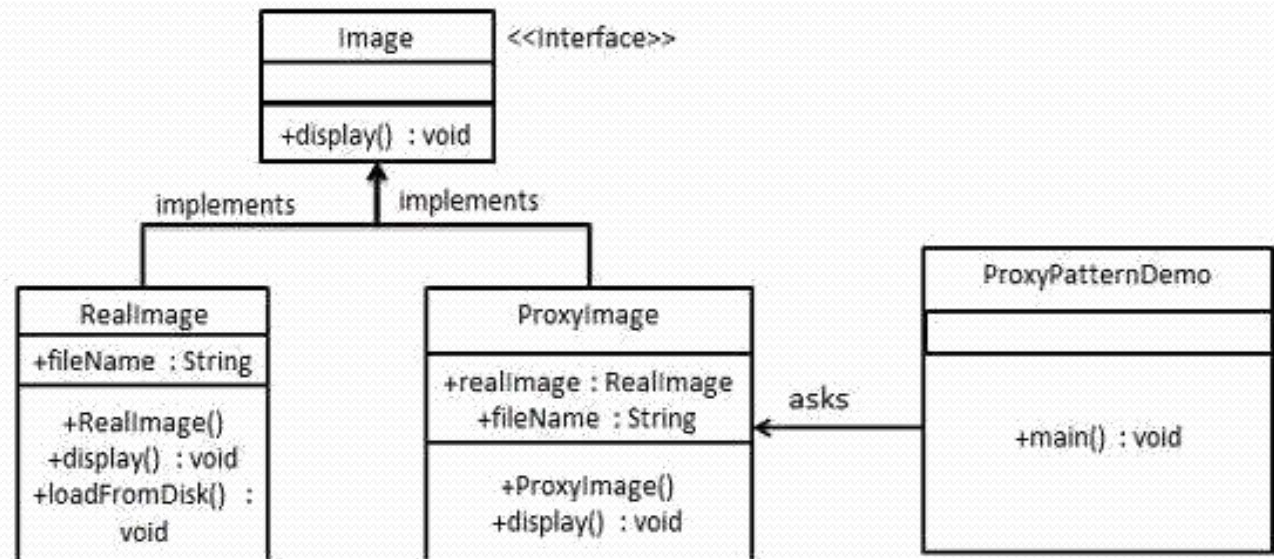
- **Example2** : Bank Cheque
- Cheque is a proxy for the actual funds in the bank



- **Example 3:** RMI API uses proxy design pattern. Stub and Skeleton are two proxy objects used in RMI.

Example 4:

- We are going to create an *Image* interface and concrete classes implementing the *Image* interface. *ProxyImage* is a proxy class to reduce memory footprint of *RealImage* object loading.
- *ProxyPatternDemo*, our demo class, will use *ProxyImage* to get an *Image* object to load and display as it needs.



Example 4:

- **Step 1:** Create an interface. *Image.java*

```
public interface Image {  
    void display();  
}
```

- **Step 2:** Create concrete classes implementing interface. *RealImage.java* , and *ProxyImage.java*

```
public class RealImage implements Image{  
    private String fileName;  
    public RealImage(String fileName){  
        this.fileName = fileName;  
        loadFromDisk(fileName);  
    }  
    @Override  
    public void display() {  
        System.out.println("Displaying " + fileName);  
    }  
    private void loadFromDisk(String fileName){  
        System.out.println("Loading " + fileName);  
    }  
}
```

Example 4:

```
public class ProxyImage implements Image {  
    private RealImage realImage;  
    private String fileName;  
    public ProxyImage(String fileName){  
        this.fileName = fileName;  
    }  
    @Override  
    public void display() {  
        if(realImage == null){  
            realImage = new RealImage(fileName);  
        }  
        realImage.display();  
    }  
}
```

Example 4:

- **Step 3:** Use the *ProxyImage* to get object of *RealImage* class when required. *ProxyPatternDemo.java*

```
public class ProxyPatternDemo {  
    public static void main(String[] args) {  
        Image image = new ProxyImage("test_10mb.jpg");  
        //image will be loaded from disk  
        image.display();  
        System.out.println("");  
        //image will not be loaded from disk  
        image.display();  
    }  
}
```

- *Output:*

```
Loading test_10mb.jpg  
Displaying test_10mb.jpg
```

```
Displaying test_10mb.jpg
```

Example 5

- Step 1:

```
public interface Internet {  
    public void connectTo(String serverhost) throws Exception;  
}
```

- Step 2:

```
public class RealInternet implements Internet {  
    @Override  
    public void connectTo(String serverhost)  
    {  
        System.out.println("Connecting to "+ serverhost);  
    }  
}
```

Example 5

- Step3:

```
import java.util.ArrayList;
import java.util.List;
public class ProxyInternet implements Internet{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;
    static
    {
        bannedSites = new ArrayList<String>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
        bannedSites.add("lnm.com");
    }
    @Override
    public void connectTo(String serverhost) throws Exception
    {
        if(bannedSites.contains(serverhost.toLowerCase()))
        {
            //System.out.println("Access Denied");
            throw new Exception("Access Denied");
        }
        internet.connectTo(serverhost);
    }
}
```

Example 5

- Step4:

```
public class DemoClient {  
    public static void main(String[] args) {  
        Internet internet = new ProxyInternet();  
        try  
        {  
            internet.connectTo("alphapeeler.sf.net");  
            internet.connectTo("abc.com");  
        }  
        catch (Exception e)  
        {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

- Output:

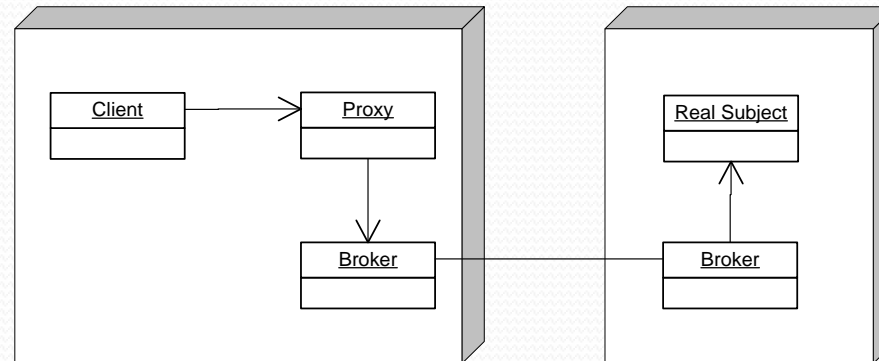
```
Connecting to alphapeeler.sf.net  
Access Denied
```

Known Uses: Java Collections

- Read-only Collections
 - Wrap collection object in a proxy that only allows read-only operations to be invoked on the collection
 - All other operations throw exceptions
 - `List Collections.unmodifiableList(List list);`
 - Returns read-only List proxy
- Synchronized Collections
 - Wrap collection object in a proxy that ensures only one thread at a time is allowed to access the collection
 - Proxy acquires lock before calling a method, and releases lock after the method completes
 - `List Collections.synchronizedList(List list);`
 - Returns a synchronized List proxy

Known Uses: Distributed Objects

- The Client and Real Subject are in different processes or on different machines, and so a direct method call will not work
- The Proxy's job is to pass the method call across process or machine boundaries, and return the result to the client (with Broker's help)



Known Uses: Secure Objects

- Different clients have different levels of access privileges to an object
- Clients access the object through a proxy
- The proxy either allows or rejects a method call depending on what method is being called and who is calling it (i.e., the client's identity)

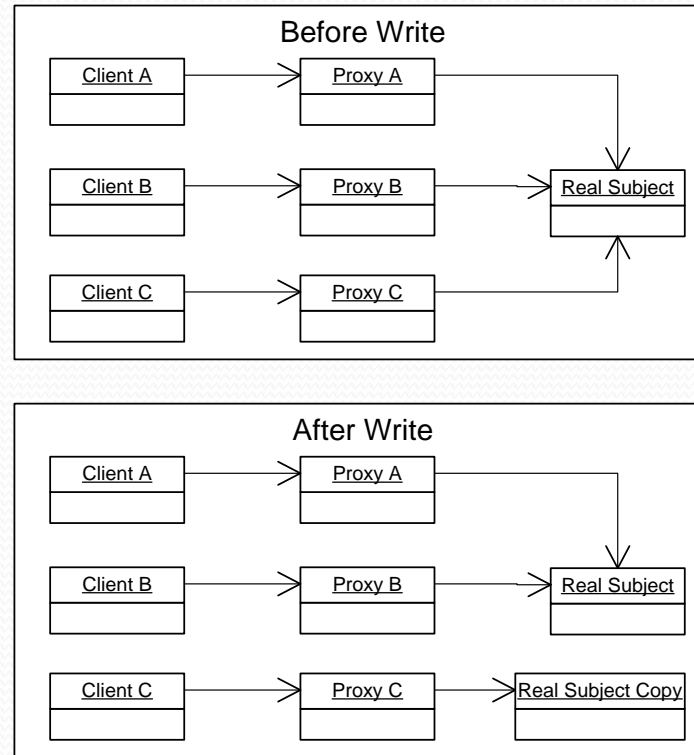
Known Uses: Lazy Loading

- Some objects are expensive to instantiate (i.e., consume lots of resources or take a long time to initialize)
- Rather than instantiating an expensive object right away, create a proxy instead, and give the proxy to the client
- The proxy creates the object on demand when the client first uses it
- If the client never uses the object, the expense of creating it is never incurred
- A hybrid approach can be used, where the proxy implements some operations itself, and only needs to create the real object if the client calls one of the operations it doesn't implement
- Proxies must store whatever information is needed to create the object on-the-fly (file name, network address, etc.)

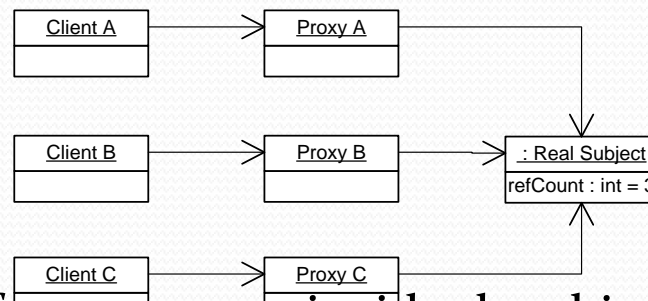
Known Uses: Lazy Loading

- Examples
- Object-Oriented Databases
 - Graph of objects stored on disk
 - Objects contain references to each other
 - Load proxies initially, and only load the real object from disk if a method is actually called on it
- Resource Conservation
 - If you need to store thousands of objects in memory at once, proxies can be used to save memory by only loading objects that are actually used
 - Objects that are used can be unloaded after awhile, freeing up memory
- Word Processor
 - Documents that contain lots of multimedia objects should still load fast
 - Create proxies that represent large images, movies, etc., and only load objects on demand as they become visible on the screen (only a small part of the document is visible at a time)

Known Uses: Copy-on-Write



Counting



- Proxies maintain the reference count inside the object
- The last proxy to go away is responsible for deleting the object (i.e., when the reference count goes to 0, delete the object)

Proxy vs. Decorator

- The Proxy pattern is very similar in structure to the Decorator pattern. Both patterns create a “wrapper” around another object. (Adapter is a third kind of “wrapper”, but is clearly different from Decorator and Proxy because an adapter has a different interface than the wrapped object).
- Here are some differences between the two patterns:
 - The intents of the patterns are different. Decorator is used to **add responsibilities** to an object (without using inheritance). A Proxy is used to **“control access”** to an object. Rather than adding functionality, a Proxy might actually prevent access to functionality.
 - Read-only collections, Secure objects
 - Decorators are often organized into chains where each decorator adds a separate responsibility. Proxies are **usually** not organized into chains (although they could be).
 - A Decorator always stores a reference to the wrapped object; a Proxy may or may not, depending on what it does
 - Distributed objects (proxy never stores direct object reference)
 - Lazy Loading (proxy sometimes stores a direct object reference)

- The *Proxy* Design Pattern
 - Consequences
 - Isolates `RealSubject` from client
 - Forces controllable indirection
 - Costs indirection
 - True for all delegation models
 - May require duplicate information in `Proxy` and `RealSubject`
 - Must support same "properties"
 - These must be supported even before instantiation of `RealSubject`

• The *Proxy* Design Pattern

- Implementation Issues

- Three basic types of proxies

1. Remote Proxy

- » Proxy used to "hide" the location of RealSubject
- » Example: RMI implementation

2. Virtual Proxy

- » Performs optimizations
- » Example: Image rendering in HTML

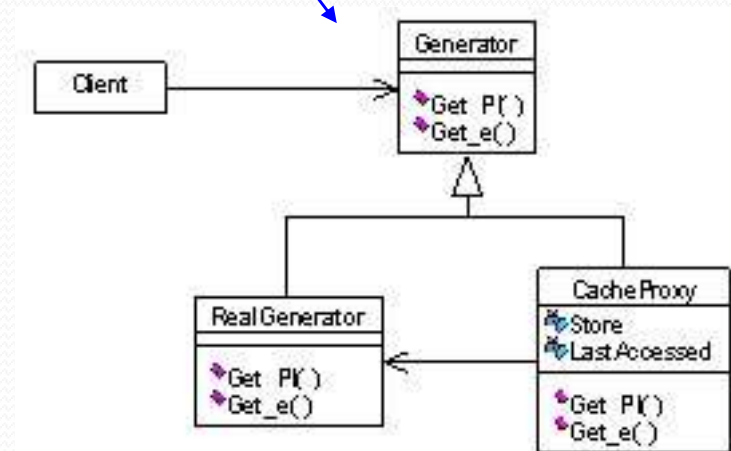
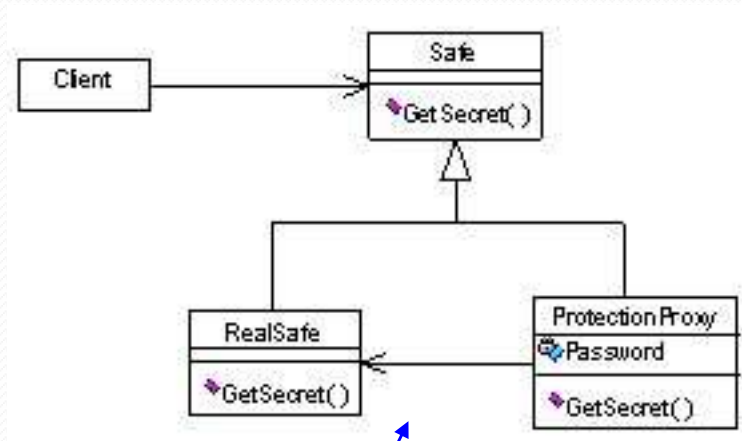
• The *Proxy* Design Pattern

- Implementation Issues

- Three basic types of proxies

3. Housekeeping Proxy (Protection Proxy)

- » Performs additional maintenance duties
- » Example: Cache Proxy



- » Example: Protection Proxy

• The *Proxy* Design Pattern

- Common Variations

- Firewall Proxy (a *protection* proxy)
 - Protects targets from bad clients (or vice versa)
- Synchronization Proxy (a *housekeeping* proxy)
 - Provides multiple accesses to a target object
- Smart Reference Proxy (a *housekeeping* proxy)
 - Provides additional actions whenever a target object is referenced
- Copy-on-Write Proxy (a *virtual* proxy)
 - Defers cloning an object until required