**Chaper 11 Notes – Proxy Pattern**

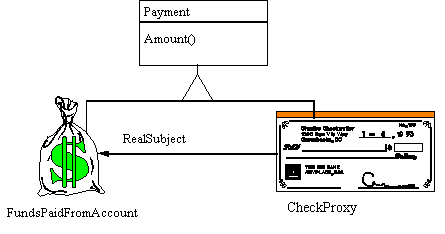
**Introduction**

1. A *proxy*, according to the Merriam-Webster dictionary is:

* A document giving authority or power to act for another
* A person authorized to act for another

For example:

1. A board of directors for a publicly owned company is a *proxy* for all the individual owners.
2. A *power of attorney* is a document that allows one person (the proxy) to conduct legal affairs for another.
3. A check or a credit card is a *proxy* for cash as shown below. Why might one prefer a credit card (check) to cash: (a) the cash is bulky, it is *heavy* and (b) the credit card is more *secure.* We will see later that *heavy* and *secure* are common reasons for using the Proxy pattern in programming.



source: <http://www.vincehuston.org/dp/real_demos.html>

<http://www.cs.uni.edu/~wallingf/teaching/062/sessions/support/pattern-examples.pdf>

**Proxy Pattern**

**Problem**: You need to control access to an object

**Solution**: Although this doesn’t capture every situation where a proxy is useful, it is a reasonably general description of the proxy.

1. Create a Proxy object that implements the same interface as the real object
2. The Proxy object contains a reference to the real object
3. Clients are given a reference to the Proxy, not the real object
4. All client operations on the object pass through the Proxy, allowing the Proxy to perform additional processing

|  |  |
| --- | --- |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\11-Proxy\pics\zz2.jpg | E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\11-Proxy\pics\zz3.jpg |

**IService service = new Proxy();**

**Client client = new Client( service );**

**client.doTask;**

**Consequences**

1. Provides an additional level of indirection between client and object that may be used to insert arbitrary services
2. Proxies are invisible to the client, so introducing proxies does not affect client code. A proxy and the object it is standing in for must share a common interface. To a client, there is no difference.

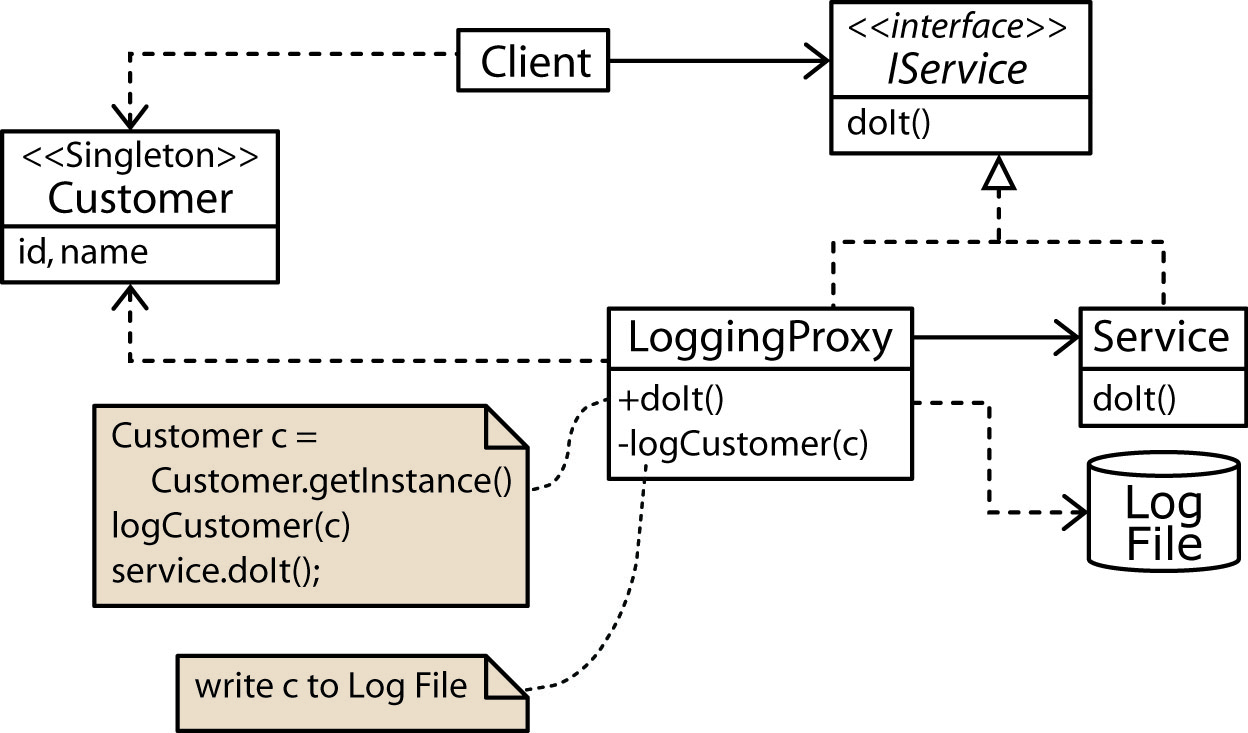
**Other Descriptions of Proxy Pattern**

1. *The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. [HFDP, p.460]*
2. *Clients must communicate with a representative (a proxy) of the desired resource, not the resource itself*.
3. *The Proxy Pattern forces method calls to an object to occur indirectly through a proxy object... Classes for proxy objects are declared in a way that usually eliminates the client object’s awareness that it is dealing with a proxy. [PIJ-1, p.91]*
4. In design patterns, a proxy is an object that stands in for some other object. The GOF says the intent of the Proxy pattern is, "Provide a surrogate or placeholder for another object to control access to it." In this sense, a *firewall* is a proxy for a webserver. However, there are many of types of situations where the proxy is appropriate where “…control access to it” really only comes into play in the most general sense.

**Types of Proxies**

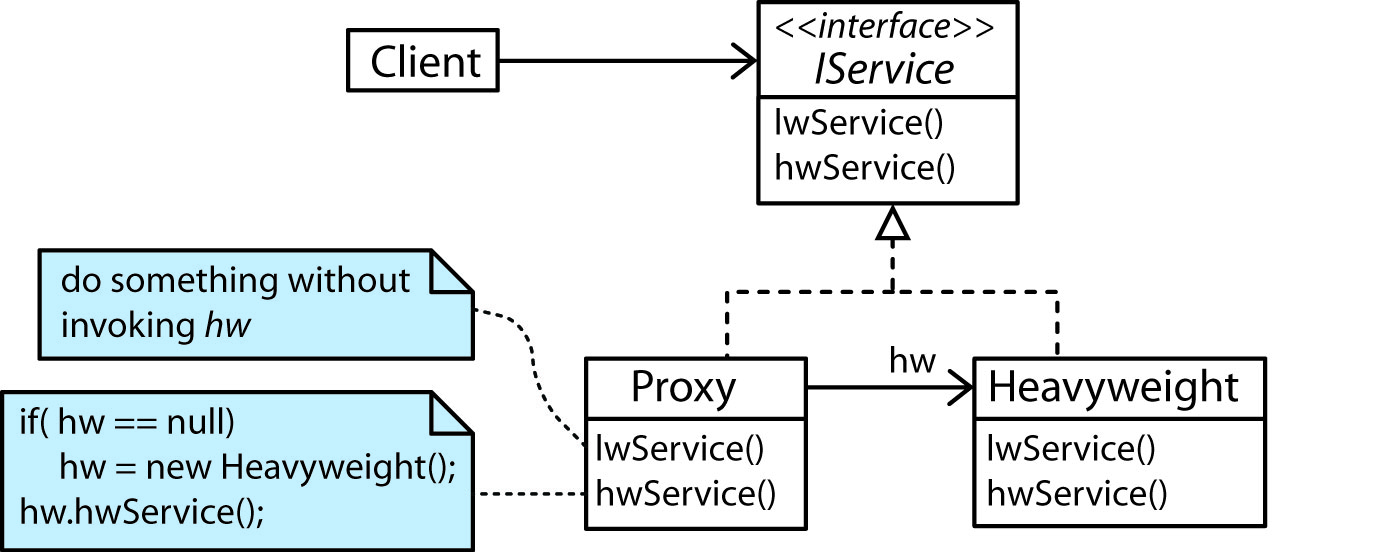
**Logging Proxy**

Consider the case of a *logging proxy*. Suppose we want to record the name and id of every user who invokes a particular service.



**Virtual Proxy**

1. The Virtual Proxy provides for lazy instantiation of services. Suppose we have a heavyweight object, one that is large and/or time-consuming to create, or to initialize and that the complete object is not always needed. We can define a *virtual proxy* to stand-in for the heavyweight object where the proxy can do some of the tasks of the heavyweight without invoking the heavyweight. Call this a *lightweight* service. However, the proxy cannot handle some of the operations and must delegate them to the heavyweight to carry out. Call this a *heavyweight* service.



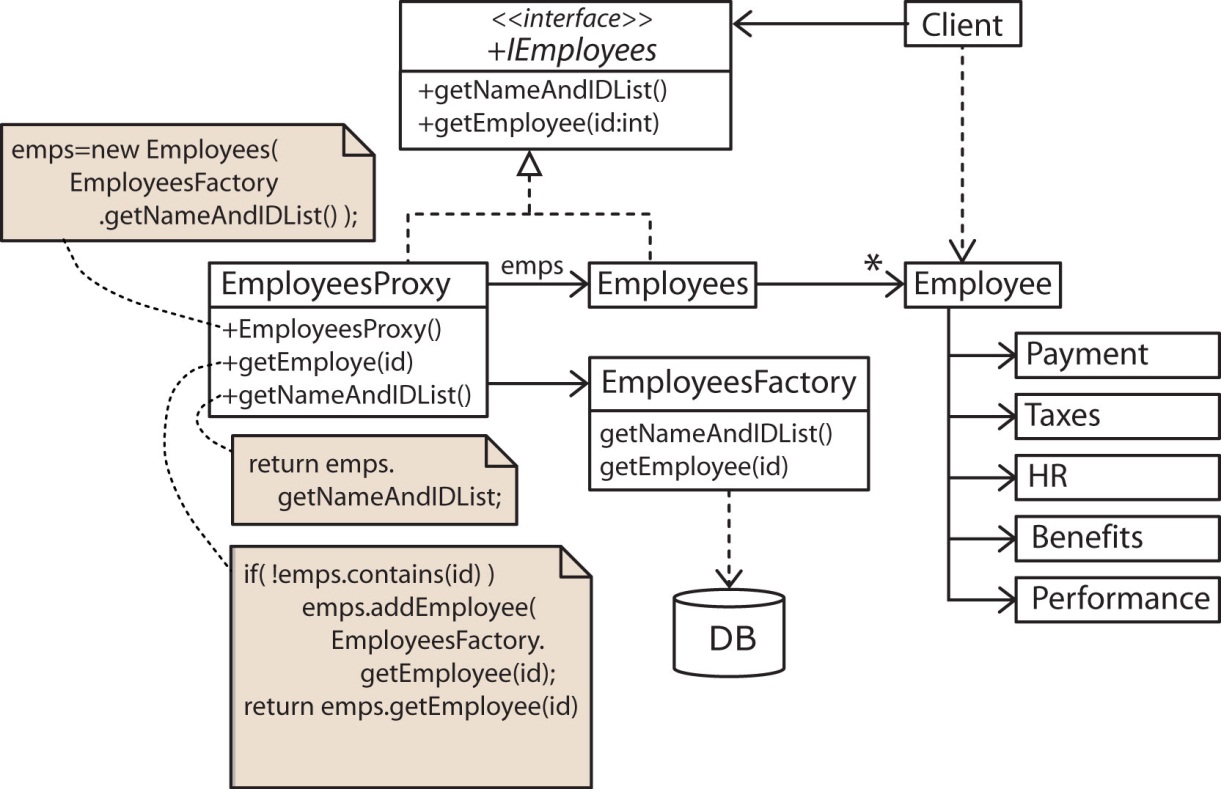
The virtual proxy allows us to create the illusion that the heavyweight object is always available.

1. Example: “The Proxy design pattern shows a way to do just in time loading of objects that would consume too much memory to keep around, or takes a lot of time to load. This can be a very useful pattern for many applications. A good example of this pattern is in Microsoft Office. When you open a large Word document that has lots of embedded pictures, Office doesn’t load them all at the time you open the document. As you scroll down, Office will pull the pictures from the disk file and insert them into the document. You can see this by scrolling very fast down the document. It takes a second or so for the document to “catch up” to you and show the visible images. “

Source: <http://aspalliance.com/827_Gang_of_Four_GOF_Design_Patterns.6>

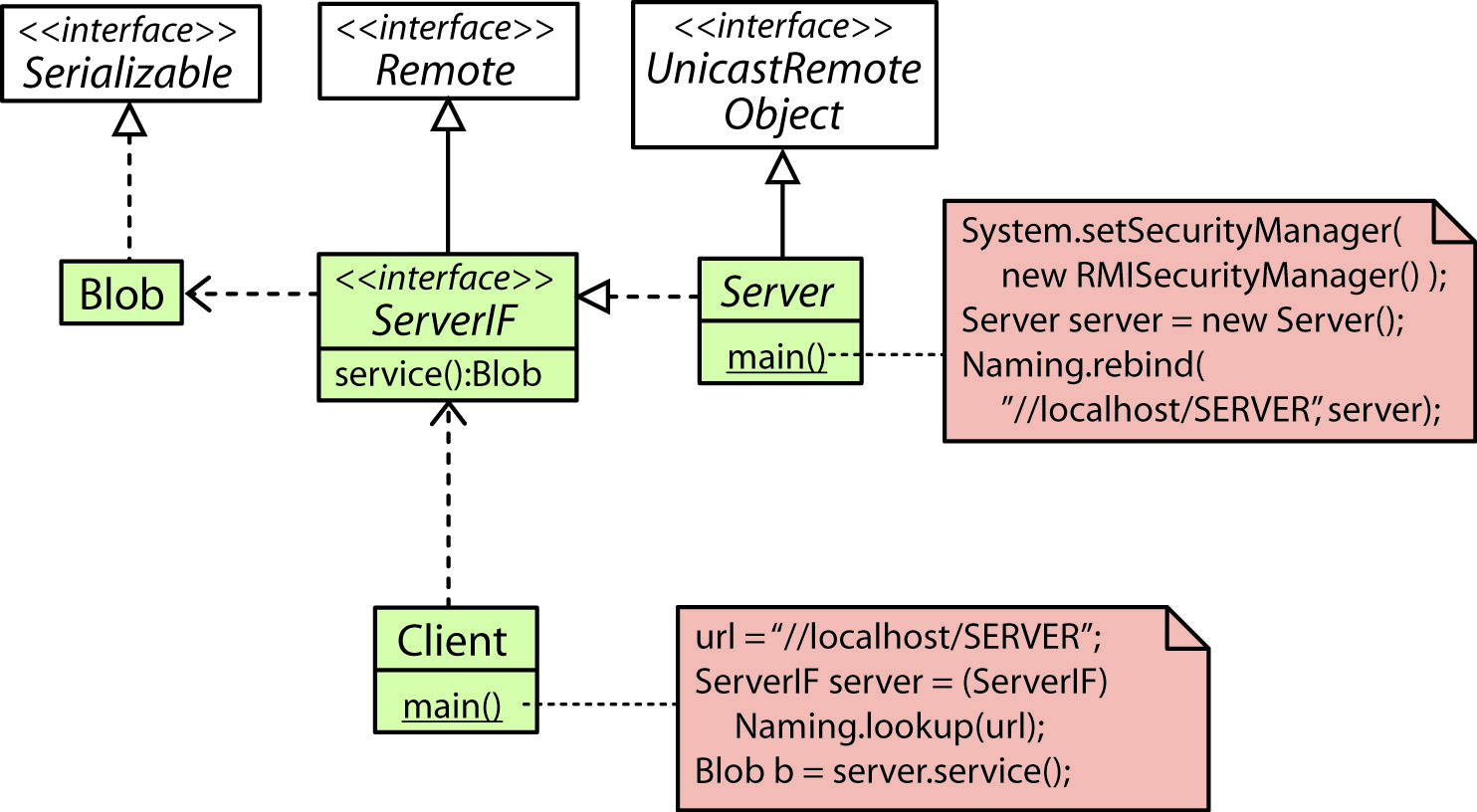
1. Example: Employee objects are heavyweight objects. Creating an Employee object requires obtaining information from multiple databases. An EmployeesFactory is defined for building and returning an Employee object. An Employees object maintains a collection of Employee objects but it is not necessary for the entire collection to be in memory. However, the name and id of each employee is needed frequently, for instance, to be used in a drop-down. When a name is selected from the drop-down, then detailed information (*e.g.* payment, taxes, human resources, *etc*.) is displayed.

The way we model this is to define a proxy for the Employees collection. The proxy can supply the name and ID list. When a request is made for an Employee object, the proxy first looks in the Employees collection to see if it is there and returns it if so. If it is not, the proxy: (a) uses the factory to obtain the Employee, (b) adds the Employee to the Employees collection, (c) returns the Employee. This allows us to write code in a client that assumes the full collection of Employees is available.

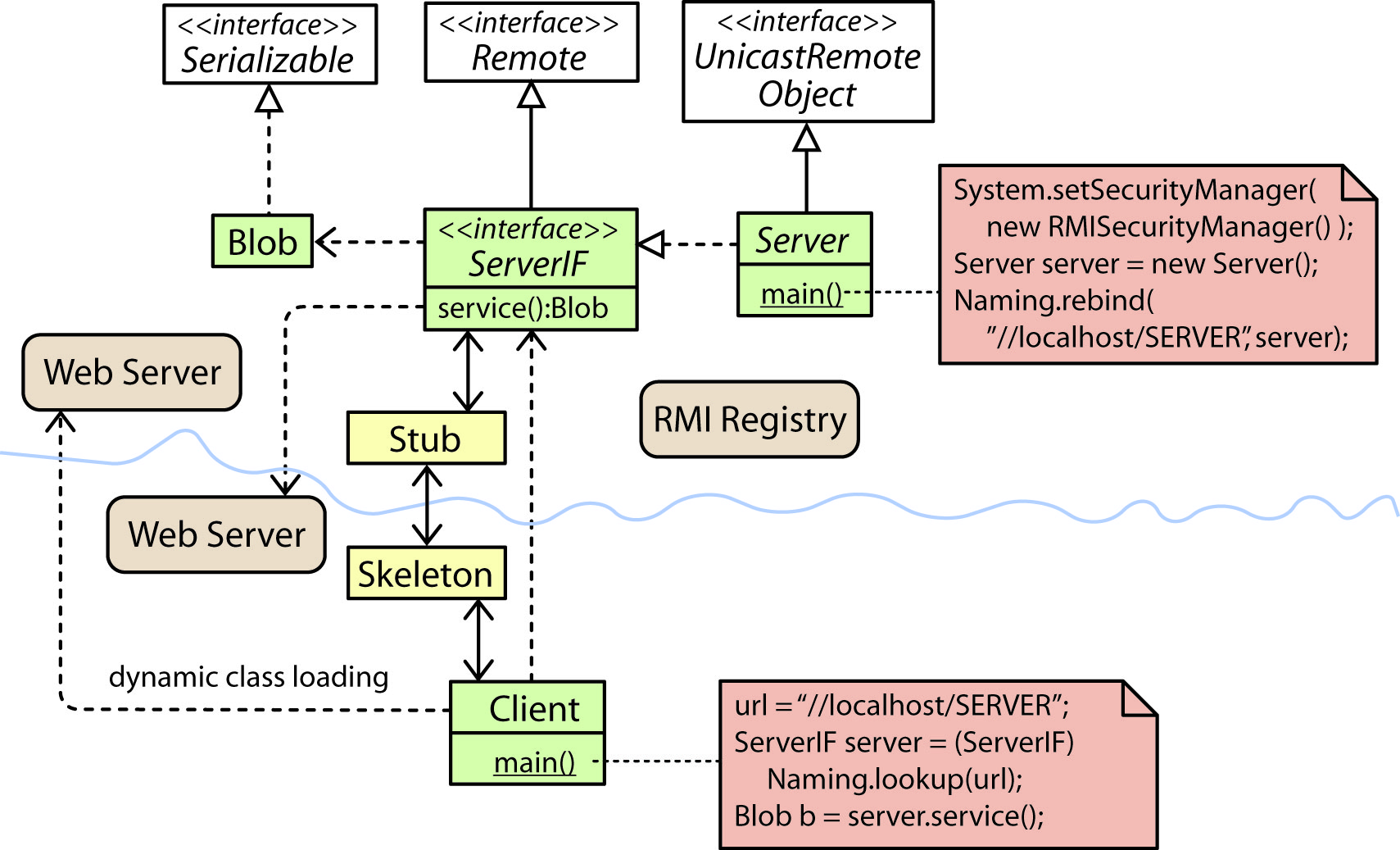


**Remote Proxy**

1. Java’s Remote Method Invocation (RMI) package (*java.rmi*) allows you to call methods on objects across a network. In the example below, the Client wants to call the *service* method on the Server which exists across a network (where the *service* method returns a Blob to the Client).Using RMI, we would model this situation as shown below. Behind the scenes, the proxy pattern is at work.



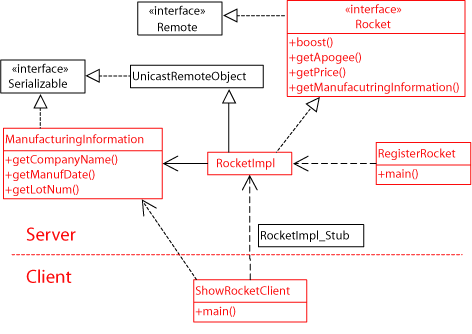
1. So, where exactly is the Proxy pattern. The Client does not directly communicate with the Server as shown above. Behind the scenes there is a Skeleton class on the Client and a Stub class on the Server. The communication goes through these classes. Thus, the Skeleton and Stub are proxies while the Client gets to use the Server as if it were local.



When the server is first run, it creates a Stub object and stores it in RMI Registry. Later, a Client looks up the Server in the RMI Registry, which creates a Skeleton on the Client and connects it with the Stub instance on the Server.

One important question is how does the Client get the Server class (or the Blob class)? It could be as simple as copying it onto the Client machine. A more flexible approach is to use dynamic class loading. This means that RMI attaches a URL to required classes which are dynamically loaded from a web server on the Server when needed. Similarly, the Server might need Client classes and can obtain them from a web server on the Client.

1. Steps to use RMI:
2. Define remote interface
3. Define remote server that implements remote interface
4. Create and register remote server instance (Naming.rebind)
5. Create client that looks-up (Naming.lookup) remote server and uses it
6. Other useful technologies: CORBA (Common Object Request Broker Architecture), Jini (Service Oriented Architecture for building distributed systems).
7. Example – Controlling a rocket across a network.



**Copy-on-Write Proxy**

1. A *copy-on-write* proxy is a technique used when multiple clients need simultaneous access to an object.
2. Example: Data (a data object) is accessed by different clients. One client needs to read a large of amount of data; however, the client does *not* want to be notified of changes to the data by another client. In other words, the client wants to work with a “snapshot” of the data as it was when it first started accessing the data.
3. Example: A hash map is used by several clients (or threads). One client wants to iterate over the set of values. Meanwhile, another client (thread) might remove a value from the hash map. The first client still wants to see the hash map as it was when it first accessed it.
4. Solutions:
5. use a lock, e.g. synchronized – but may hold lock to long
6. client clones the data object – wasteful, if no one tried to modify data object.
7. client clones data object only when necessary.
8. Idea: Client thinks he has a copy of the data. When another client tries to modify the data, the proxy creates a true copy and redirects the client to the copy while the second client modifies the actual data.
9. Used in:
10. virtual memory operating systems
11. disk storage
12. string class in c++
13. database servers
14. <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CopyOnWriteArrayList.html>
15. <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CopyOnWriteArraySet.html>

**Cache Proxy**

1. A situation can exist where it is too expensive (or too time consuming) to create resources every time they are requested. The solution is to *cache* the resources so that when requests for resources come along, they go through a *Cache* *Proxy* that will search the cache for the resource before requesting a new instance.
2. Used in:
3. CPU
4. Disk management
5. web pages, both client and server
6. file system management

**Protection Proxy**

1. Controls access to an object. Before access is allowed to object, permissions are checked.

<http://www.junginger.biz/resources/proxy/proxy3.html>

Proxy Servers - An example



1. A proxy server allows indirect, controlled access to a web server and typically provides for:
2. Faster access to cached web sites
3. Increased security (firewall, virus scan)
4. Control, limit and log user access

See: <http://en.wikipedia.org/wiki/Proxy_server>

**Firewall Proxy**

Local clients are protected from outside world.

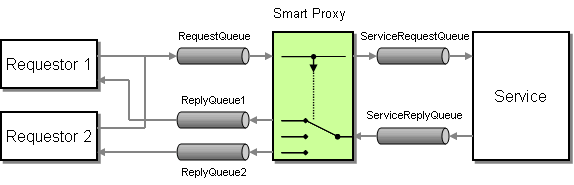
**Smart Proxy**

1. A *smart proxy* is essentially when there is added behavior to either the remote or virtual proxy.

<http://www.eaipatterns.com/SmartProxy.html>

1. Example: The Smart Proxy intercepts messages sent on the request channel to the [Request-Reply](http://www.eaipatterns.com/RequestReply.html) service. For each incoming message, the Smart Proxy stores the [Return Address](http://www.eaipatterns.com/ReturnAddress.html) specified by the original sender. It then replaces the [Return Address](http://www.eaipatterns.com/ReturnAddress.html) in the message with the channel the reply channel that the Smart Proxy is listening on. When a reply message comes in on that channel, the Smart Proxy retrieves the stored [Return Address](http://www.eaipatterns.com/ReturnAddress.html) and uses a [Message Router](http://www.eaipatterns.com/MessageRouter.html) to forward the unmodified reply address to that channel.

A *wire-tap* could be placed in such a smart proxy.



**Synchronization Proxy**

1. Control multi-threaded access to a section of code.

<http://www.cityofangels.com/Experts/proxy.pdf>

1. Control number of clients that access a server

<http://www.cs.sjsu.edu/faculty/pearce/oom/patterns/behavioral/proxy.htm>

**Counting Proxy**

1. Data collection about usage of an object. Count how many times it has been used, or the current number of references.

**Smart Pointer Proxy**

1. Counts the number of references to the object. Thus, proxy can explicitly release object (resources) from memory when there are no references to the object.

**Difference between Proxy and Adapter**

1. Both proxy and adapter wrap another class and the class diagrams are similar. In the proxy pattern, both classes implement the same interface. However, in the adapter pattern, the adaptee does not implement the interface. More importantly, their intents are different. The intent of the adapter pattern is to make two incompatible classes communicate. The intent of the proxy pattern is to control access to another class.

**Difference between Proxy and Decorator**

1. Both proxy and decorator add behavior to a class. A proxy does it through delegation while a decorator does it through composition. Again, the intents are different. The intent of the decorator is simply to add additional behavior that is generally related to task at hand.

**References**

1. [JavaWorld](http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html) – Graphics example and decorated proxies.
2. [Oracle](http://download.oracle.com/javase/tutorial/rmi/index.html) – RMI tutorial
3. [Hao Shen Blog](http://haoshen.org/?p=73) – Simple RMI tutorial
4. [RoseIndia](http://www.roseindia.net/tutorials/rmi/rmi-helloworld.shtml) – Simple RMI tutorial
5. [Sanjeev Setia](http://www.cs.gmu.edu/~setia/cs475/slides/lecture15.pdf) – Echo server example using RMI
6. [Javaworld](http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-smartproxy.html) – RMI and smart proxy.