The Proxy Pattern

The Proxy Pattern

- Intent
  - Provide a surrogate or placeholder for another object to control access to it
- Also Known As
  - Surrogate
- Motivation
  - A *proxy* is
    - a person authorized to act for another person
    - an agent or substitute
    - the authority to act for another
  - There are situations in which a client does not or can not reference an object directly, but wants to still interact with the object
  - A proxy object can act as the intermediary between the client and the target object

The Proxy Pattern

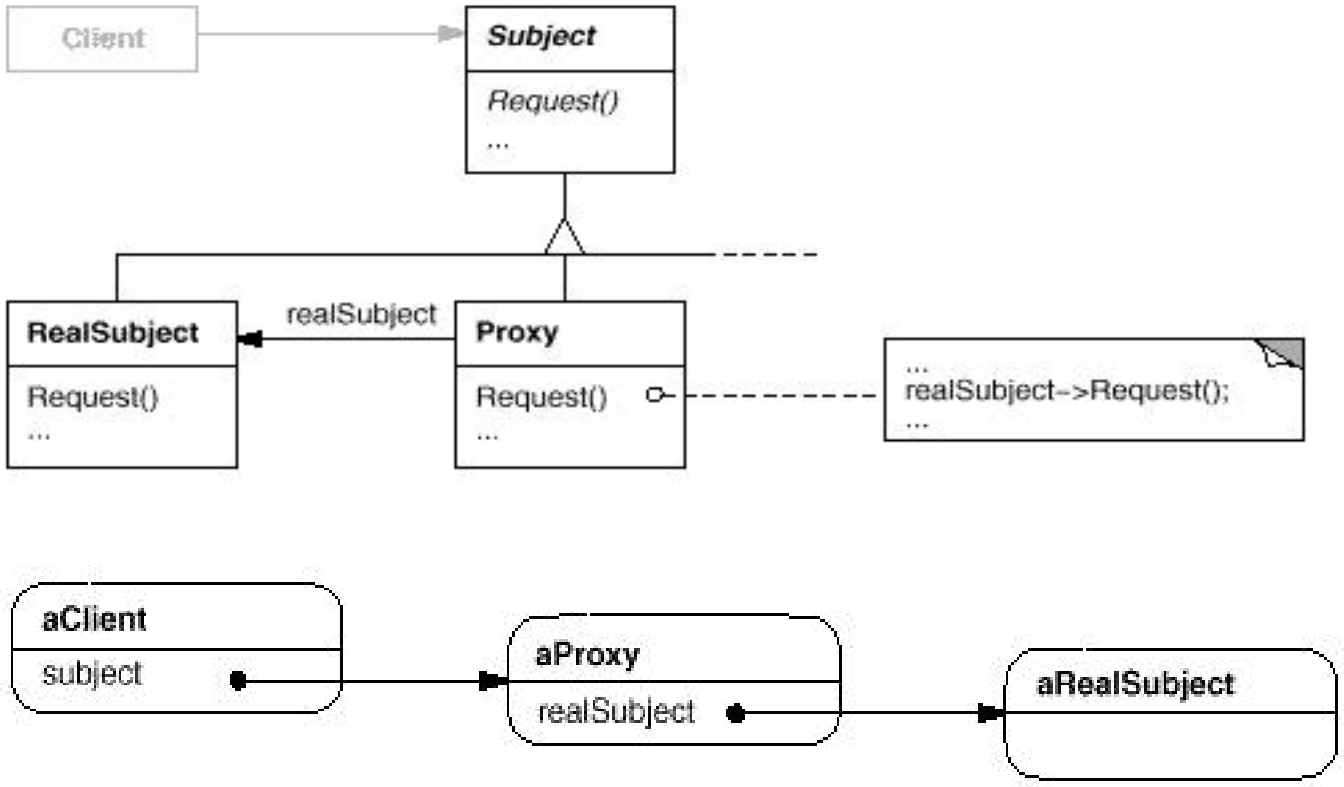
- Motivation
  - ⇨ The proxy object has the same interface as the target object
  - ⇨ The proxy holds a reference to the target object and can forward requests to the target as required (delegation!)
  - ⇨ In effect, the proxy object has the authority the act on behalf of the client to interact with the target object
- Applicability
  - ⇨ Proxies are useful wherever there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide

The Proxy Pattern

- Types of Proxies
  - *Remote Proxy* - Provides a reference to an object located in a different address space on the same or different machine
  - *Virtual Proxy* - Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed.
  - *Copy-On-Write Proxy* - Defers copying (cloning) a target object until required by client actions. Really a form of virtual proxy.
  - *Protection (Access) Proxy* - Provides different clients with different levels of access to a target object
  - *Cache Proxy* - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results
  - *Firewall Proxy* - Protects targets from bad clients (or vice versa)
  - *Synchronization Proxy* - Provides multiple accesses to a target object
  - *Smart Reference Proxy* - Provides additional actions whenever a target object is referenced such as counting the number of references to the object

The Proxy Pattern

- Structure



Copy-On-Write Proxy Example

- Scenario: Suppose we have a large collection object, such as a hash table, which multiple clients want to access concurrently. One of the clients wants to perform a series of consecutive fetch operations while not letting any other client add or remove elements.
- Solution 1: Use the collection's lock object. Have the client implement a method which obtains the lock, performs its fetches and then releases the lock.
- For example:

```
public void doFetches(Hashtable ht) {
  synchronized(ht) {
    // Do fetches using ht reference.
  }
}
```

Copy-On-Write Proxy Example (Continued)

- But this method may require holding the collection object's lock for a long period of time, thus preventing other threads from accessing the collection
- Solution 2: Have the client clone the collection prior to performing its fetch operations. It is assumed that the collection object is cloneable and provides a clone method that performs a sufficiently deep copy.
- For example, java.util.Hashtable provides a clone method that makes a copy of the hash table itself, but not the key and value objects

Copy-On-Write Proxy Example (Continued)

- The doFetches() method is now:

```
public void doFetches(Hashtable ht) {
  Hashtable newht = (Hashtable) ht.clone();
  // Do fetches using newht reference.
}
```

- The collection lock is held while the clone is being created. But once the clone is created, the fetch operations are done on the cloned copy, without holding the original collection lock.
- But if no other client modifies the collection while the fetch operations are being done, the expensive clone operation was a wasted effort!

Copy-On-Write Proxy Example (Continued)

- Solution 3: It would be nice if we could actually clone the collection only when we need to, that is when some other client has modified the collection. For example, it would be great if the client that wants to do a series of fetches could invoke the clone() method, but no actual copy of the collection would be made until some other client modifies the collection. This is a *copy-on-write* cloning operation.
- We can implement this solution using proxies
- Here is an example implementation of such a proxy for a hash table written by Mark Grand from the book *Patterns in Java, Volume 1*

Copy-On-Write Proxy Example (Continued)

- The proxy is the class LargeHashtable. When the proxy's clone() method is invoked, it returns a copy of the proxy and both proxies refer to the same hash table. When one of the proxies modifies the hash table, the hash table itself is cloned. The ReferenceCountedHashTable class is used to let the proxies know they are working with a shared hash table . This class keeps track of the number of proxies using the shared hash table.

Copy-On-Write Proxy Example (Continued)

```
// The proxy.
public class LargeHashtable extends Hashtable {

  // The ReferenceCountedHashTable that this is a proxy for.
  private ReferenceCountedHashTable theHashTable;

  // Constructor
  public LargeHashtable() {
    theHashTable = new ReferenceCountedHashTable();
  }

  // Return the number of key-value pairs in this hashtable.
  public int size() {
    return theHashTable.size();
  }
```

Copy-On-Write Proxy Example (Continued)

```
// Return the value associated with the specified key.
public synchronized Object get(Object key) {
  return theHashTable.get(key);
}

// Add the given key-value pair to this Hashtable.
public synchronized Object put(Object key, Object value) {
  copyOnWrite();
  return theHashTable.put(key, value);
}

// Return a copy of this proxy that accesses the same Hashtable.
public synchronized Object clone() {
  Object copy = super.clone();
  theHashTable.addProxy();
  return copy;
}
```

Copy-On-Write Proxy Example (Continued)

```
// This method is called before modifying the underlying
// Hashtable.  If it is being shared then this method clones it.
private void copyOnWrite() {
  if (theHashTable.getProxyCount() > 1) {
    synchronized (theHashTable) {
      theHashTable.removeProxy();
      try {
        theHashTable =  (ReferenceCountedHashTable)
                         theHashTable.clone();
      } catch (Throwable e) {
         theHashTable.addProxy();
      }
    }
  }
}
…
```

Copy-On-Write Proxy Example (Continued)

```
// Private class to keep track of proxies sharing the hash table.
private class ReferenceCountedHashTable extends Hashtable {

  private int proxyCount = 1;

  // Constructor
  public ReferenceCountedHashTable() {
    super();
  }

  // Return a copy of this object with proxyCount set back to 1.
  public synchronized Object clone() {
    ReferenceCountedHashTable copy;
    copy = (ReferenceCountedHashTable)super.clone();
    copy.proxyCount = 1;
    return copy;
  }
```

Copy-On-Write Proxy Example (Continued)

```
    // Return the number of proxies using this object.
    synchronized int getProxyCount() {
      return proxyCount;
    }

    // Increment the number of proxies using this object by one.
    synchronized void addProxy() {
      proxyCount++;
    }

    // Decrement the number of proxies using this object by one.
    synchronized void removeProxy() {
      proxyCount--;
    }
  }

}
```

Cache Proxy Example

- Scenario: An Internet Service Provider notices that many of its clients are frequently accessing the same web pages, resulting in multiple copies of the web documents being transmitted through its server. What can the ISP do to improve this situation?
- Solution: Use a Cache Proxy!
- The ISP's server can cache recently accessed pages and when a client request arrives, the server can check to see if the document is already in the cache and then return the cached copy. The ISP's server accesses the target web server only if the requested document is not in the cache or is out of date.

Synchronization Proxy Example

- Scenario: A class library provides a Table class, but does not provide a capability to allow clients to lock individual table rows. We do not have the source code for this class library, but we have complete documentation and know the interface of the Table class. How can we provide a row locking capability for the Table class?
- Solution: Use a Synchronization Proxy!
- Here is an example implementation written by Roger Whitney

Synchronization Proxy Example (Continued)

- First part of the Table class, just so we can see its interface:

```
public class Table implements ITable {

  ...

  public Object getElementAt(int row, int column) {
    // Get the element.
  }

  public void setElementAt(Object element, int row,
                           int column ) {
    // Set the element.
  }

  public int getNumberOfRows() {return numrows;}

}
```

Synchronization Proxy Example (Continued)

- Here is the table proxy:

```
public class RowLockTableProxy implements ITable {

  Table realTable;
  Integer[] locks;

  public RowLockTableProxy(Table toLock)  {
    realTable = toLock;
    locks = new Integer[toLock.getNumberOfRows()];
    for (int row = 0; row < toLock.getNumberOfRows(); row++)
      locks[row] = new Integer(row);
  }
```

Synchronization Proxy Example (Continued)

```
  public Object getElementAt(int row, int column)  {
    synchronized (locks[row])  {
      return realTable.getElementAt(row, column);
    }
  }

  public void setElementAt(Object element, int row, int column) {
    synchronized (locks[row]) {
      return realTable.setElementAt(element, row, column);
    }
  }

  public int getNumberOfRows() {
    return realTable.getNumberOfRows();
  }
}
```

Virtual Proxy Example

- Scenario: A Java applet has some very large classes which take a long time for a browser to download from a web server. How can we delay the downloading of these classes so that the applet starts as quickly as possible?
- Solution: Use a Virtual Proxy!
- When using a Virtual Proxy:
  - ⇨ All classes other than the proxy itself must access the target class indirectly through the proxy. If any class makes a static reference to the target class, the Java Virtual Machine will cause the class to be downloaded. This is true even if no instantiation of the target class is done.

Virtual Proxy Example (Continued)

- When using a Virtual Proxy (Continued):
  - ⇨ Even the proxy can not make a static reference to the target class initially. So how does the proxy reference the target class? It must use some form of dynamic reference to the target. A *dynamic reference* encapsulates the target class name in a string so that the Java compiler does not actually see any reference to the target class and does not generate code to have the JVM download the class. The proxy can then use the new Reflection API to create an instance of the target class.
  - ⇨ Both the proxy and the target object implement the same interface which in Java will be a regular Java interface. Any class can reference this interface, since the interface definition is small and will be quickly downloaded.

Virtual Proxy Example (Continued)

- Suppose one of the large classes is called LargeClass. It implements the ILargeClass interface as shown here:

```
// The ILargeClass interface.
public interface ILargeClass {
  public void method1();
  public void method2();
}

// The LargeClass class.
public class LargeClass implements ILargeClass {

  private String title;
  public LargeClass(String title) {this.title = title;}
  public void method1() {// Do method1 stuff.}
  public void method2() {// Do method2 stuff.}
}
```

Virtual Proxy Example (Continued)

- Here's the proxy class:

```
// The LargeClassProxy class.
public class LargeClassProxy implements ILargeClass {
  private ILargeClass largeClass = null;
  private String title;

  // Constructor
  public LargeClassProxy(String title) {
    this.title = title;
  }

  // Method 1. Create LargeClass instance if needed.
  public void method1() {
    if (largeClass == null)
      largeClass = createLargeClass();
    largeClass.method1();
  }
```

Virtual Proxy Example (Continued)

```
// Method 2. Create LargeClass instance if needed.
public void method2() {
  if (largeClass == null)
    largeClass = createLargeClass();
  largeClass.method2();
}

// Private method to create the LargeClass instance.
private ILargeClass createLargeClass() {
  ILargeClass lc = null;
  try {
    // Get Class object for LargeClass.
    // When we do this, the class will be downloaded.
    Class c = Class.forName("LargeClass");

    // Get Class objects for the LargeClass(String) constructor
    // arguments.
    Class[] args = new Class[] {String.class};
```

Virtual Proxy Example (Continued)

```
        // Get the LargeClass(String) constructor.
        Constructor cons = c.getConstructor(args);

        // Create the instance of LargeClass.
        Object[] actualArgs = new Object[] {title};
        lc = (ILargeClass) cons.newInstance(actualArgs);
        System.out.println("Creating instance of LargeClass");
      }
      catch (Exception e) {
        System.out.println("Exception: " + e);
      }
      return lc;
    }

  }
```

Virtual Proxy Example (Continued)

- Here's a typical client:

```
// Client of LargeClass.
public class Client {

  public static void main(String args[]) {
    // Create a LargeClass proxy.
    ILargeClass lc = new LargeClassProxy("Title");

    // Do other things...
    System.out.println("Doing other things...");

    // Now invoke a method of LargeClass.
    // The proxy will create it.
    lc.method1();
  }
}
```

Remote Proxy Example

- Scenario: A machine at the College of OOAD has several utility services running as daemons on well-known ports. We want to be able to access these services from various client machines as if they were local objects. How can this be accomplished?
- Solution: Use a Remote Proxy!
- This is the essence of modern distributed object technology such as RMI, CORBA and Jini