Lesson 3 Adapter, Proxy and Iterator

The Adapter Pattern

1. Intent

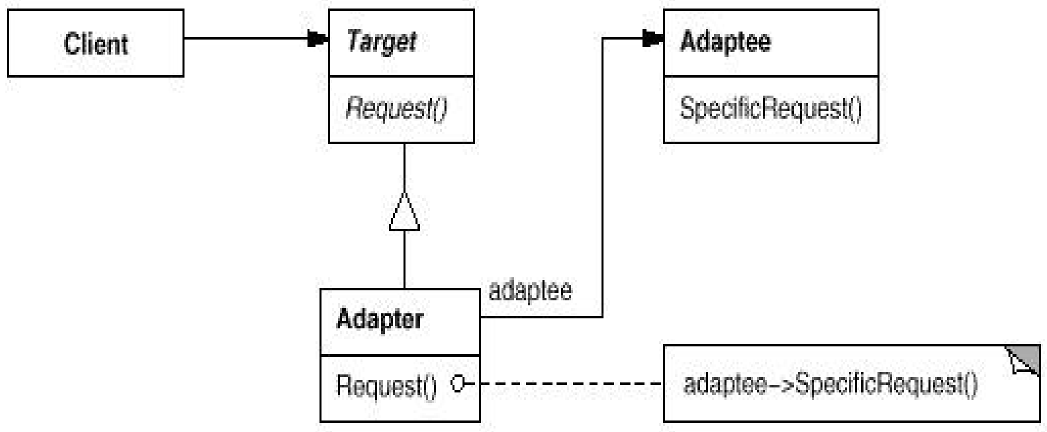
   Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

2. Motivation

   a. Sometimes a toolkit or class library cannot be used because its interface is incompatible with the interface required by an application
   b. We cannot change the library interface
   c. Even if we did have the source code, we should not change the library

3. Structure

- An adapter relies on object composition



4. Participants

a. Target
   - Defines the domain-specific interface that Client uses.

b. Adaptee
   - Defines an existing interface that needs adapting.

c. Adapter
   - Adapts the interface of Adaptee to the Target interface.

d. Client
   - Collaborates with objects conforming to the Target interface.

5. Applicability

Use the Adapter pattern when:
You want to use an existing class that performs similar functions but with a different interface. For example, a third-party report generator based on xml input.

6. How to implement the Adapter Pattern?

a. Target interface

```
public interface Target {
	public void push(String str);
	public String pop();
	public boolean isEmpty();
}
```

b. Adapter

```
public class Adapter implements Target{
	private Adaptee adaptee;

	@Override
	public void push(String str) {
		adaptee.add(str);

	}

	@Override
```

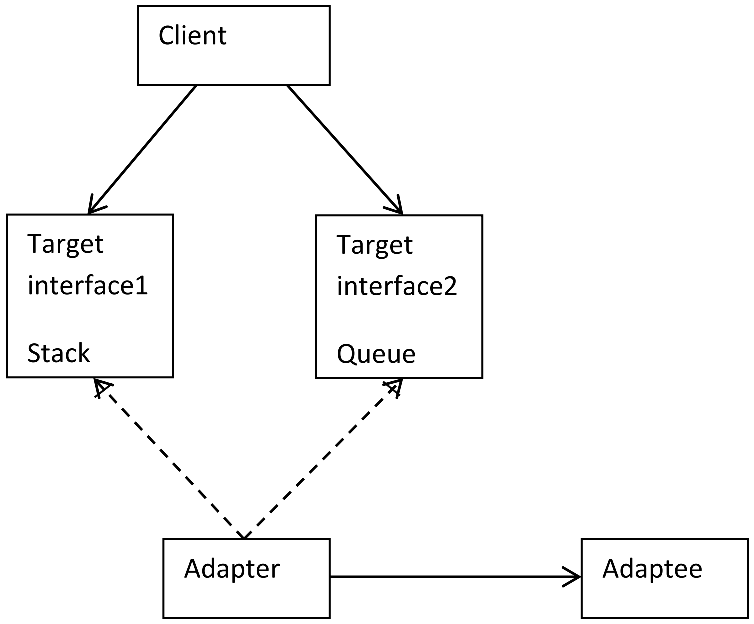
```
		public String pop() {
			int end = adaptee.getEnd();
			String str = adaptee.get(end);
			adaptee.remove(end);
			return str;
		}

		@Override
		public boolean isEmpty() {
			return adaptee.empty();
		}

	}
```

c. Adaptee

```
	public class Adaptee {
		private String[] data;
		private int start;
		private int end;

		public String startString(){
			return data[start];
		}

		public String endString(){
			return data[end];
		}

		public boolean empty(){
			return ( end == -1 );
		}

		public void add(String str){
			data[end] = str;
			end++;
		}

		public void remove(int pos){
			//remove the String object at position 'pos' and bring
			//forward all items after it

			for ( int i = pos; i < end; i++ ){
				data[i] = data[i+1];
			}
		}

		public String get(int pos){
			return data[pos];
		}

		…
```

Lab 3-1

Extend the above sample code to implement a 2-way adapter that can be used both as a Stack and a Queue.



The Iterator Pattern

1. Intent

   Provide a way to access the elements of an aggregate object (collection) sequentially without exposing its underlying representation

2. Motivation

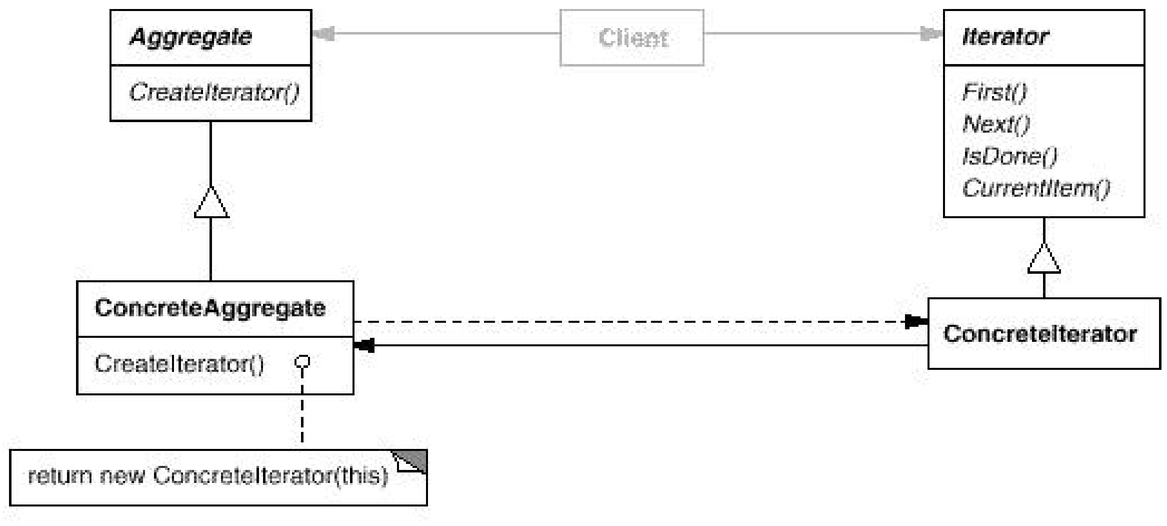
   a. An aggregate object such as a list or hash map should allow a way to traverse its elements without exposing its internal structure.
   b. It should allow different traversal methods depending on what the client needs (for example by using a functor).
   c. It should allow concurrent access by multiple threads.
   d. But we want to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object.

3. Applicability

   Use the Iterator pattern:

   a. To support traversals of aggregate objects without exposing their internal representation.
   b. To support multiple, concurrent traversals of aggregate objects.
   c. To provide a uniform interface for traversing different aggregate structures to support polymorphic iteration.

4. Structure



5. Participants

a. Iterator
   - defines an interface for accessing and traversing elements.

b. ConcreteIterator
   - implements the Iterator interface.
   - keeps track of the current position in the traversal of the aggregate.

c. Aggregate
   - defines an interface for creating an Iterator object.

d. ConcreteAggregate
   - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

6. Consequences

Simplifies the interface of the aggregate by not polluting it with traversal methods
Supports multiple, concurrent traversals
Supports variant traversal techniques

7. How to implement the Iterator pattern?

Aggregate (collection) interface

```
public interface Aggregate {
 public Iterator getIterator();
}
```

Iterator interface

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Concreate aggregate class with a nested concreate iterator class

```
public class NameRepository implements Aggregate {
         private String names[] = {"Rob" , "Jon" ,"Jul" , "Lor", "Pat",
"Ken"};

         //other methods of the NameRepository
         …

         @Override
         public Iterator getIterator() {
            return new NameIterator();
         }

         private class NameIterator implements Iterator {

            int index;

            @Override
            public boolean hasNext() {

               if(index < names.length){
                  return true;
            }
            return false;
        }

        @Override
        public Object next() {

            if(this.hasNext()){
                return names[index++];
            }
            return null;
        }
    }
}
```

Lab 3-2

Suppose the name repository in the above example uses a 2-dimensional array to store the names. Names can be dynamically added or removed from it. When you remove a name, you simply replace the name with a “-”. (You do not need to implement the add/remove methods though). Rewrite the NameIterator class that implements the same Iterator interface. But make sure that a “-” is never returned by the next() method.

The Proxy Pattern

1. Intent

   Provide a surrogate or placeholder for another object to control access to it.

2. Motivation

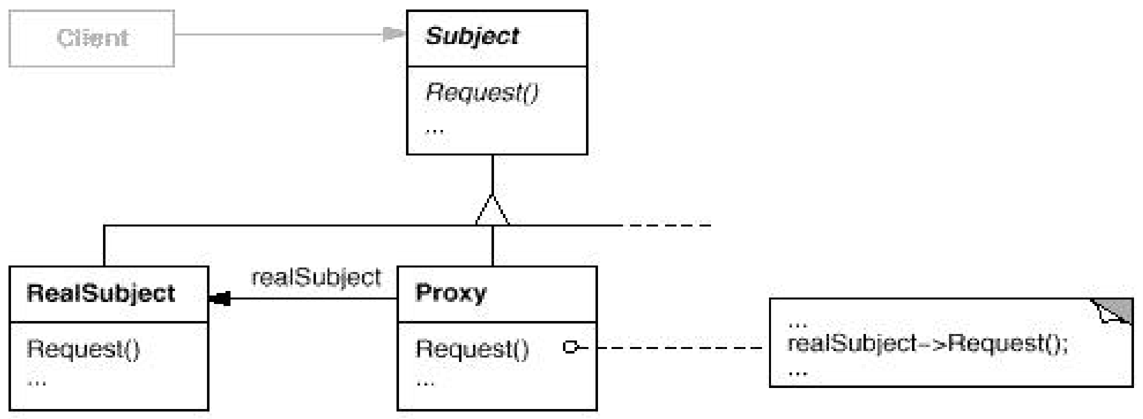
   a. To defer the full cost of its creation and initialization until we actually need to use it.
   b. There are situations in which a client does not or cannot reference an object directly, but wants to still interact with the object.
   c. A proxy object can act as the intermediary between the client and the target object.

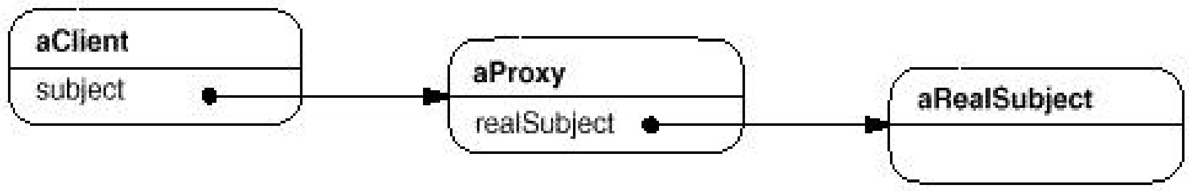
3. Applicability

   Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several situations where the Proxy pattern is applicable:

   a. Remote Proxy - Provides a reference to an object located in a different address space on the same or different machine.
   b. Protection (Access) Proxy - Provides different clients with different levels of access to a target object.
   c. Cache Proxy - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results.

4. Structure





5. Participants

a. Proxy

- maintains a reference that allows the proxy to access the real subject.
- provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
- controls access to the real subject and may be responsible for creating and deleting it.
- other responsibilities depend on the kind of proxy: **Remote proxies** are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space;

Virtual proxies may cache additional information about the real subject so that they can postpone accessing it; **Protection proxies** check that the caller has the access permissions required to perform a request.

b. Subject

- defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

c. RealSubject

- defines the real object that the proxy represents.

6. How to implement the Proxy Pattern

a. Subject interface

```
public interface ICommandExecutor {
      public void runCommand(String cmd) throws Exception;
}
```

b. Proxy class that implements the subject interface

```
public class CommandExecutorProtectionProxy implements ICommandExecutor
{

    private boolean isAdmin;
    private ICommandExecutor executor;

    public CommandExecutorProtectionProxy(String user, String pwd){
        if("username".equals(user) && "password".equals(pwd))
            isAdmin = true;
        executor = new CommandExecutor();
    }

    @Override
    public void runCommand(String cmd) throws Exception {
        if(isAdmin){
            executor.runCommand(cmd);
        }else{
            if(cmd.trim().startsWith("del")){
                throw new Exception("'delete' command is not allowed
for non-admin users.");
            }else{
                executor.runCommand(cmd);
                }
            }
        }

    }
```

c. Real subject class

```
class CommandExecutor implements ICommandExecutor {

    @Override
    public void runCommand(String cmd) throws IOException {
        Runtime.getRuntime().exec(cmd);
        System.out.println("'" + cmd + "' command executed.");
    }

}
```

7. Real scenarios where Proxy Pattern is used

Scenario 1: 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. What can we do to improve the situation?

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.
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. 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!

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. 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 deep-cloned.

Scenario 2: 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.

Scenario 3: 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: A Synchronization Proxy, which uses a locking mechanism to control the number of clients that simultaneously access the server or the real object.

Scenario 4: 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.
Even the proxy cannot 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.

Scenario 5: A machine at the College 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, etc.

Lab 3-3

There are 2 questions in this lab. You do not need to complete both. Just choose either one for submission.

Choice 1: Implement a synchronization proxy discussed in Scenario 3. Suppose the Table class implements the ITable interface

```
public interface ITable {
      public int numOfRows();
      public IRow getRow(int rowNum);
      public void addRow(IRow row, int rowNum); //add row at the end
      public void modifyRow(int rowNum, IRow row);
      public void deleteRow(int rowNum);
}

public interface IRow {
      //the interface a concrete Row class implements ...
}
```

Choice 2: Provide an implementation for Solution 3 discussed in Scenario 1. Suppose the real subject is a HashTable. Write the proxy class LargeHashtable that keeps track of client access. In case, any client program tries to modify the real subject, the proxy will provide a cloned copy (deep) to the client to work with. You do not need to provide a client for the lab.
Note: The default implementation for HashTable's clone() method provides a shallow copy only.