@Copyright to Java World (https://www.javaworld.com/article/2076233/java-se/explore-the-dynamic-proxy-api.html)

With the introduction of the Dynamic Proxy API in Java 1.3, a huge and often overlooked improvement has been made to the Java platform. The uses for dynamic proxies are sometimes hard concepts to grasp. In this article, I hope to introduce you first to the Proxy design pattern and then to the java.lang.reflect.Proxy class and the java.lang.reflect.InvocationHandler interface, which make up the heart of Dynamic Proxy's functionality.

The functionality discussed here combines the Java 1.3 dynamic proxies with abstract data types to bring strong typing to those types. I will also discuss the power of the dynamic proxy by introducing the concept of views in your Java programming. Lastly, I will introduce a powerful way to add access control to your Java objects with, of course, the use of the dynamic proxy.

**Definition of a proxy**

A proxy forces object method calls to occur indirectly through the proxy object, which acts as a surrogate or delegate for the underlying object being proxied. Proxy objects are usually declared so that the client objects have no indication that they have a proxy object instance.

Some common proxies are the access proxy, facades, remote proxies, and virtual proxies. An access proxy is used to enforce a security policy on access to a service or data-providing object. A facade is a single interface to multiple underlying objects. The remote proxy is used to mask or shield the client object from the fact that the underlying object is remote. A virtual proxy is used to perform lazy or just-in-time instantiation of the real object.

The proxy is a fundamental design pattern that is used quite often in programming. However, one of its drawbacks is the specificity or tight coupling of the proxy with its underlying object. Looking at the UML for the Proxy design pattern in Figure 1, you see that for the proxy to be useful and transparent, it usually needs to either implement an interface or inherit from a known superclass (with the exception of a facade, perhaps).

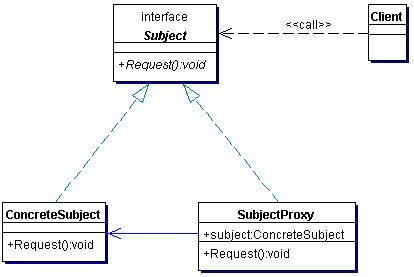


Figure 1. UML Diagram of the Proxy Design Pattern

**Dynamic proxies**

In Java 1.3, Sun introduced the Dynamic Proxy API. For the dynamic proxy to work, you must first have a proxy interface. The proxy interface is the interface that is implemented by the proxy class. Second, you need an instance of the proxy class.

Interestingly, you can have a proxy class that implements multiple interfaces. However, there are a few restrictions on the interfaces you implement. It is important to keep those restrictions in mind when creating your dynamic proxy:

1. The proxy interface must be an interface. In other words, it cannot be a class (or an abstract class) or a primitive.
2. The array of interfaces passed to the proxy constructor must not contain duplicates of the same interface. Sun specifies that, and it makes sense that you wouldn't be trying to implement the same interface twice at the same time. For example, an array { IPerson.class, IPerson.class } would be illegal, but the code { IPerson.class, IEmployee.class } would not. The code calling the constructor should check for that case and filter out duplicates.
3. All the interfaces must be visible to the ClassLoader specified during the construction call. Again, that makes sense. The ClassLoader must be able to load the interfaces for the proxy.
4. All the nonpublic interfaces must be from the same package. You cannot have a private interface from package com.xyz and the proxy class in package com.abc. If you think about it, it is the same way when programming a regular Java class. You couldn't implement a nonpublic interface from another package with a regular class either.
5. The proxy interfaces cannot have a conflict of methods. You can't have two methods that take the same parameters but return different types. For example, the methods public void foo() and public String foo() cannot be defined in the same class because they have the same signature, but return different types (see [*The Java Language Specification*](https://www.javaworld.com/article/2076233/java-se/explore-the-dynamic-proxy-api.html#resources)). Again, that is the same for a regular class.
6. The resulting proxy class cannot exceed the limits of the VM, such as the limitation on the number of interfaces that can be implemented.

To create an actual dynamic proxy class, all you need to do is implement the java.lang.reflect.InvocationHandler interface:

public Class MyDynamicProxyClass implements

java.lang.reflect.InvocationHandler

{

Object obj;

public MyDynamicProxyClass(Object obj)

{ this.obj = obj; }

public Object invoke(Object proxy, Method m, Object[] args) throws

Throwable

{

try {

// do something

} catch (InvocationTargetException e) {

throw e.getTargetException();

} catch (Exception e) {

throw e;

}

// return something

}

}

That's all there is to it! Really! I'm not lying! Okay, well, you also have to have your actual proxy interface:

public interface MyProxyInterface

{

public Object MyMethod();

}

Then to actually use that dynamic proxy, the code looks like this:

MyProxyInterface foo = (MyProxyInterface)

java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(),

Class[] { MyProxyInterface.class },

new MyDynamicProxyClass(obj));

Knowing that the above code is just horribly ugly, I'd like to hide it in some type of factory method. So instead of having that messy code in the client code, I'll add that method to my MyDynamicProxyClass:

static public Object newInstance(Object obj, Class[] interfaces)

{

return

java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(),

interfaces,

new

MyDynamicProxyClass(obj));

}

That allows me to use the following client code instead:

MyProxyInterface foo = (MyProxyInterface)

MyDynamicProxyClass.newInstance(obj, new Class[]

{ MyProxyInterface.class });

That is much cleaner code. It might be a good idea in the future to have a factory class that completely hides the entire code from the client, so that the client code looks more like:

MyProxyInterface foo = Builder.newProxyInterface();

Overall, implementing a dynamic proxy is fairly simple. However, behind that simplicity is great power. That great power is derived from the fact that your dynamic proxy can implement any interface or interface group. I'll explore that concept in the next section.

## Abstract data

The best example of abstract data is in the Java collection classes such as

java.util.ArrayList

,

java.util.HashMap

, or

java.util.Vector

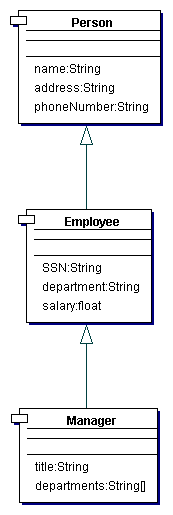
. Those collection classes are capable of holding any Java object. They are invaluable in their use in Java. The concept of abstract data types is a powerful one, and those classes bring the power of collections to any data type.

## Tying the two together

By combining the concept of dynamic proxies with abstract data types, you can gain all the benefits of abstract data types with strong typing. In addition, you can easily use the proxy class to implement access control, virtual proxies, or any other useful proxy type. By masking the actual creation and use of proxies from the client code, you can make changes to the underlying proxy code without affecting the client code.

## The concept of a view

When architecting a Java program, it is common to run into design problems in which a class must display multiple, different interfaces to client code. Take Figure 2 for example:



public class Person {

private String name;

private String address;

private String phoneNumber;

public String getName() { return name; }

public String getAddress() { return address; }

public String getPhoneNumber() { return phoneNumber; }

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

public void setAddress(String address) { this.address = address; }

public void setPhoneNumber(String phoneNumber) { this.phoneNumber =

phoneNumber; }

}

public class Employee extends Person {

private String SSN;

private String department;

private float salary;

public String getSSN() { return ssn; }

public String getDepartment() { return department; }

public float getSalary() { return salary; }

public void setSSN(String ssn) { this.ssn = ssn; }

public void setDepartment(String department) { this.department =

department; }

public void setSalary(float salary) { this.salary = salary; }

}

public class Manager extends Employee {

String title;

String[] departments;

public String getTitle() { return title; }

public String[] getDepartments() { return departments; }

public void setTitle(String title) { this.title = title; }

public void setDepartments(String[] departments) { this.departments =

departments; }

}

In that example, a Person class contains the properties Name, Address, and PhoneNumber. Then, there is the Employee class, which is a Person subclass and contains the additional properties SSN, Department, and Salary. From the Employee class, you have the subclass Manager, which adds the properties Title and one or more Departments for which Manager is responsible.

After you've designed that, you should take a step back and think about how the architecture is to be used. Promotion is one idea that you might want to implement in your design. How would you take a person object and make it an employee object, and how would you take an employee object and make it a manager object? What about the reverse? Also, it might not be necessary to expose a manager object as anything more than a person object to a particular client.

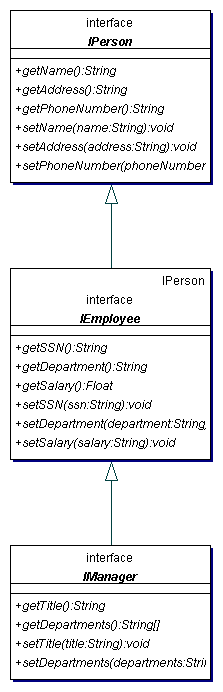
A real-life example might be a car. A car has your typical interface such as a pedal for accelerating, another pedal for braking, a wheel for turning left or right, and so forth. However, another interface is revealed when you think of a mechanic working on your car. He has a completely different interface to the car, such as tuning the engine or changing the oil. In that case, to expect the car's driver to know the car's mechanic interface would be inappropriate. Similarly, the mechanic wouldn't need to know the driver interface, although, I'd like him to know how to drive. That also means that any other car with the same driver interface is easily interchangeable, and the driver of the car doesn't have to change or learn anything new.

Of course in Java, the concept of an interface is used quite often. One might ask how dynamic proxies tie into that use of interfaces. Put simply, dynamic proxies allow you to treat any object as any interface. Sometimes mapping is involved, or the underlying object might not quite match the interface, but overall, that concept can be quite powerful.

Similar to the car example above, you could have a Bus interface that has a different, but similar BusDriver interface. Most people, who know how to drive a car, know mostly what is needed to drive a bus. Or you could have a similar BoatDriver interface but instead of pedals, you have the concept of a throttle and, instead of braking, you have reverse throttle.

In the case of a BusDriver interface, you could probably use a direct map of the Driver interface onto the underlying Bus object and still be able to drive the bus. The BoatDriver interface would most likely call for a mapping of the pedal and brake methods to the throttle method of the underlying Boat object.

By using an abstract data type to represent the underlying object, you can simply put a Person interface onto the data type, fill in person fields, and subsequently come in after that person is hired and use the Employee interface on the same underlying object. The class diagram now looks like Figure 3:



public interface IPerson {

public String getName();

public String getAddress();

public String getPhoneNumber();

public void setName(String name);

public void setAddress(String address);

public void setPhoneNumber(String phoneNumber);

}

public interface IEmployee extends IPerson {

public String getSSN();

public String getDepartment();

public Float getSalary();

public void setSSN(String ssn);

public void setDepartment(String department);

public void setSalary(String salary);

}

public interface IManager extends IEmployee {

public String getTitle();

public String[] getDepartments();

public void setTitle(String title);

public void setDepartments(String[] departments);

}

public class ViewProxy implements InvocationHandler

{

private Map map;

public static Object newInstance(Map map, Class[] interfaces)

{

return Proxy.newProxyInstance(map.getClass().getClassLoader(),

interfaces,

new ViewProxy(map));

}

public ViewProxy(Map map)

{

this.map = map;

}

public Object invoke(Object proxy, Method m, Object[] args) throws

Throwable

{

Object result;

String methodName = m.getName();

if (methodName.startsWith("get"))

{

String name = methodName.substring(methodName.indexOf("get")+3);

return map.get(name);

}

else if (methodName.startsWith("set"))

{

String name = methodName.substring(methodName.indexOf("set")+3);

map.put(name, args[0]);

return null;

}

else if (methodName.startsWith("is"))

{

String name = methodName.substring(methodName.indexOf("is")+2);

return(map.get(name));

}

return null;

}

}

In that example, the first interface lines are simply for implementing the Person/Employee/Manager objects in the inheritance example above. The real magic occurs in the invoke method of the ViewProxy class.

The ViewProxy implements the java.lang.reflect.InvocationHandler interface, which is used by the java.lang.reflect.Proxy class to provide the actual proxy implementation. The view proxy provides a builder method, newProxyInstance(), which creates the java.lang.reflect.Proxy and ViewProxy objects necessary for the whole proxy implementation.

For that to work, you would write code such as:

HashMap identity = new HashMap();

IPerson person = (IPerson)ViewProxy.newInstance(identity, new Class[]

IPerson.class });

person.setName("Bob Jones");

You can then just as easily convert your identity to an employee:

IEmployee employee = (IEmployee)ViewProxy.newInstance(identity, new Class[]

{ IEmployee.class });

employee.setSSN("111-11-1111");

You can subsequently call IPerson methods on your IEmployee object:

System.out.println(employee.getName())

That would print the line:

Bob Jones

Obviously, the code needed to actually create the proxy instance isn't pretty, so it might be a good idea to use some sort of factory to create your actual objects (making the proxy creation transparent to the client code). By doing that, you can simply add or change interfaces as needed in future programming, without having to make changes to other code.

Let's make a little change to the ViewProxy class. There are cases in which I have an object that has some of the same methods of an interface but doesn't directly implement that interface. For example, I could have a concrete Person object, and I want to treat it as an IPerson interface. It just so happens that the Person class has all the methods of IPerson. So to convert the Person class to an IPerson interface, my new ViewProxy would look like this:

public class ViewProxy implements InvocationHandler

{

private Map map;

private Object obj;

public static Object newInstance(Map map, Object obj, Class[] interfaces)

{

return Proxy.newProxyInstance(map.getClass().getClassLoader(),

interfaces,

new ViewProxy(new map, obj));

}

public ViewProxy(Map map, Object obj)

{

this.map = map;

this.obj = obj;

}

public Object invoke(Object proxy, Method m, Object[] args) throws

Throwable

{

try {

return m.invoke(obj, args);

} catch (NoSuchMethodException e)

{ // ignore }

Object result;

String methodName = m.getName();

if (methodName.startsWith("get"))

{

String name = methodName.substring(methodName.indexOf("get")+3);

return map.get(name);

}

else if (methodName.startsWith("set"))

{

String name = methodName.substring(methodName.indexOf("set")+3);

map.put(name, args[0]);

return null;

}

else if (methodName.startsWith("is"))

{

String name = methodName.substring(methodName.indexOf("is")+2);

return(map.get(name));

}

return null;

}

}

In the modified code, I've added a try-catch block that tries to call the method on the underlying object that is passed in. If that fails, then you can switch over to the normal HashMap method. Now, if you want to convert a Person class to an IPerson class, you just need to write code like this:

Person person;

HashMap map;

IPerson ip = (IPerson)ViewProxy.newInstance(map,

person,

new Class[] { IPerson.class });

Now, you can treat the person like an IPerson object, and the underlying person is changed accordingly. Similarly, you can treat this Person class like an IEmployee object as well:

Person person;

HashMap map;

IEmployee ie = (IEmployee)ViewProxy.newInstance(map,

person,

new Class[]

IEmployee.class});

Now, whenever you call IPerson methods, the underlying person is changed. However, the IEmployee calls will be changed in the HashMap of your ViewProxy. A little dangerous, perhaps, but then again, so is me driving a bus.

## Authorization implementation

Access control is often hard to implement, especially when it is added after the code has already been written and running. I've had many sleepless nights because of access control problems. However, by using the dynamic proxy, you can easily implement access control.

Access control is usually discussed in terms of coarse-grained and fine-grained access control. For coarse-grained access control, you allow a certain level of access to a whole object or object group. Fine-grained access control is usually handled at the method or property level. For example, allowing read-only access to a file is an example of coarse-grained control. Allowing access to write to only certain lines of a file is an example of fine-grained control. A good access-control implementation allows both coarse-grained and fine-grained control. With the dynamic proxy, it is easy to implement both.

To implement coarse-grained control, it is simply a matter of having a read-only interface for the implementation:

public interface IpersonRO {

public String getName();

public String getAddress();

public String getPhoneNumber();

}

public interface IemployeeRO extends IPersonRO {

public String getSSN();

public String getDepartment();

public Float getSalary();

}

public interface IManagerRO extends IEmployeeRO {

public String getTitle();

public String[] getDepartments();

}

That can be limiting in that the client code has to specifically know that it is getting a read-only object. Perhaps a better way to handle that is through the InvocationHandler, which gives the developer the ability to grant or deny access by using the java.lang.security.acl package, and enables the implementation of fine-grained access control:

public class ViewProxy implements InvocationHandler

{

public static final Permission READ= new PermissionImpl("READ");

public static final Permission WRITE = new PermissionImpl("WRITE");

private Map map;

public static Object newInstance(Map map, Class[] interfaces)

{

return Proxy.newProxyInstance(map.getClass().getClassLoader(),

interfaces,

new ViewProxy(map));

}

public ViewProxy(Map map)

{

this.map = map;

}

public Object invoke(Object proxy, Method m, Object[] args) throws

Throwable

{

Object result;

String methodName = m.getName();

if (methodName.startsWith("get"))

{

if (!acl.checkPermission(p1, read)) return null;

String name = methodName.substring(methodName.indexOf("get")+3);

return map.get(name);

}

else if (methodName.startsWith("set"))

{

if (!acl.checkPermission(p1, write)) return null;

String name = methodName.substring(methodName.indexOf("set")+3);

map.put(name, args[0]);

return null;

}

else if (methodName.startsWith("is"))

{

if (!acl.checkPermission(p1, read)) return null;

String name = methodName.substring(methodName.indexOf("is")+2);

return(map.get(name));

}

return null;

}

}

## Conclusion

By using the Dynamic Proxy API introduced in Java 1.3, you're presented with a whole range of possibilities. Not only in the typical uses of a dynamic proxy, such as debugging code or writing event handlers for Swing, but also in the underlying principles of abstract data types and the ability to map a view onto another object.

There are many other uses for the dynamic proxy that I haven't touched in this article, such as virtual proxies, in which the actual class isn't loaded until it is needed. You could also use that in conjunction with a remote proxy, which disguises the fact that the object is located remotely across the network. Use your imagination, and the possibilities are endless.