Facet: A pattern for dynamic interfaces

Author:

Eric Crahen
CSE Department
201 Bell Hall
<crahen@cse.buffalo.edu>

SUNY at Buffalo
Amherst, NY 14260
716-645-3180

Context:

Wherever it is desirable to create a context sensitive interface in order to modify or control the apparent behavior if an object.

Problem:

How can I modify the behavior of an existing object so that different behaviors are shown under different circumstances; and yet still maintain a clean separation between the policy (when each behavior is available) and implementation of each behavior allowing them to be developed independently of one another?

Forces:

Interfaces provide an essential level of abstraction to object oriented programming. Generally, objects are able define aspects of their function very well using interfaces. At times, this may not be enough. When dealing with two or more classes whose responsibilities are distinctly separate, but whose behavior is closely related, classes can begin to resist modularization. For example, adding security to an application means inserting code that performs security checks into numerous locations in existing classes. Clearly, these responsibilities are distinct; the original classes being responsible for the tasks they were designed to perform, and the security classes being responsible for providing access control. However, making those original classes secure means intermingling code that deals with security. When the classes dealing with security are changed many classes are going to be impacted.

One method of adding new behavior to an existing class might be to simply create a subclass and embed that new behavior. In the security example, this would mean creating a subclass for each class that needs to be secure and adding calls to the security code. For a small project, this might not be too bad; but in any project with a large number of objects, this mean creating a lot of new classes and repeating a lot of the same code. Not only can this quickly become quite difficult to maintain, but it also present problems dealing with change. Suppose the interface to those security classes is changed or are replaced with a new set of security classes; all that code will need to be modified to reflect those changes, even if they are very small. Creating a direct dependency between the existing and security classes is something that would be beneficial to avoid.

A simple solution for decouple that code would be to use a proxy. This would eliminate the some of the dependence between the security and existing classes making things a little more resilient to change. It may also be helpful in reducing the number of new classes that would need to be created and maintained. However, using a proxy is still going to require spending a lot of time creating proxy classes and maintaining those classes. Especially in scenarios where the security policy could be subject to change, this could potentially mean that different kinds of security measures should be taken for different kinds of actions (methods being invoked). Additionally, new actions might need to be taken as well. For example, an administrator may want to enable or disable logging mechanisms to monitor various portions of an application. The proxies, being 'fixed' in nature, would need to be updated to reflect these changes. Using a proxy alone really just shifts the focus of the original problem from the client code to the proxy code.

The Decorator pattern is offers a method of wrapping existing objects, adorning them with various behaviors. This would potentially allow a programmer to eliminate the problem of having to rewrite each class when ever there is a change in policy. It is also useful for adding new behaviors when there is a change in policy. For instance, to enable logging an object could be attached to the decorator that would allow the behavior to change at runtime. This is nice since is allows a client to add new behaviors at runtime. But it requires an educated guess ahead of time about which methods might exhibit which behaviors. All the methods of an object might not all require the same behaviors, different methods may have different requirements. Using a Decorator leaves the programmer with the task of creating the delegation code and with adding some kind of behavior selection framework.

An alternative to the Decorator pattern is the Strategy pattern. Like the Decorator this will allow some extra behavior to be grafted onto object. The difference is that a Strategy may do this internally, by being embedded into some kind of wrapper class. This places the responsibility of selecting a behavior on the strategy classes. This still leaves a programmer with adding a lot of code, a programmer now must create some way to manage how a strategy is selected. The client code would now be dependant on the wrapper classes which would be dependant on some rigid set of Strategy classes. Unrelated strategies might have different methods of selection; this can lead needing to create a framework to deal with strategy selection. Ultimately, this might be less flexible, and it still doesn't eliminate the problem of creating and maintaining those wrapper classes and including code that interacts with the correct Strategy classes.

A limitation in using existing wrapper patterns a programmer commits themselves to maintaining a large amount of repetitive code. Much of the work involved in using these patterns is in ensuring the delegation mechanisms are correct. Each method that performs some delegation is a point of dependency. Any moderate change could potentially trigger a sizable maintenance task. Each wrapper help shift the dependency out of the client code and into the wrapper code, but it doesn't really eliminate it. This is one obstacle that needs to be dealt with, especially for code that is intended for a changing environment.

The main drawback to all of these alternatives is that a programmer must be able to predict when it is appropriate to enable a behavior. These patterns all wrap an object as whole; they don't necessarily have the ability to see what the object is composed of or how it is being used, which leaves the programmer to looking for answers to some questions. Should all of the delegating methods exhibit the same behaviors or only some of them? How does a decorator decide when it is appropriate to exhibit each behavior? How can that behavior be altered to deal with new situations without having a massive impact?

An object implementing several interfaces is exposing several aspects of itself. These aspects might reflect different facets of its behavior and so it makes sense that method belonging to interfaces might require different kinds of behaviors. This is a good start to finding some hint about what a method is being used for, but even this may not be enough. A large part of defining appropriateness comes not only from static information (such as what an object or interface is for) but it also comes from dynamic information (such as how an object is being used). For instance, as a person it might be acceptable to speak in very loud voice if you giving a talk in auditorium, but in a small classroom shouting might not be the best thing to do. Similarly, a security policy could require authentication for various reasons, authenticating with a domain might be desirable if they are using a Microsoft platform, but a different method could be required if that code is access from a PDA.

Introducing an extendable notion of context allows a user to refine this even more, giving them the ability to not only define *what* behaviors are appropriate but also *when* they are appropriate. Just as component abstracts system details, allowing developers to focus on writing business logic, providing a method for abstracting integration and access control details allows developers to focus on creating classes with distinct responsibilities not on focusing the tedious details created by their interactions. Classes can be evolved independently of one another but still work together without creating a burden for the programmer. It also allows existing classes to be outfitted with new behavior allowing them to adapt a changing environment.

Solution:

If the restructuring the related code that must interact is not going to help in decoupling it, we can refine the code used to join these classes together and make it as flexible as possible. This is done by using a special kind of proxy called a Facet. A Facet helps shift the concerns revolving how to integrate the closely related classes. making them an implementation detail rather than a design issue. This not only allows each part of your code to be developed separately, but it also allows each component to be reused and recombined with other components in other Facets.

Resulting Context:

The implementation of a Facet consists mainly of two major components, a proxy and a Sentry. The proxy implements a set of interfaces and it is exposed to client. This proxy can be cast to one of these interfaces, exposing a particular set of functionality, or *Facet*, of the Sentry it is protecting. When one of these methods is invoked, it will be delegated to the Sentry, which considers the interface and context being used to make the request and selects one of it comprising objects to handle it. Each of these things will be explained below.

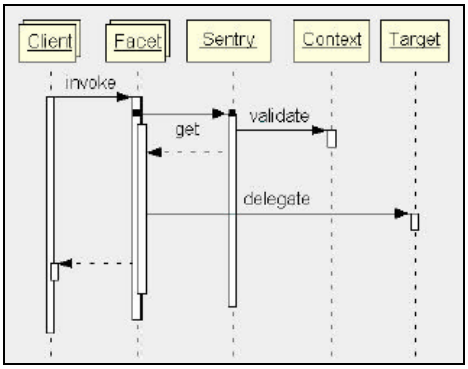

Figure 2. Sequence Diagram

The proxy, called a Facet, is not a typical proxy object. A Facet is a specialization of an abstract *Facet*[1] class that implements no interfaces and has no instance methods. A user never defines a subclass of *Facet*, but an instance can be obtained (typically through some factory method) that implements some unknown set of interfaces. From the users perspective a Facet appears to be an object with the special property of being castable to any interface. A client is given a reference, rather than a direct reference to an the delegation target which the Facet which guards.

When a method is invoked on a Facet instance, the Sentry associated with that Facet is consulted. That Sentry is responsible for selecting a valid delegation target for the method being invoked. In general, a Sentry will simply contain a set of mappings between interfaces and a delegation targets. Any Sentry is free to make its selection using whatever method is appropriate; it is not limited to using a single method. It is even possible for a Sentry to use an object pool from which similar objects could be selected to fill requests for lengthy or intensive operations.

When a Sentry has selected a delegation target, it must first validate that object before the method invocation can proceed. This task is handed off to a Context[2] object associated with the Sentry. A Context abstracts the logic for make access control decisions on behalf of a Facet. It is possible to implement Context classes that consider any number of things when making this choice. This can be anything ranging from something intrinsic to a clients process or thread, such as security information, to something as simple as counting the number accesses to a particular object. Context makes it very easy to guard any object with an interface level check.

Once a delegation target has finally been selected and validated, the Facet will then delegate the method call directly to that object. The client will be able to receive from that object, and pass the Facet

[1] See Listing 1 for a simple definition of the Facet class

[2] See Listing 2 for a definition of the Context interface

reference to other objects without ever having any direct access to the guarded object. The Sentry is free to select different delegation targets without ever requiring the client to do anything with the Facet reference it is holding. Similarly, the Context is free to change or revoke access to an object any time as well.

A Client must have some method to determine what interfaces might be available given a Facet instance. The *Facet* class contains a set of static methods that address this and other needs. The *query*[3] method allows all interfaces exposed through a particular Facet instance to the current context to be enumerated. In addition to discovering all interfaces, this method can be used to test for just a particular set of interfaces.

A Client must have some method to deal with changes the set of exposed interfaces. An instance of a Facet that implements a certain interface may exist and be in use. The class that was generated for that Facet may still implement that interface, even after the support for it has been revoked at a later point.

Similarly, a Facet instance may exist and be in use by a client and a new interface is suddenly exposed through it. The client needs to be able to use that interface, even if the class for the particular Facet instance held by does not implement that interface. A method called narrowing[4] addresses this issue. Narrowing a Facet announces the intention to use a particular interface of a *Facet*. This allows the object to perform a check to make sure it is a valid operation, as well as providing a method to generation a new *Facet* class in order top support the requested interfaces, where it is appropriate to do so. Any narrowed *Facet* can safely be cast to any one of the interfaces that were used in narrowing that *Facet*.

The configuration process of a *Facet* is interesting because it used the capabilities that have been described above. A *Facet* can be narrowed and cast to a *Manipulator*[5] interface, which defines a simple protocol for making changes to a Facet. The *Manipulator* modifies the guarded Sentry, allowing new objects and interfaces to be associated with the *Facet*. In addition to adding new interfaces to a Facet, old interfaces can be changed or revoked.

Known Uses:

This pattern has been used in the design of location and space based frameworks for mobile computing [BR02].

[3] See Listing 3 for an example of using the query method
[4] See Listing 4 for an example of using the narrow method
[5] See Listing 3 for an example of a Manipulator interface

Sample Code and Usage:

A complete reference implementation for Java is available online at :
http://www.cse.buffalo.edu/~crahen/projects/polymorph

Listings:

```
.
public abstract class Facet {

        public static Facet create()
                throws FacetException;

        public static Class[] query(Facet f, Class[] interfaces)
                throws Facet Exception;

        public static Facet narrow(Facet f, Class[] interfaces)
                throws Facet Exception;

}
```

Listing 1

```
interface Context {

        public Object validateComponent(Class interfaceClass, Object delegate);

        public boolean validateInterface(Class interfaceClass);

}
```

Listing 2

```
Facet f = Facet.create();

Class[] availableInterfaces = f.query(null);
```

Listing 3

```
f = f.narrow(new Class[]{AnInterface.class});

AnInterface a = (AnInterface)f;
```

Listing 4

A simple example:

```
// A simple Account class to which you might want to add security

interface MutableAccount {

  void credit(long amount);

  void debit(long amount);

}

interface ImmutableAccount {

  long balance();

}

class Account implements MutableAccount, ImmuatbleAccount {

  void credit(long amount) {
    // ...
  }

  void debit(long amount)  {
    // ...
  }

  long balance() {
    // ...
  }

}

// Create a Context that will determine when its appropriate to use an interface

class AccountContext extends DefaultContext {

  private Permission accountModificationPermission
       = new AccountPermission(…..);

  public boolean validateInterface(Class interfaceClass) {

    // Perform a security check before allowing any changes to an Account
    if(interfaceClass.equals(MutableAccount.class)) {

        // Security Check, substitute whatever you'd like to use, this uses
        // a simple AccessController check for clarity

        return AccessController.checkPermission(accountModificationPermission);

    }

    // Allow anyone immutable access
    if(interfaceClass.equals(ImmutableAccount.class))
      return true;

    return super.validateInterface(interfaceClass);

  }


}


// Create a Facet that will control access to the Account object at runtime

Sentry sentry = new DefaultSentry(new AccountContext());
Facet f = Facet.create(sentry, new Class[]{ImmutableAccount.class, MutableAccount.class});

// Make sure both interfaces are supported

f = f.narrow(new Class[]{ImmutableAccount.class, MutableAccount.class});

// Now f can be used as if it were an ImmutableAccount or ImmutableAccount, except
// that it will only delegate a method call to the real Account class when the
// correct permission is available.
```

See Also:

The Proxy/Surrogate [GoF] pattern provides a method to create a placeholder object that controls access to another.

The Decorator/Wrapper [GoF] pattern provides a method to add new responsibilities to an object without modifying or subclassing it.

The Role pattern [RWW97] associates new behaviors with object through association.

References:

[Cos98] Pascal Costanza. LAVA Design and Compiler. Diploma thesis, Computer Science Department III, University of Bonn. January 1998.

[GoF] Gamma, Helm, Johnson, Vlissides. *Design Patterns,* 1994.

[JSDK01] Sun MicroSystems. *Java 2 Proxy API documentation and implementation 1.3.1,* 2001. http://java.sun.com/j2se/1.3/docs/api/java/lang/reflect/Proxy.html

[KH98] Ralph Keller and Urs Holzle. *Binary Component Adaption. In Proceedings ECOOP '98.* Springer, 1998.

[LY98] Tim Lindhol, Frank Yellin. *The Java Virtual Machine Language Specification 2nd Edition,* 1998. http://java.sun.com/docs/books/vmspec

[RWW97] Dirk Riehle, Wolf Siberski, Martina Wulf. *Role Object,* 1997. http://www.riehle.org/papers/1997/plop-1997-role-object.pdf

[BR02] B.Ramamurthy, *Extending the Notion of Location in Mobile Space (LOM)*, sumbitted to Grace Hopper Celebration of Women in Computing 2002, Vancover, B.C, October 2002.