Fundamental Patterns

The patterns in this chapter are the most fundaments and most important to know. You will find these patterns used extensively in other patterns.

Delegation (When not to use Inheritance)

Synopsis

Delegation is a way of extending and reusing the functionality of a class by writing an additional class with additional functionality that uses instances of the original class to provide the original functionality.

Context

Inheritance is a common way of extending and reusing the functionality of a class. However, inheritance is inappropriate for many situations. For example:

- Inheritance is useful for capturing is-a-kind-of relationships because they are very static in nature. However, is-a-role-played-by relationships are awkward to model by inheritance. Instances of a class can play multiple roles. For example, consider the example of an airline reservation system.

  An airline reservation system will include such roles as passenger, ticket selling agent and flight crew. It is possible to represent this as a class called Person that has subclasses corresponding to these roles. The problem is that the same person can fill more than one of these roles. A person who is normally part of a flight crew can also be a passenger. Some airlines will sometimes float flight crew to the ticket counter. The means that the same person can fill any combination of these roles. To model this situation we would need seven subclasses for Person. The number of subclasses needed increases exponentially with the number of roles, so that 63 subclasses are needed to model 6 roles.

  A more serious problem is that the same person can play different combinations of roles at different times. Using inheritance will necessitate using multiple objects to represent the same person in order to capture changes in role.

- If you find there is a need for an object to be a different subclass of a class at different points in time, then it should not be a subclass of that class in the first place.

- If you find that a class is trying to hide a method or variable inherited from a superclass from other classes, then that class should not inherit from that subclass.

- Declaring a class that is related to a program's problem domain as a subclass of a utility class is usually not a good idea for two reasons:

  - When you declare a class to be a subclass of a class like `Vector` or `Hashtable`, you are running the risk that these classes that you do not control will change in an incompatible way in the future. Though it is a low risk, there is usually no corresponding benefit to offset it.

- When people write a problem domain specific class as a subclass of a utility class, the intent is usually to use the functionality of the utility class to implement problem domain specific functionality. The problem because it weakens the encapsulation of the problem domain class' implementation.

  Client classes that use the problem domain class may be written in a way that assumes the problem domain class is a subclass of the utility class. If the implementation of the problem domain changes in a way that results in its having a different superclass, those client classes that rely on its having its original superclass will break.

  An even more serious problem is that client classes can call the public methods of the utility superclass, which defeats its encapsulation.

Inappropriate use of inheritance is sufficiently common to classify as an anti-pattern.

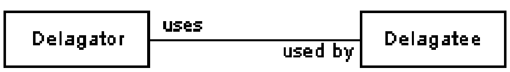
Forces

- Many or even possibly most reuse and extension of a class is not appropriately done through inheritance.

- By determining its superclass, a class' declaration determines the behavior that a class inherits from its superclass. Inheritance is not useful when the behavior that a class should build on is determined at run time.

Solution

Delegation is a way of reusing and extending the behavior of a class. It works writing a new class that incorporates the functionality of the original class by using an instance of the original class and calling its methods.



Delegation

Delegation is more general purpose than inheritance. Any extension to a class that can be accomplished by inheritance can also be accomplished by delegation.

Consequences

Delegation can be used without the problems that accompany inheritance. Another advantage of delegation is that it is easy to compose behavior at run time.

The main disadvantage of delegation is that it is less structured than inheritance. Relationships between classes built using delegation are less obvious than those built using inheritance. Here are some strategies for improving the clarity of delegation based relationships:

- Use well-known design and coding patterns. A person reading code that uses delegation will be more likely to understand the role that the objects play if the roles are part of a well know pattern or a pattern the recurs frequently in your program.

- Use consistent naming schemes to refer to objects in a particular role. For example if multiple classes delegate the creation of widget objects, the role of the delegatee object becomes more obvious if all of the classes that delegate that operation refer to delegatee objects through a variable called `widgetFactory`.

- You can always clarify he purpose of a delegation by writing comments.

Note that it is possible and advantageous to use all three of these strategies at the same time.

Implementation

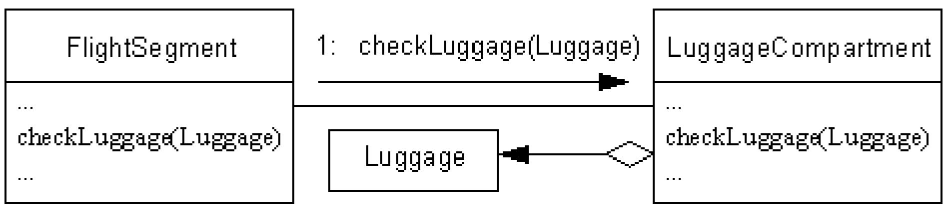
The implementation of delegation is very straightforward. It simply involves acquiring a reference to an instance of the class you want to extend.

JAVA API Usage

The Java API is full of examples of delegation. A particularly good example is the `java.awt.AWTEventMulticaster` class. Other classes that implement GUI components and are sources of `awt` events use the `AWTEventMulticaster` class. Instances of those classes delegate to instances of `AWTEventMulticaster` the responsibility of remembering what objects are listening for the events that they produce and also the responsibility of sending an event to all of those objects.

Example

For a example of delegation, we will look at another part of an airline reservation system. Suppose that the reservation system is responsible for keeping track of checked pieces of luggage. We can expect this part of the system to include classes to represent a flight segment, a luggage compartment and pieces of luggage as shown in the following collaboration diagram:



Check Luggage

In the above diagram, the `FlightSegment` class that has a method called `checkLuggage` that checks a piece of luggage onto a flight. The flight class delegates that operation to an instance of the `LuggageCompartment` class.

Another common use for delegation is to implement aggregation. A class such as `LuggageCompartment` that maintains an aggregation of other objects normally delegates that aggregation to another object, such as an instance of `java.util.Vector`. Because implementing aggregation by delegation is so common, the separate aggregation object is frequently omitted from design drawings.

Here are code fragments that implement the above design. Firstly, here is the `FlightSegment` class that delegates the `checkLuggage` operation to the `LuggageCompartment` class:

```
class FlightSegment {
...
    LuggageCompartment luggage;
...
    /**
     * Check a piece of luggage
     * @param piece The piece of luggage to be checked.
     * @exception LuggageException if piece cannot be checked.
     */
    void checkLuggage(Luggage piece) throws LuggageException {
        luggage.checkLuggage(piece);
    } // checkLuggage(Luggage)
} // class FlightSegment
```

Here is the `LuggageCompartment` class that delegates the aggregation of pieces of luggage to the Vector class

```
class LuggageCompartment {
    // Max weight for this luggage compartment in kg
    private float maxWeight;
    private float weight = 0.0F;    // Total weight of checked luggage

    // The pieces of luggage in this LuggageCompartment
    private Vector pieces; //
    /**
     * Constructor
     * @param maxWeight max weight for this luggage compartment in kg
     */
    LuggageCompartment(float maxWeight) {
        this.maxWeight = maxWeight;
        pieces = new Vector();
    } // Constructor(float)
    /**
     * Check a piece of luggage
     * @param piece The piece of luggage to be checked.
     * @exception LuggageException if piece cannot be checked.
     */
    synchronized
    void checkLuggage(Luggage piece) throws LuggageException {
        float pieceWeight = piece.getWeight();
        if (pieceWeight + weight > maxWeight)
          throw new LuggageException("Luggage compartment is full");
        weight += pieceWeight;
        pieces.addElement(piece);
    } // checkLuggage(Luggage)
} // class LuggageCompartment
```

Related Patterns

Almost every other pattern uses delegation. DelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegationDelegation

Class Decoupling

Synopsis

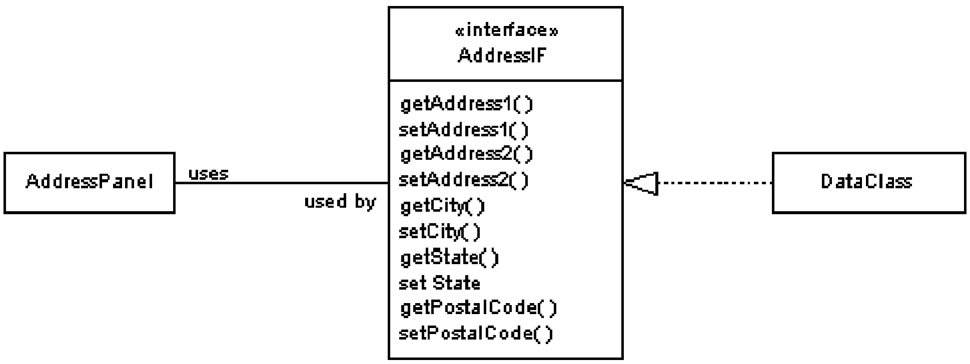
Keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.

Context

Suppose that you are writing an application to managing the purchase of goods for a business. Among the entities your program will need to be informed about will be vendors, freight companies, receiving locations and billing locations. One thing they have in common is that they all have street addresses. These street addresses will appear in different parts of the user interface. You will want to have a class for displaying and editing street addresses so that you can reuse it wherever there is an address in the user interface. Let's call that class `AddressPanel`.

You will want `AddressPanel` objects to be able to get and set address information in a separate data object. That raises the question of what can instances of the `AddressPanel` class assume about the class of the data objects that will be with them. Clearly, you will use different classes to represent vendors, freight companies and the like. If you program in a language like C++ that supports multiple inheritance, you can arrange for the data objects that instances of `AddressPanel` use to inherit from an address class in addition to the other classes they inherit from. If you program in a language like Java that uses a single inheritance object model, then you must explore other solutions.

You can solve the problem by creating an address interface. Instances of the `AddressPanel` class would then simply require data objects that implement the address interface. The would then be able to call the accessor methods of that object to get and the its address information. Using the indirection that the interface provides, instances of the `AddressPanel` are able to call the methods of the data object without having to be aware of what class it belongs to. Here is a class diagram showing these relationships:



Indirection Through Address Interface

Forces

- If the instances of a must use another object and that object is assumed to belong to a particular class the reusability of a class would be compromised.

Solution

To avoid the coupling of classes because they share a uses/used-by relationship, make the usage indirect through an interface. Here is a class diagram showing this relationship:

Class Decoupling

Here are the roles that these classes and interface play:

Client
: The `Client` class uses other classes that implement the `InterectionIF` interface.

IndirectionIF
: The `IndirectionIF` interface provides the indirection that keeps the `Client` class independent of the class that is playing the `Service` role.

Service
: Classes in this role provide a service to classes in the `Client` role.

Consequences

- Applying the Class Decoupling pattern keeps a class the needs a service from another class from being coupled to any specific class.

- Like any other indirection, the Class Decoupling pattern can make a program more difficult to understand.

Implementation

Implementation of the Class Decoupling pattern is straightforward. Define an interface to provide a service, write client classes to access the service through the interface and write service providing classes that implement the interface.

JAVA API Usage

The Java API defines the interface `java.io.FilenameFilter`. That interface declares a method that can be used to decide is a named file should be included in a collection. The Java API also defines the `java.awt.FileDialog` class that can use a `FilenameFilter` object to filter the files that it displays. You can pass the `list` method of the `java.io.File` class a `FilenameFilter` object to filter the files that it puts in the array that it returns.

Example

The example for the Class Decoupling pattern is the `AddressPanel` class and `AddressIF` interface discussed under the Context heading. Here is code for the `AddressPanel` class:

```
class AddressPanel extends Panel {
    private AddressIF data;     // Data object
    // Text fields
    TextField address1Field   = new TextField("", 35);
    TextField address2Field   = new TextField("", 35);
    TextField cityField       = new TextField("", 16);
    TextField stateField      = new TextField("", 2);
    TextField postalCodeField = new TextField("", 10);
...
/**
     * Set the data object that this panel will work with.
     * @param address The data object that this object should fetch
     *                and store data from.
     */
    public void setData(AddressIF address) {
        data = address;
        address1Field.setText(address.getAddress1());
        address2Field.setText(address.getAddress2());
        cityField.setText(address.getCity());
        stateField.setText(address.getState());
        postalCodeField.setText(address.getPostalCode());
    } // setData(AddressIF)

    /**
     * Save the contents of the TextFields into the data object.
     */
    public void save() {
        if (data != null) {
            data.setAddress1(address1Field.getText());
            data.setAddress2(address2Field.getText());
            data.setCity(cityField.getText());
```

```
            data.setState(stateField.getText());
            data.setPostalCode(postalCodeField.getText());
        } // if data
    } // save()
} // class AddressPanel
```

Notice that the Class Decoupling pattern only manifests itself in the fact that the `AddressPanel` class declares its `data` instance variable as an interface type.

The heart of the Class Decoupling pattern is the interface that provides the indirection between the client class and the service class. Here is the code for the `AddressIF` interface that provides that indirection for the `AddressPanel` class:

```
public interface AddressIF {
    /**
     * Get the first line of the street address.
     */
    public String getAddress1();

    /**
     * Set the first line of the street address.
     */
    public void setAddress1(String address1);

    /**
     * Get the second line of the street address.
     */
    public String getAddress2();

    /**
     * Set the second line of the street address.
     */
    public void setAddress2(String address2);

    /**
     * Get the city.
     */
    public String getCity();

    /**
     * Set the city.
     */
    public void setCity(String city);

    /**
     * Get the state.
     */
    public String getState();

    /**
     * Set the state.
     */
    public void setState(String state);

    /**
     * get the postal code
     */
    public String getPostalCode() ;

    /**
```

```
     * set the postal code
     */
    public void setPostalCode(String PostalCode);
} // interface AddressIF
```

The interface simply declares the methods required for the needed service.

Finally, here is code for service class. The only impact that the Class Decoupling pattern has on the class is that it implements the `AddressIF` interface.

```
class ReceivingLocation extends Facility implements AddressIF{
    private String address1;
    private String address2;
    private String city;
    private String state;
    private String postalCode;
...
    /**
     * Get the first line of the street address.
     */
    public String getAddress1() { return address1; }

    /**
     * Set the first line of the street address.
     */
    public void setAddress1(String address1) { this.address1 = address1; }

    /**
     * Get the second line of the street address.
     */
    public String getAddress2() { return address2; }

    /**
     * Set the second line of the street address.
     */
    public void setAddress2(String address2) { this.address2 = address2; }

    /**
     * Get the city.
     */
    public String getCity() { return city; }

    /**
     * Set the city.
     */
    public void setCity(String city) { this.city = city; }

    /**
     * Get the state.
     */
    public String getState() { return state; }

    /**
     * Set the state.
     */
    public void setState(String state) { this.state = state; }

    /**
     * get the postal code
     */
    public String getPostalCode() { return postalCode; }
```

```
    /**
     * set the postal code
     */
    public void setPostalCode(String postalCode) {
        this.postalCode = postalCode;
    } // setPostalCode(String)
} // class ReceivingLocation
```

Related Patterns

Delegation

The Delegation and Class Decoupling patterns are often used together.

The Class Decoupling pattern is used in many other patterns.Class DecouplingClass DecouplingClass DecouplingClass DecouplingClass Decoupling

Immutable Object

Synopsis

The Immutable Object pattern increases the robustness of objects that share references to the same object and reduces the overhead of concurrent access to an object. It accomplishes that by not allowing any of an object's state information to change after the object is constructed. The Immutable Object pattern also avoids the need to synchronize multiple threads of execution that share an object.

Context

The Immutable Object pattern is useful in a great variety of contexts. What these contexts have in common is that they use instances of a class that are shared by multiple objects and whose states are fetched more often than changed.

In situations where multiple objects share access to the same object, a problem can arise if changes to the shared object are not properly coordinated between the objects that share it. That can require careful programming that is easy to get wrong. If the changes to and fetches of the shared objects' state are done asynchronously, then in addition to the greater likelihood of bugs, correctly functioning code will have the overhead of synchronizing the accesses to the shared objects' state.

The Immutable Object pattern avoids these problems. It organizes a class so that the state information of its instances never changes after they are constructed.

Forces

- Your program uses instances of a class that is passive in nature. The instances do not ever need to change their own state. The instances of that class are used by multiple other objects.

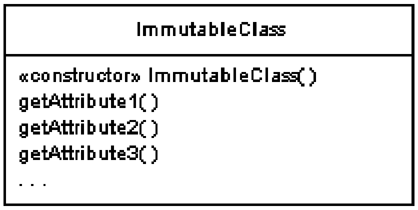
- Correctly coordinating changes to the state information of an object that is used by multiple other objects is difficult and bug prone.

- If access to a shared object's state information involves multiple threads and modification of its state information, then the threads that access the state information must be synchronized in order to ensure consistency.

- The overhead of synchronizing the threads may add an unacceptable overhead to accessing the shared object's state information.

Solution

To avoid having to manage the propagation and synchronization of changes to the state information of objects used by multiple other objects, make the shared objects immutable, disallowing any changes to their state after they are constructed. You can accomplish that by not including any methods, other than constructors, in their class that modify state information. Such a class can be organized like this:



Immutable

Notice that the class has accessor methods to get state information but not set it.

Consequences

Since the state of immutable objects never changes, there is no need to write code to manage such changes. Also, there is no need to synchronize threads that access immutable objects.

Operations that would otherwise have changed the state of an object must create a new object. This is an overhead that mutable object do not incur.

Implementation

There are two concerns you should have when implementing the Immutable Object pattern.

- No method, other than a constructor, should modify the values of a class' instance variables.

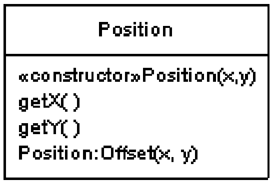
- Any method that computes new state information must store that information in a new instance of the same class, rather than modifying the existing object's state.

JAVA API Usage

Instances of the `String` class are immutable. The sequence of characters that a `String` object represents is determined when it is constructed. The `String` class does not provide any methods to change the sequence of characters represented by a `String` object. Methods of the `String` class, such as `toLowerCase` and `substring`, that compute a new sequence of characters return the new sequence of characters in a new `String` object.

Example

Suppose that you are writing a game program that involves the placement and occasional movement of objects on a playing field. In the course of designing the classes for that program, you decide that you want to use immutable objects to represent the position of objects on the playing field. The organization of a class for modeling position that way might look like this:



Immutable Position

You have a class called `Position` that has an x and y value associated with its instances. The class has a constructor that specifies the x an y value. It also has methods to fetch the x and y value associated its instances. Lastly, it has a method that creates a new `Position` object that is a given x and y offset from an existing position.

Here is what the declaration for such a position class might look like:

```
class Position {
    private int x;
    private int y;

    /**
     * Constructor
     * @param x The x position associationed with this object.
     * @param y The y position associationed with this object.
     */
    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    } // Position(int, int)

    /**
     * Return the x value assolciated with this object.
     */
    public int getX() { return x; }

    /**
     * Return the y value assolciated with this object.
     */
    public int getY() { return y; }

    /**
     * Return a Position object that has x and y values that are
     * offset from the x and y values of this object by the given
     * amount.
     * @param xOffset The x offset.
     * @param yOffset The y offset.
     */
    public Position offset(int xOffset, int yOffset) {
        return new Position(x+xOffset, y+yOffset);
    } // offset(int, int)
} // classs Position
```

Related Patterns

Single Threaded Execution

The Single Threaded Execution pattern is the pattern most frequently used to synchronize the access by multiple thread of a shared object. Immutable ObjectImmutable ObjectImmutable ObjectImmutable ObjectImmutable ObjectImmutable Object

Semantic Interface

The Semantic Interface pattern occurs rarely outside of utility classes. However, it is included in this chapter because it takes advantage of the fundamental nature of class declarations.

Synopsis

The Semantic Interface pattern uses interfaces that declare no methods or variables to indicate semantic attributes of a class. It works particularly well with utility classes that must determine something about objects without assuming they are an instance of any particular class.

Context

Java's `Object` class defines a method called `equals` that takes an argument that can be a reference to any object. Since Java's `Object` class us the ultimate superclass of all other classes in Java, all other classes inherit the `equals` method from the `Object` class. The implementation of `equals` provided by the `Object` class returns true if the object passed to it is the same object as the object it is associated with. Classes that want their instances to be considered equal if they contain the same values override the `equals` method appropriately.

Container objects, such as `java.util.Vector`, call an object's `equals` method when performing a search of their contents to find an object that is equal to a given object. Such searches might call an object's `equals` method for each object in the container objects. That is wasteful in those cases where the object being searched for belongs to a class that does not override the `equals` method. It is faster to use the == operator to determine if two objects are the same object than it is to call the `Object` class' implementation of the `equals` method. If the container class were able to determine that the object being searched for belongs to a class that does not override the `equals` method, then it could use the == operator instead of calling `equals`. The problem with that is that there is no way to determine if an arbitrary object's class overrides the `equals` method.

It is possible to provide a hint to container classes to let them know that it is correct to use the == operator for an equality test on instances of a class. You can define an interface called `EqualByIdentity` that declares no methods or variables. You can then write container classes to assume that if a class implements `EqualByIdentity` then it the equality comparison can be done using the == operator.

An interface that does not declare methods or variables and is used to indicate attributes of classes that implement them is said to be a semantic interface.

Forces

- Utility classes may need to know something about the intended use of an object's class without relying on object's being an instance of a particular class.

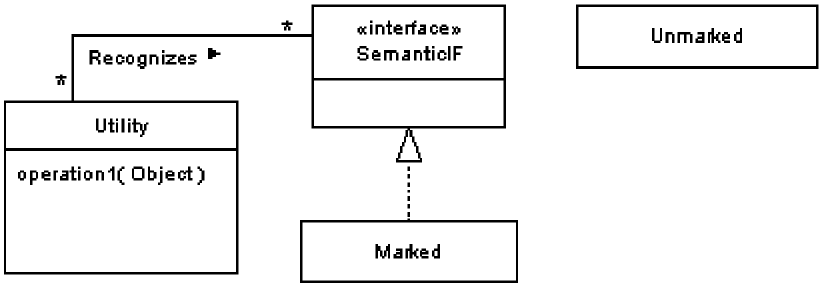
- Classes can implement any number of interfaces.

- It is possible to determine if an object's class implements a known interface without relying on the object being an instance of any particular class.

Solution

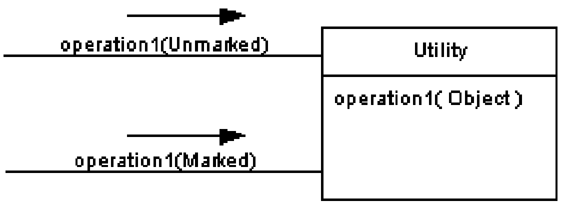
For instances of a utility class to determine if another class' instances are included in a classification without the utility class having knowledge other classes, it can determine if other classes implement a semantic interface. A semantic interface is an interface that does not declare any methods or variables. You declare a class to implement a semantic interface to indicate that it belongs to the classification associated with the semantic interface.

Here is a diagram that shows these relationships:



Semantic Interface Class Diagram

The above diagram shows a semantic interface called `SemanticIF`. There is a class called `Marked` that implements `SemanticIF` and a class called `Unmarked` that doesn't. There is also a utility class called `Utility` that is aware of the `SemanticIF` interface.



Semantic Interface Collaboration

Instances of `UtilityClass` receive calls to their `operation1` method. The parameter passed to that method can be an object that implements or does not implement `SemanticIF`.

Consequences

- Instances of utility classes are able to make inferences about objects passes to their methods without depending on the objects to be instances of any particular class.

- The relationship between the utility class and the semantic interface is transparent to all other classes except for those classes that implement the interface.

Implementation

The essence of the Semantic Interface pattern is that an object that either does or does not implement a semantic interface is passed to a method of a utility class. The formal parameter that corresponds to that object is typically declared as `Object`. If appropriate, it is reasonable to declare that formal parameter to be a more specialized. class.

It is also possible to use an interface that declares methods in the Semantic Interface method. In such cases, the interface used as a semantic interface usually extends a purely semantic interface.

Declaring that a class implements a semantic interface implies that the class is included in that classification implied by the interface. It also implies that all subclasses of that class are included in the classification. If there is any possibility that someone will declare a subclass that does not fit the classification, then you should take measures to prevent that from happening. Such measures might include declaring the class final to prevent it from being subclassed or its equals method to be final to prevent it from being overridden.

JAVA API Usage

The `ObjectOutputstream` class writes objects as a stream of bytes that can be read and turned back into an object by the `ObjectInputStream` class. The conversion of an object to a stream of bytes is called serialization. There are a number of reasons why instances of some classes should not be serialized. Because of that, the `ObjectOutputstream` class refuses to serialize objects unless their class indicates that the serialization should be allowed. The way that classes indicate that their instances are allowed to be serialized is by implementing the `Serializable` interface.

Example

For an example of an application of the Semantic Interface pattern, see the following class that implements a linked list data structure. At the bottom of the listing, you will see methods called `find`, `findEq` and `findEquals`. The purpose of all three methods is to find a `LinkedList` node that refers to a specified object. The `find` method is the only one of the three that is public. The `findEq` method performs the necessary equality tests using the == operator. The `findEquals` method performs the necessary equality tests using the equals method of the object being searched for. The `find` method decides which of the other two methods to call by determining if the object to search for implements the semantic interface `EqualByIdentity`.

```
/**
 * Instances of this class are nodes of a linked list.
 * A linked list is a chain of objects that have two object references
 * associated with them.  One is the head of the list, which is another
 * data object.  The other is the tail of the list, which is either null
 * or another linked list.
 */
public class LinkedList implements Cloneable, java.io.Serializable {
    private Object head;
    private LinkedList tail;
    private boolean traversed = false; // true when this noed is being traversed

    /**
     * This constructor creates a LinkedList with the given head and a null tail.
     * @param head The object that will be the head of this list
     */
    public LinkedList(Object head) {
        this(head, null);
    } // constructor(LinkedList)

    /**
     * This constructor creates a LinkedList with the given head and tail.
     * @param head The object that will be the head of this list
     * @param tail null or the rest of this linked list.
     */
    public LinkedList(Object head, LinkedList tail) {
        this.head = head;
        this.tail = tail;
    } // constructor(LinkedList)

    /**
     * Return the head of this linked list.
     */
    public Object getHead() {
        return head;
```

```
    } // getHead()

    /**
     * Return the tail of this linked list.
     */
    public LinkedList getTail() {
        return tail;
    } // getTail()

    /**
     * Return the number of nodes in this linked list
     */
    synchronized public int size() {
        if (tail == null)
          return 1;
        try {
            traversed = true;
            if (tail.traversed)
              return 1;
            return 1 + tail.size();
        } finally {
            traversed = false;
        } // try
    } // size()

    /**
     * Return an Enumeration of the data in this linked list (the
     * heads).
     */
    public Enumeration elements() {
        return new ListEnumeration();
    } // elements()

    /**
     * private class to enumerate data of a linked list.
     */
    private class ListEnumeration implements Enumeration {
        private LinkedList thisNode = LinkedList.this;

        /**
         * Tests if this enumeration contains more elements.
         * @return  <code>true</code> if this enumeration contains more
         *          elements;
         */
        public boolean hasMoreElements() {
            return thisNode != null;
        } // hasMoreElements()

        /**
         * Returns the next element of this enumeration.
         * @return     the next element of this enumeration.
         * @exception  NoSuchElementException  if no more elements exist.
         */
        public Object nextElement() {
            if (thisNode == null)
              throw new NoSuchElementException();
            Object next = thisNode.head;
            thisNode = thisNode.tail;
            return next;
        } // nextElement()
    } // class ListEnumeration
```

```
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is normally determined by calling the given
     * object’s equals method.  However, if the given object implements
     * the EqualByIdentity interface, then equality will be determined
     * by the ==  operator.
     * @params target The object to search for.
     * @return a LinkedList whose head is equal to the given object or
     *         null if the target is not found.
     */
    public LinkedList find(Object target) {
        if (target == null || target instanceof EqualByIdentity)
          return findEq(target);
        else
          return findEquals(target);
    } // find(Object)

    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by the == operator.
     * @params target The object to search for.
     * @return a LinkedList whose head is equal to the given object.
     */
    private synchronized LinkedList findEq(Object target) {
        if (head == target)
          return this;
        if (tail == null)
          return null;
        try {
            traversed = true;
            if (tail.traversed)
              return null;
            return tail.findEq(target);
        } finally {
            traversed = false;
        } // try
    } // find(Object)

    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by calling the given
     * object’s equals method.
     * @params target The object to search for.
     * @return a LinkedList whose head is equal to the given object.
     */
    private synchronized LinkedList findEquals(Object target) {
        if (head.equals(target))
          return this;
        if (tail == null)
          return null;
        try {
            traversed = true;
            if (tail.traversed)
              return null;
            return tail.findEquals(target);
        } finally {
            traversed = false;
        } // try
    } // find(Object)
} // class LinkedList
```

Proxy

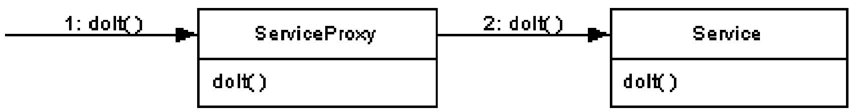
Proxy is a very general pattern that occurs in many other patterns. Also, there are patterns that consist entirely of a specialized application of proxy.

Synopsis

The Proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a surrogate for the other object, delegating method calls to that object. Classes for proxy objects are declared in a way that minimizes client object's awareness that they are dealing with a proxy.

Context

A proxy object is an object that receives method calls on behalf of another object. Client objects call the proxy object's method. The proxy object's methods do not directly provide the service that its clients expect. Instead, the proxy object's methods call the methods of the object that provides the actual service. Here is a diagram showing that structure:



Method Calls Through a Proxy

Though a proxy object's methods do not directly provide the service that its clients expect, the proxy object may provide some management of those services. Proxy objects generally share a common interface or superclass with the service providing object. That makes it possible for client objects to be unaware that the are calling the methods of a proxy object rather than the methods of the actual service providing object. Transparent management of another object's services is the basic reason for using a proxy.

There many different types of service management that a proxy can be used to provide. Some of the more important ones are documented elsewhere in this book as patterns in their own right. Here are some of the more common uses for proxies:

- Make a method that can take a long time to complete appear to return immediately. This use of proxies is documented as the Asynchronous Invocation pattern.

- Create the illusion that an object that exists on a different machine is an ordinary local object. This use of proxies is documented as the Remote Proxy pattern.

- Control access to a service providing object. This use of proxies is called the Access Proxy pattern.

- Create the illusion that a service object exists before it actually does. That can be useful if a service object is expensive to create and its services may not be needed. This use proxies is documented as the Virtual Proxy pattern.

Forces

- It is not possible for a service providing object to provide a service at a time or place that is convenient.

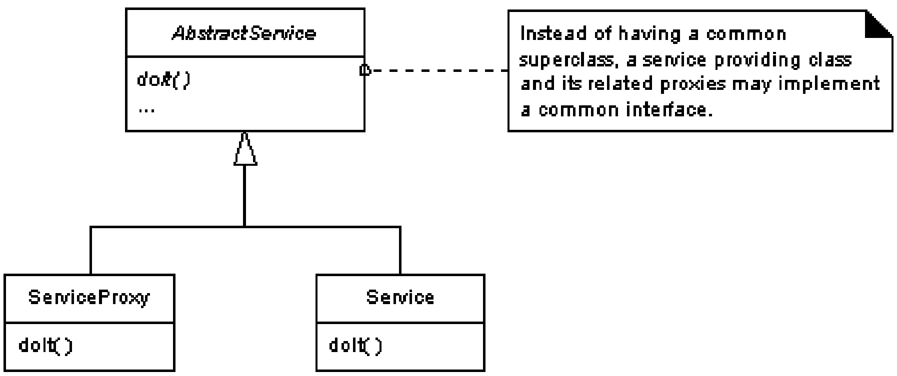
- Gaining visibility to an object is non-trivial and you want to hide that complexity.

- Access to a service providing object must be controlled without adding complexity to the service providing object or coupling the service to the access control policy.

- The management of a service should be provided in a way that is transparent as possible to the clients of that service.

Solution

Transparent management of a service providing object can be accomplished by forcing all access to the service providing object to be accomplished through a proxy object. In order for the management to be transparent, the proxy object and the service providing object either must both be instances of a common superclass or implement a common interface:



Proxy Class Diagram

The above diagram does not show any details for implementing any particular management policy.

Consequences

The service provided by a service providing object is managed in a manner transparent to that object and its clients.

Unless the use of proxies introduces new failure modes, there is normally no need for the code of client classes to reflect the use of proxies.

Implementation

Without any specific management policy, the implementation of the Proxy pattern simply involves creating a class that shares a common superclass or interface with a service providing class and delegates operations to instances of the service providing class.

JAVA API Usage

The Java API does not use the vanilla Proxy pattern. Where proxies are used in the Java API, they are used to provide a specific form of service management.

Example

The application of the Proxy pattern is not useful unless it includes some service management behavior. The example for the Proxy pattern uses proxies to defer an expensive operation until it is actually needed. In the cases that the operation turns out not to be needed, the operation is never performed.

The example is a subclass of `java.util.Hashtable` that is functionally equivalent to `Hashtable`. The different is the way that it handles the clone operation. Cloning a `Hashtable` is an expensive operation.

One of the more common reasons for cloning an object like a `Hashtable` is to avoid holding a lock on the object for a long time when all that is desired is to fetch multiple key-value pairs. In a multi-threaded program, to ensure that a `Hashtable` in a consistent state when you are fetching key-value pairs from it, you can use a synchronized method to obtain exclusive access to the `Hashtable`. While that is going on, other threads will wait to gain access to the same `Hashtable`, which may be unacceptable. In some other cases it may not be possible to retain exclusive access. An example of that is the `Enumeration` object returned by the `Hashtable` class' elements object.

Cloning a `Hashtable` prior to fetching values out of it is a defensive measure. Cloning the `Hashtable` avoids the need to obtain a synchronization lock on a `Hashtable` beyond that time that it takes for the clone operation to complete. When you have a freshly cloned copy of a `Hashtable`, you can be sure that no other thread has access to the copy. Since no other thread as access to the copy, you will be able to fetch key-value pairs from the copy without any interference with other threads.

If, after you clone a `Hashtable`, there is no subsequent modification to the original `Hashtable`, then the time and memory spent in creating the clone was wasted. The point of this example is to avoid that waste. It does that by delaying the cloning of a `Hashtable` until a modification to it actually occurs.

The main class in the example is called `LargeHashtable`. Instances of `LargeHashtable` are a copy-on-write proxy for a `Hashtable` object. When a proxy's clone method is called, it returns a copy of the proxy but does not copy the `Hashtable` object. At that point both the original and copy of the proxy refer to the same `Hashtablec` object. When either of the proxies is asked to modify the Hashtable, they recognize that they are using a shared `Hashtable` and clone the `Hashtable` before they make the modification.

The way that the proxies know that they are working with a shared `Hashtable` object is that the `Hashtable` object that the proxies work with is an instance of a private subclass of `Hashtable` called `ReferenceCountedHashTable`. A `ReferenceCountedHashTable` object keeps a count of how many proxies refer to it.

```
public class LargeHashtable extends Hashtable {
    // The ReferenceCountedHashTable that this is a proxy for.
    private ReferenceCountedHashTable theHashTable;
...
public LargeHashtable() {
        theHashTable = new ReferenceCountedHashTable();
    } // constructor()

    /**
     * Return the number of key-value pairs in this hashtable.
     */
    public int size() {
        return theHashTable.size();
    } // size()
...
    /**
     * Return the value associated with the specified key in this Hashtable.
     * @param   key   a key in the hashtable.
     */
    public synchronized Object get(Object key) {
        return theHashTable.get(key);
    } // get(key)

    /**
```

```
         * Add the given key-value pair to this Hashtable.
         * @param      key     the key.
         * @param      value   the value.
         * @return     the previous value of the given key in this hashtable,
         *             or <code>null</code> if it did not have one.
         * @exception  NullPointerException  if the key or value is null.
         */
        public synchronized Object put(Object key, Object value) {
            copyOnWrite();
            return theHashTable.put(key, value);
        } // put(key, value)
    ...
        /**
         * Return a copy of this proxy that accesses the same Hashtable as this
         * proxy. The first attempt for either to modify the contents of the
         * Hashtable results in that proxy accessing a modified clone of the
         * original Hashtable.
         */
        public synchronized Object clone() {
            Object copy = super.clone();
            theHashTable.addProxy();
            return copy;
        } // clone()

        /**
         * 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) {
                // Synchronize on the original Hashtable to allow consistent
                // recovery on error.
                synchronized (theHashTable) {
                    theHashTable.removeProxy();
                    try {
                        theHashTable
                          = (ReferenceCountedHashTable)theHashTable.clone();
                    } catch (Throwable e) {
                        theHashTable.addProxy();
                    } // try
                } // synchronized
            } // if proxyCount
        } // copyOnWrite()
    ...
        private class ReferenceCountedHashTable extends Hashtable {
            private int proxyCount = 1;
    ...
            public ReferenceCountedHashTable() {
                super();
            } // constructor()

            /**
             * 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;
            } // clone()
```

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

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

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

Related Patterns

Access Proxy
: The Access Proxy pattern uses a proxy to enforce a security policy on access to a service providing object.

Asynchronous Invocation
: The Asynchronous Invocation pattern uses a proxy to create the illusion that a method that takes a long time to complete returns immediately.

Broker
: The Proxy pattern is sometimes used with the Broker pattern to provide a transparent way of forwarding service requests to a service object selected by the Broker/Proxy object.

Facade
: The facade pattern uses a single object as a frond end to a set of interrelated objects.

Remote Proxy
: The Remote Proxy pattern uses a proxy to hide the fact that a service object is located on a different machine than the client objects that want to use it.

Virtual Proxy
: This pattern uses a proxy to create the illusion that a service providing object exists before it has actually been created. Ii is useful if the object is expensive to create and its services may not be needed.

Wrapper
: The Wrapper pattern is structurally similar to the Proxy pattern in that it forces access to a service providing object to be done indirectly through another object. The difference is a matter of intent. Instead of trying to manage the service the indirection object in some way enhances the service.