Structural Patterns

The patterns in this chapter describe common ways that different types of objects can be organized to work with each other.

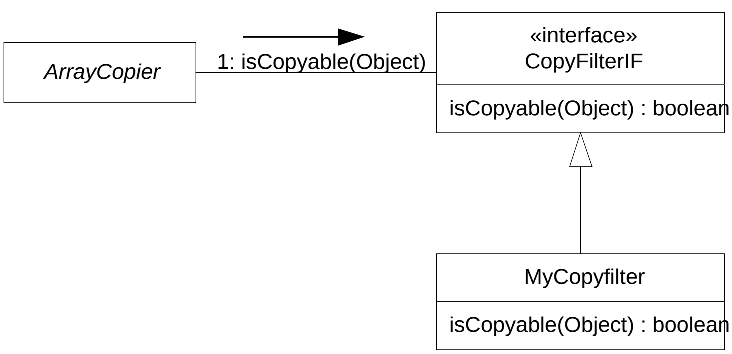
Adapter

Synopsis

An Adapter class implements an interface known to its clients and provides access to an instance of a class not know to its clients. An adapter object provides the functionality promised by an interface without having to assume what class is being used to implement that interface.

Context

Suppose that you are writing a method that copies an array of objects, filtering out objects that do not meet certain criteria. To promote reuse, you would like to make the method independent of the actual filtering criteria being used. You could achieve that by defining an interface that declares a method that the array copier can call to find out if it should include a particular object in the new array:

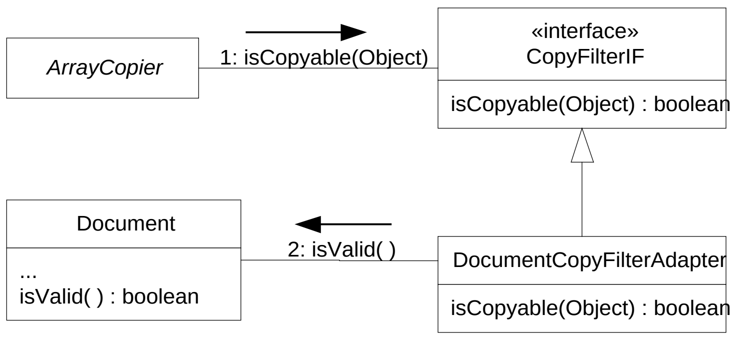


Simple Copy Filter

In the above design, an `ArrayCopier` class uses instances of classes that implement the `CopyFilterIF` interface to decide if it should copy an element of the old array to the new array. If their `isCopyable` method returns true for an object then that object is copied to the new array.

That solution solves the immediate problem of allowing the copy criteria used by the array copier to be encapsulated in a separate object without having to be concerned about what the object's class is. That solution also presents a different problem. The problem is that the filtering logic is in a different object than the objects that are being filtered. Sometimes the logic needed for the filtering is in a method of the objects to be filtered. If those objects don't implement the `CopyFilterIF` interface then there is no way for the array copier to directly ask those objects if they should be copied. However, it is possible for the array copier to indirectly ask the filtered objects if they should be copied, even if they don't implement the `CopyFilterIF` interface.

Suppose that there is a class called Document that has a method called `isValid` that returns a boolean result. Suppose that you need to use the result of the `isValid` method to do the filtering for a copy operation. Because Document does not implement the `CopyFilterIF` interface, an `ArrayCopier` object cannot directly use a document object for filtering. A class that implements the `CopyFilterIF` interface but tries to independently determine if a Document object should be copied into a new array does not work. It does not work because it has no way to get the necessary information without calling the Document object's `isValid` method. The answer is for that object to call the Document object's `isValid` method, resulting in this solution:



Copy Filter Adapter

In this solution, the `ArrayCopier` object calls the `isCopyable` method of an object that implements the `CopyFilterIF` interface, as it always does. In this case, that object is an instance of a class called `DocumentCopyFilterAdapter`. The `DocumentCopyFilterAdapter` class implements the `isCopyable` method by calling the Document object's `isValid` method.

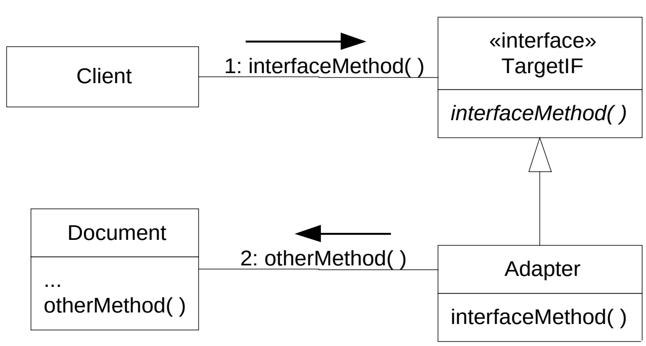
Forces

You want to use a class that calls a method through an interface, but you want to use it with a class that does not implement that interface. Modifying that class to implement the interface is not an option either because

- You do not have the source code for the class.
- The class is a general-purpose class and it would be inappropriate for it to implement an interface for a specialized purpose.

Solution

Suppose that you have a class that calls a method through an interface. You want an instance of that class to call a method of an object that does not implement the interface. You can arrange for the instance to make the call through an adapter object that implements the interface with a method that calls a method of the object that doesn't implement the interface. Here is a collaboration diagram showing how this works:



Adapter

Here are the roles the that the classes and interface play:

Client

This is a class that calls a method of another class through an interface in order not to assume that the object its calls the method through belongs to a specific class.

TargetIF

This interface declares the method that the client class calls.

Adapter

This class implements the `TargetIF` interface. It implements the method that the client calls by having it call a method of the `Adaptee` class, which does not implement the `TargetIF` interface.

Adaptee

This class does not implement the `TargetIF` method but has a method that we want the Client class to call.

It is possible for an adapter class to do more than simply delegate the method call. It may perform some transformation on the arguments. It may provide additional logic to hide differences between the intended semantics of the interface's method and the actual semantics of the adaptee class' method. There is no limit to how complex an adapter class can be. So long as the essential purpose of the class is as an intermediary for method calls, you can considered it to be an adapter class.

Consequences

- The client and adaptee classes remain independent of each other.

- The Adapter pattern introduces an additional indirection into a program. Like any other indirection, it contributes to the difficulty involved in understanding the program.

Implementation

Implementation of the adapter class is rather straightforward. However, there is an issue that you should consider. That issue is how the adapter objects will know what instance of the adaptee class to call. There are two approaches:

- Passing a reference to the client object as a parameter to the adapter object's constructor or one of its methods allows the adapter object to be used with any instance or possibly multiple instances of the adaptee class.

- Make the adapter class an inner class of the adaptee class. That simplifies the association between the adapter object and the adaptee object by making it automatic. It also makes the association inflexible.

JAVA API Usage & Example

A very common way to use adapter classes with the Java API is for event handling, like this:

```
Button ok = new Button("OK");
ok.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        doIt();
    } // actionPerformed(ActionEvent)
  } );
add(ok);
```

The above example creates an instance of an anonymous class that implements the `ActionListener` interface. That class' `actionPerformed` method is called when the Button object is pressed. This coding pattern is very common for code that handles events.

The Java API does not include any public adapter objects that are ready to use. It does include classes such as `java.awt.event.WindowAdapter` that are intended to be subclassed rather than used directly. The idea is that there are some event listener interfaces, such as `WindowListener` that declare multiple methods that may not need to all be implemented in many cases. The `WindowListener` interface declares eight methods that are called to provide notification about eight different kinds of window events. Often only one or two of those event types are of interest. The methods that correspond to the events that are not of interest will typically be given do-nothing implementations. The `WindowAdapter` class implements the `WindowListener` interface and implements all eight of its methods with do-nothing implementations. An adapter class that subclasses the `WindowAdapter` class only needs to implement the methods corresponding to events that are of interest. It inherits do-nothing implementations for the rest. For example:

```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        exit();
    } // windowClosing(WindowEvent)
  } );
```

In the above example, the anonymous adapter class is a subclass of the `WindowAdapter` class. It only implements the `windowClosing` method. It inherits do-nothing implementations for the other seven methods from the `WindowAdapter` class.

Related Patterns

Iterator

The Iterator pattern is a specialized version of the Adapter pattern for sequentially accessing the contents of collection objects.

Proxy

The Proxy pattern, like the Adapter pattern, uses an object that is a surrogate for another object. However, a Proxy object has the same interface as the object for which it is a surrogate.

Bridge

Synopsis

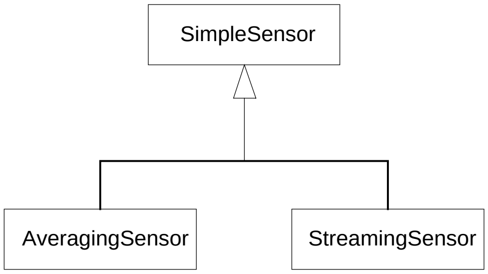
The Bridge pattern is useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations. Rather than combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically.

Context

Suppose that you need to provide Java classes that provide access to sensors for control applications. These are devices such as scales, speed measuring devices and location sensing devices. What these devices have in common is that they perform a physical measurement and produce a number on the request of a computer. One way that these devices differ is in the type of measurement that they produce.

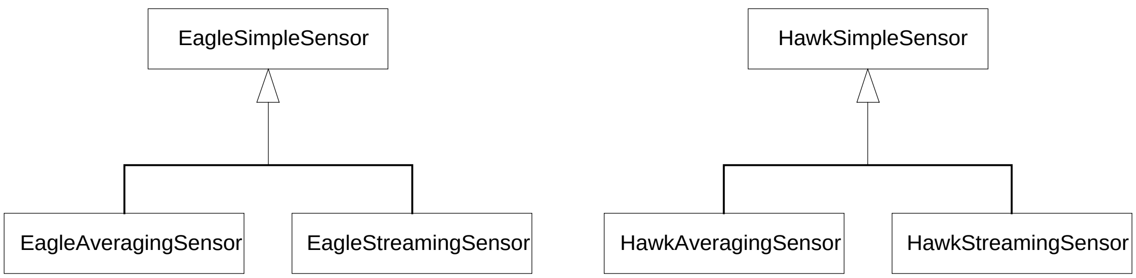
- The scale produces a single number based on a measurement at a single point in time.
- The speed measuring device produces a single measurement that is an average over a period of time.
- The location sensing device produces a stream of measurements.

This suggests that these devices can be supported by three classes that support these different measurement techniques:



Sensor Classes

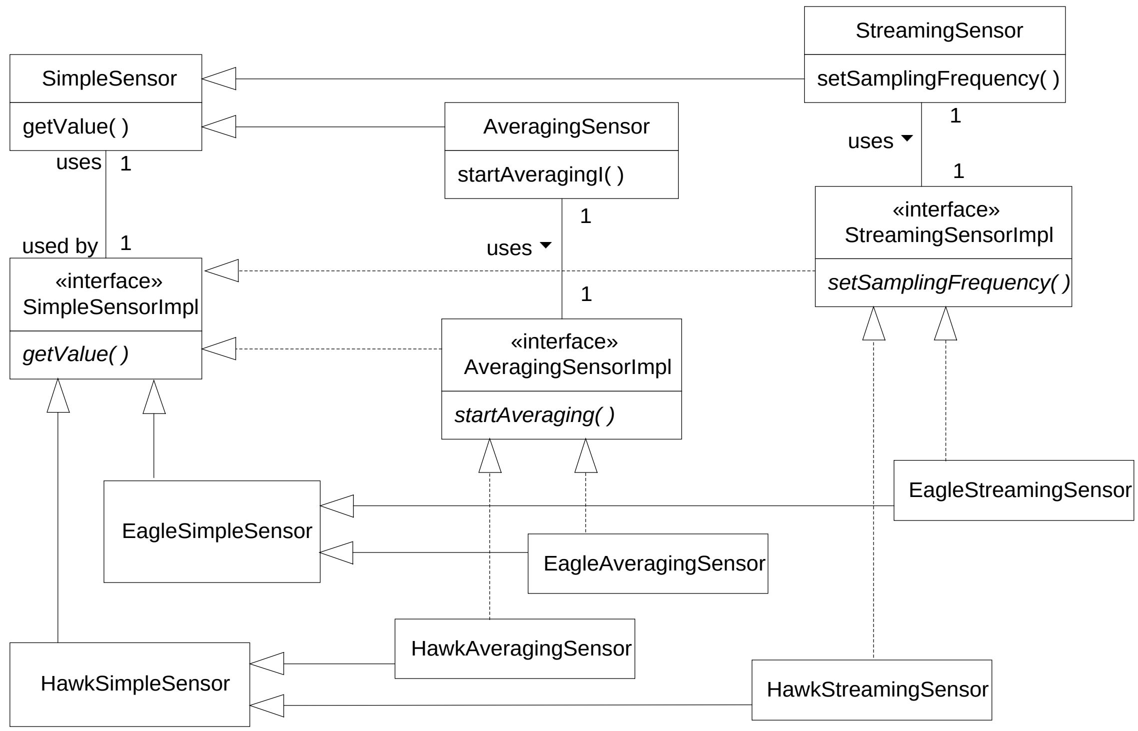
These three classes provide clean abstractions that apply to many other types of sensors that the three that inspired them. Since there are other kinds of sensors that produce simple measurements, time averaged measurements and streams of measurements, you would like to be able to reuse these classes for kinds of sensors. A difficulty in achieving such reuse is that the details of communicating with sensors from different manufacturers vary. Suppose the software that you are writing will need to work with sensors from multiple manufacturers called Eagle and Hawk. You could handle that problem by having manufacturer specific classes like this:



Manufacturer Specific Sensor Classes

The problem with this solution is not just that it does not reuse classes for simple, averaging and streaming sensors. Because it exposes differences between manufacturers to other classes, it forces other classes to recognize differences between manufacturers and therefore be less reusable. The challenge here is to represent a hierarchy of abstractions in a way that keeps the abstractions independent of their implementations.

A way to accomplish that is to shield a hierarchy of classes that support abstractions from classes that implement those abstractions by having the abstraction classes access implementation classes through a hierarchy of implementation interfaces that parallels the abstraction hierarchy.



Independent Sensor and Sensor Manufacturer Classes

Forces

- When you combine hierarchies of abstractions and hierarchies of their implementations into a single class hierarchy, classes that use those classes become tied to a specific implementation of the abstraction. Changing the implementation used for an abstraction should not require changes to the classes that use the abstraction.

- You would like to reuse logic common to different implementations of an abstraction. The usual way to make logic reusable is to encapsulate it in a separate class.

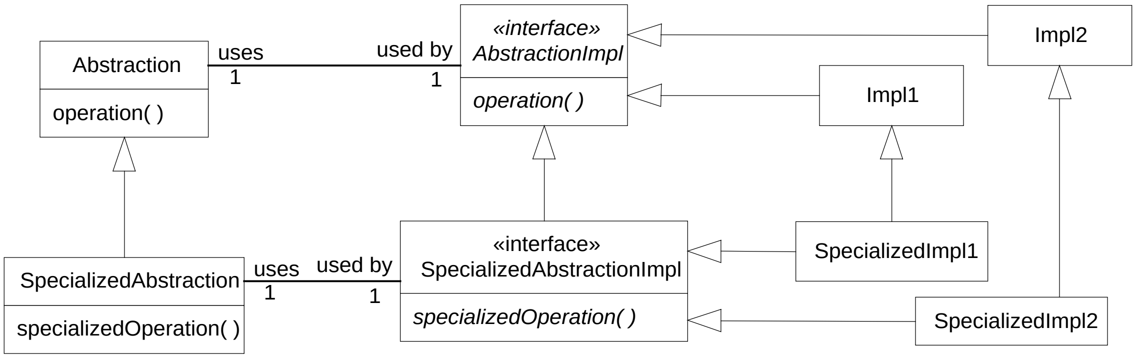
- You would like to be able to create a new implementation of an abstraction without having to re-implement the common logic of the abstraction.

- You would like to be able to extend the common logic of an abstraction by writing one new class rather that writing a new class for each combination of the base abstraction and its implementation.

- When appropriate, multiple abstractions should be able to share the same implementation.

Solution

The Bridge pattern allows classes corresponding to abstractions to be separate from classes that implement those abstractions. You can maintain a clean separation by having the abstraction classes access the implementation classes through interfaces that are in a hierarchy that parallels the inheritance hierarchy of the abstraction classes.



Bridge Pattern

Here are descriptions of the roles these classes and interfaces play in the Bridge pattern:

Abstraction
: This class represents the top-level abstraction. It is responsible for maintaining a reference to an object that implements the `AbstractionImpl` interface, so that it can delegate operations to its implementation. If an instance of the `Abstraction` class is also an instance of a subclass of the `Abstraction` class then the instance will refer to an object that implements the corresponding sub-interface of the `AbstractionImpl` interface.

SpecializedAbstraction
: This role corresponds to any subclass of the `Abstraction` class. For each such subclass of the `Abstraction` class there is a corresponding sub-interface of the `AbstractionImpl` interface. Each `SpecializedAbstraction` class delegates it operations to an implementation object that implements the interface that corresponds to the `SpecializedAbstraction` class.

AbstractionImpl
: This interface declares methods for all of the low-level operations that an implementation for the `Abstraction` class must provide.

SpecializedAbstractionImpl
: This corresponds to a sub-interface of `AbstractionImpl`. Each `SpecializedAbstractionImpl` interface corresponds to a `SpecializedAbstraction` class and declares methods for the low-level operations needed for an implementation of that class.

Impl1, Impl2
: These classes implement the `AbstractionImpl` interface and provide different implementations for the `Abstraction` class.

SpecializedImpl1, SpecializedImpl2
: These classes implement the one of the `SpecializedAbstractionImpl` interfaces and provide different implementations for a `SpecializedAbstraction` class.

Consequences

The Bridge pattern keeps the classes that represent an abstraction independent of the classes that supply an implementation for the abstraction. The abstraction and its implementations are organized into separate class hierarchies. You can extend each class hierarchy without directly impacting another class hierarchy. It is also possible to have multiple implementation classes for an abstraction class or multiple abstraction classes using the same implementation class.

Classes that are clients of the abstraction classes do not have any knowledge of the implementation classes, so an abstraction object can change its implementation without any impact on its clients.

Implementation

One issue that always must be decided when implementing the Bridge pattern is how to create implementation objects for each abstraction object. The most basic decision to make whether abstraction objects will create their own implementation objects or delegate the creation of their implementation objects to another object.

Having the abstraction objects delegate the creation of implementation objects is usually the best choice. It preserves the independence of the abstraction and implementation classes. If an abstraction class is designed to delegate the creation of implementation objects, it the design usually uses that Abstract Factory pattern to create the implementation objects.

However, if there are only a small number of implementation classes for an abstract class and the set of implementation classes is not expected to change then having the abstraction classes create their own implementation objects is a reasonable optimization.

A related decision is whether an abstraction object will use the same implementation object during its lifetime. As usage patterns or other conditions change, it may be appropriate to change the implementation object that an abstraction object is using. If an abstraction class directly creates its own implementation objects then it is reasonable to directly embed the logic for changing the implementation object in the abstraction class. Otherwise, you can use the Wrapper pattern to encapsulate the logic for switching implementation objects in a wrapper class.

JAVA API Usage

The Java API includes the package `java.awt`. That package contains the `Component` class. The `Component` class is an abstract class that encapsulates logic common to all GUI components. The `Component` class has subclasses such as `Button`, `List` and `TextField` that encapsulate the logic for those GUI components that is platform independent. The package `java.awt.peer` contains interfaces such as `ComponentPeer`, `ButtonPeer`, `ListPeer` and `TextFieldPeer` that declare methods required for implementation classes that provide platform specific support for the subclasses of the `Component` class.

The subclasses of the `Component` class use the Abstract Factory pattern to create their implementation objects. The `java.awt.Toolkit` class is an abstract class that plays the role of abstract factory. The platform supplies the concrete factory class used to instantiate the implementation classes and the implementation classes.

Example

For an example of the Bridge pattern, we will look at some code to implement the sensor related classes that were discussed under the context heading. We will assume that the objects that represent sensors and their implementation are created by a Factory Method. The Factory Method object will know what sensors are available, what objects to create to provide access to a sensor and will create those objects when access to a sensor is first requested.

Here is the code for the `SimpleSensor` class that plays the role of abstraction class:

```
public class SimpleSensor {
    // A reference to the object that implements operations specific to
    // the actual sensor device that this object represents.
    private SimpleSensorImpl impl;
    /**
     * Constructor
     *<p>
     * This constructor is intended to be called by a factory method
     * object that is in the same package as this class and the the
     * classes that implement its operations.
     * @param impl The object that implements the sensor type-specific
```

```
     *             operations this object will provide.
     */
    SimpleSensor(SimpleSensorImpl impl) {
        this.impl = impl;
    } // constructor(SimpleSensorImpl)
    /**
     * This method allows subclasses of this class to get the reference
     * to the implementation object.
     */
    protected SimpleSensorImpl getImpl() {
        return impl;
    } // getImpl()
...
    /**
     * Return the value of the sensor’s current measurement.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public int getValue() throws SensorException {
        return impl.getValue();
    } // getValue()
} // class SimpleSensor
```

As you can see, the `SimpleSensor` class is simple in that it does little more than delegate its operations to an object that implements the `SimpleSensorImpl` interface. Here is the code for the `SimpleSensorImpl` interface:

```
interface SimpleSensorImpl {
    /**
     * Return the value of the sensor’s current measurement.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public int getValue() throws SensorException;
} // interface SimpleSensorImpl
```

Some subclasses of the `SimpleSensor` class maintain the same simple structure. Here is code for the `AveragingSensor` class:

```
/**
 * Instances of this class are used to represent sensors that produce
 * values that are the average of measurements made over a period of
 * time.
 */
public class AveragingSensor extends SimpleSensor {
    /**
     * Constructor
     *<p>
     * This constructor is intended to be called by a factory method
     * object that is in the same package as this class and the the
     * classes that implement its operations.
     * @param impl The object that implements the sensor type-specific
     *             operations this object will provide.
     */
    AveragingSensor(AveragingSensorImpl impl) {
        super(impl);
    } // constructor(AveragingSensorImpl)
...
    /**
     * Averaging sensors produce a value that is the average of
```

```
     * measurements made over a period of time.  That period of time
     * begins when this method is called.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public void beginAverage() throws SensorException {
        ((AveragingSensorImpl)getImpl()).beginAverage();
    } // beginAverage()
} // class AveragingSensor
```

As you can see, the `AveragingSensor` class is also very simple, delegating its operations to the implementation objects that it is using. Here is its corresponding implementation class:

```
interface AveragingSensorImpl extends SimpleSensorImpl {
    /**
     * Averaging sensors produce a value that is the average of
     * measurements made over a period of time.  That period of time
     * begins when this method is called.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public void beginAverage() throws SensorException;
} // interface AveragingSensorImpl
```

It is reasonable for subclasses of the `SimpleSensorImpl` class to be more complex and provide additional services of their own. The `StreamingSensor` class delivers a stream of measurements to objects that have register to receive those measurements. It delivers those measurements by calling a method of the object it is delivering the measurement to. It does not place any requirements on how long that method may take before it returns. There is merely an expectation that the method will return in a reasonable amount of time. On the other hand, the implementation objects used with instances of the `StreamingSensor` class may need to deliver measurements at a steady rate or loose them. In order to avoid losing measurements, instances of the `StreamingSensor` class buffer measurements that are delivered to it, while it asynchronously delivers those measurements to other objects. Here is code for the `StreamingSensor` class:

```
/**
 * Instances of this class are used to represent sensors that produce
 * a stream of measurement values.
 */
public class StreamingSensor extends SimpleSensor implements StreamingSensorListener, Runnable {
    // These objects are used to provide a buffer that allows the
    // implementation object to asynchronously deliver measurement values
    // while this object is delivering value it has already received to its
    // listeners.
    private DataInputStream consumer;
    private DataOutputStream producer;

    private Vector listeners = new Vector(); // aggregate listeners here
    /**
     * Constructor
     *<p>
     * This constructor is intended to be called by a factory method
     * object that is in the same package as this class and the the
     * classes that implement its operations.
     * @param impl The object that implements the sensor type-specific
     *             operations this object will provide.
     * @exception SensorException if initialization of this object fails.
     */
    StreamingSensor(StreamingSensorImpl impl) throws SensorException {
        super(impl);
```

```
        // Create pipe stream that will support this object's ability
        // to deliver measurement values at the same time it is
        // receiving them.
        PipedInputStream pipedInput = new PipedInputStream();
        consumer = new DataInputStream(pipedInput);
        PipedOutputStream pipedOutput;
        try {
            pipedOutput = new PipedOutputStream(pipedInput);
        } catch (IOException e) {
            throw new SensorException("pipe creation failed");
        } // try
        producer = new DataOutputStream(pipedOutput);

        // start a thread to deliver measurement values
        new Thread(this).start();
    } // constructor(StreamingSensorImpl)
...
    /**
     * Streaming sensors produce a stream of measurement values.  The
     * stream of values is produced with a frequency no greater than
     * the given number of times per minute.
     * @param freq The maximum number of times per minute that this
     *             streaming sensor will produce a measurement value.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public void setSamplingFrequency(int freq) throws SensorException {
        // delegate this to the implementation object
        ((StreamingSensorImpl)getImpl()).setSamplingFrequency(freq);
    } // setSamplingFrequency(int)
    /**
     * StreamingSensor objects deliver a stream of values to
     * interested objects by passing each value to the object's
     * processMeasurement method.  The delivery of values is done
     * using its own thread and is asynchronous of everyting else.
     * @param value The measurement value being delivered.
     */
    public void processMeasurement(int value) {
        try {
            producer.writeInt(value);
        } catch (IOException e) {
            // If the value cannot be delivered, just discard it.
        } // try
    } // processMeasurement(int)
    /**
     * This method registers its argument as a recipient of future
     * measurement values from this sensor.
     */
    public void addStreamingSensorListener(StreamingSensorListener listener) {
        listeners.addElement(listener);
    } // addStreamingSensorListener(StreamingSensorListener)
    /**
     * This method unregisters its argument as a recipient of future
     * measurement values from this sensor.
     */
    public void removeStreamingSensorListener(StreamingSensorListener listener) {
        listeners.removeElement(listener);
    } // addStreamingSensorListener(StreamingSensorListener)
    /**
     * This method asynchronously removes measurement values from the pipe
     * and delivers them to registered listeners.
```

```
     */
    public void run() {
        while (true) {
            int value;
            try {
                value = consumer.readInt();
            } catch (IOException e) {
                // Pipes is broken so return from this method letting
                // ths trhead die.
                return;
            } // try
            for (int i=0; i < listeners.size(); i++) {
                StreamingSensorListener listener;
                listener = (StreamingSensorListener)listeners.elementAt(i);
                listener.processMeasurement(value);
            } // for
        } // while
    } // run()
} // class StreamingSensor
```

In order for the `StreamingSensor` class to deliver a measurement to an object, it requires that object to implement the `StreamingSensorListener` interface. It delivers measurements by passing them to the `processMeasurement` method that the `StreamingSensorListener` interface declares. The `StreamingSensor` class also implements the `StreamingSensorListener` interface. Implementation objects deliver measurements to instances of the `StreamingSensor` class by calling its `processMeasurement` method.

Finally, here is the implementation interface that corresponds to the `StreamingSensor` class:

```
interface StreamingSensorImpl extends SimpleSensorImpl {
    /**
     * Streaming sensors produce a stream of measurement values.  The
     * stream of values is produced with a frequency no greater than
     * the given number of times per minute.
     * @param freq The maximum number of times per minute that this
     *             streaming sensor will produce a measurement value.
     * @exception SensorException if there is a problem accessing the
     *                            sensor.
     */
    public void setSamplingFrequency(int freq) throws SensorException;
    /**
     * This method is called by an object than represents the
     * streaming sensor abstraction so that this object can perform a
     * call-back to that object to deliver measurement values to it.
     * @param abstraction The abstraction object to deliver
     *                    measurement values to.
     */
    public void setStreamingSensorListener(StreamingSensorListener listener);
} // interface StreamingSensorImpl
```

Related Patterns

Layered Architecture analysis pattern

The Bridge design pattern is a way of organizing the entities identified using the Layered Architecture analysis pattern into classes.

Abstract Factory/Toolkit

The Abstract Factory pattern can be used by the Bridge pattern to decide which implementation class to instantiate for an abstraction object.

Facade

Synopsis

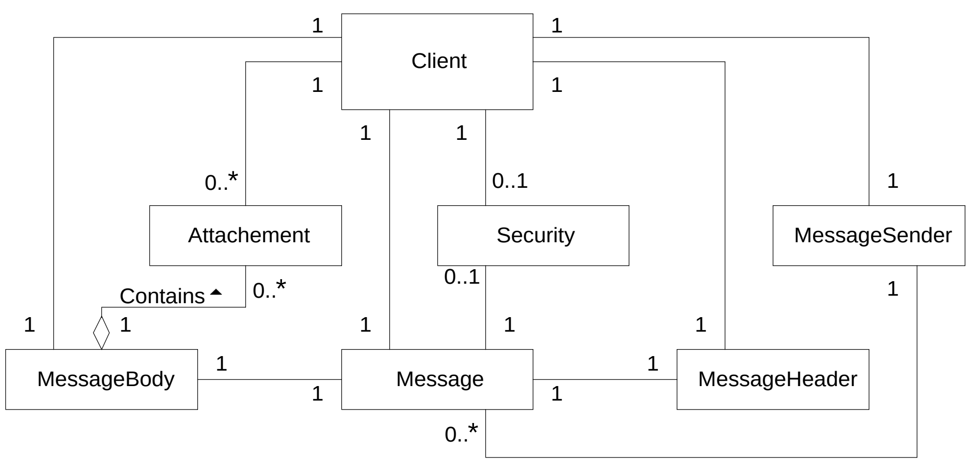
The Facade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.

Context

Consider the organization of a set of classes that supports the creation and sending of e-mail messages. A set of classes for sending e-mail messages might include the following classes:

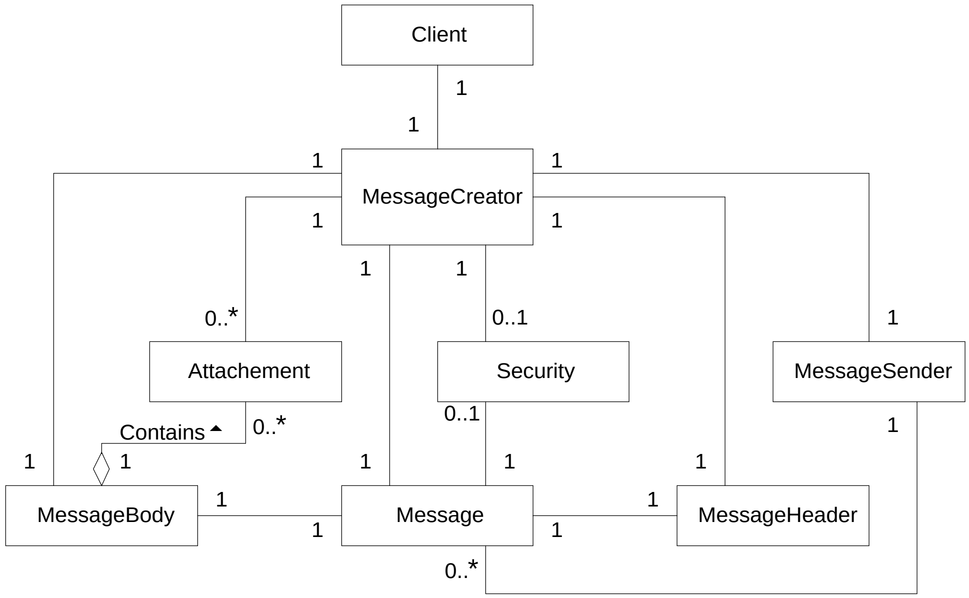
- A `MessageBody` class whose instances will contain message bodies.
- An `Attachment` class whose instances will contain message attachments that can be attached to a message body object.
- A `MessageHeader` class whose instances will contain the header information (to, from, subject…) for an e-mail message.
- A `Message` class whose instances will tie together a `MessageHeader` object and a `MessageBody` object.
- A `Security` class whose instances can be used to add a digital signature to a message.
- A `MessageSender` class whose instances are responsible for sending `Message` objects to a server that is responsible for delivering the e-mail to its destination or another server.

Here is a class diagram showing the relationships between these classes and a client class:



E-Mail Creation

As you can see, working with these e-mail classes adds complexity to a client class. To work with these classes, a client must know about at least these six classes, the relationship between them and the order in which it must create instances of those classes. If every client of these classes must take on that additional complexity, that makes the e-mail classes more difficult to reuse. The Façade pattern provides a way to shield clients of a set of classes like the e-mail classes from the complexity of using those classes. The way that it does that is to provide an additional reusable object that hides most of the complexity of working with the other classes from client classes. Here is a class diagram showing this more reusable organization:



Reusable E-Mail Creation

Client classes now need only be aware of the `MessageCreator` class. Furthermore, the internal logic of the `MessageCreator` class can shield client classes from having to create the parts of an e-mail message in any particular order.

Forces

- There are many dependencies between the classes that implement an abstraction and their client classes. The dependencies add noticeable complexity to the client classes.

- You want to simplify the client classes. Simpler client classes will result in fewer bugs. Simpler client classes also mean that less work is required to reuse the classes that implement the abstraction.

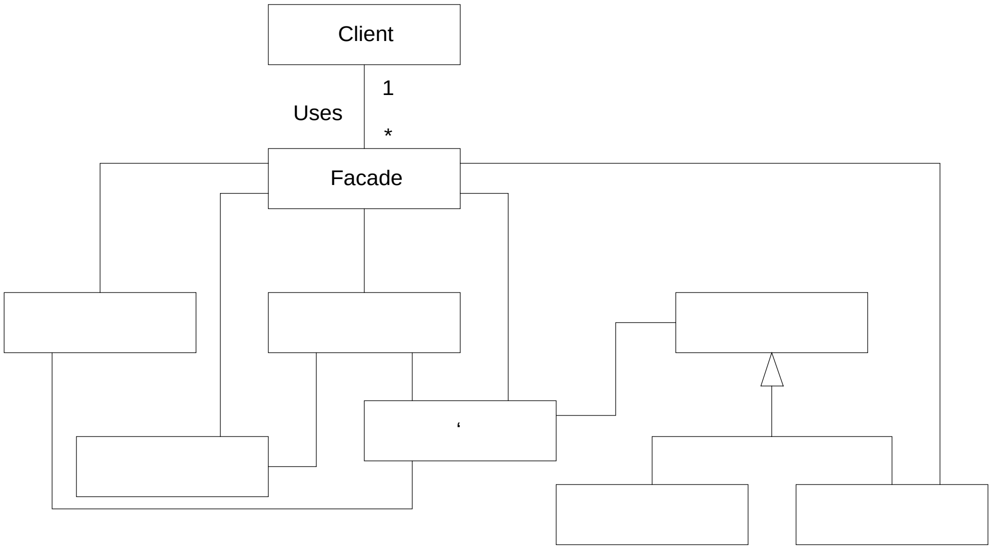
- Interposing a façade class between the classes that implement an abstraction and their client classes simplifies the client classes by moving the dependencies from the client classes to the façade class.

- It is not necessary for a façade class to act an impenetrable barrier separating the client classes from the classes that implement an abstraction. It is sufficient, and sometimes better, for a façade class to provide a default way of accessing the functionality of the classes that implement an abstraction. If some client classes need to directly access some of the abstraction implementing classes directly, then the façade class should facilitate that with a method that returns a reference to the appropriate implementation object.

  The point of the façade class is to allow simple clients, not require them.

Solution

Here is a class diagram showing the general structure of the Façade pattern:



Façade Pattern

The client object interacts with a façade object that provides necessary functionality by interacting with the rest of the objects. If there is some additional functionality that is only needed by some clients, then instead of providing it directly, the façade object may provide a method to access another object that does provide the functionality.

Consequences

Clients of façade objects do not need to know about any of the classes behind the façade.

Because the Façade pattern reduces or eliminates the coupling between a client class and the classes that implement an abstraction, it may be possible to change the classes that implement the abstraction without any impact on the client class.

Client objects that need direct access to abstraction implementing objects may access those objects.

Implementation

A façade class should provide a way for client objects to obtain a direct reference to an instance of some abstraction implementing classes. However, there may be abstraction implementing classes that clients classes have no legitimate reason to know about. The façade class should hide those classes from client classes. One way to do that is to make those classes private inner classes of the façade class.

Sometimes you want to vary the implementation classes that a façade object uses to accommodate variations on the abstraction being implemented. For example, returning to the e-mail example under the context heading, you may need a different set of classes to create MIME, MAPI or Notes compliant messages. Different sets of implementation classes usually require different façade classes. You can hide the use of different façade classes from client classes by applying the Class Decoupling pattern. Define an interface that all facade classes for e-mail creation must implement. Then have client classes access the façade class through an interface rather than directly.

JAVA API Usage

The `java.net URL` class is an example of the Façade pattern. It provides access to the contents of URLs. A class can be a client of the `URL` class and use it to get the contents of a URL without being aware of the many classes that operate behind the façade provided by the URL class. On the other hand, to send data to a URL, the client of a `URL` object may call its `openConnection` method that returns the `URLConnection` object that the `URL` object uses.

Example

Here is the code for the `MessageCreator` class show in the class diagram. It is shown here as a typical example of a Façade class.

```
/**
 * Instances of this class are used to create and send e-mail messages.
 * It assumes that an e-mail message consists of a message body and zero or
 * more attachments. The content of the message body must be provided as
 * either a String object or an object that implements an interface called
 * RichText.  Any kind of an object can be provided as the content of an
 * attachment.
 */
public class MessageCreator {
    // Constants to indicate the type of message to create
    public final static int MIME = 1;
    public final static int MAPI = 2;
    public final static int NOTES = 3;
    public final static int BANYAN = 4;

    private Hashtable headerFields = new Hashtable();
    private RichText messageBody;
    private Vector attachments = new Vector();
    private boolean signMessage;

    /**
     * Constructor to create a MessageCreator object that will create an
     * e-mail message and send it to the given address.  It will attempt to
     * infer the type of message to create from the "to" address.
     * @param to The address that this object will send a message to.
     * @param from The address that the message will say it is from.
     * @param subject The subject of this message.
     */
    public MessageCreator(String to, String from, String subject) {
        this(to, from , subject, inferMessageType(to));
    } // Constructor(String, String, String)

    /**
     * Constructor to create a MessageCreator object that will create an
     * e-mail message and send it to the given address.  It will attempt to
     * infer the type of message to create from the "to" address.
     * @param to The address that this object will send a message to.
     * @param from The address that the message will say it is from.
     * @param subject The subject of this message.
     * @param type The type of message to create.
     */
    public MessageCreator(String to, String from, String subject, int type) {
        headerFields.put("to", to);
        headerFields.put("from", from);
        headerFields.put("subject", subject);
        //...
    } // Constructor(String, String, String, int)

    /**
     * Set the contents of the message body.
     * @param messageBody The contents of the message body.
     */
```

```
    public void setMessageBody(String messageBody) {
        setMessageBody(new RichTextString(messageBody));
    } // setMessageBody(String)

    /**
     * Set the contents of the contents body.
     * @param messageBody The contents of the message body.
     */
    public void setMessageBody(RichText messageBody) {
        this.messageBody = messageBody;
    } // setMessageBody(RichText)

    /**
     * Add an attachement to the message
     * @param attachment the object to attach to the message
     */
    public void addAttachment(Object attachment) {
        attachments.addElement(attachment);
    } // addAttachment(Object)

    /**
     * set whether this message should be signed.  The default is false.
     */
    public void setSignMessage(boolean signFlag) {
        signMessage = signFlag;
    } // setSignMessage(boolean)

    /**
     * Set the value of a header field.
     * @param name The name of the field to set the value of
     * @param value The value to set the field to.
     */
    public void setHeaderField(String name, String value) {
        headerFields.put(name.toLowerCase(), value);
    } // setHeaderField(String, String)

    /**
     * Send the message.
     */
    public void send() {
        MessageBody body = new MessageBody(messageBody);
        for (int i = 0; i < attachments.size(); i++) {
            body.addAttachment(new Attachment(attachments.elementAt(i)));
        } // for
        MessageHeader header = new MessageHeader(headerFields);
        Message msg = new Message(header, body);
        if (signMessage) {
            msg.setSecurity(createSecurity());
        } // if
        createMessageSender(msg);
    } // send()

    /**
     * Infer an message type from a destination e-mail address.
     * @param address an e-mail address.
     */
    private static int inferMessageType(String address) {
        int type = 0;
...
        return type;
    } // inferMessageType(String)
```

```
    /**
     * Create a Security object appropriate for signing this message.
     */
    private Security createSecurity() {
        Security s = null;
...
        return s;
    } // createSecurity()

    /**
     * Create a MessageSender object appropriate for the type of
     * message being sent.
     */
    private void createMessageSender(Message msg) {
...
    } // createMessageSender(Message)
...
} // class MessageCreator
```

The Façade pattern places no demands on the classes that the Façade class uses. Since they contain nothing that contributes to the Façade pattern their code is not shown here.

Related Patterns

Class Decoupling

The Class Decoupling pattern can be used with the Façade pattern to allow different sets of façade and implementation classes to be used without client classes having to be aware of the different classes.

Don't Talk to Strangers

The Façade pattern is a way of satisfying with the "Don't Talk to Strangers" analysis pattern.

FacadeFacadeFacadeFacade

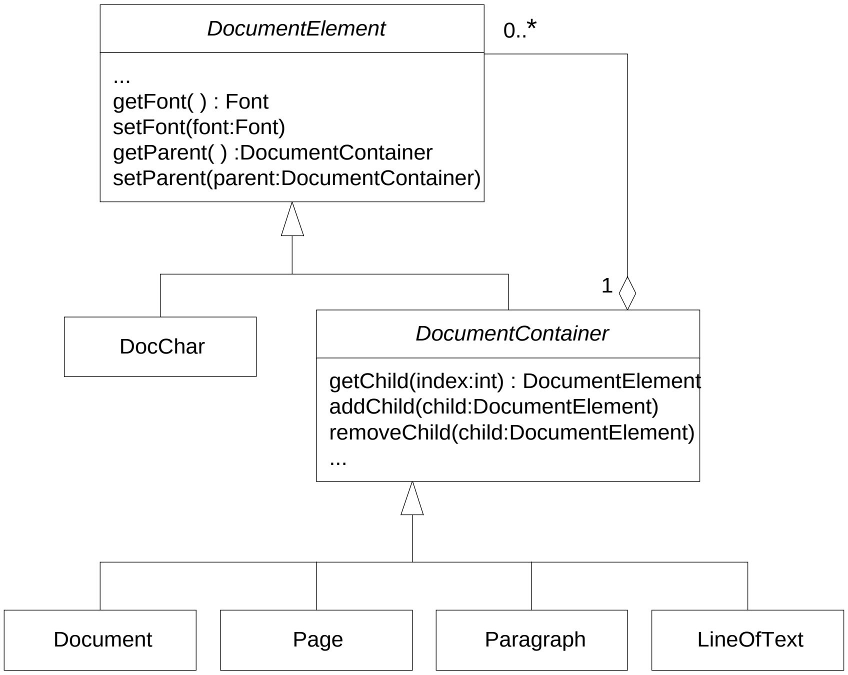
Flyweight

Synopsis

If instances of a class that contain the same information and can be used interchangeably, the Flyweight pattern allows a program to avoid the expense of multiple instances that contain the same information by sharing one instance.

Context

Suppose that you are writing a word processor. Here is a class diagram showing the basic classes you might use to represent a document:



Document Representation Classes

The above class organization includes the following classes:

- The `DocumentElement` class is ultimate superclass of all classes used to represent a document. All subclasses of the `DocumentElement` class inherit methods to set and fetch their font.

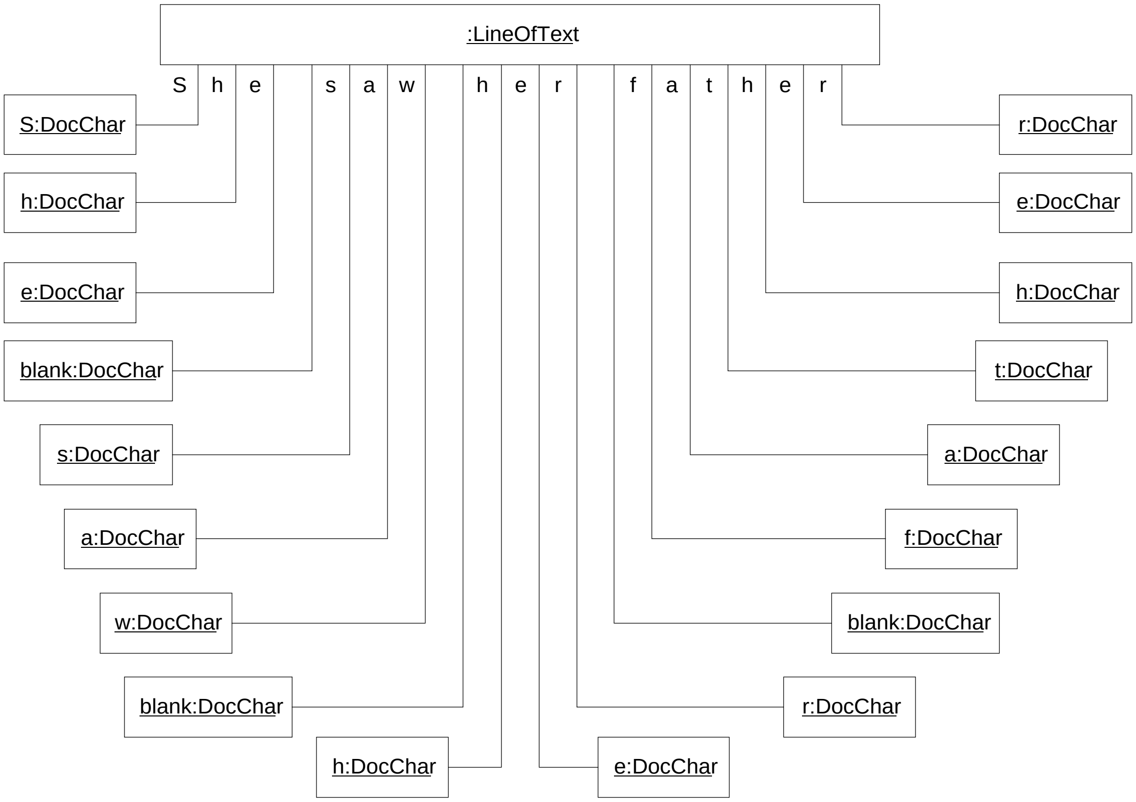
- An instance of the `DocChar` class is used to represent each character in a document.

- The `DocumentContainer` class is the superclass of container classes `Document`, `Page`, `Paragraph` and `LineOfText`.

You can specify the font of each character by calling the `setFont` method of the `DocChar` object that represents it. If character's font is unspecified, then it uses its container's font. If its container's font has not been set, then it uses its container's font.

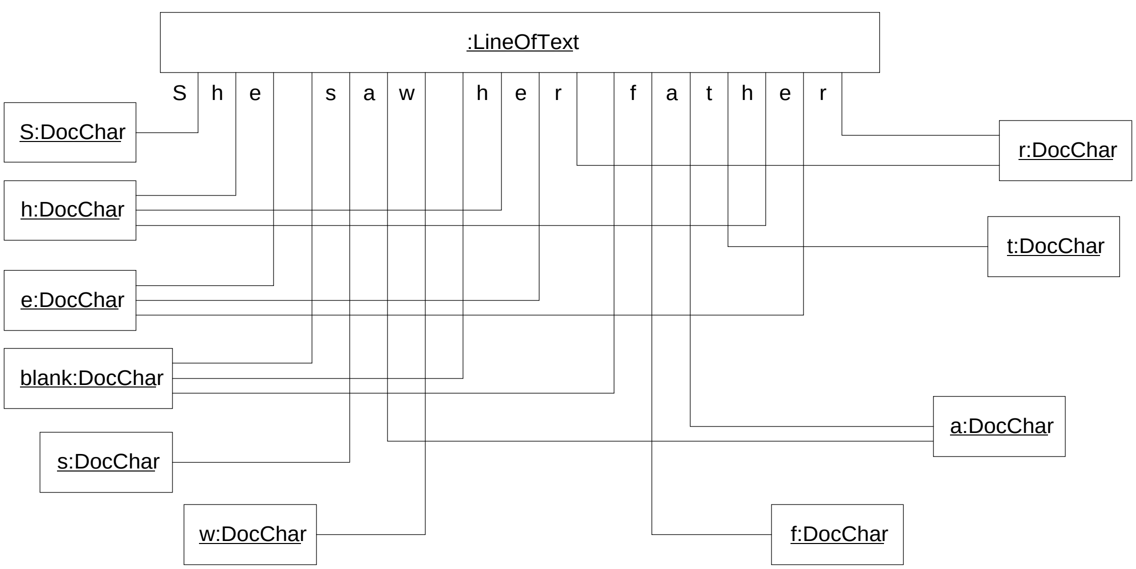
Given the above structure, one document that is a few pages long might contain tens of `Paragraph` objects that contain a few hundred `LineOfText` objects and thousands or tens of thousands of `DocChar` objects. Clearly, using this design will result in a program that uses a lot of memory to store characters.

It is possible to avoid the memory overhead of those many character objects by having only one instance of each distinct `Docchar` object. The classes in the diagram above use a `DocChar` object to represent each character in a document. To represent, "She saw her father" a `LineOfText` object uses `DocChar` objects like this:



Unshared Character Objects

As you can see, the characters ‘h’, ‘e’, ‘ ’,‘a’ and ‘e’ are used multiple times. In an entire document, all of the characters typically occur many times. It is possible to reorganize the objects so that one `DocChar` object is used to represent all occurrences of the same character like this:

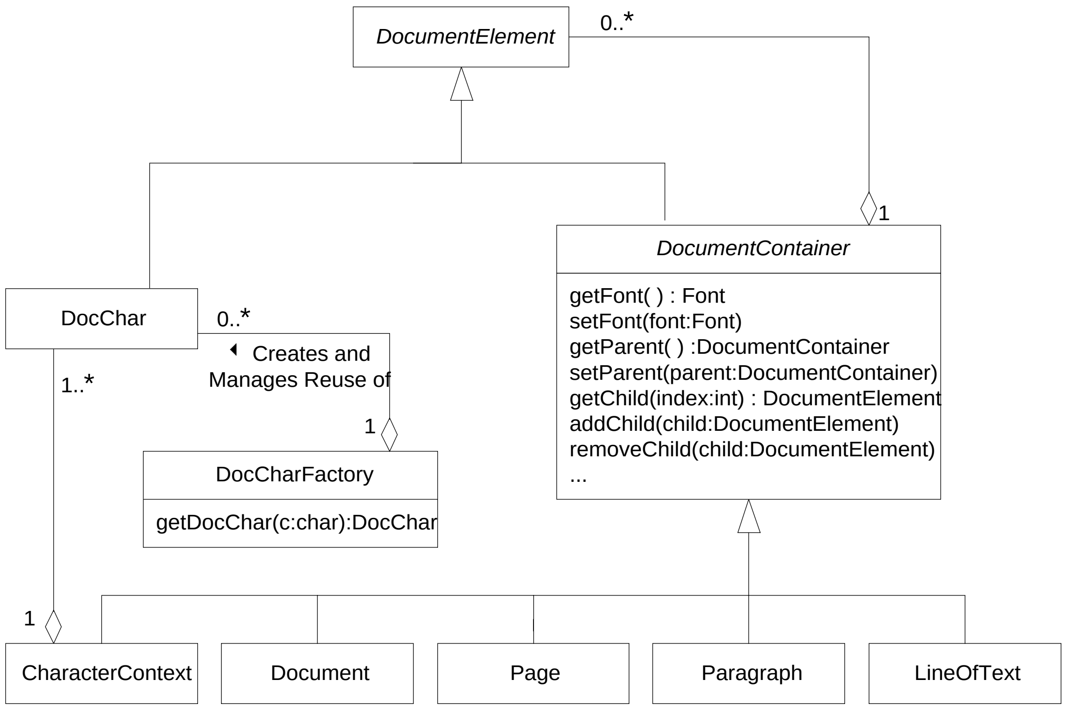


Shared Character Objects

In order to make the sharing of `DocChar` objects work, the `DocChar` objects cannot have any intrinsic attributes that are not common to every place the object is referenced. An intrinsic attribute is an attribute whose value is stored with the object. That is distinct from an extrinsic attribute, whose value is stored outside of the object that it applies to.

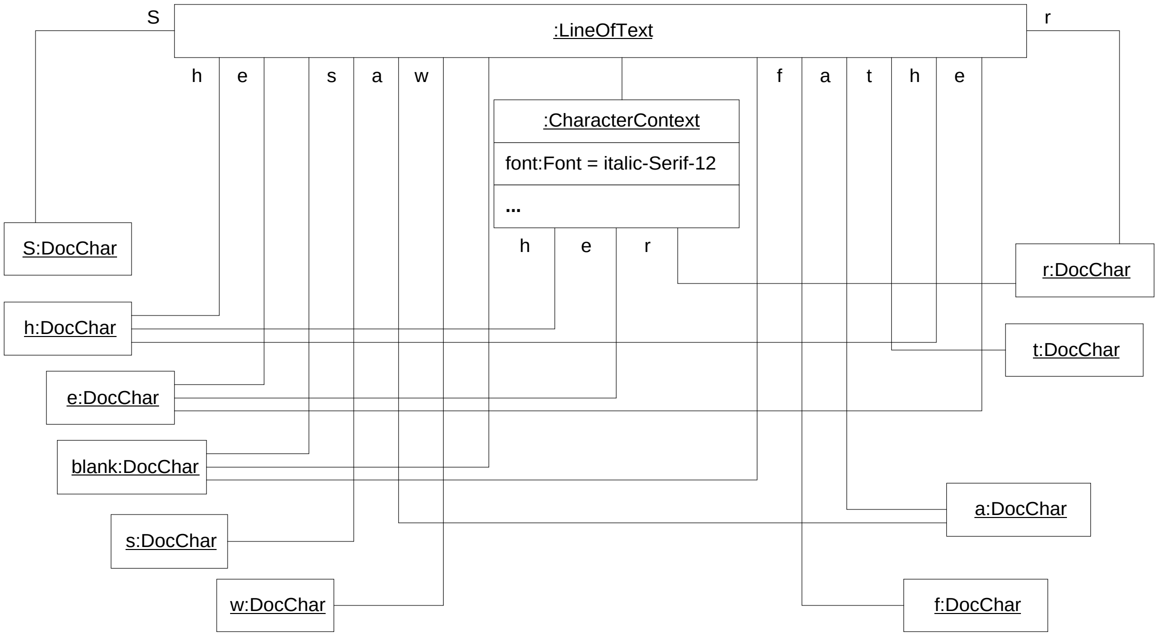
The class organization shown in the preceding class diagram captioned “Document Representation Classes” shows a `DocChar` class whose instances can have an intrinsic font attribute. Those `Character` objects that do not have a font stored intrinsically use the font of their paragraph.

To make the sharing of `DocChar` objects work, the classes need to be reorganized so that `DocChar` objects that have their own font store them extrinsically. The following class diagram includes a `ChracterContext` class whose instances store extrinsic attributes for a range of characters.



Document Shared Representation Classes

In this organization, the `DocCharFactory` class is responsible for providing a `DocChar` object that represents a given character. Given the same character to represent, a `DocCharFactory` object’s `getDocChar` method will always return the same `DocChar` object. Also, the `DocumentContainer` class defines the font methods rather than the `DocumentElement` class. All the concrete classes are subclasses of the `DocumentContainer` class, except for the `DocChar` class. That means that the `DocChar` class does not have an intrinsic font attribute. If the user wants to associate a font with a character or range of characters then the program creates a `CharacterContext` object like this:



Font in CharacterContext

Forces

- The primary motivation for using the Façade pattern is as an optimization for an application that uses a large number of similar objects.

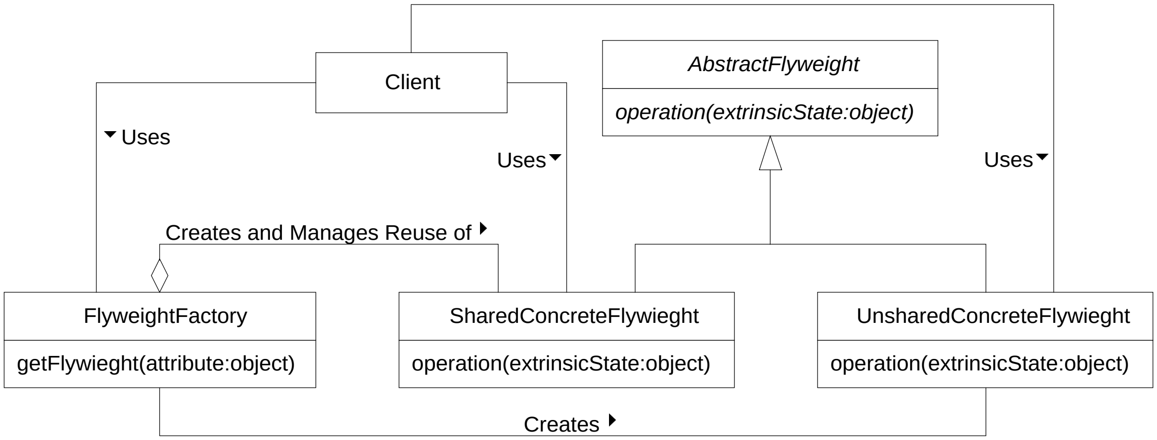
- The program does not rely on the object identity of any of the objects that you want it to share. When a program uses different objects in different contexts, it is possible to distinguish between the contexts by the object identities of the objects. When different contexts share objects, then their object identities are no longer useful for distinguishing between contexts.

- It is possible, through object sharing, to reduce a large number of similar objects to a small number of shared unique objects.

- It is possible to further reduce the number of objects by changing some of the intrinsic attributes of the unique shared object to extrinsic attributes.

Solution

The following class diagram shows the general organization of classes for the Flyweight pattern.



Flyweight Pattern

Here are descriptions of the roles that classes that participate in the Flyweight pattern play:

AbstractFlyweight
: The `AbtractFlyweight` class is the superclass of all other flyweight classes. It defines the operations common to flyweight classes. Those operations that require access to extrinsic state information obtain it as parameters.

SharedConcreteFlyweight
: Instances of classes in this role are sharable objects. If they contain any intrinsic state, it must be common to all of the entities that they represent. For example, the sharable `DocChar` objects from the example under the Context heading have the character that the represent as their intrinsic state.

UnsharedConcreteFlyweight
: Instances of classes that participate in the `UnsharedConcreteFlyweight` are not sharable. The Flyweight pattern does not require the sharing of objects. It simply allows the sharing of objects. If there are unsharable objects that are instances of the `AbstractFlyweight` class, then they will typically be instances of different subclasses of the `AbstractFlyweight` class than objects that are sharable.

FlyweightFactory
: Instances of `FlyweightFactory` classes provide instances of the `AbstractFlyweight` class to client objects. If a client object asks a `FlyweightFactory` object to provide an instance of an `UnsharedConcreteFlyweight` class, then it simply creates the instance. However, if a client object asks a `FlyweightFactory` object to provide an instance of a `SharedConcreteFlyweight` class, it first checks to see if it previously created a similar object. If it did previously create a similar object, then it provides that object to the client object. Otherwise, it creates a new object and provides that to the client.

Client
: Instances of client classes are objects that use flyweight objects.

If there is only one class in the `SharedConcreteFlyweight` role, then it may be unnecessary to have any classes in the role of `AbstractFlyweight` or `UnsharedConcreteFlyweight`.

Consequences

Using shared flyweight objects can drastically reduce the number of objects in memory. There is a price to pay for the reduced memory consumption:

- The Flyweight pattern makes a program more complex. The major sources of additional complexity are providing flyweight objects with their extrinsic state and managing the reuse of flyweight objects.

- The Flyweight pattern can increase the run time of a program because it takes more effort for an object to access extrinsic state than intrinsic state.

Usually it is possible to distinguish between entities by the objects that represent them. The flyweight pattern makes that impossible, because it results in multiple entities being represented by the same object.

Shared flyweight objects cannot contain parent pointers.

Because of the complexity that the Flyweight pattern adds and the constraints it places on the organization of classes, the Flyweight pattern should be considered an optimization to be used after the rest of a design is worked out.

Implementation

There is a tradeoff to make between the number of attributes you make extrinsic and the number of flyweight objects needed at run time. The more attributes you make extrinsic, the fewer flyweight objects will be needed. The more attributes you make intrinsic, the less time it will take objects to access their attributes.

For example, in the document representation example, if the user makes a range of characters italic, the program creates a separate `CharacterContext` object to contain the extrinsic font attribute for the `DocChar` objects that represent those characters. An alternative would be to allow the font attribute to be intrinsic to the to `DocChar` objects. If the font attribute is intrinsic then `DocChar` objects will spend less time accessing their font attribute. Letting the font attribute be intrinsic also means that the program will need a `DcoChar` object for each combination of character and font that it has to represent.

JAVA API Usage

Java uses the flyweight pattern to manage the `String` objects used to represent string literals. If there is more than one string literal in a program that consists of the same sequence of characters, Java virtual machine uses the same `String` object to represent all of those string literals.

The `String` class' intern method is responsible for managing the `String` objects used to represent string literals.

Example

Below is some of the code that implements the class diagram captioned "Document Shared Representation Classes". Some of the classes don't contain any code that is of interest with respect to the Flyweight pattern, so code for those classes not presented. For example, there is no code of interest in the `DocumentElement` class. On the other hand, the `DocumentContainer` class defines some methods that all of the container classes that are used to represent a document inherit:

```
/**
 * Instances of this class are composite objects that contain
 * DocumentElement objects.
 */
abstract class DocumentContainer extends DocumentElement {
    // Collection of this object's children
    private Vector children = new Vector();

    // This is the font associated with this object.  If the font
    // variable is null, then this object's font will be inherited
```

```
// through the container hierarchy from an enclosing object.
private Font font;

DocumentContainer parent; // this object’s container

/**
 * Return the child object of this object that is at the given
 * position.
 * @param index The index of the child.
 */
public DocumentElement getChild(int index) {
    return (DocumentElement)children.elementAt(index);
} // getChild(int)

/**
 * Make the given DocumentElement a child of this object.
 */
public synchronized void addChild(DocumentElement child) {
    synchronized (child) {
        children.addElement(child);
        if (child instanceof DocumentContainer)
          ((DocumentContainer)child).parent = this;
    } // synchronized
} // addChild(DocumentElement)

/**
 * Make the given DocumentElement NOT a child of this object.
 */
public synchronized void removeChild(DocumentElement child) {
    synchronized (child) {
        if (child instanceof DocumentContainer
            && this == ((DocumentContainer)child).parent)
          ((DocumentContainer)child).parent = null;
        children.removeElement(child);
    } // synchronized
} // removeChild(DocumentElement)

/**
 * Return this object’s parent or null if it has no parent.
 */
public DocumentContainer getParent() {
    return parent;
} // getParent()

/**
 * Return the Font associatiated with this object.  If there is no
 * Font associated with this object, then return the Font associated
 * with this object’s parent. If there is no Font associated
 * with this object’s parent the return null.
 */
public Font getFont() {
    if (font != null)
      return font;
    else if (parent != null)
      return parent.getFont();
    else
      return null;
} // getFont()

/**
 * Associate a Font with this object.
```

```
     * @param font The font to associate with this object
     */
    public void setFont(Font font) {
        this.font = font;
    } // setFont(Font)
...
} // class DocumentContainer
```

The methods shown for the `DocumentContainer` class are used to manage the state of all of the document container classes including the `CharacterContext` class. Using those inherited methods, the `CharacterContext` class is able to manage the extrinsic state of `DocChar` objects even though it doesn't declare any of its own methods for that purpose. Here is the code for the `DocChar` class:

```
/**
 * Instances of this class represent a character in a document.
 */
class DocChar extends DocumentElement {
    private char character;

    /**
     * Constructor
     * @param c the character that this object represents.
     */
    DocChar (char c) {
        character = c;
    } // Constructor(char)
...
    /**
     * Return the character that this object represents
     */
    public char getChar() {
        return character;
    } // getChar()

    /**
     * This method returns a unique value that determines where it is stored
     * internally in a hash table.
     */
    public int hashCode() {
        return getChar();
    } // hashCode()

    /**
     * Redefine equals so that two DocChar objects are considered
     * equal if they represent the same character.
     */
    public boolean equals(Object o) {
        // Call getChar rather than access character directly so that
        // this method will any alternate way a subclass has of
        // providing the character it represents.
        return (o instanceof DocChar
                && ((DocChar)o).getChar() == getChar());
    } // equals(Object)
} // class DocChar
```

Lastly, here is the code for the `DocCharFactory` class, which is responsible for the sharing of `DocChar` objects:

```
class DocCharFactory {
    private MutableDocChar myChar = new MutableDocChar();
```

```
/**
 * This is being written before the release of Java 1.2.  The
 * preliminary API documentation for Java 1.2, which is available
 * at this time but subject to change, documents a class called
 * javal.util.HashSet that will be more appropriate to use in this
 * class than java.util.Hashtable.
 */
private Hashtable docCharPool = new Hashtable();

/**
 * Return a DocChar object that represents the given character.
 * @param c The character to be represented.
 */
DocChar getDocChar(char c) {
    myChar.setChar(c);
    DocChar thisChar = (DocChar)docCharPool.get(myChar);
    if (thisChar == null) {
        thisChar = new DocChar(c);
        docCharPool.put(thisChar, thisChar);
    } // if
    return thisChar;
} // getDocChar(char)

/**
 * To allow lookups of DocChar objects in a Hashtable or simillar
 * collection, we will need a DocChar object that represents the
 * same character as the DocChar object we want to find in the
 * collection.  Creating a DocChar object to perform each lookup
 * would largely defeat the purpose of putting the DocChar objects
 * into the collection.  That purpose is to avoid creating a
 * DocChar object for each character to be represented and instead
 * use one DocChar object to represent every occurence of a
 * character.
 *<p>
 * An alternative to creatning a DocChar object for each lookup is
 * to reuse the same DocChar object, changing the character that
 * it represents for each lookup.  The problem with wanting to
 * change the character that a DocChar object represents is that
 * DocChar objects are immutable.  There is no way to change the
 * character that a DocChar object represents.
 *<p>
 * A way to get around that problem it by using this private
 * subclass of DocChar that does provide a way to change the
 * character it represents.
 */
private class MutableDocChar extends DocChar {
    private char character;

    /**
     * Constructor
     */
    MutableDocChar() {
        super('\u0000');    // It doesn't matter what we pass to super.
    } // Constructor(char)

    /**
     * Return the character that this object represents.
     */
    public char getChar() {
        return character;
    } // getChar()
```

```
        /**
         * Set the character that this object represents.
         * @param c The character that this object will represent.
         */
        public void setChar(char c) {
            character = c;
        } // setChar(char)
    } // class MutableDocChar
} // class DocCharFactory
```

Related Patterns

Recursive Composition

The Flyweight pattern is often combined with the Recursive Composition pattern to represent the leaf nodes of a hierarchical structure with shared objects.

Factory Method

The Flyweight pattern uses the factory method pattern to create new flyweight objects.

Immutable Object

Shared flyweight objects are often immutable.

Dynamic Linkage

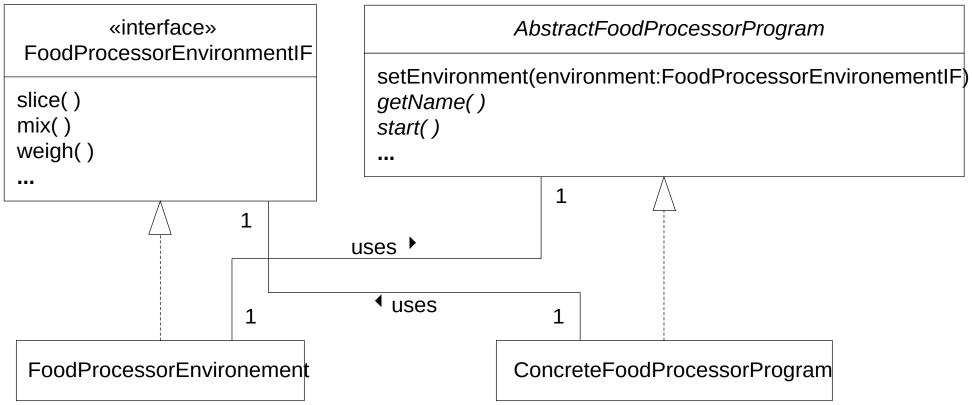
Synopsis

Allow a program, upon request, to load and use arbitrary classes that implement a known interface.

Context

Suppose that you are writing software for a new kind of smart food processor that can be fed raw ingredients and by slicing, dicing, mixing, boiling, baking, frying and stirring is able to produce cooked, ready to eat food. On a mechanical level, the new food processor is a very sophisticated piece of equipment. However, a crucial part of the food processor is a selection of programs to prepare different kinds of foods. A program that can turn flour, water, yeast and other ingredients into different kinds of bread is very different from a program that can stir-fry shrimp to exactly the right texture. The food processor will be required to run a great variety of programs that allow it to produce a great variety of foods. Because of the large variety of programs that will be required, it is not possible to build all of the necessary programs into the food processor. Instead, the food processor will load its programs from a CD-ROM or similar media.

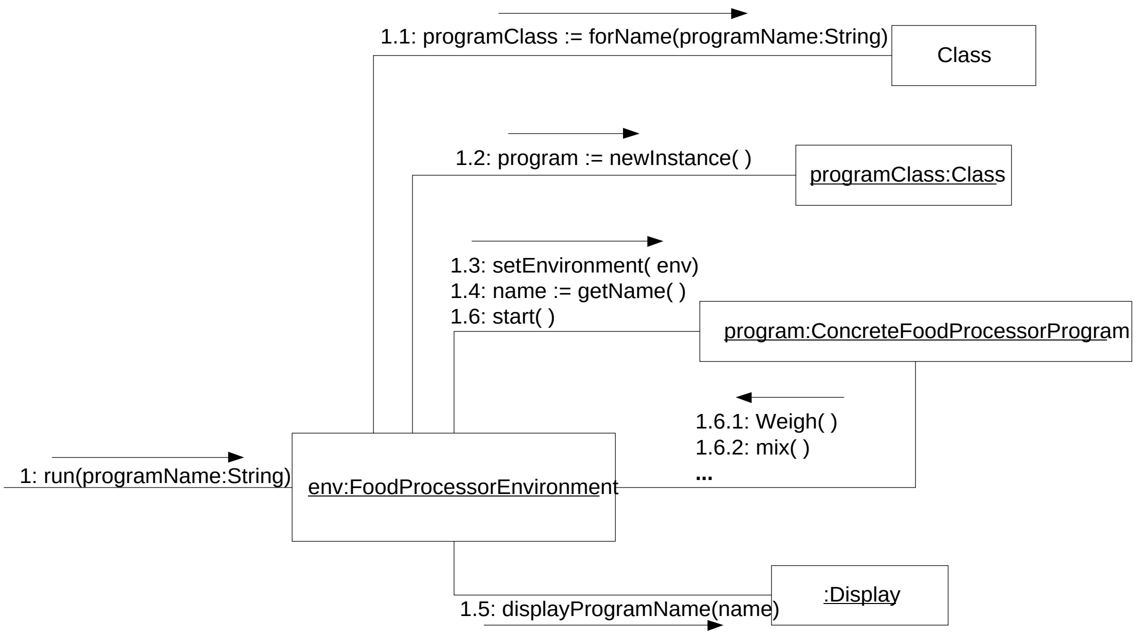
In order for these dynamically loaded programs and the food processor's operating environment to work with each other, they will need a way to call each other's methods. The following class diagram shows an arrangement of classes and interfaces that allows that:



Food Processor Program Class Diagram

The organization shown in the above class diagram allows an object in the food processor environment to call to methods of the top level object in a food processor program by calling the methods of its superclass. It also allows that top level object to call the methods of the food processor environment object through the `FoodProcessorEnvironementIF` interface that it implements.

Here is a collaboration diagram that showing these classes work together:



Food Processor Collaboration

The above collaboration diagram shows the initial steps that occur when the food processor's operating environment is asked to run a program:

1.1 The environment calls the `Class` class' `forName` method, passing it the name of the program to run. The `forName` method finds the `Class` object having the same name as the program. If necessary, it loads the class from the CD-ROM. The `forName` method concludes by returning the `Class` object that encapsulates the top-level class of the program.

1.2 The environment creates an instance of the class that is the top level class of the program. The diagram names that instance `program`.

1.3 The environment passes a reference to itself to the `program` object's `setEnvironment` method. Passing that reference to the program allows the program to call the environment's methods.

1.4 The environment gets the program's name from the program.

1.5 The environment displays the program's name.

1.6 The environment starts the program running.

1.6.1 The program weights its ingredients.

1.6.2 The program mixes its ingredients

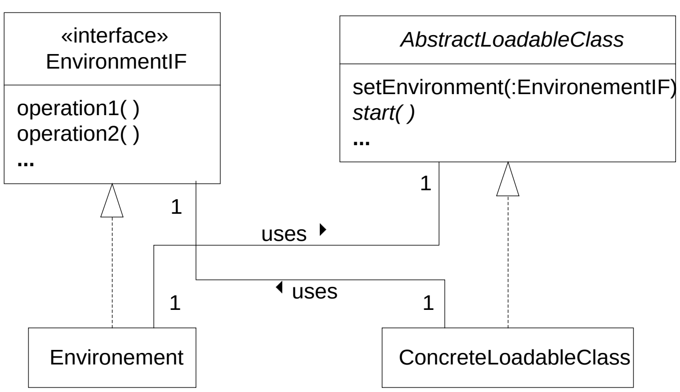
The program will continue as it executes additional steps that are beyond the scope of the drawing.

Forces

- A program must be able to load and use arbitrary classes that it has no prior knowledge of.
- A loaded class must be able to call back to the program that loaded it.

Solution

Here is a class diagram showing the interfaces and classes that participate in the Dynamic Linkage pattern:



Dynamic Linkage Pattern

Here are descriptions of the roles these classes play in the Dynamic Linkage pattern.

EnvironmentIF

An interface in this role declares the methods provided by an environment object that a loaded class can call.

Environment

A class in this role is part of the environment that loads a `ConcreteLoadableClass` class. It implements the `EnvironmentIF` interface. A reference to an instance of this class is passed to instances of the `ConcreteLoadableClass` class, so that they can call the methods of the `Environment` object that are declared by the `EnvironmentIF` interface.

AbstractLoadableClass

Any class that is the top level class of a food processor program must be a subclass of `AbstractLoadableClass`. A class in this role is expected to declare a number of other, usually abstract, methods in addition to the two that are shown. Here is a description of those methods:

- There should be a method with a name like `setEnvironment`, that allows instances of subclasses of `AbstractLoadableClass` to be passed a reference to an instance of a class that implements the `EnvironementIF` interface. The purpose of this method is to allow `AbstractLoadableClass` objects to call the methods of an `Environment` object.

- The environment calls other method, typically named `start`, to tell an instance of a loaded class to start doing whatever it is supposed to be doing.

ConcreteLoadableClass

Classes in this role are classes are subclasses of `AbstractLoadableClass` that can be dynamically loaded.

Consequences

- Subclasses of the `AbstractLoadableClass` class can be dynamically loaded.

- The operating environment and the loaded classes do not need any specific foreknowledge of each other.

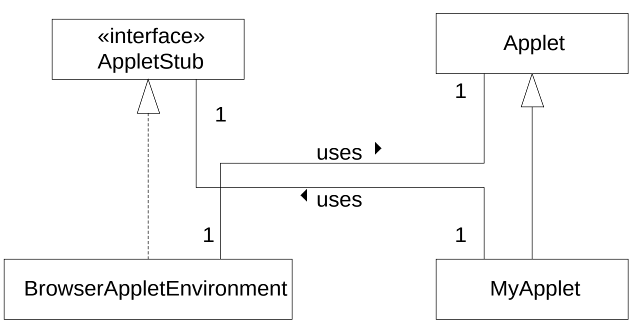
- Dynamic linkage increases the total amount of time it takes for a program to load all of the classes that it uses. However, it does have the effect of spreading out, over time, the overhead of loading. That can make an interactive program seem more responsive. The Virtual Proxy pattern can be used for that purpose.

Implementation

The Dynamic Linkage pattern, as presented, requires that the environment knows about the `AbstractLoadableClass` class and that the loaded class knows about the `EnvorinmentIF` interface. In cases where less structure than that is needed, other mechanisms for interoperation are possible. For example, JavaBeans uses a combination of reflection classes and naming conventions to allow other classes to infer how to interact with a bean.

JAVA API Usage

Web browsers use the Dynamic Linkage pattern to run applets. Here is a class diagram showing the relationship between applet and browser:



Applets and Browsers

The browser environment accesses a subclass of `Applet` that it loads through the `Applet` class. Loaded applet subclasses access the browser environment through the `AppletStub` interface.

Example

The example is the code that implements the food processor design shown under the “Context” heading. First, here is the interface for the food processor environment:

```
public interface FoodProcessorEnvironmentIF {
    /**
     * Make a slice of food of the given width.
     * @param width The width of the slice to make.
     */
    public void slice(int width) ;

    /**
     * Mix food at the given speed.
     * @param speed The speed to mix at.
     */
    public void mix(int speed) ;

    /**
     * Weight food.
     * @return the wieght in ounces.
     */
    public double weight() ;
...
} // interface FoodProcessorEnvironmentIF
```

Here is the abstract class that is the superclass for all top level program classes:

```
public abstract class AbstractFoodProcessorProgram {
    private FoodProcessorEnvironmentIF environment;

    /**
     * The food processor environment passes a reference to itself to
     * this method. That allows instances of subcalsses of this class
     * to call the methods of the food processor environement object
     * that implements the FoodProcessorEnvironmentIF interface.
     */
    public void setEnvironment(FoodProcessorEnvironmentIF environment) {
        this.environment = environment;
    } // setEnvironment(FoodProcessorEnvironmentIF)

    /**
     * Allow subclasses to fetch the reference to the environement.
     */
    protected FoodProcessorEnvironmentIF getEnvironment() {
        return environment;
    } // getEnvironment()

    /**
     * Return the name of this food processing program object.
     */
    public abstract String getName() ;

    /**
     * A call to this method tells a food processing program to start
     * doing whatever it is supposed to be doing.
```

```
     */
    public abstract void start() ;
...
} // class AbstractFoodProcessorProgram
```

Here is the class that is responsible fo the food processor environment being able to run programs:

```
public class FoodProcessorEnvironment implements FoodProcessorEnvironmentIF {
    /**
     * Make a slice of food of the given width.
     * @param width The width of the slice to make.
     */
    public void slice(int width) {
...
    } // slice(int)

    /**
     * Mix food at the given speed.
     * @param speed The speed to mix at.
     */
    public void mix(int speed) {
...
    } // mix(int)

    /**
     * Weight food.
     * @return the wieght in ounces.
     */
    public double weigh() {
        double weight = 0.0;
...
        return weight;
    } // weight()
...
    /**
     * Run the named program
     * @param programName the name of the program to run.
     */
    void run(String programName) {
        Class programClass;
        try {
            programClass = Class.forName(programName);
        } catch (ClassNotFoundException e) {
            // Not found
...
            return;
        } // try
        AbstractFoodProcessorProgram program;
        try {
            program = (AbstractFoodProcessorProgram)programClass.newInstance();
        } catch (Exception e) {
            // Unable to run
...
            return;
        } // try
        program.setEnvironment(this);
        display(program.getName());
        program.start();
    } // run(String)
...
} // class FoodProcessorEnvironment
```

Finally, here is sample code the a top level program class:

```
public class ConcreteFoodProcessorProgram
             extends AbstractFoodProcessorProgram {
    /**
     * Return the name of this food processing program object.
     */
    public String getName() { return "Chocolate Milk"; }

    /**
     * A call to this method tells a food processing program to start
     * doing whatever it is supposed to be doing.
     */
    public void start() {
        double weight = getEnvironment().weigh();
        if (weight > 120.0 && weight < 160.0)
          getEnvironment().mix(4);
...
    } // start()
...
} // class ConcreteFoodProcessorProgram
```

Related Patterns

Virtual Proxy

The Virtual Proxy pattern is often used with the Dynamic Linkage pattern.

Virtual Proxy

Synopsis

If an object is expensive to instantiate and may not be needed, it may be advantageous to postpone its instantiation until it is clear that the object is needed. The Virtual Proxy pattern hides the fact that an object may not yet exist from its clients, by having them access the object indirectly through a proxy object that implements the same interface as the object that may not exist.

Context

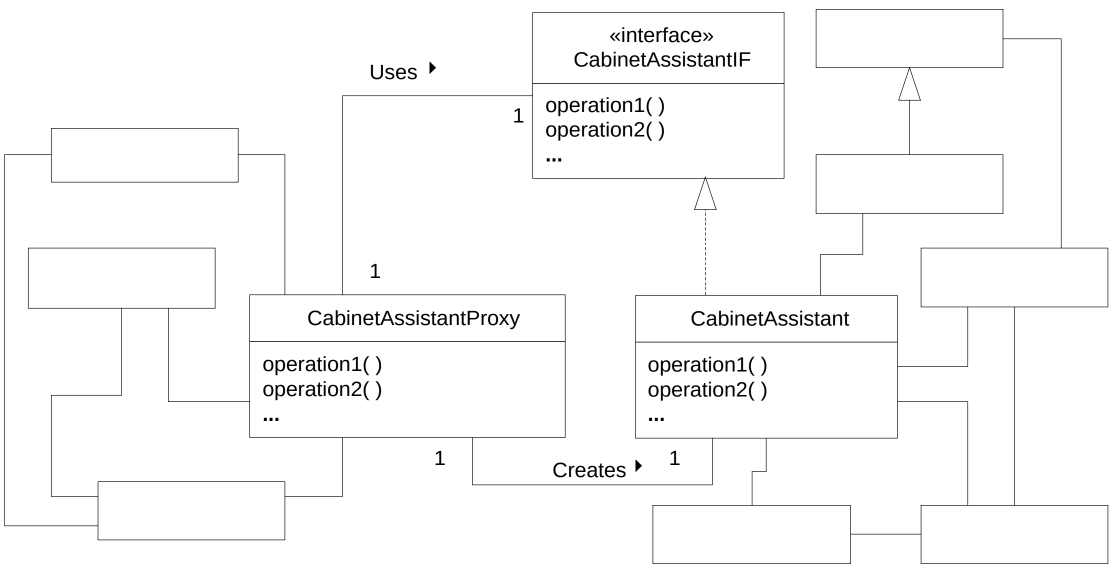
Suppose that you are part of a team that has written a large Java applet for a company that operates a chain of home improvement warehouses. The applet allows people to buy everything that the warehouses sell through a web page. In addition to offering a catalog, it includes a variety of assistants to allow customers to decide just what they need. These aides include

- A kitchen cabinet assistant that allows a customer to design a set of kitchen cabinets and then automatically order all of the pieces necessary to assemble the cabinets.
- An assistant to determine how much lumber a customer needs to build a wood deck.
- An assistant to determine the quantity of broadloom carpet needed for a particular floor plan and the best way to cut it.

There are more of these assistant, but they are not the point of this discussion. The point is that the applet is very large. Due to its size, it takes an unacceptably long amount of time for a browser to download the applet over a modem connection.

One way to reduce the time needed to download the applet is not to download any of the assistants until they are needed. The Virtual Proxy pattern provides a way to postpone downloading part of an applet in a way that is transparent to the rest of the applet. The idea is that instead of having the rest of the applet directly access the classes that comprise an assistant, they will access those classes indirectly through a proxy class. The proxy classes are specially coded so that they don't contain any static reference to the class that they are a proxy for. That means that when the proxy classes are loaded, the Java virtual machine does not see any reference to the class that those classes are a proxy for. If the rest of the applet refers only to the proxies and not to the classes that implement assistants, Java will not automatically load the assistants.

When a method of a proxy is called, it first ensures that the classes that implement the assistant are loaded and instantiated. It then calls the corresponding method through an interface. Here is a class diagram showing that organization:



Cabinet Assistant Proxy

The above diagram shows the main portion of the applet referring to a `CabinetAssistantProxy` class that implements the `CatinetAssistantIF` interface. The main portion of the applet contains no references to the classes that implement the cabinet assistant. When it is needed the `CabinetAssistantProxy` class ensures that the classes that implement the cabinet assistant are loaded and instantiated. The code that accomplishes that is below under the "Example" heading.

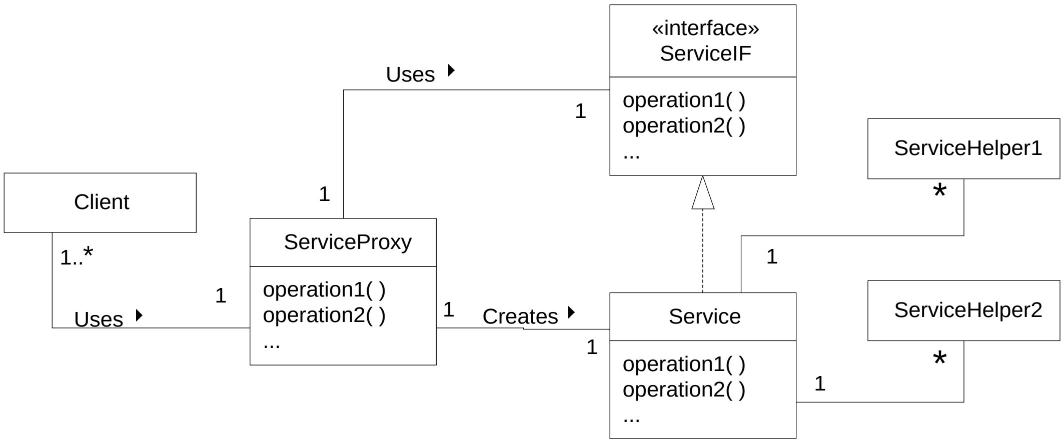
Forces

- A class is very time consuming to instantiate.
- It may not be necessary to instantiate the class.
- If there are a number of classes whose instances will not be needed until an indefinite amount of time has passed, instantiating them all at once may introduce a noticeable delay in the program's response. Postponing their instantiation until they are needed may spread out the time that the program spends instantiating them and appear to make the program more responsive.

- Managing the delayed instantiation of classes should not be a burden placed the class' clients. Therefore, the delayed instantiation of a class should be transparent to its clients.

Solution

Here is a class diagram showing the organization of classes that participate in the Virtual Proxy pattern:



Virtual Proxy Pattern

Here is an explanation of the roles played by the interface and classes of the Virtual Proxy pattern:

Service

A `Service` class provides the top level logic for a service that it provides. When you create an instance of it, it creates instances of the rest of the classes that it needs. Those classes are indicated in the diagram as `ServiceHelper1`, `ServiceHelper2`…

Client

The `Client` class is any class that uses the service provided by the `Serivce` class. `Client` classes never directly use a `Service` class. Instead, they use a `ServiceProxy` class that provides the functionality of the `Service` class. Not directly using a `Service` class keeps client classes insensitive to whether or not the instance of the `Service` class that `Client` objects indirectly use already exists.

ServiceProxy

The purpose of the `ServiceProxy` class is to delay creating instances of the `Service` class until they are actually needed.

A `ServiceProxy` class provides indirection between `Client` classes and a `Service` class. The indirection hides from `Client` objects the fact that when a `ServiceProxy` object is created, the corresponding `Service` object does not exist and the `Service` class may not even have been loaded.

A `ServiceProxy` object is responsible for creating the corresponding `Service` object. A `ServiceProxy` object creates the corresponding `Service` object the first time that it is asked to perform an operation that required the existence of the `Service` object.

A `ServiceProxy` class is specially coded to obtain access to the `Service` class through a dynamic reference. Usually, classes reference other classes through static references. A static reference simply consists of the name of a class appearing in an appropriate place in some source code. When a compiler sees that kind of reference, it generates output that causes the other class to automatically be loaded along with the class that contains the reference.

The Virtual Proxy pattern prevents the loading of the `Service` class and related classes along with the rest of the program by ensuring that the rest of the program does not contain any static references to the `Service` class. Instead, the rest of the program refers to the `Service` class through the `ServiceProxy` class and the `ServiceProxy` class refers to the `Servic` class through a dynamic reference.

A dynamic reference consists of a method call that passes a string, containing the name of a class, to a method that loads the class if it isn't loaded and returns a reference to the class. Because the name of the class only appears inside of a string, compiler are not aware that the class will be referenced and so they do not generate any output that causes that class to be loaded.

ServiceIF

A `ServiceProxy` class creates an instance of the `Service` class through method calls that do not require any static references to the `Servic` class. A `ServiceProxy` class generally also needs to call methods of the `Servic` class without having any static references to the `Service` class. It is able to do that by taking advantage of the fact that the `Service` class implements the `ServiceIF` interface.

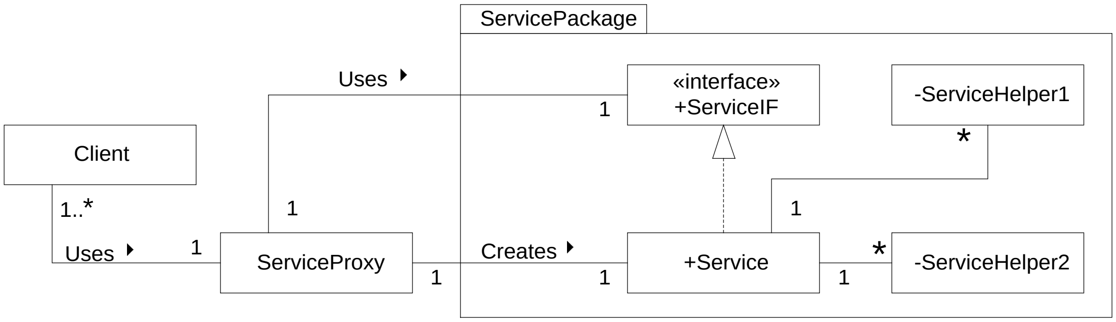
The `ServiceIF` interface is an interface that declares all of the methods that the `Service` class implements that are needed by the `ServiceProxy` class. Because of that, a `ServiceProxy` object can treat the reference to the `Service` object that it creates as a reference to a `ServiceIF` object. That means that the `Service` class can use static references to the `ServiceIF` interface to call methods of `Service` objects. No static references to the `Service` class are required.

Consequences

- Classes accessed by the rest of a program exclusively through a virtual proxy are not loaded until they are needed.
- Instances of classes accessed by the rest of a program exclusively through a virtual proxy are not created until they are needed.
- All classes other than the proxy class must access the services of the `Service` class indirectly through the proxy. That is critical. If just one class accesses the `Service` class directly, then the `Service` class will be loaded before it is needed. That is a quiet sort of bug; it generally affects performance but not function.
- Classes that use the proxy do not need to be aware of whether or not the `Service` class is loaded, an instance of it exists or that the class even exists.

Implementation

In many cases, the class accessed through a virtual proxy uses other classes that the rest of the program does not use. Because of that relationship, those classes are not loaded until the class accessed by the virtual proxy is loaded. If it is important that those classes are not loaded until the class accessed by the virtual proxy is loaded, then a problem may occur when the program is in the maintenance phase of its life cycle. A maintenance programmer may add a direct reference to one of those classes without realizing the performance implications. You can lessen the likelihood of that happening by making the relationship explicit. You can make the relationship explicit by putting the putting the classes in question in a package with only the class used by the proxy being visible outside the package:



Relationship Made Explicit by the Use of a Package

Example

To conclude the example begun under the “Context” heading, below is some of the code that implements the cabinet assistant and it proxy. First, the relevant code for the `CabinetAssistant` class:

```
/**
 * This is a skeletal example of a service class that is used by a
 * virtual proxy.  The notworthy aspect of this class is that it
 * implements an interface that is written to declare the methods of this
 * class rather than the other way around.
 */
public class CabinetAssistant implements CabinetAssistantIF {
    /**
     * constructor
     */
    public CabinetAssistant(String s) {
...
    } // Constructor(String)
...
    public void operation1() {
...
    } // operation1()

    public void operation2() {
...
    } // operation2()
} // class CabinetAssistant
```

The `CabinetAssistantIF` interface simply declares the methods defined by the `CabinetAssistant` class:

```
public interface CabinetAssistantIF {
    public void operation1();
    public void operation2();
...
} // interface CabinetAssistantIF
```

Finally, here is the code for the `CabinetAssistantProxy` class where all of the interesting things happen:

```
public class CabinetAssistantProxy {
    private CabinetAssistantIF assistant = null;
    private String myParam;     // for assistant object’s constructor

    /**
```

```
     * Constructor
     */
    public CabinetAssistantProxy(String s) {
        myParam = s;
    } // constructor(String)

    /**
     * Get the the CabinetAssistant object that is used to implement
     * operations.  This method creates it if it did not exist.
     */
    private CabinetAssistantIF getCabinetAssistant() {
        if (assistant == null) {
            try {
                // Get class object that represents the Assistant class.
                Class clazz = Class.forName("CabinetAssistant");

                // Get a constructor object to access the
                // CabinetAssistant class' constructor that takes a
                // single string argument.
                Constructor constructor;

                // Get the constructor object to create the
                // CabinetAssistant object.
                Class[] formalArgs = new Class [] { String.class };
                constructor = clazz.getConstructor(formalArgs);

                // User the constructor object.
                Object[] actuals = new Object[] { myParam };
                assistant
                  = (CabinetAssistantIF)constructor.newInstance(actuals);
            } catch (Exception e) {
            } // try
            if (assistant == null) {
                // deal with failure to create CabinetAssistant object
                throw new RuntimeException();
            } // if
        } // if
        return assistant;
    } // getCabinetAssistant()

public void operation1() {
        getCabinetAssistant().operation1();
    } // operation1()

    public void operation2() {
        getCabinetAssistant().operation2();
    } // operation2()
...
} // class CabinetAssistantProxy
```

Related Patterns

Façade

The Façade pattern can be used with the Virtual Proxy pattern to minimize the number of proxy classes that are needed.

Proxy

The Virtual Proxy pattern is a specialized version of the Proxy pattern.

Wrapper

The Wrapper pattern is also known as the Decorator pattern.

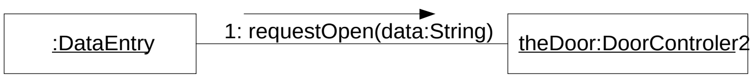
Synopsis

The Wrapper pattern extends the functionality of an object in a way that is transparent to its clients, by using an instance of a subclass of the original class that delegates operations to the original object.

The Wrapper pattern is also known as the decorator pattern because when it is applied to user interfaces it is often used to add additional user interface elements or decorations to an object.

Context

Suppose that you have become responsible for maintaining software that runs the portion of a security system responsible for controlling physical access to a building. Its basic architecture of that is that a card reader or other data entry device captures some identifying information and passes that information to an object the controls a door. If the object that controls the door is satisfied with the information, it unlocks the door. Here is a collaboration diagram showing that:

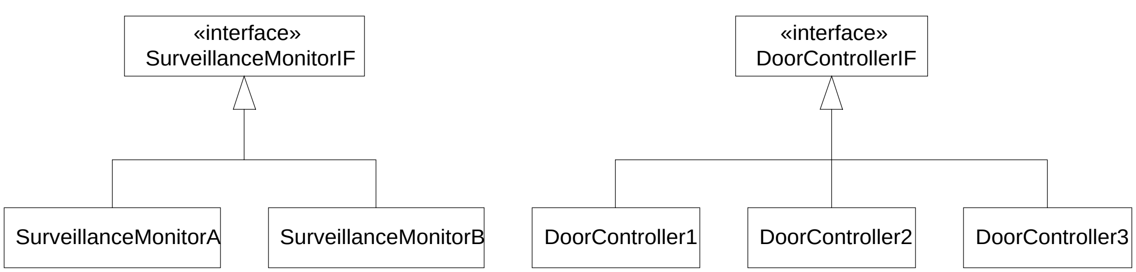


Basic Physical Access Control

Now suppose that you need to integrate this access control mechanism with a surveillance system. A surveillance system typically has many more cameras connected to it than it has TV monitors. Most of the TV monitors will cycle through the images of different cameras, showing the picture from each camera for a few seconds and then moving on to the next camera for which that monitor is responsible. There are some rules about how the surveillance system is supposed to be set up to ensure its effectiveness. For this discussion, the relevant rules are:

- At least one camera covers each doorway connected to the access control system.
- Each monitor is responsible for not more than one camera that covers an access-controlled doorway.

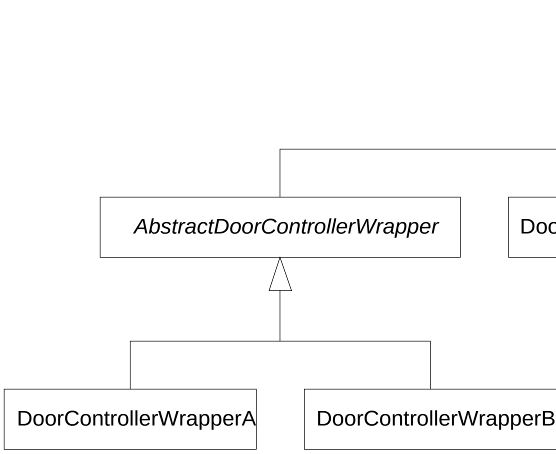
The specific integration requirement is that when an object that controls a door receives a request for that door to open, the monitors responsible for the cameras pointed at the doorway display that doorway. Your first thought about satisfying this requirement is that you will enhance a class or write some subclasses. Then you discover the relationships shown in this class diagram:



Security System Classes

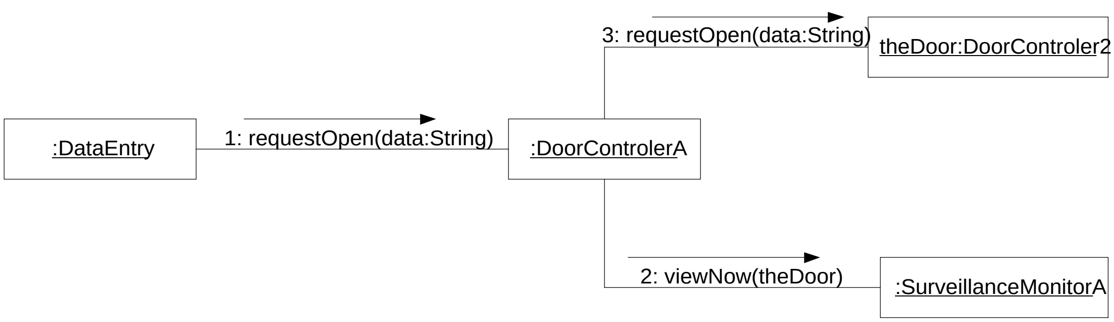
There are three different kinds of doors installed and two different kinds of surveillance monitors in use. You could resolve the situation by writing two subclasses of each of the door controller classes, but you would rather not have to write six classes. Instead, you use the Wrapper pattern that solves the problem by delegation rather than inheritance.

What you do is write two new classes called `DoorControllerA` and `DoorControllerB`. These classes both implement the `DoorControllerIF` interface:



Door Controller Classes

The new class, `AbstractDoorControllerWrapper`, is an abstract class that implements all of the methods of the `DoorController` interface with implementations that simply call the corresponding method of another object that implements the `DoorController` interface. The `DoorControllerA` and `DoorControllerB`, are concrete wrapper classes. They extend the behavior of the `requestOpen` implementation that they inherit to also ask a surveillance monitor to display its view of that doorway. Here is a collaboration diagram showing this:



Door Surveillance Collaboration

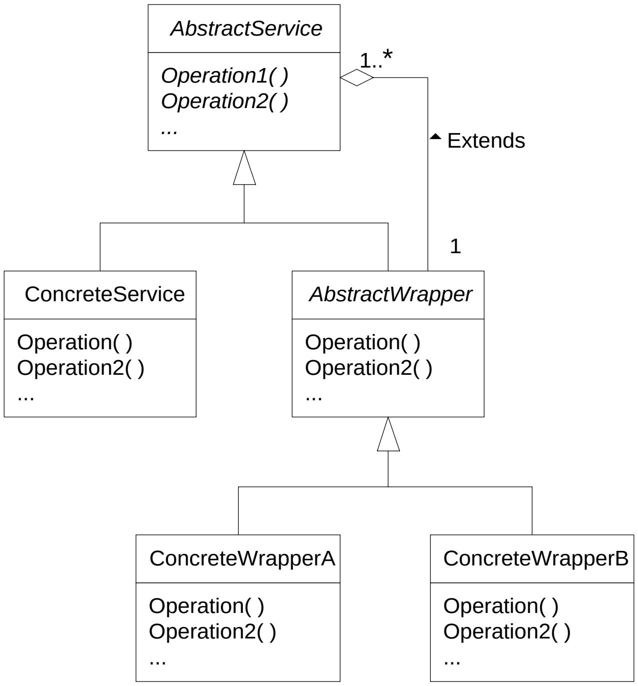
This approach allows doorways viewed by multiple cameras to be handled by simply putting multiple wrappers in front of the `DoorControllerIF` object.

Forces

- There is a need to extend the functionality of a class but there are reasons not to extend it through inheritance.
- There is the need to dynamically extend the functionality of an object and possibly also to withdraw the extended functionality.

Solution

Here is a class diagram showing the general structure of the Wrapper pattern:



Wrapper Pattern

AbstractService

An abstract class in this roll is the common superclass of all of the service objects that may potentially be extended through the Wrapper pattern. In some cases the service objects to be extended do not have a common superclass but do implement a common interface. In that case, the common interface takes the place of the abstract class.

ConcreteService

The Wrapper pattern extends classes in this role by using objects that delegate to instances of a `ConcreteService` class.

AbstractWrapper

The abstract class in this roll is the common super class for wrapper classes. This class takes responsibility for maintaining a collection of references to the service objects that wrapper objects delegate to.

This class also normally overrides all of the methods it inherits from the `AbstractService` class so that they simply call the like named method of the service object that the wrapper object delegates to. That default implementation provides exactly the behavior needed for methods whose behavior is not being extended.

ConcreteWrapperA, ConcreteWrapperB…

These concrete wrapper classes extend the behavior of the methods they inherit from the `AbstractWrapper` class in whatever way is needed.

Consequences

The Wrapper pattern provides more flexibility than inheritance. It allows you to dynamically alter the behavior of individual objects by adding and removing wrappers. Inheritance, on the other hand, determines the nature of all instances of a class statically.

By using different combinations of a few different kinds of wrapper objects, you can create many different combinations of behavior. To create that many different kinds of behavior with inheritance requires that you define that many different classes.

The flexibility of wrapper objects makes them more error prone than inheritance. For example, it is possible to combine wrapper objects in ways that do not work, or to create circular references between wrapper objects.

Using the Wrapper pattern generally results in fewer classes than using inheritance. Having fewer classes simplifies the design and implementation of programs. On the other hand, using the Wrapper pattern usually results in more objects. The larger number of objects can make debugging more difficult, especially since the objects tend to look mostly alike.

One last difficulty associated with using the Wrapper pattern is that it make using object identity to identify service objects difficult, since it hide service objects behind wrapper objects.

Implementation

Most implementations of the Wrapper pattern are simpler than the general case. Here are some of the common simplifications:

- If there is only one `ConcreteService` class and no `AbstractService` class then the `AbstractWrapper` class is usually a subclass of the `ConcreteService` class.
- Often the Wrapper pattern is used to delegate to a single object. In that case, there is no need for the `AbstractWrapper` class to maintain a collection of references. Just keeping a simple reference is sufficient.
- If there will only be one concrete wrapper class then there is no need for a separate `AbstractWrapper` class. You can merge the `AbstractWrapper` class' responsibilities with the concrete wrapper class. It may also be reasonable to dispense with the `AbstractWrapper` class if there will be two concrete wrapper classes, but no more than that.

Example

Here is some code that implements some of the door controller classes shown in diagrams under the "Context" heading. Here is the DoorControllerIF interface:

```
interface DoorControllerIF {
    /**
     * Ask the door to open if the given key is acceptable.
     * @param key A data string presented as a key to open the door.
     */
    public void requestOpen(String key);

    /**
     * close the door
     */
    public void close();
...
} // interface DoorControllerIF
```

Here is the `AbstractDoorControllerWrapper` class that provides default implementations to its subclasses for the methods declared by the `DoorControllerIF` interface:

```
abstract class AbstractDoorControllerWrapper implements DoorControllerIF {
    private DoorControllerIF wrappee;
    /**
     * Constructor
     * @param wrappee The DoorController object that this object will
     *        delegate to.
     */
    AbstractDoorControllerWrapper(DoorControllerIF wrappee) {
        this.wrappee = wrappee;
    } // constructor(wrappee)

    /**
     * Ask the door to open if the given key is acceptable.
     * @param key A data string presented as a key to open the door.
     */
    public void requestOpen(String key) {
        wrappee.requestOpen(key);
    } // requestOpen(String)

    /**
     * close the door
     */
    public void close() {
        wrappee.close();
    } // close()
...
} // class AbstractDoorControllerWrapper
```

Finally, here is one the subclasses of the `AbstractDoorControllerWrapper` class that extends the default behavior by asking a monitor to display the image from an named camera:

```
class DoorControllerWrapperA extends AbstractDoorControllerWrapper {
    private String camera;      // name of camera that views this doorway
    private SurveillanceMonitorIF monitor; // monitor for camera.

    /**
     * Constructor
     * @param wrappee The DoorController object that this object will
     *        delegate to.
     * @param camera The name of a camera that views this door
     * @param monitor The monitor to ask to view camera's image.
     */
    DoorControllerWrapperA(DoorControllerIF wrappee,
                           String camera,
                           SurveillanceMonitorIF monitor) {
        super(wrappee);
        this.camera = camera;
        this.monitor = monitor;
    } // constructor(wrappee)

    /**
     * Ask the door to open if the given key is acceptable.
     * @param key A data string presented as a key to open the door.
     */
    public void requestOpen(String key) {
        monitor.viewNow(camera);
        super.requestOpen(key);
    } // requestOpen(String)
} // class DoorControllerWrapperA
```

Related Patterns

Delegation

The Wrapper pattern is a structured way of applying the Delegation pattern.

Filter

The Filter pattern is a specialized version of the Wrapper pattern that focuses on manipulating a data stream.

Strategy

The Wrapper pattern is useful for arranging for things to happen before or after the methods of another object are called. If you want to arrange for different things to happen in the middle of calls to a method, consider using the Strategy pattern.

Template Method

The Template Method pattern is another alternative to the Wrapper pattern that allows variable behavior in the middle of a method call instead or before or after it.

Cache Management

Synopsis

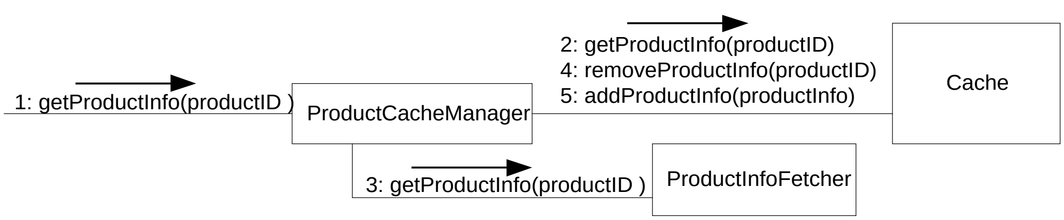
The Cache Management pattern involves keeping a local copy of objects that are fetched from outside of a program, such as a remote server or database. There reason for doing that is to save the relatively high expense of fetching such objects.

Context

Suppose that you are writing a program that allows people to fetch information about products in a catalog. Fetching all of the information for a product can take a few seconds, because it may have to be gathered from multiple sources. Keeping the information for a product in the program's memory can speed things up the next time that information for that product is requested, since it would not be necessary to spend the time to gather the information.

The technique of keeping information that takes a relatively long time to fetch into memory in memory for quick accessed the next time it is needed is called *caching*. Because there are a few hundred thousand products in the catalog, it is not feasible to cache information for all of the products in memory. What can be done is to keep information for as many of the products as feasible in memory, trying to insure that those products guessed to be the most likely to be used are in memory when they are needed. Deciding which and how many objects to keep in memory is called *cache management*.

Here is how cache management would work for the product information example:



Product Cache Management Collaboration

1. A product ID is passed to a `ProductCacheManager` object's `getProductInfo` method.

2. The `ProductCacheManager` object's `getProductInfo` method attempts to retrieve the product information from a `Cache` object. If it successfully retrieves the information form the cache, then it returns that information.

3. If it was not able to retrieve the information from the cache, then it calls a `ProductInfoFetcher` object's `getProductInfo` method to fetch the product information.

4. Cache managers generally implement a policy to limit the number of objects in a cache because keeping too many objects in the cache can be wasteful or even counterproductive. If the cache manager decides that the retrieved information should be stored in the cache and the cache already contains as many objects as it should, the cache manager will avoid increasing the number of objects in the cache. It does that by picking a product's information to remove from the cache and pass its product ID to the `Cache` object's `removeProductInfo` method.

5. Finally, if the cache manager had decided that the fetched product information should be stored in the cache, it now calls the `Cache` object's `addProductInfo` method.

Forces

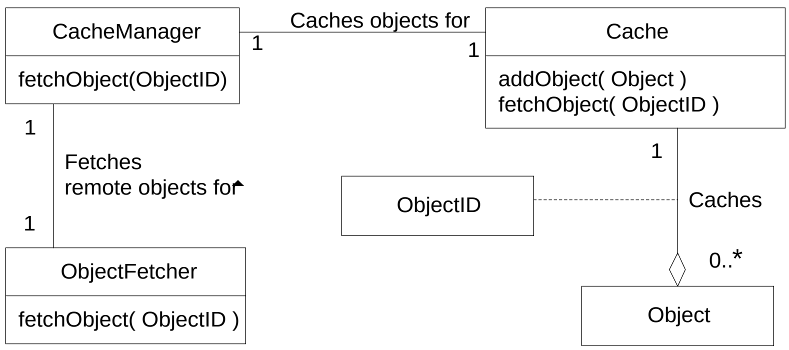
- Fetching objects from external sources can take thousands or even millions of times longer that accessing an object that is already cached in internal memory.

- When the number of objects that can be fetched from external sources is small enough that they can all fit comfortably in local memory, then keeping all of the objects in local memory will provide the best results. If there are very many objects that may potentially be fetched from external sources, then they may not fit in memory. If they do fit in memory, they may use memory that will later be needed for other purposes. Therefore, it may be necessary to set an upper bound on the number of objects cached in local memory.

- An upper bound on the number of objects in a cache requires an enforcement policy. The enforcement policy will determine which fetched objects to cache and which to discard when the number of objects in the cache reaches the upper bound. Such a policy should attempt to predict which objects are the most and least likely to be used in the near future.

Solution

Here is the general structure of the cache management pattern:



Cache Management Pattern

Here are descriptions of the classes that participate in the Cache Management pattern and the roles that they play:

ObjectID

Instances of the `ObjectID` class are used to identify objects to be fetched.

CacheManager

All requests for objects from classes that do not participate in the Cache Management pattern are presented to a `CacheManager` object by calling its `fetchObject` method. The argument to the `fetchObject` method is an `ObjecdtID` object that identifies the object to fetch. The `fetchObject` method works by first calling the `Cache` object's `fetchObject` method. If that fails, it calls the `ObjectFetcher` object's `fetchObject` method.

ObjectFetcher

`ObjectFetcher` objects are responsible for fetching objects that are not in the cache.

Cache

A `Cache` object is responsible for managing the collection of objects in the cache so that given an `ObjectID` object, it quickly finds the corresponding object. The `CacheManager` object passes an `ObjectID` object to the `Cache` object's `fetchObject` method to try to get an object from the cache. If the `CacheManager` object does not get the object it requested from the `fetchObject` method, then it requests the object from the `ObjectFetcher` object. If the `ObjectFetcher` object gives it the object that it requested, then it will pass the fetched object to this object's `addObject` method. The `addObject` method adds the object to the cache if that is consistent with its cache management policy. The `addObject` method may remove an object from the cache to make room for the object that it is adding to the cache.

Consequences

The impact of the Cache Management pattern on the rest of a program is minimal. If the `CacheManager` class is implemented as a subclass of the `ObjectFetcher` class then, using the Wrapper pattern, an implementation of the Cache Management pattern can be inserted into a working program with minimal modification to existing code.

The primary consequence of using the Cache Management pattern is that a program spends less time fetching objects from expensive sources. The simplest way of measuring the effectiveness of caching is by computing a statistic called its *hit rate*. The hit rate is the percentage of object fetch requests that the cache manager is able to satisfy with objects stored in the cache. If every request is satisfied with an object from the cache then the hit rate is 100%. If no request is satisfied then the hit rate is 0%. The hit rate depends largely on how well the implementation of the Cache Management pattern matches the way that objects are requested.

Another consequence of using the Cache Management pattern is that the cache may become inconsistent with the original data source. The consistency problem breaks down into two separate problems that can be solved independently of each other. Those problems are *read consistency* and *write consistency*.

Read consistency means that the cache always reflects updates to information in the original object source. If the objects being cached are stock prices, then the prices in the object source can change while the prices in the cache will no longer be current. To achieve absolute read consistency while caching objects in a cache, the object source must notify the program of updates to cached objects. You can accomplish that using the Publish-Subscribe pattern.

If it is not feasible to get the object source to send updates, you may be able to settle for relative read consistency. Relative read consistency does not ensure that the contents of a cache are current. Instead, the guarantee is that the objects in the cache were current within some amount of time in the past. For example, you may want to ensure that stock prices in a cache are not more than 15 minutes old. To accomplish that you can simply remove an object from the cache after it has been there for 15 minutes.

Write consistency means that updates to the the original object source always reflects updates to the cache.

Implementation

Implementing the Cache Management pattern involves making some potentially complex choices. Making optimal choices can involve much statistical analysis, queuing theory and other sorts of mathematical analysis. However, it is usually possible to produce a reasonable implementation by being aware of what the choices are and experimenting with different solutions.

The most basic decision to make when implementing the Cache Management pattern is how to implement the cache itself. The considerations for picking a data structure for the cache are:

- It must be able to quickly find objects when given their `ObjectID`.
- Since search operations will be done more frequently than addition or removal, searching should be as fast or faster than those operations.
- Since we expect frequent additions and removals of objects, the data structure must not make those operations a lot more expensive than search operations.

A hash table satisfies these needs. When implementing in Java, a cache is usually implemented using an instance of the `java.util.Hashtable` class.

The remaining implementation issues relate to performance tuning. Performance tuning is not something to spend time on until after you program is functioning correctly. In the design and initial coding stages of your development effort, make some initial decisions about how to deal the these issues and then ignore them until you are ready to deal with performance related issues.

There is always a maximum amount of memory that you can afford to devote to a cache. That means that you will have to set a limit on the objects that can be in the cache. If the potential set of objects that are available for collection in a cache is small, you don't have to impose an explicit limit. Most problems are not so conveniently self-limiting.

Specifying in advance a maximum amount of memory to devote to a cache is difficult since you may not know in advance how much memory will be available or how much memory the rest of your program will need. Enforcing a limit on the amount of memory a cache can use is especially difficult in Java because there is no definite relationship between an object and the amount of physical memory that it occupies.

An alternative to specifying and enforcing a limit that measures memory is to simply count objects. Object counting is a workable alternative to measuring actual memory usage if the average memory usage for each object is a reasonable approximation of the memory usage for each object. Counting objects is very straight forward, so you can simplify things by limiting the contents of a cache to a certain number of objects. Of course, the existence of a limit on the size of a cache raises the question of what should happen when the size of the cache reaches the maximum number of objects and another object is fetched. At that point, there is one more object than the cache is supposed to hold. The cache manager must then discard an object.

The selection of which object to discard is important because it directly affects the hit rate. If the discarded object is always the next one requested then the hit rate will be 0%. On the other hand, if the object discarded will not be requested before any of the other object in the cache, then discarding that object has the least negative impact on the hit rate. Clearly, making good choice of which object to discard requires a forecast of future object requests.

In some cases, it is possible to make an educated guess about which objects a program will need in the near future, based on knowledge of the application domain. In the most fortunate cases, it is possible to predict with high probability that a specific object will be the next one requested. In those cases, if the object is not already in the cache, it may be advantageous to load it immediately rather than wait for the program to request it. That is called *prefetching* the object.

In most cases, the application domain will not provide enough clues to make such precise forecasts. However, there is a pattern that turns up in so many cases that it is the basis for a good default strategy for deciding which object to discard. That pattern is that the more recently a program has requested an object, the more likely it is to request the object again. The strategy to from that is to always discard the least recently used object. People often abbreviate that as LRU.

Now let's take a look a setting a numeric limit on the number of objects in a cache. A mathematical analysis can give a precise value to use for the maximum number of objects that may be placed in a cache. It is unusual to do such an analysis for two reasons. The first is that the mathematical analysis involves probability and queuing theory that is beyond the knowledge of most programmers. The other reason is that such an analysis can be prohibitively time consuming. The number of details that need to be gathered about the program and its environment can be prohibitively large. However, you can usually arrive at a reasonable cache size empirically.

Begin by adding code to your `CacheManager` class to measure the hit rate as the number of object requests satisfied from the cache divided by the total number of object requests. You can the try running with different limits on the object size. As you do that, you will be looking for two things. The most important thing to look out for its that if the cache is too large is can cause the rest of you program to fail or slow down. The program can fail by running out of memory. If the program is garbage collected, as most Java programs are, it can slow down waiting for the garbage collector to finish scavenging memory for new objects. If the program is running in a virtual memory environment, a large cache can cause excessive paging.

Suppose that you want to tune a program that uses a cache. You run the program, under otherwise identical conditions, with different maximum cache sizes set. Let's say that you try values as large as 6,000. At 6,000 you find that the program takes three times as long to run as at 4000. That means that 6000 is too large. Suppose that the hit rate you got at the other values was

Max Cache Size	**Hit Rate**
250	20%
500	60%
1000	80%
2000	90%
3000	98%
4000	100%
5000	100%

Clearly, there is no need to allow the cache to be larger than 4000 objects since that achieves a 100% hit rate. Under the conditions that you ran the program, the ideal cache size is 4000. If the program will only be run under those exact conditions, then no further tuning may be needed. Many programs will be run under other conditions. If you are concerned that your program will be run under other conditions, you may want to use a smaller cache size to avoid problems under conditions where less memory is available. The number you pick will be a compromise between wanting a high hit rate and a small cache size. Since lowering the cache size to 3,000 only reduces the hit rate to 98% then 3,000 might be an acceptable cache size. If a 90% hit rate is good enough, then 2,000 is an acceptable cache size.

If it is not possible to achieve a high hit rate with available memory and fetching objects from the original data source is sufficiently expensive, then you should consider using a secondary cache. A secondary cache is typically a disk file that is used as a cache. The secondary cache takes longer to access than the primary cache that is in memory. However, if it takes sufficiently less time to fetch objects out of a local disk file than is does to fetch them from the original object source then it can be advantageous to use a secondary cache.

The way that you use a secondary cache is to move objects from the primary cache to the secondary cache instead of discarding the objects when the primary cache is full.

Example

Suppose that you are involved in writing software for an employee timekeeping system. The system consists of timekeeping terminals and a timekeeping server. The terminals are small boxes mounted on the walls of a place of business. When an employee arrives at work or leaves work, the employee notifies the timekeeping system. The employee notifies the timekeeping system by running his or her ID card through a timekeeping terminal. The terminal reads the employee's id on the card and acknowledges the card by displaying the employee's name and options. The employee then presses a button to indicate that he or she is starting work, ending work, going on break or other options. The timekeeping terminals transmit the comings and goings of each employee to the timekeeping server. At the end of each pay period, the business' payroll system gets the number of hours each employee worked from the timekeeping system and prepares paychecks.

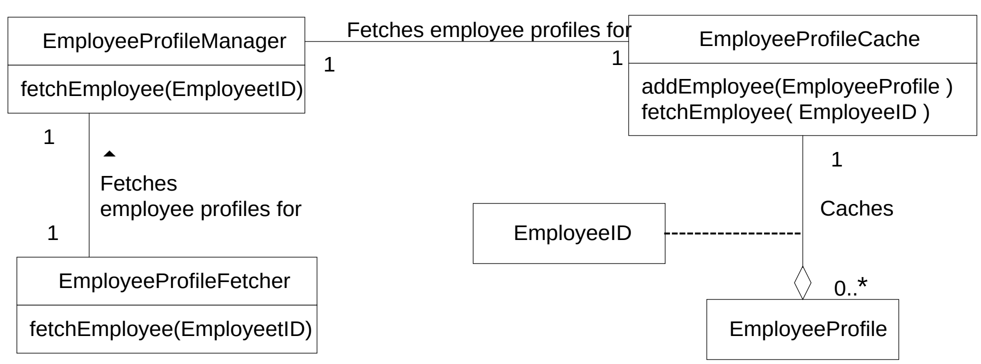
The exact details of what an employee sees will depend on an employee profile that a terminal receives from the timekeeping server. The employee profile will include the employee's name, the language in which to display prompts for the employee and what special options apply to the employee.

Most businesses assign their employees a fixed location in the business place to do their work. Employees with a fixed work location will normally use the timekeeping terminal nearest to their work location. To avoid long lines in front of timekeeping terminals, it is recommended that the terminals be positioned so that fewer than 70 employees with fixed work locations will use the same timekeeping terminal.

Because a substantial portion of the cost of the timekeeping system will be the cost of the terminals, to keep their cost down, the timekeeping terminals will have a minimal amount of memory. On the other hand, to keep response time down, we will want the terminals to cache employees profiles so that most of the time they will be able to respond immediately when presented with an employee's ID card. That means that you will have to impose a maximum cache size that is rather modest. A reasonable basis for an initial maximum cache size is the recommendation that the terminals be position so that no more than 70 employees with fixed work locations use the same terminal. Based on that we come up with an initial cache size of up to 80 employee profiles.

The reason for picking a number larger than 70 is that under some situations more than 70 employees may use the same timekeeping terminal. Sometimes one part of a business will borrow employees from another part of a business when they experience a peak workload. Also, there will be employees, such as maintenance staff, that float from one location to another.

Here is a class diagram that show how the Cache Management pattern is applied to this problem.



Timekeeping Cache Management

Here is the code that implements the timekeeping terminal's cache management. First, here is the code for the `EmployeeProfileManager` class:

```
class EmployeeProfileManager {
    private EmployeeCache cache = new EmployeeCache();
    private EmployeeProfileFetcher server
      = new EmployeeProfileFetcher();

    /**
     * Fetch an employee profile for the given employee id from the
     * internal cache or timekeeping server if the profile is not
     * found in the internal cache.
     * @param id the employee's id
     * @ return the employee's profile or null if the employee's
     *          profile is not found on the timekeeping server.
     */
    EmployeeProfile fetchEmployee(EmployeeID id) {
        EmployeeProfile profile = cache.fetchEmployee(id);
        if (profile == null) {   // if profile not in cache try server
            profile = server.fetchEmployee(id);
            if (profile != null) { // Got the profile from the server
                // put profile in the cache
                cache.addEmployee(profile);
            } // if != null
        } // if == null
        return profile;
    } // fetchEmployee(EmployeeID)
} // class EmployeeProfileManager
```

The logic in the `EmployeeProfileManager` class is rather straightforward conditional logic. The logic of the `EmployeeCache` class is more intricate, since it has to manipulate a data structure to determine which employee profile to remove from the cache when the adding an employee profile to a full cache.

```
class EmployeeCache {
    /**
     * We use a linked list to determine the least recently used employee
     * profile.  The cache that itself is implemented by a Hashtable
     * object. The Hashtable values are linked list objects that refer
     * to the actual EmployeeProfile object.
     */
    private Hashtable cache = new Hashtable();

    /**
     * This is the head of the linked list that refers to the most
     * recently used EmployeeProfile.
     */
    LinkedList mru = null;

    /**
     * this is the end of the linked list that referes to the least
     * recently used EmployeeProfile.
     */
    LinkedList lru = null;

    /**
     * The maximum number of EmployeeProfile objects that may be in the
     * cache.
     */
```

```
private final int MAX_CACHE_SIZE = 80;

/**
 * The number of EmployeeProfile objects currently in the cache.
 */
private int currentCacheSize = 0;

/**
 * Objects are passed to this method for addition to the cache.
 * However, this method is not required to actually add an object
 * to the cache if that is contrary to its policy for what object
 * should be added.  This method may also remove objects already in
 * the cache in order to make room for new objects.
 * @param emp The employeeProfile that is being proposed as an
 *            addition to the cache.
 */
public void addEmployee(EmployeeProfile emp) {
    EmployeeID id = emp.getID();
    if (cache.get(id) == null) { // if profile not in cache
        // Add profile to cache, making it the most recently used.
        if (currentCacheSize == 0) {
            // treate empty cache as a special case
            lru = mru = new LinkedList();
            mru.profile = emp;
        } else {            // currentCacheSize > 0
            LinkedList newLink;
            if (currentCacheSize >= MAX_CACHE_SIZE) {
                // remove least recently used EmployeeProfile from the cache
                newLink = lru;
                lru = newLink.previous;
                cache.remove(newLink);
                lru.next = null;
            } else {
                newLink = new LinkedList();
            } // if >= MAX_CACHE_SIZE
            newLink.profile = emp;
            newLink.next = mru;
            newLink.previous = null;
            mru = newLink;
        } // if 0
        // put the now most recently used profile in the cache
        cache.put(id, mru);
        currentCacheSize++;
    } else {                // profile already in cache
        // addEmployee shouldn’t be called when the object is already
        // in the cache.  Since that has happened, do a fetch so
        // that so object becomes the most recently used.
        fetchEmployee(id);
    } // if cache.get(id)
} // addEmployee(EmployeeProfile)

/**
 * Return the EmployeeProfile associated with the given EmployeeID
 * the cache or null if no EmployeeProfile is associated witht the
 * given EmployeeID.
 * @param id the EmployeeID to retrieve a profile for.
 */
public EmployeeProfile fetchEmployee(EmployeeID id) {
    LinkedList foundLink = (LinkedList)cache.get(id);
    if (foundLink == null)
      return null;
```

```
        if (mru != foundLink) {
            if (foundLink.previous != null)
              foundLink.previous.next = foundLink.next;
            if (foundLink.next != null)
              foundLink.next.previous = foundLink.previous;
            foundLink.previous = null;
            foundLink.next = mru;
            mru = foundLink;
        } // if currentCacheSize > 1
        return foundLink.profile;
    } // fetchEmployee(EmployeeID)

    /**
     * private doublely linked list class for managing list of most
     * recently used employee profiles.
     */
    private class LinkedList {
        EmployeeProfile profile;
        LinkedList previous;
        LinkedList next;
    } // class LinkedList
} // class EmployeeCache
```

Finally, here are the `EmployeeProfile` and `EmployeeID` classes:

```
class EmployeeProfile {
    private EmployeeID id;      // Employee Id
    private Locale locale;      // Language Preference
    private boolean supervisor;
    private String name;        // Employee name

    /**
     * Constructor
     * @param id Employee Id
     * @param locale The locale of the employee’s language of choice.
     * @param supervisor true if this employee is a supervisor.
     * @param name Employee’s name
     */
    public EmployeeProfile(EmployeeID id,
                           Locale locale,
                           boolean supervisor,
                           String name) {
        this.id = id;
        this.locale = locale;
        this.supervisor = supervisor;
        this.name = name;
    } // Constructor(EmployeeID, Locale, boolean, String)

    /**
     * Return the employee’s ID
     */
    public EmployeeID getID() { return id; }

    /**
     * return the Locale indicating the Employee’s preferred language.
     */
    public Locale getLocale() { return locale; }

    /**
     * Return true if the employee is a supervisor.
     */
```

```
    public boolean isSupervisor() { return supervisor; }
} // class EmployeeProfile


class EmployeeID {
    private String id;

    /**
     * constructor
     * @param id A string containing the employee ID.
     */
    public EmployeeID(String id) {
        this.id = id;
    } // constructor(String)

    /**
     * Returns a hash code value for this object.
     */
    public int hashCode() { return id.hashCode(); }

    /**
     * Return true if the given object is an employee id that is equal to this
     * one.
     * @param obj The object to compare with this one.
     */
    public boolean equals(Object obj) {
        return ( obj instanceof EmployeeID
                 && id.equals(((EmployeeID)obj).id) );
    } // equals(Object)

    /**
     * Return the string representation of this EmployeeID.
     */
    public String toString() { return id; }
} // class EmployeeID
```

Related Patterns

Façade

The Cache Management pattern uses the Façade pattern.

Publish-Subscribe

You can use the Publish-Subscribe pattern to ensure that read consistency of a cache.

Remote Proxy

The Remote Proxy provides an alternative to the Cache Management pattern by working with objects that exist in a remote environment rather than fetching them into the local environment.

Template Method

The Cache Management pattern uses the Template Method pattern to keep its `Cache` class reusable across application domains.

Virtual Proxy

The cache management pattern is often used with a variant of the Virtual Proxy pattern to make the cache transparent to objects that access object in the cache.