Partitioning Patterns

In the analysis stage the entities that will make up a program, along with their relationships and responsibilities, are identified. The patterns in this chapter provide guidance on how to partition complex entities into multiple classes.

Layered Initialization

Synopsis

When specialized processing is required to implement an abstraction, the most common solution is to define a class that encapsulates common logic and then define subclasses that contain the different forms of specialized logic. That does not work when some common logic must be used to decide which specialized subclass to create. The Layered Initialization pattern solves this problem by encapsulating the common and specialized logic in separate objects.

Context

Suppose that you are implementing a business rule server for an enterprise. This business rule server will be asked questions like, “What format should we use to display store numbers?” The answer to simple questions like that is normally embedded directly in one of the business rule manager’s rules. More complicated questions may require the business rule manager to consult one or more databases. Consider the question, “How far into the future can we guarantee a price quote for this item?” To answer that question, there will likely be rules that break it down into subquestions like:

- Do we have a price guarantee from our supplier and if so, when does the guarantee expire?

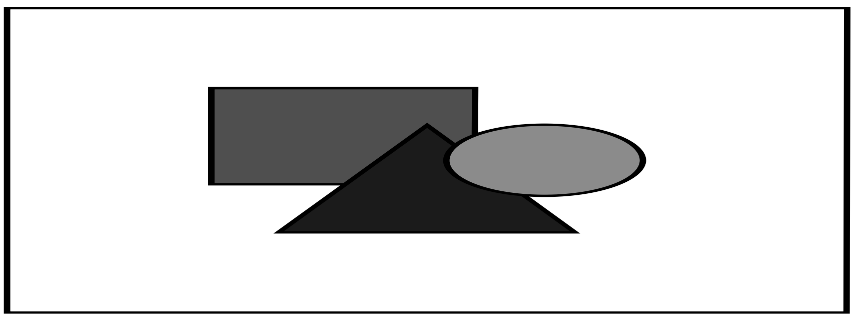
- We don’t have a price guarantee from out supplier. Based on how often the item’s price has changed in the past and our sales projections, how long will the amount of inventory we have of that item shield us from price changes?

Question such as these will require the business rule manager to query information from one or more databases. Clearly, the set of rules will be complex. Because of that, you will want to keep information about how to get different kinds of data from a database separate from the business rules that request the information.

That way, changes to the organization of the database that the business rule manager works with will not require changes to the business rules themselves.

Having determined those requirements, during analysis you will likely identify a set of entities that includes an inference engine to interpret the business rules and a data query to fetch information requested by the inference engine. Designing classes to implement the data query entity poses an interesting challenge.

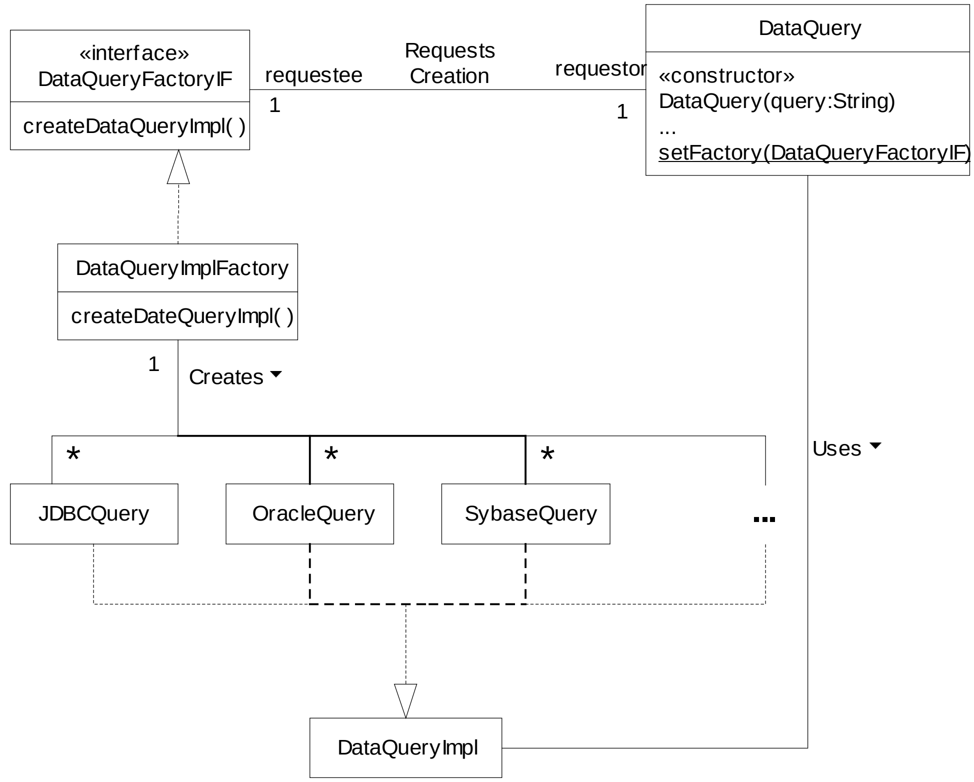
You will want to have a `DataQuery` class that you can instantiate by passing its constructor a request for information. It will be up to the constructor to determine which databases need to be queried to get the requested information. Since the techniques for getting data from a database vary with the type of database, we will want to have a class that corresponds to each type of supported database. So there may be a class for accessing relational databases through JDBC, additional classes for natively accessing relational database engines like Oracle and Sybase and perhaps another class for accessing object oriented databases. The obvious way to organize this is with a `DataQuery` class that has subclasses like this:



DataQuery

There is a problem with using this organization. Using this organization, you must decide which kind of `DataQuery` object to create before you pass the request for information to its constructor. Since you want to hide the details of data queries from the business rule inference engine, requiring it to decide which subclass of `DataQuery` to use is not a good thing.

To keep the business rule inference engine independent of which of database will be used, you will have a separate object to encapsulate the logic used to analyze the data request and determine the database that should be used. You will also want to have a factory method object that determines which class to use to access that database. Here is a class diagram showing all this:



DataQuery Factory

This design is an example of the Layered Initialization pattern. A data request is passed into the constructor for a `DataQuery` object. The constructor analyzes the data request to determine which database to consult to get the necessary information. Using an object that implements the `DataQueryFactoryIF` interface, it creates instances of the appropriate classes that implement `DataQueryImplIF`. Those `DataQueryImplIF` objects retrieve the data. The `DataQueryFactoryIF` is passed to the `DataQuery` object at an earlier time through its `setFactory` method.

You can use the Layered Initialization pattern in any situation where preprocessing must be done on selection data before deciding which specialized class to instantiate.

Forces

- A specialized class must be chosen to process complex data.
- The logic to choose a specialized class to process complex data should be encapsulated so that it is transparent to the classes providing data to process.

- To maintain low coupling, only one of the objects that participate in the Layered Initialization pattern should be visible to the object that provides the complex data.

- Putting the decision of which class to instantiate into a separate class reduces the effort required to maintain the other classes. If a database migrates to a different type of engine or a new class becomes available that provides better access to it, then the corresponding change in the program is limited to the class that decides what class to instantiate.

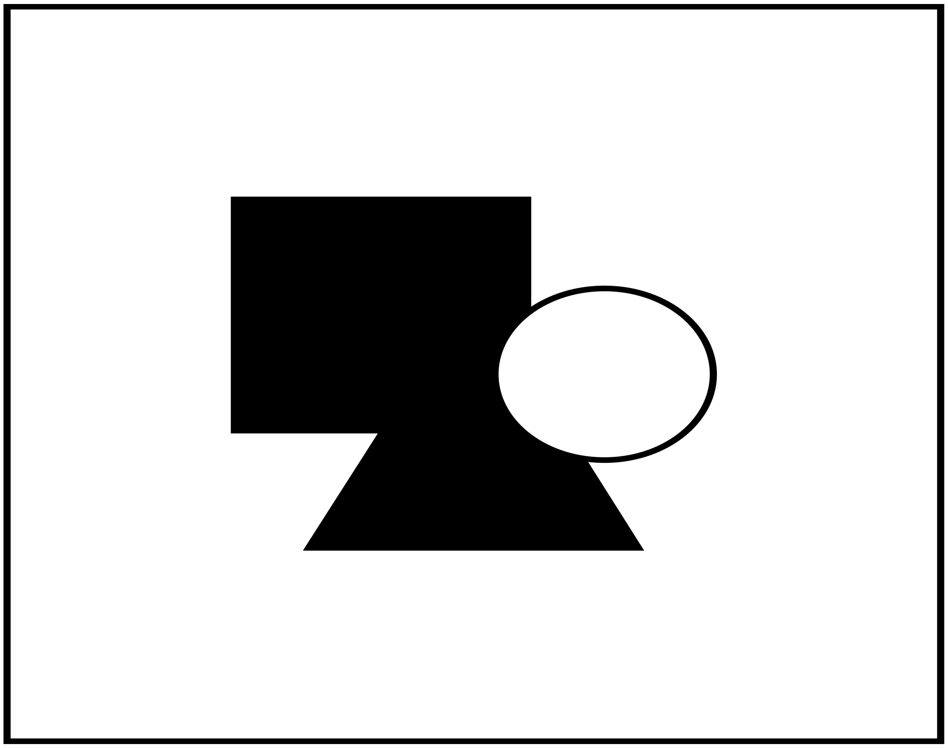
Solution

Objects that participate in the Layered Initialization pattern cooperate to provide a service to objects outside the pattern.

The essence of the Layered Initialization pattern is that initialization of the objects participating in the pattern happens in layers. First objects that perform logic common to all cases are initialized. That initialization concludes by determining the type of objects to create that will perform the next layer of more specialized logic and creating those objects. Those objects initialize themselves and create the next more specialized layer if there is one.

After the objects that participate in the Layered Initialization pattern have completed their initialization, there will be one top-level object whose methods are called by objects outside the pattern. If a method in that object requires any specialized logic, it calls the appropriate method in an object one layer down.

Here is a class diagram that shows the participants of the Layered Initialization pattern:



Layered Initialization

The preceding class diagram only shows two layers. However, using recursive composition, each of the `service1`, `service2`, … classes can be the top-level class in another application of the Layered Initialization pattern.

Here are descriptions of the participants shown in the above class diagram:

Service

All the classes that participate in the Layered Initialization pattern cooperate to provide the same service. Instances of the service class contribute to this in two ways:

- Instances of the service class are the only objects participating in this pattern that are visible to objects outside this pattern.

- The service class encapsulates logic that is common to all of the specialized cases that are supported. It delegates specialized operations and specialized portions of common operations to classes that implement the `ServiceImplIF` interface.

After a service object is sufficiently initialized to have gathered the information needed to create a specialized object that implements the `ServiceImplIF` interface, it passes that information to a `ServiceFactory` object that is responsible for the creation of those objects.

If the service class is intended to be reusable, the it will probably have a static method, indicated in the class diagram as `setFactory`, that sets the `ServiceFactory` object that all instances of the service class will use. If that sort of reusability is not needed, then neither the `setFactory` method nor the `ServiceFactoryIF` interface is needed and the service class can directly refer to a `ServiceFactory` class.

ServiceImplIF
: The service object accesses service1, service2, … objects of the lower layer through this interface.

ServiceFactoryIF
: The service object uses this interface to access `ServiceFactory` object.

ServiceFactory
: This corresponds to any class that creates `ServiceImpl` objects and implements the `ServiceFactoryIF` interface.

ServiceImpl1, ServiceImpl2…
: These classes implement the `ServiceImplIF` interface and provide the specialized logic needed by methods of the service class.

Consequences

- The complexity of initializing an object using data that requires analysis before the initialization can proceed is hidden from client objects.

- The clients of the service class do not have any dependencies on the objects participating in the Layered Initialization pattern except for the service object.

Implementation

One idea in the Layered Initialization pattern is that out of the objects participating in the Layered Implementation pattern, only the service object should have clients that are outside the pattern. A way to enforce that is to put the classes and interfaces that participate in an application of the Layered Initialization pattern into a separate package, making only the service class and `ServiceFactoryIF` interface public.

Normally the service class' `setFactory` method is called during a program's initialization. Once a factory object has been provided to the service class, it is not normally necessary to provide it with another factory object. If you know that changing the factory object will be unnecessary, then it is a reasonable assertion that calling the service class' `setFactory` method a second time is an error. If that is the case, you can make the service class more robust by putting code in the `setFactory` method that signals an error if a factory object was previous set.

JAVA API Usage

The `java.net.URL` class uses the Layered Initialization pattern.

When you create a `URL` object, you can pass a string specifying a URL to the object's constructor. These strings can look like

```
http://www.mindspring.com/~mgrand
```

or

```
mailto:mgrand@mindspring.com
```

The portion of the string before the first colon is the protocol to use for the URL. The syntax of what follows the colon depends on the protocol specified before the colon. Because the `URL` object must parse the entire string before its initialization is complete, it uses the Layered Initialization pattern.

The `URL` class participates in the Layered Initialization pattern as the service class. There is an abstract class that it uses called `URLStreamHandler`. The `URLStreamHandler` class participates in the Layered Initialization pattern as a `ServiceFactoryImplIF` interface. To parse the portion of the URL string after the colon, the `URL` class creates an instance of the appropriate subclass of the `URLStreamHandler` class. It can pick the subclass of `URLStreamHandler` to instantiate using a default mechanism. Alternatively, if an object that implements the `URLStreamHandlerFactory` interface is passed to the URL class' `setURLStreamHandlerFactory` static method then all instances of URL will use that object to indirectly create instances of the appropriate subclass of `URLStreamHandler`.

Example

The example for the Layered Initialization pattern is some skeletal code that implements the data query design shown under the Context heading of this pattern. Here is code for the `DataQuery` class that takes a data query in its constructor so that its instances can produce a result:

```
public class DataQuery {
    // Factory object for creating DataQueryImplIF objects.
    private DataQueryFactoryIF factory;
    /**
     * Set the factory object
     * @exception Error if this method is called after a factory has
     *                  been set
     */
    public void setFactory(DataQueryFactoryIF factory) {
        if (this.factory != null)
          throw new Error(ÒData query factory already definedÓ);
        this.factory = factory;
    } // setFactory(DataQueryFactoryIF)
    /**
     * Constructor
     * @param query A string containing the query
     */
    public DataQuery(String query) {
        ...
        while ( É ) {
            String dbName = null;
            ...
            // Construct a database specific query object
            DataQueryImplIF dq;
            dq = (DataQueryImplIF)factory.createDataQueryImpl(dbName);
            ...
        } // while
        //...
    } // Constructor(String)
    ...
} // class DataQuery
```

Here is the declaration of the `DataQueryFactoryIF` interface that all factory objects that create database specific query objects must implement:

```
public interface DataQueryFactoryIF {
    /**
     * Create a DataQueryImplIF object that retrieves data from the
     * specified database.
     * @param dbName the name of the database that will be queried
```

```
     * @return An instance of a class that that is specific to either
     *         JDBC or the physical database engine that the database
     *         runs on.
     *         If the specified database is not know to this method, it
     *         returns null.
     */
    public DataQueryFactoryIF createDataQueryImpl(String dbName);
} // DataQueryFactoryIF
```

Here is a sample class that implements the `DataQueryFactoryIF` interface:

```
class MyDataQueryFactory implements DataQueryFactoryIF {
    private static Hashtable classes = new Hashtable();
    // populate the classes hashtable
    static {
    classes.put("INVENTORY", dataQuery.OracleQuery.class);
    classes.put("SALES",     dataQuery.SybaseQuery.class);
    classes.put("PERSONNEL", dataQuery.OracleQuery.class);
    classes.put("WHEATHER",  dataQuery.JDBCQuery.class);
    ...
    }
    /**
     * Create a DataQueryImplIF object that retrieves data from the
     * specified database.
     * @param dbName the name of the database that will be queried
     * @return An instance of a class that that is specific to either
     *         JDBC or the physical database engine that the database
     *         runs on.
     *         If the specified database is not know to this method, it
     *         returns null.
     */
    public DataQueryFactoryIF createDataQueryImpl(String dbName) {
        Class clazz = (Class)classes.get(dbName);
        try {
            return (DataQueryFactoryIF)clazz.newInstance();
        } catch (Exception e) {
            return null;
        } // try
    } // createDataQueryImpl(String)
} // class MyDataQueryFactory
```

Related Patterns

Delegation (When not to use Inheritance)

The service class delegates specialized operations to objects that implement the `ServiceImpl` interface.

Facade

The Layered Initialization pattern uses the Facade pattern by hiding all of the other objects participating in the pattern from clients of service objects.

Factory Method

In situations where the choice of which kind of object to create does not involve any significant preprocessing of data, the Factory Method pattern may be a more appropriate choice.

Layered Architecture

The Layered Initialization pattern recognizes a division of responsibilities into layers during design. The Layered Architecture pattern recognizes a division of responsibilities into layers during analysis.

Recursive Composition

When more than two layers of initialization are needed for initialization you can combine the Layered Initialization pattern with the Recursive Composition pattern to multiple perform initialization in as many layers as needed. Layered InitializationLayered InitializationLayered InitializationLayered InitializationLayered InitializationLayered Initialization

Filter

Synopsis

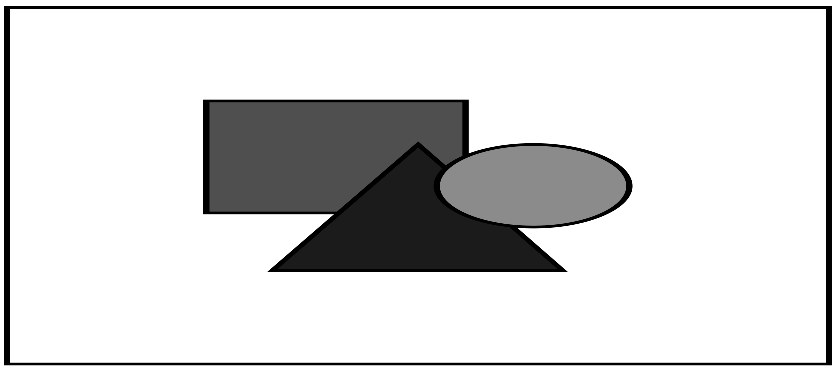
The Filter pattern allows objects that have compatible interfaces and perform different transformations and computations on streams of data to be dynamically connected to perform arbitrary operations on streams of data.

Context

There are many programs whose entire purpose is to perform computations on or perform analysis of a stream of data. A program that performs simple transformations on the contents of a data stream is the UNIX `uniq` program. The `uniq` program organizes its input into lines. The `uniq` program normally copies all of the lines that it reads to its output. However, when it finds consecutive lines that contain identical characters, it only copies the first to its output. UNIX also comes with a program called `wc` that does a simple analysis of a data stream. it produces a count of the number of characters, words and lines that were in the data stream. Compilers perform a complex series of transformations and analysis on their source code input to produce their binary output.

Since many programs perform transformations and analysis on data streams, it would clearly be beneficial to define classes that perform the more common transformations and analyses. Such classes will get a lot of reuse.

Classes that perform simple transformations and analysis on data streams tend to be very generic in nature. When writing such classes, it is not possible to anticipate all the possible ways they will be used. Some applications will want to apply some transformations and analyses to only selected parts of a data stream. Clearly, these classes should be written in a way that allows great flexibility in how their instances can be connected together. One way to accomplish that flexibility is to define a common superclass for all of these classes so an instance of one can use an instance of another without have to care which class the object is an instance of.



File Filters

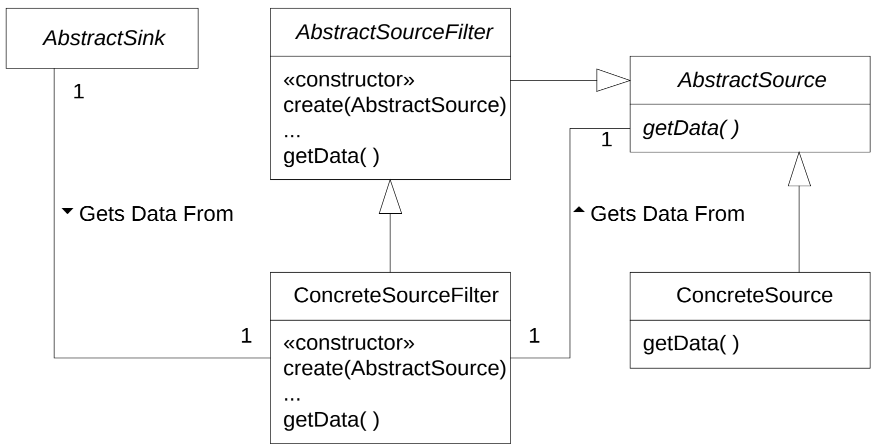
Forces

- Classes that implement common data transformations and analyses can be used in a great variety of programs.
- It should be possible to dynamically combine data analysis and transformation objects by connecting them together.
- The use of transformation/analysis objects should be tranparent to other objects.

Solution

Through a combination of abstract classes and delegation, a solution can be arrived at. The Filter pattern organizes the classes that participate in it as data sources, data sinks and data filters. The data filter classes perform the transformation and analysis operations. There are two basic variations on the Filter pattern. In one, data flows as a result of a data sink object calling a method in a data source object. In the other, data flows when a data source object passes data to a method of a data sink object.

Here is a class diagram for the version of Filter where data sink objects get data by calling methods in data sources.



Source Filter

Here are descriptions of how the classes in the above diagram participate in the Filter pattern:

AbstractSource
: This abstract class declares a method, indicated in the diagram as getData, that returns data when it is called.

ConcreteSource
: This corresponds to any concrete subclass of `AbstractSource` that is primarily responsible for providing data rather than transforming or analyzing data.

AbstractSourceFilter
: This abstract class is the superclass of classes that transform and analyze data. It has a constructor that takes an argument that is the instance of the abstract source object class that instances of this class will delegate the fetching of data to. Because instances of this class are also instances of the abstract source class, instances of the abstract sink class can treat

instances of the `AbstractSourceFilter` class the same as instances of a concrete source class.

The abstract source filter class defines a `getData` method that simply calls the `getData` method of the abstract source object that was passed to its object's constructor.

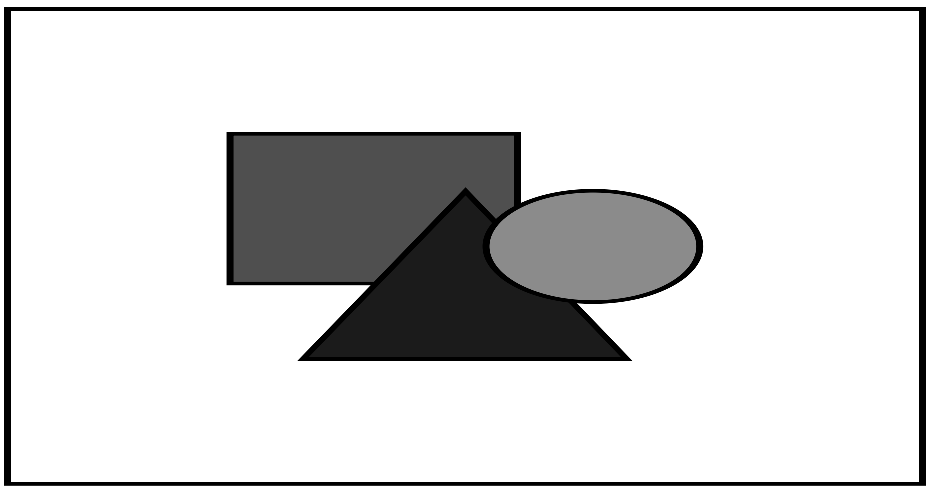
ConcreteSourceFilter

This corresponds to any concrete subclass of `AbstractSourceFilter`. Subclasses of `AbstractSourceFilter` should extend the behavior of the `getData` method that they inherit from `AbstractSourceFilter` to perform the appropriate transformation or analysis operations.

AbstractSink

Instances of abstract sink classes call the `getData` method of a `AbstractSouce` object. Unlike `ConcreteSourceFilter` objects, instances of abstract sink classes use the data without passing it on to another `AbstractSourceFilter` object.

Here is a class diagram for the version of Filter where data source objects pass data to methods of data sink objects:



Sink Filter

AbstractSink

This abstract class declares a method, indicated in the diagram as `putData`, that takes data through one of its parameters.

ConcreteSink

This corresponds to any concrete subclass of `AbstractSink` that is primarily responsible for receiving data rather than transforming or analyzing data.

AbstractSinkFilter

This abstract class is the superclass of classes that transform and analyze data. It has a constructor that takes an argument that is the instance of the abstract sink object class that instances of this class will delegate the putting of data to. Because instances of this class are also instances of the abstract sink class, instances of the abstract source class can treat instances of the `AbstractSinkFilter` class the same as instances of a concrete sink class.

The `AbstractSinkFilter` class defines a `putData` method that simply calls the `putData` method of the `AbstractSink` object that was passed to its object's constructor.

ConcreteSinkFilter

This corresponds to any concrete subclass of `AbstractSinkFilter`. Subclasses of `AbstractSinkFilter` should extend the behavior of the `putData` method that they inherit from `AbstractSinkFilter` to perform the appropriate transformation or analysis operations.

AbstractSource

Instances of `AbstractSource` call the `putData` method of an `AbstractSink` object. Unlike `ConcreteSinkFilter` objects, instances of `AbstractSource` provide the data without getting it from another `AbstractSinkFilter` object.

Consequences

The portion of a program the follows the Filter pattern can be structured as a set of sources, sinks and filters.

Filter objects that do not maintain internal state can be dynamically replaced while a program is running. This property of stateless filters allows dynamic change of behavior and adaptation to different requirements at run time.

It is quite reasonable for a program to incorporate both versions of the Filter pattern. However, it is unusual for the same class to participate in both versions.

Implementation

Filter classes should be implemented in a way that does not assume anything about the programs that they will be used in or with which other filter classes they will be used. Because of that, it follows that filter objects should not have side effects and should communicate with each other only through the date that they exchange.

Making filter classes independent of the programs that they are used in increases their reusability. However, in some cases there can be performance penalties if a filter object is not allowed to use context-specific information. The best design is sometimes a compromise between these considerations. For example, you could define one or more interfaces that declare methods for providing context specific information to a filter object. If a program detects that a filter object implements one of those interfaces, it can use the interface to provide additional information to the filter.

JAVA API Usage

The `java.io` package includes the `FilterReader` class that participates in the Filter pattern as an abstract source filter class. The corresponding abstract source class is Reader. Concrete subclasses of the `FilterReader` class include `BufferedReader`, `FileReader` and `LineNumberReader`.

The `java.io` package includes the `FilterWriter` class, which participates in the Filter pattern as an abstract sink filter class. The corresponding abstract sink class is Writer. Concrete subclasses of the `FilterWriter` class include `BufferedWriter`, `FileWriter` and `PrintWriter`.

Here is a common arrangement of `FilterReader` objects for a program that reads lines of text as a commands and needs to track line numbers for producing error messages:

```
LineNumberReader in;
void init(String fName) {
    FileReader fin;
        try {
            fin = new FileReader(fName);
            in = new LineNumberReader(new BufferedReader(fin));
        } catch (FileNotFoundException e) {
            System.out.println("Unable to open "+fName);
        }
    ...
```

Example

For an example of classes that implement the Filter pattern, here are classes that read and filter a stream of bytes. Firstly, here is a class that participates in the Filter pattern as an abstract source:

```
/**
 * Abstract class for reading a stream of bytes into an byte[].
 */
public abstract class InStream {
    /**
     * Read bytes from a stream of bytes and fill an array with those
     * bytes.
     * @param array The array of bytes to fill.
     * @return If not enough bytes are available to fill the array
     *         then this method returns after having only put that many
     *         bytes in the array.  This methods returns -1 if the end
     *         of the data stream is encountered.
     * @exception IOException  if a I/O error occurs.
     */
    public abstract int read(byte[] array) throws IOException;
} // class InStream
```

Here is a concrete subclass of `InStream` that participates in the Filter pattern as a concrete source:

```
/**
 * This class reads a stream of bytes from a file.
 */
public class FileInStream extends InStream {
    private RandomAccessFile file;
    /**
     * Constructor
     * @param fName The name of the file to read
     */
    public FileInStream(String fName) throws IOException {
        file = new RandomAccessFile(fName, ÒrÓ);
    } // Constructor(String)
    /**
     * Read bytes from a file and fill an array with those bytes.
     * @param array The array of bytes to fill.
     * @return If not enough bytes are available to fill the array
     *          then this method returns after having only put that
     *          many bytes in the array.  This methods returns -1 if
     *          the end of the data stream is encountered.
     * @exception IOException  if a I/O error occurs.
     */
    public int read(byte[] array) throws IOException {
```

```
        return file.read(array);
    } // read(byte[])
} // class FileInStream
```

The following class participates in the Filter pattern as an abstract source filter:

```
/**
 * Abstract filter class for InStream objects.
 * This class does no actual tranformation or analysis of data.  It
 * just provides a read method that delegates the actual read to
 * another InStream object.
 */
public class FilterInStream extends InStream {
    private InStream inStream;
    /**
     * Constructor
     * @param inStream The InStream that this object should delegate
     *                 read operations to.
     */
    public FilterInStream(InStream inStream) throws IOException {
        this.inStream = inStream;
    } // Constructor(InStream)
    /**
     * Read bytes from a stream of bytes and fill an array with those
     * bytes.
     * @param array The array of bytes to fill.
     * @exception IOException  if a I/O error occurs.
     */
    public int read(byte[] array) throws IOException {
        return inStream.read(array);
    } // read(byte[])
} // class FilterInStream
```

Finally, we will look at some classes that participate in the Filter pattern as a concrete source filter. The first of these performs the simple analysis of counting the number of bytes that is has read:

```
public class ByteCountInStream extends FilterInStream {
    private long byteCount = 0;
    /**
     * Constructor
     * @param inStream The InStream that this object should delegate
     * read operations to.
     */
    public ByteCountInStream(InStream inStream) throws IOException {
        super(inStream);
    } // Constructor(InStream)
    /**
```

```
     * Read bytes from a stream of bytes and fill an array with those
     * bytes.
     * @param array The array of bytes to fill.
     * @exception IOException  if a I/O error occurs.
     */
    public int read(byte[] array) throws IOException {
        int count;
        count = super.read(array);
        if (count >0)
          byteCount += count;
        return count;
    } // read(byte[])
    /**
     * return the number of bytes that have been read by this object.
     */
    public long getByteCount() {
        return byteCount;
    } // getByteCount()
} // class ByteCountInStream
```

Lastly, here is a class that performs character code translations of a stream of bytes:

```
/**
 * Filter class to perform eight bit character translation.
 *<p>
 * This class treats the bytes in a bytes stream as eight bit character
 * codes and translates them to other character codes using a
 * translation table.
 */
public class TranslateInStream extends FilterInStream {
    private byte[] translationTable;
    /**
     * Constructor
     * @param inStream The InStream that this object should delegate
     * read operations to.
     * @param table An array of bytes that is used to
     *        determine translation values for character codes.
     *        The value to replace charactr code n with is found in at
     *        index n of the translation table.  If the array is longer
     *        than 256 elements, the additional elements are ignored.
     *        If the array is shorter than 256 elements, the not
     *        translation is done on character codes greater than or
     *        equal to the length of the array.
     */
    public TranslateInStream(InStream inStream,
                             byte[] table) throws IOException {
        super(inStream);
```

```
        // Create translation table by copying translation data.
        translationTable = new byte[256];
        System.arraycopy(table, 0, translationTable, 0,
                         Math.min(256, table.length));
        for (int i = table.length; i < 256; i++) {
            translationTable[i] = (byte)i;
        } // for
    } // Constructor(InStream)
    /**
     * Read bytes from a stream of bytes and fill an array with those
     * bytes.
     * @param array The array of bytes to fill.
     * @exception IOException  if a I/O error occurs.
     */
    public int read(byte[] array) throws IOException {
        int count;
        count = super.read(array);
        for (int i = 0; i < count; i++) {
            array[i] = translationTable[array[i]];
        } // for
        return count;
    } // read(byte[])
} // class ByteCountInStream
```

Related Patterns

Composite

The Composite pattern can be an alternative to the Filter pattern when the objects involved do not have a consistent interface and they can be composed statically.

Layered Architecture

The Layered Architecture pattern is similar to the Filter pattern. The most important difference is that the objects involved in the layered Architecture pattern correspond to different levels of abstraction.

Pipes

The Pipes pattern is sometimes an alternative to the Filter pattern.

Wrapper/Decorator

The Filter pattern is usually implemented as a special case of the Wrapper pattern. FilterFilterFilterFilterFilterFilter

Recursive Composition

The Recursive Composition pattern is also known as the Composite pattern.

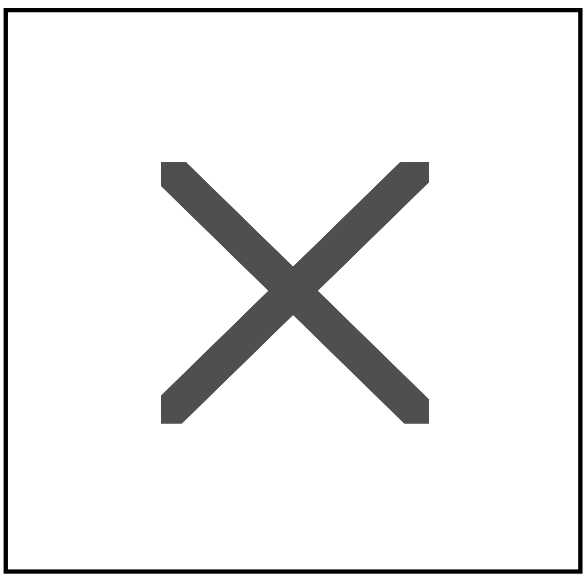
Synopsis

The Recursive Composition pattern allows you to build complex objects by composing similar objects in a tree-like manner. The Recursive Composition pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring that all of the objects in the tree have a common superclass.

The following description of the Recursive Composition pattern describes it in terms of building a composite object from other objects. The reason it appears in this partitioning patterns chapter is that during the design process the Recursive Composition pattern is often used to recursively decompose a complex object.

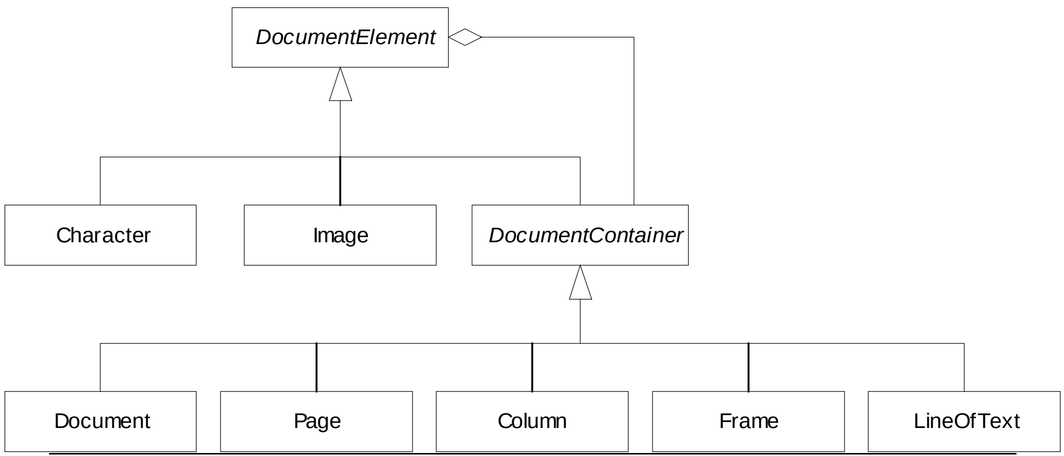
Context

Suppose that you are writing a document formatting program. It formats characters into lines of text organized into columns that are organized into pages. However, a document may contain other elements. Columns and pages can contain frames that can contain columns. Columns, frames and lines of text can contain images. Here is a class diagram that shows those relationships:



Document Container Relationships

As you can see, there is a fair amount of complexity here. Page and Frame objects must know how to handle and combine two kinds of elements. Column objects must know how to handle and combine three kinds of elements. The Recursive Composition pattern removes that complexity by allowing these objects to only know host to handle one kind of element. It accomplishes that by insisting that document element classes all have a common subclass. Here is how you can simplify the document element class relationships by using the Recursive Composition pattern:



Document Container Recursive Composition

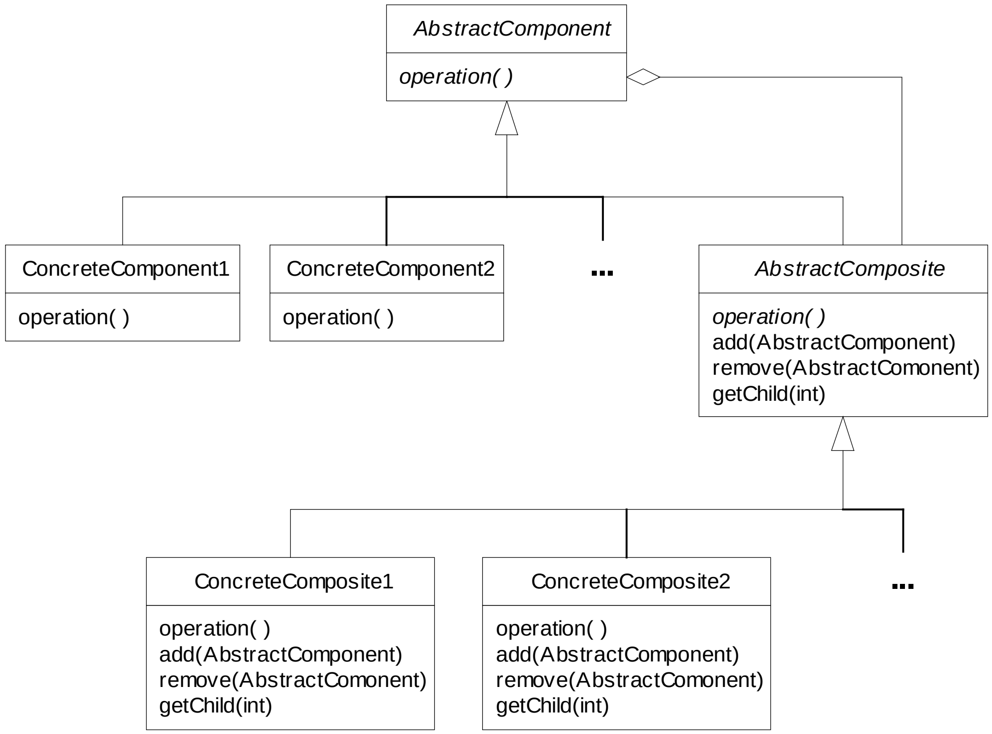
By applying the Recursive Composition pattern, you have introduced a common superclass for all document elements and a common superclass for all the document container classes. Doing that reduced the number of aggregation relationships to one. Management of the aggregation is now the responsibility of the `DocumentContainer` class. The concrete container classes (`Document`, `Page`, `Column`, …) only need to understand how to combine one kind of element.

Forces

- You have a complex object that you want to decompose into a part-whole hierarchy of objects.

- You want to minimize the complexity of the part-whole hierarchy by minimizing the number of different kinds of child objects that objects in the tree need to be aware of.

Solution



You can minimize the complexity of a composite object that is organized into a part-whole hierarchies by providing an abstract superclass for all of the objects in the hierarchy and an abstract superclass for all of the composites in the hierarchy. The class relationships for such an organization look like this:

Recursive Composition Class Relationships

Here are descriptions of the classes that participate in the Recursive Composition pattern:

AbstractComponent

`AbstractComponent` is an abstract class and the common superclass of all of the objects that are in the tree of objects that make up a composite object. Composite objects normally treat the objects that they contain as instances of `AbstractComponent`. Clients of composite objects normally treat them as instances of `AbstractComponent`.

ConcreteComponent1, ConcreteComponent2…

Instances of these classes are used as leaves in the tree organization.

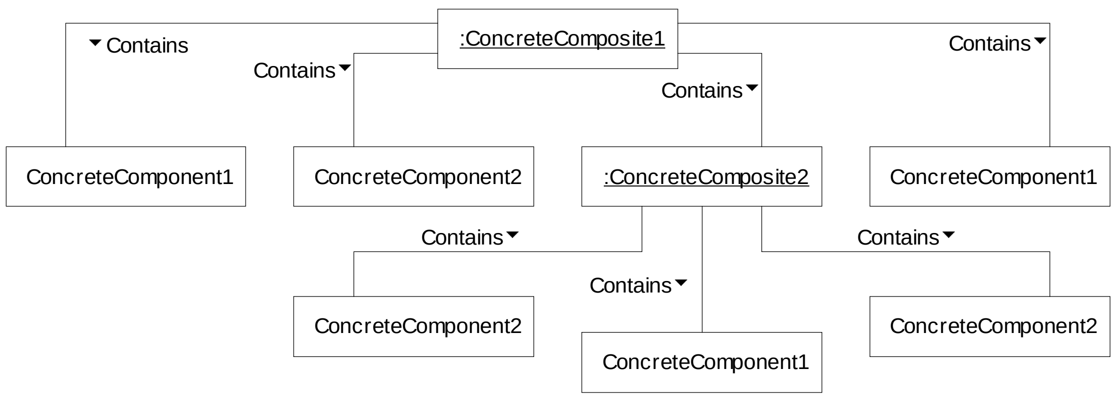
AbstractComposite

`AbstractComposite` is the abstract superclass of all composite objects that participate in the Recursive Composition pattern. `AbstractComposite` defines and provides default implementations for methods for managing a composite object's components. The `add` method adds a component to a composite object. The `remove` method removes a component from a composite object. The `getChild` method returns a reference to a component object of a composite object.

ConcreteComposite1, ConcreteComposite2, …

Instances of these are composite objects that use other instances of `AbstractComponent`.

Instance of these classes can be assembled in a tree-link manner like this:



Recursively Composed Object

Note that you don't need to have an abstract composite class if there is only one concrete composite class.

Consequences

- You can create a tree-structured composite object that simply treats the objects that comprise it as instances of `AbstractComponent`, whether they are simple objects or composite.

- Client objects of an `AbstractComponent` can simply treat it as an `AbstractComponent`, without having to be aware of any subclasses of `AbstractComponent`.

- If a client invokes a method of an `AbstractComponent` that is supposed perform an operation and the `AbstractComponent` object is a `AbstractComposite` object, then it will delegate that operation to the `AbstractComponent` objects that comprise it. Similarly, if a client object calls a method of an `AbstractComponent` object that is **not** a `AbstractComposite` and the method requires some contextual information, then the `AbstractComponent` delegates the request for contextual information to its parent.

- The Recursive Composition pattern allows any `AbstractComponent` to be a child of an `AbstractComposite`. If you need to enforce a more restrictive relationship then you will have to add type aware code to `AbstractComposite` or its subclasses. That reduces some of the value of the Recursive Composition pattern.

- Some components may implement operations that are unique to that component. For example, under the context heading of this pattern is a design for the recursive composition of a document. At the lowest level, is has a document consisting of character and image elements. It is very reasonable for the character elements of a document to have a `getFont` method. A document's image elements have no need for a `getFont` method. The main benefit that the Recursive Composition pattern provides is to allow the clients of a composite object and the objects that comprise it to be unaware of the specific class of the objects they deal with. To allow other classes to be able to call `getFont` without being aware of the specific class they are dealing with, all of the objects that can comprise a document can inherit the `getFont` method from `DocumentElelement`. In general, when applying the Recursive Composition pattern, the class in the role of the `AbstractComponent` class declares specialized methods if they are needed by any `ConcreteComponent` class.

  A principle of object oriented design is that specialized methods should only appear in classes that need them. Normally, a class should have methods that provide related functionality and form a cohesive set. That principle is the essence of the High Cohesion analysis pattern. Putting a specialized method in a general-purpose class rather than the specialized class that needs the method is contrary to the principle of high cohesion. It is contrary to that principle because it adds a method unrelated to the other methods of the general purpose class. That unrelated method is inherited by subclasses of the general purpose class that are unrelated to the method.

Because simplicity through ignorance of class is the basis of the Recursive Composition pattern, when applying the pattern it is ok to sacrifice high cohesion for simplicity.

Implementation

If classes that participate in the Recursive Composition pattern implement any operations by delegating to their parent object, then the best way to preserve speed and simplicity is by having each instance of `AbstractComponent` contain a reference to its parent. It is important to implement the parent pointer in a way that ensures consistency between parent and child. It must always be the case that an `AbstractComponent` identifies an `AbstractComposite` as its parent if and only if the `AbstractComposite` identifies it as one of its children. The best way to enforce that is to only modify parent and child references in the `AbstractComposite` class' add and remove methods.

Sharing components among multiple parents using the Flyweight pattern is a way to conserve memory. However, it is difficult for shared components to properly maintain parent references.

The `AbstractComposite` class may provide a default implementation of child management for composite objects. However, it is very common for concrete composite classes to override that implementation.

If a concrete composite class delegates an operation to the objects that comprise it, then caching the result of the operation may improve performance. If a concrete composite class caches the result of an operation, then it is important that the objects that comprise the composite notify the composite objects so that it can invalidate its cached values.

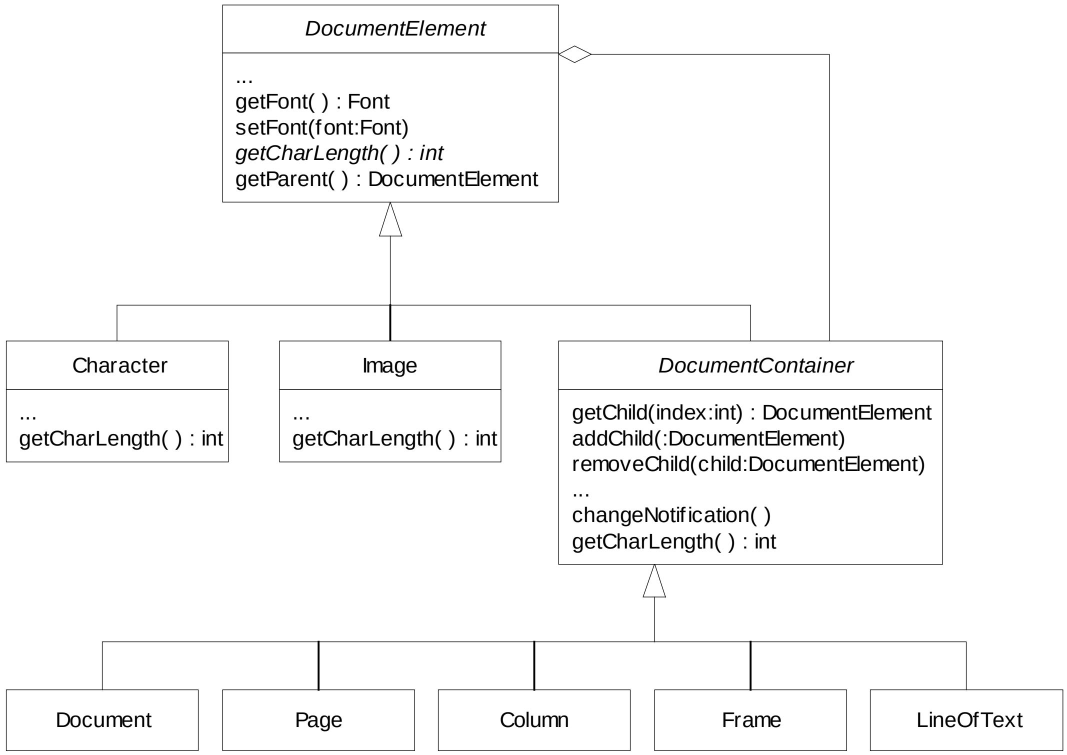
JAVA API Usage

The `java.awt` package contains a good example of the Recursive Composition pattern. Its `Component` class fills the `AbstractComponent` role.

Its `Container` class fills the `AbstractComposite` role. It has a number of classes in the `ConcreteComponent` role, including `Label`, `TextField` and `Button`. The classes in the `ConcreteComposite` role include `Panel`, `Frame` and `Dialog`.

Example

The example of the applying the Recursive Composition pattern is a more detailed version of the document related classes that appeared under the "context" heading. Here is a more detailed class diagram:



Detailed Document Recursive Composition

The above diagram shows some of methods. As you look through the following code, you will see that the `setFont` method is an example of a method that consults an object's parent object. The `getCharLength` method gathers information from an object's children and caches it for later use. The `changeNotification` method is used to invalidate cached information.

Here is code for the `DocumentElement` class:

```
abstract class DocumentElement {
    // 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 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)

    /**
     * Return the number of characters that this object contains.
     */
    public abstract int getCharLength() ;
} // class DocumentElement
```

Here is the code for the `DocumentContainer` class.

```
abstract class DocumentContainer extends DocumentElement {
    // Collection of this object’s children
    private Vector children = new Vector();

    // The cached value from the previous call to getCharLength or -1
    // to indicate that charLength does not contain a cached value.
    private int cachedCharLength = -1;

    /**
     * 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);
            child.parent = this;
            changeNotification();
        } // synchronized
    } // addChild(DocumentElement)

    /**
     * Make the given DocumentElement NOT a child of this object.
     */
    public synchronized void removeChild(DocumentElement child) {
        synchronized (child) {
            if (this == child.parent)
              child.parent = null;
            children.removeElement(child);
            changeNotification();
        } // synchronized
    } // removeChild(DocumentElement)
...
    /**
     * A call to this method means that one of this object’s children
     * has changed in a way that invalidates whatever data this object
     * may be cahcing about its children.
     */
    public void changeNotification() {
        cachedCharLength = -1;
```

```
        if (parent != null)
          parent.changeNotification();
    } // changeNotification()

    /**
     * Return the number of characters that this object contains.
     */
    public int getCharLength() {
        int len = 0;
        for (int i = 0; i < children.size(); i++) {
            DocumentElement child;
            child = (DocumentElement)children.elementAt(i);
            len += child.getCharLength();
        } // for
        cachedCharLength = len;
        return len;
    } // getCharLength()
} // class DocumentContainer
```

The Character class implements `getCharLength` in the obvious way:

```
class Character extends DocumentElement {
...
    /**
     * Return the number of characters that this object contains.
     */
    public int getCharLength() {
        return 1;
    } // getCharLength()
} // class Character
```

The Image class is an example of a class that implements a method so that the other classes that comprise a document do not need to be aware of the `Image` class as requiring any special treatment.

```
class Image extends DocumentElement {
...
    /**
     * Return the number of characters that this object contains.
     * Though images don't really contain any characters, for the sake
     * of consistenecy, we will treat an image as if it is a character.
     */
    public int getCharLength() {
        return 1;
    } // getCharLength()
} // class Image
```

The other classes in the class diagram that are subclasses of `DocumentContainer` do not have any features that are interesting with respect to the Recursive Composition pattern. In the interest of brevity, just one of them is shown below:

```
class Page extends DocumentContainer {
...
} // class Page
```

Related Patterns

Chain of Responsibility

The Chain of Responsibility pattern can be combined with the Recursive Composition pattern by adding child to parent links so that children can get information from an ancestor without having to know which ancestor the information came from.

High Cohesion

The High Cohesion analysis pattern discourages putting specialized methods in general purpose classes, which is something that the Recursive Composition pattern encourages.

Visitor

The Visitor pattern can be used to encapsulate operations in a single class that would otherwise be spread across multiple classes.