Creational Patterns

Creational patterns provide guidance as to how objects should be created when their creation requires decisions to be made. These decisions will typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to. The value of creational patterns is to tell us how to structure and encapsulate these decisions.

Often, there is more that one creational pattern that can be applied to a situation. Sometimes multiple patterns can be combined advantageously. In other cases you must choose between competing patterns. For these reasons, it is important to be acquainted will all five of the patterns described in this chapter.

If you only have time to learn one pattern in this chapter, the most commonly used one is Factory Method.

Factory Method

Synopsis

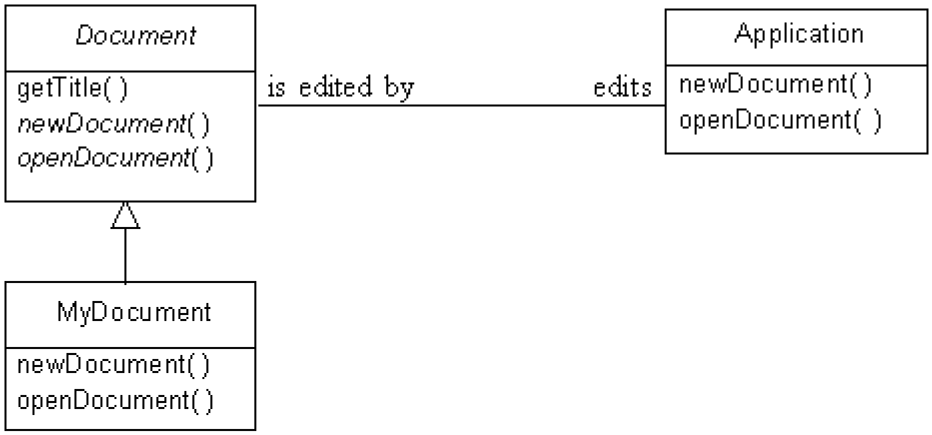
When you are writing a class to be reusable with arbitrary data types, it can be useful to organize it so that when it wants to instantiate a class, it delegates the choice of which class to instantiate to another object. The Factory Method pattern provides a way to do that.

Context

Consider the problem of writing a framework for desktop applications. Such applications are typically organized in a document or file centered manner. Their operation will typically begin with a command to create or edit a word processing document, spreadsheet, time line or whatever type of document or file the application is intended to work with. In the case of a word processor, the program may be required to work with multiple types of files.

A framework to support this type of application will clearly include an Application class that has methods to create and open documents in addition to other common operations. The reason to put those methods in an Application class is to give menu items and the like a consistent set of methods to call when the user issues a command.

Because the logic to implement most of these commands varies with the type of document, an Application object usually delegates most of these commands to some sort of document object. The logic in document objects for implementing these commands varies with the type of document. However there are some operations, such a getting the title string to display for a document, that will be common to all document objects. That suggests that there be an application independent abstract Document class and application specific subclasses for specific types of documents. Here is a class diagram to show these classes:



Application Framework

What neither the above diagram or the preceding discussion show is how an Application object can create instances of application specific document classes without itself being application specific.

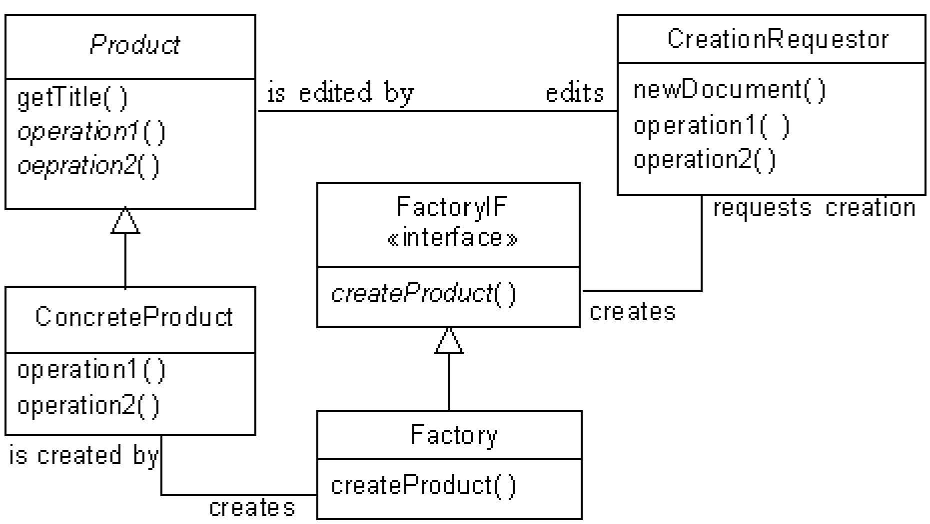
Forces

- It should be possible to organize a class that is able to create an instance of a class that is a subclass of a given class or implements a given interface. In order to be reusable, it must be able to do that without knowing what subclasses of the given class are available or what classes that implement the given interface are available.

- The set of classes a class may be expected to be able to instantiate may be dynamic as new classes become available.

Solution

The Factory Method pattern provides an application independent object with an application specific object to which it can delegate the creation of other application specific objects. Here is a class diagram that shows the interfaces and classes that typically make up the Factory Method pattern:



Factory Method Pattern

The above class diagram shows the roles in which classes and interfaces can participate in the Factory Method pattern:

Product

The product class is an abstract class that is the superclass of objects produced by the Factory Method pattern. The product class corresponds to the Document class mentioned in the context section.

An actual class in this role would not normally be called product, but have a name like Document or Image.

Concrete Product

This is any concrete class instantiated by the objects participating in the Factory Method pattern. If these classes share no common logic, then the product can be an interface instead of an abstract class.

An actual class in this role would not normally be called ConcreteProduct, but have a name like RTFDocument or JPEGImage.

Creation Requestor

The creation requestor is an application independent class that needs to create application specific classes. It does so indirectly through an instance of a factory class.

Factory Interface

This is an application independent interface that is implemented by the objects that create product objects on behalf of creation requestor objects. Interfaces of this sort declare a method that can be called by a creation requestor object to create concrete product objects. The method typically has a name like `createDocument` or `createImage`. The method takes whatever arguments are needed to deduce the class to instantiate.

Interfaces filling this role will typically have a name like `DocumentFactoryIF` or `ImageFactoryIF`.

Factory Class

This is an application specific class that implements the appropriate factory interface and has a method to create concrete product objects. Classes filling this role will typically have a name like `DocumentFactory` or `ImageFactory`.

Consequences

The primary consequences of using the Factory Method pattern are:

- The creation requester class is independent of the class of the concrete product objects actually created.
- The set of product classes that can be instantiated can be changed dynamically.

Implementation

In situations where all the concrete product classes are know in advance, the indirection of a product interface may not be necessary. In these situations, the only benefit realized from the Factory Method pattern is that the creation requestor class is kept independent of the actual concrete product classes instantiated. The way this works is that the creation requestor class refers directly to a factory object. That factory object has a `createProduct` method implemented with the necessary logic to instantiate the correct concrete product class.

If all classes that implement a product interface create only one kind of concrete product, then the `createProduct` method defined by the interface may not need n parameters. However, if factory objects are required to create multiple kinds of product objects then their createProduct method will need to take the necessary parameters to allow the method to deduce which product class to instantiate. Parametric createProduct methods often look something like this:

```
Image createImage (String ext) {
    if (ext.equals("gif")
      return new GIFImage();
    if (ext.equals("jpeg"))
      return new JPEGImage();
    É
} // createImage(String)
```

The above sequence of `if` statements works well for createProduct methods that have a fixed set of product classes to instantiate. To write a createProduct method that handles a dynamic or large number of product classes, you can use the Hashed Adapter Objects pattern. Alternatively, if you are writing in Java, you can use the various objects that indicate which class to instantiate as keys in a hash table with `java.lang.reflect.Constructor` objects for values. Using that technique, you look up an argument value in the hash table and the use the Constructor object that is its value in the hash table to instantiate the desired object.

JAVA API Usage

The Java API uses the Factory Method pattern in a few different places to allow the integration of the applet environment with its host program. For example, each URL object has associated with it a `URLConnection` object. `URLConnection` objects can be used to read the raw bytes of a URL. `URLConnection` objects also have a method called `getContent` that returns the content of the URL packaged in an appropriate sort of object. For example, if the URL contains a `gif` file then the `URLConnection` object's `getContent` method returns an Image object.

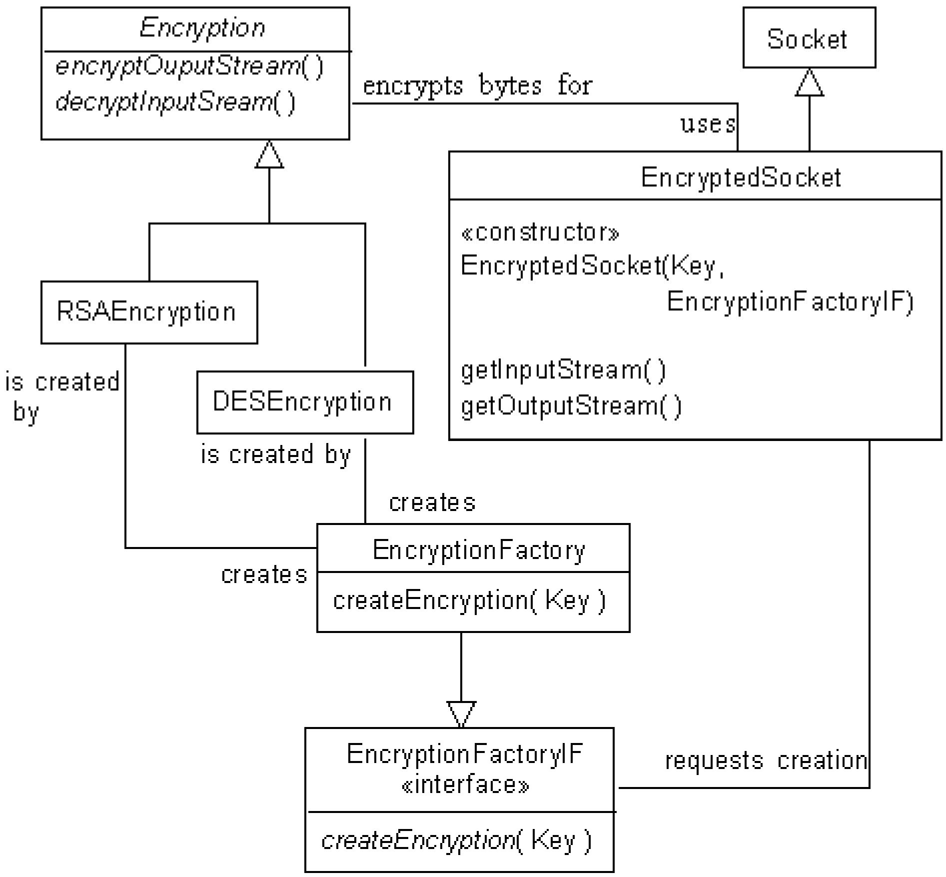
The way it works is that `URLConnection` objects play the role of creation requester in the Factory Method pattern. They delegate the work of the `getContent` method to a `ContentHandler` object. `ContentHandler` is an abstract class that serves as a product class that knows about handling a specific type of content. The way that a `URLConnection` object gets a `ContentHandler` object is through a `ContentHandlerFactory` object. The `ContentHandlerFactory` class is an abstract class that participates in the Factory Method pattern as a factory interface. The `URLConnection` class also has a method called `setContentHandlerFactory`. Programs that host applets call that method to provide a factory object used for all `URLConnection` objects.

Example

For our example, suppose we are developing an extension to the Socket class to encrypt the stream of bytes written to a socket and decrypt the bytes read from the socket. We will call this class `EncryptedSocket`.

We will want the `EncryptedSocket` class to support multiple encryption algorithms. Because of U.S.A. legal restrictions on the import and export of encryption software, we will want to keep the `EncryptedSocket` class independent of the encryption classes being used.

The requirement that `EncryptedSocket` objects be able to work with multiple encryption algorithms without knowing in advance what classes will encapsulate those algorithms suggests the use of the Factory Method pattern. Here is a class diagram to show this:



Factory Method Example

Here is a description of the classes and interface used in this design:

`EncryptedSocket`
: This subclass of `java.net.Socket` fills the role of creation requestor.

`EncryptionFactoryIF`
: This interface fills the role of factory interface.

`EncryptionFactory`
: This class fills the role of factory class.

`Encryption`
: This class fills the role of product.

`RSAEncryption`
: This class is in the role of concrete product.

`DESEncryption`
: This class is in the role of concrete product.

Let's look at the code that implements these classes. Here is the code for the `EncryptedSocket` class:

```
/**
 * This class extends socket so that the stream of bytes that goes over
 * the net is encrypted.
 */
public class EncryptedSocket extends Socket {
    private static Encryption crypt;
    private Key key;
    /**
     * Constructor
```

```
     * @param key The key to use for encryption and decryption.  This
     *            object will determine the encryption technique to use
     *            by calling the key object's getAlgorithm() method.
     * @param factory The Factory object to use to create Encryption
     *                objects.
     * @exception NoSuchAlgorithmException if the key specifies an
     *            encryption technique that is not available.
     */
    public EncryptedSocket(Key key, EncryptionFactoryIF factory)
                        throws NoSuchAlgorithmException {
        this.key = key;
        crypt = factory.createEncryption(key);
    } // Constructor(Key, EncryptionFactoryIF)
    /**
     * Returns an input stream for this socket that decrypts the
     * inbound stream of bytes.
     * @return     an input stream for reading decrypted bytes from
     *             this socket.
     * @exception  IOException  if an I/O error occurs when creating
     *               the input stream.
     */
    public InputStream getInputStream() throws IOException {
            return crypt.decryptInputStream(super.getInputStream());
    } // getInputStream()
    /**
     * Returns an output stream for this socket that encrypts the
     * outbound stream of bytes.
     * @return     an output stream for reading decrypted bytes from
     *             this socket.
     * @exception  IOException  if an I/O error occurs when creating
     *               the output stream.
     */
    public OutputStream getOutputStream() throws IOException {
        return crypt.encryptOutputStream(super.getOutputStream());
    } // getOutputStream()
} // class EncryptedSocket
```

An `EncryptedSocket` object works by first getting an Encryption object from the `EncryptionFactoryIF` object that is passed to its constructor. It then accomplishes the encryption and decryption using the Filter pattern. It extends the `getInputStream` and `getOutputStream` methods so that the `inputStream` objects and `outputStream` objects that they would otherwise return are filtered through objects created by the Encryption object.

Here is code for the `EncryptionFactoryIF` interface:

```
/**
 * This interface must be implemented by all factory classes used to
 * create instances of subclasses of Encryption.
 */
public interface EncryptionFactoryIF {
    /**
     * This method returns an instance of the appropriate subclass of
     * Encryption as determined from information provided by the given
     * Key object.
     * @param key The key that will be used to perform the encryption.
     */
    public Encryption createEncryption(Key key)
                    throws NoSuchAlgorithmException;
} // interface EncryptionFactoryIF
```

Here is the code for the `EncryptionFactory` class:

```
/**
 * This interface must be implemented by all factory classes used to
 * create instances of subclasses of Encryption.
 */
public class EncryptionFactory {
    /**
     * This method returns an instnace of the appropriate subclass of
     * Encryption as determined from information provided by the given
     * Key object.
     * @param key The key that will be used to perform the encryption.
     */
    public Encryption createEncryption(Key key)
                    throws NoSuchAlgorithmException{
        String algorithm = key.getAlgorithm();
        if (ÒDES".equals(algorithm))
          return new DESEncryption(key);
        if ("RSA".equals(algorithm))
          return new RSAEncryption(key);
        throw new NoSuchAlgorithmException(algorithm);
    } // createEncryption(Key)
} // class EncryptionFactory
```

Finally, here is the code for the Encryption class:

```
/**
 * Abstract class to encrypt/decrypt streams of bytes.
 */
abstract public class Encryption {
    private Key key;
    /**
     * Constructor
     * @param key The key to use to perform the encryption.
     */
    public Encryption(Key key) {
        this.key = key;
    } // Constructor(Key)
    /**
     * Return the key this object used for encryption and decryption.
     */
    protected Key getKey() {
        return key;
    } // getKey()
    /**
     * This method returns an OutputStream that encrypts the bytes
     * written to it and writes the encrypted bytes to the given
     * OutputStream.
     * @param out The OutputStream that the OutputStream returned by
     *            this method will write encrypted bytes to.
     */
    abstract OutputStream encryptOutputStream(OutputStream out);
    /**
     * This method returns an InputStream that decrypts the stream of
     * bytes that it reads from the given InputStream.
     * @param in The InputStream that the InputStream returned by this
     *           method will read bytes from.
     */
    abstract InputStream decryptInputStream(InputStream in);
} // class Encrypt
```

Related Patterns

Abstract Factory

The Factory Method pattern is useful for constructing individual objects for a specific purpose without the construction requestor knowing the specific classes being instantiated. If you need to create a matched set of such objects, then the Abstract Factory pattern is a more appropriate pattern to use.

Template Method

The full Factory Method pattern is often used with the Template Method pattern.

Prototype

The Prototype pattern provides an alternate way for an object to work with other objects without knowing the details of their construction.Factory MethodFactory Method

Abstract Factory

Abstract Factory is also known as kit or toolkit.

Synopsis

Given a set of related abstract classes, the Abstract Factory pattern gives us a way to create instances of those abstract classes from a matched set of concrete subclasses. The Abstract Factory pattern can be very useful for allowing a program to work with a variety of complex external entities such as different windowing systems that have similar functionality.

Context

Suppose that you have the task of building a user interface framework that works on top of multiple windowing systems, like MS-Windows, Motif or MacOS. You can make it work with a native look and feel by creating an abstract class for each type of widget (text field, push button, list box,…) and then writing a concrete subclass of each of those classes for each supported platform. To make that way of doing things work robustly, you will need a way of ensuring that the widget objects that are created are all for the desired platform. That is where the abstract factory comes into play.

In the context of this situation, an abstract factory class is a class that defines methods to create an instance of each one of the abstract classes that correspond to a user interface widget. Concrete factories are classes that are a concrete subclass of an abstract factory that implements its methods to create instances of concrete widget classes for the same platform.

In a more general context, an abstract factory class and its concrete subclasses organize sets of concrete classes that work different but related products.

Forces

- A system should function is a way that is independent of the specific product that it interfaces with.

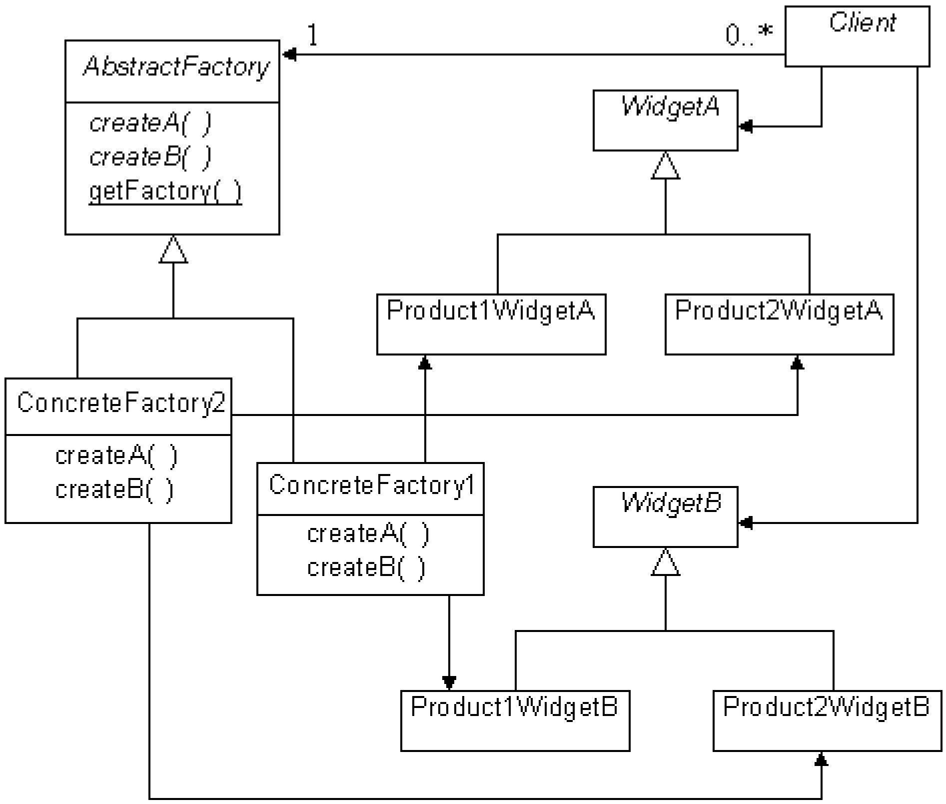
- It should be possible for a system to be configured to work with one or multiple members of a family of products.

- Instances of classes intended to interface with a product should be used together and only with that product. That constraint must be enforced.

- The rest of a system should be able to work with a product without being aware of the specific classes used to interface with the product.

- It should be possible to extend a system to work with additional products by adding additional sets of classes and changing at most only a few lines of code.

Solution



Abstract Factory

The Abstract Factory pattern involves the following kinds of classes:

Client

Client classes are classes that use various widget classes to request or receive services from the product that is being interfaced with. Client classes only know about the abstract widget classes. They should have no knowledge of any concrete widget classes.

Abstract Factory

Abstract factory classes define abstract methods for creating instances of concrete widget classes.

Abstract factory classes provide client classes with a method that returns an instance of a concrete factory appropriate for interfacing with a particular product. If all clients will be interfacing with the same product then the method may not need any arguments. Otherwise, it will usually be necessary for the method to take an argument that tells it what products the client wants to interface with.

At most one instance of an abstract factory is needed, so instances of abstract factory classes can be created using the Singleton pattern

Concrete Factory

Concrete factory classes implement the methods defined by their abstract factory superclasses to create instances of concrete widget classes. The client classes that call these methods should not have any direct knowledge of these concrete factory classes, but instead access singleton instances of these classes by calling a method of their abstract factory superclass.

Abstract Widget

Abstract widget classes correspond to a feature of a product that their concrete subclasses will interface with.

Concrete Widget

Concrete widget classes correspond to a feature of a product that they interface with.

Consequences

- The concrete widget classes that are used to interface with products are independent of the classes that use them, because the abstract factory class encapsulates the process of creating widget objects.

- Adding (as opposed to writing) classes to interface with additional products is simple. The class of a concrete factory object needs to appear where it is instantiated. The only other place that it may need to appear is the code or construction of the data structure in the abstract factory class that is responsible for selecting what concrete factory class the abstract factory class should return to a client. That also makes it easier to change the concrete factory used to interface with a particular product.

- By forcing client classes to go through concrete factory objects to create concrete widget objects, the abstract factory pattern ensures that client objects use a consistent set of objects to interface with the features of a product.

- The main drawback to the Abstract Factory pattern is that it can be a lot of work to write a set of classes to interface with a new product. It can also take a lot of work to extend the set of features that the existing set of classes is able to exercise in the products that they interface with.

  Adding support for a new product involves writing a complete set of concrete widget classes to support that product. A concrete widget class must be written for each abstract widget class. If there is a large number of abstract widget classes then it will be a lot of work to support an additional product.

  Adding access to an additional feature of the products interfaced to can also take a lot of work if there are many supported products. It involves writing a new abstract widget class corresponding to the new feature and a new concrete widget class corresponding to each product.

- Client objects may have a need to organize widget classes into a hierarchy that serves the needs of client objects. The basic Abstract Factory pattern does not lend itself to that because it requires concrete widget classes to be organized into a class hierarchy that is independent of client objects. That difficulty can be overcome by mixing the Bridge pattern with the Abstract Factory pattern: Create a hierarchy of product-independent widget classes that suites the needs of the client classes. Have each product-independent widget class delegate product specific logic to a product specific instance of an abstract widget class.

  Java's `java.awt` package contains a number of classes that are implemented using this variation. Classes like `Button` and `TextField` contain logic that is independent of the windowing system being used. These classes delegate windowing system operations to concrete widget classes that implement interfaces defined in the `java.awt.peer` package.

Implementation

If you are writing in Java and you implement the abstract factory class as an abstract class, as recommended, then its `getFactory` method will have to be static. The reason for is that you will want to call the `getFactory` method without having to create an instance of the abstract factory class.

A deeper implementation issue for the Abstract Factory pattern is the mechanism the abstract factory class' `getFactory` method uses to select the class of the concrete factory it supplies to client objects. The simplest situation is an abstract factory object used to interface with only one product during its lifetime. In that case, the code that creates that abstract factory object can provide it with the concrete factory objects that will always provide to clients.

If the abstract factory object will use information provided by the requesting client to select among multiple concrete factory objects, you can hard code the selection logic and choice of concrete factory objects in the abstract factory class. That strategy has the advantage of simplicity. It has the drawback of requiring a source code modification to add a new concrete factory class.

A different strategy is to put references to the concrete factory objects, along with the information used to select the concrete factory objects, into a data structure that allows the abstract factory object to select a concrete factory object by looking up the selection information in the data structure. The advantage of using the data structure is that it is possible to devise schemes that allow an abstract factory to work with new concrete factory classes without any source code modification.

Another implementation issue arises with the variation of the Abstract Factory pattern mentioned under consequences that uses a hierarchy of product independent classes that delegate product specific logic to product specific objects. In that variation, because all the widget independent logic is in the widget independent objects, the abstract widget classes contain no product specific logic. To help ensure that all of the product independent logic remains in the product independent widget classes when the product is in its maintenance phase, it is better to substitute widget interfaces for abstract widget classes.

JAVA API Usage

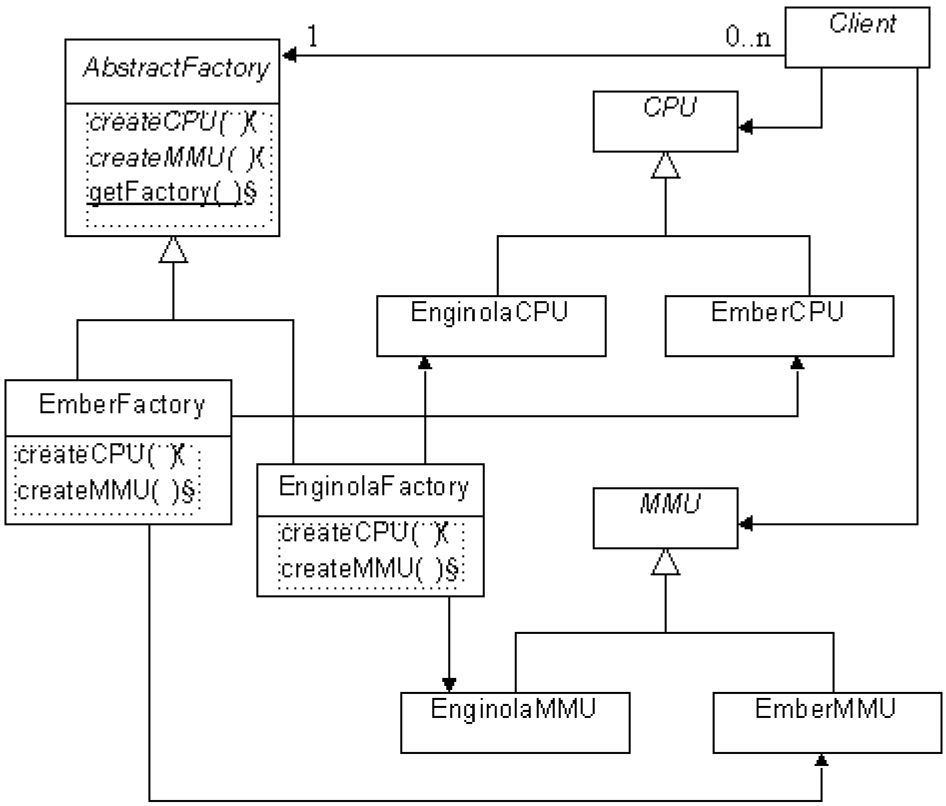
The Abstract Factory pattern is used in the Java API to implement the `java.awt.Toolkit` class. The `java.awt.Toolkit` class is an abstract factory class that is used for creating objects that interface with the native windowing system. The concrete factory class that it uses is determined by initialization code and the singleton concrete factory object is returned by its `getDefaultToolkit` method.

Example

Let us suppose that we are writing a program that performs remote diagnostics on computers for a computer manufacturer called Stellar computers. Over time, Stellar has produced computer models having substantially different architectures. Their oldest computers used CPU chips from Enginola that had a traditional complex instruction set. Since then, they have released three generations of computers based on their own RISC architectures called ember, super-ember and ultra-ember. The core components used in these models perform similar functions, but involve different sets of components.

In order for the program we are writing to be able to know that tests to run and how to interpret the results, it will need to be able to instantiate objects that correspond to each one of the core components in the computer being diagnosed. The class of each object will correspond to the type of component to be tested. That means that we will have a set of classes for each computer architecture. There will be a class in each set corresponding to the same type of computer component. Because this situation fits the Abstract Factory so well, we will use that pattern to organize the creation of objects that correspond to core computer components.

Here is a class diagram that shows classes for only two types of components in only two architectures:



Abstract Factory Example

Here is some Java code that implements some of these classes. The abstract widget classes have the obvious structure:

```
/**
 * This is an abstract class for objects that perform remote tests
 * on CPUs.
 */
public abstract class CPU extends ComponentTester {
    //...
} // class CPU
```

The concrete widget classes are simply concrete subclasses of the abstract widget classes:

```
/**
 * This is a class for objects that perform remote tests on Ember
 * architecture CPUs.
 */
class EmberCPU extends ComponentTester {
    //...
} // class EmberCPU
```

Here is code for a concrete factory class to create instance of classes to test ember architecture computers:

```
/**
 * This is a concrete factory class for creating objects used to
 * perform remote tests on core components of ember architecture
 * computers.
 */
class EmberFactory extends AbstractFactory {
    /**
     * Method to create objects for remote testing ember CPUs.
```

```
     */
    public CPU createCPU() {
        return new EmberCPU();
    } // createCPU()
    /**
     * Method to create objects for remote testing ember MMUs.
     */
    public MMU createMMU() {
        return new EmberMMU();
    } // createMMU()
    ...
} // class EmberFactory
```

Finally, here is the code for the abstract factory class:

```
/**
 * This is an abstract factory class for creating objects that are
 * used to perform remote tests on core components of computers.
 */
public abstract class AbstractFactory {
    private static final EmberFactory emberFactory
      = new EmberFactory();
    private static EnginolaFactory enginolaFactory
      = new EnginolaFactory();
...
    // Symbolic names to identify types of computers
    public final static int ENGINOLA = 900;
    public final static int EMBER    = 901;
...

    /**
     * This method returns a concrete factory object that is an
     * instance of the concrete factory class that is appropriate for
     * the given computer architecture.
     * @param architecture a value indicating the architecture that a
     *        concrete factory should be returned for.
     */
    static final AbstractFactory getFactory(int architecture) {
        switch (architecture) {
          case ENGINOLA:
              return enginolaFactory;

          case EMBER:
              return emberFactory;
          // ...
        } // switch
        String errMsg = Integer.toString(architecture);
        throw new IllegalArgumentException(errMsg);
    } // getFactory()

    /**
     * Method to create objects for remote testing CPUs.
     */
    public abstract CPU createCPU() ;

    /**
     * Method to create objects for remote testing MMUs.
     */
    public abstract MMU createMMU() ;

    //...
} // AbstractFactory
```

Client classes will typically create concrete widget objects using code that looks something like this:

```
/**
 * Sample client class to show how a client class can create concrete
 * widget objects using an abstract factory
 */
public class Client {
    public void doIt () {
        AbstractFactory af;
        af = AbstractFactory.getFactory(AbstractFactory.EMBER);
        CPU cpu = af.createCPU();
        //...
    } //doIt
} // class Client
```

Related Patterns

Factory Method

In the preceding example, the abstract factory class uses the Factory Method pattern to decide which concrete factory object to give to a client class.

Singleton

Concrete Factory classes are usually implemented as a Singleton classes.Abstract FactoryAbstract Factory

Builder

Synopsis

The Builder pattern allows a client object to construct a complex object by specifying only its type and content. The client is shielded from the details of the object's construction.

Context

When a company is building a commercial quality compiler, one of the typical goals is that the compiler be able to generate native code for multiple platforms. The part of a compiler responsible for generating the code produce by the compiler is called a code generator. The rest of the compiler should not need to contribute any more to the code generation process than specifying what the target platform is and the low-level operations that are to be performed by the generated program. Another goal is that a code generator be able to work with multiple compilers. Once the company has built a C++ compiler that runs on multiple platforms, it might want to build a COBOL compiler that uses the same code generators as the C++ compilers and so is able to generate code for all of the same platforms. Those goals require that the code generator have no knowledge of how the rest of the compiler represents a program. They also require that the rest of the compiler be independent of the code generated by the code generator.

You can satisfy these requirements by defining an abstract class that declares abstract methods corresponding to all of the low-level operations that code generators need to support. You can then write concrete subclasses of the abstract class that will be code generators for specific platforms. To put the pieces together, you will also need an object to take the knowledge that the compiler has produced from parsing source code and turn it into a series of calls to the code generator's methods.

The objects and relationships described in the preceding paragraph are the essence of the Builder pattern.

Forces

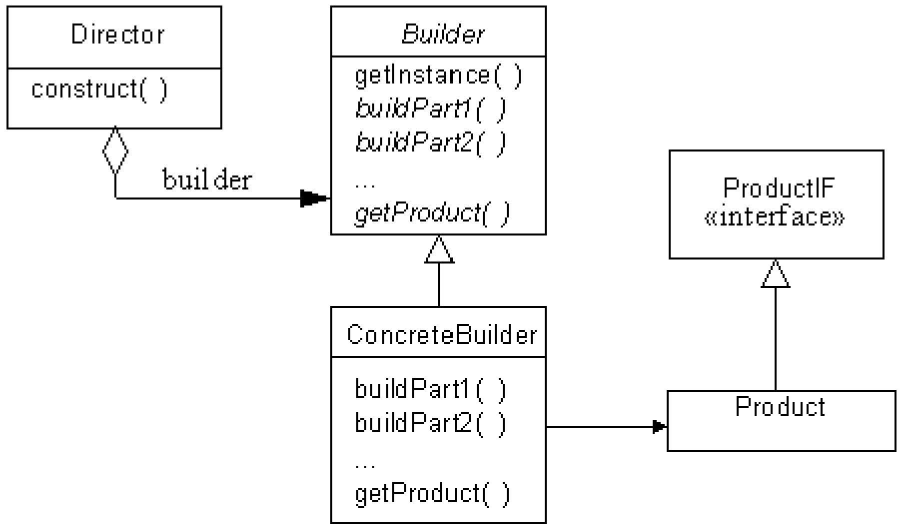
- Multiple data representations may need to be provided.

- The object providing the content for the object being built is independent of the data representation being built.

- The object that builds the data representation object is independent of the object that provides the content and can work with multiple content providing objects.

Solution

Here is a class diagram showing the participants in the Builder pattern



Builder

Here are the roles that these classes in interface play in the Builder pattern:

Product

The product class is not a specific class but any class that corresponds to a data representation. All product classes should implement the ProductIF interface.

ProductIF

The ProductIF interface defines no methods. It is a semantic marker that provides a common type for product objects. The reason for not defining any methods is that product classes will differ greatly in their functionality. All that they have in common is content rather than function.

Concrete Builder

A concrete builder class is a concrete subclass of the builder class that is used to build a specific kind of data representation.

Builder

The builder class is an abstract class. It defines a class method, typically called `getInstance`, which takes an argument that specifies a data representation. The `getInstance` method returns an instance of a concrete builder class that produces the specified data representation.

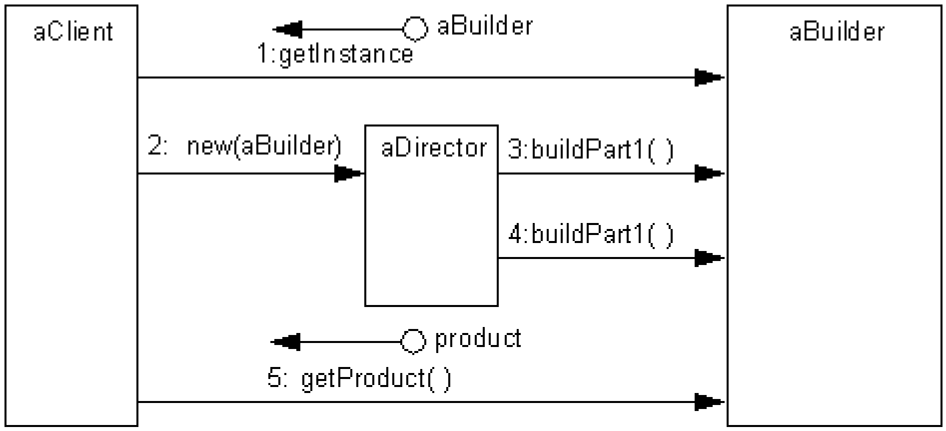
The builder class also defines methods, shown in the class diagram as buildPart1, buildPart2… that can be called to tell the object returned by the `getInstance` method what to content to put in the created object.

Finally, the builder class defines a method, typically called `getProduct`, that returns the product object created by the concrete builder object.

Director

A director object has the responsibility of calling the methods of a concrete builder object to provide it with the content of the product object hat it builds.

Here is a collaboration diagram to show how these classes work together:



Builder Collaboration

Consequences

- Content determination and the construction of a specific data representation are independent of each other. The data representation of the product can change without any impact on the objects that provide the content. Also, builder objects can work with different content providing object without requiring any changes.

- Builder provides finer control over construction than other patterns such as Factory Method by giving the director object step by step control over creation of the product object. Other patterns simply create entire object in one step.

Implementation

The key design and implementation issue for the Builder pattern is the set of methods defined by the builder class to provide content to concrete builder objects. These methods can be a major concern because there can be a large number of them. The methods should be general enough to allow all reasonable data representation to be constructed. On the other hand, an excessively general set of methods can be more difficult to implement and to use. The consideration of generality versus difficulty of implementation raises these issues in the implementation phase:

- Each of the content providing methods declared by the builder class can be declared abstractly or provided with a default do-nothing implementation. Abstract method declarations force concrete builder classes to provide an implementation for that method. Forcing concrete builder classes to provide implementations for methods is good in those cases where the method provides essential information about content. It prevents implementers of those classes from forgetting to implement those methods.

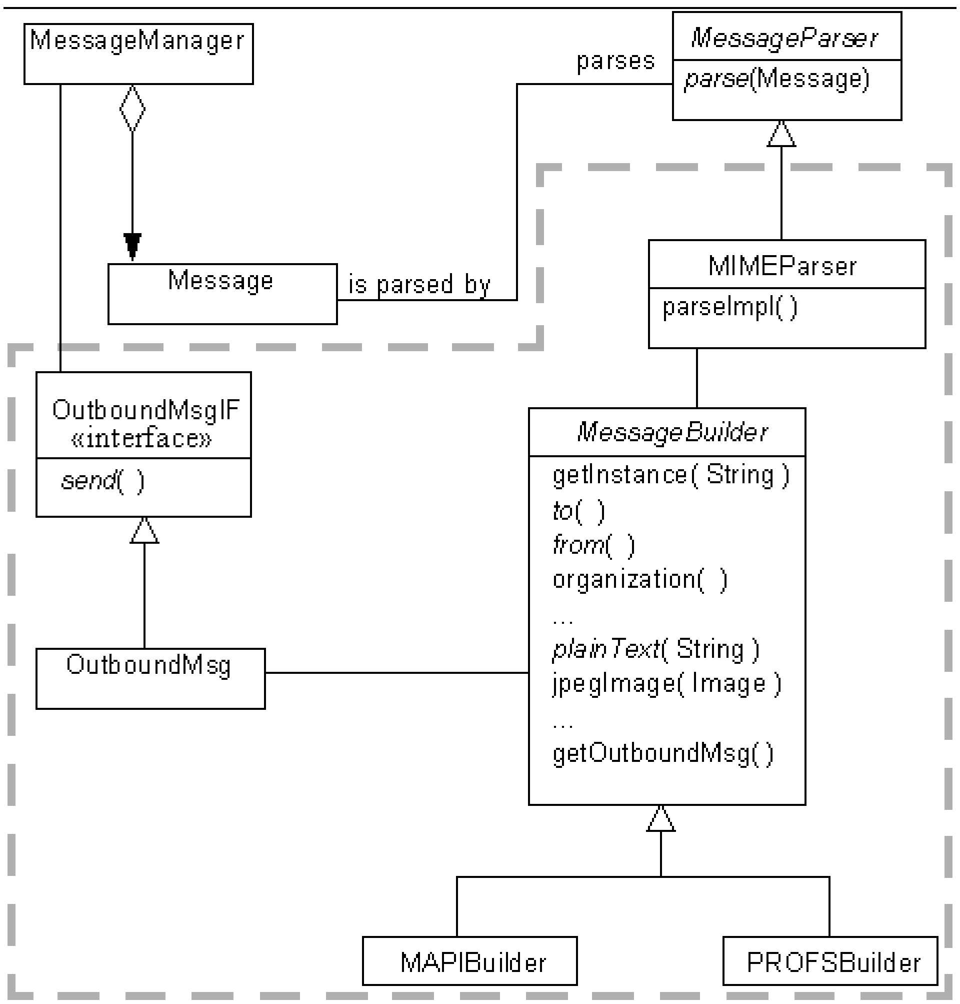
  However, methods that provide optional content or supplementary information about the structure of the content may be unnecessary or even inappropriate for some data representations. Providing a default do-nothing implementation for methods such as these saves effort in the implementation of concrete builder classes that do not need those methods.

- Organizing concrete builder classes so that call to content providing methods simply add data to the product object is often good enough. In some cases there will be no simple way to tell the builder where in the finished product a particular piece of the product will go. In those situations it may be simplest for the content providing method to return a partial product object that contains part of the data so that the director object can pass it back into another content providing method at a later appropriate time.

Example

Consider the problem of writing a program that receives e-mail messages that are in MIME format* and forwarding it as a message in a different format for a different kind of e-mail system. This situation is a good fit for the Builder pattern. It is very straightforward to organize this program into an object that parses MIME messages and as it recognizes each header field and message body part calls the corresponding method of the concrete builder object that it is working with. Also, that is a design that will allow the program to be extended to convert from message formats other than MIME.

Here is a class diagram showing that structure:



Builder Example

In the above diagram, the `MessageManager` class is responsible for receiving e-mail messages.

The `MessageParser` class is an abstract class that is the superclass for classes that knows how to parse e-mail messages and passes their content to a builder object.

* MIME is an acronym for Multipurpose Internet Mail Extensions. It is the standard that most e-mail messages on the internet conform to. You can find a description of MIME at http://mgrand.home.mindspring.com/mime.html.

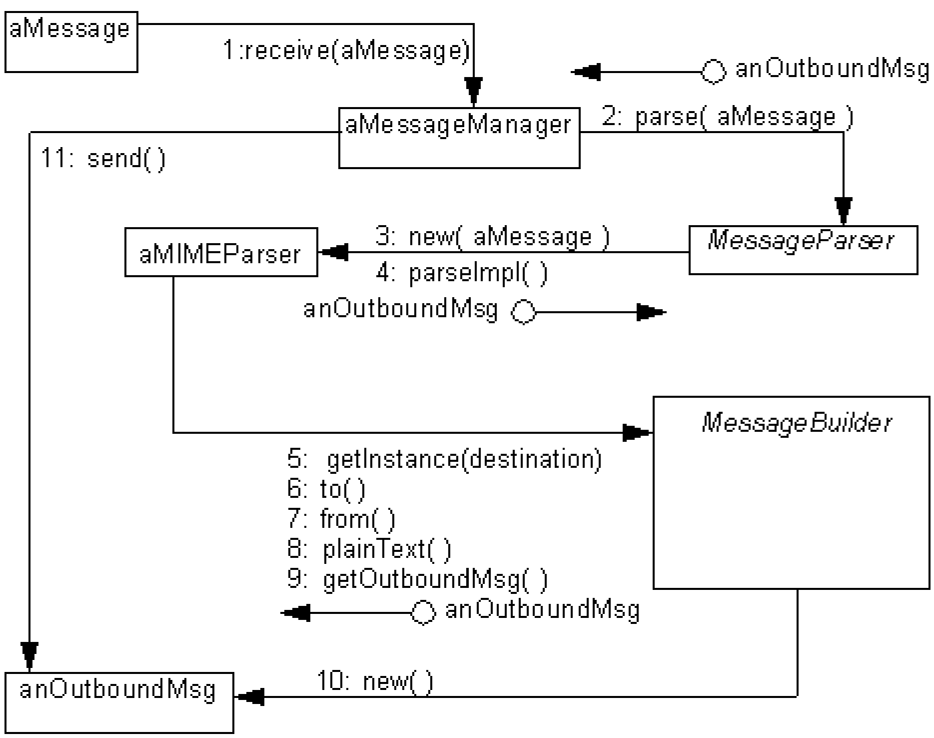
That brings us to the classes in the dashed line, which are the classes that implement the Builder pattern.

The `MIMEParser` class is a subclass of that `MessageParser` that is able to parse MIME e-mail messages and pass their contents on to a builder object. `MessageBuilder` is an abstract builder class. It defines methods that correspond to the various header fields and body types that MIME supports. It declares methods that correspond to required header fields and the most common body types to be abstract. It declares them abstract because all concrete subclasses of `MessageBuilder` should define those method. However, some of the optional header fields such as organization and fancier body types such as Image/Jpeg may not be supported in all message formats, so the `MessageBuilder` class provides do-nothing implementations of those methods.

The `MAPIBuilder` and `PROFSBuilder` classes are concrete builder classes for building MAPI and PROFS messages, respectively.

The builder classes create product objects that implement the `OutboundMsgIF` interface. That interface defines a method called send that is intended to send the e-mail message wherever it is supposed to go.

Here is a collaboration diagram that shows how these classes work together:



Builder Example Collaboration

Here is what's happening in the above diagram:

1. A `MessageManager` object receives a message.

2. The `MessageManager` object calls the `MessageParser` class' parse method.

3. That parse method always assumes that the e-mail message to be parsed is a MIME message, so it always creates a `MIMEParser` object. If the program is extended to convert from other message types that MIME, then the message type would most likely be passed into parse.

4. The `MessageParser` class' parse method calls the `MIMEParser` object's `parseImpl` method to do the actual message parsing.

5. The `MIMEParser` object calls the `MessageBuilder` class' `getInstance` method, passing it the destination e-mail address. By analyzing the address, the method selects a concrete subclass of the `MessageBuilder` class and creates an instance of it.

6. The `MIMEParser` object passes the destination e-mail address to the `MessageBuilder` object's to method.

7. The `MIMEParser` object passes the originating e-mail address to the `MessageBuilder` object's from method.

8. The `MIMEParser` object passes the e-mail message's simple content to the `MessageBuilder` object's `plainText` method.

9. The `MIMEParser` object calls the `MessageBuilder` object's `getOutboundMsg` method.

10. The `MessageBuilder` object's `getOutboundMsg` method completes the creation of the `OutboundMsg` object.

11. The `MessageManager` object calls the `OutboundMsg` object's send method. That sends the message off, and completes the processing of that message.

Let's look at some sample code for the classes in this example that collaborate in the Builder pattern. Instances of the `MIMEParser` class fill the role of director objects. Here is the source for the `MIMEParser` class:

```
class MIMEParser extends MessageParser {
    private Message msg;        // The message being parsed
    private MessageBuilder builder; // The builder object
    É
    MIMEParser(Message msg) {
        this.msg = msg;
    } // constructor(Message)
    /**
     * parse a MIME message, calling the builder methods that
     * correspond to the messageÕs header fields and body parts.
     */
    OutboundMsgIF parseImpl() {
        builder = MessageBuilder.getInstance(getDestination());
        MessagePart hdr = nextHeader();
        while (hdr != null) {
            if (hdr.getName().equals("to"))
              builder.to((String)hdr.getValue());
            else if (hdr.getName().equals(ÒfromÓ))
              builder.from((String)hdr.getValue());
            É
            hdr = nextHeader();
        } // while hdr
        MessagePart bdy = nextBodyPart();
        while (bdy != null) {
            if (bdy.getName().equals(Òtext/plainÓ))
              builder.plainText((String)bdy.getValue());
            É
            else if (bdy.getName().equals(Òimage/jpegÓ))
```

```
                builder.jpegImage((Image)bdy.getValue());
              É
              bdy = nextHeader();
          } // while bdy
          return builder.getOutboundMsg();
      } // parseImpl(Message)

      private MessagePart nextHeader() {
          MessagePart mp = null;
          É
          return mp;
      } // nextHeader()

      private MessagePart nextBodyPart() {
          MessagePart mp = null;
          É
          return mp;
      } // nextBodyPart()

      // return the destination e-mail address for the message
      private String getDestination() {
          String dest = null;
          É
          return dest;
      } // getDestination()

      private class MessagePart {
          private String name;
          private Object value;
          /**
           * Consructor
           */
          MessagePart(String name, Object value) {
              this.name = name;
              this.value = value;
          } // Consructor(String, String)

          String getName() { return name; }

          Object getValue() { return value; }
      } // class MessagePart
  } // class MIMEParser
```

The chains of if statements that occur in the parseImpl method of the above class would be rather long. MIME supports over 25 different kinds of header fields alone. A less awkward way to organize a chain of tests of object equality that result in a method call is to use the Hashed Adapter Objects pattern.

Here is code for the `MessageBuilder` class, that fills the role of abstract builder class:

```
  /**
   * This is an abstract builder class for building e-mail messages
   */
  abstract class MessageBuilder {
      /**
       * Return an object of the subclass appropriate for the e-mail
       * message format implied by the given destination address.
       * @param dest The e-mail address the message is to be sent to
       */
      static MessageBuilder getInstance(String dest) {
          MessageBuilder builder = null;
```

```
        É
        return builder;
    } // getInstance(String)
    /**
     * pass the value of the ÒtoÓ header field to this method.
     */
    abstract void to(String value);
    /**
     * pass the value of the ÒfromÓ header field to this method.
     */
    abstract void from(String value);
    É
    /**
     * pass the value of the ÒorganizationÓ header field to this
     * method.
     */
    void organization(String value) { }
    /**
     * pass the content of a plain text body part to this method.
     */
    abstract void plainText(String content);
    /**
     * pass the content of a jpeg image body part to this method.
     */
    void jpegImage(Image content) { }
    /**
     * complete and return the outbound e-mail message.
    */
    abstract OutboundMsgIF getOutboundMsg() ;
} // class MessageBuilder
```

Finally, here is the code for the `OutboundMsgIF` interface:

```
public interface OutboundMsgIF {
    public void send() ;
} // interface OutboundMsgIF
```

Related Patterns

Composite

The object built using the Builder pattern is typically a Composite.

Factory Method

The Builder pattern uses the Factory Method pattern to decide which concrete builder class to instantiate.

Semantic Marker

The ProductIF interface uses the Semantic Marker pattern.

Visitor

The visitor pattern allows the client object to be more closely couple to the construction to the new complex object. Instead of describing the content of the objects to be built through a series of method calls, the information is presented in bulk as a complex data structure.BuilderBuilder

Prototype

Synopsis

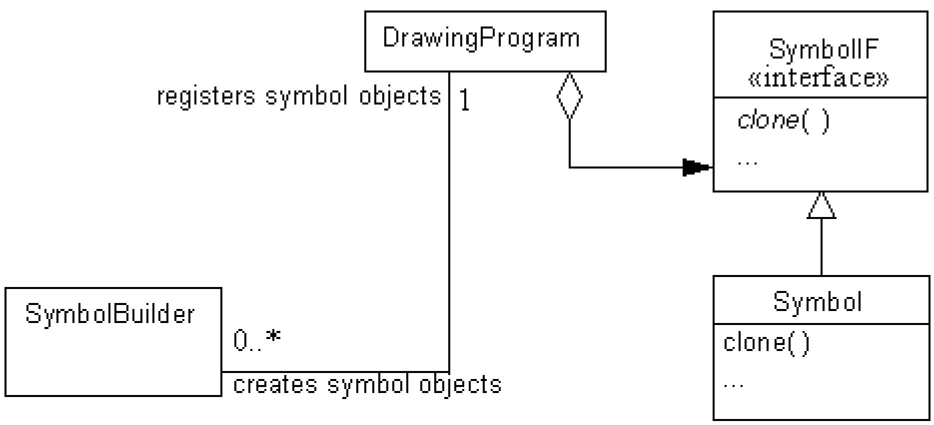
The Prototype pattern allows an object to create other objects without knowing their exact class or the details of how to create them. It works by giving prototypical instances of the objects to be created to the object that will initiate the creation of objects. The object initiating the creation then creates objects by asking the prototypical objects to clone themselves.

Context

Suppose that we are writing a program such as a CAD program that allows its users to draw diagrams using a palette of symbols. The program will have a core set of symbols that are built into it. However, the program is intended for use by people with different and specialized interests. The core set of symbols will not be adequate for people with a specialized interest. Those people will want additional symbols that are specific to their interests. Most users of this program will be in that category. Therefore, it must be possible to provide additional sets of symbols that users can add to the program to suite their needs.

That gives us the problem of how to provide these palettes of additional symbols. We can easily organize things so that all symbols, both core and additional, are all descended from a common ancestor class. That will give the rest of our diagram drawing program a consistent way of manipulating symbol objects. It does leave open the question of how the program will create these objects. Creating objects such as these is often more complicated than knowing how to instantiate a class. It may also involve setting values for data attributes of objects or combining objects to form a composite object.

The solution that the Prototype pattern suggests to provide the drawing program with previously created objects to use as prototypes to create similar objects. The most important requirement for objects to be used as prototypes is that they have a method, typically called clone, that returns a new object that is a copy of the original object. Here is a class diagram that shows how this would be organized:



Symbol Prototype

`Symbol` objects implement the `SymbolIF` interface that declares a method called `clone`. The `clone` method returns a copy of the object. The drawing program maintains a collection of prototypical `Symbol` objects that it uses by cloning. `SymbolBuilder` objects create `Symbol` objects and register them with the drawing program.

Forces

- A system must be able to work with objects without knowing their exact class, how they are created or what data they represent.

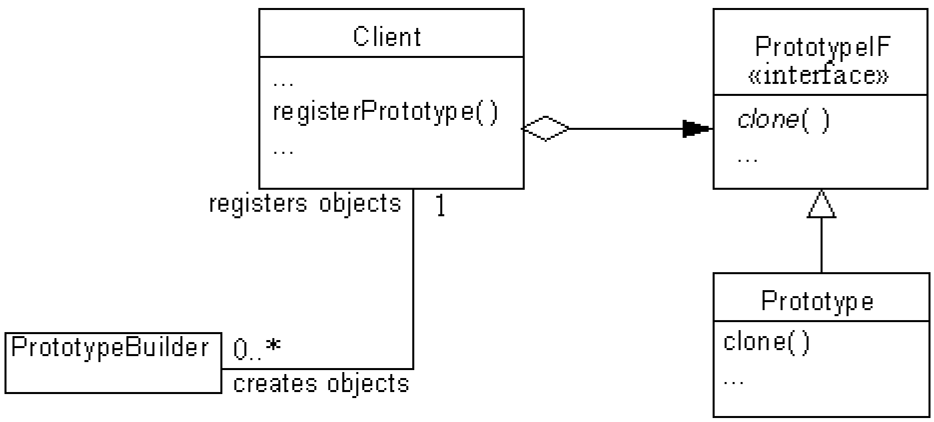
- Classes to be instantiated are not known by the system until run time, when they are acquired on the fly by a technique such as dynamic loading.

- Creating a large hierarchy of factory classes or abstract factories to parallel a hierarchy of classes to be instantiated is undesirable.

- The difference between different objects that a system is to work with may be instances of the same class that contain different state information or data content.

Solution

Here are the classes and interfaces that participate in the Prototype pattern:



Prototype Pattern

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

Client

The client class represents the rest of the program for the purposes of the Prototype pattern. The client class needs to create objects that it knows little about. The client class will have a method that can be called to add a prototypical object to a client object's collection. In the above diagram, that method is indicated with the name `registerPrototype`. However, a name that reflects the sort of object being prototyped, such as `registerSymbol`, is more appropriate in an actual implementation.

PrototypeIF

All prototypical objects that a client object clones to create new objects must belong to a class that implements an interface such as this. A `PrototypeIF` interface minimally declares a clone method that is intended to return a new copy of an object. Normally, additional methods are declared for the client to use to manipulate objects that implement the `PrototypeIF` interface.

The name of the interface used in this role is not normally `PrototypeIF`, but rather a name that indicates the sort of object that will be implementing the interface, like `SymbolIF`.

Prototype

This corresponds to multiple classes that implement the `PrototypeIF` interface and are instantiated for the purpose of being cloned by the client.

PrototypeBuilder

This corresponds to any class that is instantiated to supply prototypical objects to the client object. Such classes should have a name that denotes the type of prototypical object that they build, such as `SymbolBuilder`.

Consequences

- Prototypical objects can be dynamically added and removed at run time. That is a distinct advantage offered by none of the other creational patterns in this book.

- A `PrototypeBuilder` object can simply supply a fixed set of prototypical objects. A `PrototypeBuilder` object provides the additional flexibility of allowing new kinds of prototypical objects to be created by object composition and changes to the values of object attributes.

- The client object may also be able to create new kinds of prototypical objects. In the drawing program example we looked at previously, the client object could very reasonably allow the user to identify a sub-drawing and then turn the sub-drawing into a new symbol.

- The client class is independent of the exact class that of the prototypical objects that it uses. Also, the client class does not need to know the details of how to build the prototypical objects.

- The `PrototypeBuilder` objects encapsulate the details of constructing prototypical objects.

- By insisting that prototypical objects implement an interface such a s `PrototypeIF`, the Prototype pattern ensures that the prototypical objects provide a consistent set of methods for the client object to use.

- A drawback of the Prototype pattern is the additional work to writing `PrototypeBuilder` classes.

- Programs that use the Prototype pattern rely on dynamic linking or similar mechanisms. Installation of programs that rely on dynamic linking or similar mechanisms can be more complicated.

- There is no need to organize prototypical objects into any sort of class hierarchy.

Implementation

An essential implementation issue is how the `PrototypeBuilder` objects add objects to a client object's palette of prototypical objects. The simplest strategy is for the client class to provide a method for that purpose that can be called by `PrototypeBuilder` objects. A possible drawback to that is that the `PrototypeBuilder` objects will need so know the class of the client object. If that is a problem, the `PrototypeBuilder` objects can be shielded from knowing the exact class of the client objects by providing an interface or abstract class for the client object.

How to implement the clone operation for the prototypical objects is another important implementation issue. There are two basic strategies for implementing the clone operation:

- Shallow copying means that the variables of the cloned object contain the same values as the variables of the original object and that all object references are to the same objects. In other words, only the object being cloned is copied, not that objects that it refers to.

- Deep copying means that the variables of the cloned object contain the same values as the variables of the original object, except that those variables that refer to objects refer to copies of the objects referred to by the original object. Implementing deep copying can be tricky. You will need to decide if you want to make deep or shallow copies of the indirectly copied objects. You will also need to be careful about handling any circular references.

If you are coding in Java, the shallow copying is easy to implement because all classes inherit a clone method that does just that. However, unless an object's class implements the `Cloneable` interface, it will refuse to work. If all of the prototypical objects your program uses will be cloning themselves by shallow copying, you can save yourself some work by declaring the `PrototypeIF` interface to extend the `Cloneable` interface. That way, all classes that implement the `PrototypeIF` interface also implement the `Cloneable` interface.

Unless the client object's palette of prototypical objects consists of a fixed number of objects having fixed purposes, it will be inconvenient to use individual variables to refer each prototypical object. Instead, you will want to use some sort of collection object that can contain a dynamically growing or shrinking palette of prototypical objects. A collection object that plays this role in the Prototype pattern is called a prototype manage*r*. Prototype managers can be fancier than just a simple collection. They may allow objects to be retrieved by their attribute values or other keys.

If your program will have multiple client objects, then you have another issue to consider. Will the client objects have their own palette of prototypical objects or will they all share the same palette? The answer will depend of the needs of your application.

JAVA API Usage

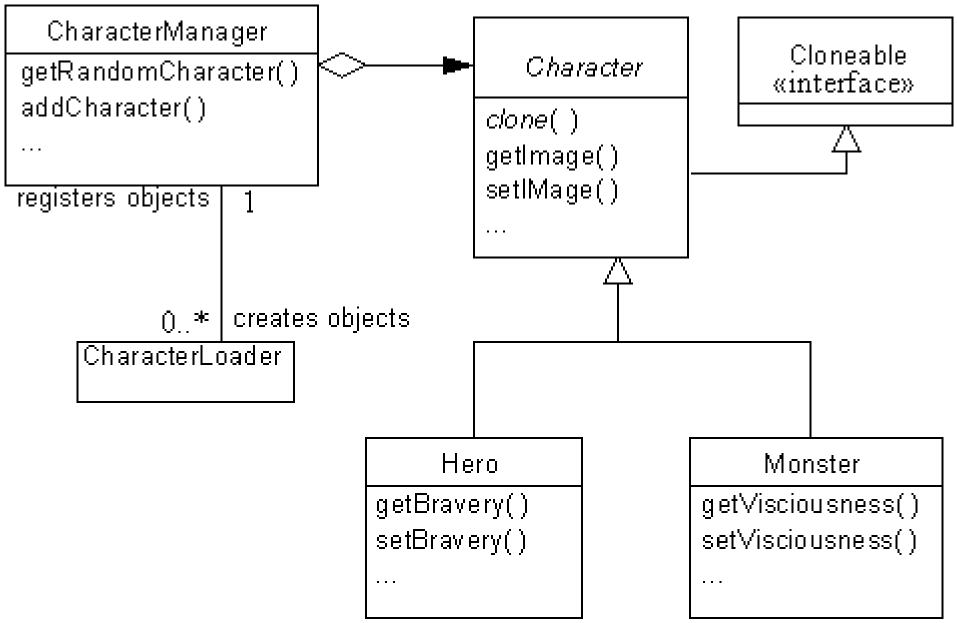
The Prototype pattern is the very essence of Java Beans. Java Beans are instances of classes that conform to certain naming conventions that allow a bean creation program to know how to customize them. After a bean object has been customized for use in an application, the object is saved to a file to be loaded by the application while it is running.

Example

Suppose that you are writing an interactive role playing game. That is, a game that allows the user to interact with computer simulated characters. One of the expectations for this game is that the people who play it will grow tired of interacting with the same characters and want to interact with new characters. Because of that expectation, you are also developing an add-on to the game that consists of a few pre-generated characters and a program to generate additional characters.

The characters in the game are instances of a relatively small number of classes such as Hero, Fool, Villain and Monster. What makes instances of the same class different from each other is the different attributes values that are set for them, such as the images that are used to represent them on the screen, height, weight, intelligence and dexterity.

Here is a class diagram that shows some of the classes involved in the game:



Prototype Example

Here is the code for the Character class, an abstract class that serves in the role of PrototypeIF:

```
public abstract class Character implements Cloneable {
    private String name;
    private Image image;
    private int strength;
...
    /**
     * Override clone to make it public.
     */
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            // should never happen because this class implements
            // Cloneable.
            throw new InternalError();
        } // try
    } // clone()

    public String getName() {
        return name;
    } // getName()

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

    public Image getImage() {
        return image;
    } // getImage(Image)

    public void setImage(Image image) {
        this.image = image;
    } // setImage(Image)

    public int getStrength() {
```

```
        return strength;
    } // getStrength()

    public void setStrength(int strength) {
        this.strength = strength;
    } // setStrength(int)
...
} // class Character
```

As you can see most of this is just simple accessor methods. The one less than obvious method is the clone method. All objects inherit a clone method from the Object class. Because that clone method is not public, the character class must override it with a public declaration, just to make it accessible to other classes.

Here is source code for the Hero and Monster classes that serve as Prototype classes:

```
public class Hero extends Character {
    private int bravery;
...
    public int getBravery() {
        return bravery;
    } // getBravery()

    public void setBravery(int bravery) {
        this.bravery = bravery;
    } // setBravery(int)
} // class Hero

public class Monster extends Character {
    private int visciousness;
...
    public int getVisciousness() {
        return visciousness;
    } // getVisciousness()

    public void setVisciousness(int visciousness) {
        this.visciousness = visciousness;
    } // setVisciousness(int)
} // class Monster
```

Here is the code for the `CharacterManager` class that serves in the role of client class:

```
/**
 * This class manages the collection of prototypical objects for the
 * game.  When asked to, it clones an appropriate prototypical object
 * and returns it to the requesting object.
 */
public class CharacterManager {
    private Vector characters = new Vector();
...
    /**
     * Return a random character from the collection.
     */
    Character getRandomCharacter() {
        int i = (int)(characters.size()*Math.random());
        return (Character)((Character)characters.elementAt(i)).clone();
    } // getRandomCharacter()
    /**
     * Add a prototypical object to the collection.
     * @param character The character to add.
     */
```

```
    void addCharacter(Character character) {
        characters.addElement(character);
    } // addCharacter(Character)
...
} // class CharacterManager
```

Here is the code for the `CharacterLoader` class that fills the role of PrototypeBuilder:

```
/**
 * This class loads character objects and adds them to the
 * the CharacterManager.
 */
class CharacterLoader {
    private CharacterManager mgr;
    /**
     * Constructor
     * @param cm The CharacterManager that this object will work with.
     */
    CharacterLoader(CharacterManager cm) {
        mgr = cm;
    } // Constructor(CharacterManager)
    /**
     * Load character objects from the specified file.
     * Since failure only affects the rest of the program to the extent
     * that new character objects are not loaded, we need not throw any
     * exceptions.
     * @param fname The name of the file to read objects from.
     * @return The number of Charcter objects loaded.
     */
    int loadCharacters(String fname) {
        int objectCount = 0;    // The number of objects loaded
        // If construction of the InputStream fails, just return
        try {
            InputStream in;
            in = new FileInputStream(fname);
            in = new BufferedInputStream(in);
            ObjectInputStream oIn = new ObjectInputStream(in);
            while(true) {
                Object c = oIn.readObject();
                if (c instanceof Character) {
                    mgr.addCharacter((Character)c);
                } // if
            } // while
        } catch (Exception e) {
        } // try
        return objectCount;
    } // loadCharacters(String)
} // class CharacterLoader
```

Related Patterns

Abstract Factory

The Abstract Factory pattern can be a good alternative to the Prototype pattern where the dynamic changes that the Prototype pattern allows to the prototypical object palette are not needed.

PrototypeBuilder classes may use the Abstract Factory pattern to create a set of prototypical objects.

Facade

The client class commonly acts as facade that separates the other classes that participate in the Prototype pattern from the rest of the program.

Factory Method

The Factory Method pattern can be an alternative to the Prototype pattern when the palette of prototypical objects never contains more than ne object.

Composite and Wrapper

The Prototype pattern is often used with the Composite and Wrapper patterns.PrototypePrototype

Singleton

Synopsis

The Singleton pattern ensures that all objects that use an instance of a class use the same instance of that class.

Context

Some classes should have exactly one instance. These classes usually involve the centrally management of a resource. The resource may be external, as is the case with an object than manages the reuse of database connections. The resource may be internal, such as an object that keeps an error count and other statistics for a compiler.

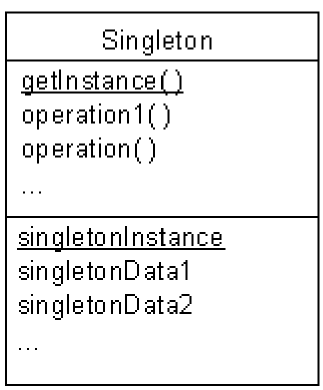
Forces

- There must be exactly one instance of a class.

- The one instance of a class must be accessible to all clients of that class.

Solution

The Singleton pattern is relatively simple, since it only involves one class:



Singleton

A singleton class has a static variable that refers to the one instance of the class that you want use. That instance is created when the class is loaded into memory. The class should be implemented in a way that prevents other classes from creating any additional instances of a singleton class. That means ensuring that all of the class' constructors are private.

To access the one instance of a singleton class, the class provides a static method, typically called `getInstance` or `get`*`Classname`*, which returns a reference to the one instance of the class.

Consequences

- Exactly one instance of a singleton class exists.
- Other classes that want a reference to the one instance of the singleton class must get that instance by calling the class' `getInstance` static method, rather than by constructing the instance themselves.

Implementation

To enforce the nature of a singleton class, the class must be coded in a way that prevents other classes from creating instances of the class. The way to accomplish that is to declare all of the class' constructors private. Be careful to declare at least one private constructor. If a class does not declare any constructors, then a default public constructor is generated for it.

One common variation on the Singleton pattern occurs in situations where the instance of a Singleton may not be needed. In situations like that, you can use the Lazy Materialization pattern to postpone creation of the instance until it is needed.

JAVA API Usage

The Java API class `java.lang.Runtime` is a singleton class. It has exactly one instance. It has no public constructors. To get a reference to its one instance, other classes must call its static method `getRuntime`.

Example

Here is a Java class that can be used to avoid playing two audio clips at the same time. The class is a singleton class. Its instance can be accessed by calling its static `getInstance` method. When you play audio clips through that object, it stops the last audio clip it was playing before it starts the newly requested one. If all audio clips are played through the `AudioClipManager` object then there will never be more than one audio clip playing at the same time.

```
public class AudioClipManager implements AudioClip{
    private static AudioClipManager instance = new AudioClipManager();
    private AudioClip prevClip; // previously requested audio clip
    /**
     * Return a reference to the only instance of this class.
     */
    public static AudioClipManager getInstance() {
        return instance;
    } // getInstance()
    /**
     * Starts playing this audio clip. Each time this method is called,
     * the clip is restarted from the beginning.
     */
    public void play() {
        if (prevClip != null)
          prevClip.play();
    } // play()
    /**
     * Stop the previously requested audio clip and play the given
     * audio clip.
     * @param clip the new audio clip to play.
     */
    public void play(AudioClip clip) {
        if (prevClip != null)
          prevClip.stop();
```

```
            prevClip = clip;
            clip.play();
        } // play(AudioClip)
        /**
         * Starts playing this audio clip in a loop.
         */
        public void loop() {
            if (prevClip != null)
              prevClip.loop();
        } // loop()
        /**
         * Stop the previously requested audio clip and play the given
         * audio clip in a loop.
         * @param clip the new audio clip to play.
         */
        public void loop(AudioClip clip) {
            if (prevClip != null)
              prevClip.stop();
            prevClip = clip;
            clip.loop();
        } // play(AudioClip)
        /**
         * Stops playing this audio clip.
         */
        public void stop() {
            if (prevClip != null)
              prevClip.stop();
        } // stop()
    } // class AudioClipManager
```

Related Patterns

The Singleton pattern can be used with many other patterns. In particular, it is often used with the Abstract Factory, Builder and Prototype patterns.Singleton