Part I: Commonly Used Patterns

Chapter 1. Creational Patterns

Introduction to Creational Patterns

These patterns support one of the most common tasks in object-oriented programming—the creation of objects in a system. Most OO systems of any complexity require many objects to be instantiated over time, and these patterns support the creation process by helping to provide the following capabilities:

Generic instantiation – This allows objects to be created in a system without having to identify a specific class type in code.

Simplicity – Some of the patterns make object creation easier, so callers will not have to write large, complex code to instantiate an object.

Creation constraints – Some patterns enforce constraints on the type or number of objects that can be created within a system.

The following patterns are discussed in this chapter:

Abstract Factory – To provide a contract for creating families of related or dependent objects without having to specify their concrete classes.

Builder – To simplify complex object creation by defining a class whose purpose is to build instances of another class. The Builder produces one main product, such that there might be more than one class in the product, but there is always one main class.

Factory Method – To define a standard method to create an object, apart from a constructor, but the decision of what kind of an object to create is left to subclasses.

Prototype – To make dynamic creation easier by defining classes whose objects can create duplicates of themselves.

Singleton – To have only one instance of this class in the system, while allowing other classes to get access to this instance.

Of these patterns, the Abstract Factory and Factory Method are explicitly based on the concept of defining flexible object creation; they assume that the classes or interfaces to be created will be extended in an implementing system. As a result, these two patterns are frequently combined with other creational patterns.

Abstract Factory

Also known as Kit, Toolkit

Pattern Properties

Type: Creational, Object

Level: Component

Purpose

To provide a contract for creating families of related or dependent objects without having to specify their concrete classes.

Introduction

Suppose you plan to manage address and telephone information as part of a personal information manager (PIM) application. The PIM will act as a combination address book, personal planner, and appointment and contact manager, and will use the address and phone number data extensively.

You can initially produce classes to represent your address and telephone number data. Code these classes so that they store the relevant information and enforce business rules about their format. For example, all phone numbers in North America are limited to ten digits and the postal code must be in a particular format.

Shortly after coding your classes, you realize that you have to manage address and phone information for another country, such as the Netherlands. The Netherlands has different rules governing what constitutes a valid phone number and address, so you modify your logic in the `Address` and `PhoneNumber` classes to take the new country into account.

Now, as your personal network expands, you need to manage information from another foreign country... and another... and another. With each additional set of business rules, the base `Address` and `PhoneNumber` classes become even more bloated with code and even more difficult to manage. What's more, this code is brittle—with every new country added, you need to modify and recompile the classes to manage contact information.

It’s better to flexibly add these paired classes to the system; to take the general rules that apply to address and phone number data, and allow any number of possible foreign variations to be “loaded” into a system.

The Abstract Factory solves this problem. Using this pattern, you define an *AddressFactory*—a generic framework for producing objects that follow the general pattern for an `Address` and `PhoneNumber`. At runtime, this factory is paired with any number of concrete factories for different countries, and each country has its own version of `Address` and `PhoneNumber` classes.

Instead of going through the nightmare of adding functional logic to the classes, extend the `Address` to a `DutchAddress` and the `PhoneNumber` to a `DutchPhoneNumber`. Instances of both classes are created by a `DutchAddressFactory`. This gives greater freedom to extend your code without having to make major structural modifications in the rest of the system.

Applicability

Use the Abstract Factory pattern when:

The client should be independent of how the products are created.

The application should be configured with one of multiple families of products.

Objects need to be created as a set, in order to be compatible.

You want to provide a collection of classes and you want to reveal just their contracts and their relationships, not their implementations.

Description

Sometimes an application needs to use a variety of different resources or operating environments. Some common examples include:

Windowing (an application's GUI)

A file system

Communication with other applications or systems

In this sort of application you want to make the application flexible enough to use a variety of these resources without having to recode the application each time a new resource is introduced.

An effective way to solve this problem is to define a generic resource creator, the Abstract Factory . The factory has one or more create methods, which can be called to produce generic resources or abstract products.

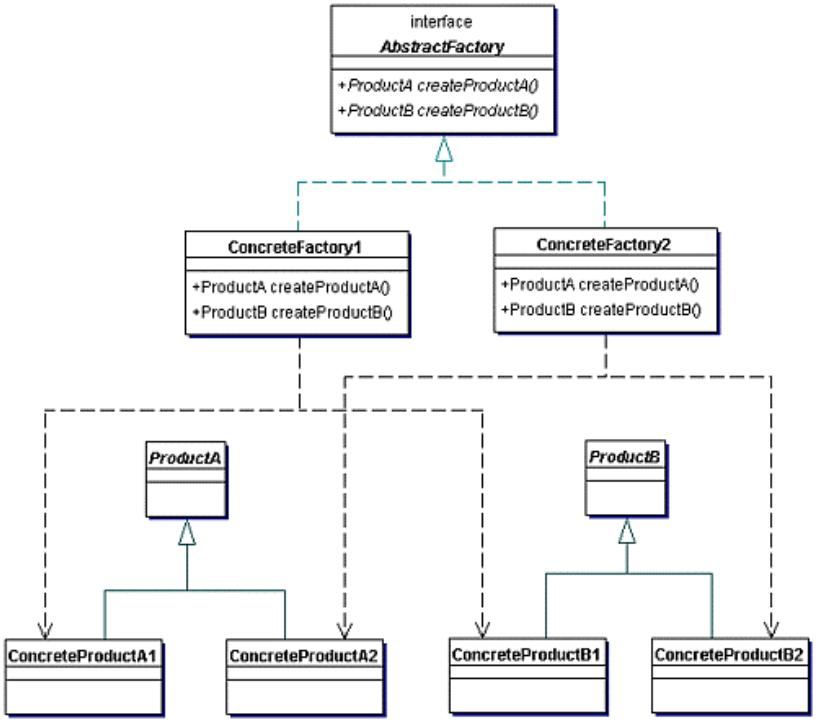
Java ("Java technology") runs on many platforms, each with many different implementations of a file system or windowing. The solution Java has taken is to abstract the concepts of files and windowing and not show the concrete implementation. You can develop the application using the generic capabilities of the resources as though they represented real functionality.

During runtime, `ConcreteFactories` and `ConcreteProducts` are created and used by the application. The concrete classes conform to the contract defined by the `AbstractFactory` and `AbstractProducts`, so the concrete classes can be directly used, without being recoded or recompiled.

Implementation

The Abstract Factory class diagram is shown in Figure 1.1.

Figure 1.1. Abstract Factory class diagram



You typically use the following to implement the Abstract Factory pattern:

`AbstractFactory` - An abstract class or interface that defines the create methods for abstract products.

`AbstractProduct` - An abstract class or interface describing the general behavior of the resource that will be used by the application.

`ConcreteFactory` – A class derived from the abstract factory . It implements create methods for one or more concrete products.

`ConcreteProduct` – A class derived from the abstract product, providing an implementation for a specific resource or operating environment.

Benefits and Drawbacks

An Abstract Factory helps to increase the overall flexibility of an application. This flexibility manifests itself both during design time and runtime. During design, you do not have to predict all future uses for an application. Instead, you create the generic framework and then develop implementations independently from the rest of the application. At runtime, the application can easily integrate new features and resources.

A further benefit of this pattern is that it can simplify testing the rest of the application. Implementing a `TestConcreteFactory` and `TestConcreteProduct` is simple ; it can simulate the expected resource behavior.

To realize the benefits of this pattern, carefully consider how to define a suitably generic interface for the abstract product. If the abstract product is improperly defined, producing some of the desired concrete products can be difficult or impossible.

Pattern Variants

As mentioned earlier, you can define the `AbstractFactory` and `AbstractProduct` as an interface or an abstract class, depending on the needs of the application and your preference.

Depending on how the factory is to be used, some variations of this pattern allow multiple `ConcreteFactory` objects to be produced, resulting in an application that can simultaneously use multiple families of `ConcreteProducts`.

Related Patterns

Related patterns include the following:

Factory Method (page 21) – Used to implement the Abstract Factory.

Singleton (page 34) – Often used in the Concrete Factory.

Data Access Object [CJ2EEP] – The Data Access Object pattern can use the Abstract Factory pattern to add flexibility in creating Database-specific factories.

Note –

“ [CJ2EEP] ” refers to J2EE patterns, listed in the bibliography (see page 559).

Example

The following code shows how international addresses and phone numbers can be supported in the Personal Information Manager with the Abstract Factory pattern. The `AddressFactory` interface represents the factory itself:

Example 1.1 `AddressFactory.java`

```
1.    public interface AddressFactory {
2.        public Address createAddress();
3.        public PhoneNumber createPhoneNumber();
4.    }
```

Note that the `AddressFactory` defines two factory methods, `createAddress` and `createPhoneNumber`. The methods produce the abstract products `Address` and `PhoneNumber`, which define methods that these products support.

Example 1.2 `Address.java`

```
1. public abstract class Address {
2.     private String street;
3.     private String city;
```

```
4.    private String region;
5.    private String postalCode;
6.
7.    public static final String EOL_STRING =
8.          System.getProperty("line.separator");
9.    public static final String SPACE = " ";
10.
11.   public String getStreet() {
         return street;
      }
12.   public String getCity() {
         return city;
      }
13.   public String getPostalCode() {
         return postalCode;
      }
14.   public String getRegion() {
         return region;
      }
15.   public abstract String getCountry();
16.
17.   public String getFullAddress() {
18.       return street + EOL_STRING +
19.            city + SPACE + postalCode + EOL_STRING;
20.   }
21.
22.   public void setStreet(String newStreet) {
         street = newStreet;
      }
23.   public void setCity(String newCity) {
         city = newCity;
      }
24.   public void setRegion(String newRegion) {
         region = newRegion;
      }
25.   public void setPostalCode(String newPostalCode) {
         postalCode = newPostalCode;
      }
26. }
```

Example 1.3 PhoneNumber.java

```
1. public abstract class PhoneNumber{
2.    private String phoneNumber;
3.    public abstract String getCountryCode();
4.
5.    public String getPhoneNumber() {
         return phoneNumber;
      }
6.
7.    public void setPhoneNumber(String newNumber) {
8.       try {
9.          Long.parseLong(newNumber);
10.         phoneNumber = newNumber;
11.      }
12.      catch (NumberFormatException exc) {
13.      }
14.   }
15. }
```

`Address` and `PhoneNumber` are abstract classes in this example, but could easily be defined as interfaces if you did not need to define code to be used for all concrete products.

To provide concrete functionality for the system, you need to create Concrete Factory and Concrete Product classes. In this case, you define a class that implements `AddressFactory`, and subclass the `Address and PhoneNumber` classes. The three following classes show how to do this for U.S. address information.

Example 1.4 USAddressFactory.java

```
1.   public class USAddressFactory implements AddressFactory{
2.      public Address createAddress(){
3.         return new USAddress();
4.      }
5.
6.      public PhoneNumber createPhoneNumber(){
7.         return new USPhoneNumber();
8.      }
```

```
9.    }
```

Example 1.5 USAddress.java

```
1.    public class USAddress extends Address{
2.       private static final String COUNTRY = "UNITED STATES";
3.       private static final String COMMA = ",";
4.
5.       public String getCountry(){ return COUNTRY; }
6.
7.       public String getFullAddress(){
8.          return getStreet() + EOL_STRING +
9.             getCity() + COMMA + SPACE + getRegion() +
10.            SPACE + getPostalCode() + EOL_STRING +
11.            COUNTRY + EOL_STRING;
12.      }
13.   }
```

Example 1.6 USPhoneNumber.java

```
1.    public class USPhoneNumber extends PhoneNumber{
2.       private static final String COUNTRY_CODE = "01";
3.       private static final int NUMBER_LENGTH = 10;
4.
5.       public String getCountryCode(){ return COUNTRY_CODE; }
6.
7.       public void setPhoneNumber(String newNumber){
8.          if (newNumber.length() == NUMBER_LENGTH){
9.             super.setPhoneNumber(newNumber);
10.         }
11.      }
12.   }
```

The generic framework from `AddressFactory`, `Address`, and `PhoneNumber` makes it easy to extend the system to support additional countries. With each additional country, define an additional Concrete Factory class and a matching Concrete Product class. These are files for French address information.

Example 1.7 FrenchAddressFactory.java

```
1.    public class FrenchAddressFactory implements AddressFactory{
2.       public Address createAddress(){
3.          return new FrenchAddress();
4.       }
5.
6.       public PhoneNumber createPhoneNumber(){
7.          return new FrenchPhoneNumber();
8.       }
9.    }
```

Example 1.8 FrenchAddress.java

```
1.    public class FrenchAddress extends Address{
2.       private static final String COUNTRY = "FRANCE";
3.
4.       public String getCountry(){ return COUNTRY; }
5.
6.       public String getFullAddress(){
7.          return getStreet() + EOL_STRING +
8.             getPostalCode() + SPACE + getCity() +
9.             EOL_STRING + COUNTRY + EOL_STRING;
10.      }
11.   }
```

Example 1.9 FrenchPhoneNumber.java

```
1.    public class FrenchPhoneNumber extends PhoneNumber{
2.       private static final String COUNTRY_CODE = "33";
3.       private static final int NUMBER_LENGTH = 9;
4.
5.       public String getCountryCode(){ return COUNTRY_CODE; }
6.
7.       public void setPhoneNumber(String newNumber){
8.          if (newNumber.length() == NUMBER_LENGTH){
9.             super.setPhoneNumber(newNumber);
10.         }
11.      }
12.   }
```

Builder

Pattern Properties

Type: Creational, Object

Level: Component

Purpose

To simplify complex object creation by defining a class whose purpose is to build instances of another class. The Builder produces one main product, such that there might be more than one class in the product, but there is always one main class.

Introduction

In a Personal Information Manager, users might want to manage a social calendar. To do this, you might define a class called `Appointment` to the information for a single event, and track information like the following:

Starting and ending dates

A description of the appointment

A location for the appointment

Attendees for the appointment

Naturally, this information is passed in by a user when he or she is setting up the appointment, so you define a constructor that allows you to set the state of a new `Appointment` object.

What exactly is needed to create an appointment, though? Different kinds of information are required depending on the specific type of the appointment. Some appointments might require a list of attendees (the monthly Monty Python film club meeting). Some might have start and end dates (JavaOne conference) and some might only have a single date—a plan to visit the art gallery for the M.C. Escher exhibit. When you consider these options, the task of creating an `Appointment` object is not trivial.

There are two possibilities for managing object creation, neither of them particularly attractive. You create constructors for every type of appointment you want to create, or you write an enormous constructor with a lot of functional logic. Each approach has its drawbacks—with multiple constructors, calling logic becomes more complex; with more functional logic built into the constructor, the code becomes more complex and harder to debug. Worse still, both approaches have the potential to cause problems if you later need to subclass `Appointment`.

Instead, delegate the responsibility of `Appointment` creation to a special `AppointmentBuilder` class, greatly simplifying the code for the `Appointment` itself. The `AppointmentBuilder` contains methods to create the parts of the `Appointment`, and you call the `AppointmentBuilder` methods that are relevant for the appointment type. Additionally, the `AppointmentBuilder` can ensure that the information passed in when creating the `Appointment` is valid, helping to enforce business rules. If you need to subclass `Appointment`, you either create a new builder or subclass the existing one. In either case, the task is easier than the alternative of managing object initialization through constructors.

Applicability

Use the Builder pattern when a class:

Has complex internal structure (especially one with a variable set of related objects).

Has attributes that depend on each other. One of the things a Builder can do is enforce staged construction of a complex object. This would be required when the Product attributes depend on one another. For instance, suppose you're building an order. You might need to ensure that you have a state set before you move on to "building" the shipping method, because the state would impact the sales tax applied to the Order itself.

Uses other objects in the system that might be difficult or inconvenient to obtain during creation.

Description

Because this pattern is concerned with building a complex object from possibly multiple different sources, it is called the Builder. As object creation increases in complexity, managing object creation from within the constructor method can become difficult. This is especially true if the object does not depend exclusively on resources that are under its own control.

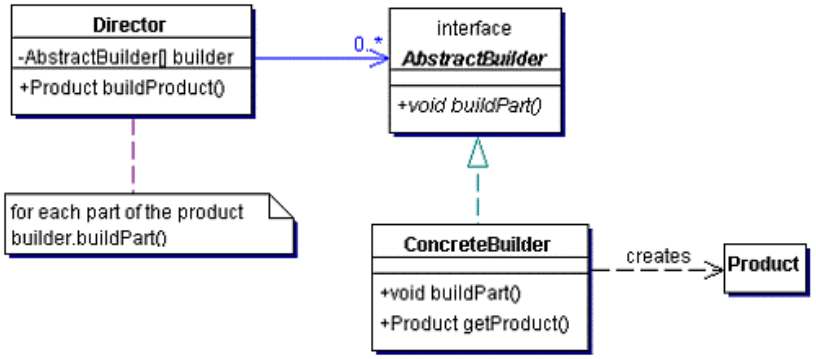
Business objects often fall into this category. They frequently require data from a database for initialization and might need to associate with a number of other business objects to accurately represent the business model. Another example is that of composite objects in a system, such as an object representing a drawing in a visual editing program. Such an object might need to be related to an arbitrary number of other objects as soon as it's created.

In cases like this, it is convenient to define another class (the Builder) that is responsible for the construction. The Builder coordinates the assembly of the product object: creating resources, storing intermediate results, and providing functional structure for the creation. Additionally, the Builder can acquire system resources required for construction of the product object.

Implementation

The Builder class diagram is shown in Figure 1.2.

Figure 1.2. Builder class diagram



To implement the Builder pattern, you need:

`Director` – Has a reference to an `AbstractBuilder` instance. The `Director` calls the creational methods on its builder instance to have the different parts and the Builder build.

`AbstractBuilder` – The interface that defines the available methods to create the separate parts of the product.

`ConcreteBuilder` – Implements the `AbstractBuilder` interface. The `ConcreteBuilder` implements all the methods required to create a real `Product`. The implementation of the methods knows how to process information from the `Director` and build the respective parts of a `Product`. The `ConcreteBuilder` also has either a `getProduct` method or a creational method to return the `Product` instance.

`Product` – The resulting object. You can define the product as either an interface (preferable) or class.

Benefits and Drawbacks

The Builder pattern makes it easier to manage the overall flow during the creation of complex objects. This manifests itself in two ways:

For objects that require phased creation (a sequence of steps to make the object fully active), the Builder acts as a higher-level object to oversee the process. It can coordinate and validate the creation of all resources and if necessary provide a fallback strategy if errors occur.

For objects that need existing system resources during creation, such as database connections or existing business objects, the Builder provides a convenient central point to manage these resources. The Builder also provides a

single point of creational control for its product, which other objects within the system can use. Like other creational patterns, this makes things easier for clients in the software system, since they need only access the Builder object to produce a resource.

The main drawback of this pattern is that there is tight coupling among the Builder, its product, and any other creational delegates used during object construction. Changes that occur for the product created by the Builder often result in modifications for both the Builder and its delegates.

Pattern Variants

At the most fundamental level, it is possible to implement a bare-bones Builder pattern around a single `Builder` class with a creational method and its product. For greater flexibility, designers often extend this base pattern with one or more of the following approaches:

Create an abstract `Builder`. By defining an abstract class or interface that specifies the creational methods, you can produce a more generic system that can potentially host many different kinds of builders.

Define multiple create methods for the `Builder`. Some Builders define multiple methods (essentially, they overload their creational method) to provide a variety of ways to initialize the constructed resource.

Develop creational delegates. With this variant, a `Director` object holds the overall `Product` create method and calls a series of more granular create methods on the `Builder` object. In this case, the `Director` acts as the manager for the Builder's creation process.

Related Patterns

Related patterns include Composite (page 157). The Builder pattern is often used to produce Composite objects, since they have a very complex structure.

Example

Note:

For a full working example of this code example, with additional supporting classes and/or a RunPattern class, see " Builder " on page 343 of the " Full Code Examples " appendix.

This code example shows how to use the Builder pattern to create an appointment for the PIM. The following list summarizes each class's purpose:

`AppointmentBuilder, MeetingBuilder` – Builder classes

`Scheduler` – Director class

`Appointment` – Product

`Address, Contact` – Support classes, used to hold information relevant to the Appointment

`InformationRequiredException` – An Exception class produced when more data is required

For the base pattern, the AppointmentBuilder manages the creation of a complex product, an Appointment here. The AppointmentBuilder uses a series of build methods—buildAppointment, buildLocation, buildDates, and buildAttendees—to create an Appointment and populate it with data.

Example 1.10 `AppointmentBuilder.java`

```
1.    import java.util.Date;
2.    import java.util.ArrayList;
3.
4.    public class AppointmentBuilder{
5.
6.        public static final int START_DATE_REQUIRED = 1;
7.        public static final int END_DATE_REQUIRED = 2;
8.        public static final int DESCRIPTION_REQUIRED = 4;
```

```
9.       public static final int ATTENDEE_REQUIRED = 8;
10.      public static final int LOCATION_REQUIRED = 16;
11.
12.      protected Appointment appointment;
13.
14.      protected int requiredElements;
15.
16.      public void buildAppointment(){
17.         appointment = new Appointment();
18.      }
19.
20.      public void buildDates(Date startDate, Date endDate){
21.         Date currentDate = new Date();
22.         if ((startDate != null) && (startDate.after(currentDate))){
23.            appointment.setStartDate(startDate);
24.         }
25.         if ((endDate != null) && (endDate.after(startDate))){
26.            appointment.setEndDate(endDate);
27.          }
28.       }
29.
30.       public void buildDescription(String newDescription){
31.          appointment.setDescription(newDescription);
32.       }
33.
34.       public void buildAttendees(ArrayList attendees){
35.          if ((attendees != null) && (!attendees.isEmpty())){
36.             appointment.setAttendees(attendees);
37.          }
38.       }
39.
40.       public void buildLocation(Location newLocation){
41.          if (newLocation != null){
42.             appointment.setLocation(newLocation);
43.          }
44.       }
45.
46.       public Appointment getAppointment() throws InformationRequiredException{
47.          requiredElements = 0;
48.
49.          if (appointment.getStartDate() == null){
50.             requiredElements += START_DATE_REQUIRED;
51.          }
52.
53.          if (appointment.getLocation() == null){
54.             requiredElements += LOCATION_REQUIRED;
55.          }
56.
57.          if (appointment.getAttendees().isEmpty()){
58.             requiredElements += ATTENDEE_REQUIRED;
59.          }
60.
61.          if (requiredElements > 0){
62.             throw new InformationRequiredException(requiredElements);
63.          }
64.          return appointment;
65.       }
66.
67.       public int getRequiredElements(){ return requiredElements; }
68.   }
```

Example 1.11 `Appointment.java`

```
1.    import java.util.ArrayList;
2.    import java.util.Date;
3.    public class Appointment{
4.       private Date startDate;
5.       private Date endDate;
6.       private String description;
7.       private ArrayList attendees = new ArrayList();
8.       private Location location;
9.       public static final String EOL_STRING =
10.         System.getProperty("line.separator");
11.
12.      public Date getStartDate(){ return startDate; }
13.      public Date getEndDate(){ return endDate; }
14.      public String getDescription(){ return description; }
15.      public ArrayList getAttendees(){ return attendees; }
```

```
16.      public Location getLocation(){ return location; }
17.
18.      public void setDescription(String newDescription){ description = newDescription; }
19.      public void setLocation(Location newLocation){ location = newLocation; }
20.      public void setStartDate(Date newStartDate){ startDate = newStartDate; }
21.      public void setEndDate(Date newEndDate){ endDate = newEndDate; }
22.      public void setAttendees(ArrayList newAttendees){
23.         if (newAttendees != null){
24.            attendees = newAttendees;
25.         }
26.      }
27.
28.      public void addAttendee(Contact attendee){
29.         if (!attendees.contains(attendee)){
30.            attendees.add(attendee);
31.         }
32.      }
33.
34.      public void removeAttendee(Contact attendee){
35.          attendees.remove(attendee);
36.      }
37.
38.      public String toString(){
39.         return " Description: " + description + EOL_STRING +
40.            " Start Date: " + startDate + EOL_STRING +
41.            " End Date: " + endDate + EOL_STRING +
42.            " Location: " + location + EOL_STRING +
43.            " Attendees: " + attendees;
44.      }
45.   }
```

The `Scheduler` class makes calls to the `AppointmentBuilder`, managing the creation process through the method `createAppointment`.

Example 1.12 `Scheduler.java`

```
1.    import java.util.Date;
2.    import java.util.ArrayList;
3.    public class Scheduler{
4.       public Appointment createAppointment(AppointmentBuilder builder,
5.          Date startDate, Date endDate, String description,
6.          Location location, ArrayList attendees) throws InformationRequiredException{
7.             if (builder == null){
8.                builder = new AppointmentBuilder();
9.             }
10.         builder.buildAppointment();
11.         builder.buildDates(startDate, endDate);
12.         builder.buildDescription(description);
13.         builder.buildAttendees(attendees);
14.         builder.buildLocation(location);
15.         return builder.getAppointment();
16.      }
17.   }
```


The responsibilities of each class are summarized here:

`Scheduler` – Calls the appropriate build methods on `AppointmentBuilder`; returns a complete `Appointment` object to its caller.

`AppointmentBuilder` – Contains build methods and enforces business rules; creates the actual `Appointment` object.

`Appointment` – Holds information about an appointment.

The `MeetingBuilder` class in Example 1.13 demonstrates one of the benefits of the Builder pattern. To add additional rules for the `Appointment`, extend the existing builder. In this case, the `MeetingBuilder` enforces an additional constraint: for a meeting `Appointment`, start and end dates must be specified.

Example 1.13 `MeetingBuilder.java`

```
1.    import java.util.Date;
2.    import java.util.Vector;
3.
4.    public class MeetingBuilder extends AppointmentBuilder{
```

```
5.       public Appointment getAppointment() throws InformationRequiredException {
6.          try {
7.             super.getAppointment();
8.          }
9.          finally {
10.            if (appointment.getEndDate() == null) {
11.               requiredElements += END_DATE_REQUIRED;
12.            }
13.
14.            if (requiredElements > 0) {
15.               throw new InformationRequiredException(requiredElements);
16.            }
17.         }
18.         return appointment;
19.      }
20.   }
```

Factory Method

Also known as Virtual Constructor

Pattern Properties

Type: Creational

Level: Class

Purpose

To define a standard method to create an object, apart from a constructor, but the decision of what kind of an object to create is left to subclasses.

Introduction

Imagine that you're working on a Personal Information Manager (PIM) application. It will contain many pieces of information essential to your daily life: addresses, appointments, dates, books read, and so on. This information is not static; for instance, you want to be able to change an address when a contact moves, or change the details of an appointment if your lunch date needs to meet an hour later.

The PIM is responsible for changing each field. It therefore has to worry about editing (and therefore the User Interface) and validation for each field. The big disadvantage, however, is that the PIM has to be aware of all the different types of appointments and tasks that can be performed on them. Each item has different fields and the user needs to see an input screen appropriate to those fields. It will be very difficult to introduce new types of task information, because you will have to add a new editing capability to the PIM every time, suitable to update the new item type. Furthermore, every change in a specific type of task, such as adding a new field to an appointment, means you also have to update the PIM so that it is aware of this new field. You end up with a very bloated PIM that is difficult to maintain.

The solution is to let items, like appointments, be responsible for providing their own editors to manage additions and changes. The PIM only needs to know how to request an editor using the method `getEditor`, which is in every editable item. The method returns an object that implements the `ItemEditor` interface, and the PIM uses that object to request a `JComponent` as the GUI editor. Users can modify information for the item they want to edit, and the editor ensures that the changes are properly applied.

All the information on how to edit a specific item is contained in the editor, which is provided by the item itself. The graphical representation of the editor is also created by the editor itself. Now you can introduce new types of items without having to change PIM.

Applicability

Use Factory Method pattern when:

You want to create an extensible framework. This means allowing flexibility by leaving some decisions, like the specific kind of object to create, until later.

You want a subclass, rather than its superclass, to decide what kind of an object to create.

You know when to create an object, but not what kind of an object.

You need several overloaded constructors with the same parameter list, which is not allowed in Java. Instead, use several Factory Methods with different names.

Description

This pattern is called Factory Method because it creates (manufactures) objects when you want it to.

When you start writing an application, it's often not clear yet what kind of components you will be using. Normally you will have a general idea of the operations certain components should have, but the implementation is done at some other time and will not be of consequence at that moment.

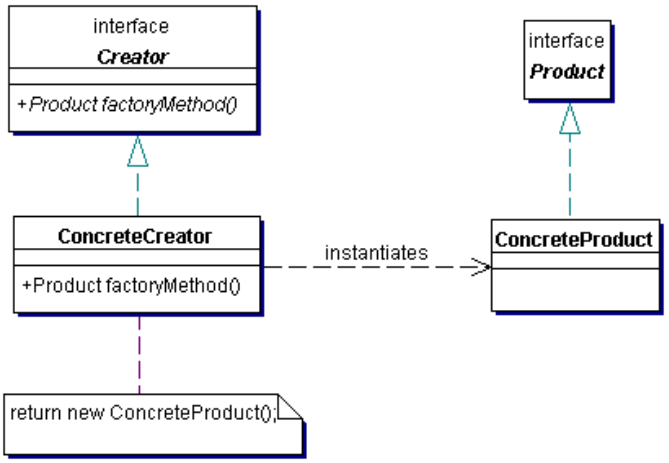
This flexibility can be achieved by using interfaces for these components. But the problem with programming to interfaces is that you cannot create an object from an interface. You need an implementing class to get an object. Instead of coding a specific implementing class in your application, you extract the functionality of the constructor and put it in a method. That method is the factory method.

To create these objects, instead of coding a specific implementing class in your application, you extract the functionality of the constructor and put it in a method. This produces a `ConcreteCreator` whose responsibility it is to create the proper objects. That `ConcreteCreator` creates instances of an implementation (`ConcreteProduct`) of an interface (`Product`).

Implementation

The class diagram is shown in Figure 1.3.

Figure 1.3. Factory Method class diagram



To implement the Factory Method you need:

`Product` – The interface of objects created by the factory.

`ConcreteProduct` – The implementing class of `Product`. Objects of this class are created by the `ConcreteCreator`.

`Creator` – The interface that defines the factory method(s) (`factoryMethod`).

`ConcreteCreator` – The class that extends `Creator` and that provides an implementation for the `factoryMethod`. This can return any object that implements the `Product` interface.

Benefits and Drawbacks

A major benefit to this solution is that the PIM can be very generic. It only needs to know how to request an editor for an item. The information about how to edit a specific item is contained in the editor. The editor can also create the graphical user interface (GUI) for editing. This makes the PIM more modular, making it easier to add new types of information to be managed without changing the core program itself.

JDBC (Java database connectivity) uses the Factory Method pattern in many of its interfaces. You can use another JDBC driver as long as the correct driver is loaded. The rest of your application remains the same. (For more information on patterns in JDBC, see “ JDBC ” on page 308.)

The drawback to this pattern is the fact that to add a new type of product, you must add another new implementing class, and you must either change an existing `ConcreteCreator` or create a new class that implements `Product`.

Pattern Variants

There are several variations for this pattern:

`Creator` can provide a standard implementation for the factory method. That way `Creator` doesn't have to be an abstract class or interface, but can be a full-blown class. The benefit is that you aren't required to subclass the `Creator`.

`Product` can be implemented as an abstract class. Because the `Product` is a class, you can add implementations for other methods.

The factory method can take a parameter. It can then create more than one type of `Product` based on the given parameter. This decreases the number of factory methods needed.

Related Patterns

Related patterns include the following:

Abstract Factory (page 6) – Might use one or more factory methods.

Prototype (page 28) – Prevents subclassing of `Creator`.

Template Method (page 131) – Template methods usually call factory methods.

Data Access Object [CJ2EEP] – The Data Access Object pattern uses the Factory Method pattern to be able to create specific instances of Data Access Objects without requiring knowledge of the specific underlying database.

Example

Note:

For a full working example of this code example, with additional supporting classes and/or a `RunPattern` class, see " Factory Method " on page 352 of the " Full Code Examples " appendix.

The following example uses the Factory Method pattern to produce an editor for the PIM. The PIM tracks a lot of information, and there are many cases where users need an editor to create or modify data. The example uses interfaces to improve the overall flexibility of the system.

The `Editable` interface defines a builder method, `getEditor`, which returns an `ItemEditor` interface. The benefit is that any item can provide an editor for itself, producing an object that knows what parts of a business object can change and how they can be changed. The only thing the user interface needs to do is use the `Editable` interface to get an editor.

Example 1.14 `Editable.java`

```
1.    public interface Editable {
2.        public ItemEditor getEditor();
3.    }
```

The `ItemEditor` interface provides two methods: `getGUI` and `commitChanges`. The `getGUI` method is another Factory Method—it returns a `JComponent` that provides a Swing GUI to edit the current item. This makes a very flexible system; to add a new type of item, the user interface can remain the same, because it only uses the `Editable` and the `ItemEditor` interfaces.

The `JComponent` returned by `getGUI` can have anything in it required to edit the item in the PIM. The user interface can simply the acquired `JComponent` in its editor window and use the `JComponent` functionality to edit the item. Since not everything in an application needs to be graphical, it could also be a good idea to include a `getUI` method that would return an `Object` or some other non-graphical interface.

The second method, `commitChanges`, allows the UI to tell the editor that the user wants to finalize the changes he or she has made.

Example 1.15 `ItemEditor.java`

```
1.    import javax.swing.JComponent;
2.    public interface ItemEditor {
3.        public JComponent getGUI();
4.        public void commitChanges();
5.    }
```

The following code shows the implementation for one of the PIM items, `Contact`. The `Contact` class defines two attributes: the name of the person and their relationship with the user. These attributes provide a sample of some of the information, which could be included in an entry in the PIM.

Example 1.16 `Contact.java`

```
1.    import java.awt.GridLayout;
2.    import java.io.Serializable;
3.    import javax.swing.JComponent;
4.    import javax.swing.JLabel;
5.    import javax.swing.JPanel;
6.    import javax.swing.JTextField;
7.
8.    public class Contact implements Editable, Serializable {
9.       private String name;
10.      private String relationship;
11.
12.      public ItemEditor getEditor() {
13.         return new ContactEditor();
14.      }
15.
16.      private class ContactEditor implements ItemEditor, Serializable {
17.         private transient JPanel panel;
18.         private transient JTextField nameField;
19.         private transient JTextField relationField;
20.
21.         public JComponent getGUI() {
22.            if (panel == null) {
23.               panel = new JPanel();
24.               nameField = new JTextField(name);
25.               relationField = new JTextField(relationship);
26.               panel.setLayout(new GridLayout(2,2));
27.               panel.add(new JLabel("Name:"));
28.               panel.add(nameField);
29.               panel.add(new JLabel("Relationship:"));
30.               panel.add(relationField);
31.            } else {
32.               nameField.setText(name);
33.               relationField.setText(relationship);
34.            }
35.            return panel;
36.         }
37.
38.         public void commitChanges() {
39.            if (panel != null) {
40.               name = nameField.getText();
41.               relationship = relationField.getText();
42.            }
43.         }
44.
45.         public String toString(){
46.            return "\nContact:\n" +
47.               "    Name: " + name + "\n" +
48.               "    Relationship: " + relationship;
49.         }
50.      }
51.   }
```

`Contact` implements the `Editable` interface, and provides its own editor. That editor only applies to the `Contact` class, and needs to change certain attributes of the `Contact`, it is best to use an inner class. The inner class has direct access to the attributes of the outer class. If you used another (non-inner) class, `Contact` would need to provide accessor and mutator methods, making it harder to restrict access to the object's private data.

Note that the editor itself is not a Swing component, but only an object that can serve as a factory for such a component. The greatest benefit is that you can serialize and send this object across a stream. To implement this feature, declare all Swing component attributes in `ContactEditor` transient—they're constructed when and where they're needed.

Prototype

Pattern Properties

Type: Creational, Object

Level: Single Class

Purpose

To make dynamic creation easier by defining classes whose objects can create duplicates of themselves.

Introduction

In the PIM, you want to be able to copy an address entry so that the user doesn't have to manually enter all the information when creating a new contact. One way to solve this is to perform the following steps:

Create a new `Address` object.

Copy the appropriate values from the existing `Address`.

While this approach solves the problem, it has one serious drawback—it violates the object-oriented principle of encapsulation. To achieve the solution mentioned above, you have to put method calls to copy the `Address` information, outside of the `Address` class. This means that it becomes harder and harder to maintain the `Address` code, since it exists throughout the code for the project. It is also difficult to reuse the `Address` class in some new project in the future.

The copy code really belongs in the `Address` class itself, so why not instead define a "copy" method in the class? This method produces a duplicate of the `Address` object with the same data as the original object—the prototype. Calling the method on an existing `Address` object solves the problem in a much more maintainable way, much truer to good object-oriented coding practices.

Applicability

Use the Prototype pattern when you want to create an object that is a copy of an existing object.

Description

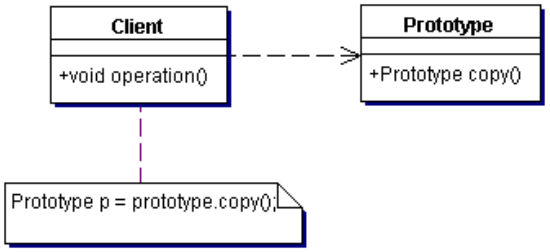
The Prototype pattern is well named; as with other prototypes, it has an object that is used as the basis to create a new instance with the same values. Providing a "create based on existing state " behavior allows programs to perform operations like user-driven copy, and to initialize objects to a state that has been established through use of the system. This is often preferable to initializing the object to some generic set of values.

Classic examples for this pattern exist in graphic and text editors, where copy-paste features can greatly improve user productivity. Some business systems use this approach as well, producing an initial model from an existing business object. The copy can then be modified to its desired new state.

Implementation

The Prototype class diagram is shown in Figure 1.4.

Figure 1.4. Prototype class diagram



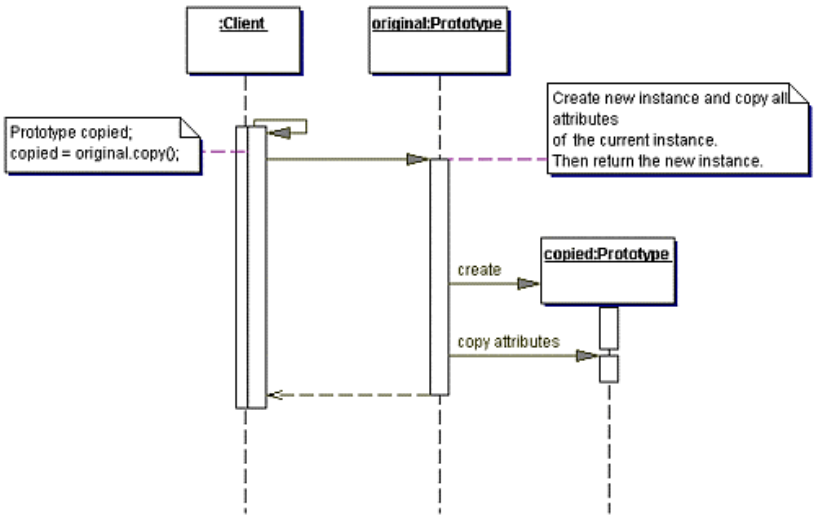
To implement Prototype, you need:

`Prototype` – Provides a copy method. That method returns an instance of the same class with the same values as the original `Prototype` instance. The new instance can be a deep or shallow copy of the original (see the Benefits and Drawbacks section of this pattern).

Benefits and Drawbacks

The Prototype is helpful because it allows systems to produce a copy of a usable object, with variables already set to a (presumably) meaningful value, rather than depending on some base state defined in the constructor. An example of Prototype use is shown in Figure 1.5.

Figure 1.5. Example of Prototype use



A key consideration for this pattern is copy depth.

A shallow copy duplicates only the top-level elements of a class; this provides a faster copy, but isn't suitable for all needs. Since references are copied from the original to the copy, they still refer to the same objects. The lower-level objects are shared among copies of the object, so changing one of these objects affects all of the copies.

Deep copy operations replicate not only the top-level attributes, but also the lower-level objects. This typically takes longer than shallow copy operations, and can be very costly for objects with an arbitrarily complex structure. This makes sure that changes in one copy are isolated from other copies.

By its nature, the clone method in `Object` supports only one form of copy. For cases where you must support multiple methods of post-creation initialization.

Pattern Variants

Pattern variants include the following:

Copy constructor – One variant of the prototype is a copy constructor. A copy constructor takes an instance of the same class as an argument and returns a new copy with the same values as the argument.

Example 1.17 `Copy constructor`

```
public class Prototype {
  private int someData;
  // some more data
  public Prototype(Prototype original) {
    super();
    this.someData = original.someData;
    //copy the rest of the data
  }
  // rest of code
}
```

An example is the `String` class, where you can create a new String instance by calling for instance: `new String("text");`

The benefit of this variant is that the intention of creating a new instance is very clear, but only one type of copy (deep or shallow) can be executed. It is possible to have a constructor that can use both. The constructor would take two arguments: the object to be copied and a boolean to mark whether it should apply a deep or shallow copy.

A drawback is that the copy constructor must check the incoming reference to see if it is not null. With the normal Prototype implementation, the method is certain to be called on a valid object.

`clone` method – The Java programming language already defines a clone method in the `java.lang.Object` class—the superclass of all Java classes. For the method to be usable on an instance, the class of that object has to implement the `java.lang.Clonable` interface to indicate that an instance of this class may be copied. Because the `clone` method is declared protected in `Object`, it has to be overridden to make it publicly available.

According to Bloch, "clone() should be used judiciously" [Bloch01]. As mentioned, a class has to implement `Clonable`, but that interface does not provide a guarantee that the object can be cloned. The `Clonable` interface does not defined the `clone` method, so it is possible that the `clone` method is not available when it is not overridden. Another drawback of the `clone` method is that it has a return type of `Object`, requiring you to cast it to the appropriate type before using it.

Related Patterns

Related patterns include the following:

Abstract Factory (page 6) – Abstract Factories can use the Prototype to create new objects based on the current use of the Factory.

Factory Method (page 21) – Factory Methods can use a Prototype to act as a template for new objects.

Example

Note:

For a full working example of this code example, with additional supporting classes and/or a `RunPattern` class, see " Prototype " on page 357 of the " Full Code Examples " appendix.

The `Address` class in this example uses the Prototype pattern to create an address based on an existing entry. The core functionality for the pattern is defined in the interface `Copyable`.

Example 1.18 `Copyable.java`

```
1.    public interface Copyable{
2.       public Object copy();
3.    }
```

The `Copyable` interface defines a copy method and guarantees that any classes that implement the interface will define a copy operation. This example produces a shallow copy—that is, it copies the object references from the original address to the duplicate.

The code also demonstrates an important feature of the copy operation: not all fields must necessarily be duplicated. In this case, the address type is not copied to the new object. A user would manually specify a new address type from the PIM user interface.

Example 1.19 `Address.java`

```
1.    public class Address implements Copyable{
2.       private String type;
3.       private String street;
4.       private String city;
5.       private String state;
6.       private String zipCode;
7.       public static final String EOL_STRING =
8.          System.getProperty("line.separator");
9.       public static final String COMMA = ",";
10.      public static final String HOME = "home";
11.      public static final String WORK = "work";
```

```
12.
13.      public Address(String initType, String initStreet,
14.         String initCity, String initState, String initZip){
15.            type = initType;
16.            street = initStreet;
17.            city = initCity;
18.            state = initState;
19.            zipCode = initZip;
20.      }
21.
22.      public Address(String initStreet, String initCity,
23.         String initState, String initZip){
24.            this(WORK, initStreet, initCity, initState, initZip);
25.      }
26.      public Address(String initType){
27.         type = initType;
28.      }
29.      public Address(){ }
30.
31.      public String getType(){ return type; }
32.      public String getStreet(){ return street; }
33.      public String getCity(){ return city; }
34.      public String getState(){ return state; }
35.      public String getZipCode(){ return zipCode; }
36.
37.      public void setType(String newType){ type = newType; }
38.      public void setStreet(String newStreet){ street = newStreet; }
39.      public void setCity(String newCity){ city = newCity; }
40.      public void setState(String newState){ state = newState; }
41.      public void setZipCode(String newZip){ zipCode = newZip; }
42.
43.      public Object copy(){
44.         return new Address(street, city, state, zipCode);
45.      }
46.
47.      public String toString(){
48.         return "\t" + street + COMMA + " " + EOL_STRING +
49.            "\t" + city + COMMA + " " + state + " " + zipCode;
50.      }
51.   }
```

Singleton

Pattern Properties

Type: Creational

Level: Object

Purpose

To have only one instance of this class in the system, while allowing other classes to get access to this instance.

Introduction

Once in a while, you need a global object: one that's accessible from anywhere but which should be created only once. You want all parts of the application to be able to use the object, but they all should use the same instance.

An example is a history list—a list of actions a user has taken while using the application. Multiple parts of the application use the same `HistoryList` object to either add actions a user has taken or to undo previous actions.

One way to achieve this is to have the main application create a global object, then pass its reference to every object that might ever need it. However, it can be very difficult to determine how you want to pass the reference, and to know up front which parts of the application need to use the object. Another drawback to this solution is that it doesn't prevent another object from creating another instance of the global object—in this case, `HistoryList`.

Another way to create global values is by using static variables. The application has several static objects inside of a class and accesses them directly.

This approach has several drawbacks.

A static object will not suffice because a static object will be created at the time the class loads and thus gives you no opportunity to supply any data before it instantiates.

You have no control over who accesses the object. Anybody can access a publicly available static instance.

If you realize that the singleton should be, say, a trinity, you're faced with modifying every piece of client code.

This is where the Singleton pattern comes in handy. It provides easy access for the whole application to the global object.

Applicability

Use the Singleton when you want only one instance of a class, but it should be available everywhere.

Description

The Singleton ensures a maximum of one instance is created by the JVM (not surprisingly, that's why it's called a singleton). To ensure you have control over the instantiation, make the constructor private.

This poses a problem: it's impossible to create an instance, so an accessor method is provided by a static method (`getInstance`()). That method creates the single instance, if it doesn't already exist, and returns the reference of the singleton to the caller of the method. The reference to the singleton is also stored as a static private attribute of the singleton class for future calls.

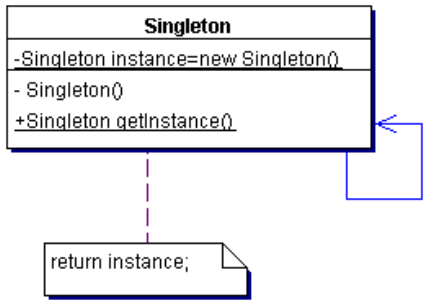
Although the accessor method can create the singleton, most of the times it is created as the class is loaded. Postponing the construction is only necessary if some form of initialization has to be done before the singleton is instantiated.

An example of a singleton is the president of the United States of America. At any given time there should only be one president. When the president of Russia picks up the red phone, he expects to get a handle to the current United States president.

Implementation

The Singleton class diagram is shown in Figure 1.6.

Figure 1.6. Singleton class diagram



To implement the Singleton pattern, you need:

`Singleton` — Provides a private constructor, maintains a private static reference to the single instance of this class, and provides a static accessor method to return a reference to the single instance.

The rest of the implementation of the Singleton class is normal. The static accessor method can make decisions about what kind of an instance to create, based on system properties or parameters passed into the accessor method (see the Pattern Variants section for this pattern).

Benefits and Drawbacks

Benefits and drawbacks include the following:

The Singleton is the only class that can create an instance of itself. You can't create one without using the static method provided.

You don't need to pass the reference to all objects needing this Singleton.

However, the Singleton pattern can present threading problems, depending upon the implementation. You must take care regarding control of the singleton initialization in a multithreaded application. Without the proper control, your application will get into "thread wars."

Pattern Variants

Pattern variants include the following:

One of the Singleton's often-overlooked options is having more than one instance inside the class. The benefit is that the rest of the application can remain the same, while those that are aware of these multiple instances can use other methods to get other instances.

The Singleton's accessor method can be the entry point to a whole set of instances, all of a different subtype. The accessor method can determine at runtime what specific subtype instance to return. This might seem odd, but it's very useful when you're using dynamic class loading. The system using the Singleton can remain unchanged, while the specific implementation of the Singleton can be different.

Related Patterns

Related patterns include the following:

Abstract Factory (page 6)

Builder (page 13)

Prototype (page 28)

Example

Application users want the option of undoing previous commands. To support that functionality, a history list is needed. That history list has to be accessible from everywhere in the PIM and only one instance of it is needed. Therefore, it's a perfect candidate for the Singleton pattern.

Example 1.20 `HistoryList.java`

```
1.    import java.util.ArrayList;
2.    import java.util.Collections;
3.    import java.util.List;
4.    public class HistoryList{
5.       private List history = Collections.synchronizedList(new ArrayList());
6.       private static HistoryList instance = new HistoryList();
7.
8.       private HistoryList(){ }
9.
10.      public static HistoryList getInstance(){
11.         return instance;
12.      }
13.
14.      public void addCommand(String command){
15.         history.add(command);
16.      }
17.
18.      public Object undoCommand(){
19.         return history.remove(history.size() - 1);
20.      }
21.
22.      public String toString(){
23.         StringBuffer result = new StringBuffer();
24.         for (int i = 0; i < history.size(); i++){
25.            result.append(" ");
26.            result.append(history.get(i));
27.            result.append("\n");
28.         }
29.         return result.toString();
30.      }
31.   }
```

The `HistoryList` maintains a static reference to an instance of itself, has a private constructor, and uses a static method `getInstance` to provide a single history list object to all parts of the PIM. The additional variable in `HistoryList`, `history`, is a `List` object used to track the command strings. The `HistoryList` provides two methods, `addCommand` and `undoCommand` to support adding and removing commands from the list.