Object-Oriented Analysis and Design with UML
Rational Software

Supplement – Design Patterns

1. Introduction

In the excellent book "Design Patterns, Elements of Reusable Object-Oriented Software" [1], design patterns are defined as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

This document presents the following design patterns:

- Command
- Proxy
- Singleton
- Factory
- Observer
- Mediator

2. The Command pattern

This is a very basic and simple pattern, which constitutes an excellent summary of the analysis section of the course as it involves class diagrams describing the class structure and packaging of a simple reusable GUI interface and interaction diagrams describing the dynamic behavior of the system.

The problem: imagine we want to build a reusable GUI component. To keep it simple, we will limit ourselves to the implementation of generic menus in a windowing system (in such a way that it will be possible to add new menus without having to modify the GUI component).

Let's start with the class structure:

- The class *Application* is the client class, it "simulates" the application.
- The class *Menu*: to simplify, we will make the assumption that our "application" has only one menu represented by an association with a multiplicity of 1 and a role name *menu*
- A menu is composed of menu items: this is represented by the association between *Menu* and *MenuItem*. *MenuItem* has an attribute *label* (the text of the menu item as it is displayed to the user).
- The key element of the pattern is the class *Command*. It is an abstract class. It has one operation called *Process*. This operation does not do anything: we will represent

[1] Authors: Erich Gamma, Richard Helm, Ralph Johnsson, John Vlissides; publisher: Addison-Wesley; ISBN 0-201-63361-2

implementation code by using a Java-like pseudo-code. The implementation of *Process* is represented by a single statement:

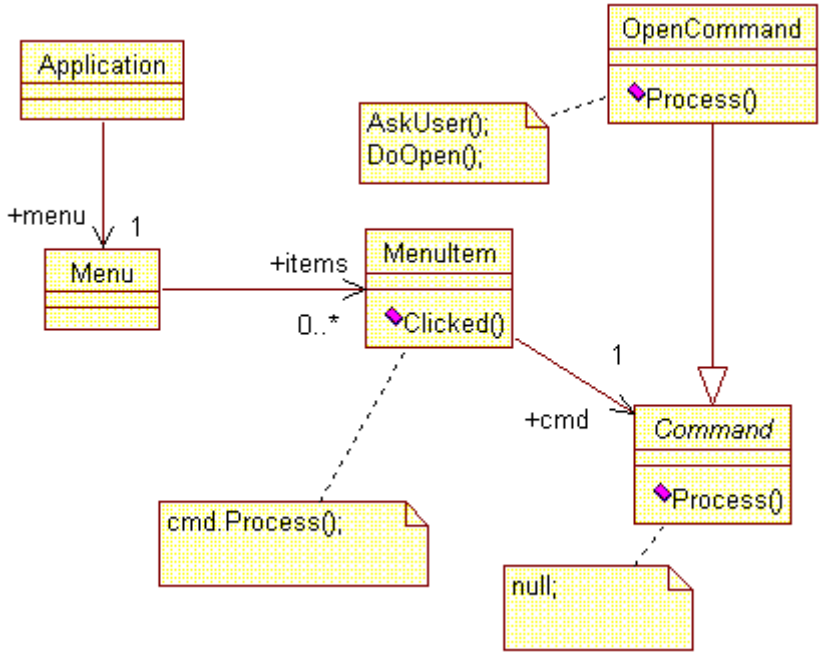
```
null;
```

- We will now make the assumption that the underlying windowing system requires that we define in *MenuItem* an operation called *Clicked* that will be automatically invoked when the (physical) user selects the corresponding menu during execution time. The code for *Clicked* is:

```
cmd.Process();
```

- Imagine now that we want to implement the menu command *Open*…. Let's create a new class called *OpenCommand* that inherits from *Command*. This class overrides the operation *Process* to prompt the user for the file to open and to open it:

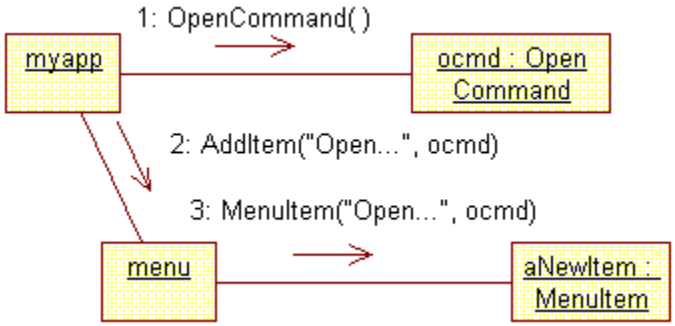
```
AskUser();
DoOpen();
```



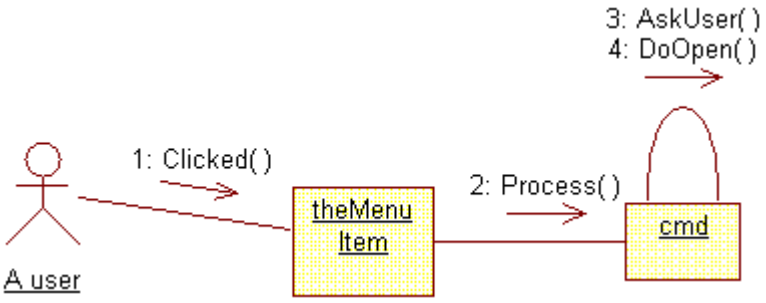
- Let's now look at the dynamic behavior of the system: first, let's create an interaction diagram showing the initialization of the system. When the system starts up, the objects *myapp:Application* and *menu:Menu* are created. To implement the *Open*… menu item, *myapp* first creates the object *ocmd:OpenCommand* (message 1). It then invokes a new operation from *menu* called *AddItem* that takes 2 arguments: *s* of type *String* (the label of the menu item to create) and *c* of type *Command*. Note that *myapp* passes to *AddItem* a subclass of *Command* (message 2).
- *menu* creates a new menu item (message 3). The arguments of the constructor for *MenuItem* are the same as *AddItem*. The code of the constructor is straightforward:

```
label = s;
cmd = c;
```

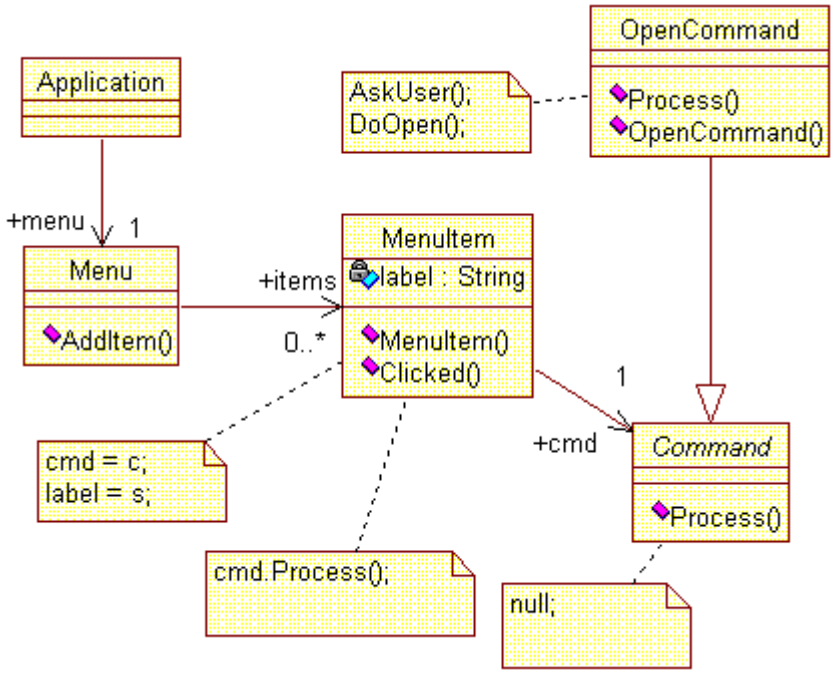
- It is very important to note that *cmd* is initialized with a subclass of *Command* but that *MenuItem* "thinks" it is a *Command* object! This is all the magic of polymorphism!



- Now let's draw a second interaction diagram that will show what happens when the user selects the menu item *Open…*: the *Clicked* operation is invoked (message 1 below). *Clicked* simply executes the *Process* operation of the associated *cmd* object. In our case, the *cmd* object is actually the *OpenCommand* object (although this is transparent to the *MenuItem* object).

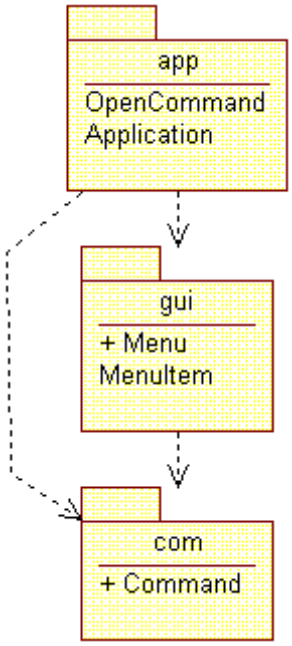


- Looking back at our class diagram, we should now have something like this: (the main difference with the previous class diagram are the operations)



- The class diagram is actually incomplete: there should be a Uses relationship from *Application* to *OpenCommand* to account for the fact that the former creates the latter, and there should be another Uses relationship from *Menu* to *Command* as the *AddItem* operation takes one argument of type *Command*.

- So what about reuse? Let's assign the classes to packages: *OpenCommand* and *Application* to *app*, *Menu* and *MenuItem* to *gui* and *Command* to *com*. Then let's add the appropriate dependencies. The result is shown below: We have created a reusable set of components (*gui* and *com*) independent of the application packages using them! Also note that *MenuItem* can be made an Implementation class only! Only *Menu* is exported.



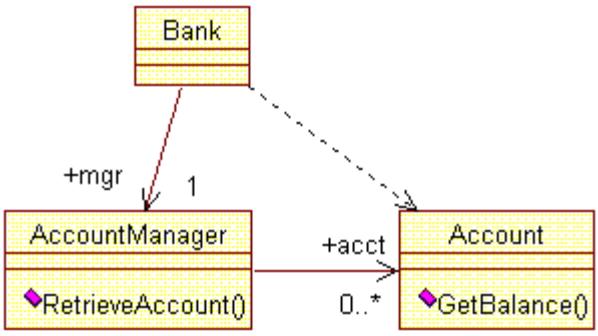
Note: The Command pattern is the OO equivalent of a "callback" procedure. It can be used in many other cases, for instance, to implement timeout processing.

3. The Proxy pattern

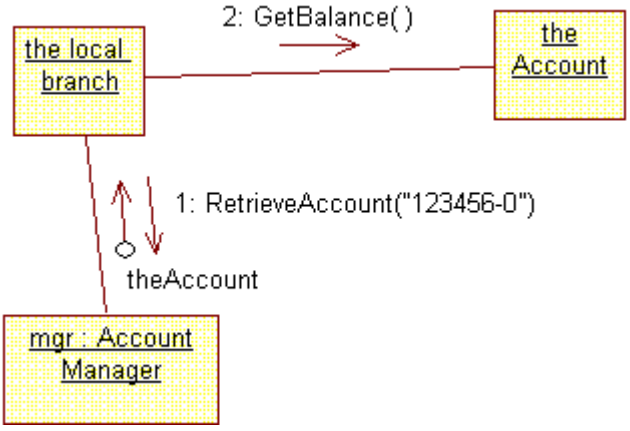
This second pattern is used to demonstrate:
1) How to handle distributed objects in a way that is transparent to the local client application.
2) How a design model can be derived from an analysis "by adding elements rather than by replacing them". [2]

- Imagine that we are implementing a software to allow branches from a bank to retrieve any account information such as the account balance. To simplify we will assume that an account is identified by a unique account number (in other words, it is not required to know the owner of the account). Our model could be expressed as follows:

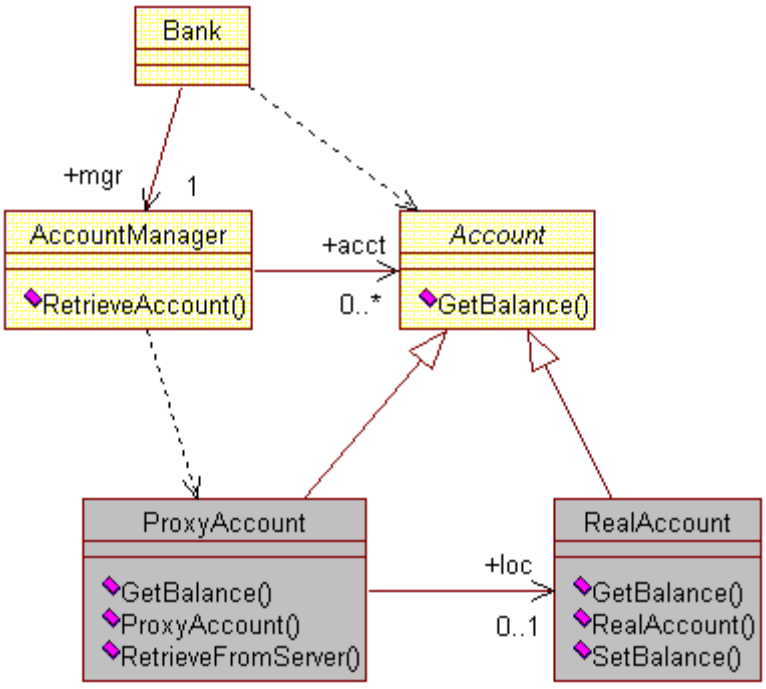
[2] Jim Rumbaugh, OMT Insights, p.1: "In Layered Additive Models I argue that the goal of a model is to capture design decisions as directly as possible, and the best way to do this is to evolve the model by adding elements rather than by replacing them."



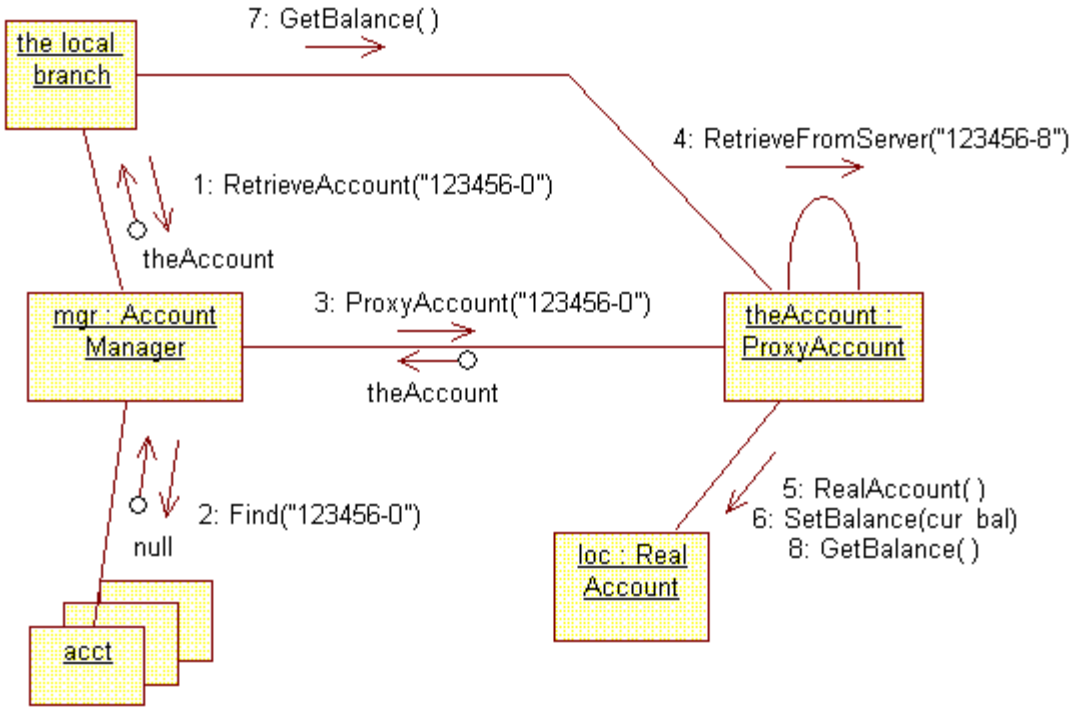
- Retrieving the balance of an account whose number is 123456-0 is a no-brainer:



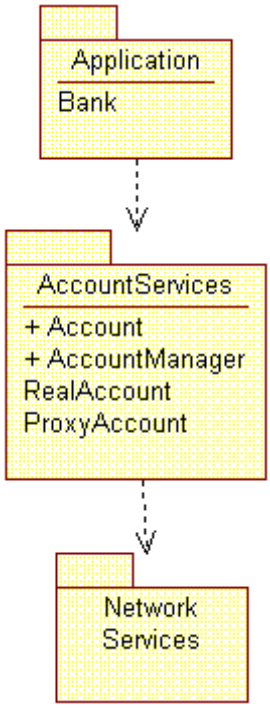
- The model above was developed independently of the notion of having distributed account objects. The use of the proxy pattern on the *Account* class will allow us to handle distributed account objects without having to change our model (and code) above. For that purpose, let's add two new classes *ProxyAccount* and *RealAccount*, both derived from *Account* that becomes in the process an abstract class. In addition, *ProxyAccount* has a zero-or-one association with *RealAccount*.



- To understand how the pattern works, let's go back to the interaction diagram and let's expand the processing implied in the *RetrieveAccount* and *GetBalance* operations.
- *RetrieveAccount* operation (message 1): the account manager first verifies whether or not the account object is in the locally maintained account list (message 2). In this scenario, we will assume that the account object does not exist locally. It therefore needs to be retrieved from the bank server. For that purpose, the account manager creates a *ProxyAccount* object (message 3). The proxy object in its initialization procedure calls the *RetrieveFromServer* operation (message 4). This operation uses some kind of network services that we have not represented here (see note 2 below). Once the info has been successfully retrieved from the server, *RetrieveFromServer* creates a local copy of the object (message 5) and populates it with the retrieved data (message 6). **The object returned to the caller** (return value of message 1) **is the proxy itself and not the local copy**. The reason for this is explained later. The "local branch" can now use this proxy object to retrieve the actual balance (message 7). The proxy then passes the request to the local copy (message 8). Of course the local branch is not aware of the fact that the object it is exchanging data with is not the original object but only an intermediate.



- The proxy object acts as a relay between the local branch and the account. If the original account object is modified (e.g. if the balance is changed), the proxy object will be notified by the server to "clear" its local copy (resetting the association to 0). (This notification can be performed very efficiently by using another pattern, called the Observer.) If a further request to get the balance is made by the local branch, the proxy will then call *RetrieveFromServer* again to get a new copy from the server.
- The account classes and the account manager are placed in an *AccountServices* package. As the diagram below shows, *ProxyAccount* and *RealAccount* are implementation classes only and cannot be used by the client application. This is essential for reusability and maintenance reasons. Also note that the *Application* package is independent of the *NetworkServices* package.



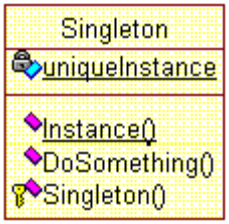
Notes:

1) It is important to note that *RetrieveAccount* returns an object of type *Account*. Of course, the actual object is always an instance of the subclass *ProxyAccount* but the clients of *Account* do not know that (and don't need to).
2) The proxy pattern has the added benefit of isolating the client application from the technology used to actually implemented the distributed objects (i.e. the implementation of *RetrieveFromServer*): COM, CORBA, Java RMI, etc.
3) We have used here the proxy pattern to implement a "remote" proxy. It can also be used as a virtual proxy (e.g. to load from disk "expensive" objects on demand), a protective proxy (to control access to the original object) or a "smart reference".
4) During a session, there can be only one instance of the *AccountManager*. *AccountManager* is called a singleton. The singleton pattern is explained in below.

4. The Singleton pattern

This is a very simple pattern that makes it possible to guarantee that at most one instance of a given class will be created.

- The structure of a singleton is represented below. The static variable (or class variable) *uniqueInstance* is the only instance of *Singleton* that can exist in the system. *Instance* is an operation that will allow users to get access to *uniqueInstance*.



- Now let's take a look at the code of *Instance* (using some form of pseudo-code):

```
Public Instance (): Singleton {
      If uniqueInstance = 0 Then
            uniqueInstance = new Singleton()
      Endif
      return uniqueInstance
}
```

- The first time *Instance* is called, it creates the instance of *Singleton* and it returns it to the caller. After that, every call to *Instance* will return *uniqueInstance*. This is of course completely transparent to the caller. To prevent the creation of unwanted instances of *Singleton*, we have made its constructor protected. To finish, let's take a look at a sample code using the singleton to "do something":

```
Singleton.Instance().DoSomething()
Singleton s = Singleton.Instance()
s.DoSomething()
```

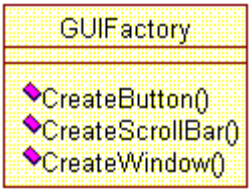
Note: Another way to implement a singleton's functionality is to use class operations (like *DoSomething*). The singleton pattern however provides more flexibility: for instance to allow for more than one instance or to take full advantage of polymorphism (particularly in the case of C++ where static operations are never virtual and cannot be overridden by subclasses).

5. The Factory pattern

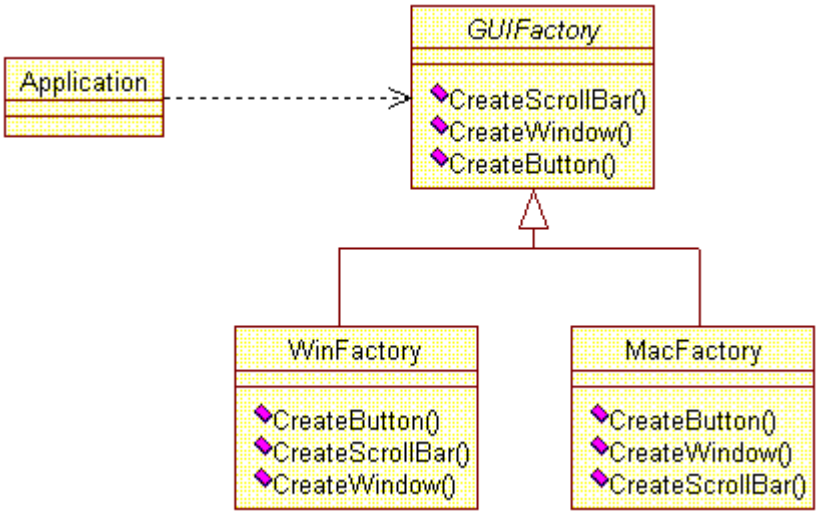
Consider a system that needs to support Windows, MacOS and possibly others OS. If we want to create a push button, a scrollbar or a window, each object needing this functionality would have to write something like:

```
Button b = new WinButton()
or
Button b = new MacButton()
```

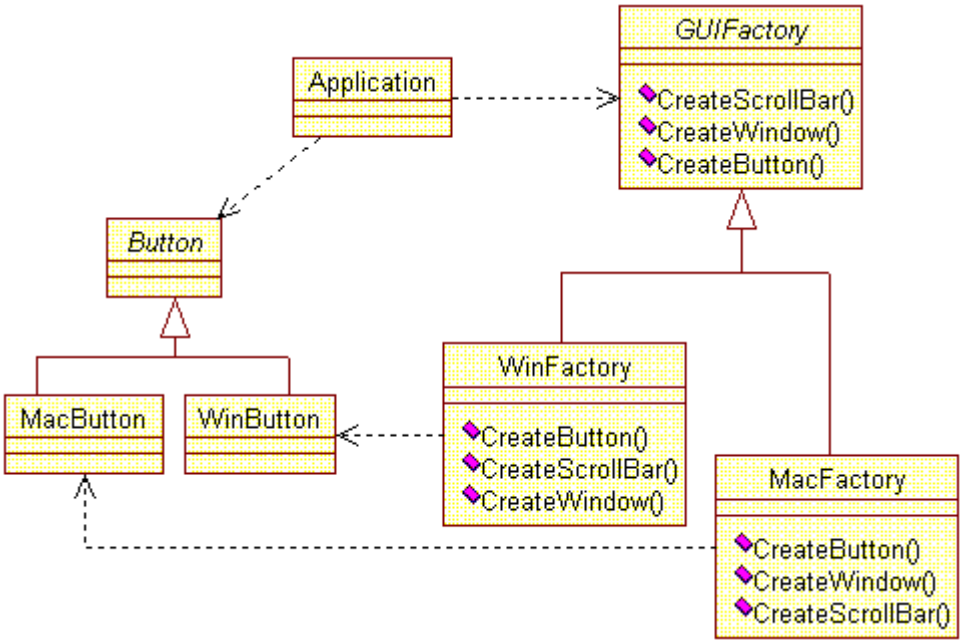
This has an obvious inconvenient: if we want to port our system from a Windows environment to a Macintosh, it would be necessary to identify every line of the types above and replace them appropriately. A solution is to regroup the creation of the buttons, scrollbars, windows, etc., in a single class that we will call here *GUIFactory*:



To handle the different operating systems, let's subclass *GUIFactory* (which becomes an abstract class) for every possible case:



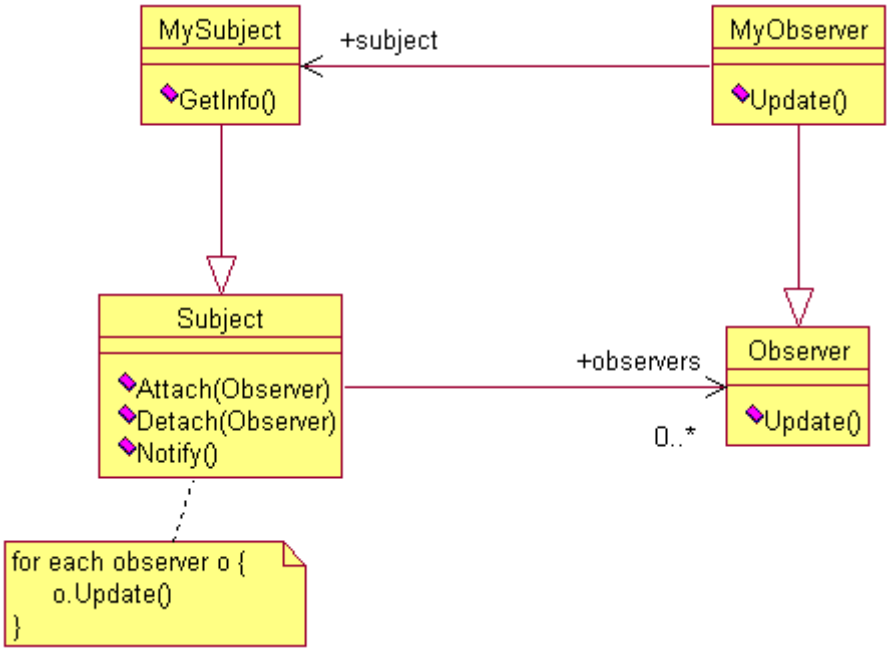
Users of the GUI only need to be aware of *GUIFactory* and therefore will not need to change when new OS have to be supported. To be completely independent of the OS, it is also necessary to create abstract classes for buttons, windows, scrollbars, etc. *Create* functions return objects of the type corresponding to the abstract class (*Button* for *CreateButton*).



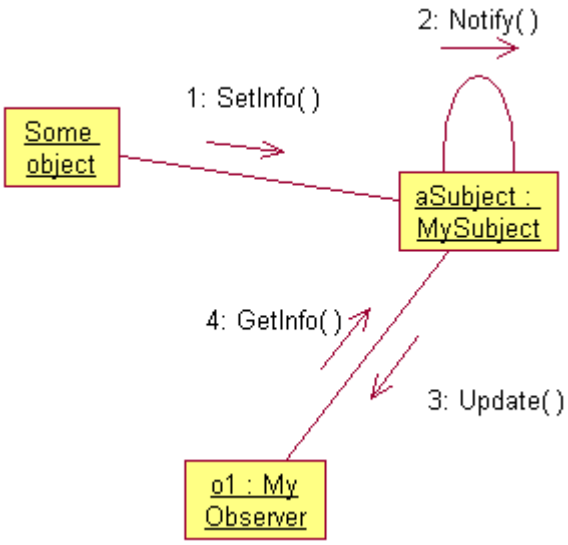
6. The Observer pattern

Consider several graphical windows presenting the same information, possibly in different ways (e.g. spreadsheet table, pie chart, bar chart). When the information changes, all windows must be updated to reflect the changes. Obviously the object responsible for maintaining the information should not be aware of the graphical (or other) objects using this information.

In order to have the "observed" object independent of the observers, we will use the Observer pattern (equivalent to publish-subscribe). A subject object (the "observed" object) notifies upon change anonymous observers as shown in the diagram below.



When a subject is updated, it will notify its observers, which, in turn, can respond to the update by whatever action is required, like retrieving the info of the observed object. The main interest of the pattern is that the observed object is not required to know about the observers ahead of time.

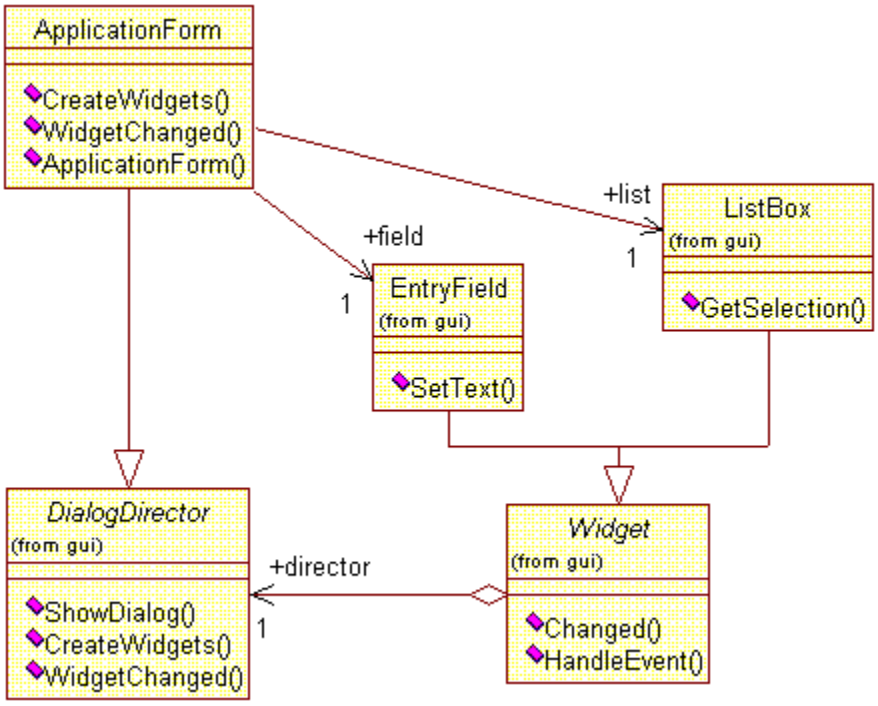


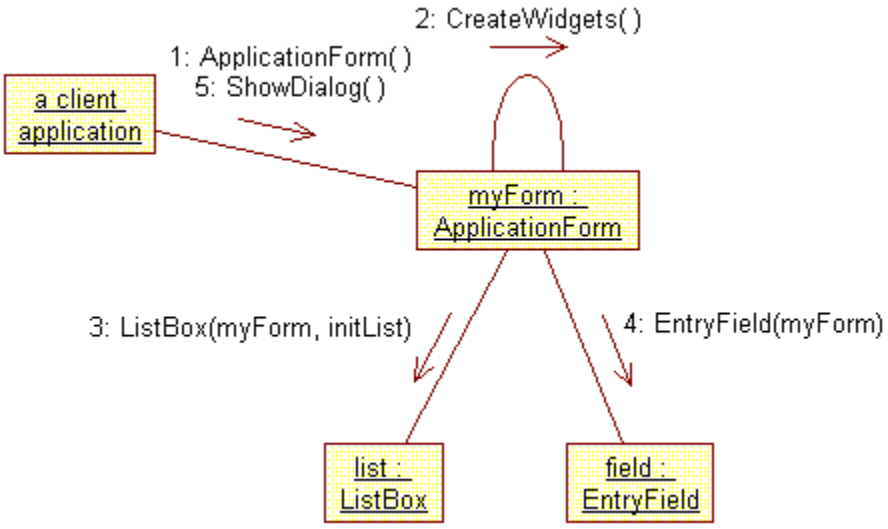
7. The Mediator pattern

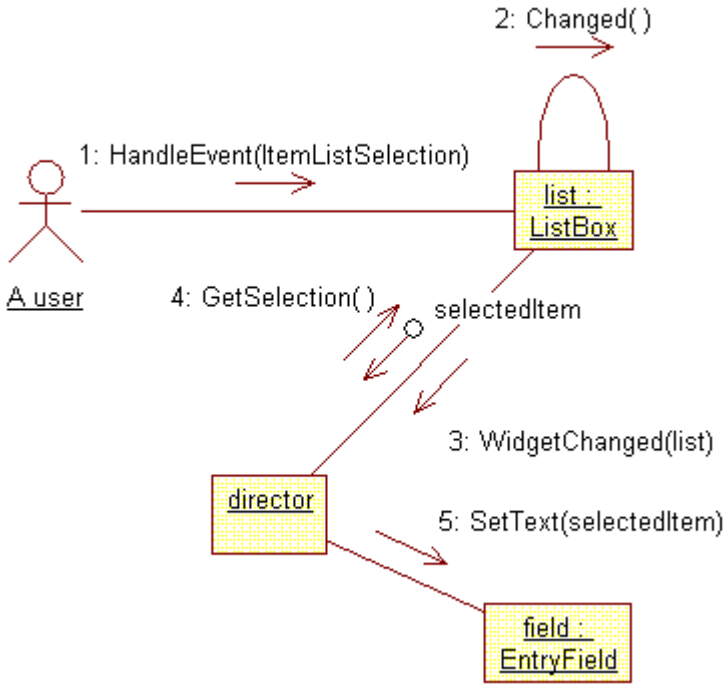
Consider the implementation of a dialog box in a graphical user interface. The dialog box uses a window to present a collection of widgets such as buttons, menus, entry fields, etc. The widgets often have dependencies between themselves. For instance, selecting an entry in a list box might

change the contents of an entry field.

In order to use generic widget objects, it becomes necessary to encapsulate their collective behavior in a separate mediator object. (In the example above, the list box must not know that the entry field must be updated.)






2: Changed( )
1: HandleEvent(ItemListSelection)
list :
ListBox
A user
4: GetSelection( )
selectedItem
3: WidgetChanged(list)
director
5: SetText(selectedItem)
field :
EntryField