

# Software Architectures

## Assignment 1 : Design Patterns

Arnaud Rosette, Simon Picard

March 5, 2015

### 1 Exercise 1 : Find Instances of Design Patterns

#### 1.1 Singleton

The `org.gjt.sp.jedit.buffer.KillRing` class is an instance of the singleton pattern.

##### 1.1.1 Purpose

Creational pattern.

##### 1.1.2 Participants<sup>1</sup>

The `KillRing` class is the singleton class.

##### 1.1.3 Class diagram

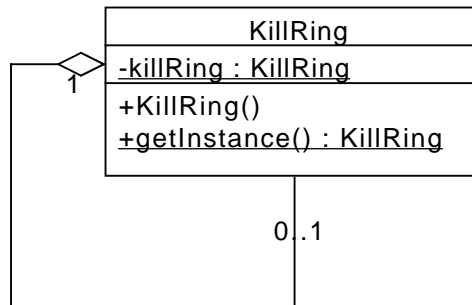


Figure 1: `KillRing` class diagram

##### 1.1.4 Concrete situation description

In this situation, the singleton pattern is used to keep track of deleted text in a single place in the application.

The constructor is here public. However, the common usage of the singleton pattern uses a private constructor in order to only have one instance of this class living in the system. The constructor is here made public because the plug-ins may want to create their own `KillRing`.

---

<sup>1</sup>The participants are those described in the book : *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, Vlissides)

## 1.2 Abstract Factory

The `org.gjt.sp.jedit.gui.statusbar.StatusWidgetFactory` interface is an example of the abstract factory pattern.

### 1.2.1 Purpose

Creational pattern.

### 1.2.2 Participants<sup>1</sup>

The participants are the classes : `org.gjt.sp.jedit.gui.statusbar.StatusWidgetFactory`, `org.gjt.sp.jedit.gui.statusbar.BufferSetWidgetFactory`, `org.gjt.sp.jedit.gui.statusbar.BufferSetWidget`, `org.gjt.sp.jedit.gui.statusbar.Widget` and `org.gjt.sp.jedit.gui.StatusBar`.

- **StatusWidgetFactory** : Abstract Factory
- **BufferSetWidgetFactory** : Concrete Factory
- **BufferSetWidget** : Concrete Product
- **Widget** : Abstract Product
- **StatusBar** : Client

### 1.2.3 Class diagram

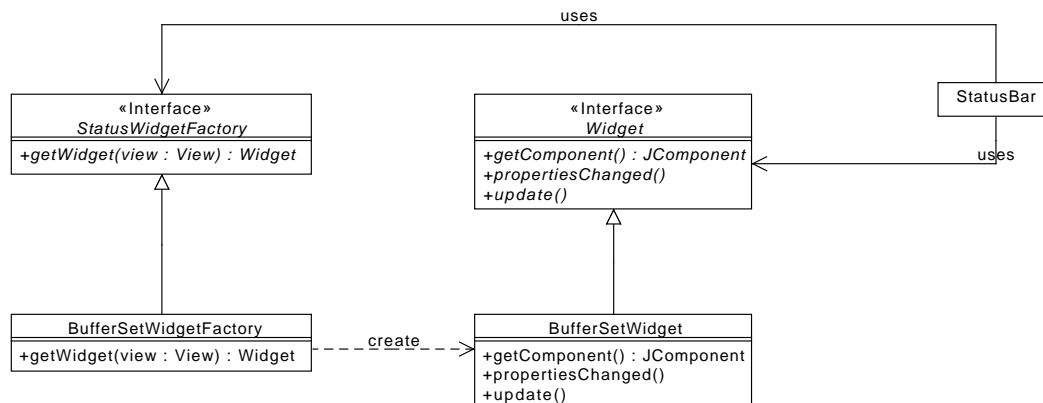


Figure 2: StatusWidgetFactory class diagram

### 1.2.4 Concrete situation description

In this situation, the abstract factory pattern is used to let a `StatusBar` object creating different kind of `Widget` without specifying their concrete class.

## 1.3 Observer

The `org.gjt.sp.jedit.buffer.BufferListener` interface is an example of the observer pattern.

### 1.3.1 Purpose

Behavioral pattern.

### 1.3.2 Participants<sup>1</sup>

The participants are the classes : `org.gjt.sp.jedit.buffer.BufferListener`, `org.gjt.sp.jedit.buffer.BufferAdapter`, `org.gjt.sp.jedit.textarea.ElasticTabStopBufferListener` and `org.gjt.sp.jedit.buffer.JEditBuffer`.

- **BufferListener** : Observer
- **BufferAdapter** : does not belong to this design pattern
- **ElasticTabStopBufferListener** : Concrete Observer
- **JEditBuffer** : Subject, Concrete Subject

### 1.3.3 Class diagram

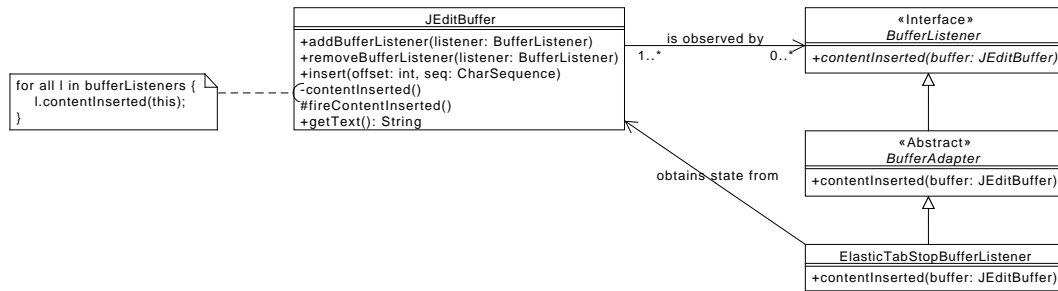


Figure 3: BufferListener class diagram

### 1.3.4 Concrete situation description

A **JEditBuffer** object represents the content of an open text file that resides in the main memory. So, when the state of a **JEditBuffer** object is modified, the objects that have to modify their own state according to the state of the **JEditBuffer** object are notified because they previously registered themselves to the **JEditBuffer** object.

In this case, the **JEditBuffer** class has two roles : subject and concrete subject. These two roles belong to different classes in the theoretical pattern.

In the class diagram, we only show methods that are relevant for the insert action.

## 1.4 Adapter

The `org.gjt.sp.jedit.buffer.BufferAdapter` class is an example of the adapter pattern.

### 1.4.1 Purpose

Structural pattern.

### 1.4.2 Participants<sup>1</sup>

The participants are the classes : `org.gjt.sp.jedit.buffer.BufferAdapter`, `org.gjt.sp.jedit.buffer.BufferListener`, `org.gjt.sp.jedit.textarea.ElasticTabStopBufferListener` and a client.

- **BufferAdapter** : Adapter
- **BufferListener** : Adaptee
- **ElasticTabStopBufferListener** : Target

### 1.4.3 Class diagram

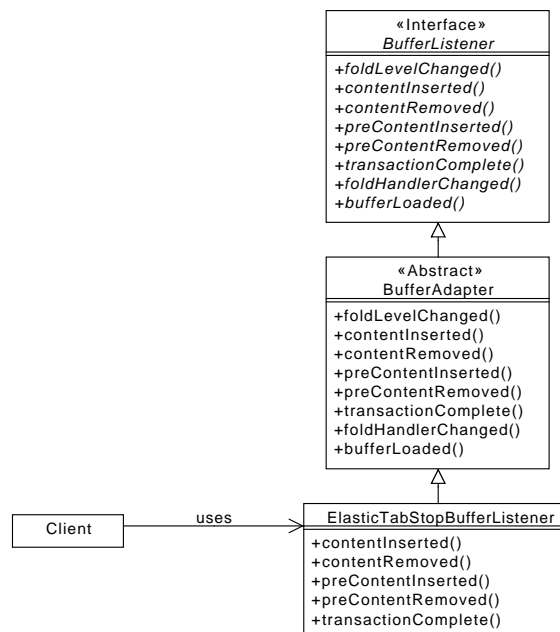


Figure 4: BufferAdapter class diagram

### 1.4.4 Concrete situation description

In this situation, the `BufferListener` interface contains several methods but not all of them are necessary for its potential subclasses. The `BufferAdapter` abstract class implements all the methods of the `BufferListener` interface but these methods contain an empty body in order to let the next Target (`ElasticTabStopBufferListener`) classes only implement the methods they need.

In this case, the adapter pattern is not used to let two incompatible existing classes work together but this pattern is used to let new concrete classes only implement the methods they need from this interface without implementing the methods which are not useful for these concrete classes.

## 1.5 Visitor

The `org.gjt.sp.jedit.visitors.JEditVisitor` interface is an example of the visitor pattern.

### 1.5.1 Purpose

Behavioral pattern.

### 1.5.2 Participants<sup>1</sup>

The participants are the classes : `org.gjt.sp.jedit.visitors.JEditVisitor`, `org.gjt.sp.jedit.visitors.JEditVisitorAdapter`, `org.gjt.sp.jedit.View.SetCursorVisitor`, `org.gjt.sp.jedit.visitors.SaveCaretInfoVisitor`, `org.gjt.sp.jedit.jEdit`, `org.gjt.sp.jedit.View`, `org.gjt.sp.jedit.EditPane` and `org.gjt.sp.jedit.textarea.JEditTextArea`.

- **JEditVisitor** : Visitor
- **JEditVisitorAdapter** : an Adapter that does not belong to this pattern
- **SetCursorVisitor** : Concrete Visitor
- **SaveCaretInfoVisitor** : Concrete Visitor
- **jEdit** : Object Structure
- **View** : Concrete Element and Client
- **EditPane** : Concrete Element
- **JEditTextArea** : Concrete Element

### 1.5.3 Class diagram

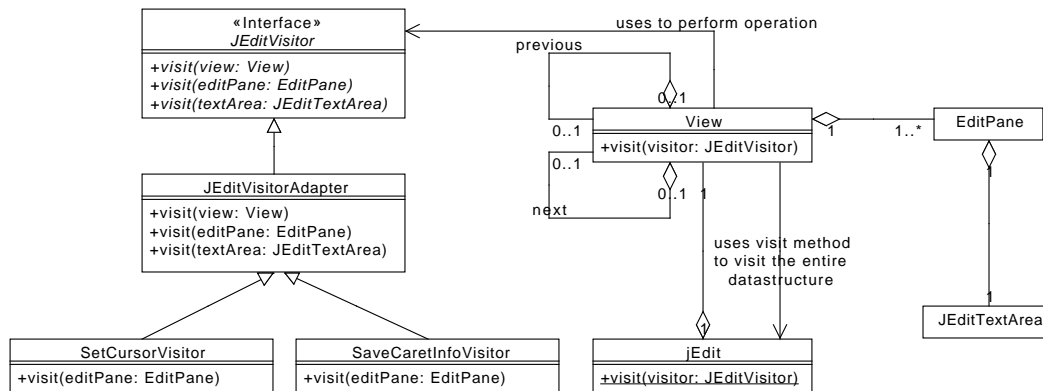


Figure 5: JEditVisitor class diagram

### 1.5.4 Concrete situation description

In this situation, the visitor design pattern is used to perform different types of action (set cursor, save caret) on a data structure containing different types of objects. The data structure is hold in the `jEdit` class (main class of the application). `jEdit` contains a current `View` and a `View` has references about its previous and next `View`. So the `Views` form a list structure. Further, each `View` may contain an `EditPane` and each `EditPane` contains a `JEditTextArea`.

A `View` may want to apply a specific action on the entire data structure. In order to perform this action on the entire data structure without depending on the concrete implementation of the elements of this data structure, the `View` will use the `visit` method of the `jEdit` class by giving to it the concrete action that has to be performed, this method will visit each element of the data structure and perform the concrete action on it.

The classes of the elements of the data structure (`View`, `EditPane`, `JEditTextArea`) do not share a common interface. However, the theoretical version of this pattern uses a common interface between the different elements of the data structure.

## 2 Exercise 2 : Recognize Design Patterns

The design patterns found are the command pattern (behavioral pattern) and the facade pattern (structural pattern).

### 2.1 Class diagram

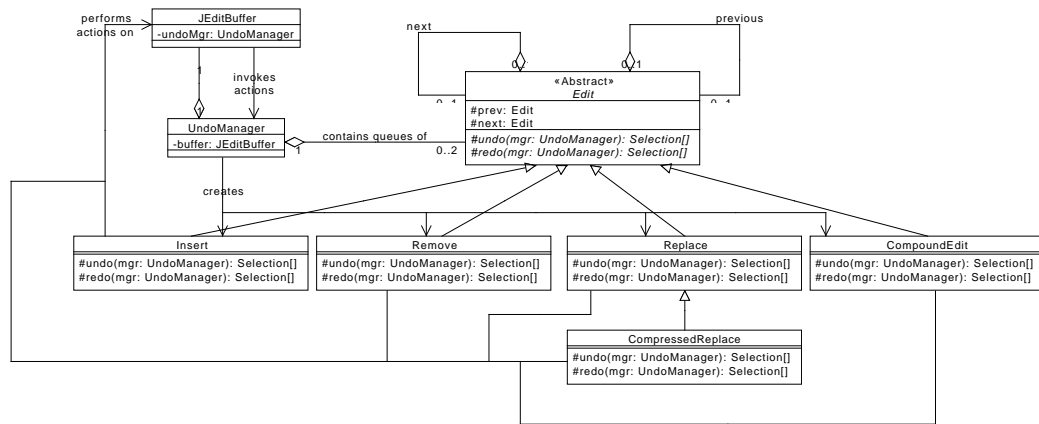


Figure 6: UndoManager class diagram

### 2.2 Command pattern

#### 2.2.1 Purpose

Behavioral pattern.

#### 2.2.2 Participants<sup>1</sup>

- **Edit** : Command
- **Insert** : Concrete Command
- **Remove** : Concrete Command
- **Replace** : Concrete Command
- **CompressedReplace** : Concrete Command
- **CompoundEdit** : Concrete Command
- **UndoManager** : Client, Invoker
- **JEditBuffer** : Receiver

#### 2.2.3 Concrete situation description

The command design pattern is used here to be able to store the actions (insert, remove, replace, ...) performed on a text (JEditBuffer). Thanks to this design pattern, the application can keep track of the actions performed during its own execution and so the application is able to undo and redo these actions. The difference between this concrete case and the theoretical case is that the client and the invoker are the same object in the practical one while they are two different objects in the theory.

## 2.3 Facade pattern

### 2.3.1 Purpose

Structural pattern.

### 2.3.2 Participants<sup>1</sup>

- **Edit** : Subsystem class
- **Insert** : Subsystem class
- **Remove** : Subsystem class
- **Replace** : Subsystem class
- **CompressedReplace** : Subsystem class
- **CompoundEdit** : Subsystem class
- **UndoManager** : Facade
- **JEditBuffer** : Client

### 2.3.3 Concrete situation description

The facade pattern is used to hide the action classes (Insert, Remove, ...) from the rest of the application. If a class want to use these classes, it has to use the UndoManager class (facade).

## 3 Exercise 3 : Coupling and Cohesion

### 3.1 Question a

- A high cohesion is preferable because it means that all the function of a class are in that class for a good reason, the class has a well defined purpose.
- A loose coupling is better because it allow the developer to modify the content of some module without jeopardize the interaction between the module and the others.

### 3.2 Question b

#### 3.2.1 MiscUtilities

The cohesion type is coincidental because this class regroup all the small functions needed at several places in the projects which therefore do not have anything in common.

It is a low type of cohesion, even the worst one. So it is indeed a bad type of cohesion.

To improve the cohesion the class could be divided in several sub classes which take care of a single feature, by example, there is several functions which focus the path, they could be regrouped in one class in order to have a logical cohesion.

#### 3.2.2 GUIUtilities

This class has a logical cohesion, the class handle all the function to create the GUI, they are grouped in this class because they all act in the same purpose even though they do not interact with each other.

It is a low type of cohesion. So it is a bad type of cohesion.

The class could be sub divided to focus on single element of the GUI per classes but it will only improve the logical cohesion, in order to achieve a functional cohesion, the functions could be dispatched where they are actually useful if possible, i.e if they are not used at several places in the project.

#### 3.2.3 io/VFSFile.java

This class represents an ADT, therefore the cohesion is maximal.

The cohesion is high in this class. It is the highest type of cohesion and so it is a good type of cohesion.

### 3.3 Question c

The coupling between these two classes is of the type Data because some data shared between the modules are done through parameters (e.g `advanceSplashProgress`).  
It is a quite loose type of coupling and so it is a quite good one.