Software Architectures - Assignement 1: Design patterns

Abdeslam Bakkali Taheri Vincent Lindivat

February 23, 2015

Exercise 1: Find instances of Design Patterns

In this exercise, we had to find instances of design patterns in Jedit's code, Jedit being an open source text editor written in Java.

Singleton

The singleton pattern is categorized as a creational pattern because it ensures that a specific class is **instantiated** only once.

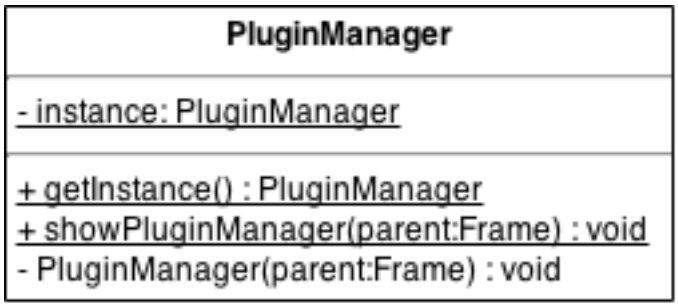
It is often criticized because an instance of a class using this pattern can be used like a global variable. A global variable causes problem for unit testing, because it can be modified by everyone and causes unpredictability in the state of the system. It also introduces coupling with classes using it, which renders them hardly reusable.

While using global variables may be seen as bad practice, and while there are alternatives (e.g. dependency injection), these alternatives may bring drawbacks such as having the code difficult to read, or the intent of the developpers may not be easy to grasp at first glance. Global variables may even be the simplest way of using tools such as a Logging class which is not directly part of the application but only here to help debugging.

The example we chose for a singleton implementation is the *PluginManager*[1] class. This class handles the window (extends *JFrame*) where plugins are installed and updated. The singleton pattern is used in this case so that only one plugin manager window can be instantiated and displayed at once when invoking it via the menu (*Plugins -> Plugin Manager...*). The following methods are involved :

[1] *org.gjt.sp.jedit.pluginmgr* package

- the *showPluginManager()* method instantiate a PluginManager if it has not already been instantiated, else it brings the plugin manager window to the front;
- The *getInstance()* method retrieve the current instance of PluginManager (can be null if it has not yet been instantiated or if the instance has been disposed of with the *dispose()* method).



The following classes are also implementing a singleton pattern, but we won't describe them in details.

- *ServicesManager*'s *Descriptor* inner class (*org.gjt.sp.jedit*)
- *ReflectManager* (*org.gjt.sp.jedit.bsh*)
- *KillRing* (*org.gjt.sp.jedit.buffer*)
- *ModeProvider* (*org.gjt.sp.jedit.syntax*)
- *TransferHandler* (*org.gjt.sp.jedit.datatransfer*)
- *DockableWindowFactory* (*org.gjt.sp.jedit.gui*)

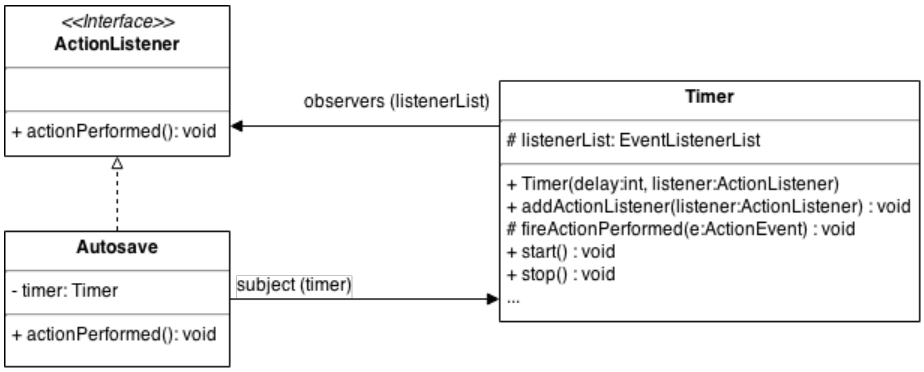
Abstract Factory

- Creational
- ServiceManager
  - "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." -> here we create related objects which are the services
- deviation from original pattern: no abstract class from which the Service-Manager inherits

Observer

The Observer is a behavioral pattern, it is justified by the fact that subjects communicate with observers to which they are registered.

The example we chose for this pattern is composed of the *Autosave*[2], *ActionListener*[3] & *Timer*[4] classes. The *Autosave* class' purpose is to automatically save all buffers with unsaved changes after a certain amount of time has elapsed (default value is 30 secondes). This class implements the *ActionListener* (awt) interface which contains the *actionPerformed()* method. This method is called by a subject, here the subject being a *Timer* (swing) which calls *actionPerformed()* after the specified amount of time has elapsed. Finally, *actionPerformed()* calls the *autosave()* method on all the buffers (the buffers are globally accessible from *jEdit*'s *getBuffer()* static method).



The following classes are also implementing an observer pattern but were not chosen as our example.

- *RegistersListener* as the observer interface, *JEditRegistersListener* as the concrete observer, and *Registers* as the subject
- *HelpHistoryModelListener* as the observer interface, *HelpViewer* as the concrete observer and *HelpHistoryModel* as the subhect (*org.gjt.sp.jedit.help*)

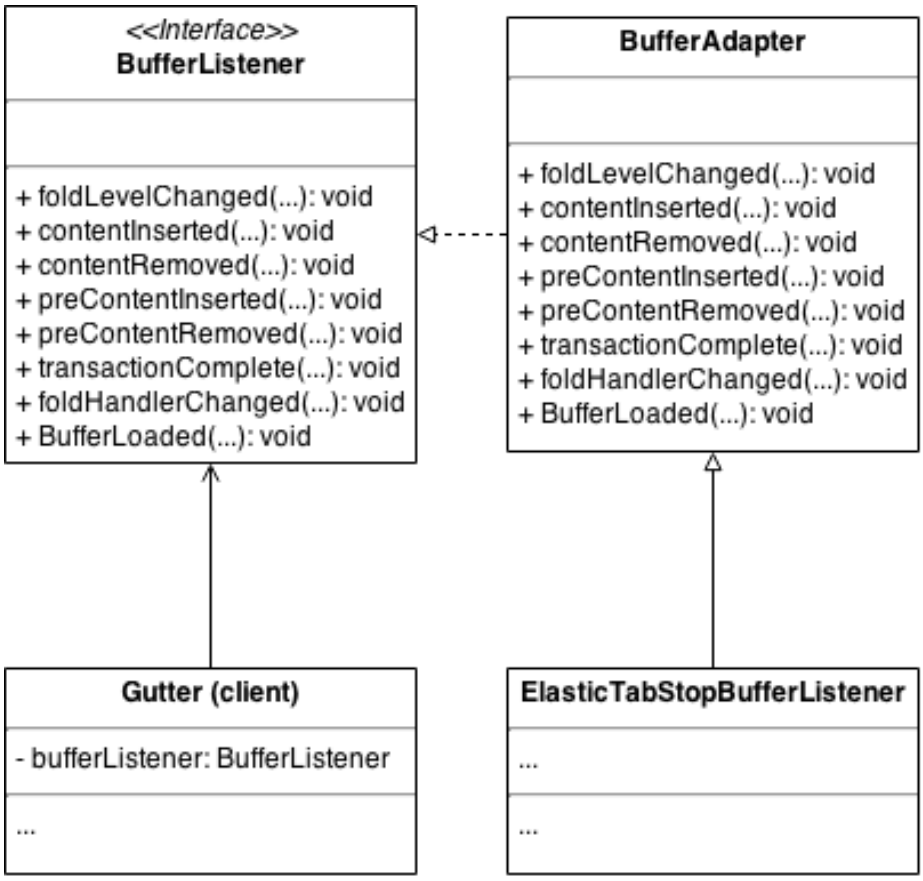
Adapter

The adapter pattern is a structural pattern used to pass an instance of a class having an interface to a client expecting another interface through a wrapper.

Here we have the *BufferAdapter*[5] class which implements the *BufferListener*[6] interface. This class contains all the *BufferListener*'s methods with empty bodies. Then, one class can inherit from *BufferAdapter* in order to avoid having to implement all *BufferListener*'s methods (e.g. *ElasticTabStopBufferListener*[7], or the anonymous class in the *Gutter*[8] class' constructor). As stated in *BufferListener*'s

[2] *org.gjt.sp.jedit.Autosave*
[3] *java.awt.event.ActionListener*
[4] *javax.swing.Timer*
[5] *org.gjt.sp.jedit.buffer* package
[6] *org.gjt.sp.jedit.buffer* package
[7] *org.gjt.sp.jedit.textarea* package
[8] *org.gjt.sp.jedit.textarea* package

documentation, this interface may change in the future. By using *BufferAdapter* instead of *BufferListener* as the expected interface, developpers will only have to modify *BufferAdapter* if *BufferListener* is modified.



Other Adapter patterns were found in jedit's code which have the same purpose as described above :

- *BufferSetAdapter* implementing *BufferSetListener* (*org.gjt.sp.jedit.bufferset*)
- *JEditVisitorAdapter* implementing *JEditVisitor* (*org.gjt.sp.jedit.visitors*)

Visitor

- Behavioral

Exercise 2: Recognize Design Patterns

Exercise 3: Coupling & Cohesion

References

Miscellaneous

The following notes must not be included in the report, there are just some interesting things I noticed after searching information for this assignement.

- A **nested class** is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private.) (source)
- Resources related to the Abstract Factory pattern :
  - https://en.wikipedia.org/wiki/Abstract_factory_pattern
  - http://sourcemaking.com/design_patterns/abstract_factory
  - http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
  - http://www.oodesign.com/abstract-factory-pattern.html (!!!)
  - http://www.dofactory.com/net/abstract-factory-design-pattern
  - "An abstract factory has multiple factory methods, each creating a different product. The products produced by one factory are intended to be used together (your printer and cartridges better be from the same (abstract) factory). As mentioned in answers above the families of AWT GUI components, differing from platform to platform, are an example of this (although its implementation differs from the structure described in Gof)." (source)
  - "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."
- categories of design patterns
  - "Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case." (wikipedia)

– "Structural: These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality." (wikipedia)
– "Behavioral: Most of these design patterns are specifically concerned with communication between objects." (wikipedia)