Examples of GoF Design Patterns in Java's core libraries

Asked 10 years, 4 months ago Active 7 months ago Viewed 494k times

672

3214

This question's answers are a community effort. Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

I am learning GoF Java Design Patterns and I want to see some real life examples of them. What are some good examples of these Design Patterns in Java's core libraries?

java oop design-patterns java-api

edited Apr 11 '15 at 3:40

community wiki
16 revs, 7 users 56%
unj2

comments disabled on deleted / locked posts / reviews

7 Answers

Active | Oldest | Votes

3186

+250

You can find an overview of a lot of design patterns in Wikipedia. It also mentions which patterns are mentioned by GoF. I'll sum them up here and try to assign as many pattern implementations as possible, found in both the Java SE and Java EE APIs.

Creational patterns

Abstract factory (recognizeable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type)

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

Builder (recognizeable by creational methods returning the instance itself)

- `java.lang.StringBuilder#append()` (unsynchronized)
- `java.lang.StringBuffer#append()` (synchronized)
- `java.nio.ByteBuffer#put()` (also on `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` and `DoubleBuffer`)
- `javax.swing.GroupLayout.Group#addComponent()`
- All implementations of `java.lang.Appendable`
- `java.util.stream.Stream.Builder`

Factory method (recognizeable by creational methods returning an implementation of an abstract/interface type)

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`

- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String)` (Returns singleton object per protocol)
- `java.util.EnumSet#of()`
- `javax.xml.bind.JAXBContext#createMarshaller()` and other similar methods

Prototype **(recognizeable by creational methods returning a *different* instance of itself with the same properties)**

- `java.lang.Object#clone()` (the class has to implement `java.lang.Cloneable`)

Singleton **(recognizeable by creational methods returning the *same* instance (usually of itself) everytime)**

- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

Structural patterns

Adapter **(recognizeable by creational methods taking an instance of *different* abstract/interface type and returning an implementation of own/another abstract/interface type which *decorates/overrides* the given instance)**

- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream)` (returns a `Reader`)
- `java.io.OutputStreamWriter(OutputStream)` (returns a `Writer`)
- `javax.xml.bind.annotation.adapters.XmlAdapter#marshal()` and `#unmarshal()`

Bridge **(recognizeable by creational methods taking an instance of *different* abstract/interface type and returning an implementation of own abstract/interface type which *delegates/uses* the given instance)**

- None comes to mind yet. A fictive example would be `new LinkedHashMap(LinkedHashSet<K>, List<V>)` which returns an unmodifiable linked map which doesn't clone the items, but *uses* them. The `java.util.Collections#newSetFromMap()` and `singletonXXX()` methods however comes close.

Composite **(recognizeable by behavioral methods taking an instance of *same* abstract/interface type into a tree structure)**

- `java.awt.Container#add(Component)` (practically all over Swing thus)
- `javax.faces.component.UIComponent#getChildren()` (practically all over JSF UI thus)

Decorator **(recognizeable by creational methods taking an instance of *same* abstract/interface type which adds additional behaviour)**

- All subclasses of `java.io.InputStream` , `OutputStream` , `Reader` and `Writer` have a constructor taking an instance of same type.
- `java.util.Collections` , the `checkedXXX()` , `synchronizedXXX()` and `unmodifiableXXX()` methods.
- `javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`
- `javax.swing.JScrollPane`

Facade **(recognizeable by behavioral methods which internally uses instances of *different* independent**

- `javax.faces.context.FacesContext`, it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection).
- `javax.faces.context.ExternalContext`, which internally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

Flyweight **(recognizeable by creational methods returning a cached instance, a bit the "multiton" idea)**

- `java.lang.Integer#valueOf(int)` (also on `Boolean`, `Byte`, `Character`, `Short`, `Long` and `BigDecimal`)

Proxy **(recognizeable by creational methods which returns an implementation of given abstract/interface type which in turn *delegates/uses* a *different* implementation of given abstract/interface type)**

- `java.lang.reflect.Proxy`
- `java.rmi.*`
- `javax.ejb.EJB` (explanation here)
- `javax.inject.Inject` (explanation here)
- `javax.persistence.PersistenceContext`

Behavioral patterns

Chain of responsibility **(recognizeable by behavioral methods which (indirectly) invokes the same method in *another* implementation of *same* abstract/interface type in a queue)**

- `java.util.logging.Logger#log()`
- `javax.servlet.Filter#doFilter()`

Command **(recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a *different* abstract/interface type which has been *encapsulated* by the command implementation during its creation)**

- All implementations of `java.lang.Runnable`
- All implementations of `javax.swing.Action`

Interpreter **(recognizeable by behavioral methods returning a *structurally* different instance/type of the given instance/type; note that parsing/formatting is not part of the pattern, determining the pattern and how to apply it is)**

- `java.util.Pattern`
- `java.text.Normalizer`
- All subclasses of `java.text.Format`
- All subclasses of `javax.el.ELResolver`

Iterator **(recognizeable by behavioral methods sequentially returning instances of a *different* type from a queue)**

- All implementations of `java.util.Iterator` (thus among others also `java.util.Scanner`!).
- All implementations of `java.util.Enumeration`

Mediator **(recognizeable by behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance)**

- `java.util.concurrent.ExecutorService` (the `invokeXXX()` and `submit()` methods)
- `java.util.concurrent.ScheduledExecutorService` (all `scheduleXXX()` methods)
- `java.lang.reflect.Method#invoke()`

Memento **(recognizeable by behavioral methods which internally changes the state of the *whole* instance)**

- `java.util.Date` (the setter methods do that, `Date` is internally represented by a `long` value)
- All implementations of `java.io.Serializable`
- All implementations of `javax.faces.component.StateHolder`

Observer (or Publish/Subscribe) **(recognizeable by behavioral methods which invokes a method on an instance of *another* abstract/interface type, depending on own state)**

- `java.util.Observer` / `java.util.Observable` (rarely used in real world though)
- All implementations of `java.util.EventListener` (practically all over Swing thus)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

State **(recognizeable by behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally)**

- `javax.faces.lifecycle.LifeCycle#execute()` (controlled by `FacesServlet`, the behaviour is dependent on current phase (state) of JSF lifecycle)

Strategy **(recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a *different* abstract/interface type which has been *passed-in* as method argument into the strategy implementation)**

- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).
- `javax.servlet.Filter#doFilter()`

Template method **(recognizeable by behavioral methods which already have a "default" behaviour defined by an abstract type)**

- All non-abstract methods of `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`.
- All non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap`.
- `javax.servlet.http.HttpServlet`, all the `doXXX()` methods by default sends a HTTP 405 "Method Not Allowed" error to the response. You're free to implement none or any of them.

Visitor **(recognizeable by two *different* abstract/interface types which has methods defined which takes each the *other* abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it)**

- `javax.lang.model.element.AnnotationValue` and `AnnotationValueVisitor`
- `javax.lang.model.element.Element` and `ElementVisitor`

- `javax.faces.component.visit.VisitContext` and `VisitCallback`

edited Oct 2 '18 at 15:08

community wiki
43 revs, 14 users 73%
BalusC

22 impressive.. :) +1. `javax.lang.model.element` defines visitors ;) I'm not quite sure whether `doXXX` and `doFilter` are "strategies". – Bozho Apr 26 '10 at 13:14

14 The mentioned builders e.g. StrinbgBuilder are all not an example for the Builder-Pattern. It is a very common mistake however to consider them as builders (so you are not really to blame ^_^) – Angel O'Sphere May 25 '11 at 13:41

76 @BalusC, I have a question to ask you. Did you read the **WHOLE** source code of Java and JSF? – Tapas Bose Jan 9 '13 at 21:39

20 @Tapas: I did not read everything, only parts which I needed to, or were just curious as to how "they" did it. – BalusC Jan 9 '13 at 21:41

6 Most of the examples under "Factory Method"are examples of "static factory" which is not a GoF pattern. Not correct. – ring bearer May 4 '15 at 11:16

107

1. Observer pattern throughout whole swing (`Observable` , `Observer`)
2. MVC also in swing
3. Adapter pattern: InputStreamReader and OutputStreamWriter NOTE: `ContainerAdapter` , `ComponentAdapter` , `FocusAdapter` , `KeyAdapter` , `MouseAdapter` are *not* adapters; they are actually Null Objects. Poor naming choice by Sun.
4. Decorator pattern (`BufferedInputStream` can decorate other streams such as `FilterInputStream`)
5. AbstractFactory Pattern for the AWT Toolkit and the Swing pluggable look-and-feel classes
6. `java.lang.Runtime#getRuntime()` is Singleton
7. `ButtonGroup` for Mediator pattern
8. `Action` , `AbstractAction` may be used for different visual representations to execute same code -> Command pattern
9. Interned Strings or CellRender in JTable for Flyweight Pattern (Also think about various pools - Thread pools, connection pools, EJB object pools - Flyweight is really about management of shared resources)
10. The Java 1.0 event model is an example of Chain of Responsibility, as are Servlet Filters.
11. Iterator pattern in Collections Framework
12. Nested containers in AWT/Swing use the Composite pattern
13. Layout Managers in AWT/Swing are an example of Strategy

and many more I guess

edited Aug 9 '19 at 7:36

community wiki
9 revs, 5 users 55%
jitter

1 Thanks for the tip on MouseAdapter. I found this exaplanation: stackoverflow.com/questions/9244185/… – Lincoln May 20 '15 at 14:24

Note that Swing is only loosely based on MVC. It has collapsed View and Controller into one class. – Matthias Braun Dec 23 '19 at 8:05