Introduction to Design Patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Uses of Design Patterns :

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Design patterns provide general solutions which are easy to apply to a broader range of problems, documented in a format that doesn't require specifics tied to a particular problem. In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

1 Creational Design Patterns

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class). But everyone knows an object is created by using new keyword in java. Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

- Factory Method
- Abstract Factory
- Singleton
- Prototype
- Builder
- Object Pool

2 Structural Design Patterns

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplifies the structure by identifying the relationships. These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

3 Behavioral Design Patterns

Behavioral design patterns are concerned with the interaction and responsibility of objects. In these design patterns,the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled. That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

- Observer
- State
- Strategy
- Chain of Responsibiliy
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Template
- Visitor
- Null Object

Few examples of Design Patterns used in JDK

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.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `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 abstract/interface types)**

- `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.Timer` (all `scheduleXXX()` methods)
- `java.util.concurrent.Executor#execute()`
- `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.lang.model.type.TypeMirror` and `TypeVisitor`
- `java.nio.file.FileVisitor` and `SimpleFileVisitor`
- `javax.faces.component.visit.VisitContext` and `VisitCallback`