Analyze typical design patterns in Java and C++

Abstract

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Keywords:

OOP,patterns,Java,C++,software design,Maven.

1.1 Design patterns in Java

In this part, there is the description of all core design patterns that are used in Java[3].

Abstract factory (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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) (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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 (recognizable 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

Summing up Java part

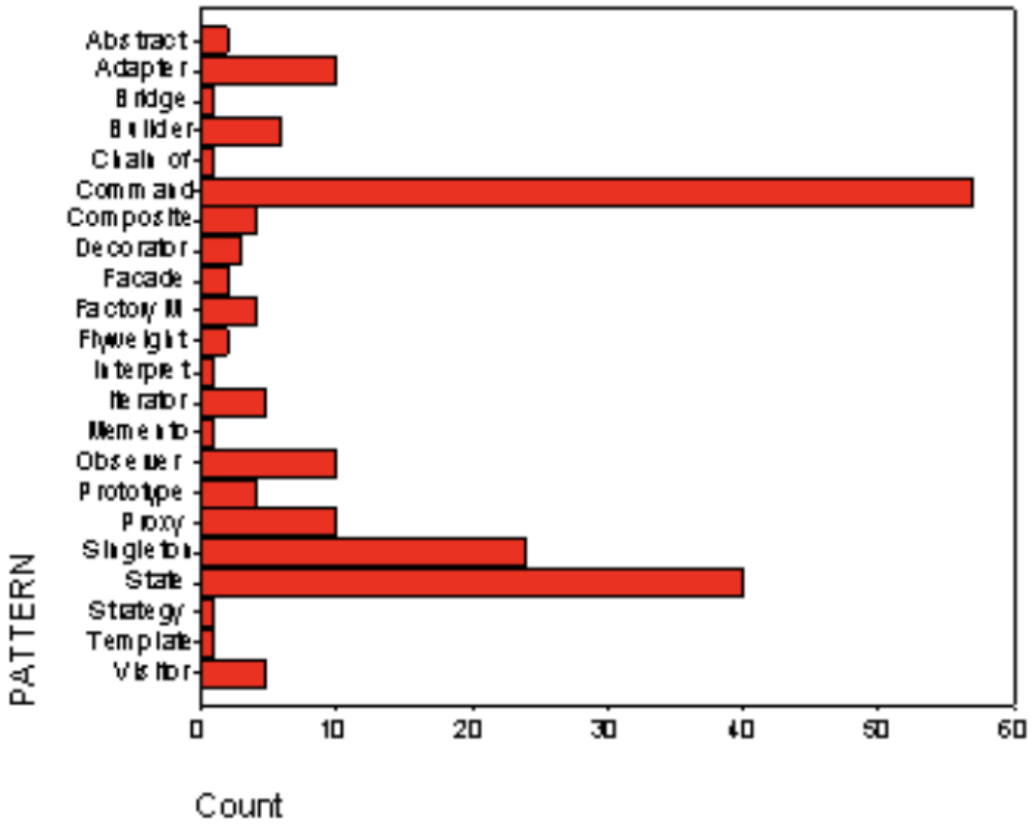

Figure 6: Number of projects using individual patterns

According to the quantitative research of Michael Hahsler,in his paper, A Quantitative Study of the Application of Design Patterns in Java. The used data set includes 1319 open source projects from SourceForge using Java as the main programming language.

The pattern names *Command* and *State* appear very often in the data[4]. Since both words are very common in programming and design, this could indicate a problem in our way to identify patterns by their name and some keywords from the intents. We have to keep this observation in mind. In third place appears *Singleton* which is a very unusual word for developers not familiar with design patterns. This leads to the conclusion that for Java the application of the pattern Singleton seems to provide important design advantages.

2.1 Design patterns in C++

In this section, there is the description of all core design patterns,that are used in C++.

Creational Patterns

- Abstract Factory, families of product objects
- Builder, how a composite object gets created
- Factory Method, subclass of object that is instantiated
- Prototype, class of object that is instantiated
- Singleton, the sole instance of a class

Structural Patterns

- Adapter, interface to an object
- Bridge, implementation of an object
- Composite, structure and composition of an object
- Decorator, responsibilities of an object without subclassing
- Façade, interface to a subsystem
- Flyweight, storage costs of objects
- Proxy, how an object is accessed (its location)

Behavioral Patterns

- Chain of Responsibility, object that can fulfill a request
- Command, when and how a request is fulfilled
- Interpreter, grammar and interpretation of a language
- Iterator, how an aggregate's elements are accessed
- Mediator, how and which objects interact with each other
- Memento, what private information is stored outside an object, and when
- Observer, how the dependent objects stay up to date
- State, states of an object
- Strategy, an algorithm
- Template Method, steps of an algorithm
- Visitor, operations that can be applied to objects without changing their classes

Summing up C++ part

Patterns	Occurrences	%
No Pattern	183634	77.5 %
Factory	20237	8.5 %
Singleton	3331	1.4 %
Observer	16061	6.8 %
Template Method	5381	2.3 %
Decorator	1513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2279	1.0 %
Observer + Singleton	2390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

TABLE IX

FREQUENCIES OF PATTERN OCCURRENCES, AND PERCENTAGE OF CODE COVERED BY THE PATTERNS

As it was analyzed in research work of Marek Voka´c, Defect Frequency and Design Patterns: An Empirical Study of Industrial Code,with the total number of test C++ projects of 200, the most used design patterns across C++ projects was Factory and Observer, Template and Sigleton are much less used.

3.1 Conclusion

This first study of the application of design patterns in real-world software development projects has many limitations, e.g. it does not include additional information on the quality of the produced code, the code itself is not analyzed using object-oriented metrics and the actual effort used for the projects is unknown. Also the projects are open source projects which means that the development process differs significantly from industrial projects. But even with this limitation the quantitative results confirm the most important claim of design patterns, namely that design patterns are used to facilitate communication between developers which, without doubt, is also vital for industrial software development.

To compare mostly common design patterns between C++ and Java it is possible to notice Singleton and Factory.

According to the direct meaning of Sigleton:
The sole purpose of a singleton/creational software design pattern is to create a single purpose instance.Such a software design pattern is used for logging, thread pool, driver objects, and caching. The Java singleton pattern is interoperable and works well within other advanced designs.So, the usage of it in open-source projects are completely understandable.

Factory is a creational pattern that helps create an object without the user getting exposed to creational logic. The only problem with a factory method is it relies on the concrete component. What happens is when you use a factory method, there is no specific definition of class. So, instead of the constructor class, Someclass is used. Creating a new object is coupled with the concrete component that can have issues with such a class.

References

1. https://www.javatpoint.com/design-patterns-in-java
2. https://refactoring.guru/design-patterns/cpp
3. https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries
4. https://michael.hahsler.net/research/patterns_working2003/designpatterns_java.html
5. http://www.ptidej.net/courses/log6306/fall12/readingnotes/4/Vokac%20-%20Defect%20Frequency%20and%20Design%20Patterns%20-%20An%20Empirical%20Study%20of%20Industrial%20Code.pdf
6. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7162509/