Analysis of typical design patterns in Java and C++

Abstract

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. This paper seeks to analyse typical design patterns common to Java and C++ and their usage statistics.

***Keywords**: OOP, design patterns, Java, C++, software design, Maven.*

Introduction

Developers need to understand software systems before they can maintain them, even in cases where documentation and/or design models are missing or of a poor quality. In most cases only the source code as the basic form of documentation is available [1]. Maintenance is a time-consuming activity within software development, and it requires a good understanding of the system in question. The knowledge about design patterns can help developers to understand the underlying architecture faster.

A design pattern is a general reusable 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. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints [2], [3].

Design patterns help to effectively speed up development and engineering processes by providing proven development patterns/paradigms. Quality software design requires considering issues that may not be visible until later in the implementation. Reusing design patterns helps to avoid subtle issues that may be catastrophic and help improve code reliability for programmers and architects familiar with the patterns.

Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem. They help software engineers 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 [4].

In this paper, we analyse typical design patterns in Java and C++ and detect common patterns and further look at their usage statistics. There are many design patterns in software development and several of them are common to Java and C++. These design patterns come under three main types.

Creational

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype.

Use case of creational design pattern

1) Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance, we can control load balance, unnecessary connections, etc.

2) Suppose you want to create multiple instances of similar kind and want to achieve loose coupling then you can go for Factory pattern. A class implementing factory design pattern works as a bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as back end, but in future need to change database to oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation, we should go for the factory design pattern in order to achieving loose coupling and the creation of a similar kind of object.

Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.*

Use Case Of Structural Design Pattern-

1) When 2 interfaces are not compatible with each other and want to establish a relationship between them through an adapter it’s called an adapter design pattern. Adapter pattern converts the interface of a class into another interface or class that the client expects, i.e adapter lets classes

works together that could not otherwise because of incompatibility. so in these type of incompatible scenarios, we can go for the adapter pattern.

Behavioural

Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns.

Behavioral patterns are *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor*

Use Case of Behavioral Design Pattern-

1) The template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. For example, in your project, you want the behavior of the module to be able to extend, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, no one is allowed to make source code changes to it, i.e you can add but can't modify the structure in those scenarios a developer can approach template design pattern [4] [5].

Sample implementation of these patterns [6] are available here in both Java and C++.

Criticism

The concept of design patterns has been criticized by some in the field of computer science.

Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay ***Revenge of the Nerds*** [4].

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern [4].

Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature [4].

Common design patterns in popular open-source repositories and their usage statistics

To Do

To Do

To Do

Conclusion

To Do

References

[1] D. Streitferdt, C. Heller, and I. Philippow, ‘Searching design patterns in source code’, in *29th Annual International Computer Software and Applications Conference (COMPSAC’05)*, Jul. 2005, vol. 2, pp. 33-34 Vol. 1. doi: 10.1109/COMPSAC.2005.135.

[2] ‘Software Design Patterns - GeeksforGeeks’. https://www.geeksforgeeks.org/software-design-patterns/ (accessed Jul. 23, 2021).

[3] W. Schaffer and A. Zundorf, ‘Round-trip engineering with design patterns, UML, java and C++’, in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, May 1999, pp. 683–684. doi: 10.1145/302405.302956.

[4] ‘Design Patterns and Refactoring’. https://sourcemaking.com (accessed Jul. 23, 2021).
[5] ‘Design Patterns | Set 1 (Introduction)’, *GeeksforGeeks,* Aug. 06, 2015. https://www.geeksforgeeks.org/design-patterns-set-1-introduction/ (accessed Jul. 23, 2021).
[6] ‘Huston Design Patterns’. http://www.vincehuston.org/dp/ (accessed Jul. 26, 2021).