# Analysis of Design Patterns
# Stage V

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

## Abstract

Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Eolang programming language is a new Object-Oriented Programming language that has been going through many phases of development in this research. In this phase, we analyze typical design patterns in Java and C++ by researching multiple popular open-source repositories, to detect common design patterns and their usage statistics and then suggest alternatives in Eolang that could replace such patterns which are not supported in the language. We additionally provide explanation why Eolang alternative could be better

***Keywords:*** OOP, C++, Java, design patterns

## 1 Introduction

This report includes pattern analysis, usage statistics, comparison of design patterns implementation in C++ and Java, implementation of some popular design patterns in Eolang, explanation of why Eolang does not support some design patterns, description of alternatives and explanation of why Eolang alternatives could be better.

### 1.1 Backround

Design is one of the most difficult task in software development [1] and Developers, who have eagerly adopted them over the past years [2], needed to understand not only design patterns [3] but the 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 [4]. 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. Using design patterns is a widely accepted method to improve software development [5]. A design pattern is a general reusable solution to a commonly occurring[6] problem in software design [7]–[9]. A design pattern isn't a finished design that can be transformed directly into code neither are they static entities, but evolving descriptions of best practices [10]. 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 [11], [12]. 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 [13], [14]. In short, the advantages of design patterns include decoupling a request from specific operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), independent from algorithmic solutions (Iterator, Strategy, Visitor), or avoid modifying implementations (Adapter, Decorator, Visitor) [15]. Design patterns, overall, helps to thoroughly and designed well implemented frameworks enabling a degree of software reusability that can significantly improve software quality [16], [17].

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.

## 2 Design Patterns

### 2.1 Creational design Patterns

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 [18], [19].

#### 2.1.1 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.

**2.1.2 Factory Method:** Factory Method, also known as virtual constructor, is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created [20] in a way such that it doesn't have tight coupling with the class hierarchy of the library [21]. Factory Method is one of the most used design patterns in Java [22]. It is widely used in C++ code and is very useful when you need to provide a high level of flexibility for your code [23].

**2.1.3 Abstract Factory.** Abstract Factory patterns work around a super-factory which creates other factories. Thus, it defines a new Abstract Product Factory for each family of products. This factory is also called as factory of factories. It provides one of the best ways to create an object. Abstract Factory design pattern covers the instantiation of the concrete classes behind two kinds of interfaces, where the first interface is responsible for creating a family of related and dependent products, and the second interface is responsible for creating concrete products. The client is using only the declared interfaces and is not aware which concrete families and products are created [24]. Adding a new family of products affects any existing class that depends on it, and requires complex changes in the existing Abstract Factory code, as well as changes in the application or client code [24]. Bulajic and Jovanovic [24] demonstrates a solution where adding a new product class does not require complex changes in existing code, and the number of product classes is reduced to one product class per family of related or dependent products.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern [25]. Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern [26].

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So, at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type [26], [27]. Fig. 1 shows a UML class diagram example for an Abstract Factory Design pattern.

This pattern is particularly useful when the client doesn't know exactly what type to create. The Abstract Factory pattern helps you control the classes of objects that an application creates by isolating concrete classes. The class of a concrete factory appears only once in an application, that is where it's instantiated. This makes it easy to change the concrete factory an application uses. Abstract Factory makes this easy for an application use object from only one family at a time when product objects in a family are designed to work together. Abstract Factory interface fixes the set of products that can be created. This serves as a disadvantage because extending abstract factories to produce new kinds of Products isn't easy. The abstract factory design pattern can be implemented in both Java and C++ as demonstrated at [26], [28]. The Abstract Factory pattern is pretty common in C++ code. Many frameworks and libraries use it to provide a way to extend and customize their standard components

#### 2.1.4 Builder.

Builder pattern builds a complex object using simple objects and using a step-by-step approach and the final step will return the object. The builder is independent of other objects. Fig. 2 show a UML diagram of builder design pattern. Immutable objects can be build without much complex logic in object building process. Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor. As a disadvantage, it requires creating a separate ConcreteBuilder for each different type of Product [29], [30].

#### 2.1.5 Singleton.

In software engineering, the term singleton implies a class which can only be instantiated once, and a global point of access to that instance is provided [31]. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. It one of the simplest design patterns in Java and C++.

#### 2.1.6 Object Pool.

Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high. An Object pool is a container which contains some number of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.

#### 2.1.7 Prototype.

Prototype pattern refers to creating duplicate object while keeping performance in mind. This pattern involves implementing a prototype interface which tells to create a clone of the current object.

### 2.2 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 [18], [19].

#### 2.2.1 Use Case of Structural Design Pattern.

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 types of incompatible scenarios, we can go for the adapter pattern.

**2.2.2 Adapter.** Adapter design pattern allows objects with incompatible interfaces to collaborate [32]. Adapter pattern works as a bridge between those two incompatible interfaces. A real-life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop [33].

**2.2.3 Bridge.** The bridge pattern is used when we need to decouple [2] an abstraction from its implementation so that the two can vary independently [32]. It helps you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

**2.2.4 Composite.** Composite design pattern helps in composing objects into tree structures and then work with these structures as if they were individual objects. It is used where a group of objects need to be treated in similar way as a single object. This pattern creates a class that contains group of its own objects and provides ways to modify this group of same objects.

## 2.3 Decorator

Decorator design pattern allows a user to add new functionality to an existing object without altering its structure. It allows the attachment of new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

**2.3.1 Facade.** Facade design pattern, as the name goes, hides the complexities of the system and provides an interface to the client by which the client can access the system. It provides a simplified interface to a library, a framework, or any other complex set of classes. In essence, it provides methods required by the client and delegates calls to methods of existing system classes.

**2.3.2 Flyweight.** The Flyweight pattern enables you to fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object. It is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.

**2.3.3 Private Class Data.** Private Class Data is used to encapsulate class data and control write access to class attributes as it separates data from methods that us it.

**2.3.4 Proxy.** The Proxy pattern controls access to the original object [2], allowing you to perform something either before or after the request gets through to the original object. It represents functionality of another class.

## 2.4 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 and Visitor [18], [19].

**2.4.1 Use Case of Behavioral Design Pattern.** 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 [13] [34].

Sample implementation of these patterns [35] are available here in both Java and C++.

**2.4.2 Chain of responsibility.** Chain of responsibility pattern suggests a chain of receiver objects for a request. It decouples sender and receiver of a request based on type of request. It allows you to pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

**2.4.3 Command.** Command pattern is also known as action or transaction pattern. This pattern turns a request into a stand-alone object that contains all information about the request. The transformation allows you pass requests as a method argument, delay or queue a request's execution, and support undoable operations. It is data driven and it wraps an object under an object as command [2] and passes it to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

**2.4.4 Interpreter.** Interpreter pattern provides a way to evaluate language grammar or expression. Given a language, interpreter defines a representation for the language's grammar along with an interpreter that uses the representation to interpret sentences in the language. Then it maps a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design. This pattern is used in SQL parsing, symbol processing engine etc. The Iterator pattern is very commonly used design pattern in Java and .Net programming environment. It allows sequential traversal through a complex data structure without exposing its internal details [32]. This pattern is also common in C++ code. Mediator is used to reduce communication complexity between multiple objects or classes by providing a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling [32]. It encapsulates how a set of objects interact.

**2.4.5 Memento.** Memento pattern is used to restore state of an object to a previous state. Without violating encapsulation, you can capture and externalize an object's internal state so that the object can be returned to this state later.

**2.4.6 Null Object.** In Null Object pattern, a null object replaces check of NULL object instance. The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do-nothing behaviour in case data is not available.

**2.4.7 Observer.** Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically [2].

**2.4.8 State.** In State pattern, objects are created to represent various states and a context object whose behaviour varies as its state object changes. So, in State pattern, a class behaviour changes based on changes in its internal state.

**2.4.9 Strategy.** Strategy design pattern is the pattern where a class behaviour or its algorithm can be changed at run time.

**2.4.10 Template.** In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class. It allows the algorithm to vary independently from the clients that use it.

**2.4.11 Visitor.** In Visitor pattern, a visitor class changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. Visitor allows the definition of a new operation without changing the classes of the elements on which it operates.

## 3 Criticism

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

### 3.1 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 [13].

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.

### 3.2 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 $\frac{2}{3}$ of the "jurors" who attended the trial.

### 3.3 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 [13].

### 3.4 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 [13].

According to [36], generality, precision, and understandability are the most important goals to consider in order to simplify software design pattern description.

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

Hahsler [5] analyses the application of design patterns in Java by identifying patterns in projects using their log messages to look for names and descriptions. This attempt was done based on the idea that the names of design patterns become part of a common design language which developers use to communicate more efficiently. Fig. 3 shows the graph of the usage statistics according the approach of Hahsler [5].

Vokac [37] analysed the weekly evolution and maintenance of a large commercial product (C++, 500,000 LOC) over three years, comparing defect rates for classes that participated in selected design patterns to the code at large. He extracted design pattern information and concluded that Observer and Singleton patterns are correlated with larger code structures. The Template Method pattern was used in both simple and complex situations, leading to no clear tendency. The frequencies of pattern occurrence are shown in table 1.

## 5 Comparison of Design Patterns in C++ and Java

## 6 Conclusion

## References

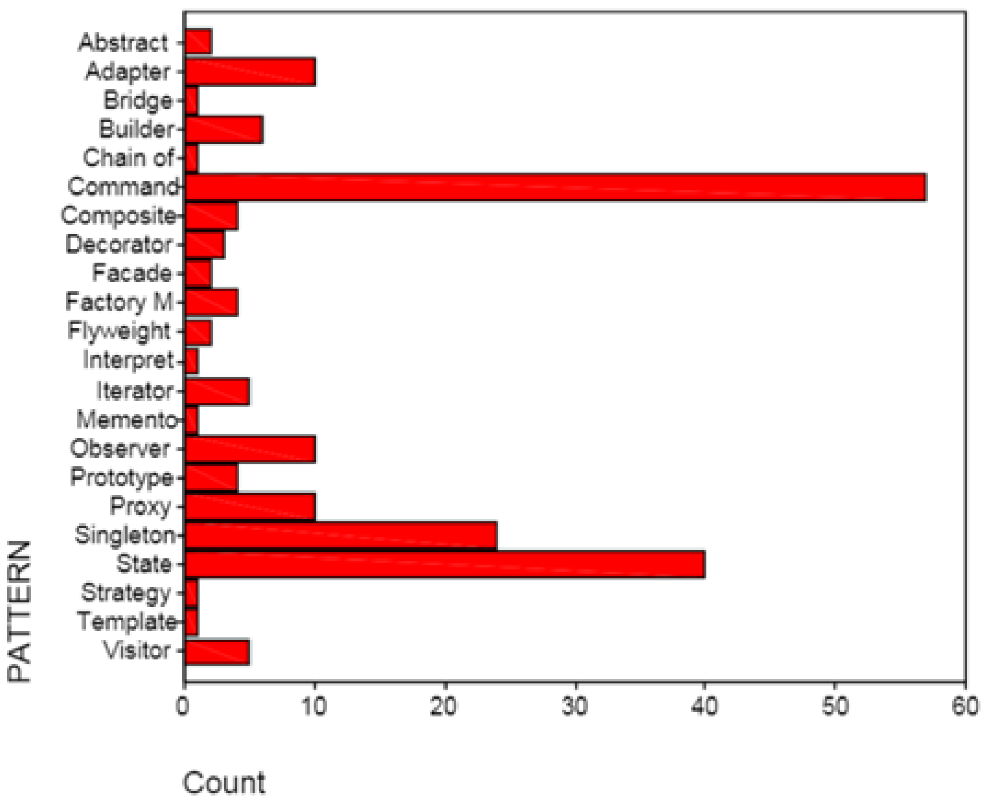
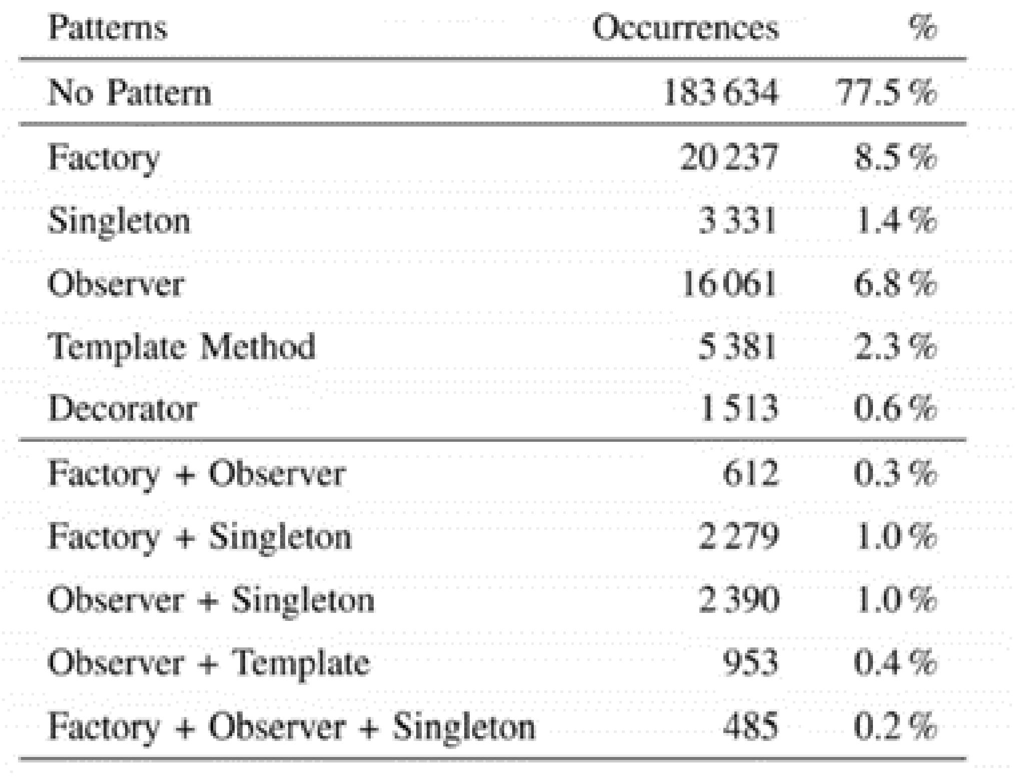


**Figure 1.** Number of Java Projects using individual design patterns [5]



| Patterns | Occurrences | % |
|---|---|---|
| No Pattern | 183 634 | 77.5 % |
| Factory | 20 237 | 8.5 % |
| Singleton | 3 331 | 1.4 % |
| Observer | 16 061 | 6.8 % |
| Template Method | 5 381 | 2.3 % |
| Decorator | 1 513 | 0.6 % |
| Factory + Observer | 612 | 0.3 % |
| Factory + Singleton | 2 279 | 1.0 % |
| Observer + Singleton | 2 390 | 1.0 % |
| Observer + Template | 953 | 0.4 % |
| Factory + Observer + Singleton | 485 | 0.2 % |

**Figure 2**