What is an Architectural Pattern?

*An****architectural pattern****is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.*

In this article, I will be briefly explaining the following 10 common architectural patterns with their usage, pros and cons.

1. **Layered pattern**
2. **Client-server pattern**
3. **Master-slave pattern**
4. **Pipe-filter pattern**
5. **Broker pattern**
6. **Peer-to-peer pattern**
7. **Event-bus pattern**
8. **Model-view-controller pattern**
9. **Blackboard pattern**
10. **Interpreter pattern**

1. Layered pattern

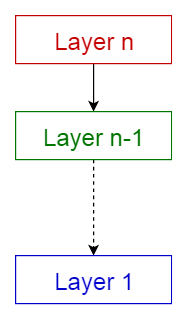
This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.

The most commonly found 4 layers of a general information system are as follows.

* **Presentation layer**(also known as **UI layer**)
* **Application layer**(also known as **service layer**)
* **Business logic layer**(also known as **domain layer**)
* **Data access layer** (also known as **persistence layer**)

Usage

* General desktop applications.
* E commerce web applications.



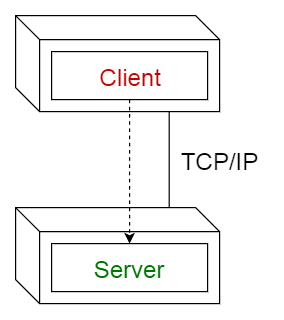
Layered pattern

2. Client-server pattern

This pattern consists of two parties; a **server** and multiple **clients**. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

Usage

* Online applications such as email, document sharing and banking.



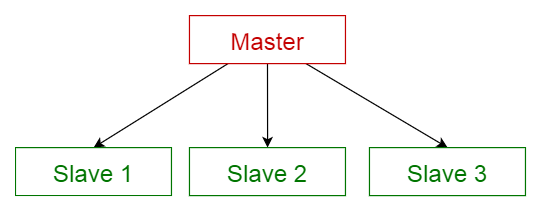
Client-server pattern

3. Master-slave pattern

This pattern consists of two parties; **master** and **slaves**. The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

Usage

* In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
* Peripherals connected to a bus in a computer system (master and slave drives).



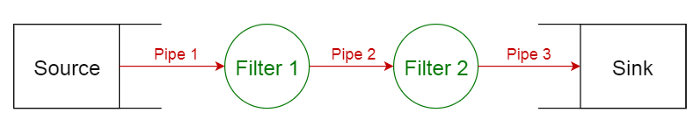
Master-slave pattern

4. Pipe-filter pattern

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a **filter**component. Data to be processed is passed through **pipes**. These pipes can be used for buffering or for synchronization purposes.

Usage

* Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
* Workflows in bioinformatics.



Pipe-filter pattern

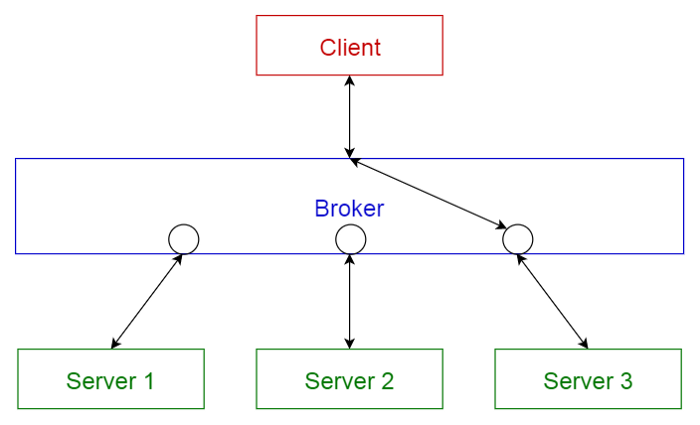
5. Broker pattern

This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations. A **broker**component is responsible for the coordination of communication among **components**.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Usage

* Message broker software such as [**Apache ActiveMQ**](https://en.wikipedia.org/wiki/Apache_ActiveMQ), [**Apache Kafka**](https://en.wikipedia.org/wiki/Apache_Kafka), [**RabbitMQ**](https://en.wikipedia.org/wiki/RabbitMQ) and [**JBoss Messaging**](https://en.wikipedia.org/wiki/JBoss_Messaging).



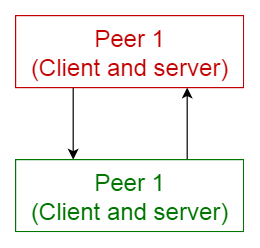
Broker pattern

6. Peer-to-peer pattern

In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

Usage

* File-sharing networks such as [**Gnutella**](https://en.wikipedia.org/wiki/Gnutella)and [**G2**](https://en.wikipedia.org/wiki/Gnutella2))
* Multimedia protocols such as [**P2PTV**](https://en.wikipedia.org/wiki/P2PTV)and [**PDTP**](https://en.wikipedia.org/wiki/Peer_Distributed_Transfer_Protocol).
* Cryptocurrency-based products such as [**Bitcoin**](https://bitcoin.org/en/) and [**Blockchain**](https://www.blockchain.com/)



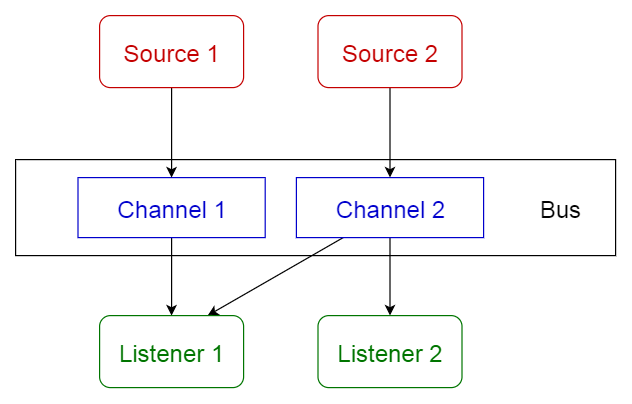
Peer-to-peer pattern

7. Event-bus pattern

This pattern primarily deals with events and has 4 major components; **event source**, **event listener**, **channel**and **event bus**. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.

Usage

* Android development
* Notification services



Event-bus pattern

8. Model-view-controller pattern

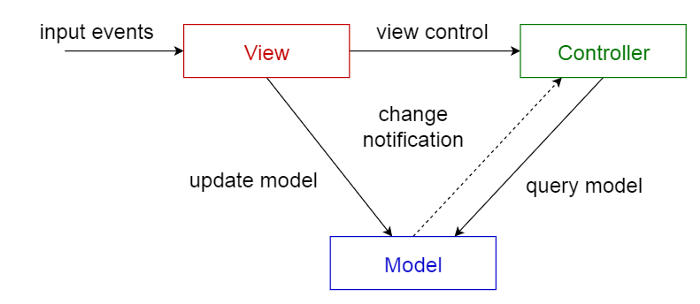
This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

1. **model** — contains the core functionality and data
2. **view** — displays the information to the user (more than one view may be defined)
3. **controller** — handles the input from the user

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

* Architecture for World Wide Web applications in major programming languages.
* Web frameworks such as [**Django**](https://en.wikipedia.org/wiki/Django_(web_framework))and [**Rails**](https://en.wikipedia.org/wiki/Ruby_on_Rails).



Model-view-controller pattern

9. Blackboard pattern

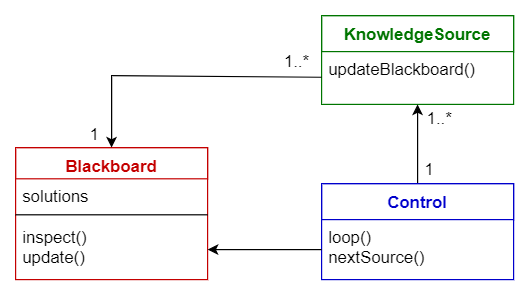
This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.

* **blackboard**— a structured global memory containing objects from the solution space
* **knowledge source** — specialized modules with their own representation
* **control component** — selects, configures and executes modules.

All the components have access to the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

Usage

* Speech recognition
* Vehicle identification and tracking
* Protein structure identification
* Sonar signals interpretation.



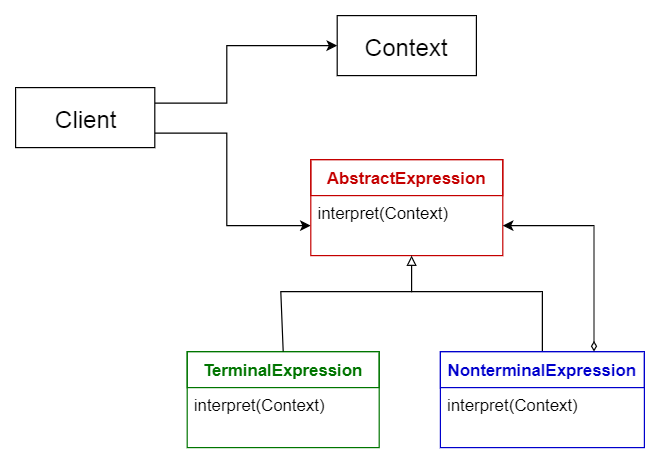
Blackboard pattern

**10. Interpreter pattern**

This pattern is used for designing a component that interprets programs written in a dedicated language. It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language. The basic idea is to have a class for each symbol of the language.

Usage

* Database query languages such as SQL.
* Languages used to describe communication protocols.



Interpreter pattern

Comparison of Architectural Patterns

The table given below summarizes the pros and cons of each architectural pattern.



Comparison of Architectural Patterns

<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>

# Software Architectural Guide

architecture was **the shared understanding that the expert developers have of the system design.**

**the decisions you wish you could get right early in a project**.

**High internal quality leads to faster delivery of new features**, because there is less cruft to get in the way.

## Application Architecture

* A body of code that's seen by developers as a single unit
* A group of functionality that business customers see as a single unit
* An initiative that those with the money see as a single budget

### [Application Boundary](https://martinfowler.com/bliki/ApplicationBoundary.html)

### [Microservices Guide](https://martinfowler.com/microservices)

### [Serverless Architectures](https://martinfowler.com/articles/serverless.html)

### [Micro Frontends](https://martinfowler.com/articles/micro-frontends.html)

### [GUI Architectures](https://martinfowler.com/eaaDev/uiArchs.html)

### [Presentation Domain Data Layering](https://martinfowler.com/bliki/PresentationDomainDataLayering.html)

## Enterprise Architecture

While application architecture concentrates on the architecture within some form of notional application boundary, enterprise architecture looks architecture across a large enterprise. Such an organization is usually too large to group all its software in any kind of cohesive grouping, thus requiring coordination across teams with many codebases, that have developed in isolation from each other, with funding and users that operate independently of each other.

### [Enterprise Architects Join the Team](https://martinfowler.com/ieeeSoftware/enterpriseArchitects.pdf)

### [The Role of an Enterprise Architect in a Lean Enterprise](https://martinfowler.com/articles/ea-in-lean-enterprise.html)

### [Products Over Projects](https://martinfowler.com/articles/products-over-projects.html)

### [The Architect Elevator — Visiting the upper floors](https://martinfowler.com/articles/architect-elevator.html)

### [Enterprise Integration Using REST](https://martinfowler.com/articles/enterpriseREST.html)

## Make architecture fit for purpose

Often, we look to the architect to give us a "good" architecture. The catch is that architecture is rarely good or bad - it's either fit or unfit for purpose. The purpose is usually dependent on the context in which the system will live rather than specific user requirements. An architect's job therefore is to consider architecture options in a broader context, which can involve diverse factors such as commercial and legal agreements, skills availability, or an installed base.

## Validate decisions through feedback loops

# SOLID Design Principles: The Single Responsibility Principle

<https://stackify.com/solid-design-principles/>

* Single Responsibility Principle
* Open/Close Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion

## Single Responsibility Principle:

## *A class should have one, and only one, reason to change.*

it makes your software easier to implement and prevents unexpected side-effects of future changes.

Asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?

If your answer includes the word “**and**”, you’re most likely breaking the single responsibility principle. Then it’s better to take a step back and rethink your current approach. There is most likely a better way to implement it.

## Open/Closed Principle

*“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”*

*A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”*

The main benefit of this approach is that an interface introduces an additional level of abstraction which enables loose coupling. The implementations of an interface are independent of each other and don’t need to share any code. If you consider it beneficial that two implementations of an interface share some code, you can either use [inheritance](https://stackify.com/oop-concept-inheritance/) or [composition](https://stackify.com/oop-concepts-composition/).

### **Liskov Substitution Principle**

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an [exception](https://stackify.com/specify-handle-exceptions-java/), if it gets called with an object of the subclass.

Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass. You can only decide to apply even stricter rules by returning a specific subclass of the defined return value, or by returning a subset of the valid return values of the superclass.

### **interface segregation principle**

*“Clients should not be forced to depend upon interfaces that they do not use.”*

We should not put all unnecessary function in interface, in-place of create multiple interface based on function and only implement those interface in independent way.

<https://stackify.com/interface-segregation-principle/>

### **Dependency Inversion Principle**

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

Based on this idea, Robert C. Martin’s definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

<https://github.com/thjanssen/Stackify-SOLID-Liskov>

New Methodology:

<https://martinfowler.com/articles/newMethodology.html>

* *Agile methods are adaptive rather than predictive.* Plan-driven methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.
* *Agile methods are people-oriented rather than process-oriented.* The goal of plan-driven methods is to define a process that will work well whoever happens to be using it. Agile methods assert that no process will ever make up the skill of the development team, so the role of a process is to support the development team in their work.
* In software: construction is so cheap as to be free
* In software all the effort is design, and thus requires creative and talented people
* Creative processes are not easily planned, and so predictability may well be an impossible target.
* We should be very wary of the traditional engineering metaphor for building software. It's a different kind of activity and requires a different process

The most important, and still difficult part is to know accurately where we are. We need an honest feedback mechanism which can accurately tell us what the situation is at frequent intervals.

The key to this feedback is iterative development. This is [not a new idea](https://www.amazon.com/gp/product/0131111558/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0131111558&linkCode=as2&tag=martinfowlerc-20). Iterative development has been around for a while under many names: incremental, evolutionary, staged, spiral... lots of names. The key to iterative development is to frequently produce working versions of the final system that have a subset of the required features. These working systems are short on functionality, but should otherwise be faithful to the demands of the final system. They should be fully integrated and as carefully tested as a final delivery.

Iterative development makes sense in predictable processes as well. But it is essential in adaptive processes because an adaptive process needs to be able to deal with changes in required features. This leads to a style of planning where long term plans are very fluid, and the only stable plans are short term plans that are made for a single iteration. Iterative development gives you a firm foundation in each iteration that you can base your later plans around.

In an adaptive process the customer has much finer-grained control over the software development process. At every iteration they get both to check progress and to alter the direction of the software development. This leads to much closer relationship with the software developers, a true business partnership. This level of engagement is not for every customer organization, nor for every software developer; but it's essential to make an adaptive process work properly.

All this yields a number of advantages for the customer. For a start they get much more responsive software development. A usable, although minimal, system can go into production early on. The customer can then change its capabilities according to changes in the business, and also from learning from how the system is used in reality.

Often the most valuable features aren't at all obvious until customer have had a chance to play with the software. Agile methods seek to take advantage of this, encouraging business people to learn about their needs as the system gets built, and to build the system in such a way that changes can be incorporated quickly.

All this has an important bearing what constitutes a successful project. A predictive project is often measured by how well it met its plan. A project that's on-time and on-cost is considered to be a success. This measurement is nonsense to an agile environment. For agilists the question is business value - did the customer get software that's more valuable to them than the cost put into it. A good predictive project will go according to plan, a good agile project will build something different and better than the original plan foresaw.

### XP (Extreme Programming)

XP begins with five values (Communication, Feedback, Simplicity, Courage, and Respect). It then elaborates these into fourteen principles and again into twenty-four practices. The idea is that practices are concrete things that a team can do day-to-day, while values are the fundamental knowledge and understanding that underpins the approach. Values without practices are hard to apply and can be applied in so many ways that it's hard to know where to start. Practices without values are rote activities without a purpose. Both values and practices are needed, but there's a big gap between them - the principles help bridge that gap. Many of XP's practices are old, tried and tested techniques, yet often forgotten by many, including most planned processes. As well as resurrecting these techniques, XP weaves them into a synergistic whole where each one is reinforced by the others and given purpose by the values.

### Scrum

Scrum also developed in the 80's and 90's primarily with OO development circles as a highly iterative development methodology. It's most well known developers were Ken Schwaber, Jeff Sutherland, and Mike Beedle.

Scrum concentrates on the management aspects of software development, dividing development into thirty day iterations (called 'sprints') and applying closer monitoring and control with daily scrum meetings. It places much less emphasis on engineering practices and many people combine its project management approach with extreme programming's engineering practices. (XP's management practices aren't really very different.)

### Crystal

All crystal methods have three priorities: safety (in project outcome), efficiency, habitability (developers can live with crystal). They also share common properties, of which the most important three are: Frequent Delivery, Reflective Improvement, and Close Communication.

### Context Driven Testing

From the beginning it's been software developers who have been driving the agile community. However many other people are involved in software development and are affected by this new movement. One obvious such group is testers, who often live in a world very much contained by waterfall thinking. With common guidelines that state that the role of testing is to ensure conformance of software to up-front written specifications, the role of testers in an agile world is far from clear.

### Lean Development

### (Rational) Unified Process

RUP is a very large collection of practices and is really a process *framework* rather than a process. Rather than give a single process for software development it seeks to provide a common set of practices for teams to choose from for an individual project. As a result a team's first step using RUP should be to define their individual process, or as RUP calls it, a *development case*.

The key common aspects of RUP is that it is Use Case Driven (development is driven through user-visible features), iterative, and architecture centric (there's a priority to building a architecture early on that will last the project through).

Software Design Patterns

<https://www.geeksforgeeks.org/software-design-patterns/>

A design pattern provides a general reusable solution for the common problems that occur in software design. The pattern typically shows relationships and interactions between classes or objects. The idea is to speed up the development process by providing well-tested, proven development/design paradigms. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using design patterns, you can make your code more flexible, reusable, and maintainable.

1. 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.*

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.*

3. Behavioral

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*

# Design Patterns | Set 2 (Factory Method)

Factory method is a [creational design pattern](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/), i.e., related to object creation. In Factory pattern, we create objects without exposing the creation logic to the client and the client uses the same common interface to create a new type of object.

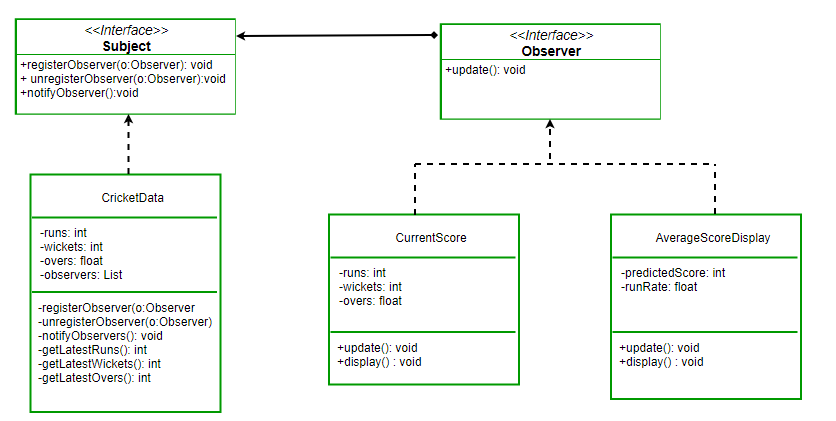
The idea is to use a static member-function (static factory method) that creates & returns instances, hiding the details of class modules from the user.  
A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn’t have tight coupling with the class hierarchy of the library.

***What is meant when we talk about*libraries*and clients?***   
A library is something that is provided by some third party that exposes some public APIs and clients make calls to those public APIs to complete their tasks. A very simple example can be different kinds of Views provided by Android OS. 

# Observer Pattern

Suppose we are building a cricket app that notifies viewers about the information such as current score, run rate etc. Suppose we have made two display elements CurrentScoreDisplay and AverageScoreDisplay. CricketData has all the data (runs, bowls etc.) and whenever data changes the display elements are notified with new data and they display the latest data accordingly

**Applying Observer pattern to above problem:**  
Let us see how we can improve the design of our application using observer pattern. If we observe the flow of data we can easily see that the CricketData and display elements follow subject-observers relationship.

**New Class Diagram:**  
*[](https://media.geeksforgeeks.org/wp-content/uploads/ObserverPatternSet-2.png)*

# Singleton Design Pattern

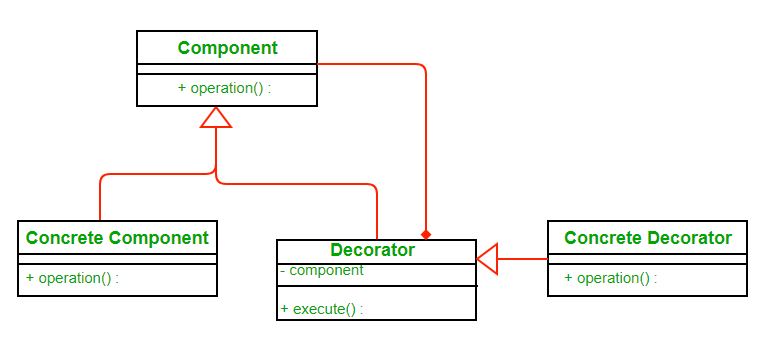
The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.  
**Definition:**   
*The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*   
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

# Decorator Pattern

# <https://www.geeksforgeeks.org/decorator-pattern-set-3-coding-the-design/?ref=lbp>

**Definition:**

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Class Diagram:**Image src: [Wikipedia](https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Decorator_UML_class_diagram.svg/600px-Decorator_UML_class_diagram.svg.png)

* Each component can be used on its own or may be wrapped by a decorator.
* Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).
* The ConcreteComponent is the object we are going to dynamically decorate.

**Advantages:**

* The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime**.**
* The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

**Disadvantages:**

* Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
* It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

# Strategy Pattern

# fighter2

It’s much cleaner. We took out some actions (which some characters might not perform) out of **Fighter**class and made interfaces for them. That way only characters that are supposed to jump will implement the **JumpBehavior.**

**What are the problems with above design?**

The main problem with the above design is code reuse. Since there is no default implementation of jump and roll behavior we may have code duplicity. You may have to rewrite the same jump behavior over and over in many subclasses.

**How can we avoid this?**

What if we made **JumpBehavior** and **RollBehavior**classes instead of interface? Well then we would have to use multiple inheritance that is not supported in many languages due to many problems associated with it.

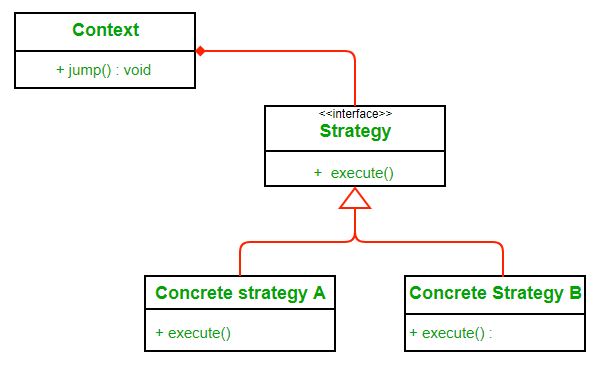
*Here strategy pattern comes to our rescue. We will learn what the strategy pattern is and then apply it to solve our problem.*

**Definition:**

Wikipedia defines strategy pattern as:

*“In computer programming, the****strategy pattern****(also known as the****policy pattern****) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern*

* *defines a family of algorithms,*
* *encapsulates each algorithm, and*
* *makes the algorithms interchangeable within that family.”*

***Class* Diagram:**

Here we rely on composition instead of inheritance for reuse. **Context**is composed of a **Strategy**. Instead of implementing a behavior the **Context** delegates it to **Strategy**. The context would be the class that would require changing behaviors. We can change behavior dynamically. **Strategy** is implemented as interface so that we can change behavior without affecting our context.

We will have a clearer understanding  of strategy pattern when we will use it to solve our problem.

**Advantages:**

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages:**

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

# Adapter Pattern

# <https://www.geeksforgeeks.org/adapter-pattern/>

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

***Definition:***

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

**Class Diagram:**  


The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

**Advantages:**

* Helps achieve reusability and flexibility.
* Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

**Disadvantages:**

* All requests are forwarded, so there is a slight increase in the overhead.
* Sometimes many adaptations are required along an adapter chain to reach the type which is required.

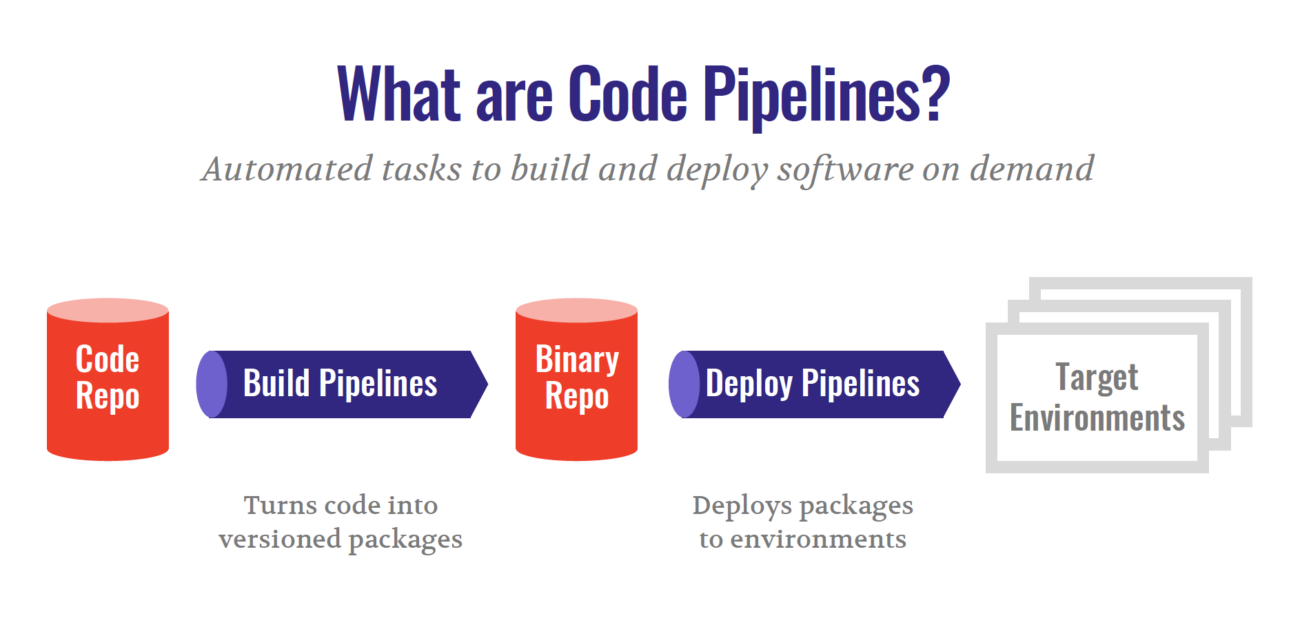
# Command Pattern

# Version control concepts

# <https://homes.cs.washington.edu/~mernst/advice/version-control.html>

# Pipeline Design Patterns for Continuous Delivery

<https://www.singlestoneconsulting.com/blog/7-pipeline-design-patterns-for-continuous-delivery/>



# DevOps Automation: Best Practices and Benefits

<https://www.sumologic.com/blog/devops-automation-best-practices-benefits/>

This decision is different for every DevOps team, of course. But in general, it's wise to prioritize the following processes:

* **CI/CD**: Rapid application development and delivery is at the core of DevOps. It's difficult to achieve this goal without automating your Continuous Integration/Continuous Delivery, or CI/CD, process.
* **Software testing**: Testing software prior to release is important, but your ability to do so manually is limited by the number of team members you have. Test automation tools like Selenium and Appium make it easy to run tests automatically, increasing the thoroughness of your testing routine.
* **Monitoring**: Keeping track of all of the components of a fast-moving DevOps environment is difficult to do manually, at scale. Automation tools that can monitor for availability, performance or security problems and generate alerts based on them help solve this challenge.
* **Log management**: The amount of log data generated by DevOps environments is vast. Collecting and analyzing all of that data by hand is not feasible for most teams. Instead, they rely on log management solutions that automatically aggregate and analyze log data for them.

# What is devsecops?

# <https://www.csoonline.com/article/3245748/what-is-devsecops-developing-more-secure-applications.html>

Devsecops is a culture shift in the software industry that aims to bake security into the rapid-release cycles that are typical of modern application development and deployment, also known as the devops movement. Embracing this shift-left mentality requires organizations to bridge the gap that usually exists between development and security teams to the point where many of the security processes are automated and handled by the development team itself.

Three key things make a real devsecops environment:

* Security testing is done by the development team.
* Issues found during that testing is managed by the development team.
* Fixing those issues stays within the development team.
* [Cross-site scripting](https://www.csoonline.com/article/3269028/what-is-cross-site-scripting-xss-low-hanging-fruit-for-both-attackers-and-defenders.html) (46.82)
* Out-of-bounds write (46.17)
* Improper input validation (33.47)
* Out-of-bounds read (26.5)
* Improper restriction of operations within the bounds of a memory buffer (23.73)
* [SQL injection](https://www.csoonline.com/article/3257429/what-is-sql-injection-how-sqli-attacks-work-and-how-to-prevent-them.html) (20.69)
* Exposure of sensitive information to an unauthorized actor (19.16)
* Use after free (18.87)
* Cross-site reques forgery (CSRF) (17.29)
* OS command injection (16.44)

The Veracode report shows that the most common types of flaws are:

* Information leakage (64%)
* Cryptographic issues (62%)
* CRLF injection (61%)
* Code quality (56%)
* Insufficient input validation (48%)
* Cross-site scripting (47%)
* Directory traversal (46%)
* Credentials management (45%)

Spring Java Framework

<https://www.baeldung.com/spring-tutorial>