Design Pattern - One of success keys for good software developers

*This essay's examples are written in Java. Please get the basic Java core and OOP skills first, or you can mark it somewhere in your device to read later*

What Is Design Pattern?

First, let back to the past, when programming is only inceptive structure. Our forerunners in the profession has developed the typical app several time, based on Object-Oriented programming languages. They meet the same problem,follow a conception and work in the same solution .

But noone make it more architectural and aggregate these solutions, or make it more 'easy-access' for other junior. So, the software development process often extended caused many negative impacts:

Consuming a lot of time, effort and money of customers, reducing the credibility of the business Decreasing the product lifespan

Easy lead to potential vulnerabilities or overlooked issues

Make difficulties in maintaining and upgrading the project in the future Causing challenges for developers to unify common conventions.

Over time, Libraries and frameworks appeared, supported much for making faster software development process and went a long towards a development model where we can just pick and choose componets and plug them right in.

But... they don't help us structure our own applications in ways that are easier to understand, more maintainable and flexible. That’s where Design Patterns come in.

In 1995, there are 4 senior developers who aggregate and write to the Design Patterns: Elements Of Reusable Object-Oriented Software - the "bible" of many software developer over time (The authors is call "Gang of Four" ). This book has architect 23 "forms" for developers to structure classes and objects to solve certain problems in many OOP languages. It encouraged the software engineers should study and structure the product in a more effective way. Some language's corporations like Oracle (Java) and Microsoft (C#) on supporting a lot of features for their language to conform with these "forms".

These "forms" is called "design pattern" - which we will explore today.

Characterists of Design Patterns

Like data structures and algorithms, it isn't in a special programming language , but it can implement easiest and powerest in the OO programming language (general and pure like Java, C# or support for OOP like Python, C++, JavaScript).

Each design patterns just is the pattern for one or some problem which it can solve effectively.

Definition :

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. It offer a best practice approach to support object-oriented software design, which is easier to design, implement, change, test and reuse.

How's a Design Pattern structured?

Each design pattern solve it's problem, and base on the problem, it suggest one or more design pattern which is suitable and approciated (in both conception and implementation).

Best practices of exploring Design Patterns is follow 3 issue of this design pattern : Problem : "What's it solve?"

UML : "How's it solve?"

Code example : "How to implement it?"

Types of Design Patterns

Here is my essay's content about the 5 most popular design patterns which I explored (I divided it into 3 groups by its type).

Creational Patterns

[Singleton](#_Singleton - One of a Kind Objects) [Factory Method](#_Factory - Return one object as your requirement What’s Factory?) Abtract Factory Builder Prototype

Structural Patterns

Adapter [Facade](#_Facade - make a simpler \“interface\”) Decorator Flyweight Proxy Composite Bridge

Behavior Patterns

Interpreter Template Method

Chain of Responsibility Command

Iterator Mediator Memento [Observer](#_Observer - Do you miss something interesting which happening?) Stategy State Visitor

Benefit of using Design Paterns

- It help you solve the problem you meet effectively.

- Improve your code Your code can be more reusable and easier to scale up whenever the project / product extend. In addition, it can improves your code's readability and maintainability.

- Less errors and less costing time for test: Design pattern help us to void potential problem, fallibility by using tested, proven development paradigms (Not to "Reinvent the wheel")

- Better communitcation DP provide developers the easier way to communicate about problems.

If used well, DP can speed up your development process and reduce the chance of errors.

Cristicism of using Design Patterns

- Target wrong problem.

Some developer desmonstrate that 16/23 design pattern can be eliminated in some other programming language, which can help developer avoid to "choose" wrong solution.

- Require extensive knowledge

Conquering design pattern is need more time to deep into, both the time for learning the OO programming language. It consume a large space for concepts inside the programmer’s head.

- Limit creativity and the power of programming language

Because of the "path thinking" whenever meet a problem solved before and use it anytime, you may limit your innovation

and cannot use programming language's features in the effectest way. You will scare to try another technology, no hope when move language or try a new and high risk way to solve the problem.

- Lack of formal foundations

DP are created by the experience of software developers, not the study in "ad hoc" way of computer scientists. In 1999, Gang of Four were "charged" by numberous computer scientist, architecture,... in OOPSLA show trial.

- Differ unsignificantly from other abstractions

The MVC , MVP and MVVM are use the same logic for implementation, just "rewrite" some feature in a layer. Used by several year, if you expect a new architecture will make "big update" in these, it wont't be possible, at least for now. The "path thinking" make both us and big corperation can't leave the shadow of safety and lazinessb.

*For more information, please research about the “****ANTI-PATTERN****” - which is the pattern that opposite the standard design pattern, developed by most scientist and high-experience engineers.*

Despite of some cristicism, desgin patterns have contributed a lot to the Software Engineering''s history, so I recommend you shoud learn it if you want to become a better engineer.

Special : [Architecture Pattern](#_Architectural Pattern)

References

[Dzone blog](https://dzone.com/articles/what-is-design-pattern) [Redgate forum](https://www.red-gate.com/simple-talk/blogs/why-following-design-patterns-is-a-bad-idea/) [SourceMaking](https://sourcemaking.com/design_patterns)

What's the worst issue junior usually meet when learn how to use it?

Learn quickly, learrn fast, not learn deep. It result to the "target wrong problem" and can cause other impacts.

Design Patterns is divided into layers, based on level of problem which you meet (medium, difficult), not on the coding level you have. So, please don't use them for show off your coding skill by using so many design patterns into a project. It may cause many opposite effects and bad affects to your work.

*REMEMBER : Design patterns are created to solve the complex problem, not to make the problem more complex.*

*QUOTE*

- When we haven't studied the religion, we see mountains as mountains, rivers as rivers.

- When we first learn the religion, we see that mountains are not mountains, rivers are not rivers.

- After studying the religion, we see that mountains are just rivers, rivers are just rivers.

# Singleton - One of a Kind Objects

## What’s Singleton ?

The Singleton Pattern - like its name : “singleton” - assures that at any one time there is **only one instance** of the Singleton object and provides a **global access point** to this instance. In other words, the Singleton’s purpose is to control object creation, limiting the number of objects to one only per application.

## UML

## 

**How to implement Singleton ?**

**public class** Duck {

**private** static Duck instance;

**private** Duck() {

}

**public** static Duck getInstance() {

**if** (instance == **null**) { instance = **new** Duck();

}

**return** instance;

}

*// other methods*

}

**public class** DuckShop {

**public** static void main(String[] args) { Duck \_duck1 = Duck.getInstance(); Duck \_duck2 = Duck.getInstance();

System.out.println(\_duck1.equals(\_duck2)); *// true*

}

}

## Where used it?

* Singeton pattern is used for logging, driver objects, caching and thread pool.
* It is used in some standard API of Java (java.lang.Runtime, java.util.Calendar, java.awt.Desktop)
* Some other DP used Singleton for its implementation: Abstract Factory, Builder, Facade …

Note : Implementation like above code is Eager Implementation. In reality, it can be destructured by reflection API in Java. You can search about some other implement ways (synchronize, double-check,…).

## Factory - Return one object as your requirement

## What’s Factory?

* Factory Method is a creational design pattern that Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

## Example Problem

* Its purpose is make your code loose coupled and more flexible. In the ordinary way, when you need to set up a CoffeeFactory you will type something like :

**public abstract class** Coffee { String typeName;

int voteOfUsers;

boolean isCoveredByLatteArt();

### ...

}

**public class** Americano **extends** Coffee { String typeName = "Americano";

### ...

}

**public class** Capucino **extends** Coffee { String typeName = "Capucino";

### ...

}

* But that make your code depend on the extensions of Coffee. If missing interface, you will be diﬀicult to recognize what is a Coffee.

Coffee americano = **new** Americano(); Coffee capucino = **new** Capucino();

* Now your sir want to extend coffee’s type to Espressino without change their code so much, because their CoffeeShop is still working. You can cre- ate again class Espressino extends Coffee, but you must change almost code which refer to them. Now, you are need for Factory Design Pattern.

## UML

## 

**Implementation**

**public interface** Coffee { String getCoffeeName();

}

Java 8 support \_enum\_ type, so I will use it for easier switch-case. You can use string, int or anythng other.

**public enum** CoffeeType { ESPRESSINO, CAPUCINO, AMERICANO

}

* Coffee implementation

**public class** Americano **implements** Coffee { @Override

**public** String getCoffeeName() {

**return** "Americano";

}

}

**public class** Espressino **implements** Coffee { @Override

**public** String getCoffeeName() {

**return** "Espressino";

}

}

**public class** Capucino **implements** Coffee { @Override

**public** String getCoffeeName() {

**return** "Capucino";

}

}

* My Factory

**public class** CoffeeFactory {

**private** CoffeeFactory() {};

**public** static Coffee getCoffee(CoffeeType coffeeType){

**switch** (coffeeType) {

**case** AMERICANO:

**return new** Americano();

**case** ESPRESSINO:

**return new** Espressino();

**case** CAPUCINO:

**return new** Capucino();

### default:

**throw new** IllegalArgumentException("Error, we don't supply this coffee type

}

}

}

* How My Factory be used

**public class** CoffeeShop {

**public** static void main(String[] args) {

Coffee capucino = CoffeeFactory.getCoffee(CoffeeType.CAPUCINO); System.out.println(capucino.getCoffeeName());

Coffee espressino = CoffeeFactory.getCoffee(CoffeeType.ESPRESSINO); System.out.println(espressino.getCoffeeName());

Coffee americano = CoffeeFactory.getCoffee(CoffeeType.AMERICANO); System.out.println(americano.getCoffeeName());

}

}

## Result

## 

**Where to use it?**

* When we need reduce dependability among modules (loose coupling)
* When we want to create a “easy-to-scale” product
* When we want to keep the logic of object creation process from user
* When we want to manage life cycle of objects created by Factory.
* This DP help to implement naming convention in team.

## Where used it?

* JDK : util.Calendar, NumberFormat,…
* Redefinement of it : Dependency Injection (Spring, .Net, Django)
* BeanFactory in Spring Framework, SessionFactory in Hibernate

# Facade - make a simpler “interface”

## What’s Facade Pattern?

* In an application, we often have to work on lots of object, which will have a close relationship later. If you use them with business logic, import some library… your code will be abstruse, diﬀicult for maintaining and expansion. (This is ‘tight coupling’ code.)
* Facade Pattern in structural pattern. It provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
* This pattern can be confused with Adapter Pattern, but each pattern has its purpose. Facade allow encapsuling source code for easier interact with other source.

## UML

## 

**How to implement it?**

* Facade often implement Client layer by Singleton.
* Account service

### package services;

**public class** AccountService {

**public** void getMail(String email) { System.out.println("Get account from " + email);

}

**public** void getNumberPhone(String numberPhone) { System.out.println("Get account from " + numberPhone);

}

}

Email Service

### package services;

**public class** EmailService {

**public** void sendMail(String email){ System.out.println("Send mail to " + email);

}

}

Payment Service

### package services;

**public class** PaymentService {

**public** void payByPayPal() { System.out.println("Paid by PayPal");

}

**public** void payByEBankin() { System.out.println("Paid by E-Banking");

}

**public** void payByMasterCard() { System.out.println("Paid by MasterCard");

}

**public** void payByCashing() { System.out.println("Paid by Cashing");

}

}

Shipping Service

### package services;

**public class** ShippingService {

**public** void freeShip() { System.out.println("Free Shipping");

}

**public** void standardShip() { System.out.println("Standard Shipping");

}

**public** void expressShip() { System.out.println("Express Shipping");

}

}

### package services;

### **public class** SmsService {

**public** void sendSms(String mobilePhone){ System.out.println("Send SMS to " + mobilePhone);

}

}

Facade Shop

**public class** ShopFacade {

**private** final static ShopFacade shop = **new** ShopFacade();

**private** AccountService accountService; **private** PaymentService paymentService; **private** ShippingService shippingService; **private** SmsService smsService;

**private** EmailService emailService;

**private** ShopFacade() {

accountService = **new** AccountService(); paymentService = **new** PaymentService(); shippingService = **new** ShippingService(); smsService = **new** SmsService(); emailService = **new** EmailService();

};

**public** static ShopFacade getInstance() {

**return** shop;

}

**public** void buyByPayPalWithFreeShip(String email) { accountService.getMail(email); paymentService.payByPayPal(); emailService.sendMail(email); shippingService.freeShip();

}

**public** void buyByCashingWithExpressShip(String email, String numberPhone) { accountService.getMail(email); accountService.getNumberPhone(numberPhone); paymentService.payByCashing();

emailService.sendMail(email); smsService.sendSms(numberPhone); shippingService.freeShip();

}

*/\**

*\* Other methods : ....*

*\*/*

}

Shop’s application

**public class** Shop {

**public** static void main(String[] args) {

ShopFacade shopFacade = ShopFacade.getInstance();

shopFacade.buyByCashingWithExpressShip(["mailtruong@huce.edu.vn"](mailto:mailtruong@huce.edu.vn), "012345678"); System.out.println();

shopFacade.buyByPayPalWithFreeShip("email@email");

}

}

## Result Usage

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.

# Observer - Do you miss something interesting which happening?

**What’s Observer Pattern**

* The Observer Pattern define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* It provide a solution for the case : some object need to be notified fre- quently about all changes happen in the other object.
* Benefit : increase extensibility, reusability and loose coupling between Observer and Subject (Define in UML)

# Case

* In a Big Tech, Our system need to tracking user data when they access to our website.
* When a user login, we must write log, checking IP address, block if their access is unknown… If their account expired, we must send a notification to their mail, set status to EXPIRED in database.

# UML

# 

# Code example

**public enum** Login\_Status {

SUCCESS, FAILURE, INVALID, EXPIRED

}

**public class** User { String email; String ip;

Login\_Status status;

**public** User(String email, String ip) {

**this**.email = email;

**this**.ip = ip;

}

}

**public interface** Monitor { void update(User user);

}

**public class** Logger **implements** Monitor { @Override

**public** void update(User user) {

**if** (user.status == **null** || user.status != Login\_Status.INVALID) { System.out.println("Wrote log for user : " + user.email + "\t" + user.ip);

}

}

}

**public class** Mailer **implements** Monitor { @Override

**public** void update(User user) {

**if** (user.status == Login\_Status.EXPIRED) {

System.out.println("Sent a mail to expired account : " + user.email);

}

}

}

**public class** IpProctector **implements** Monitor { @Override

**public** void update(User user) {

**if** (user.status == Login\_Status.INVALID) { System.out.println("Account invalid, blocked " + user.ip);

}

}

}

**public interface** Accessor {

void add(Monitor monitor);

void remove(Monitor monitor); void notifyToAllMonitor();

}

**public class** Service **implements** Accessor {

**private** User user;

**private** List<Monitor> monitors = **new** LinkedList<>();

**public** Service(String email, String ip) { user = **new** User(email, ip);

}

**public** String getUserName() {

**return this**.user.email;

}

@Override

**public** void add(Monitor monitor) {

**if** (monitors.add(monitor)) {

System.out.println("Add success " + monitor.getClass().getName() + ".service");

}

}

@Override

**public** void remove(Monitor monitor) {

**if** (monitors.remove(monitor)) {

System.out.println("Remove success " + monitor.getClass().getName() + ".service

}

}

@Override

**public** void notifyToAllMonitor() {

**for** (Monitor monitor : monitors) { monitor.update(user);

}

}

**public** void changeConnectionStatus(Login\_Status status) { System.out.println("\nChanged user status. " + user.status + " -> " + status); user.status = status;

notifyToAllMonitor();

}

*// Mock method. In real, we will call a list blocked IP / email not signed up from data*

**private** boolean isInvalidIP() {

**return** user.ip.equals("192.168.0.1");

}

**private** boolean isValidEmail() {

**return** user.email.equalsIgnoreCase(["duyhelloworld@mail.com"](mailto:duyhelloworld@mail.com));

}

**private** boolean isExpiredAccount() {

**return** user.email.equalsIgnoreCase(["expired@mail.com"](mailto:expired@mail.com)) || user.status == Login\_Stat

}

**public** void login() {

**if** (isInvalidIP()) {

user.status = Login\_Status.INVALID;

} **else if** (isExpiredAccount()) { user.status = Login\_Status.EXPIRED;

} **else if** (isValidEmail() && !isInvalidIP()) { user.status = Login\_Status.SUCCESS;

} **else** {

user.status = Login\_Status.FAILURE;

}

notifyToAllMonitor();

**if** (user.status == Login\_Status.SUCCESS && isValidEmail() && !isExpiredAccount() && System.out.println("Logged as " + user.email);

}

}

}

**public class** GoToWebsite {

**private** static Service newAccount(String email, String ip) { Service ser = **new** Service(email, ip);

ser.add(**new** Logger()); ser.add(**new** IpProctector()); ser.add(**new** Mailer());

System.out.println("\nAccount " + ser.getUserName() +

**return** ser;

}

**public** static void main(String[] args) {

Service acc1 = newAccount(["expired@mail.com"](mailto:expired@mail.com), "123.25.12.0"); acc1.login();

Service acc2 = newAccount(["ip-blocked@mail.com"](mailto:ip-blocked@mail.com), "192.168.0.1"); acc2.login();

Service acc3 = newAccount(["duyhelloworld@mail.com"](mailto:duyhelloworld@mail.com), "120.03.2.103"); acc3.login();

acc3.changeConnectionStatus(Login\_Status.EXPIRED); acc3.login();

}

}

## Result

## 

**Where used it?**

* Broadcast-type communication, when need change a object which refer to change the same at other objects
* JDK : mark interface Observerable, logic in many standard library.
* MVC Pattern : used to prevent Model from View (through Controller)

# Architectural Pattern

With Design Pattern, now we could develop an application more eﬀitively than before. But, have you ever wondered that, Design Pattern just help to strucutre a library, so what would structure something bigger, like systems or frameworks?

Yeah, the answer is Architectural patterns. There is a lot of confusion around that what is difference between Architectural pattern and Design pat- tern, so I just show you my opinion about this topic.

## What’s an Architectural Pattern?

* By Wikipedia :

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and min- imization of a business risk.

So unclear! - I think the best definition of **architectural pattern** that

an architectural pattern is a design pattern but has ability

to solve more problems than one ordinary. It can implement many design pattern inside it.

## What’s the difference?

* **Architectural Patterns** are broader in scope than **Design Patterns**.
  + **Design Patterns** provide very specific software related tasks whereas **Architectural Patterns** are solutions for business problems.
  + **Architectural Pattern** focuses more on the abstract view of idea while **Design Patterns** focuses on the implementation view of idea.
* Implementation of **Design Patterns** are defined at granular level where as **Architectural Patterns** are defined at high level. For example, dif- ferent implementations of Factory and Builder pattern might look very similar in different projects. But same at Architectural Patterns can be varied a lot in different projects.
* One **Architectural Pattern** can be implemented by using many **Design Patterns**. There is one to many relationship between **Architectural Patterns** and **Design Patterns**.

## Popular Architectural Patterns

* There are many popular architectural patterns (by formal documentation and by team developer rule …). In this article, I will make a brief introduction about 4 most common architectural patterns :

1. Client - Server

* The easiest pattern for understand and implement. 2 parties (1 server and multi clients) interact through a safe connection (almost TCP/IP, network).
* Advantage : using authorized access, client access via “UI” without com- mandline, distributed model, easy maintainmance.
* Disadvantage : overload server, make “single point of failure”.
* Popular in online application (email, share file, …)

1. MVC

[Model-View-Controller, legendary architecture pattern](#_The Legendary Architectural Pattern in developer community)

1. N-Layer Pattern

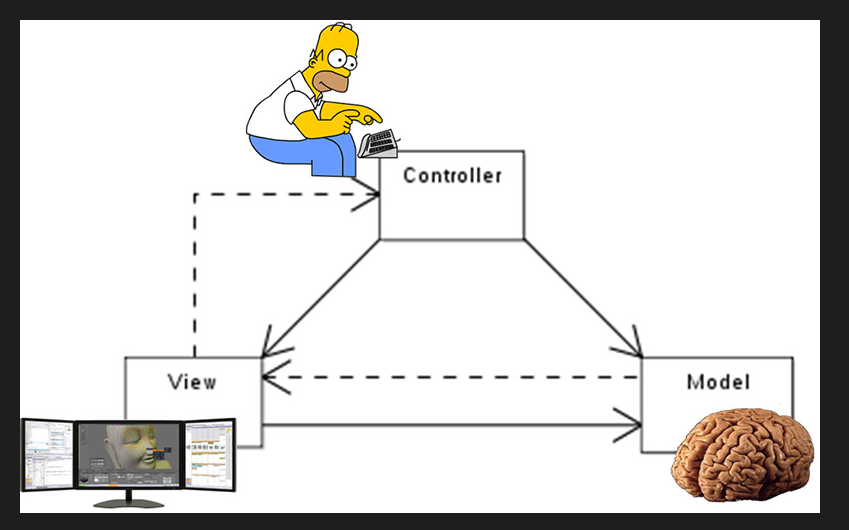
* Decompose programs into group of subtasks (Layer). Each layer provides services to the next higher layer.
* Most common type is 3-tier architect : > + Presentation layer (UI) : make UI and of logic UI. > + Business logic layer (Service) : responsible for all logic of the instruction. > + Data access layer (DA) : interact with database at below.
* Advantage : easy for maintain, manage large-scale product
* Disadvantage : Code logic is large; tasks only passing data must go through Service layer, make lower performance
* Popular in desktop app, web-app of commerce company.

1. MVVM

* A higher refinement of MVC, which Controller is replaced by View-Model. The specification of this pattern is the Flow one-way, which mean only View “know” about VM, and only VM “know” about Model.
* Advantage : Easy unit-test; communitation between designer and devel- oper; develop product fast, easy scale-up and maintainmance.
* Disadvantage : lower performance for small and ordinary applications; cost time for develop a good VM for larger applications.

## Conclustion : Architectural Patterns is useful for many tasks when develop “big-case” applications. If you want to become a better engineer, let explore about it, you will see a big impact on your mindset.

## The Legendary Architectural Pattern in developer community



**What’s it? History?**

* Definition : MVC (Model-View-Controller) is a software architecture pattern for creating user interfaces on computers. MVC divides an appli- cation into three interoperable parts that separate the way the information is internally processed and the information presented and received by the user.
* The MVC pattern was first introduced in 1979 by computer scientist Trygve Mikkjel Heyerdahl Reenskaug and first used in the programming language Small Talk.

## Structure

* MVC divide application into 3 building blocks, each block refer a fuction like its name: >
* + **M** : The application data resides in the model. The most important role of a model is connect Database, manipulate and pre- pare data for Controller use.
* + **V** : Users see the application data (from Model) through the view, however, the view can’t influence what the user will do with the data. View can’t get data directly from Model, this process must be throgh Controller.
* + **C** : The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action itypically a method call to the model. The response is shown in the view. In Controller, we should handle any business logic here.

## Flow

## 

* First, when application start, nothing happen.
* When user access application, user request View for interacting.
* Now View send the data it need to Controller
* Controller ask Model for this data.
* Model interact Database, get, maniplate data and response for Controller
* Controller render data to View (stardardize datetime, format name, …)
* View use them to display for user

### A quote : Why we need Controller and make lower performance when we can interact directly View - Model?

1. Follow SOLID

Controller follow S (Single Resibility) in SOLID - 5 principle of OOP. So, if you don’t create Controller, all logic will handle at Model, and we will make this pattern back to “Client-Server” - a worst architectural pattern for developing an non-commerce application.

1. Reusability and Flexibility

You can change model or the view without changing the other. You can reuse all code in 2 above part just by change controller. The program is concenrated in Controller

1. Easy testing

We can test each part and won’t affect much to application. Moreover, we can test app when it’s running (I don’t recommend this action)

1. Higher quality Product

The idea of original MVC is to separate out the different “chunks” of logic so that they don’t overlap. By follow it, the less these areas of concern overlap, the easier it is to do interesting things with them.

## Pros and Con of MVC

1. Advantage

- **Faster the development processing** : Divide task into 3 team : Frontend (View), Backend (Controller) and Backend+System Designer (Model). The process working parallel.

- **Increase code reusability, lower complexibility** : Keep logic apart help increase this code’s reusability and simpler.

- **Higher Maintainance** : Lower Dependency in code logic will make easier for maintaining and scale the application.

- **Maxium usage of a Server:** This architect save much bandwith and space compare to the others.

- **Easier for detect error** : We only need to test each part for find and fix bug in this part.

- **Simple structure** : Support junior, some developer who aren’t good at system design and technical can use and follow.

- **Support for SEO (Search Engine Optimization)** : Easy create SEO code, SEO tag and URL for abtract user traﬀic.

- **Fit with small, normal application (size)**

- **Used commonly in Framework and Library** : Optimize the Framework performance when developing.

1. Disadvantage

* **Diﬀicult for durable maintainance** : Controller logic will larger when- ever support a new application’s feature
* **Controller and View has ‘tight coupling’** : The more View changed, the more Controller affected
* **Unconform for small project** : Implement with small application is

over-complex. This instruction only mak benefit when the app is bigger (1 of the worst affection of application development, when life cycle of an app may be very short)

* **Diﬀicult for navigate code** : Navigate flow code from UI to data is harder when follow this AP.

### Need to know multiple technogies

**Code example**

**public class** ModelStudent {

**private** static long count = 0;

**public** int MAX\_LENGTH\_NAME = 10;

**private** Long id;

**private** String name;

**private** LocalDate updatedDate;

**public** boolean save() {

**if** (name.length() <= MAX\_LENGTH\_NAME) { **this**.updatedDate = LocalDate.now(); **return true**;

}

### return false;

}

**public** ModelStudent(String name, LocalDate updatedDate) {

**this**.id = ++count;

**this**.name = name;

**this**.updatedDate = updatedDate;

}

*// Getter, setter*

}

**public class** ControllerStudent { **private** ModelStudent model; **private** ViewStudent view;

**public** void initial() {

model = **new** ModelStudent("unknown", LocalDate.of(2023, 2, 12)); view = ViewStudent.getView();

}

**public** void showDetail() { view.showDetailPage(model);

}

**public** void rename(String newName) { model.setName(newName);

}

**public** void save() {

boolean isSuccess = model.save();

**if** (isSuccess) { view.showUpdatedPage(model);

} **else** {

view.showErrorMessage("Unable to create user with name's length greater than "

}

}

}

Singleton used for View

**public class** ViewStudent {

**private** static ViewStudent instance;

**private** ViewStudent() {} ;

**public** static ViewStudent getView() {

**if** (instance == **null**) {

instance = **new** ViewStudent();

}

**return** instance;

}

**public** void showHome() {

System.out.println("Home page of Application.");

}

**public** void showDetailPage(ModelStudent student) { System.out.println("Student: "); System.out.println("+ Id: " + student.getId()); System.out.println("+ Name: " + student.getName()); System.out.println("+ Updated date: " +

student.getUpdatedDate().format(DateTimeFormatter.ofPattern("dd/MM/yyyy"))) System.out.println(" ");

}

**public** void showUpdatedPage(ModelStudent student) { System.out.println("Saved ID(" + student.getId() + ") successfully"); showDetailPage(student);

}

**public** void showErrorMessage(String messageError) { System.out.println("Something wrong!!!\n" + messageError);

}

}

**public class** MVCExample {

**public** static void main(String[] args) {

ControllerStudent controller = **new** ControllerStudent(); controller.initial();

controller.showDetail();

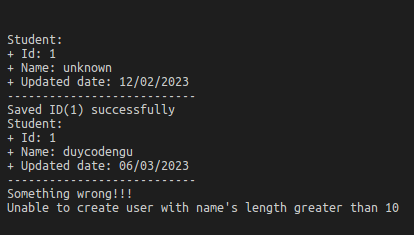
controller.rename("duycodengu"); controller.save();

controller.rename("MVC Pattern is the best, all other is trash!"); controller.save();

}

}

**Result**



## References

* [FreeCodeCamp](https://www.freecodecamp.org/news/mvc-architecture-what-is-a-model-view-controller-framework/" \l "%3A~%3Atext%3DThe%20MVC%20pattern%20was%20first%2Cthe%20programming%20language%20Small%20Talk)
* [Wiki MVC’s History](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)
* [Linkedin - Blog](https://www.linkedin.com/pulse/architectural-pattern-vs-design-praveen-kumar-kushwaha/)

All above is my Essay about “Explore Design Patterns and Implemeted at least 1 DP in Java”.

Hope you’ve got a brief exploration about one of the most important thing

every IT fresher have to learn when work in Software Engineering.

Thank for reading!

THE END