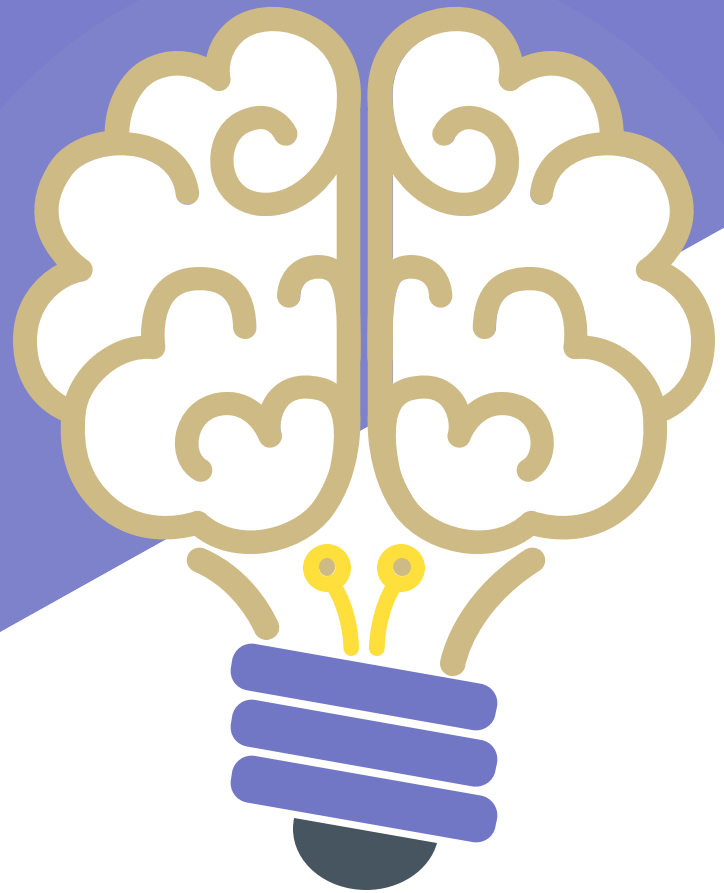
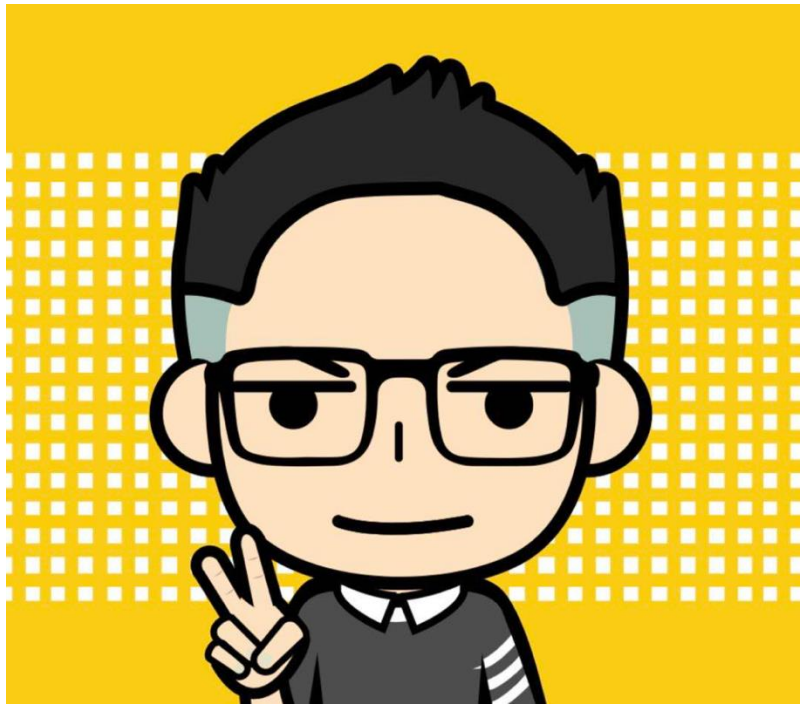
Software Architecture in User Application

Architecture patterns



于洋 @ ByteDance



于洋

14年互联网老兵

@ 字节跳动 （抖音集团）

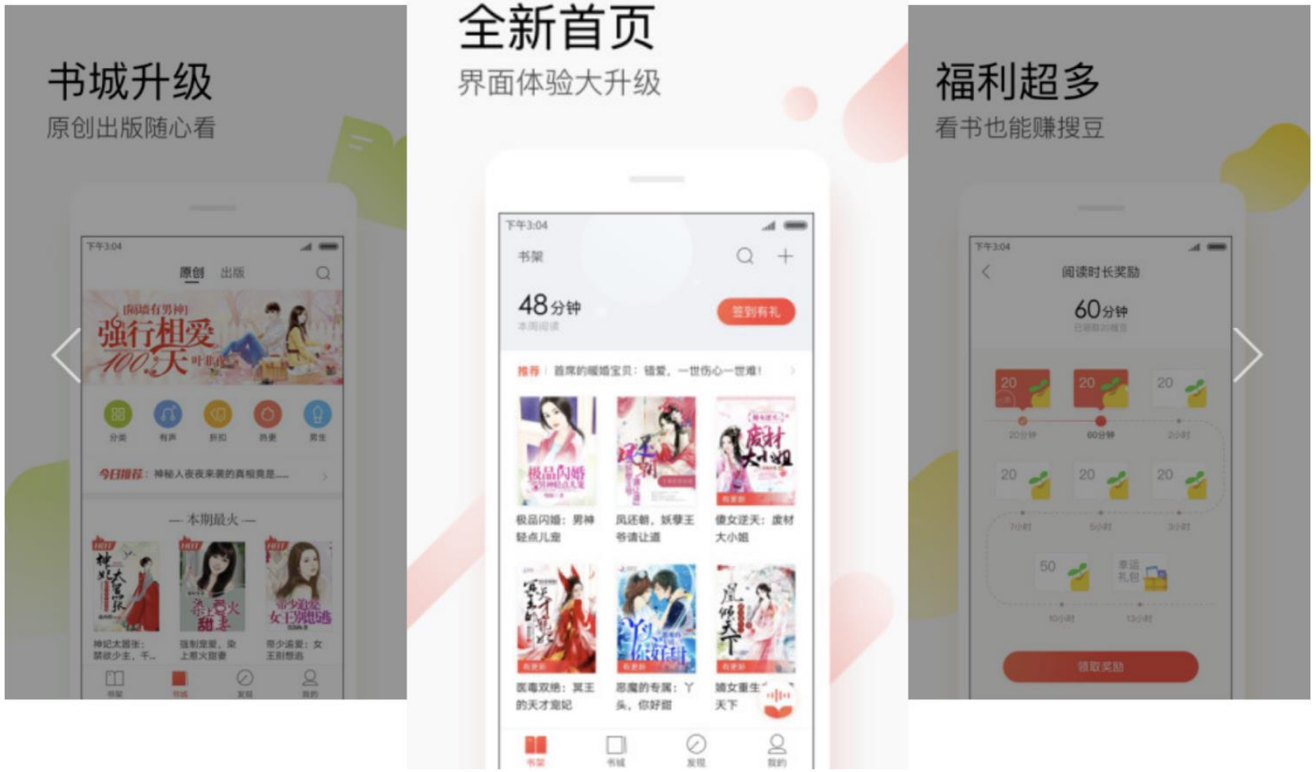
今日头条、西瓜视频、番茄小说、番茄畅听、红果短剧、皮皮虾

商业化客户端技术负责人


SOSO 搜搜
搜 搜 更 懂 你
网页 图片 视频 音乐 问问 新闻 地图 更多»
搜搜


搜狗搜索


123
搜索


书城升级
原创出版随心看
全新首页
界面体验大升级
福利超多
看书也能赚搜豆


23:12
App Store
搜全网 搜索全网小说和我的书架
书架 推荐 分类 书荒互助 故事 福利 男生
每日精选12本
换一换 3/12
遮天：无情天帝
8.7分
帝路争锋，星空中，帝路上，神体尽出，而这个时代，却有两位凡体力压诸多天骄。“...
高分佳作 轻小说 · 连载中 · 杀伐果断
收尸档案
9.8分
自古以来，人死留尸。但很少有人知道，尸体里面其实藏有很多不为人知的秘密。
20.4万书友在读 悬疑推理 · 连载中
神医毒妃
9.7分
前世，她是侯府嫡长女，倾尽一切助夫君登上皇位，换来的却是被囚暴室一尸两命！...
2.3万加书 古代言情 · 已完结 · 重生
推荐榜 高分榜 完本榜 全部
全部 都市 升级流 系统流 轻小说 修仙
女总裁的超级兵王
军事 · 已完结 · 推荐指数99
16.9万书友在读
2 大明最后一个
3 校园超级高手
首页 免费小说 直达 文件 我的

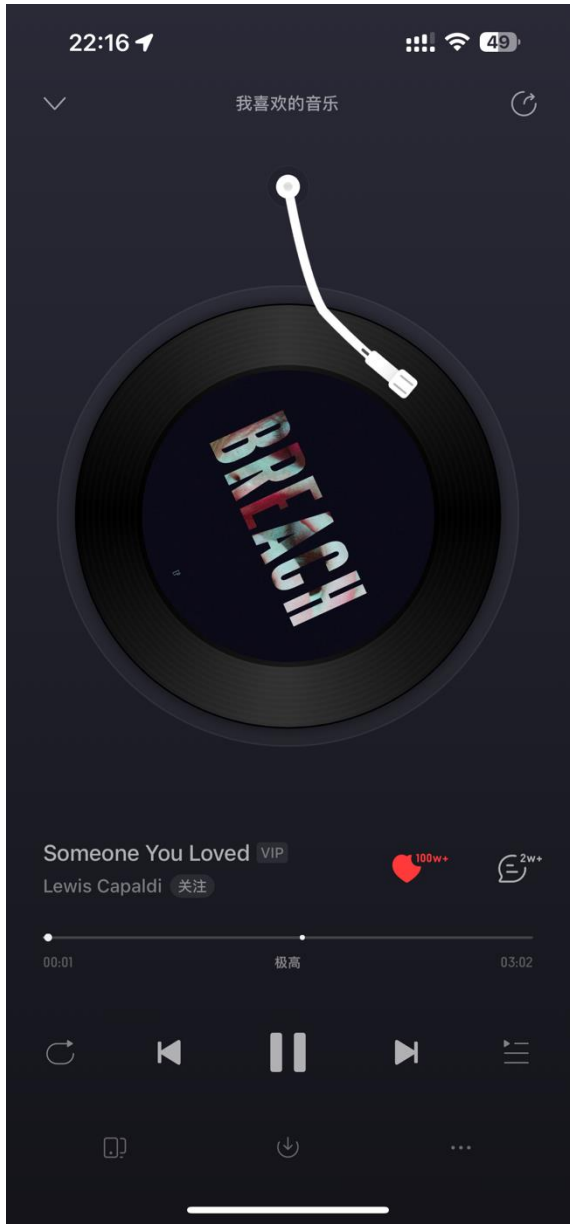
搜狗阅读

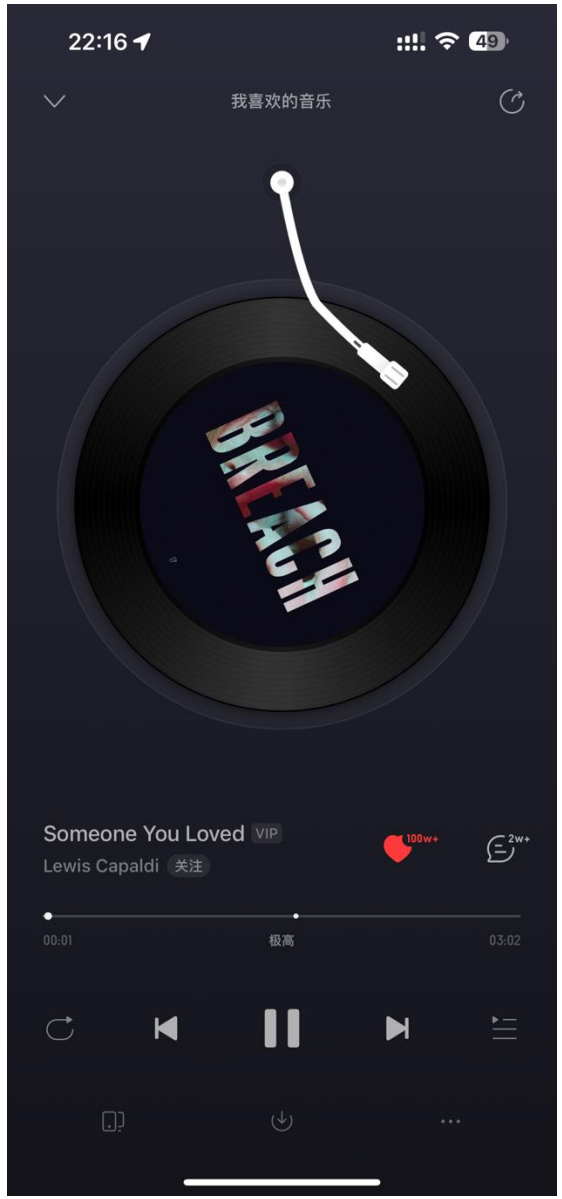
QQ浏览器


软件架构师手册
软件架构师手册
通过贯彻有效的架构理念成为一名成功的软件架构师
约瑟·因格纳（Joseph Ingeno） 著
米庆 于洋 译
东南大学出版社

content

Example:
A Music Player




22:16
我喜欢的音乐
BREACH
Someone You Loved VIP
Lewis Capaldi 关注
100w+
2w+
00:01
极高
03:02

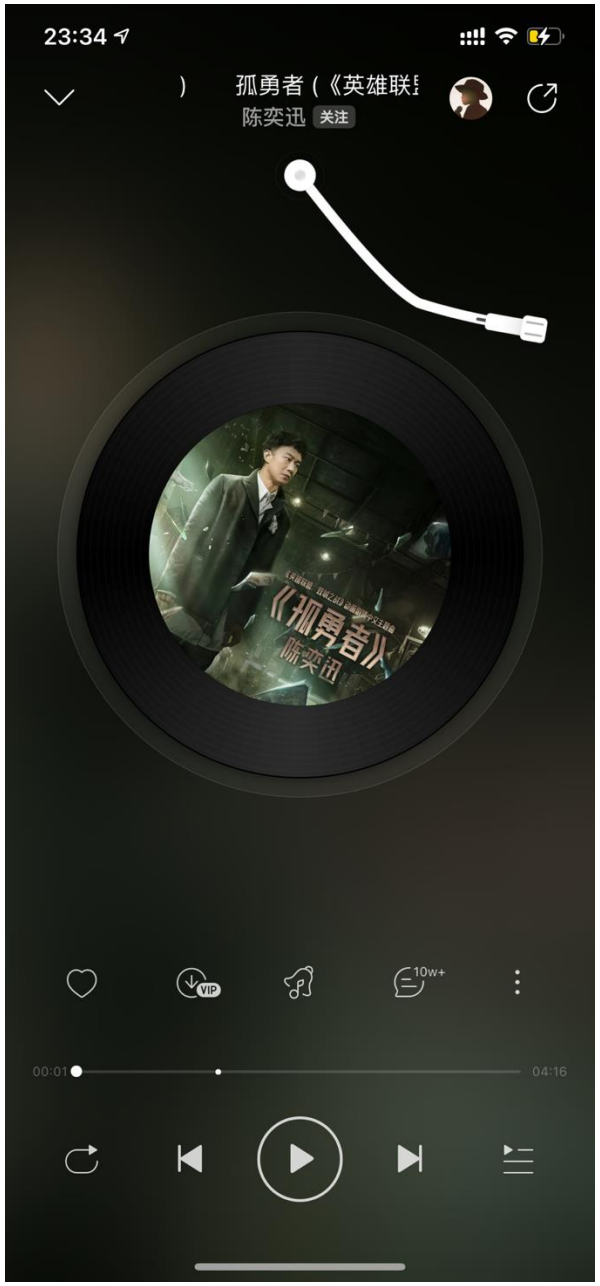

22:14
QQ音乐
疲惫的爱
50w+
关注
1k人在听
HQ
VIP
原唱
视频
我怎么能让你承受孤独
99+
On
999+
00:21
04:45

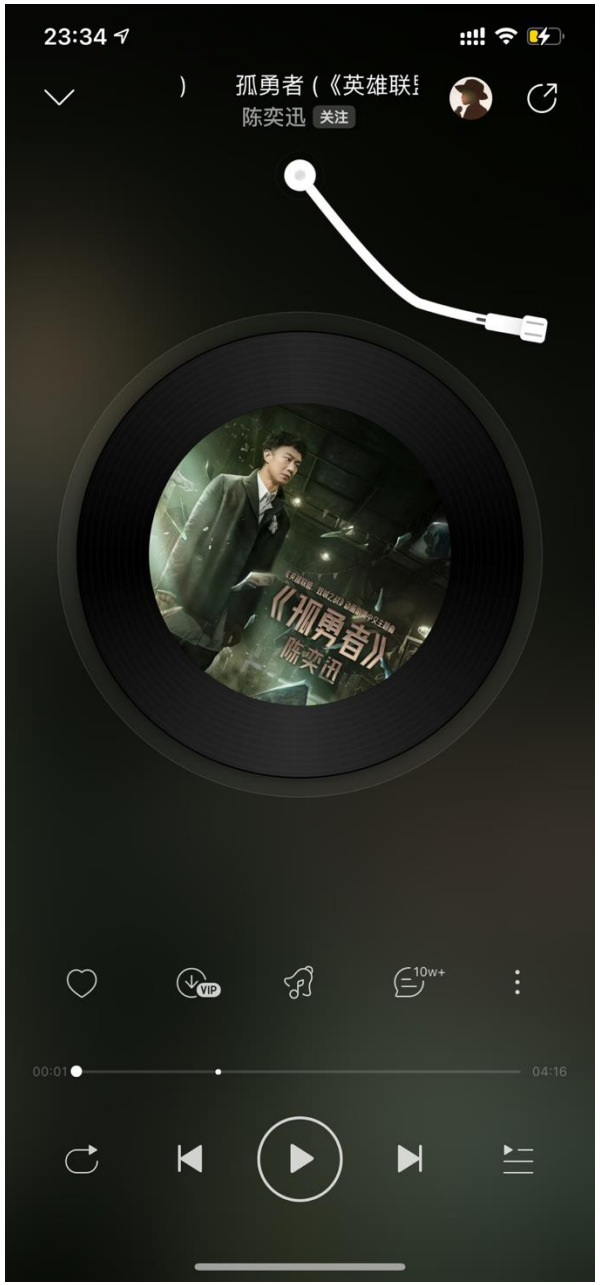
How many CODES need to be REWRITTEN

How many TESTS need to be REDONE

Architecture

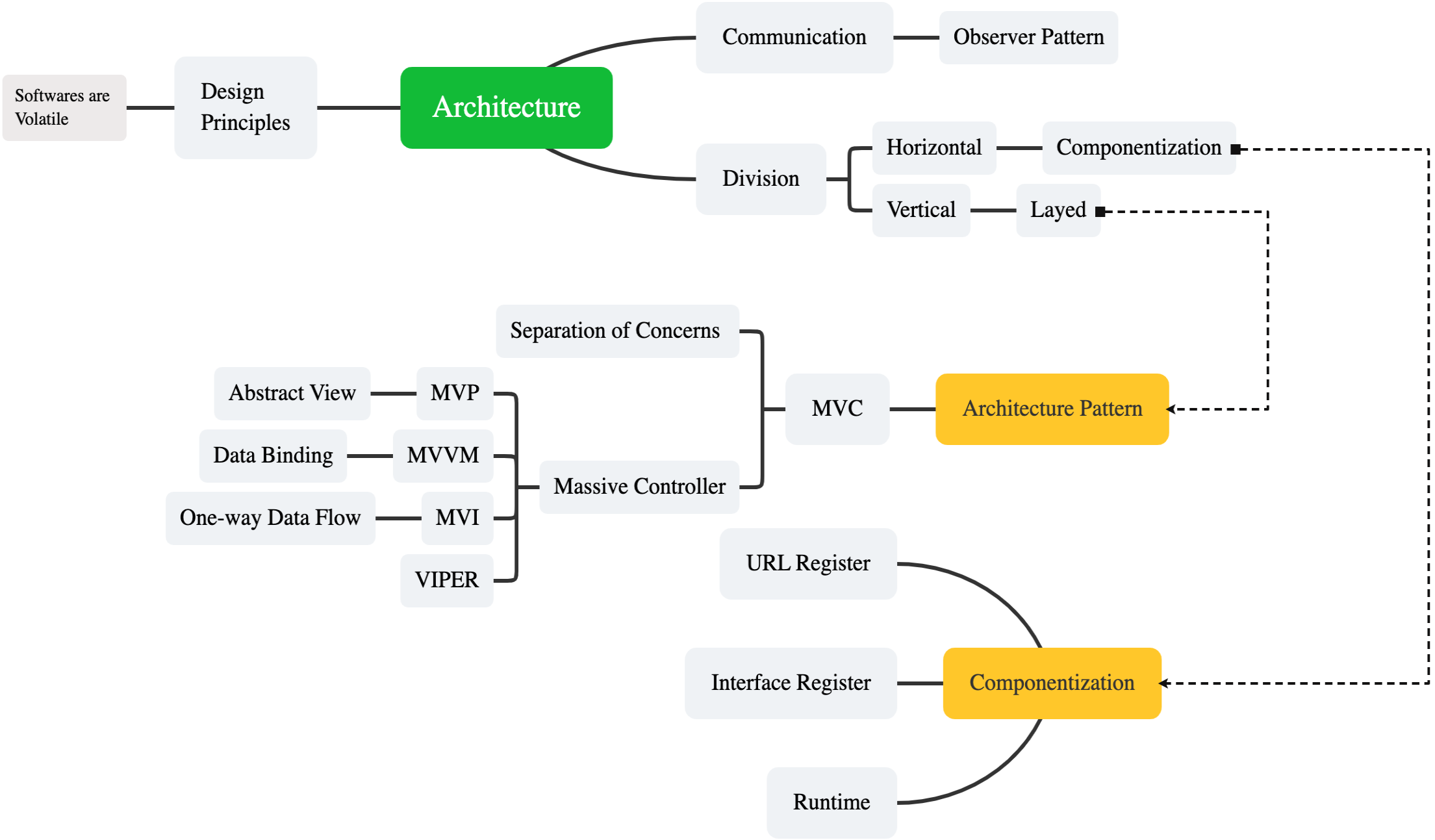
Example: Next Track





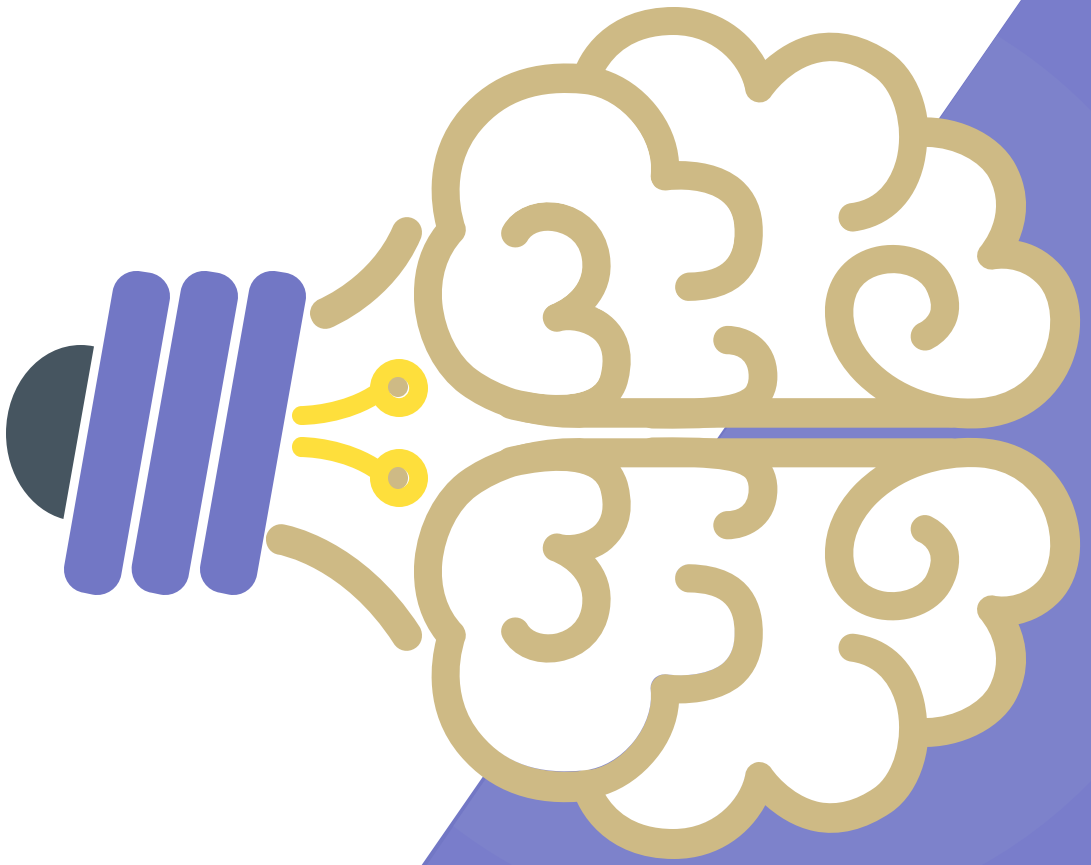
Click “Next Track” What would happen ?

1. Load successfully
2. Load failure
3. Loading
4. Always Clickable?


Architecture
Communication
Observer Pattern
Design Principles
Softwares are Volatile
Division
Horizontal
Componentization
Vertical
Layed
Architecture Pattern
MVC
Separation of Concerns
Massive Controller
MVP
Abstract View
MVVM
Data Binding
MVI
One-way Data Flow
VIPER
Componentization
URL Register
Interface Register
Runtime

/01

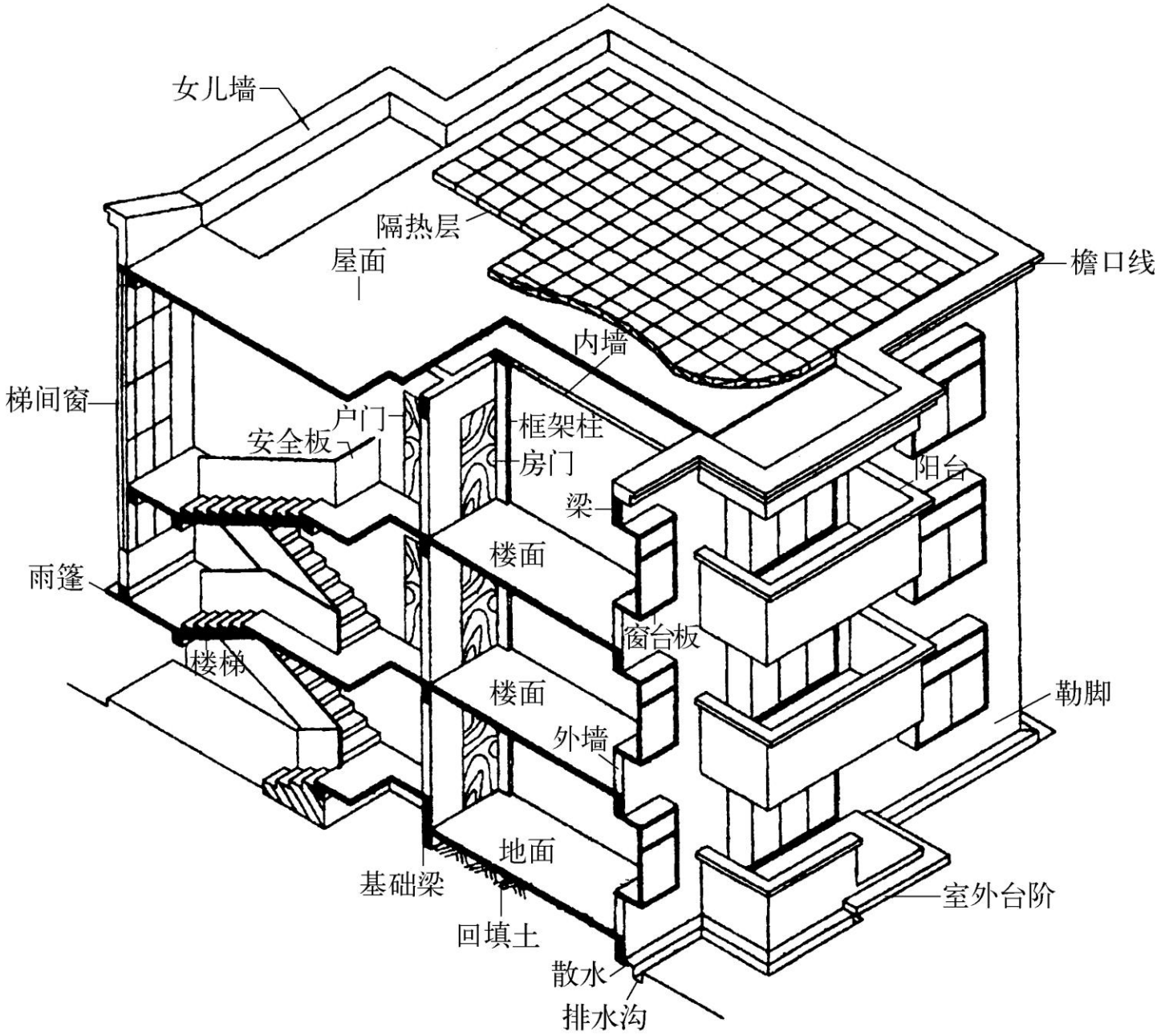
Why Architecture



What

Is

Architecture

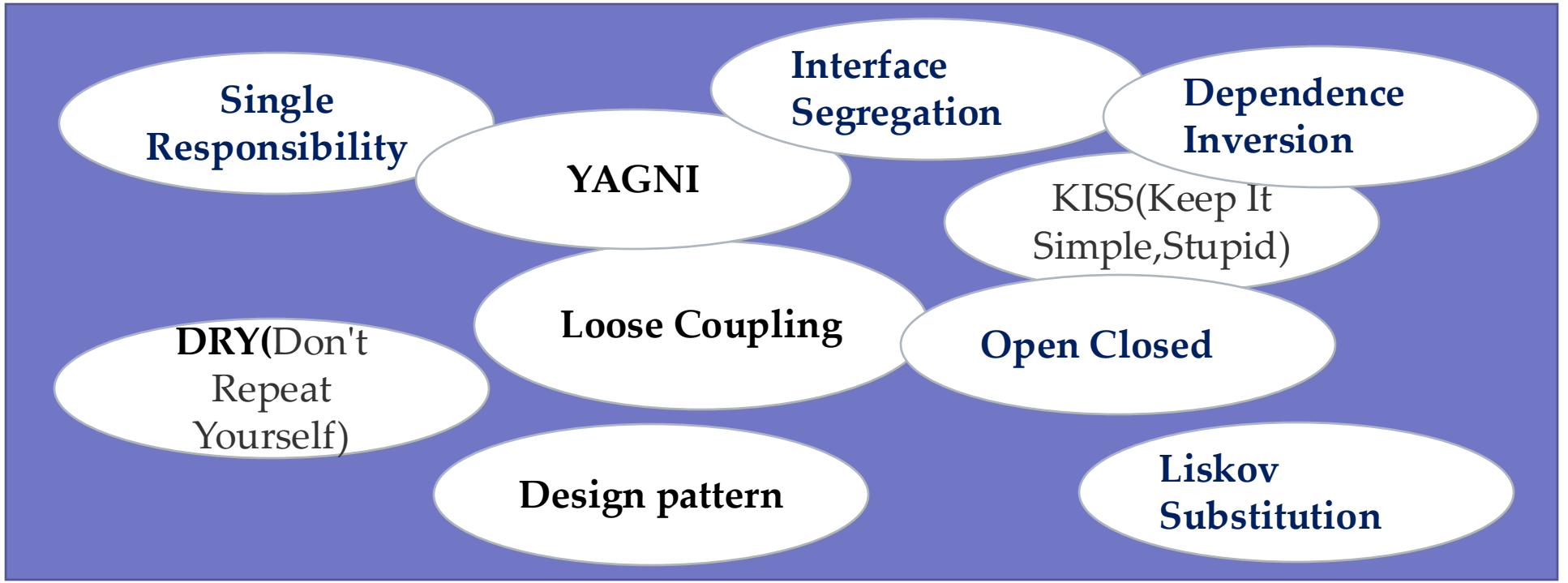



Change is inevitable.
Change is constant.

Why Architecture

Why Architecture **?**

When we have



Why Architecture

Methodology

中线理论
正面朝敌

Documentation

Communication

小念头
寻桥
标指

Prediction

埋身搏击

Why Architecture

Methodology

- Design principle.

Division

- Different layer

Prediction

- Quality attributes

Documentation & Communication

- Model, view, controller, presenter, viewmodel

Who Need

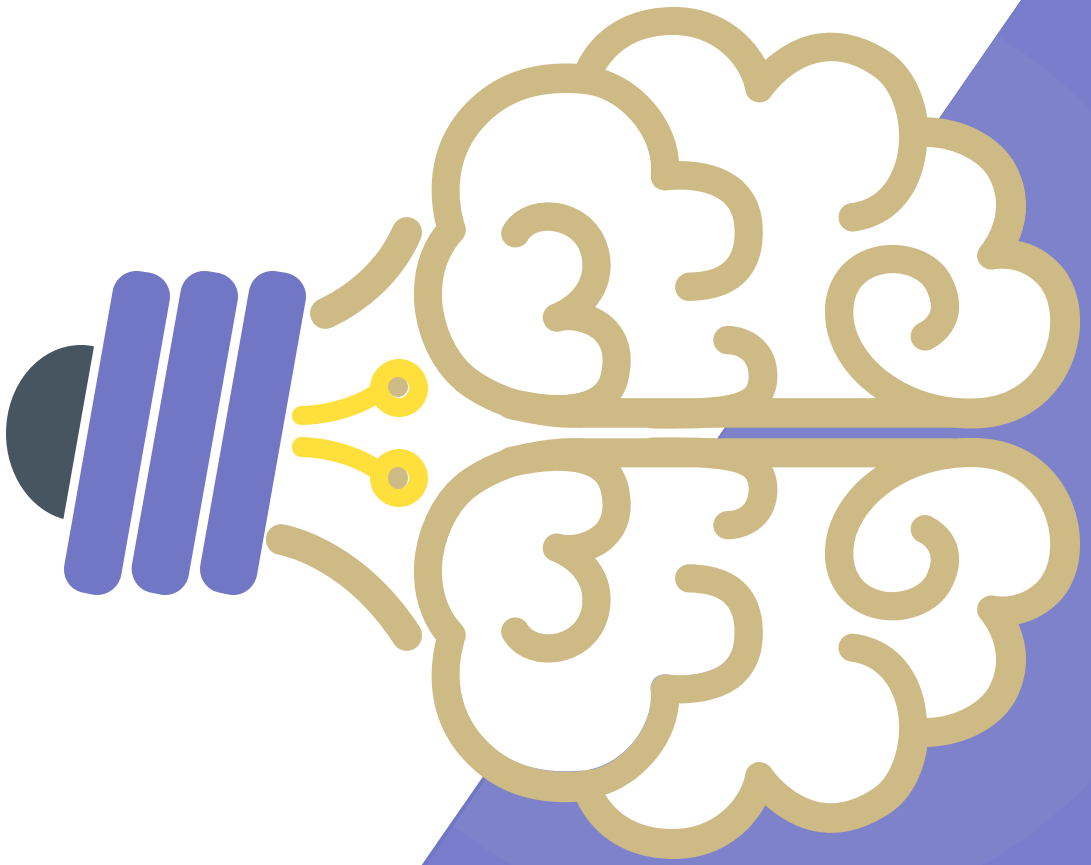
Architect?

Every Developer

/02

Architecture Principle

Layered, Observer



Issues

1.How to divide a system into
different parts

2.How those parts
communicate
with each other.

Features of a *Good Architecture Pattern*

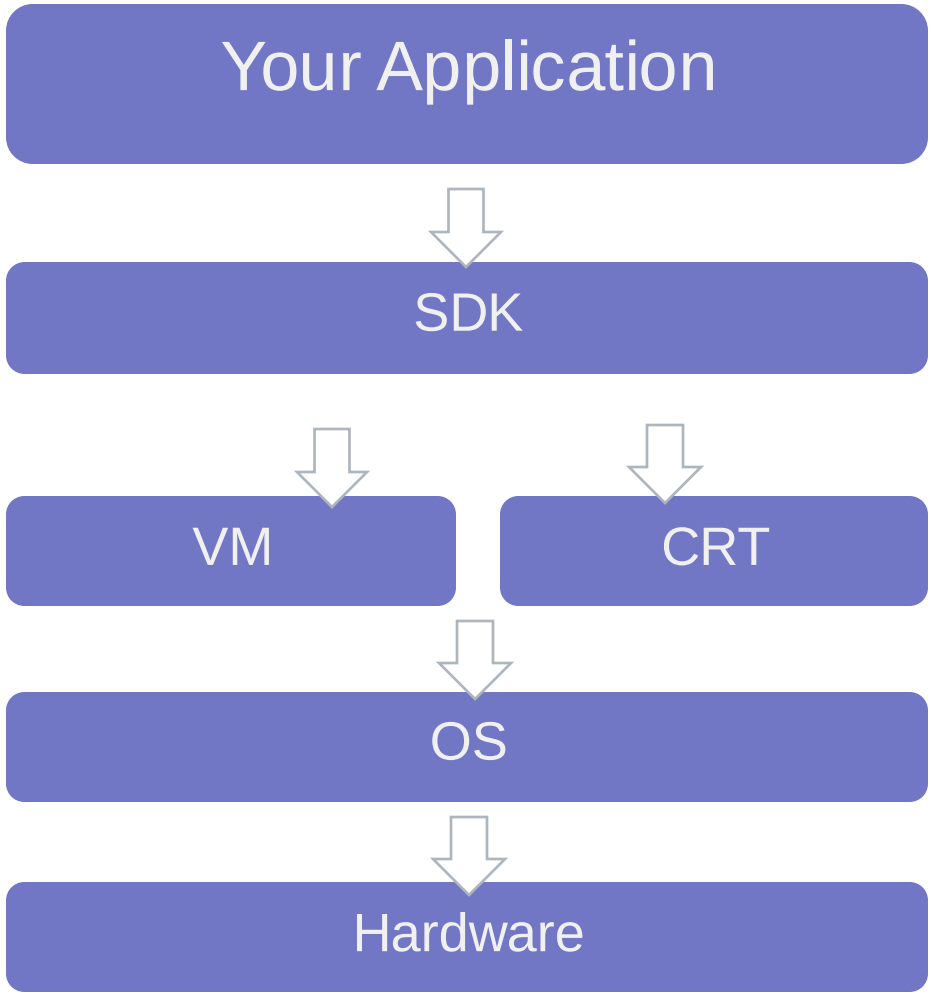
1.Balanced **distribution** of responsibilities among entities with strict roles.

2.Testability usually comes from the first feature (testability is easy with appropriate architecture).

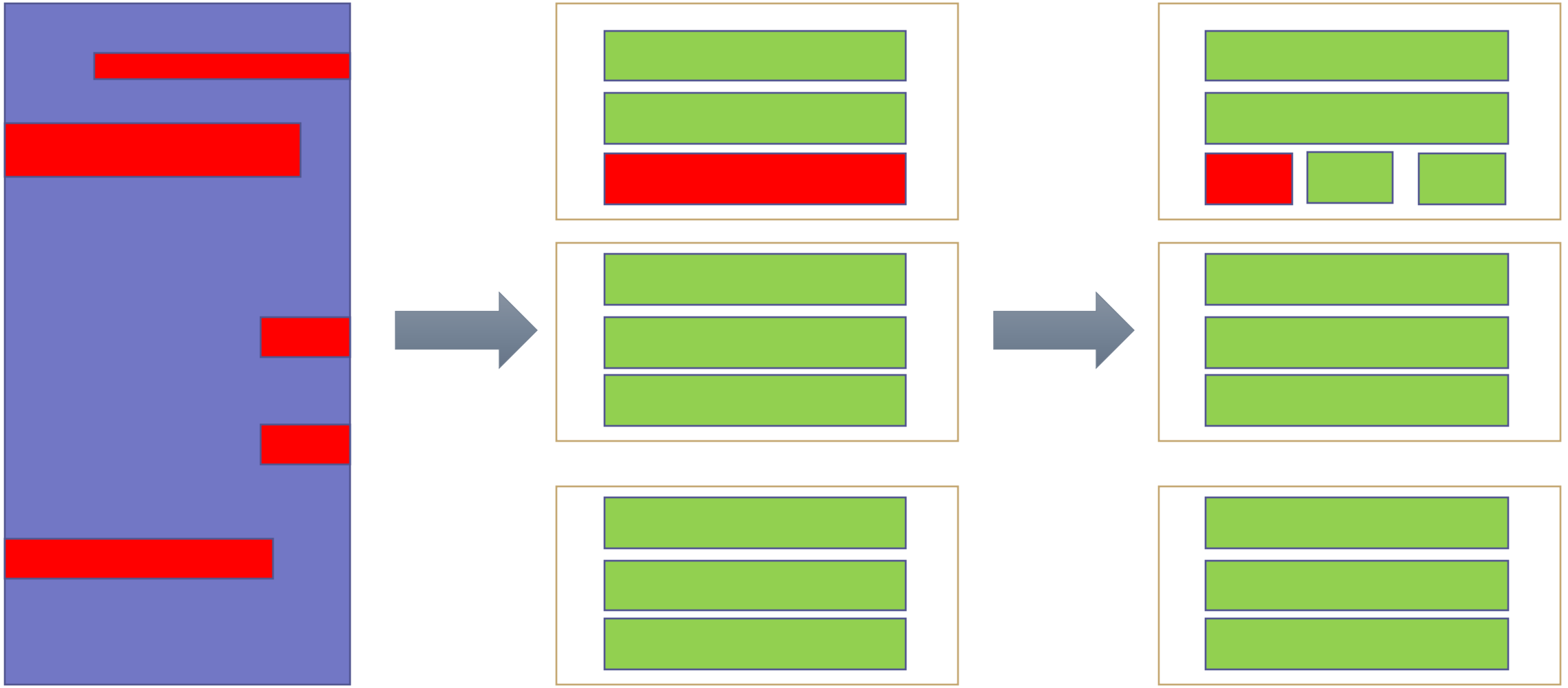
3. Ease of use and a low maintenance cost.

Layered Architecture

Computer System：



Layered Architecture



Layered Architecture

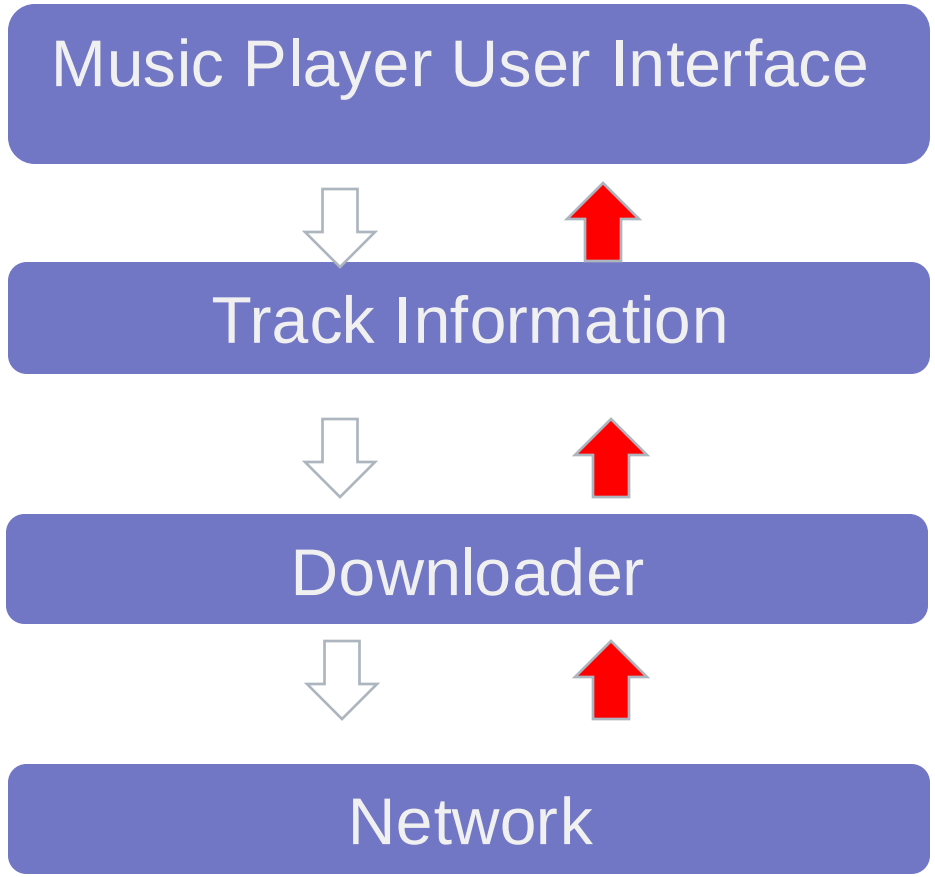
- Layered reduces complexity by achieving a **Separation of Concerns (SoC)**. Each layer is independent and you can understand it on its own without the other layers. Complexity can be abstracted away in a layered application, allowing us to deal with more complex problems.

- Each layer requires a particular skill set and suitable resources can be assigned to **work on each layer**. For example, UI developers for the presentation layer, and backend developers for the business and data layers.

- Partitioning the application into layers and using interfaces for the interaction between layers allows us **to isolate a layer for testing** and either mock or stub the other layers.

- Applications using a layered architecture may have **higher levels of reusability** if more than one application can reuse the same layer.

Layered Architecture

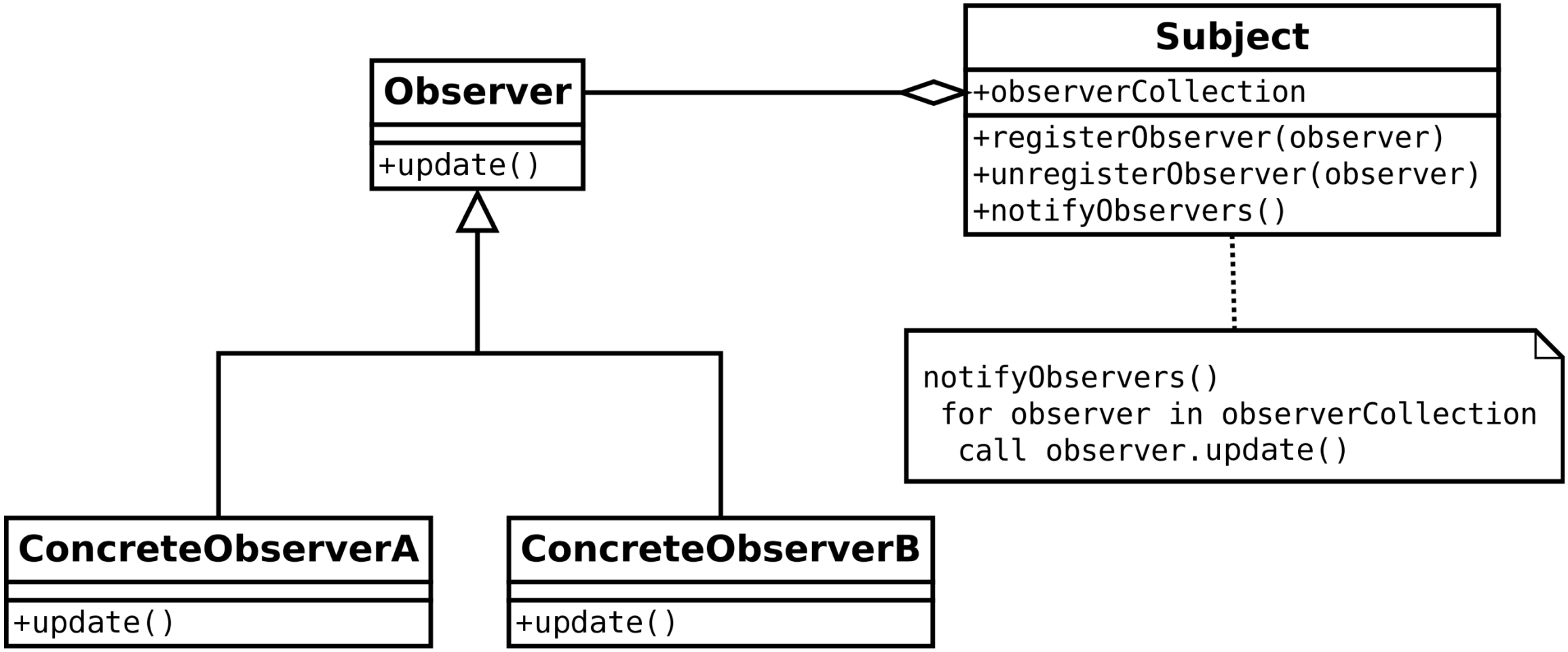


Design Pattern

Observer Pattern

The **observer pattern** is a software design pattern in which an object, called the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods.

Observer Pattern



Observer Pattern

Player View

?

Downloader

Downloader model

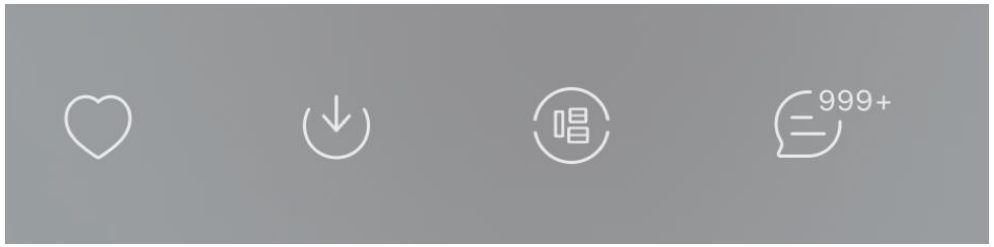
Some Action →
Downloader model start a download process →
Downloader model complete a download process →
Download Button in player view turn to downloaded state

Downloader *Notify* Player View

```
class Downloader {
    PlayerView playerView;
    void startDownload(Track track){
        //download file from internet...
    }
    void onTrackDownloadFinish(Track track){
        this.playerView.changeDownloadButton(true);
        Account.singleton.quotaChange(-1);
    }
}

class PlayerView {
    void changeDownloadButton(bool downloaded){
        //change view
    }
}
```

```
class Account{
    void quotaChange(int diff) {
        //do something
    }
}
```



Downloader → PlayerView
Downloader → Account

WHAT IF
There is 10+ target to notify?

Player View *Observe* Downloader

```
interface DownloadObserver {
    void downloadFinish(Track track);
}

class Downloader {
    List<DownloadObserver> observers;
    void addObserver(DownloadObserver ob){
        this.observers.add(ob);
    }
    void onTrackDownloadFinish(Track track){
        for( DownloadObserver ob in observers){
            ob.downloadFinish(track);
        }
    }
}
```

```
class PlayerView implements DownloadObserver{
    public void downloadFinish(Track track) {
        // change view
    }
}
```

Advantage：

- Observable subject less care about observer，more reusable
- **Reverse Dependency**

Various implementations:

- C: Function pointer,
- std::function,
- Objc: block, delegate
- Java: interface,
- C++/python: lambda

Observer Pattern implementations

CallBack

C:	Function Pointer
C++:	std::function
Objective-C:	block
JavaScript:	function
Swift/python:	Lambda

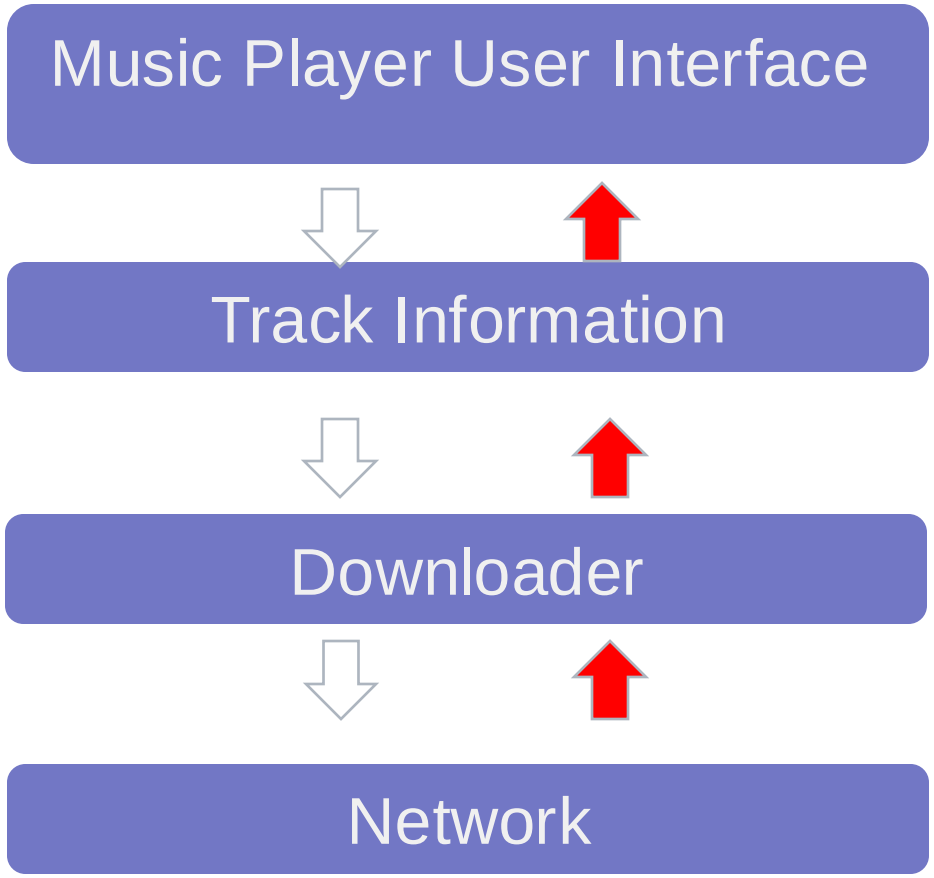
Parent Class

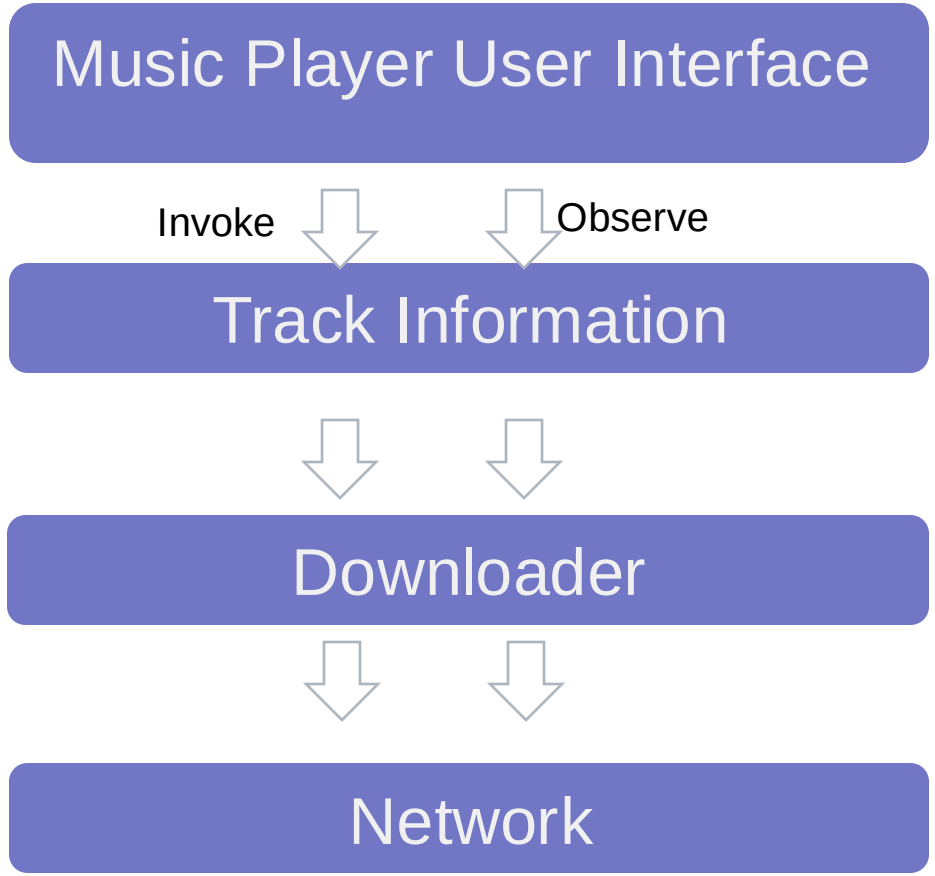
C#:	Abstract Class
Java:	Interface
Objective-C:	Protocol
C++:	Virtual Base Class

Message

iOS:	Notification
Android:	Bradcast
Windows:	Win32 Message

Layered Architecture



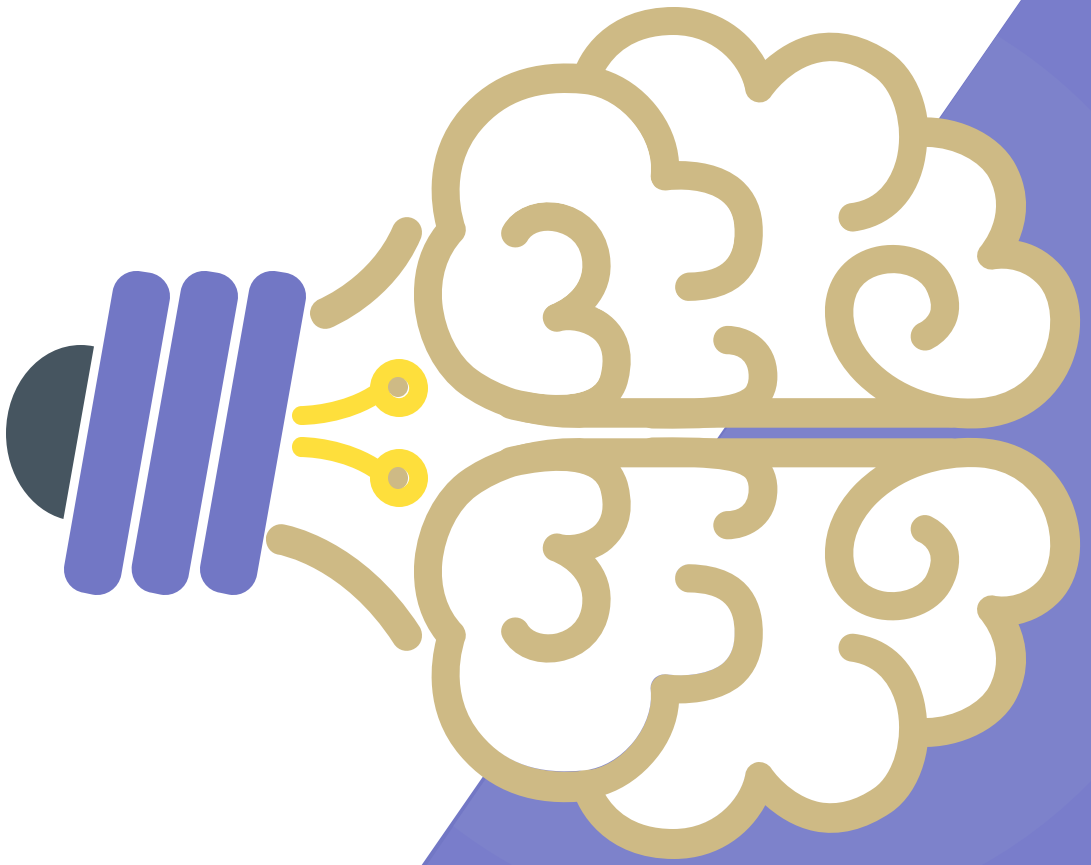


/03

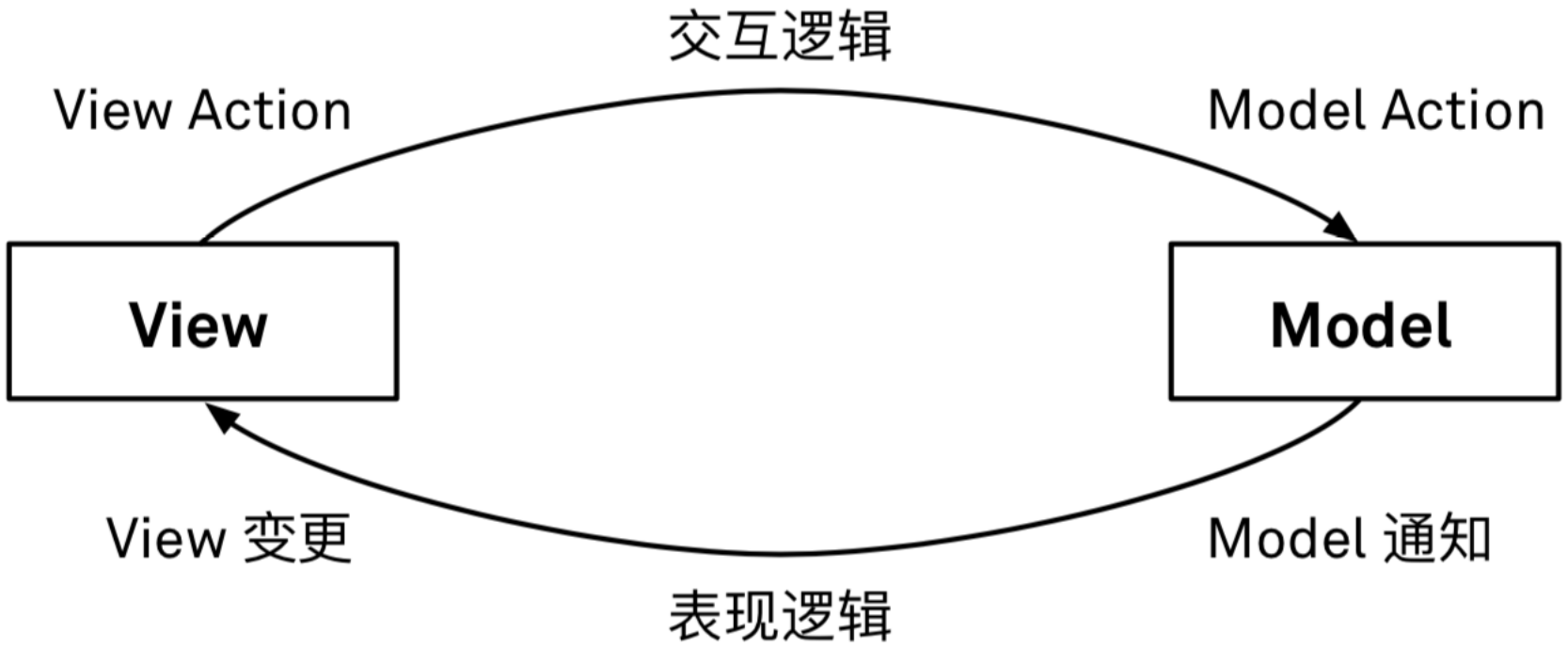
Architecture Patterns

Layered.

MVC，MVP，MVVM



Application’s Feedback

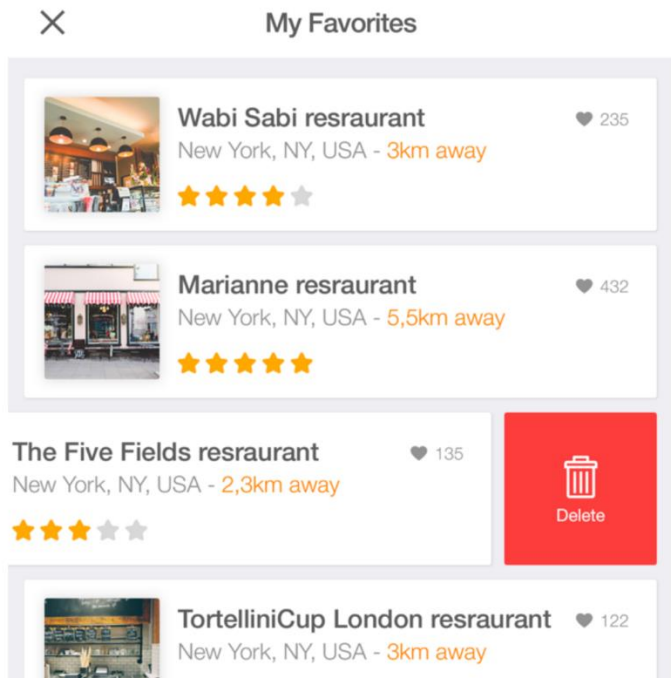


Application is all about **Feedback**

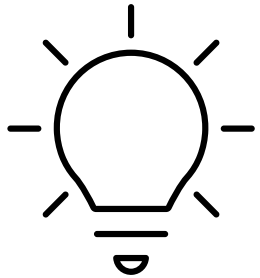
View Action → View Update

```
void onTableDeleteClick(int index){
    this.model.removeAt(index);
    this.tableView.remoteCellAt(index);
)

void onDeleteGesture(int index){
    this.model.removeAt(index);
    this.tableView.remoteCellAt(index);
)
```



View Action → Model Action →Model Update → View Update



```
void viewLoaded(){
    this.model.init();
    this.model.registerObserver(onModelChanged);
}


void onModelChanged(ChangeInfo info){
    if(info['type'] == Delete) {
        this.tableView.removeCellAt(info['index']);
    }
    else{
        this.tableView.reload();
    }
}
```

```
void onTableDeleteClick(int index){
    this.model.removeAt(index);
)
void onDeleteGesture(int index){
    this.model.removeAt(index);
)
```

1. **Data Synchronizaiton**
   Which can lead to more logical chaos

2. **Stable Control Flow**
   Easy to debug

View state 【Optional】

Does every state changing need to be persisted?

Does every view action lead to a model update.

View state VS Model state

View state: Not model relevant state

Which one is view state:

1. Comments count
2. Is Data Loading
3. Favorite
4. Is Audio Playing
5. Next track button enable

Flutter： StatefulWidget vs StatelessWidget

Demo – A Music Player

View

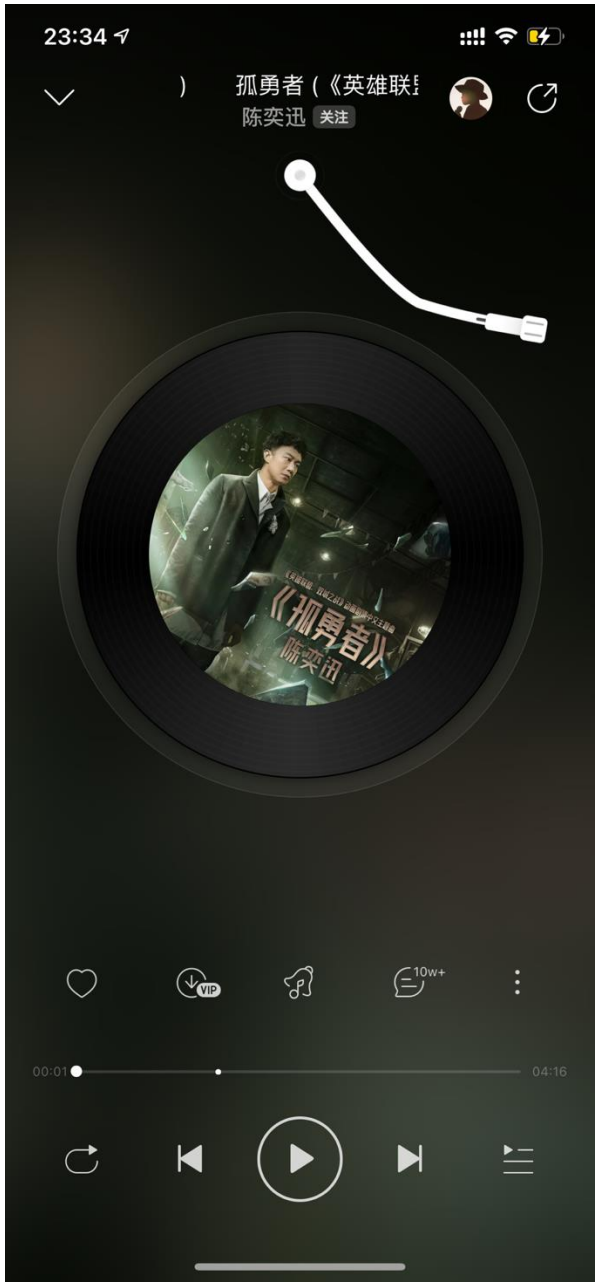
Player View

audio Play

Track

Track List

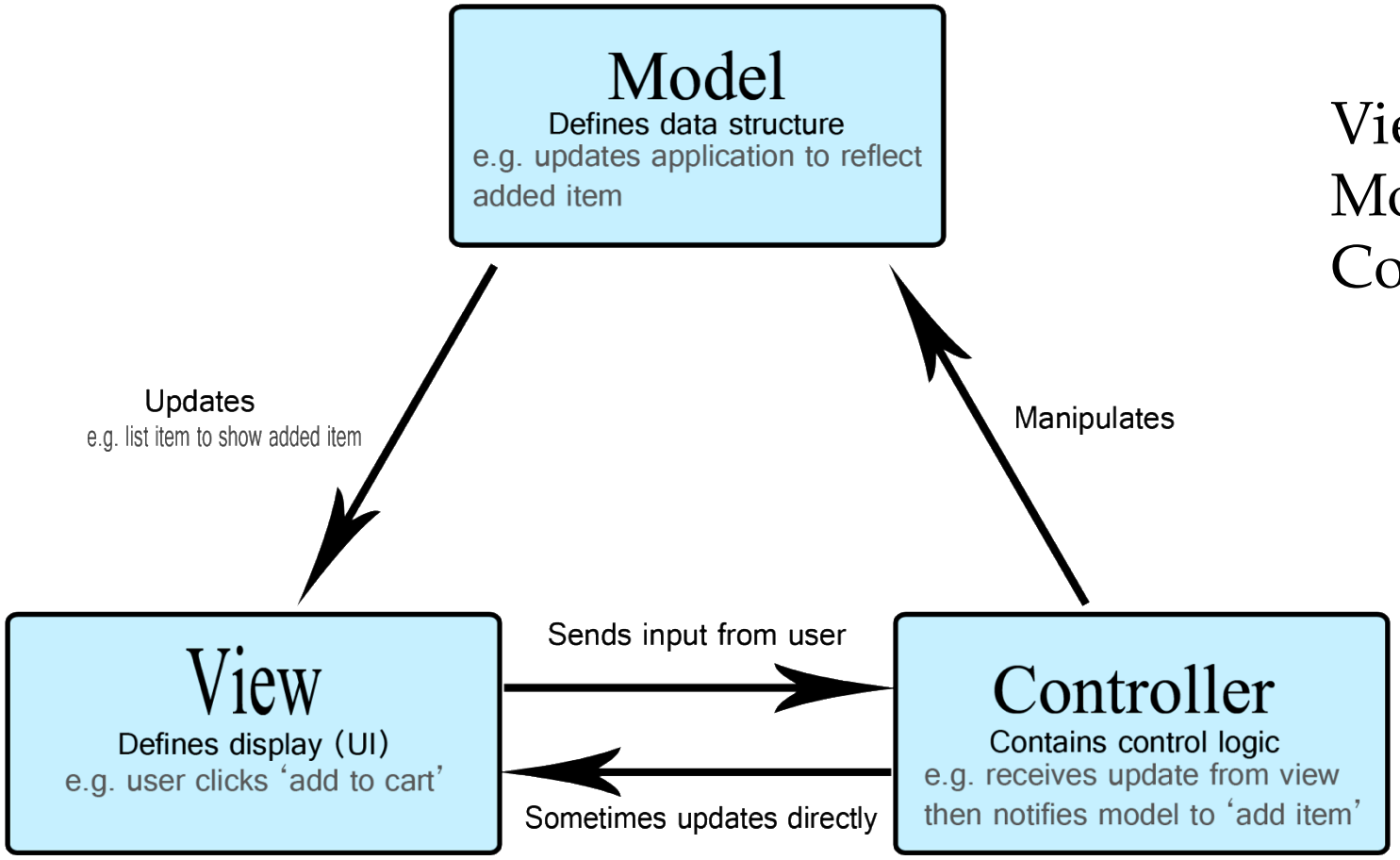
Model



Click “Next Track” What would happen ?

1. Load successfully
2. Load failure
3. Loading
4. Always Clickable?

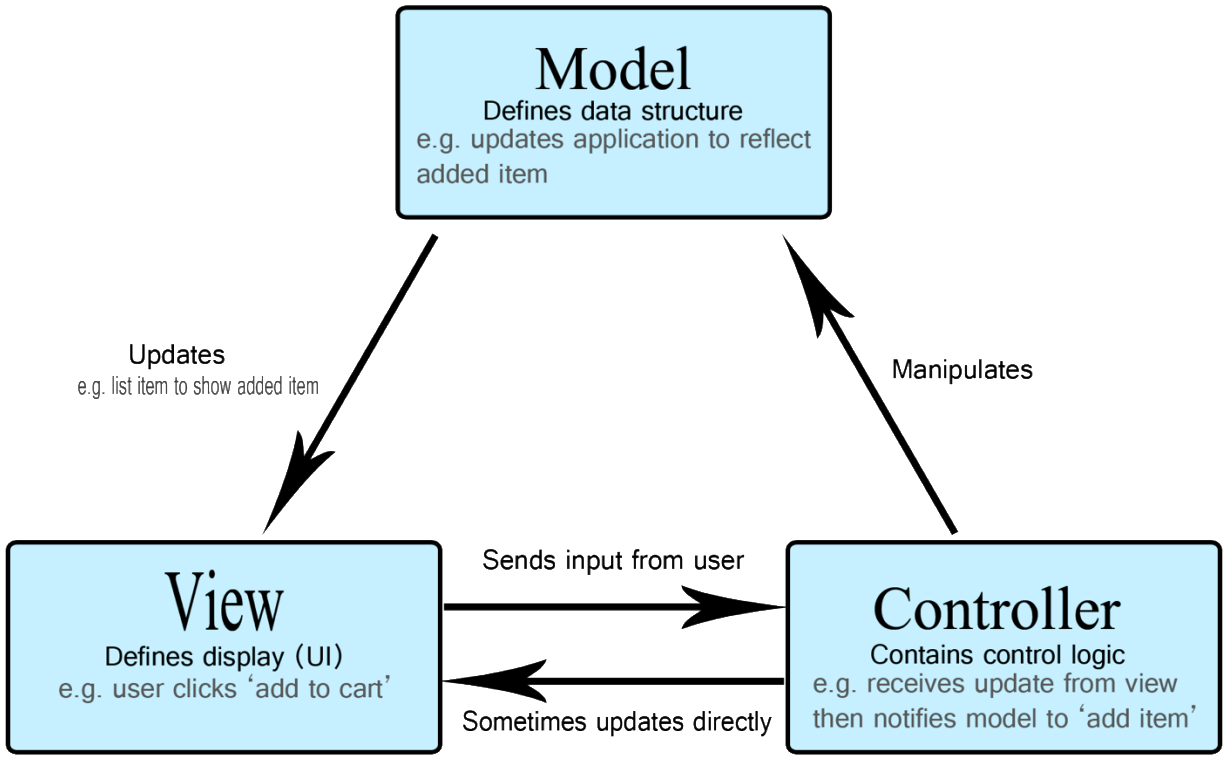
Model-View-Controller

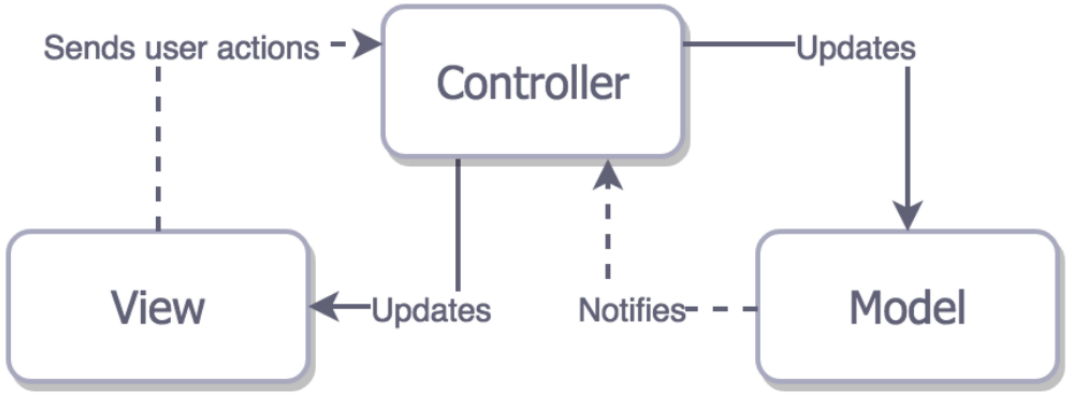


Traditional MVC:

View: Presentation.
Model: Data
Controller: Control logic.

Traditional MVC VS. Improved MVC(MVC in iOS)





Cocoa MVC

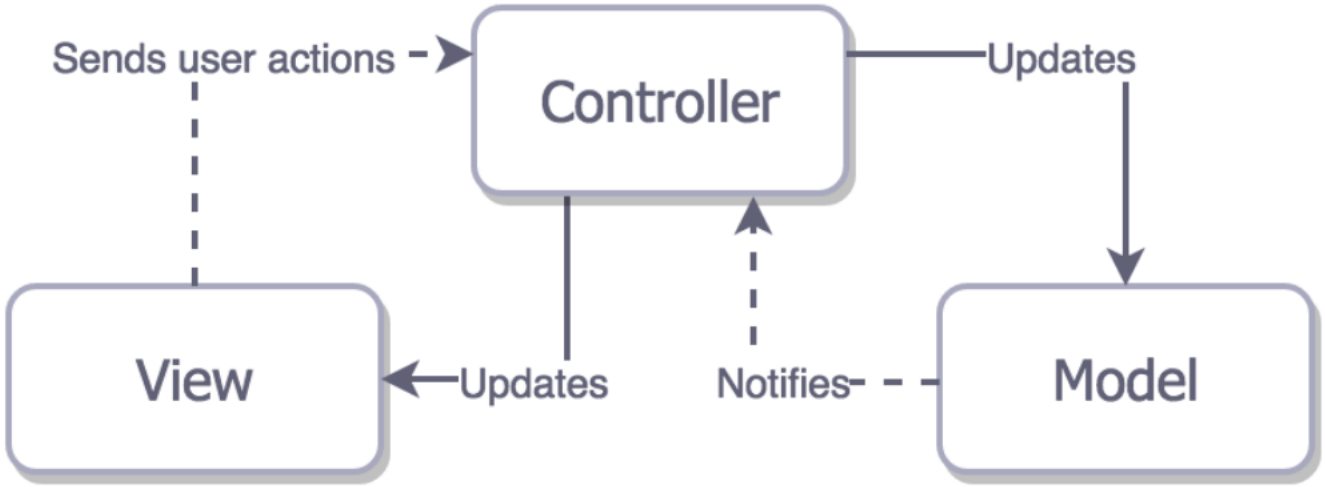
Model-View-Controller

Improved MVC: MVC in iOS

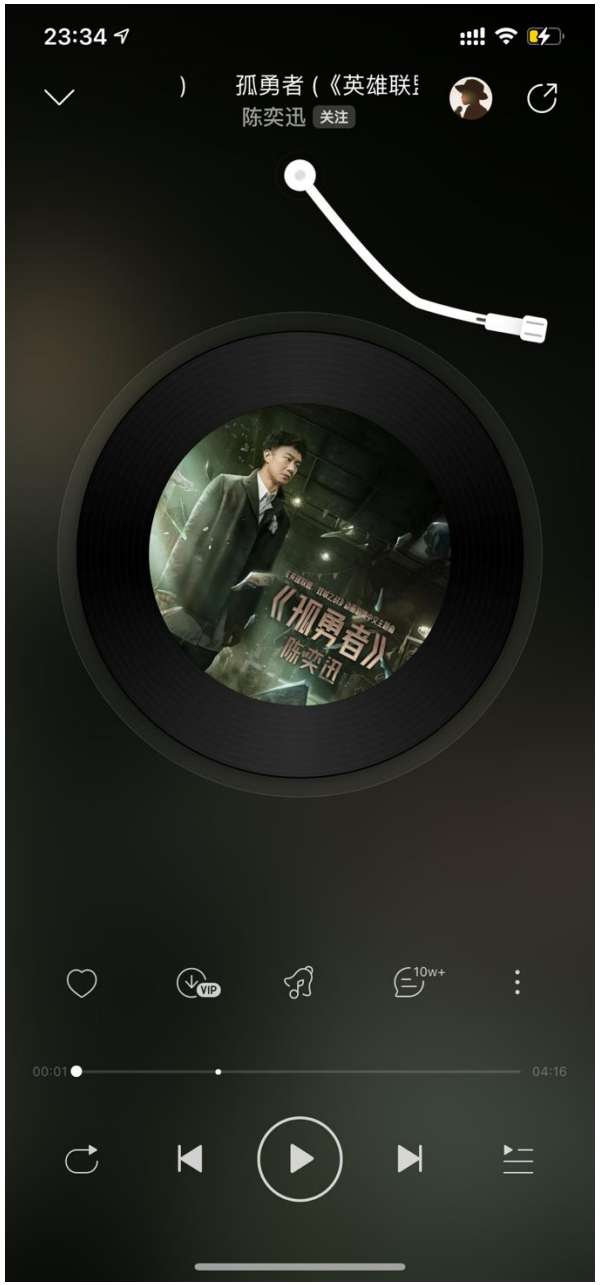
View **NEVER** communicate with model directly

Compile dependency:

Controller->Model (reuse)
Controller->View (reuse)

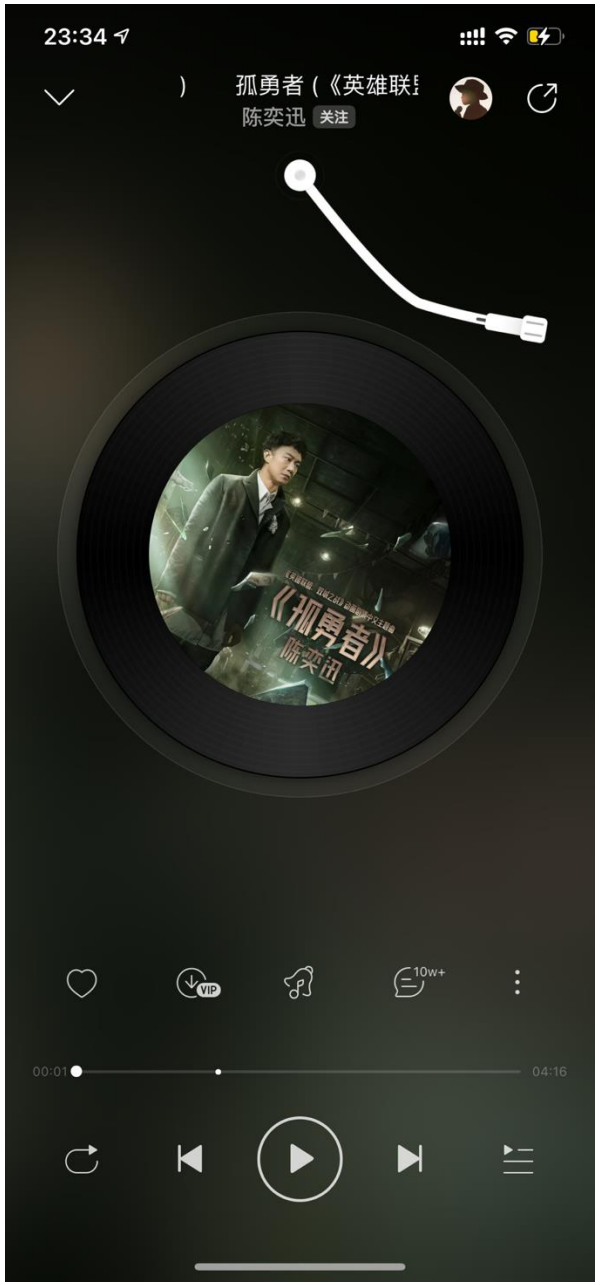


Cocoa MVC



Press “Next Track” What would happen ?

1. **Load successfully**
2. **Load failure**
3. **Loading**
4. **Always Clickable?**



Press “Next Track” What would happen ?

1. Load successfully

Show new title and play

2. Load failure

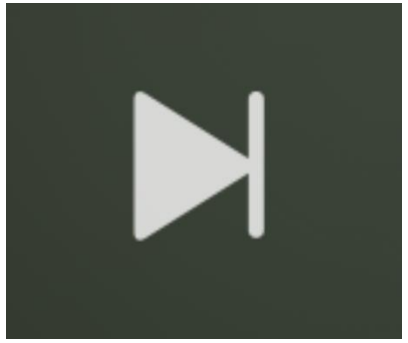
Show error message

3. Loading

Unclickable

4. Always Clickable?

Loading or no next track



MVC Codes – press “Next Track”

```
////////////////////////////////////MVC
//Controller
class PlayerController {
    PlayerModel model;
    TextView titleView;
    ErrorView errorView;
    ...
    void initController(PlayerModel model){
        this.model = model
        this.titleView.text = this.model.currentTrack.title;
        this.audioPlayModel.play(this.model.currentTrack.audioFile)
        this.errorView.hide();

        initViewObserve();
        initModelObserve();
    }

    void initViewObserve() {
        this.nextTrackButtonView.addClickHandler({
            this.nextTrackButtonView.enable = false;

            this.trackListModel.loadNextTrack();
        });
    }
```

View Action → Model Action

MVC Codes – press “Next Track”


```
//PlayerController
void initModelObserve() {
    this.model.addObserver(new IModelObserver {
        void trackLoadSucceed(){
            this.titleView.text = this.model.currentTrack.title;
            this.nextTrackButtonView.enable = !this.model.isCurrentTrackLastTrack()

            this.audioPlayModel.play(this.model.currentTrack.audioFile);
        }

        void trackLoadFailed(){
            this.nextTrackButtonView.enable = true;
            this.errorView.show('Load failed, please check network')
        }

    });
}
```

Model Update → View Update

```
//TrackListModel
class TrackListModel {
    void loadNextTrack(){
        new Thread() {
            Int trackId = this.trackList[this.currentIndex + 1]
            Track newTrack = downloader.download(trackId)
            if newTrack != null {
                this.currentTrack = newTrack
                this.observer.trackLoadSucceed()
            }
            else {
                this.observer.trackLoadFailed()
            }
        }
    }
}
```

Model

MVC Roles

View :

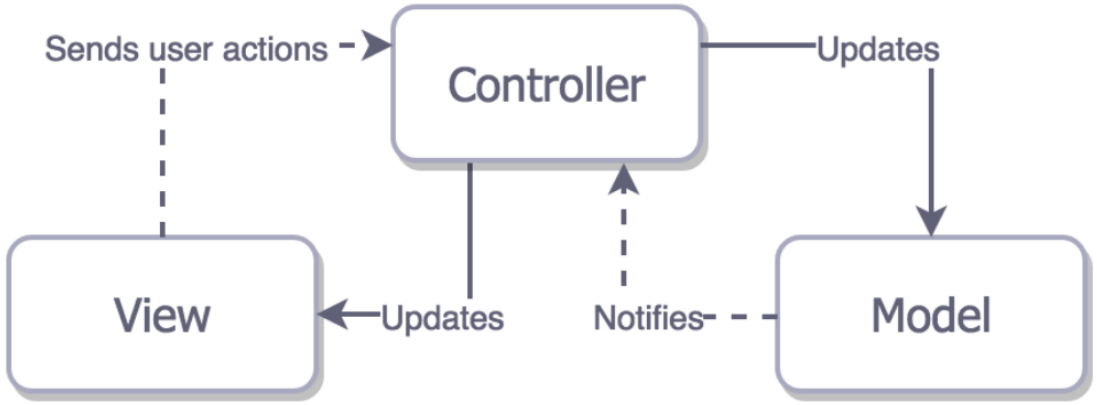
1. Presentation: subview management

Controller:

1. View action handling / update model.
2. Observing model / update view.
3. Model data transform
4. View state management .

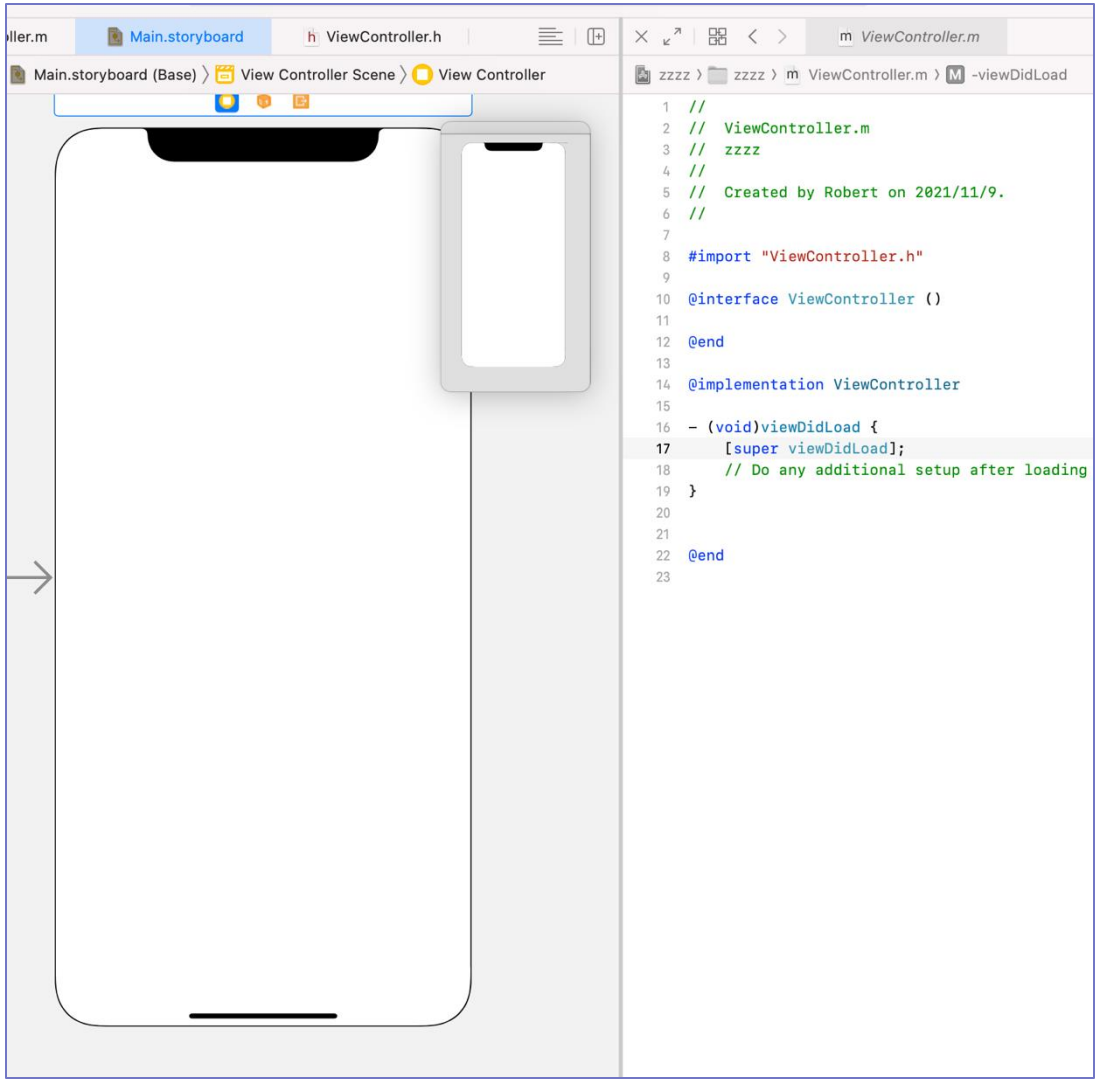
Model :

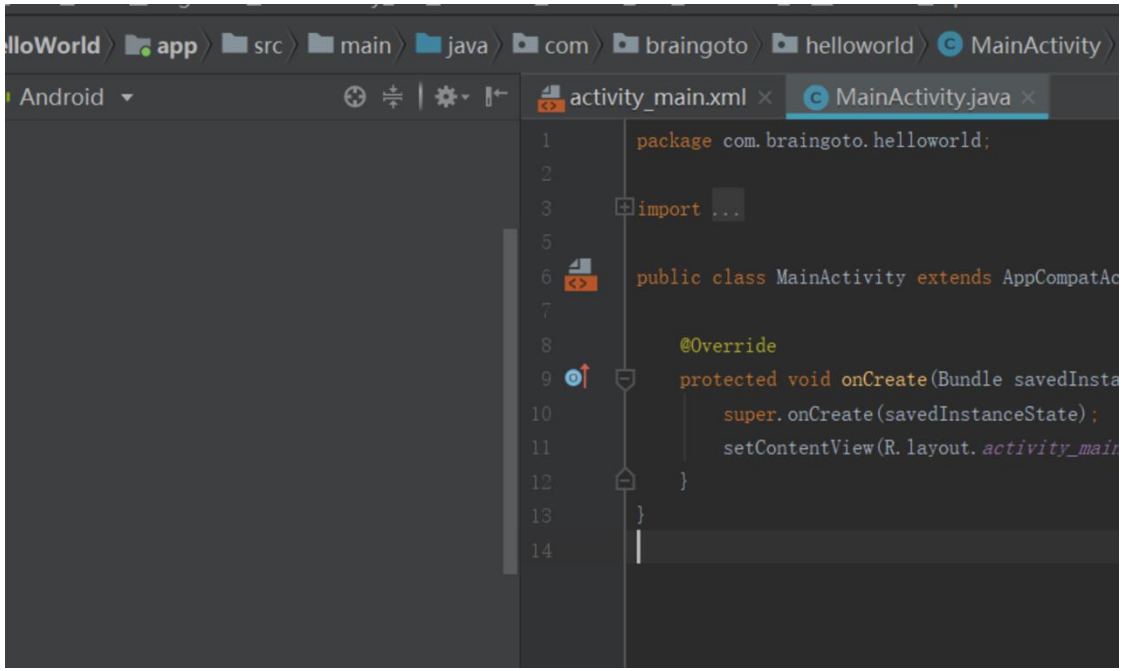
1. Model state management / persistence.



Cocoa MVC

MVC In Mobile Application Development





MVC Advantage

- Using the MVC pattern allows for a **separation of concerns**. By separating the presentation from the data, it makes it easier to change one of them without affecting the other. It also makes each part easier to test.

- The MVC pattern makes presentation objects **more reusable**. Separating the user interface from the data allows UI components to be reused. It also means that a model can be reused with more than one view.

- The separation of the presentation from the business logic and data allows developers to specialize in either frontend or backend development. This can also **speed up the development process** as some tasks can take place in parallel.

MVC Disadvantage comparing to using NO architecture

- Because we need extra overhead due to layers which will do **negative impact on the performance**.
- **Development of user-intensive applications can sometime take longer** if the layering prevents the use of user interface components that directly interact with the database.
- The use of layers helps to control and encapsulate the complexity of large applications but **adds complexity to simple applications**.

MVC Disadvantage comparing to using OTHER architecture

- In a project built with the Model-View-Controller pattern, you are often faced with the question which code goes where. Code that doesn't fit or belong in the model or view layer is often put in the controller layer. **This inevitably leads to fat controllers** that are difficult to test and manage.

- **Controllers** are notoriously **hard to test** because of their relation to the view layer.

Massive Controller

master

firefox-ios / Client / Frontend / Browser / **BrowserViewController.swift**

nbhasin2 Fix #6893 - Added new tab open button for landscape UI (#6947) ...

85 contributors +55

2509 lines (2113 sloc) | 107 KB

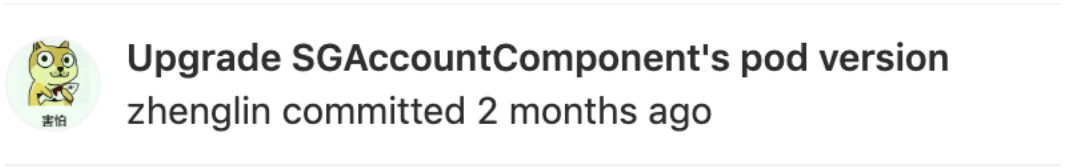
```
1  /* This Source Code Form is subject to the terms of the Mozilla Public
2   * License, v. 2.0. If a copy of the MPL was not distributed with this
3   * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
4
5  import Foundation
6  import Photos
7  import UIKit
8  import WebKit
```

Massive Controller Solution

1. Bigger Model (Data Fetch, Data Calculation)

2. More (Child) Controllers

Data transform :Model or Controller?



UI-independent transform

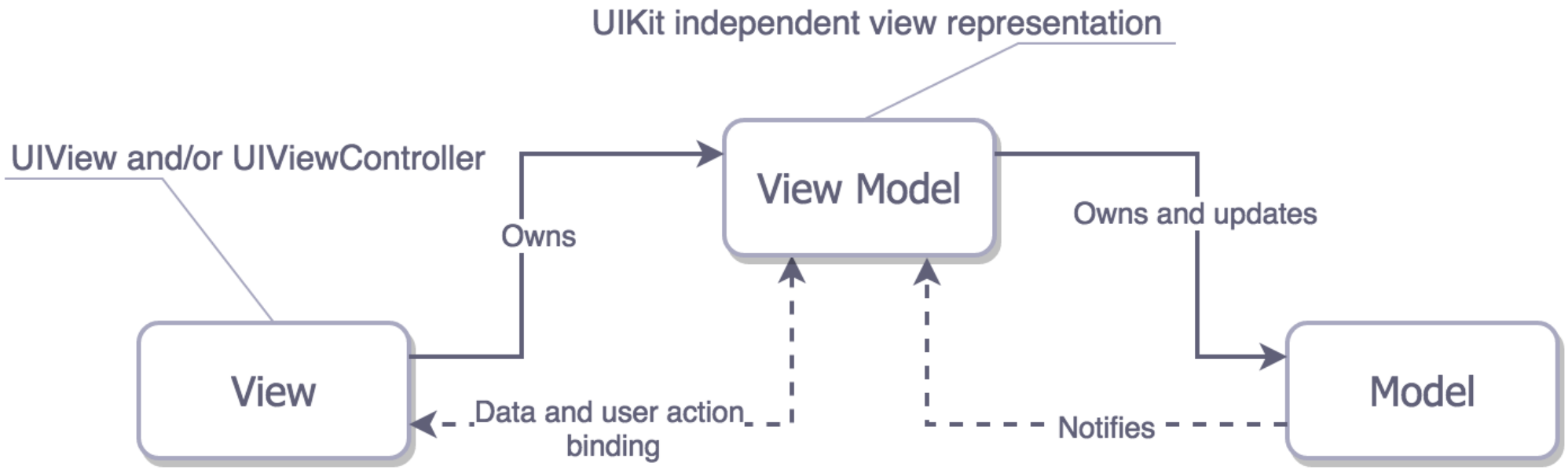
Model

UI-oriented transform

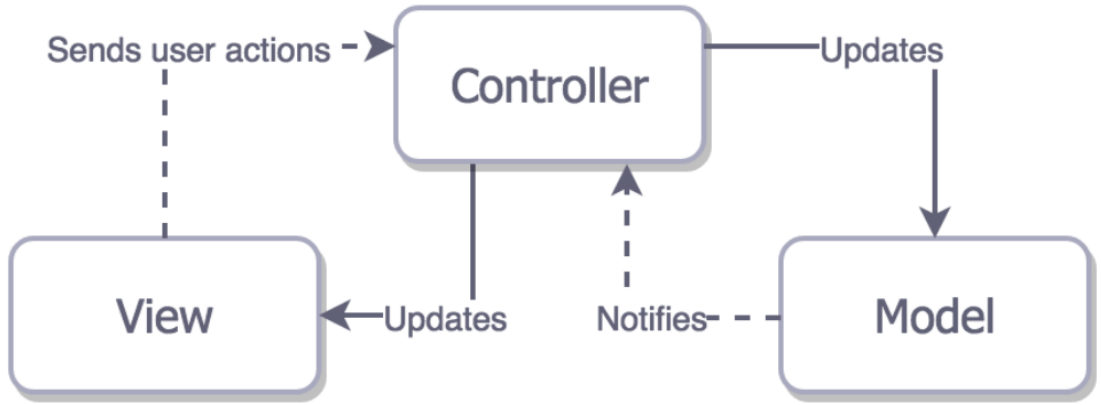
Controller

What if I told you, the Controller is the View.

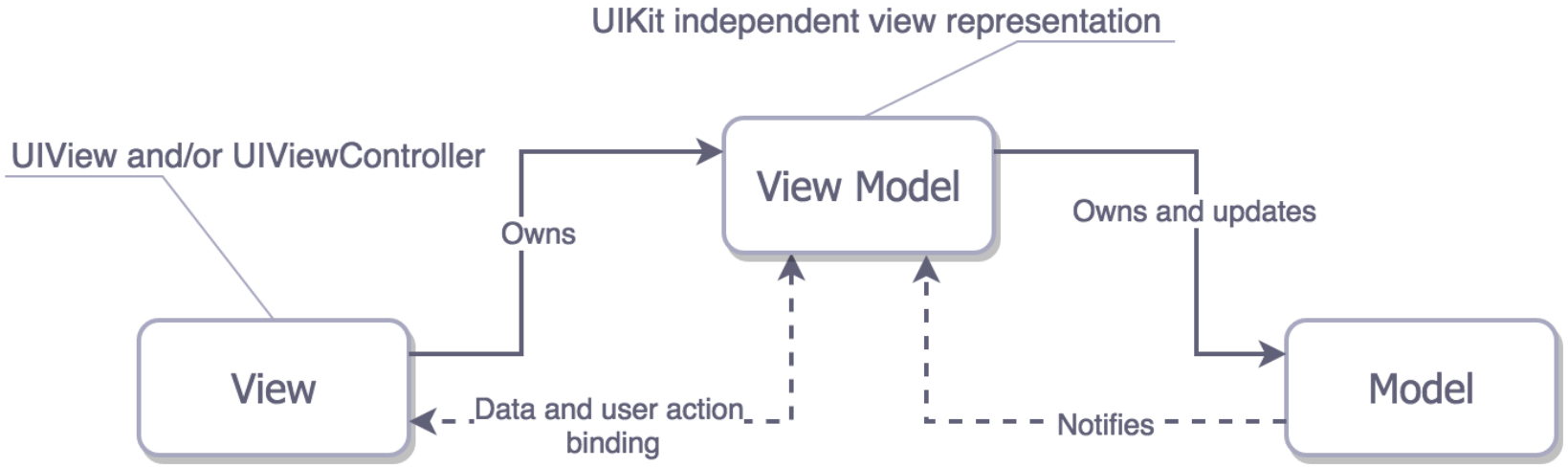
Model-View-View Model



MVC VS MVVM



Cocoa MVC



Reactive Programming

Reactive programming *is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.*

Imperative Programming

How to maintain the relationship: **y = x + 1**

```
//setup
int y = x + 1;

//action
void doA(){
    x = 15;
    y = x + 1;
}

void doB(){
    x = 23;
    y = x + 1;
}
```

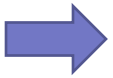
Reactive Programming

How to maintain the relationship: **y = x + 1**

```
//setup
int y = x + 1;

//action
void doA(){
    x = 15;
    y = x + 1;
}

void doB(){
    x = 23;
    y = x + 1;
}
```



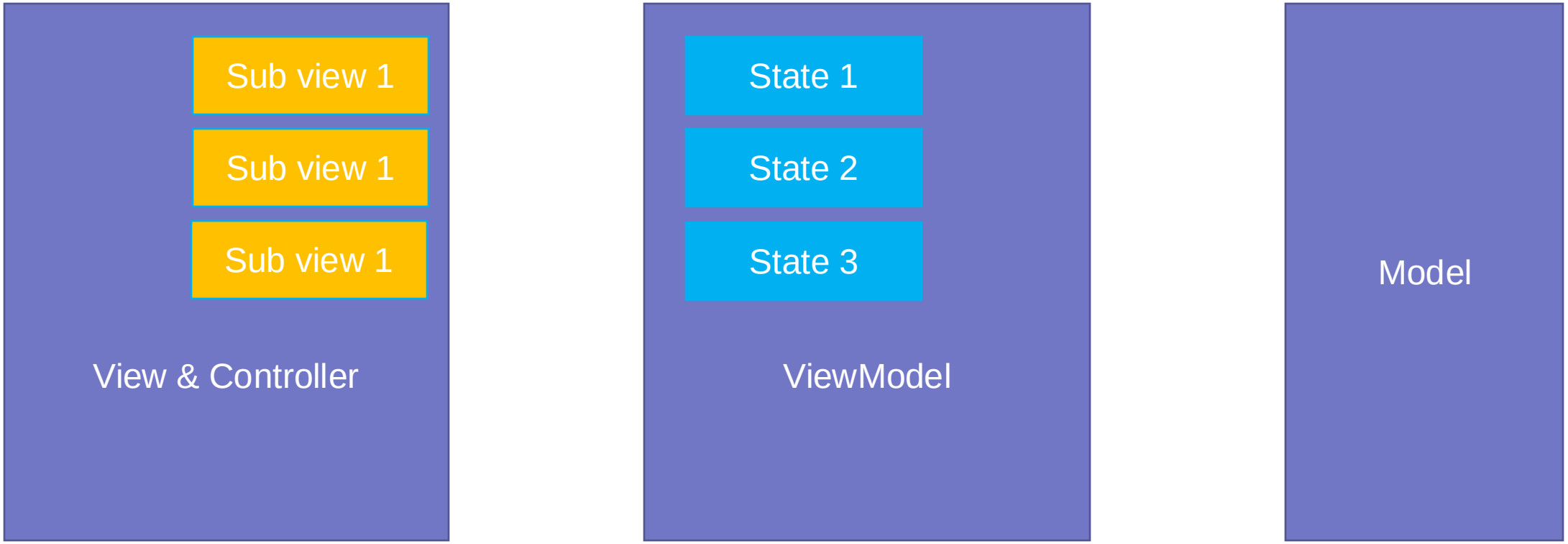
```
//setup

int y bind (x + 1);

//action
void doA(){
    x = 15;
}

void doB(){
    x = 23;
}
```

ViewModel


Sub view 1
Sub view 1
Sub view 1
View & Controller
State 1
State 2
State 3
ViewModel
Model

MVVM Codes – press “Next Track”

```
////////////////////////////////////////MVVM
//Controller
class PlayerController {
    PlayerViewModel viewModel;
    ...
    void initView(PlayerViewModel viewModel){
        this.viewModel = viewModel;

        this.nextTrackButtonView.addClickHandler({
            this.viewModel.loadNextTrack();
        });

        // Data bind
        BIND(this.titleView.text, this.viewModel.currentTrackTitle);
        BIND(this.nextTrackButtonView.enable, this.viewModel.canNextTrack);
        BIND(this.errorView.display, !this.viewModel.loadSucceed)
        BIND(this.errorView.message, this.viewModel.errorMessage)
    }
}
```

View **bind** ViewModel


```
//MVVM
//PlayerViewModel
class PlayerViewModel {
    String currentTrackTitle;
    Bool canNextTrack;
    Bool loadSucceed;
    Bool errorMessage;

    void initViewModel() {
        this.audioPlayModel.play(this.model.currentTrack.audioFile)

        this.model.addObserver(new IModelObserver {
            void trackLoadSucceed(){
                this.currentTrackTitle = this.model.currentTrack.title;
                this.canNextTrack = !this.model.isCurrentTrackLastTrack();
                this.loadSucceed = true;

                this.audioPlayModel.play(this.model.currentTrack.audioFile);
            }

            void trackLoadFailed(){
                this.canNextTrack = true;
                this.loadSucceed = false ;
                this.errorMessage = "Load failed, please check network";
            }

        });
    }

    void loadNextTrack(){
        self.canNextTrack = false
        this.model.loadNextTrack();
    }
}
```

ViewModel Update **Self**

No UI involved in ViewModel

MVVM Roles

View :

1. Presentation: subview management

Controller :

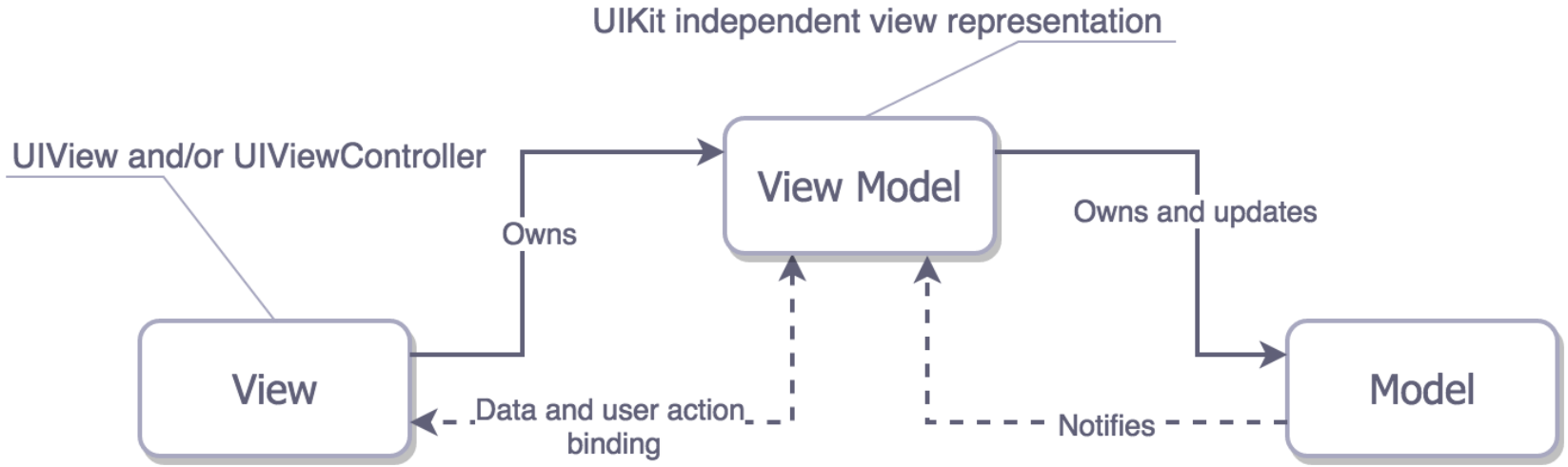
1. View action handling .
2. Create binding between view and view model.

View Model :

1. View state management .
2. Model data transform .
3. Observing model / update self.

Model :

1. Model state management / persistence



MVVM Advantage

- The MVVM pattern presents a **better separation of concerns** by adding view models to the mix. The view model translates the data of the model layer into something the view layer can use. The controller is no longer responsible for this task.

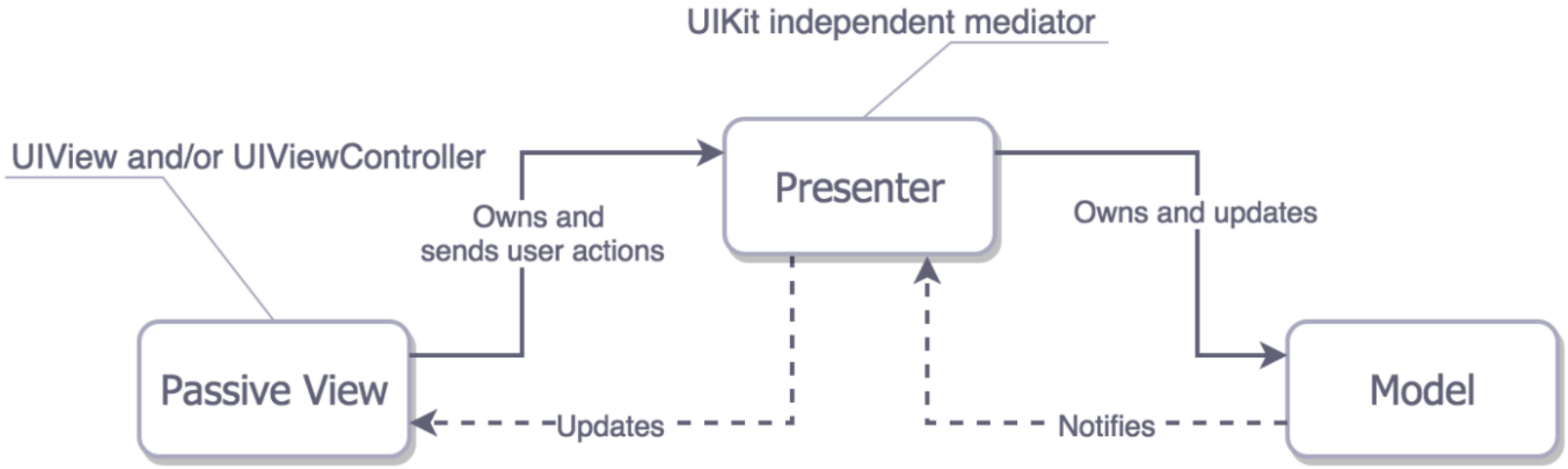
- By migrating data manipulation to the view model, testing becomes much easier**. Testing view models is easy**. Because a view model doesn't have a reference to the object it is owned by, it easy to write unit tests for a view model.

- The view model provides a transparant interface to the view controller, which it uses to populate the view layer and interact with the model layer. This results in a **transparant communication** between the four layers of your application..

MVVM Disadvantage

- Communication between various MVVM components and data binding can be painful
- Managing view models and their state in nested views and complex UI's is difficult
- Data flow is not **one-way**

Model-View-Presenter



Passive View variant of MVP

Model-View-Presenter

View does not contain any logic. It will route user inputs and commands (events) to the Presenter.
View usually has a reference to its Presenter.

When compared to the View and Controller in the MVC pattern, the View and Presenter present in the MVP pattern are fully decoupled from each other and they **communicate by means of an interface**. (The Presenter interact with the Views via View interface, it means that the Presenter can perform all presentation and navigation tasks without any dependency on the actual UI technology being used)

Model-View-Presenter

Abstract View

```
////////////////////////////////////MVP
//Controller
interface IAbstractPlayerView {
    void showTitle(String title);
    void enableNextTrack(Bool enable);
    void showErrorTip(String message);
}
```

```
//PlayerController
void initController(PlayerPresenter presenter){
    this.presenter = presenter
    this.titleView.text = this.presenter.currentTrack.title;
    this.errorView.hide();

    this.nextTrackButtonView.addClickHandler({
        this.nextTrackButtonView.enable = false;
        this.presenter.loadNextTrack();
    });

    this.presenter.abstractView = this;
}
```

```
class PlayerController :IAbstractPlayerView {
    PlayerPresenter presenter;
    ...

    void showTitle(String title) {
        this.titleView.text = title;
    }

    void enableNextTrack(Bool enable)  {
        this.nextTrackButtonView.enable = enable;
    }

    void showErrorTip(String message) {
        if(message == null) {
            this.errorView.hide()
        }
        else {
            this.errorView.show(message)
        }
    }
```

Model-View-Presenter

Presenter

```
//PlayerPresenter
class PlayerPresenter{
    IAbstractPlayerView abstractView
    void initPresenter() {
        this.audioPlayModel.play(this.model.currentTrack.audioFile)

        this.model.addObserver(new IModelObserver {
            void trackLoadSucceed(){
                this.abstractView.showTitle(this.model.currentTrack.title);
                this.abstractView.enableNextTrack(!this.model.isCurrentTrackLastTrack());
                this.abstractView.showErrorTip(message: null);

                this.audioPlayModel.play(this.model.currentTrack.audioFile);
            }

            void trackLoadFailed(){
                this.abstractView.enableNextTrack(enable: true)
                this.abstractView.showErrorTip(message: "Load failed, please check network");
            }
        });
    }
    void loadNextTrack(){
        this.model.loadNextTrack();
    }
}
```

MVP Advantage

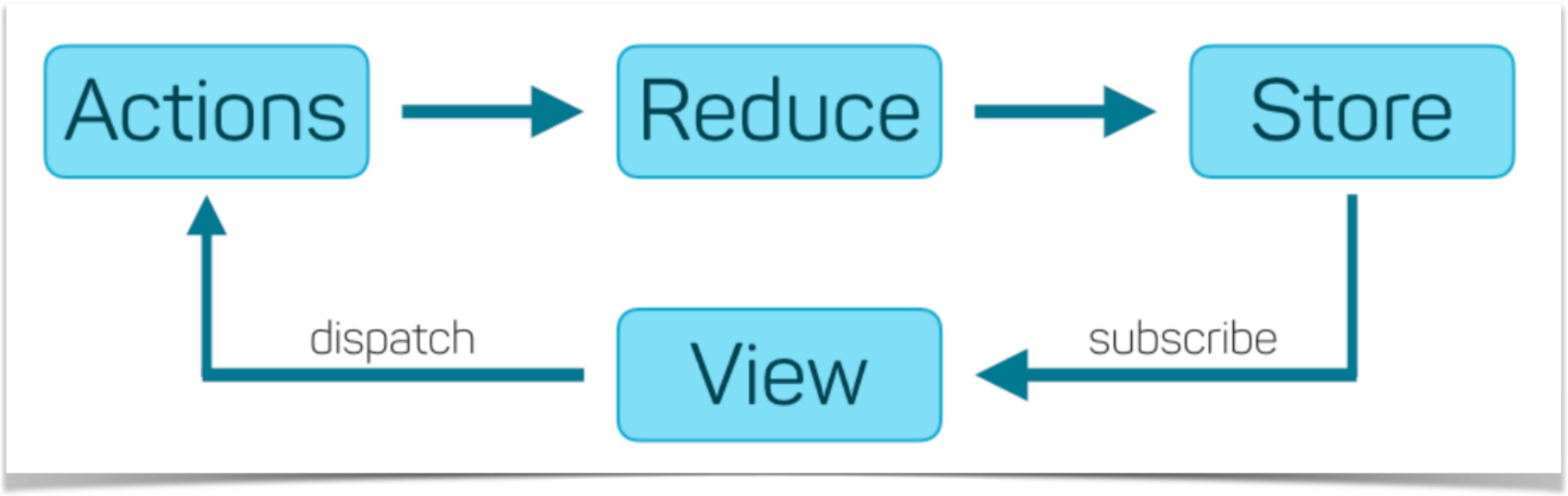
- **We can easily test the Presenter logic** as it is not tied to any OS-specific views and APIs.
- **The View and Presenter are entirely separate**, which makes mocking a view easy, making unit testing more superficial than then in the MVC.
- **We only have one class** that handles everything related to the presentation of a view – the Presenter.

MVP Disadvantage

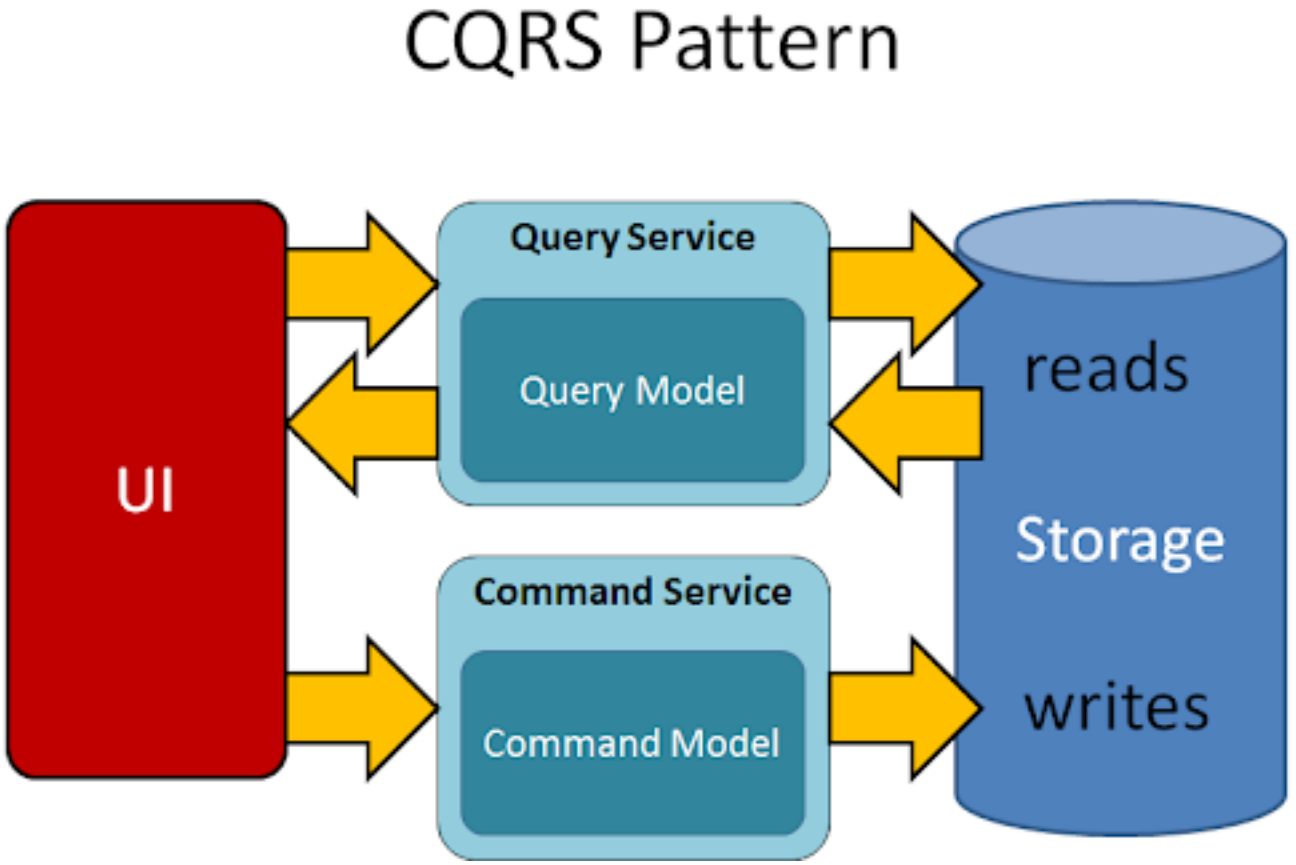
- **The Presenter,** like the Controller, **tends to accumulate additional business logic.** To solve this problem, break down your code and remember to create classes with only one responsibility.

- While this is a great pattern for an Android app, it can feel **overwhelming when developing a small app** or prototype.

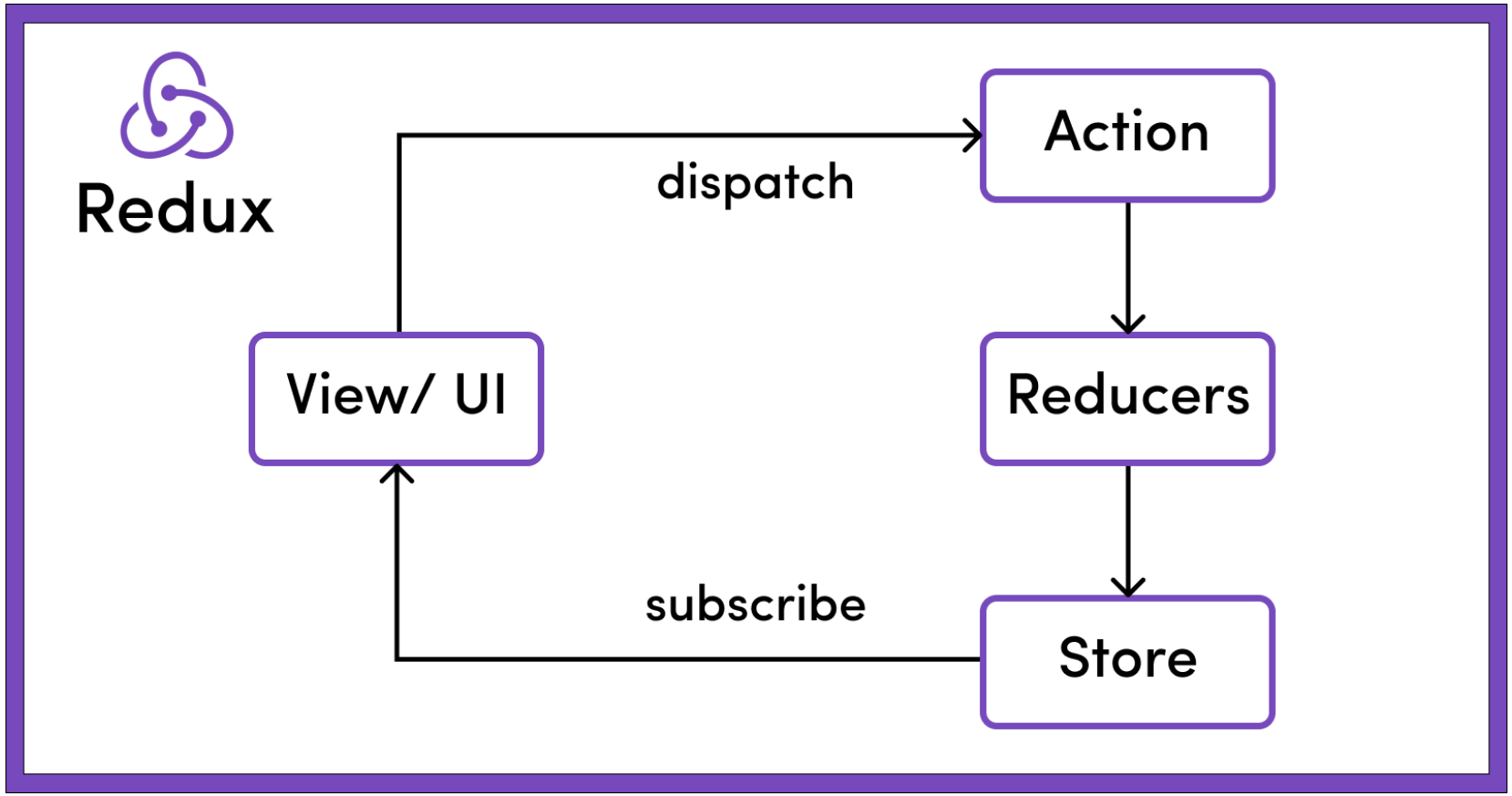
CQRS / Redux / Model-View-Intent



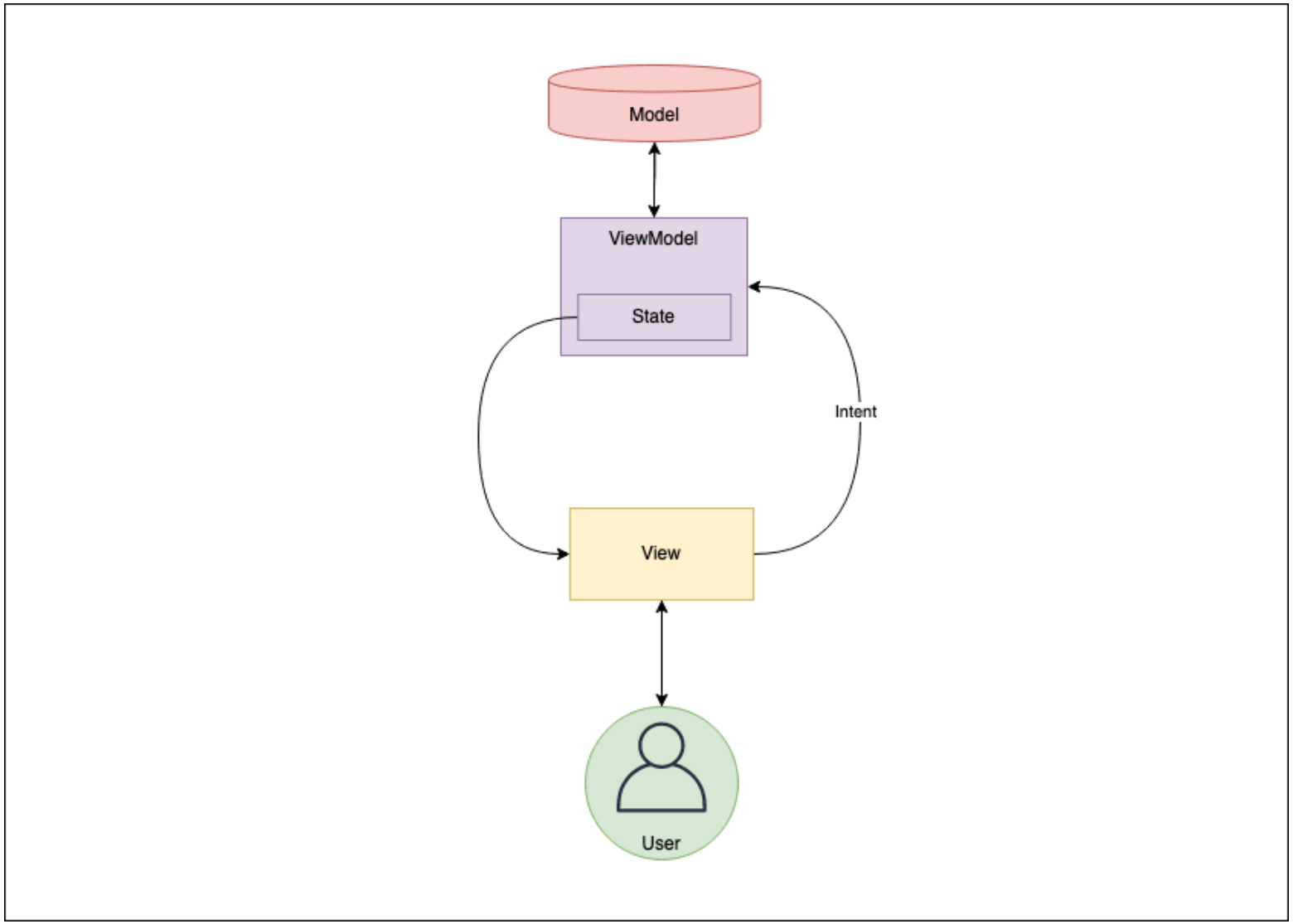
Command Query Responsibility Segregation



Redux



Model-View-Intent



Model-View-Intent

```
//State
class PlayerState {
    String title;
    Image cover;
    Bool isLastTrack;
    Bool isFirstTrack;
    Bool loadSucceed;
    String errorMessage;

}
```

```
//Intent
class PlayerIntent {}
class InitIntent() : PlayerIntent {}
class NextTrackIntent : PlayerIntent {}
class PrevTrackIntent : PlayerIntent {}
class SelectTrackIntent : PlayerIntent {
    Int index;
    SelectTrackIntent(Int index);
}
```

State & Intent

Model-View-Intent

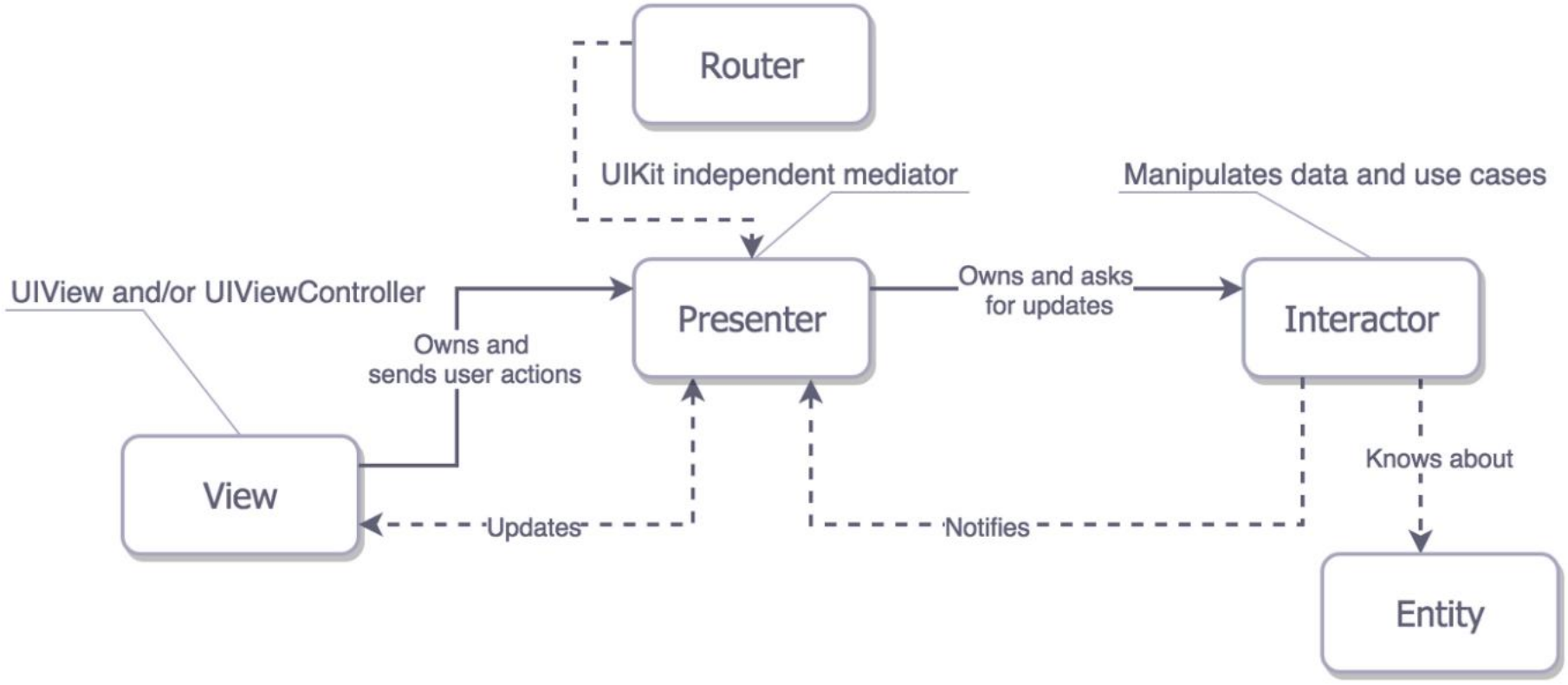
```
//Controller
class PlayerController {
    PlayerViewModel viewModel;
    ...
    void initView(PlayerViewModel viewModel){
        this.viewModel = viewModel;

        this.viewModel.subscribeState((PlayerState state){
            this.titleView.text = state.title
            this.this.nextTrackButtonView.enable = state.loadSucceed && !state.isLastTrack;
            if (!state.loadSucceed) {
                this.errorView.show(state.errorMessage);
            }
        });

        this.nextTrackButtonView.addClickHandler({
            this.viewModel.publishIntent(NextTrackIntent());
        });

    }
}
```

VIPER


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks
for updates
Interactor
Owns and
sends user actions
View
Updates
Notifies
Knows about
Entity

VIPER

VIPER

•**Interactor** — contains business logic related to the data (**Entities**) or networking, like creating new instances of entities or fetching them from the server. For those purposes you'll use some *Services* and *Managers* which are not considered as a part of VIPER module but rather an external dependency.

•**Presenter** — contains the UI related (but UIKit *independent*) business logic, invokes methods on the **Interactor**.

•**Entities** — your plain data objects, not the *data access layer,* because that is a responsibility of the **Interactor**.

•**Router** — responsible for the segues between the VIPER **modules**.

VIPER

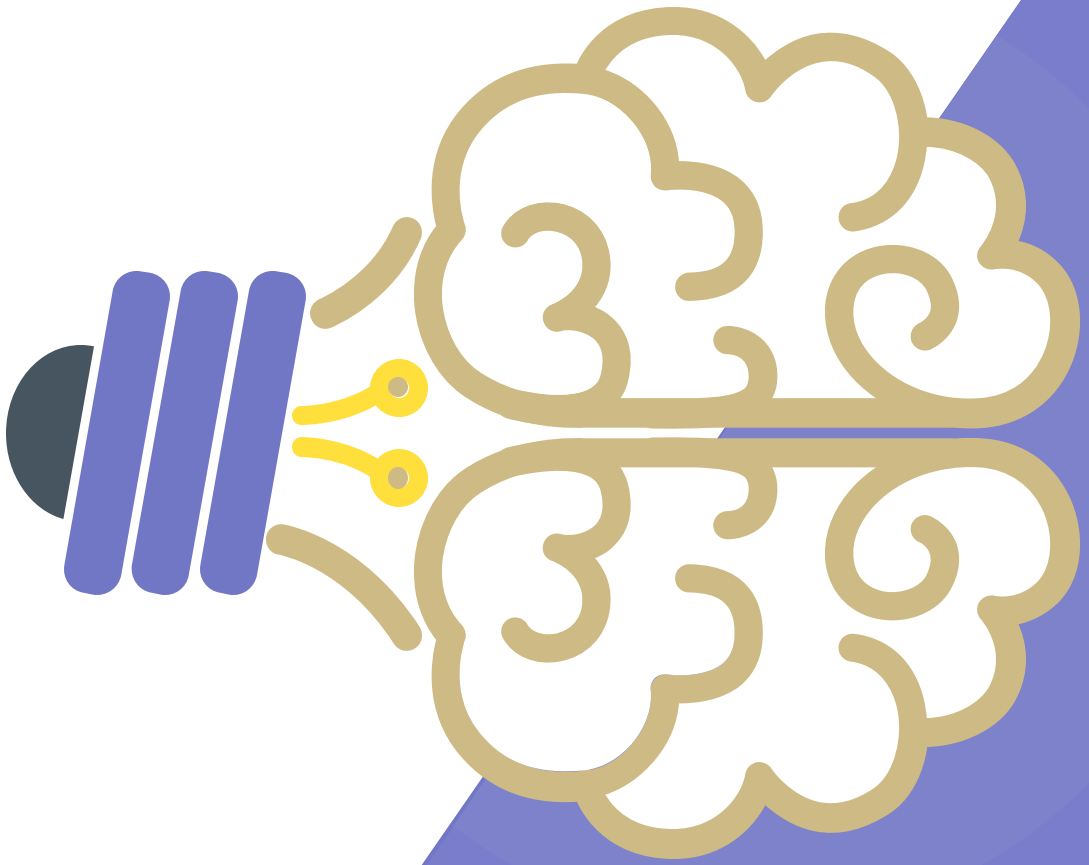
- **Model** (data interaction) logic shifted into the **Interactor** with the **Entities** as dumb data structures.
- Only the UI representation duties of the **Controller/Presenter/ViewModel** moved into the **Presenter,** but not the data altering capabilities.
- **VIPER** is the first pattern which explicitly addresses navigation responsibility, which is supposed to be resolved by the **Router.**

/04

Componentization

Supporting text here.

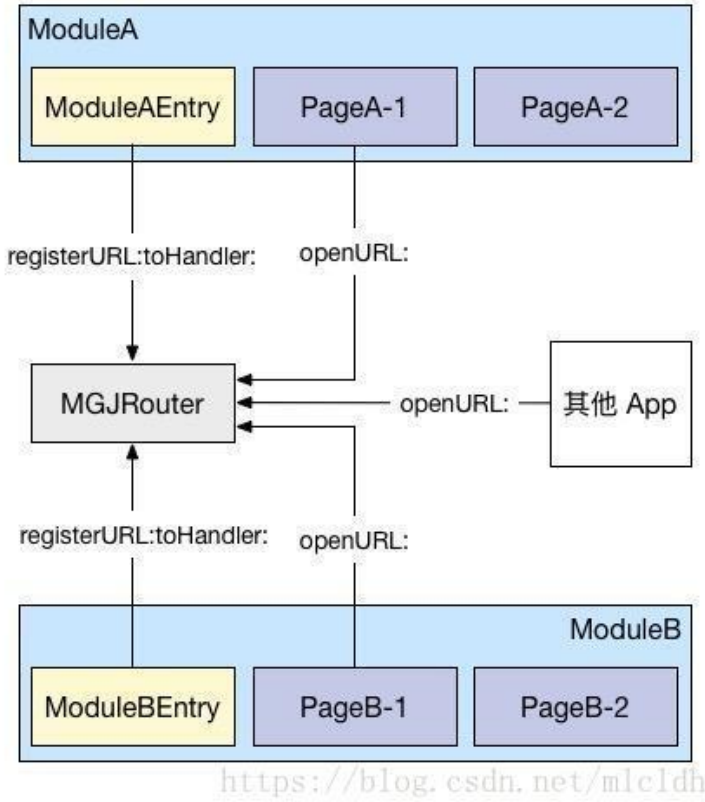
When you copy & paste, choose "keep text only" option.



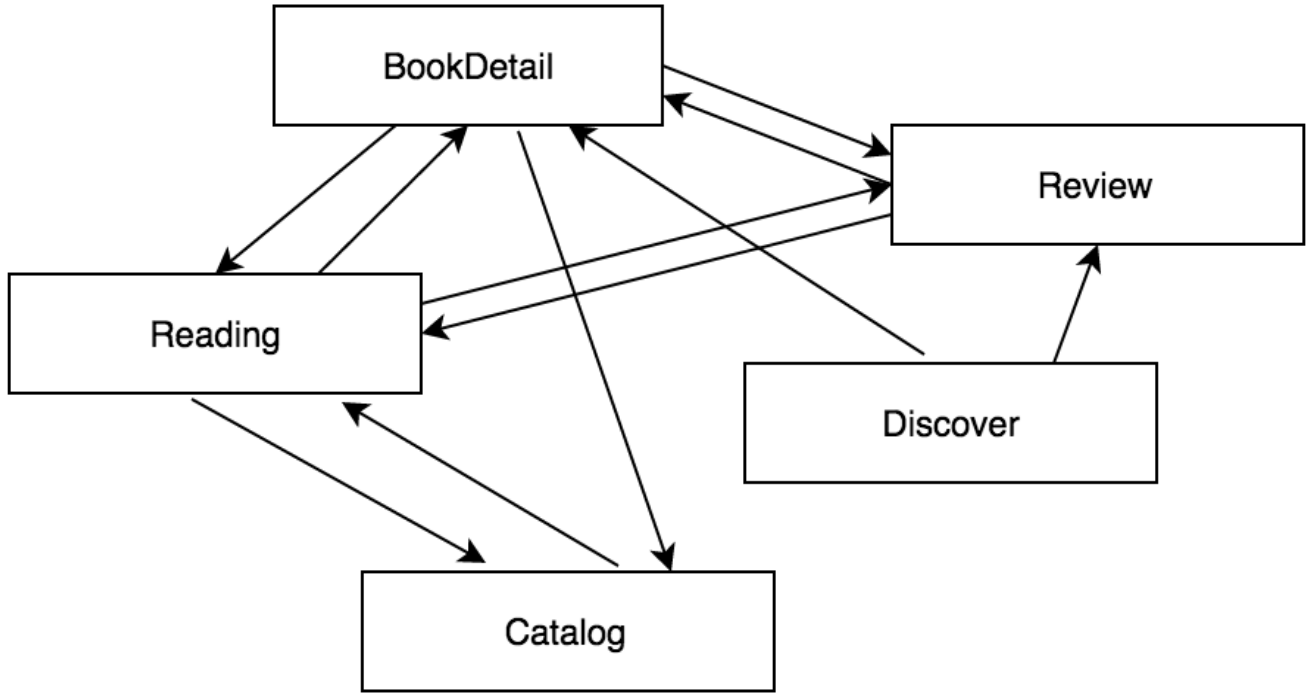
Componentization

Componentization is an approach to software development that involves breaking software down into identifiable pieces that application developers **independently write and deploy**.

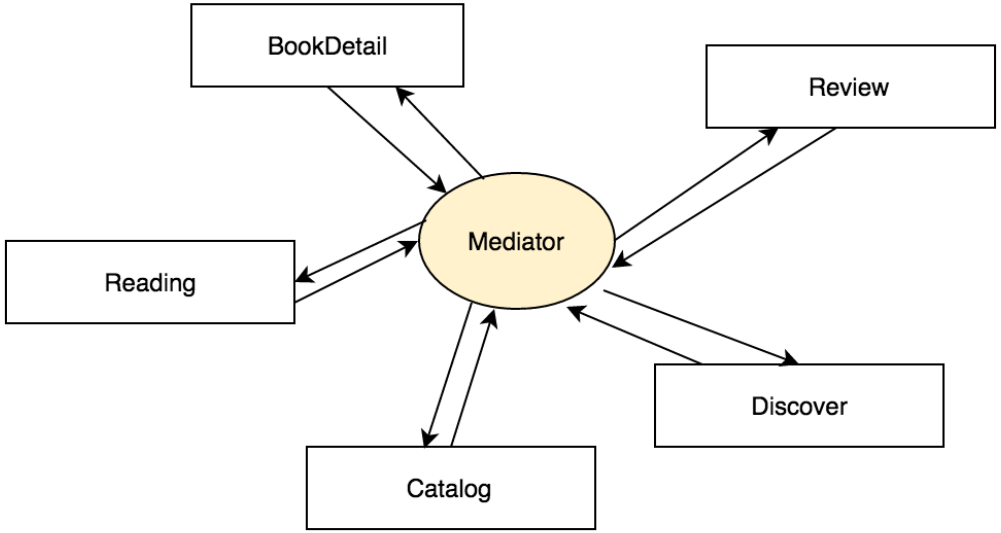
Componentization aims to facilitate software development through the use of reusable components that connect together using standard interfaces. The components themselves must conform to a known model that dictates how the components connect.

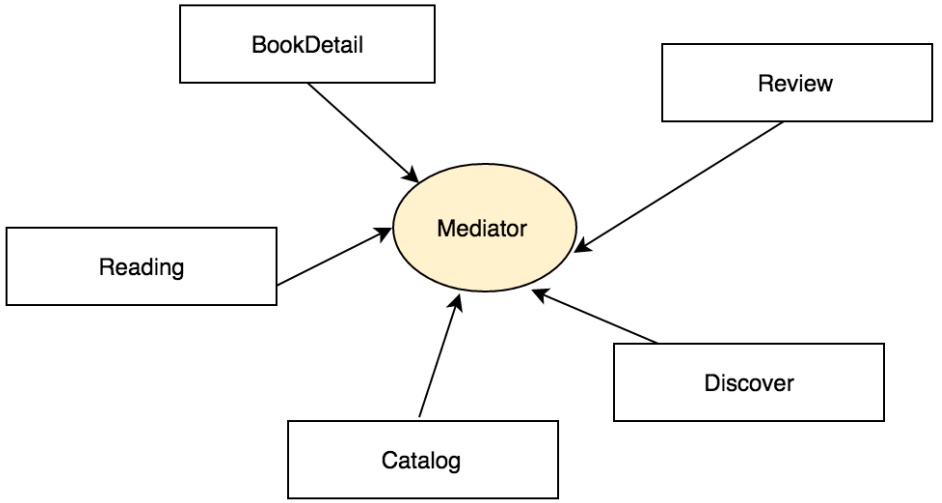


Original Solution



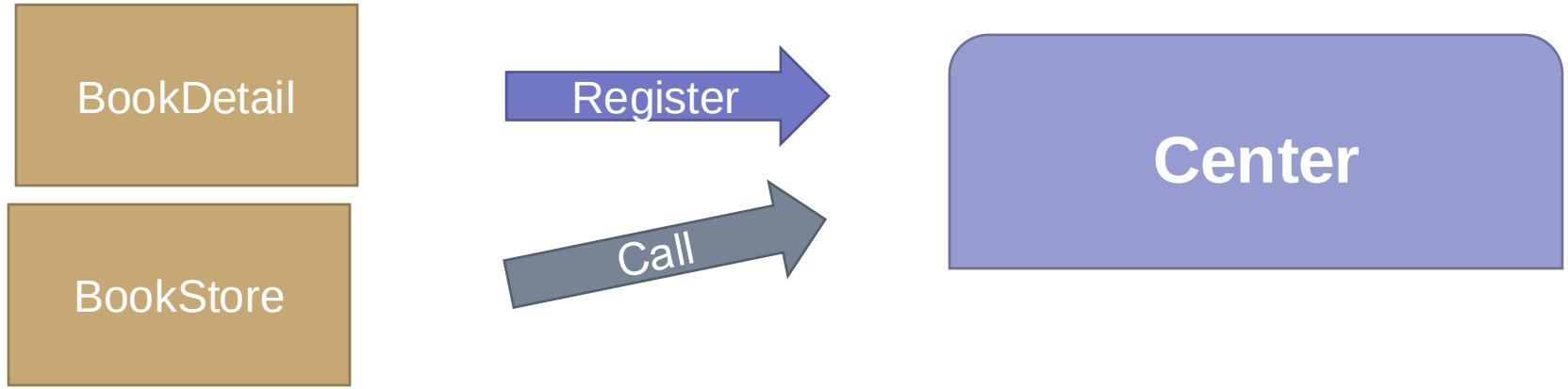
Componentization





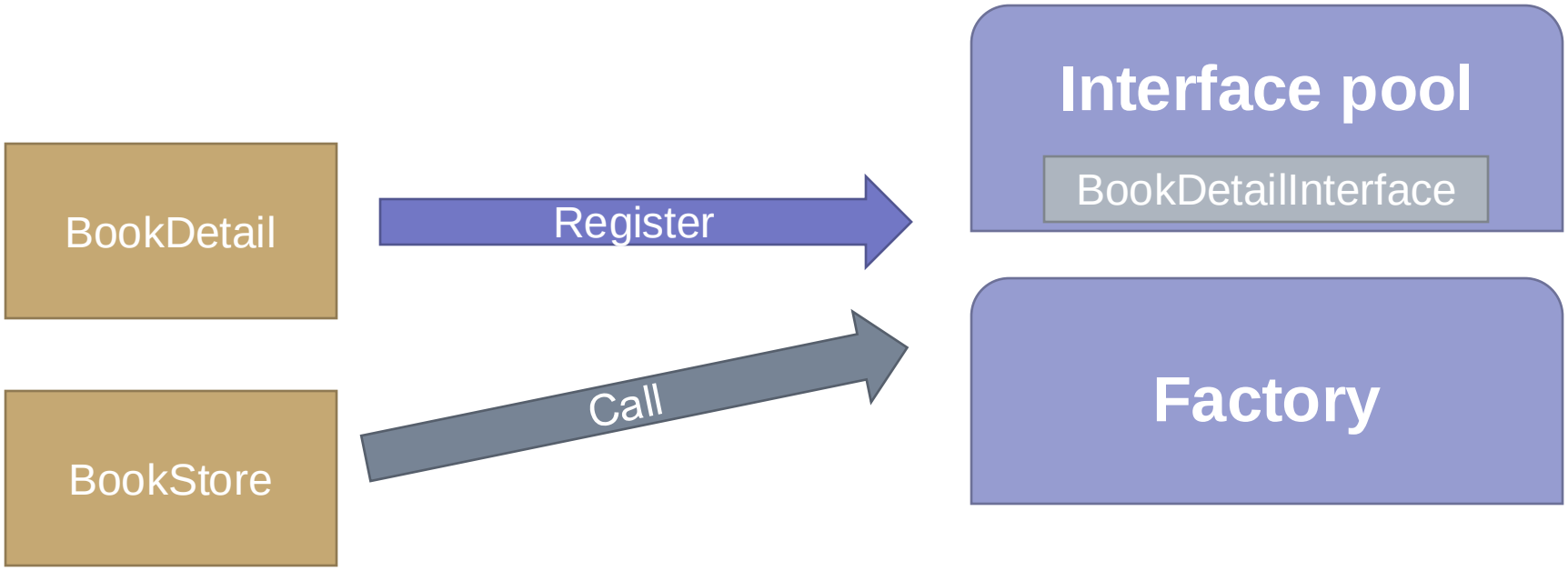
Solution1:URL Register

Url	Class
bookreader://account/bookdetail/openbook?id=12345	BookDetailController
…	…



Solution2: Interface Register

interface	Class
BookDetailInterface	BookDetailController
…	…



Solution3: Reflection – No Register

bookreader://account/UserInfo/reload


```
Class UserInfo {

    void reload() {



    }

}
```

Componentization advantage

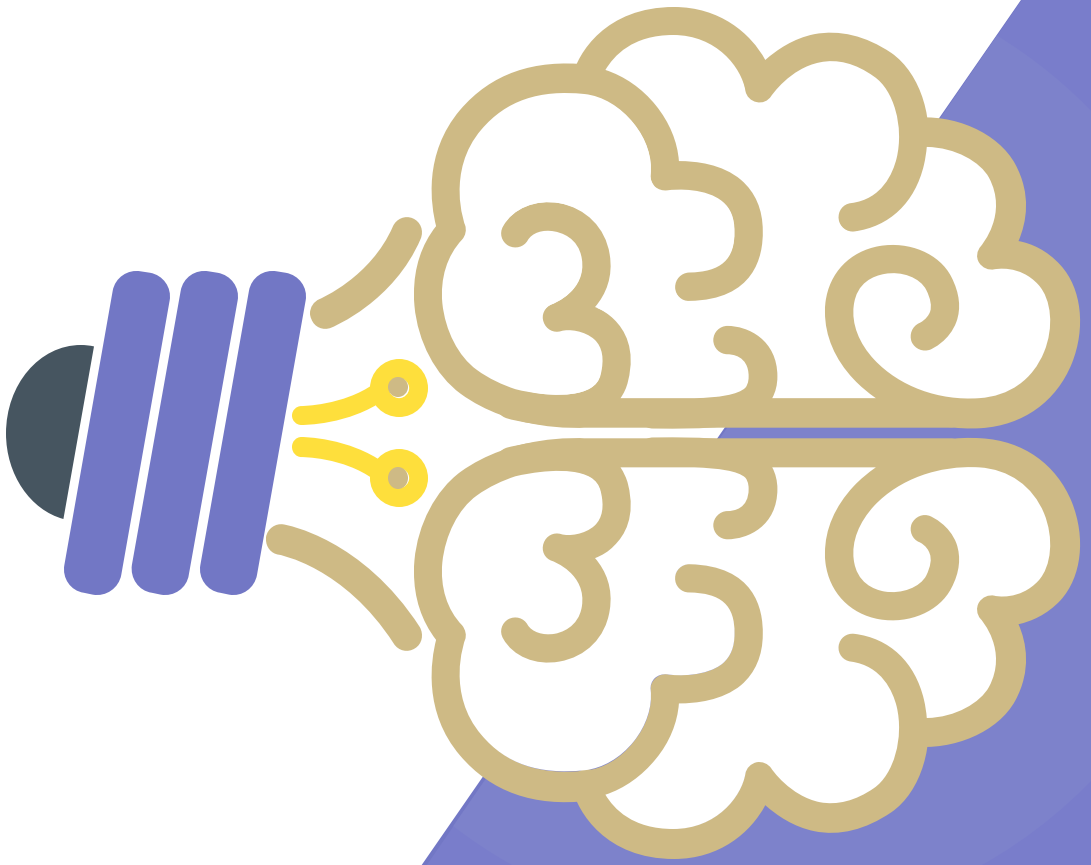
In the **short-term**, componentization enables software development teams to more easily collaborate. Reusing components that meet well-defined specifications also helps accelerate product development while increasing software reliability. As a result, teams can bring better quality software to market faster.

In the **long-term**, componentization provides flexibility. Components can be easily replaced or added to address changing business requirements.

/05

Practice

overdesign



Overdesign

“

Overdesigning

is a Cardinal Sin of Design

“

Overdesign

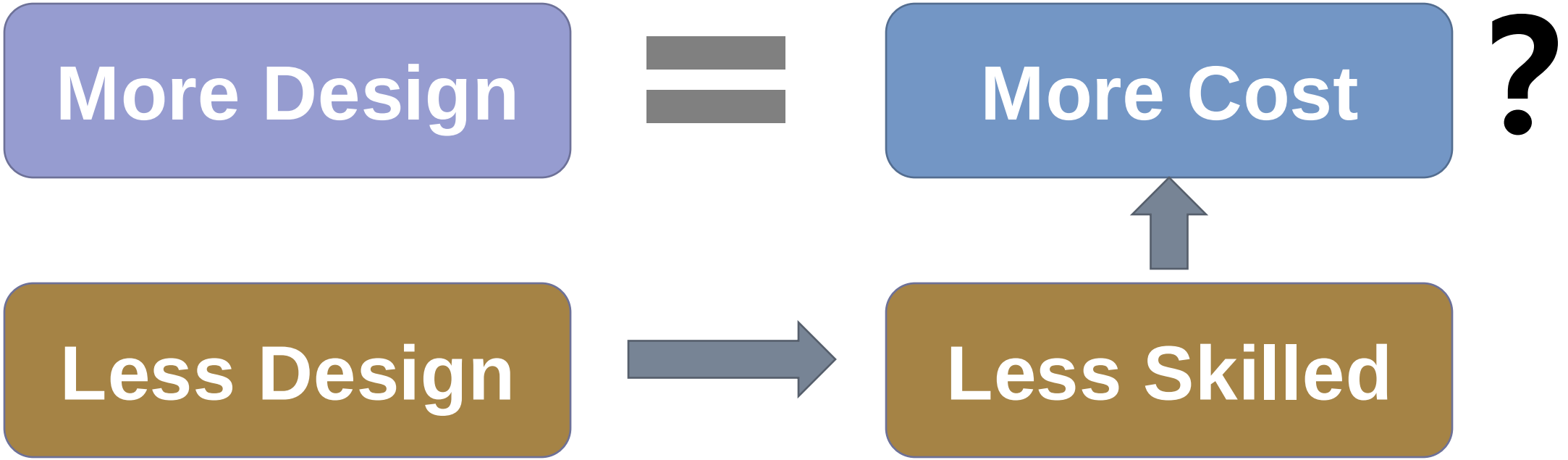
The issue is not:

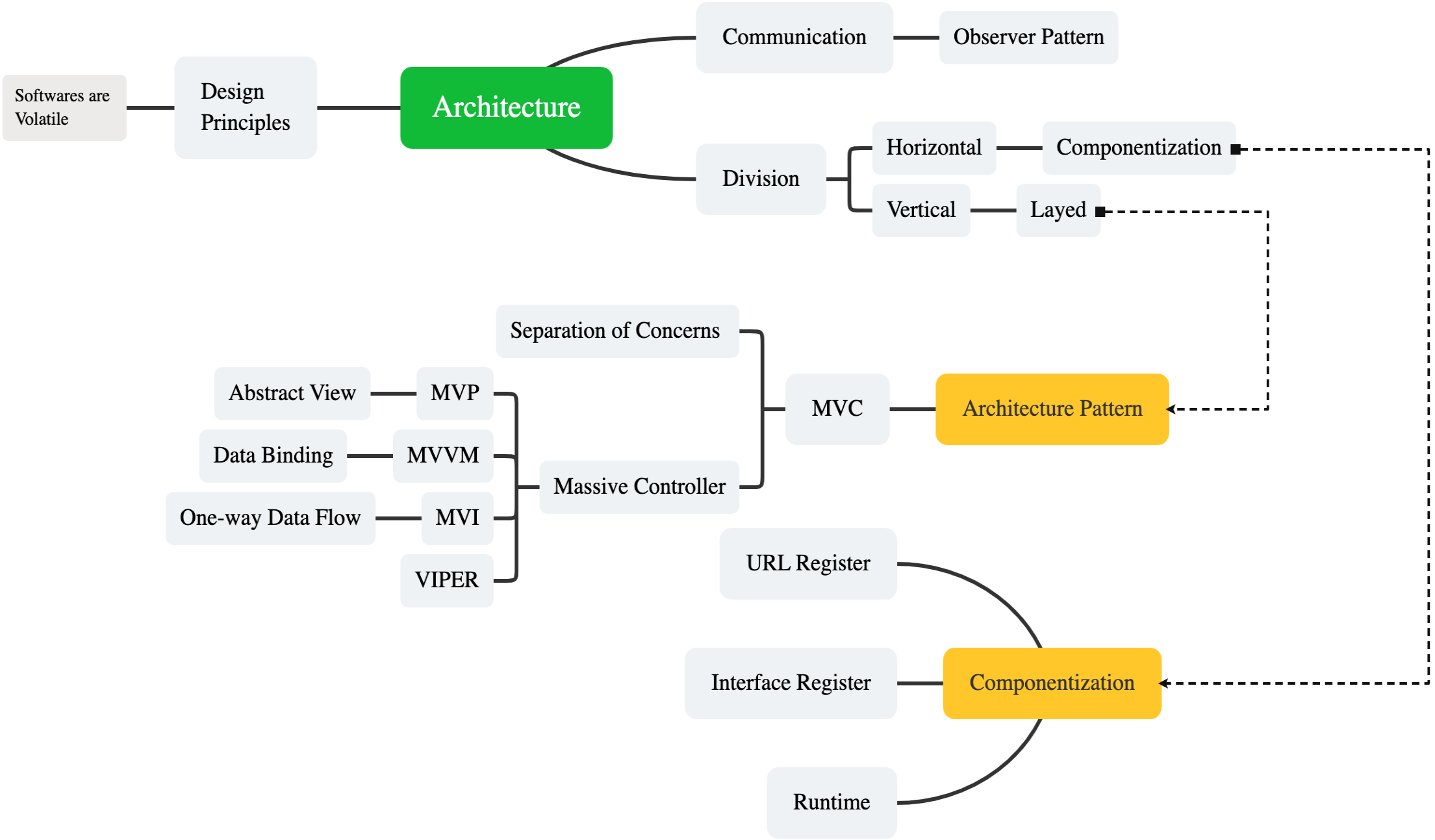
“Overdesign”

The issue is:

“Over **NOT** Design”

Overdesign




Softwares are Volatile
Design Principles
Architecture
Communication
Observer Pattern
Division
Horizontal
Componentization
Vertical
Layed
Separation of Concerns
Abstract View
MVP
Data Binding
MVVM
One-way Data Flow
MVI
VIPER
Massive Controller
MVC
Architecture Pattern
URL Register
Interface Register
Componentization
Runtime

Keep Coding

Keep Practicing

Keep Thinking

Robert

北京 海淀

扫一扫上面的二维码图案，加我为朋友。

THANKS

于洋