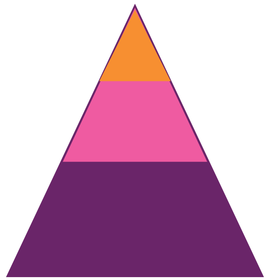


UI	Application Service	Domain Service	Domain Model
接口数据库实现 外部服务接口实现 外部依赖集成 -Spring/Mysql/Feign	外部服务接口定义 业务逻辑实现	基础接口定义	

前端 →

Web 2.0

→ 后端

什么时候应该关注架构?

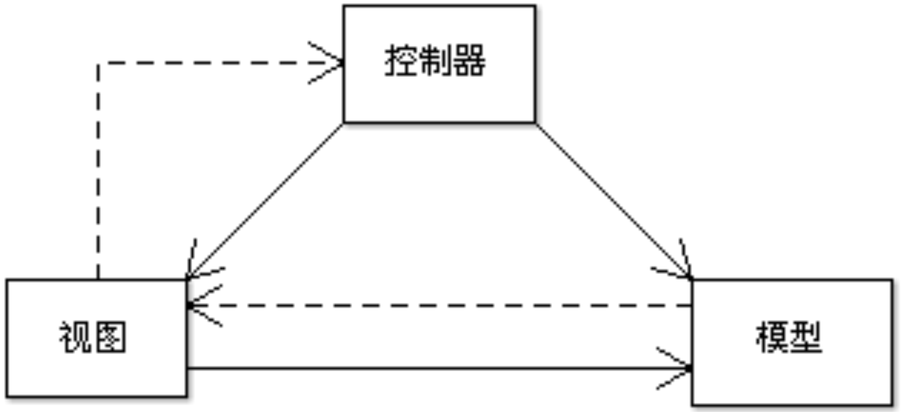
前端架构演进

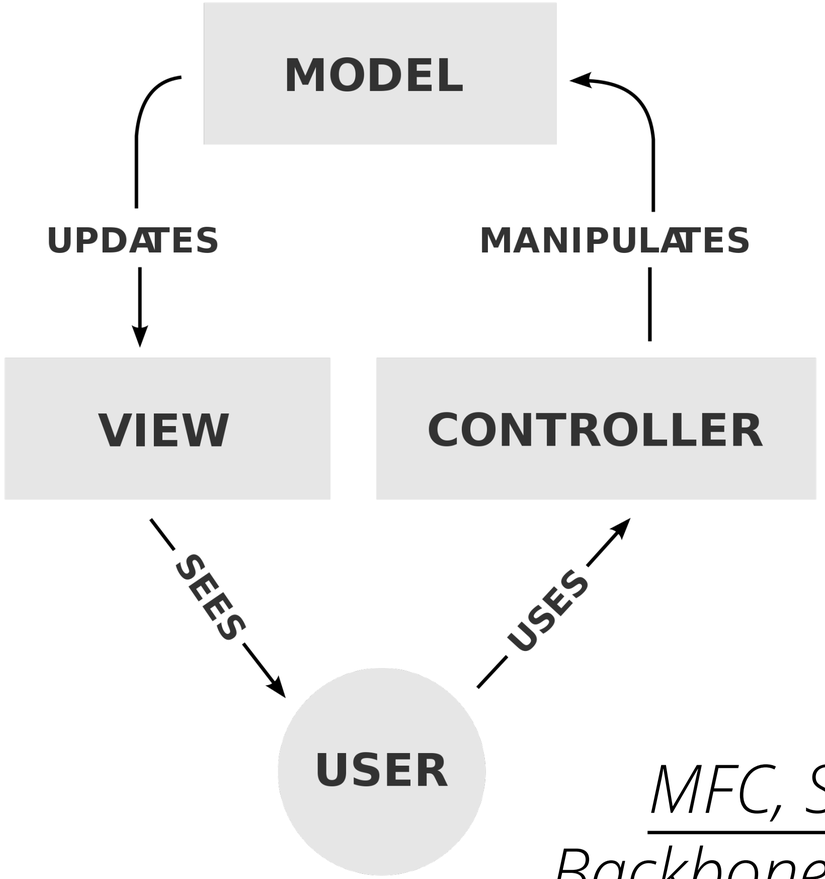
MVC *MVP* *MVVM* *Viper*

Clean

组件化

MVC





MFC, Swing, iOS, Backbone, PureMVC

Model: 业务逻辑相关的**数据**、对数据的**处理方法**

View: 实现数据有目的的显示，非必需，无逻辑，监听模型变更

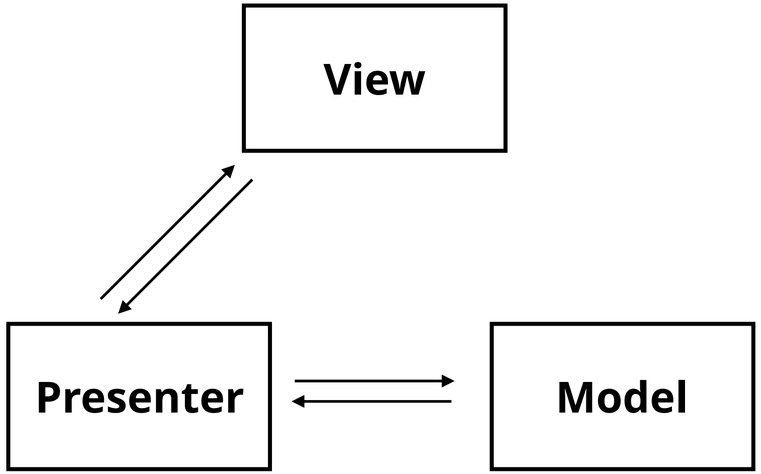
Controller: 组织View和Model，控制流程，处理事件（用户的行为和数据 Model 上的改变）

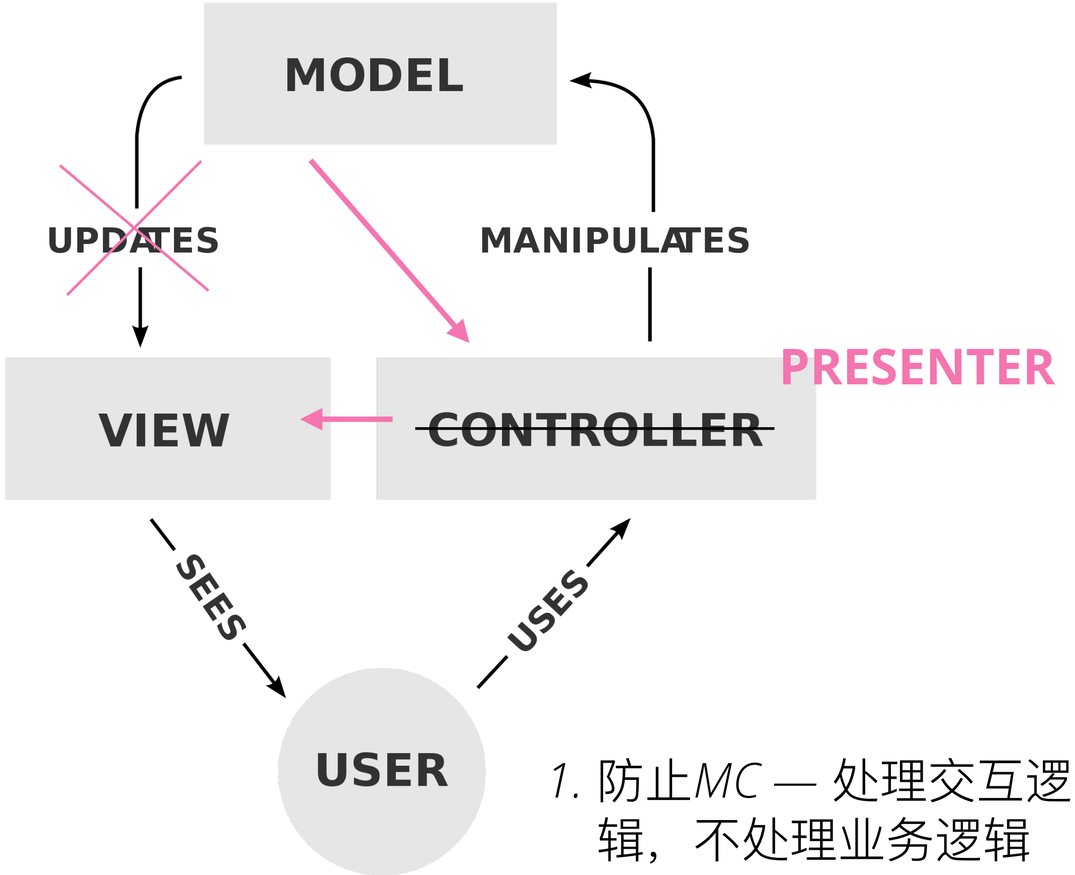
Massive View Controller

1. 事件处理
2. 复杂的交互处理
3. 给后端同步数据
4. 业务逻辑
5. 更新模型
6. 准备数据给View展示
7. ...

```
const { ...props } = this.props;
const { dataSource, isMobile } = props;
delete props.dataSource;
delete props.isMobile;
let clearFloatNum = 0;
const children = dataSource.block.children.map((item, i) => {
  const childObj = item.children;
  const delay = isMobile ? i * 50 : this.getDelay(i, 24 / item.md)
  const liAnim = {
    opacity: 0,
    type: 'from',
    ease: 'easeOutQuad',
    delay,
  };
  const childrenAnim = { ...liAnim, x: '+=10', delay: delay + 100
  clearFloatNum += item.md;
  clearFloatNum = clearFloatNum > 24 ? 0 : clearFloatNum;
  return (
    <TweenOne
      component={Col}
      animation={liAnim}
      key={item.name}
      {...item}
      componentProps={{ md: item.md, xs: item.xs }}
      className={
        !clearFloatNum
          ? `${item.className || ''} clear-both`.trim()
          : item.className
      }
    >
      <TweenOne
        animation={{
          x: '-=10',
          opacity: 0,
          type: 'from',
          ease: 'easeOutQuad',
        }}
        key="img"
        {...childObj.icon}
      >
        <img src={childObj.icon.children} width="100%" alt="img" /
      </TweenOne>
      <div {...childObj.textWrapper}>
        <TweenOne
          key="h2"
          animation={childrenAnim}
          component="h2"
          {...childObj.title}
        >
          {childObj.title.children}
        </TweenOne>
        <TweenOne
```

MVP

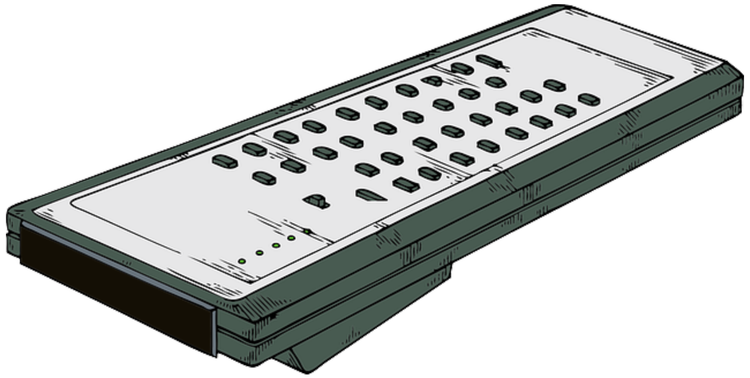


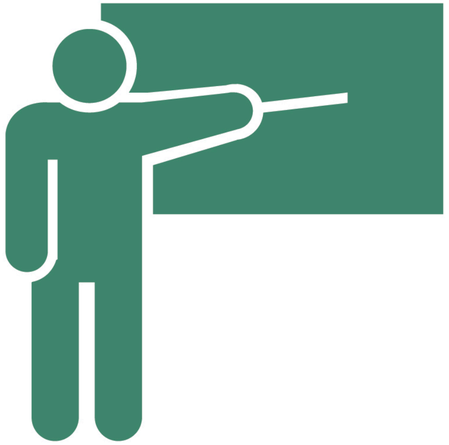


1. 防止*MC* — 处理交互逻辑，不处理业务逻辑
2. 辅助存储*View*的临时数据
3. *View*与*Model*不直接通信
4. *Passive View*

Presenter: 事件处理，获取Model数据，格式转换，与View通信。

改进点在哪里?



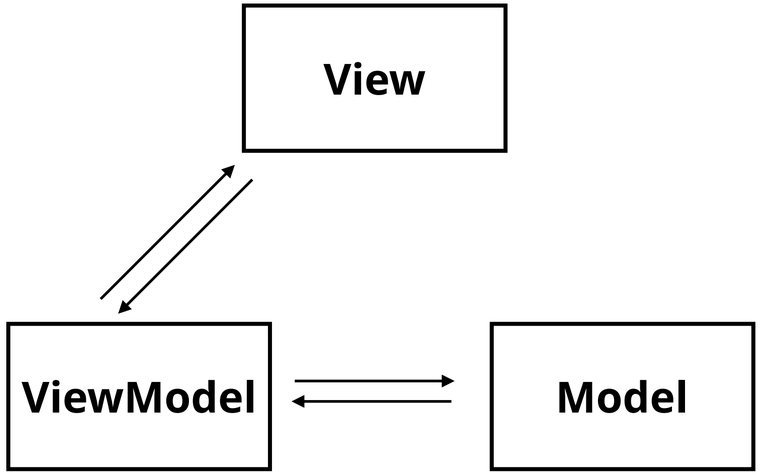


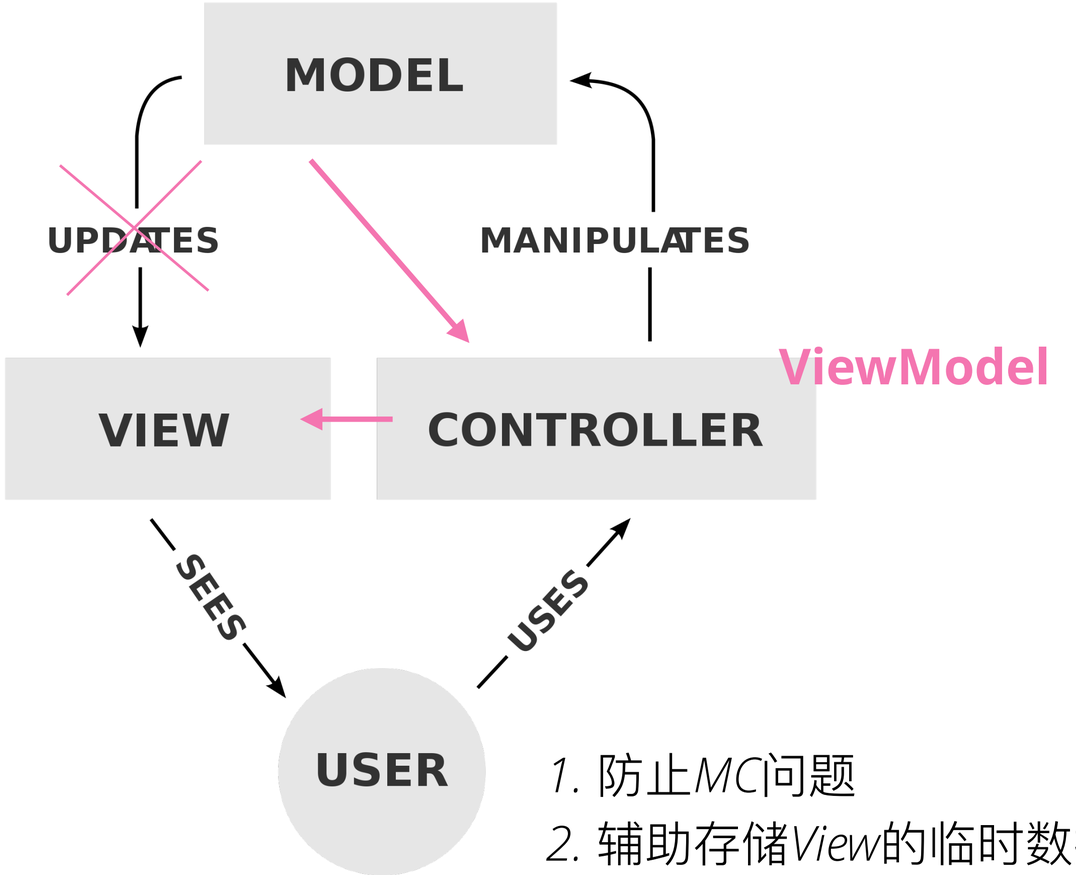
Controller → Presenter

Passive View

1. Stupid View：只负责渲染
2. 事件直接传递给Presenter — 没有业务逻辑
3. 不修改数据 — 强调单向数据流

MVVM





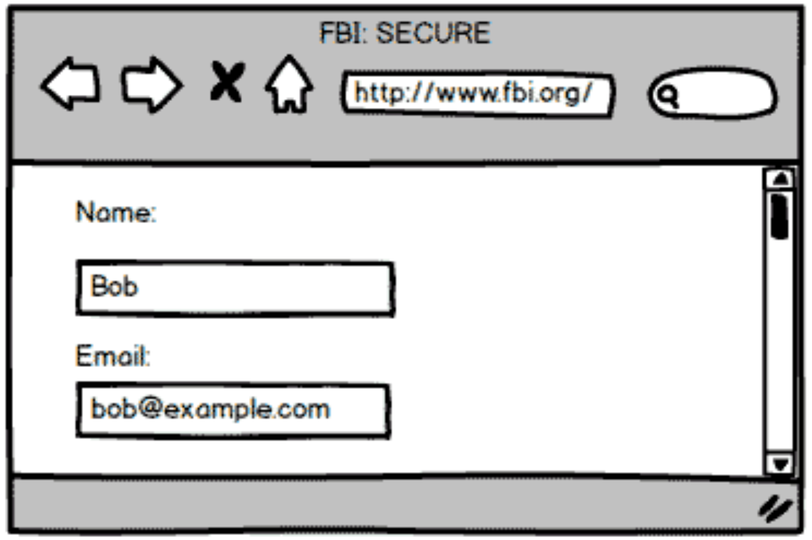
1. 防止*MC*问题
2. 辅助存储*View*的临时数据
3. *View*与*Model*不直接通信
4. 通常实现为*Passive View*
5. 数据双向绑定

ViewModel: 事件处理，获取Model数据，格式转换，与View通信。

数据双向绑定

```
// javascript:
var data = {
  name: 'Bob',
  email: 'bob@example.com'
};
```

```
<!-- html -->
<input name="name" type="text">
<input name="email" type="text">
```

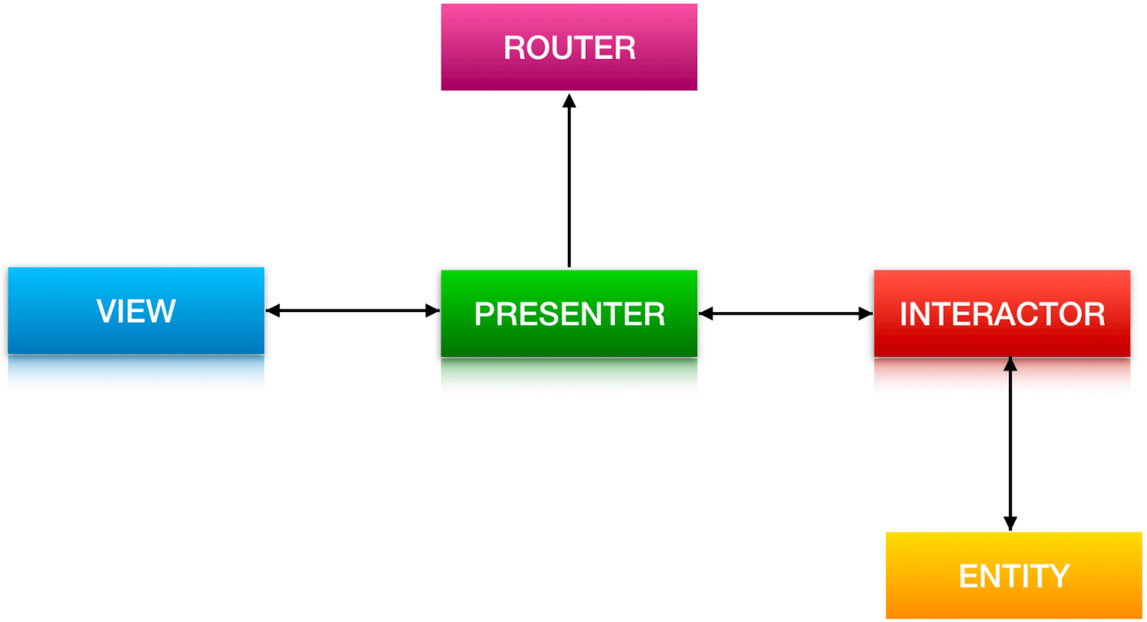


代码变简单了，但是

1. 应用复杂时，难以理解和调试
2. UI状态容易侵入到模型层

Viper & Ribs

View, Interactor, Presenter, Entity, Router



- View — Send actions to the presenter, show presenter asks it to
- Interactor — Business logic
- Presenter — View logic, reacting to user interactions
- Entity — Model used by interactor
- Router — Logic for routing of screens.

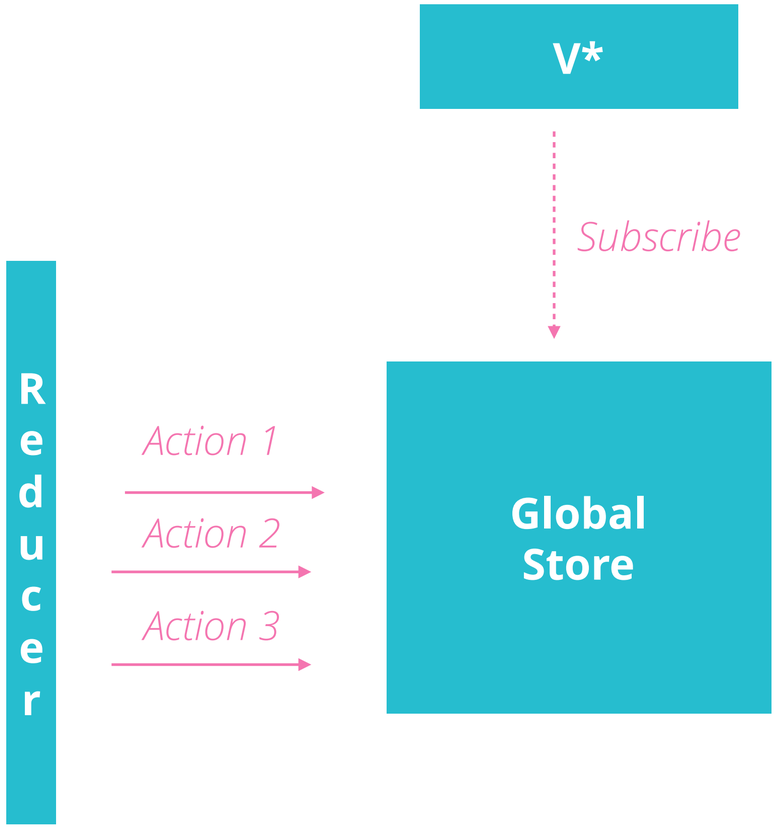
1. 业务驱动应用，而不是*UI*驱动
2. 业务和*UI*保持独立

https://github.com/GABHISEKBUNTY/Viper-Architecture

https://github.com/uber/RIBs/

前端的其他组件

Status Store

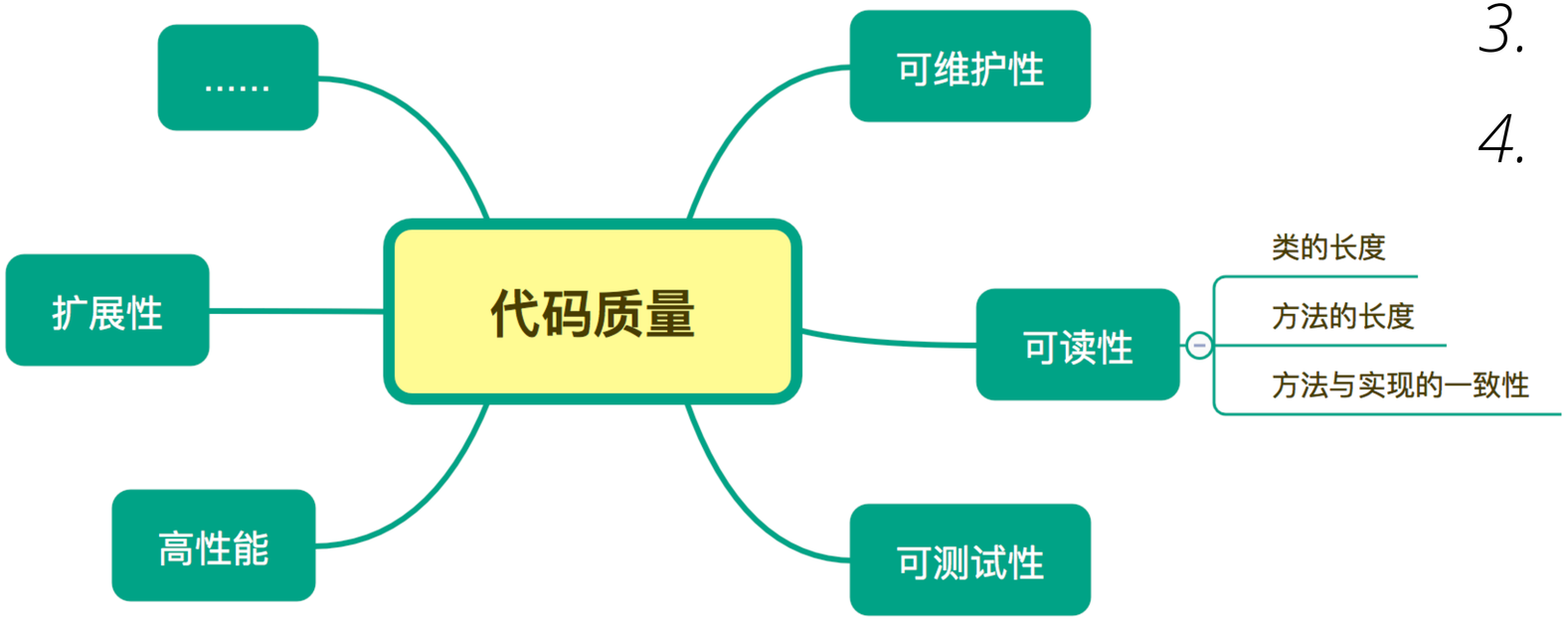


1. 减少请求次数
2. 独立的状态管理 — 对比数据库的概念
3. 便于面向领域进行设计

ThoughtWorks®

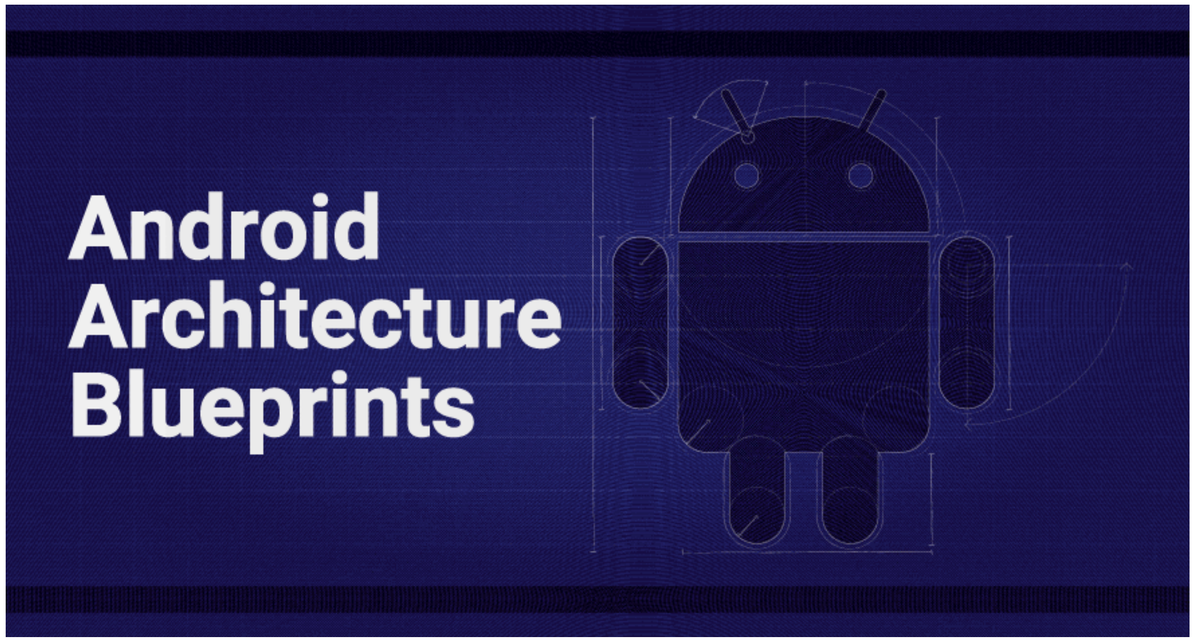
Service, HttpClient, Directive, Pipe, ...

组件设计与好代码的标准



1. 为复用而拆分
2. 为降低复杂度而拆分
3. 单向数据流 — *Passive View*
4. 谨慎的状态管理

 ThoughtWorks®



todo-mvp

todo-mvp-clean

todo-mvp-dagger

todo-mvp-rxjava

todo-mvvm-databinding

todo-mvvm-live

https://github.com/googlesamples/android-architecture