Published in 程式愛好者

YC Following
Aug 26, 2020 · 4 min read

使人疯狂的SOLID 原则：开放封闭原则(Open-Closed Principle)

> *今天我们要说的是第二个原则：开放封闭原则(OCP)。*

相比第一篇 SRP 原则来说，开放封闭原则可以说是好懂三百倍了。

> *定义：一个软件制品应该对于扩展是开放的，但对于修改是封闭的。*

这样的中文定义老实我个人认为逻辑不够严谨。这样的定义是指「一个软件应该对于扩展是开放的***或***一个软件应该对修改是封闭的。」还是「一个软件制品在面对扩展时是开放的，且扩充时不应修改到原有的程式。」呢?

> *维基百科的定义是：*
> *系统一旦完成，一个类的实现只应该因错误而修改，新的或者改变的特性应该通过新建不同的类实现。*
>
> *但在 Uncle Bob 的文章中，他定义为：*
> *You should be able to extend the behavior of a system* without having to modify that system.

我们清楚理解到这是一个「且」的关系，即：

> *一个软件制品在面对扩展时是开放的，且扩充时不应修改到原有的程式。*

对，就是这么简单！

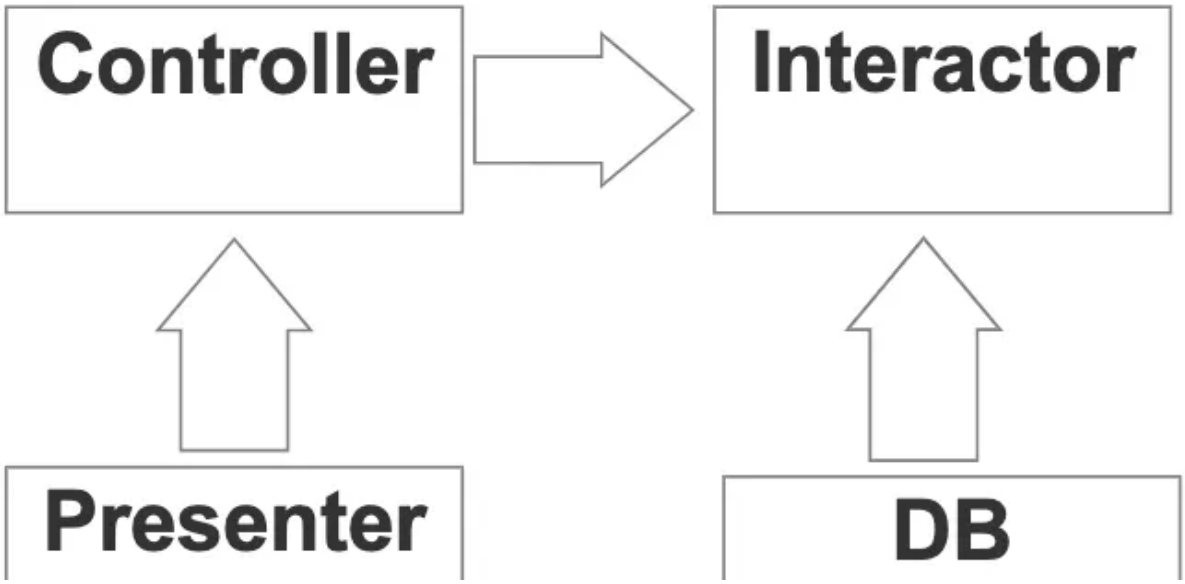
当年他们会提出这样的原则是有理由的，比如说 Linux Kernel，他们尽量不希望在为一个稳定的 Linux Kernel 版本添加功能时，因为会改动到原有的程序码而使 Kernel 也要做出大量的修改了，然后出现很多新的 Bug。而是希望在添加功能时，原有稳定的版本还是保持一致，否则 debug 起来就是地狱。

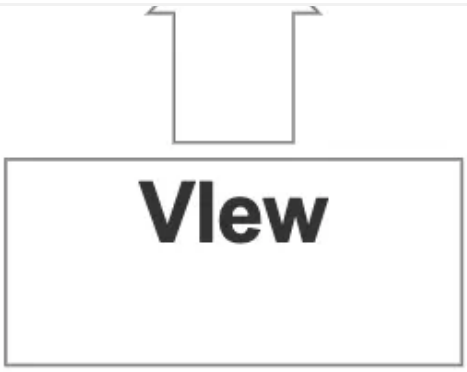
我们可以看到最美妙地遵守 OCP 的几个常用程式，如 Google Chrome、VSCode等等。我们可以轻易为他们添加很多的 plugin，且不会对原来的主体造成影响。

那怎样才能达成 OCP？简单来说，透过 SRP 我们可以就不同因素而改变的模块分类好，再透过 DIP (依赖反向，之后的文章我们会更详细的说到)来为系统创建一个单向的流程。

具体来说， 我们在说架构时都会把程式分成很多层。而在系统中的最高层，通常都是业务逻辑层，其他层次都是围绕着业务逻辑层而进行分工。而这种分工比较像以下的架构

图：





在上图中，整个架构的内核是 Interactor (业务逻辑层)，他会被 Controller 与 DB 所依赖。而 Presenter 又会依赖于 Controller 。

> *这层层的单向依赖有效于解耦。*

在软件设计中，组件不应依赖于不会直接使用到的东西，如 Interactor 不会直接使用到 View， Interactor 跟 View 之间当然不应有着依赖的关系。同理地，作为底层的组件只

需做好自己的「本份」，如 View 就应该是只处理视图的逻辑，不应也不需要知道 Presenter 是在处理什么逻辑，这样的单向依赖可以让高层组件免受非有依赖关系的低层组件改变影响。

> *所以在设计架构时，我们要根据如何、为什么及何时发生变更来分离功能，然后将分离的功能组织到组件阶层中。*

说了这么多，OCP 不是要告诉我们程式该怎么分层、工作该怎么写好，而是给出一个解耦的概念，我们应该要朝着就算[illegible]功能，我们还是不会影响到原有的程式，更能优雅地设计着！

优雅地设计一个「即使需要功能扩充，亦不需要修改原程序码的系统」。而这样的目标就有赖于我们怎么样设计我们的程式架构了！

如果你觉得我的文章帮助到你，希望你也可以为文章拍手，分别 Follow 我的个人页与程式爱好者出版，按赞我们的粉丝页喔，支持我们推出更多更好的内容创作！

Solid Ocp Software Architecture Software Development Object Oriented