代码评审标准

好代码和坏代码的差别

当一个功能被完成的时候，一个不怎么懂编程技术的老板将会从哪几个维度来评价 这个程序员完成的这个工作呢？完成这个功能花费的时间一定是其中一个要素，除此之外就是这个功能是否贴合了需求，贴合了需求之后就是测试在真实运行环境下是否运作正常不报错。我想这就是一般老板考察我们工作质量的几个维度。但问题是，如果已经满足了以上几点，这里是否还有我们可以作出努力的地方呢？我想答案是：肯定有的。

为了完成同一个功能，每个程序员写出来的代码都不可能一样，甚至同一个程序相隔一段时间分两次写的代码也不太可能完全一致。这么说起来，代码从零到最终成型的样子仿佛是一个不可控的随缘的过程，难道写代码真的是如此佛性不做作的一件事吗？我承认一定程度上是“条条大路通罗马”，但这条条大路路况不一定都一样吧，只要这其中有差别，我想就有高下之分。那我们看见完成了同一个功能的两段不同的代码，我们应该基于一些什么准则来做出一个是好是坏的评价？

代码质量的评价标准某种意义上类似于文学作品，由个体主观评价然后形成一个相对客观的评价。

但代码与文学作品不一样的地方是，代码实际上有两个读者：计算机和程序员。

所以对于代码质量的定义我们需要从两个维度来分析：主观的，被人类理解的部分；还有客观的，计算机里的运行情况。

**计算机运行效率的维度：**

目的：运行效率高

手段：优化执行逻辑、减少执行时间、减少重复计算、对象池（空间换时间）......

**程序员方便理解的维度：**

目的：可读性

手段：命名方法、函数设计、格式、注释......

**程序员易于修改的维度：**

目的：可维护、可复用、可拓展、灵活性好

手段：设计模式六大原则......

**可读性量化标准**

在很多跟代码质量有关的书里都强调了一个观点：程序首先是给人看的，其次才是能被机器执行。

在评价可读性这个维度的好坏的时候，可采用一种方法，就是将代码翻译成中文读给其他程序员听。不可避免的会有需要联系上下文才能理解的句子，但这就是相对可以量化的地方，如果需要联系的上下文越多，意味着代码的可读性越差。

**BUG是如何产生的**

有些时候一个原本运行良好的功能，在根本就没有动它的情况下，莫名其妙就坏掉了。写代码的时候，我们也许会或多或少地意淫出一些看似正确的地基，然后在这些所谓正确的地基之上构建逻辑架构。这些地基也可以被称为前提条件，那我们的程序只有在这些前提条件下才能正确地运行，一旦这些前提条件不被满足，BUG便由此产生了。我认为的BUG产生的大多数原因就是，原本代码逻辑的前置逻辑变更了，或是出现了你没有想到的前置逻辑。

针对于此，我们要做的则应该是将可能出现的前置逻辑考虑周全，并让自己的代码做到可以应对这些考虑到的情况。但这个问题还有着另一面，就是如果代码可以应对的情况根本就不会出现，并且代码为这种不可能出现的情况做出了只针对于它的努力，那就产生了代码执行的冗余。所以现实中代码面对项目的需求而提供的实现量，随着项目周期的推进往往是 由简入繁，再化繁为简的过程。由简入繁就是开发，化繁为简即是优化。打个不那么恰当的比方，开发的时候就是尽量把口袋做大，以至于可以装下尽量多的需求；想要的需求都放进来之后，则是尽量把口袋中的空气都排挤出来，以至于口袋的体积是尽量小的。项目开发，即是需求与实现两者互相匹配的过程。

**代码的重构**

重构的定义：重构就是在不改变软件系统外部行为的前提下，改善它的内部结构。

项目的需求（即代码需要应对的情况）总是不断地迭代的，这是很正常的情况。一个好的程序架构，可以做到面对需求一定程度的变更（无论是增删还是改）都能轻松应对，好的程序架构对于需求应该是支持而不是限制。产品要做的就是将未来的各种需求，尽可能地考虑周全；而程序要做的就是要将未来的各种需求，都囊括到你现在写的程序架构可以尽可能轻松实现的范畴之中。如果两边都能完全地做到，我相信，世界将美好很多。理想是丰满的，但现实往往很骨感。

我们不能要求产品将未来的各种需求都细无巨细地考虑完全，所以一旦有超出目前程序架构能实现的需求出现，可能就面临着重构。当然这个时候，你也可以在老架构之上先将需求实现，这之后再来考虑重构的事，但这往往是代码变烂的开始，因为你已经欠下一笔账了。当这个账欠到一定程度多的时候，写代码将无异于屎上盖楼；或者这个时候你打算重构了，那就先把这个屎上的楼房拆了，再用带屎的砖头盖新楼。所以我认为重构代码应该是即时的（仅限于模块内的重构，项目重构涉及到项目周期等更多因素）。

重构再定义：保留对原有功能的支持，再增加对未来需求的适应性。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 条款 | 说明 | 工具 | 5分 | 4分 | 3分 | 2分 | 1分 |
| 平均函数代码行 | 反映函数的单一职责性。单一职责即功能内聚。行数越多，可能是函数承担了太多职责造成的，往往隐含故障就越多，构建和维护成本就越高 | SourceMonitor | <5 | 5-10 | 10-15 | 15-20 | >20 |
| 平均函数复杂度 | 反映函数的复杂性。越复杂，函数隐含的故障就越多，构建和维护成本就越高。复杂度定义见代码大全19.6 | SourceMonitor | <3 | 3-4 | 4-5 | 5-6 | >6 |
| 最大函数复杂度 | 反映函数的复杂性。越复杂，函数隐含的故障就越多，构建和维护成本就越高。复杂度定义见代码大全19.6 | SourceMonitor | <8 | 8-10 | 10-20 | 20-30 | >30 |
| 注释率 | 注释在一定程度可以帮助理解代码，提升可维护性。但需要注意不要加可有可无的注释，应追求代码自注释。 | SourceMonitor | 15%-25% | |  |  | | --- | --- | | 25%-40% |  | | 40%-50% | >50% | <15% |
| 平均块深度 | 块深度是指被if/for/while等包围的嵌套层次。嵌套越多，逻辑越复杂，代码可读性越差，构建和维护成本就越高 | SourceMonitor | <1.5 | 1.5-2 | 2-2.5 | 2.5-3 | >3.0 |

**编码质量评分标准：**