代码质量

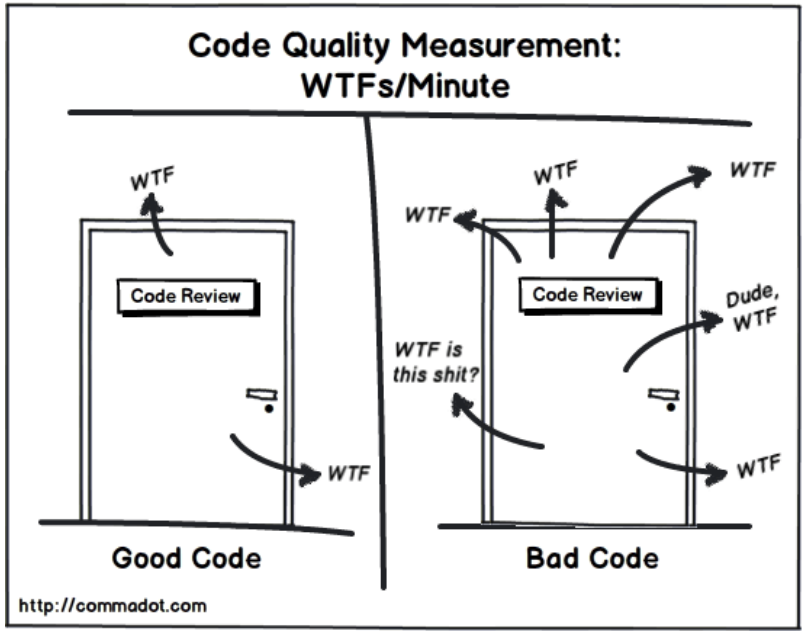
什么是好代码

好代码的前提是实现其预期目标——实现预想的功能、不会在运行时发生意料之外的异常、没有安全性漏洞等。

运行不出错的代码就是好代码。 

除此以外，好代码还应具有代码结构设计合理、命名规范、注释文档充分有效等特点。即使过了很长一段时间，代码的开发者稍稍浏览代码，便可以接续上写代码时的思路。想要复用或修改代码的其他人也可以很容易地了解代码的调用方法、理解代码的意义。

同事吐槽少的代码是好代码。 



为什么要重视代码质量

- 对于庞大软件开发团队中的一员，工作中花费时间最长的环节，往往是看懂别人写的代码。
- **代码是软件工程师用来沟通想法和目标的媒介**。
- 代码质量关系到一段代码在较长时期内**好不好复用、好不好维护**。
- 高质量的代码更容易理解、维护和扩展，从而降低后续开发的难度，不仅方便他人，也方便自己。
- 具有工匠精神的软件工程师在写代码时，不仅要关注语法是否正确、功能是否可用，更要时刻关注自己出品的代码质量——**为自己的代码负责**。

提高代码质量的十点建议

为自己的代码负责

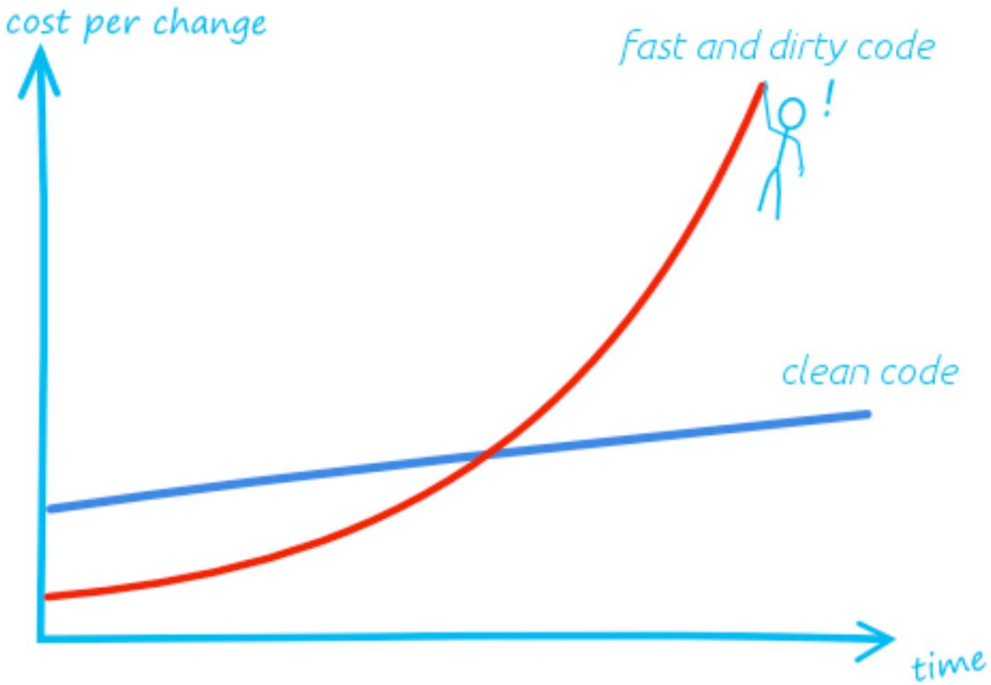
Tip1

从某个角度来说，成为一个专家需要坦诚面对自己花了多少时间把一件事做好，而不能容许自己把一件事情做糟。

- “Does one thing well” - Bjarne Stroustrup
- “Reads like well written prose” - Grady Booch
- “Can be read and enhanced by a developer other than its original author” - Dave Thomas
- “Always looks like it was written by someone who cares” - Michael Feathers
- “Contains no duplication” - Ron Jeffries
- “Turns out to be pretty much what you expected” - Ward Cunningham

为自己的代码负责

Tip1

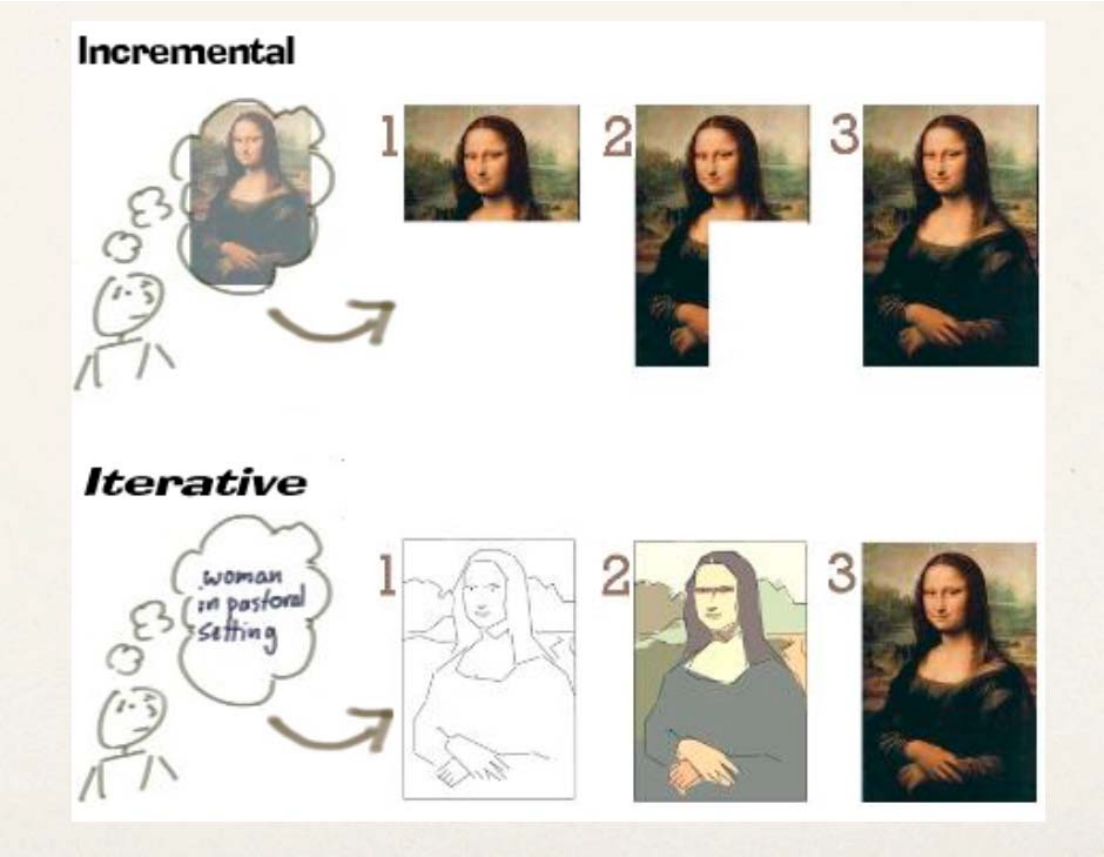


我们以为我们把时间都花在了写代码上，但其实我们把大量的时间花在了读懂已经写好的代码上。据统计，读代码的时间与写代码的时间之比大约是10:1。

不要急
急诊室医生
不会手忙脚乱地做手术
写代码
需要工匠精神

每次只做一件小事

Tip2



学会将一个巨大的问题分解成一个又一个小问题，并以**增量式**或**迭代式**工作。

增量式：把大问题分解成相互独立的若干小问题，每次保证解决一个小问题，最后再将其合并。

迭代式：将大问题分解成若干步骤，并逐步完成细节（如图所示勾线—上底色—画细节）。每次只做好本步的工作，并且保证工作的效果。

使用有意义的命名

Tip3

- 不管是类、函数或是变量，请使用有意义的命名——从名称的英文中就可以知道它为什么存在、做什么用。
- 几种常见的命名格式：
- **下划线式命名**：int room_height = 0;
- **驼峰式命名**：分为两种，一种称为大驼峰式（Pascal式），一种称为小驼峰式。例如：
  - 大驼峰式：int RoomHeight = 0;
  - 小驼峰式：int roomHeight = 0;
- **匈牙利式命名**：int htRoom = 0;（用ht代表高度）
- 不管采用哪种命名，整个代码项目中应尽量统一。

不好的命名

```
int h = 0;
// h is the height of the room
```

当h的值在程序中变来变去，很容易搞不清楚h是什么。

好的命名

```
int room_height = 0;
```

即使不需要注释，到哪里都知道这个变量表示房间高度。

像写故事一样写代码 Tip4

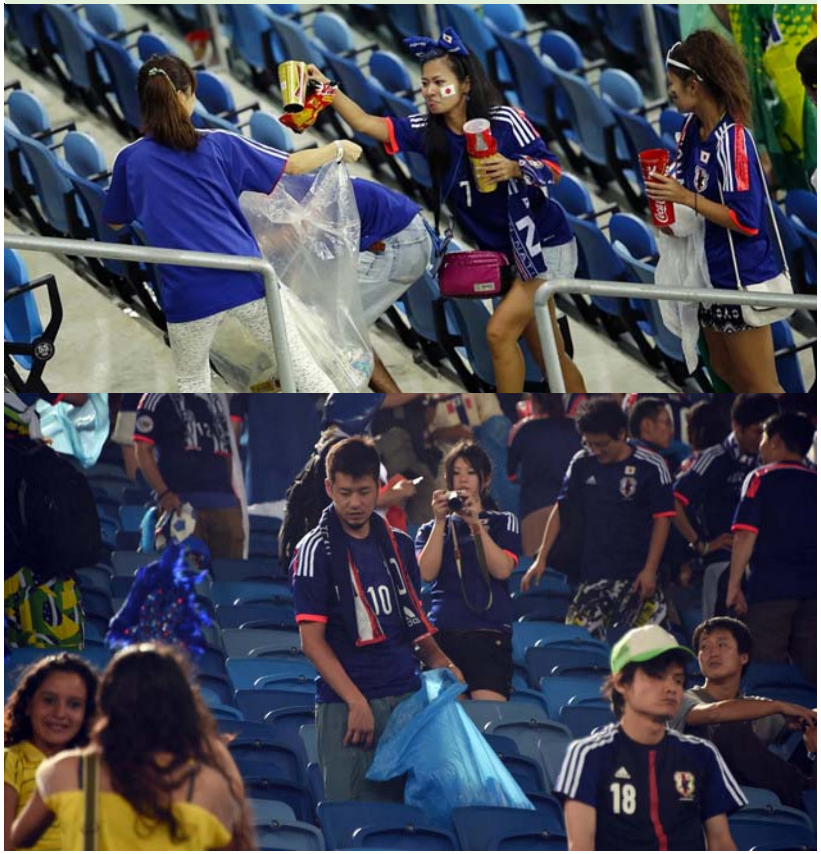
- 函数应该具有明确的名称表明其作用；
- 定义函数时应按其逻辑顺序排列；
- 尽量减少函数内部 if…else…的层数；
- 能在一个函数中解决的问题没有必要拆分成几个函数（除非函数太长）。

不要写没有意义的注释 Tip5

- 高质量的代码自己就可以解释自己在做什么。
- **当你想写注释时，请暂停！想一想除了这种低端方法之外，还有什么方法可以让自己的代码更好理解。**

手下留情 Tip6

当你离开时，请让代码比你来时更好



少即是多 Tip7

单一功能原则：每个类（函数）都应该有一个单一的功能，并且该功能应该由这个类（函数）完全封装起来。

可以做，并不总是意味着应该做

避免无意义的复杂化 Tip8

能读懂的、适当抽象凝练的代码体现了编程者的智慧

读不懂的、故弄玄虚的复杂代码体现了编程者的愚蠢

代码风格保持一致 Tip9

勤能补拙 Tip10

C# XML注释

- 什么样的注释是“有意义的”注释?
- 假如你是一个庞大的软件开发团队中的一员，要让别人看懂你写的代码，最重要的是给你的每个**类**及其**成员变量**、**成员函数**加上一些通俗易懂的描述，对**函数参数**和**返回值**给出明确的说明。
- 幸运的是，几乎每一种常用的编程语言，都有辅助程序员完成上述注释说明的插件。Microsoft Visual Studio集成开发环境原生支持C#语言的XML注释。它通过<summary><param><returns>等标签，为每个类和函数添加描述。通过Sandcastle Help File Builder等工具，还可以方便地生成网页班的代码说明手册。
- doxygen(C++)、docstring(Python)、readthedocs等工具，希望大家举一反三，自主学习。

Code Review
单元测试
重构

Code Review

每个人编程时的思维方式不同，因为自己总是按照习惯的逻辑去思考，自己检查代码并不容易发现问题。但让别人审查代码，他会从另一种思路去思考代码，因而更容易发现问题，或是帮你发现不符合程序开发规范的地方。

大型软件团队的每个成员，在将自己写的代码合并进代码仓库前，一般都需要邀请至少一个同事审查自己的代码。

Code Review 关注的问题：

代码格式：每个团队都会对代码格式有一些约定俗成的规范，代码审查可以发现那些违反规范的地方。

代码可读性：函数、变量是否命名得当？函数是否逻辑清晰？函数是否过长？是否有冗余或模棱两可的注释？

异常处理：作者是否忘记验证用户输入的合法性？是否忘记验证对象为空的情况？是否忘记catch程序运行过程中可能抛出的异常？

Corner Case（极端情况）：函数中的if...else...是否封闭？是否忽略了某些非常少见的执行逻辑？

架构合理性：重复代码段应提取为函数，常量应在文件开头全局定义，某个函数在另一个类中定义会更好……

软件测试

单元测试 → 集成测试 → 系统测试 → 验收测试

单元测试

单元测试是对软件中的**最小可验证单元进行检查和验证**。比如对代码中的类和方法的测试。

集成测试

集成测试是在单元测试的基础上，把软件单元按照软件概要设计规格说明的规格要求，**组装成模块、子系统或系统**的过程中各部分工作是否达到或实现相应技术指标及要求。

系统测试

将经过集成测试的软件，作为计算机系统的一部分，与系统中其他部分结合起来，**在实际运行环境下进行一系列严格有效的测试**，以发现软件潜在的问题，保证系统的正常运行。

验收测试

也称**交付测试**，是针对用户需求、业务流程进行的正式的测试，**以确定系统是否满足验收标准**，由用户、客户或其他授权机构决定是否接受系统。

单元测试

- 单元测试是指开发者编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确。
- 单元测试用于判断某个特定条件下某个特定函数的行为。例如给定一组输入和一个预想的输出，如果函数输出的结果与预想输出一致，则说明函数的运行在这种情形下是正确的。
- 假如我们能保证软件的每个最小单元（模块、类、函数等）都可以正确地工作，那么最终得到的软件产品很可能也可以正确工作。
- 我们每天都在做单元测试。你写完一个函数总是要执行一下，看看功能是否正常，这种单元测试称为临时单元测试，这种不规范的单元测试往往容易遗漏问题。
- 正规的软件开发团队往往需要借助专业的工具来进行全面的单元测试。我们以Visual Studio中支持的MS TEST为例，对四则运算类进行单元测试。

重构

- **重构**：在不改变软件系统外部行为的前提下，改善它的内部结构。通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性。
- **为什么要重构**：软件产品最初制造出来，是经过精心的设计，具有**良好架构**的。但是随着时间的发展、需求的变化，必须不断的**修改原有的功能**、**追加新的功能**，还免不了有一些**缺陷需要修改**。为了实现变更，不可避免的要违反最初的设计构架。经过一段时间以后，软件的架构就千疮百孔了。重构就能够最大限度的避免这样一种现象。系统发展到一定阶段后，使用重构的方式，不改变系统的外部功能，只对内部的结构进行重新的整理，使系统对于需求的变更始终具有较强的适应能力。

什么时候需要重构

- **项目中存在重复代码**：如果同一个类中有相同的代码块，请把它提炼成类的一个独立方法，如果不同类中具有相同的代码，请把它提炼成一个新类，永远不要重复代码。
- **存在功能相近的不同代码**：团队成员不约而同地开发了相似的功能。取长补短，最终保留一个就好。
- **存在模棱两可的变量名或方法名。**修改变量名或方法名。如果是公有方法，注意同时修改调用的位置。
- **臃肿的类**：可能属于把子类的方法包含到父类中、把其它类的方法包含到本类中、把值得提炼出来反复利用的方法写进某个具体类中等情形，需要根据其成因加以解决。
- **过长的方法**：包含的逻辑过于复杂，需要将其分解为多个小方法。
- **牵一发而动全身的修改**：当你发现修改或增加一个小功能牵涉到改动整个项目的很多地方，就说明程序设计不够理想、功能代码太过分散。需要将分散的功能整合到一起。
- **类与类之间联系过密：**A类的方法反复调用B类的变量或方法，则这两个类逻辑上本应合并在一起。
- **类与类中间环节过多**：A类的方法每次调用B类方法时，都需要经过多级中转，应考虑直接让A操作B。
- **最初没有考虑清楚的设计**：随着后续需求的不断变化，很多最初没有仔细考虑的设计也有必要作出改动。
- **趁着还记得自己写了什么，赶紧补充注释。**

作业

- 基于第二次作业的工程项目，**全组成员分工**合作为昨天作业的代码编写单元测试代码，并根据你们的想法重构该代码，使其结构、性能更加优化。
- 基于第二次作业的工程项目，撰写注释，并使用自动文档生成工具生成说明文档。
- 第三次作业提交截止时间：2022年8月30日结题答辩之前，将作业打包发到助教邮箱，如果未收到回复，三天后再发送一次。