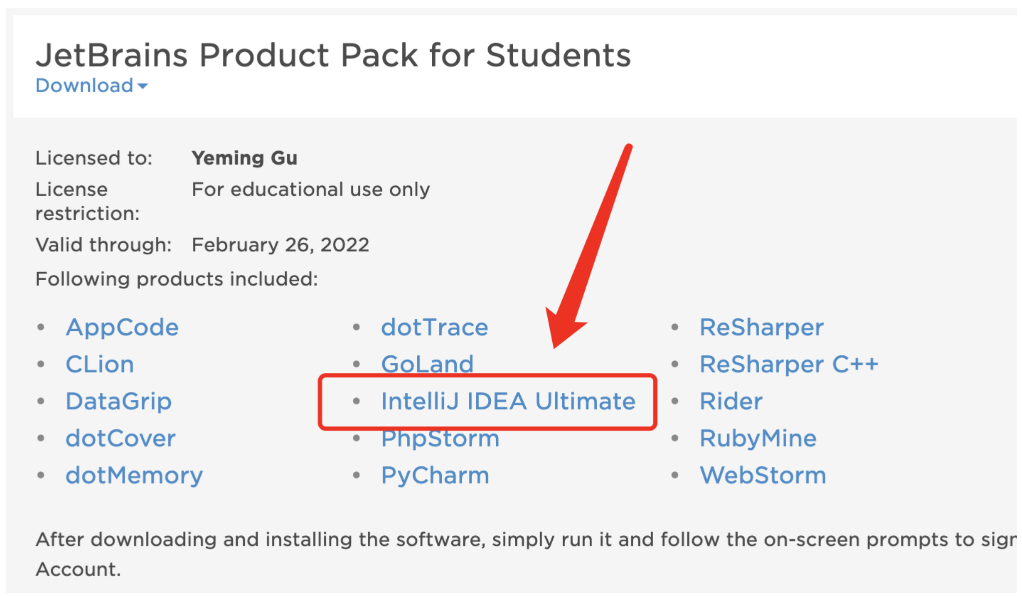
开源实训Lab-01：软件开发基础

作者：熊节
内容原创来自微信号: programmer_gym，由顾业鸣编辑整理。仅供课程学习，请勿传播。

请先安装 IntelliJ IDEA
首先，进入JetBrain官网，登录/注册。
其次，访问https://www.jetbrains.com/shop/eform/students 按要求申请学生认证。
最后，进入JetBrain的账户个人页面，就能看到有一个教育License，选择Idea下载即可。

1 License



如果你觉得这样麻烦，你可以直接下载云盘中的 community版本
链接: https://pan.baidu.com/s/1bnTNMp_dlnZk6Q0-O-t8yQ 提取码: tu9y

一、前言

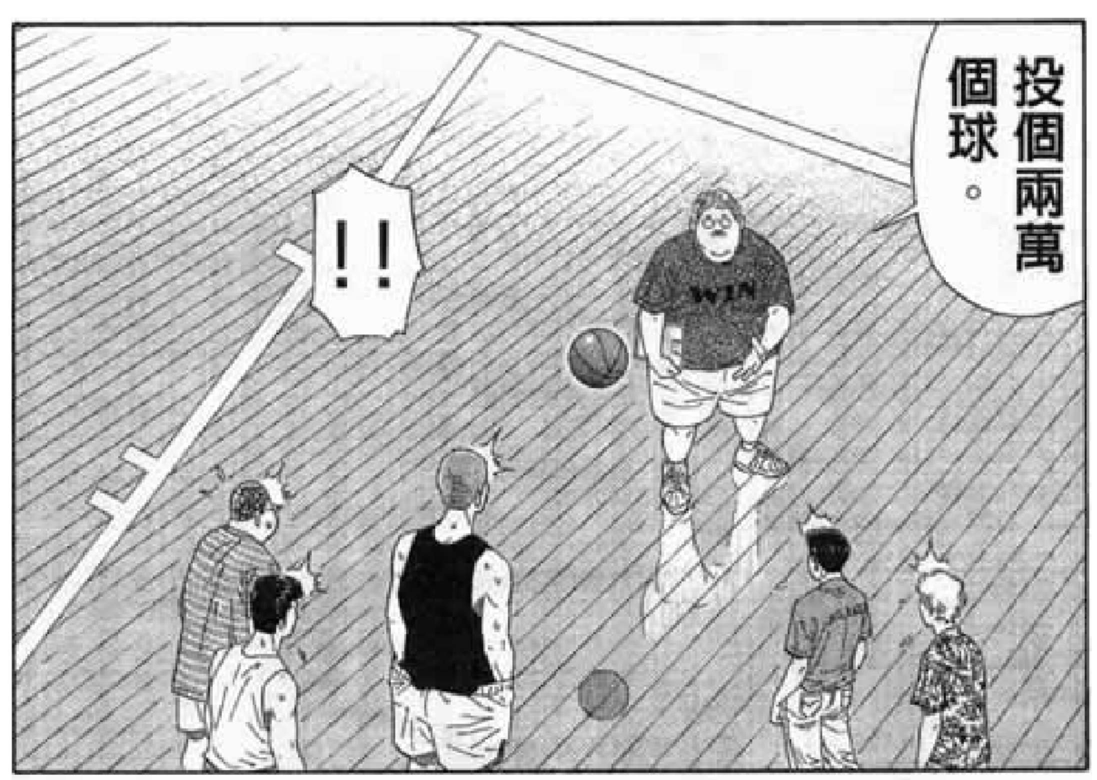
我翻译过的最有名的一本书《重构》，很多人跟我说，读过，太经典了。然而我在咱们行业里看过这么多项目、这么多团队，没见过几个人正经会重构的。发行十几万册的书，对读者的影响如此有限，这问题到底出在哪儿？

后来是仝键老师的一篇文章点醒了梦中人：

培训到底有没有用？当充分准备好的人，遇到充分准备好的内容时，是有用的。而这里面最难满足的条件是充分准备好的人。这是可遇而不可求的。

编程是一门手艺活，这个世界上的手艺都是练会的。基本功不熟练，说别的都是骗人。

然后有些同学会说，好啊，那你教我基本功嘛。其实这事在现代企业管理名著《灌篮高手》里面已经讲得很清楚了。安西教练是什么时候开始教樱木花道投篮的呢?



是在樱木开始大量重复练习之后，安西教练才开始给他提供反馈和指导。

所以具体到编程这件事，一定是要把练习放在第一位，一定是要驱动着学习者去做大量重复的练习，然后对这些练习中出现的问题提出反馈和指导，学习者才会真的有所收获。

于是我就做了「TDD 练功房」这个在线的培训。形式很简单：一道非常轻松的编程题目，来练个二十遍吧。不是不会 TDD 么？练上二十遍你就会了。轻松的题目会做了，再换到稍微复杂一点的题目，多练几遍，你也一样会做。题目练得多了，转换到工作中也会做了。

就这么一个简单的形式，收到了意想不到的效果。很多学员跟我说，经过这一个月的练习，编程能力有了很大的提升。于是，我感觉已经可以把练功房的过程以一种更传统、更具象的形式呈现出来，让更多的人了解这种“非传统”的学习方式。

这就是我现在开始做的「TDD 练功房」系列文章连载了。这个系列文章的想法跟我们在线的练功房课程一样，不要指望一口吃成个胖子，每天读一点、练一点、反思一点，两个月以后你也能看到自己编程能力的大幅提升。

总而言之一句话：功不练不成。练功房已经为你准备好了，就请你开练吧！

二、练功前的热身

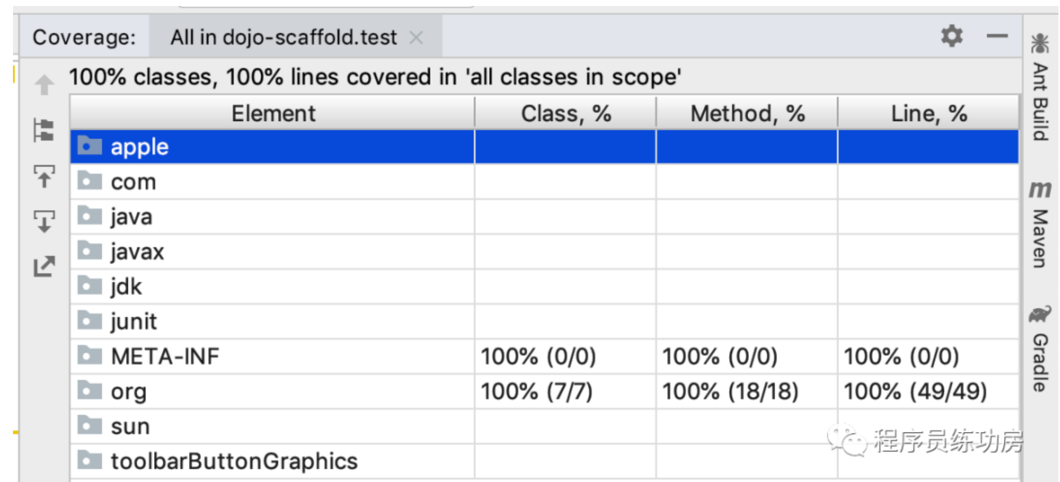
> ✏️**今天的任务**
> 写一个程序，打印出从 1 到 100 的数字，将其中 3 的倍数替换成“Fizz”，5 的倍数替换成“Buzz”。既能被 3 整除、又能被 5 整除的数则替换成“FizzBuzz”。
> 要求：每行代码都必须有单元测试覆盖。

看完 FizzBuzz 的题目描述，是不是都有点想要嗤之以鼻了？“我满怀希望进来练功房，你就给我看这个？ ”

别着急。现在你就放下书，开始掐表计时，把这道题从头到尾做一遍。另外别忘了，我们这是 TDD 练功房，所以你的实现必须有充分的单元测试覆盖。

做完了吧?

如果你能在 10 分钟内完成，并且每行代码都有单元测试覆盖，那么你还算是个合格的程序员。

> **怎么看单元测试覆盖率**
> 在 IntelliJ IDEA（这是我推荐的 IDE）中，你可以通过菜单“Run –> Run...with Coverage”来运行所有单元测试，并得到测试覆盖率报告。
>
> 
>
>
> 像 FizzBuzz 这么简单的题目，你看到的测试覆盖率应该是 100%。也就是说，每行代码都应该有单元测试覆盖。

在我线下组织的 TDD 练功房中，我看到大部分同学在第一次热身时，都做不到在 10 分钟内完成 FizzBuzz、并保证 100%的单元测试覆盖。有些同学需要用到 20、30 分

钟甚至更长的时间，才能完成这道简单的题目。近距离观察这些同学的情况，我发现很多同学有一个共同的问题：没有准备好开发环境。

有一次我在成都组织一个练功房活动。开始之前我问大家：“大家都知道怎么写单元测试吗？”所有人都告诉我：“知道！”然后我说：“既然知道，那就来把 FizzBuzz 做一遍吧。”然后就发现，很多同学还没有可以编写和运行测试的环境。一听说要求 100%单元测试覆盖，很多同学马上开始下载 JUnit 的 jar 包，准备导入到项目……

停！

现在已经 2021 年了，咱不能再用这么原始的方式来管理工程。几乎每种现代编程语言都有对应的构建管理和依赖管理工具，在 Java 这里，你可以使用 Maven 或 Gradle。例如，下面就是我缺省的 `build.gradle` 配置文件：
(注意：assertThat是Junit4带的方法，最新的gradle构建项目可能使用的是Junit5，推荐修改为Junit4后再继续下面的工作)

```
plugins {
    id 'java'
}

group 'org.codingdojo.kata'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```
程序员练功房

在 IntelliJ IDEA 中新建一个 Gradle 项目，加上最后这句对 JUnit 的依赖，你就可以在 `src/test/java` 目录下开始编写单元测试了。不用操心从哪里下载 jar 包，也不用自己管理 jar 包，构建自动化工具（Gradle）和 IDE（IntelliJ IDEA）会帮你搞定。

> 如果需要依赖任何开源的 jar 包，你都应该到 Maven 中央仓库（ https://mvnrepository.com ）去寻找，并在你的 `build.gradle` 或 `pom.xml` 文件中声明依赖。
> 有些企业的内网不允许访问 Maven 中央仓库。如果你恰好在这样的企业，请询问你的技术主管，企业内的 Maven 仓库在哪里，并在配置文件中做对应的修改。
> 如果你所在的企业既不允许访问 Maven 中央仓库、又没有提供自己的 Maven 仓库，你还可以考虑使用阿里云提供的 Maven 仓库镜像（ https://maven.aliyun.com ）。

从大学的软件工程课开始，我们就不断地说：单元测试很重要，工程自动化很重要。可是每次突然袭击让一群人开始写代码，我都会发现，很多人日常使用的工作环境中，既没有考虑单元测试，也没有基本的工程自动化。这是件很讽刺的事。希望大家都能养成好的习惯，每次要写代码的时候，至少应该把工程自动化和单元测试的环境都准备好。所以，你的第一遍 FizzBuzz 写得怎么样了？

三、软件是需要设计的

不是一次两次，我看到过很多次：没接触过 TDD 的同学，上手做 FizzBuzz 这道题，一写就写成这样，

```
public class FizzBuzz {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            if (i % 15 == 0) {
                System.out.println("FizzBuzz");
                continue;
            }|
            if (i % 3 == 0) {
                System.out.println("Fizz");
                continue;
            }
            if (i % 5 == 0) {
                System.out.println("Buzz");
                continue;
            }
            System.out.println(i);
        }
    }
}
```

程序员练功房

然后，我就会提出那个千篇一律却又直指人心的灵魂拷问：“**你这个程序，打算怎么给人用呢？**”

很多人写程序都有这个毛病：拿到一个需求，不问三七二十一，噼哩咯喳就写出一堆代码。严格说起来呢，这堆代码也确实实现了他听到的需求，但偏偏少做了两件事：

1. **确认自己听到的需求，是不是真实、完整的需求**。客户也好，产品经理也好，不懂技术的人在描述需求的时候，通常不会用严格的逻辑语言来表述，而是会举一些例子，因为这样可以更形象地阐明需求。然而例子必然是具体的、零散的。具体的例子，是否真实、完整地呈现了需求的全貌？这个问题，程序员必须要确认。
2. **考虑自己的代码，将会如何被使用**。究竟谁、会以什么方式使用我们的代码？是以二进制可执行程序的形式，还是类库的形式，或者是源代码的形式？什么情况下有

可能修改软件？修改哪一部分？谁来修改？这些问题，客户和产品经理很可能都给不了你答案，这是技术专业人士必须要自己回答的问题。

这两件事，用洋气一点的词汇来说，就是“设计”：拿到一个需求，你得先做设计，既要设计软件的功能（到底做什么，做到什么程度），也要设计软件的形态（做成什么样，怎么给别人使用）。很多人写程序写得不好，一个症结就在于他们拿起问题就开始写代码，没有做设计。

例如前面的这段代码，就是一个明显缺乏设计的例子。对软件功能和软件形态欠缺设计，会很容易使你陷入困境：

- 如果明天用户想在一个更大的人群里玩这个游戏，于是想把输出的范围改成 1 到 200，怎么办？或者如果想在比较小的人群里玩，想把输出范围改成 1 到 50 呢？谁来做这个修改？必须由写这段代码的程序员来改吗？
- 如果这段逻辑是要放在另一个更大的系统中给别人调用的，怎么办？这段程序怎么给人调用？
- 如果标准输出（`System.out`）被关闭了，怎么办？在很多真实部署的场景下，标准输出 `STDOUT` 会被关闭，只有标准错误输出 `STDERR` 才会被打印到日志。这种情况下，这段程序还怎么用？

我们经常抱怨客户不讲道理、需求变化太多，动不动就要叫程序员改代码，害我们辛苦加班。可是看着这样一段代码，我不禁要问：如果我们拿到一个需求时，对软件功能和软件形态的设计是如此无视，我们做出来的软件怎么可能好用？客户怎么可能不找我们改代码？我们怎么可能不加班？

有些同学又走到了另一个极端，用大量的时间和精力做详尽的设计，写出大堆大堆的文档。且不说这些设计是否真的就能预见到未来可能的变化，光是想到这么繁重的一个设计过程、看到那长篇累牍的设计文档，就叫人不愿再做改动。

所谓敏捷，既不是做过度复杂的设计，也不是完全不做设计埋头就开始编码。要想快速前进、并且灵活响应变化，你需要掌握一系列的能力。掌握了这些能力，你才能快速获得相对合理的设计，并且在需要调整时能够迅速做出调整。能力不会从天而降，必须要通过学习和练习才能获得。所以有些人以讹传讹地说“敏捷就是不写文档”，稍微想想就知道这说法有多不合理：原本不具备这种能力的人，仅仅通过少做一件事（“不写文档”）就能突然获得这种能力了吗？可见这种传言有多荒谬。

TDD 给你的，正是能够快速获得合理设计、并能迅速做出设计调整的能力。在我们的练功房里，你可以慢慢体会，测试是如何引领我们“驱动”出好的软件设计的。

> **TDD 到底是什么的缩写？**
> 按照官方定义，TDD 是“**测试驱动开发**”的缩写。
>
> 但是，因为 TDD 方法会深刻地影响软件的设计，也有不少人私下里说，TDD 其实是“测试驱动设计”。用单元测试一步步驱动出来的，首先是软件的设计，然后才是开

> 发。
>
> 我以前的同事、ThoughtWorks 中国区第一任 CTO，Michael Robinson 曾经说过一句妙语：TDD 是“可测试性驱动设计”（Testability Driven Design）。因为在每一个小步、每一行代码都必须先考虑“怎么测”，可测试性是潜移默化地植根在软件整个建造过程中的，所以用 TDD 方法开发出来的软件具有极佳的可测试性，质量也自然地更可靠。
>
> 总之，TDD 是一种影响深远的开发方法。你可以慢慢体会它的滋味。

四、任务二：初识 TDD

> ✏️**今天的任务**
> 跟着熊老师的步子，按照 TDD 的节奏，一步步把 FizzBuzz 的核心逻辑部分实现一遍。
>
> 要求：步子别迈得太大，慢慢来，注意体会每个动作的细节。

不说那么多废话了，我来带着你一起，按照 TDD 的节奏，把 FizzBuzz 这道题做一遍吧。

FizzBuzz 这道题虽然简单，也不是不动脑筋一锅烩就行的——在前一节里，我们已经看到了“一锅烩”的代码，那不是个值得鼓励的做法。拿到任何需求，我们都应该首先思考：这件事，我可以把它拆分成几个任务？拆出各自独立的几个任务来，我们才好各个击破。

读者可能要奇怪了：FizzBuzz 这么简单的一个需求，还能拆分成几个任务？我这里就拆出了三个任务，你看是否合理：

1. 创建一个对象，这个对象可以对输入的数字做必要的转换，输出一个字符串（可能是“Fizz”、“Buzz”、“FizzBuzz”，或者是原数字的字符串形式）。
2. 创建一个列表，其中包含从 1 到 100 的整数，依次使用前面说的这个对象进行转换，转换后的结果是另一个长度为 100 的列表。
3. 打印输出任务 2 得到的结果列表。

看，这么一拆分，是不是觉得清晰了很多？原来在这么简单的一个需求里，也潜藏着三件事：数字到字符串的转换逻辑、转换特定的列表、打印输出结果。把三件事搅在一起，写出来的程序自然就是难以维护、无法复用，情况发生任何变化自然都得找到原来开发的程序员来改代码了。

在这三个任务中，任务 3（“打印输出结果列表”）简单直白，不太可能出错，也不太可能发生多大变化；任务 2（“指定列表进行转换”）也非常简单，虽然有变化的可能，但这种变化可以在一定程度上交给用户自己去变（例如未来可以提供一个配置文件，指定

游戏的范围）；任务 1（“数字到字符串的转换”）则是这个软件的核心逻辑。所以，我会首先从这里开始，用测试逐步驱动出软件的设计。

任务 1 提到了“一个对象”，这个对象所属的类叫什么好呢？我拍脑袋一想：既然这是一个关于“数字”的“游戏”，那就把这个类叫做“游戏数”（`GameNumber`）吧。从需求中找出合理的对象设计，是软件开发者的一项必备能力。不过这事也没有那么难，只要专心听需求的描述、留意其中提到的“名词”，你就会找到对象的蛛丝马迹。
决定了这里有一个叫做 `GameNumber` 的类，我要做的第一件事不是创建这个类，而是创建一个它的测试类。这个类应该位于 `src/test/java` 目录下，它的名字就叫 `GameNumberTest`：

```
public class GameNumberTest {
}
```

程序员练功房

接下来的事情就很简单了：我想要什么，就在测试里说出来。测试就像一个许愿池，你许的愿，过一会儿就会实现——由你自己实现。这个许愿池有个特点：越小的愿望，就越是容易实现。所以为了让愿望成真，我最好不要一下子就许很宏大的愿望，最好是从小愿望开始。正所谓贪多嚼不烂，十鸟在林不如一鸟在手嘛。

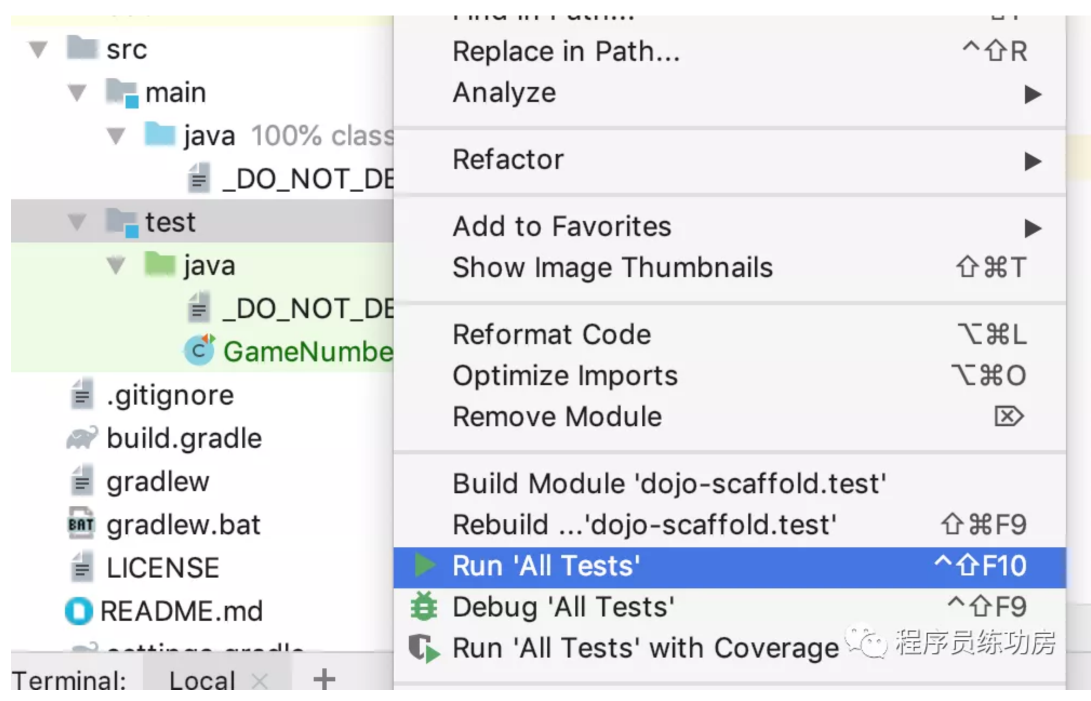
那么对于 `GameNumber` 这个类，我们能许的最小的愿望是什么？很多同学马上就会说：“能把数字 3 替换成‘Fizz’！”

这个愿望挺好，但是还不够小，还可以再小一点。我许的第一个愿望简直可以说是微不足道：**可以用一个数字作为输入，创建一个** `GameNumber` 对象。我用代码把这个愿望写下来。因为它太渺小了，代码写起来也超级简单：

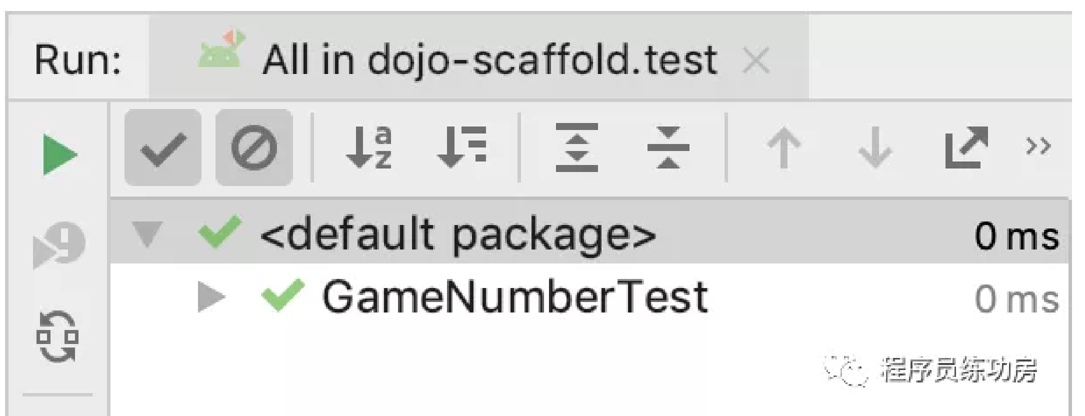
```
import org.junit.Test;

public class GameNumberTest {
    @Test
    public void should_create_game_number_from_raw_number() {

    }
}
```

程序员练功房

这是个空的函数，可以说我只是摆了一个许愿的架势。但是这时我就要运行所有测试了——请注意，我说的是**所有测试**。你可以在命令行中执行 `gradle clean build` 来运行所有测试；也可以在IntelliJ IDEA中右键点击 `src/test` 目录，在弹出菜单中选择“Run ’All Tests’”。



不管用哪种方式，你都应该会看到象征“成功”的浅绿色。我建议你在 IntelliJ IDEA 中运行所有测试，因为只要运行过一次，下次你就可以直接按快捷键 `Shift+F10` 重新运行所有测试。在后面的练功过程中，这个操作你可能会做上几千遍，所以有必要尽早习惯它的快捷键。



为什么要运行一个空的测试？有两个原因。实用的原因是：这次运行会告诉你，你的测试框架是生效的，当你没有犯错的时候，你能看到绿色的运行结果。如果——出于不知道什么神秘的原因——你根本运行不起来单元测试，你最好现在就发现，而不是等到开始写代码之后才发现。

另一个原因比较玄学，但我认为更加重要：这次运行是一种仪式，让你进入测试驱动开发的状态。就像习武之人开始打拳之前要先凝神静气一样，程序员开始编程之前也需要这样一个仪式，给自己一个暗示：从现在开始，我编写的每一行代码都将是高质量的，我编写的软件会始终处于“绿色”的可用状态。

有了这个空的架子之后，我就可以正式开始许愿了。对编程一事，我始终保持着敬畏之心，每次只敢许一个小小的愿望。现在，我小心翼翼地在测试中说：“**我想要一个** `GameNumber` 对象。”

```java
public class GameNumberTest {
    @Test
    public void should_create_game_number_from_raw_number() {
        new GameNumber()
    }
}
```

程序员练功房

如你所见，这一句代码不能通过编译——甚至这都不算“一句”代码，只是一个表达式。但它恰如其分地表达了我的要求：我想要一个`GameNumber`对象。现在`GameNumber`类还不存在，所以IDE提示我红色的编译错误。红色是让我紧张的颜色，我要尽快让它回到绿色的状态。于是我把光标移动到红色的类名上，按下`Option+Enter`键，尝试修复它：

> 🔑**快捷键提示：** `Option+Enter` => 快速修复（Quick Fix）

```java
public class GameNumberTest {
    @Test
    public void should_create_game_number_from_raw_number() {
        new GameNumber()
    }
}
```

```
Create class 'GameNumber'
Create inner class 'GameNumber'
Introduce local variable
```

程序员练功房

在弹出的对话框中，记得要把目标路径选到`src/main/java`下面，按下回车键，我就得到了`GameNumber`类。

```java
public class GameNumber {
}
```

程序员练功房

此时`GameNumberTest`类仍然显示编译错误，因为我写的半句测试代码不仅没有给变量赋值，连行末的分号都还没加。我把光标移动到行末代表编译错误的红色波浪线上，按下`Command+Option+v`键，把`new`出来的对象实例提炼成一个变量。

> 🔑**快捷键提示：** `Command+Option+v` => 提炼变量（Extract Variable）重构

```java
public class (        ;st {
    @Test
    public void sh  ld_create_game_number_from_raw_number() {
        GameNumber gameNumber = new GameNumber();
    }
}
```

```
Declare final

gameNumber
number
Press ⇧→ to change type
```

程序员练功房

回车，我就得到了完整的测试代码。按`Shift+F10`运行所有测试，我应该会看到绿色的提示。

```java
@Test
public void should_create_game_number_from_raw_number() {
    GameNumber gameNumber = new GameNumber();
}
```

程序员练功房

就在刚才短短的时间里，我已经完成了测试驱动开发的一个完整循环：我首先在测试中描述我想要的东西（“一个 `GameNumber` 对象”），由于 `GameNumber` 类尚不存在，我得到了一个编译错误（红色）；随后我——借助IDE的快捷键——创建这个类，使刚编写的测试能够通过（绿色）。测试驱动开发的“红–绿”循环是以这样短辄数十秒、长辄一两分钟的周期来进行的。如果一次“红–绿”的周期超过了5分钟，很可能你的步子迈得太大了——你许下了一个太贪心的愿望。

或许有人会问：这样一个测试有什么用呢？其实它是很有用的。刚才我们一起走过的这个只有数十秒的周期告诉你，你的编译器在正常工作，你的 IDE 在正常工作，你的测试能正常引用 `src/main` 目录下的生产代码……总而言之，TDD所需的环境已经都准备好了，解决问题需要的关键对象也已经到位，你可以正式抬步上路了。

> **关于快捷键**
> 我在这一节里提到了几个快捷键组合。很可能你按下同样的快捷键组合却得不到同样的效果，甚至你的键盘上根本找不到我提到的某些键（例如 `Option` 和 `Command`）。这是因为我使用 Mac 电脑，并且我的 IntelliJ IDEA Keymap 设置为“Mac OS X”风格，与你的电脑及 Keymap 风格未必一样。
>
> 如果你一时还搞不定这些快捷键，不用太担心。你可以在 IntelliJ IDEA 的“Help -> Keymap Reference”菜单项中找到所有快捷键的映射。后面我们还会拿出篇幅专门讨论快捷键。
>
> 为了直观地展现快捷键能带来的效率提升，我在这一节里详细地展示了几个快捷键操作的截图效果。在后面的章节里，除非有必要，我尽量少用这种方式展示快捷键的使用。在用到新的快捷键时，我会用这个方式来提醒读者注意：
> 🔑**快捷键提示**： `<快捷键>` => <用途>

五、实现 FizzBuzz 主体逻辑

有了 `GameNumber` 对象以后，我的信心更足了。现在我可以向测试许更多的愿望。不过，我还是保持着敬畏之心，不敢一次索求太多。下一个我要许的愿，是“当原始数值为 1 时，输出的字符串是“1””。我可以在原先的测试基础上增加这个新的愿望，但要记得同时修改测试的名字，使其反映现在我想要实现的愿望。于是，原来的 `should_create_game_number_from_raw_number` 测试方法就变成了：

```
@Test
public void should_say_1_when_raw_number_is_1() {
    GameNumber gameNumber = new GameNumber(1);
    assertThat(gameNumber.toString(), is("1"));
}
```

程序员练功房

我向 GameNumber 的构造函数传入了 1 这个参数，这会引起一个编译错误。按 Option+Enter 快速修复，IDE 会帮你创建出这个构造函数。增加的这一行代码，用到了 JUnit 4 新增的 Assert.assertThat 断言方法。相比更早的 assertEquals 等断言方法，assertThat 写出的代码更易读，就像在直白地说一句话："断言 gameNumber 的 toString()方法返回值是字符串 " 1 "" — —这句断言用英文读起来就是在讲这句话。许愿用的语言越是直白，想必愿望也会越容易实现吧。

要让这句断言通过编译，你需要导入 org.junit.Assert.assertThat 这个静态方法。Java 语法要求把 import 语句都写在文件最开头的地方，这会强迫我们离开正在编写的代码。还好 IntelliJ IDEA 给我们提供了便利的快捷键支持。只要把光标移到尚未导入的方法名上，按 Option+Enter,你就会看到"import static method.."的提示，

```
public void should_say_1_when_raw_number_is_1() {
    GameNumber gameNumber = new GameNumber( rawNumber: 1);
    assertThat(gameNumber.toString(), is("1"));
}
    Import static method...
    Qualify static call...
    Rename reference
    Debug 'GameNumberTest.shoul...'
    Run 'GameNumberTest.shoul...'
```

程序员练功房

再按下回车，会看到可选的同名函数，从中选择一个合适的来导入即可。

```
public void should_say_1_when_raw_number_is_1() {
    GameNumber gameNumber = new GameNumber( rawNumber: 1);
    assertThat(gameNumber.toString(), is("1"));
}
    Method to Import
    MatcherAssert.assertThat (org.hamcrest)    Gradle: org.hamcrest:hamcrest-core:1.3 (hamcrest-core-1.3.jar)
    Assert.assertThat (org.junit)    Gradle: junit:junit:4.12 (junit-4.12.jar)
```

程序员练功房

习惯这套操作，就不用离开代码主体、跑到文件顶上去写 import 语句了。

按 Shift+F10 运行所有测试，这时你会看到红色（在 IntelliJ IDEA 中其实更像是橙色）的失败提示。这说明我们已经成功地许下了一个愿望，现在该去实现这个愿望了。

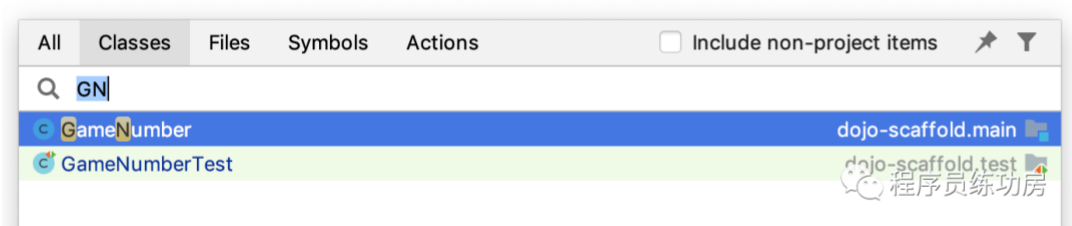
> 一般来说，当你想要许下新的愿望、要求新的功能时，你都应该新建一个测试方法。在这个例子中，因为我们的第一个测试实在太简单了，简单到连断言都没有，所以我直接把这个测试方法改名，令其做最基本的断言验证。这是我常用的套路：第一个测试，验证可以创建对象，并验证对象最简单的行为。

有了测试代码中的"愿望"在驱动，我们修改产品代码的目标就很明确了：每次修改产品代码，

只能做恰好让愿望通过的修改。如果所有的愿望都处于已经通过的状态（运行所有测试看到绿色的状态），就不允许修改产品代码。

> 编程原则
> 1.没有失败的测试就不允许修改产品代码。
> 2.只允许做恰好让测试通过的修改。

于是我们去到 GameNumber 类，准备开始做修改。从 `GameNumberTest` 去到 `GameNumber` 有很多方法，其中最慢也最糟糕的一种，就是拿起鼠标，在左侧的项目导航栏里点选 `GameNumber.java` 文件。鼠标操作和导航操作都是很慢、很耗费精力的操作，应该尽量避免这些操作。你可以按下 `Command+n （这里和我快捷键不同，我是Command+O）` 键，在弹出的搜索框中输入 `GameNumber` 的首字母缩写 " GN"，IntelliJ IDEA 就会找到 `GameNumber` 类，按回车即可打开它。



不过还有一个更简便的办法。因为 `GameNumberTest` 是针对 `GameNumber` 类的测试类，你可以用 `Command+Shift+t` 快捷键在产品类和测试类之间切换。在 TDD 的过程中，这个快捷键应该是你最常用的在文件之间切换的方式。

> 快捷键提示
> 1. `Command+n` => 导航到类（Navigate to Class）
> 2. `Command+Shift+t` => 导航到测试对象/被测目标（Navigate to Test/Test Subject）

在 `GameNumber` 这里，我们做最简单的修改，恰好让测试能通过就好。这一步非常简单，想必已经不需要任何解释了。

```
public class GameNumber {
    private int rawNumber;

    public GameNumber(int rawNumber) {
        this.rawNumber = rawNumber;
    }

    @Override
    public String toString() {
        return String.valueOf(rawNumber);
    }
}
```

程序员练功房

再次运行所有测试。这时你应该会看到绿色的成功提示。我们又达成了一个小愿望。接下来，我想要“当原始数值为 3 时输出字符串“Fizz””。我新增一个测试方法来许愿：

```
@Test
public void should_say_fizz_when_raw_number_is_3() {
    GameNumber gameNumber = new GameNumber(3);
    assertThat(gameNumber.toString(), is("Fizz"));
}
```

程序员练功房

运行所有测试，红色。现在去到 GameNumber 类，实现这个小愿望：

```
@Override
public String toString() {
    if (rawNumber % 3 == 0) {
        return "Fizz";
    }
    return String.valueOf(rawNumber);
}
```

程序员练功房

运行所有测试，绿色。再回到 GameNumberTest，再许一个小愿望：“当原始数值为 5 时输出字符串“Buzz””，

```
@Test
public void should_say_buzz_when_raw_number_is_5() {
    assertThat(new GameNumber(5).toString(), is("Buzz"));
}
```

程序员练功房

运行所有测试，红色。再去到 GameNumber，实现这个小愿望：

```
@Override
public String toString() {
    if (rawNumber % 3 == 0) {
        return "Fizz";
    }
    if (rawNumber % 5 == 0) {
        return "Buzz";
    }
    return String.valueOf(rawNumber);
}
```

程序员练功房

运行所有测试，绿色。再回到 GameNumberTest，再许一个小愿望：“当原始数值为 15 时输出字符串“FizzBuzz””，

```java
@Test
public void should_say_fizzbuzz_when_raw_number_is_15() {
    assertThat(new GameNumber(15).toString(), is("FizzBuzz"));
}
```
程序员练功房

运行所有测试，红色。再去到 GameNumber，实现这个小愿望：

```java
@Override
public String toString() {
    if (rawNumber % 3 == 0 && rawNumber % 5 == 0) {
        return "FizzBuzz";
    }
    if (rawNumber % 3 == 0) {
        return "Fizz";
    }
    if (rawNumber % 5 == 0) {
        return "Buzz";
    }
    return String.valueOf(rawNumber);
}
```
程序员练功房

运行所有测试，绿色。这时我可以长舒一口气：FizzBuzz 的主体逻辑，我已经实现出来了，并且代码 100%有测试覆盖。你可以跑一下测试覆盖率统计看看。

你有没有下意识地跟着我一起，不断重复"运行所有测试，红色；运行所有测试，绿色"？这就是 TDD 的节奏感。采用 TDD 开发软件时，每隔几十秒、顶多一两分钟，你就应该运行所有测试，看到红色或绿色的反馈，然后继续下一步操作。这也是为什么 TDD 必须依赖于单元测试：因为只有单元测试才有足够快的执行速度。慢慢习惯这个节奏，不断练习，让它成为你的肌肉记忆。

> 细心的读者或许会发现，我在描述两个文件之间的切换动作时，用了不同的动词："去到"GameNumber 和"回到"GameNumberTest 遵循 TDD 的规则，每一次对产品代码的修改，都应该是从"添加（或修改）测试开始。所以在我的潜意识里，测试类才是我常驻的工作地点而添加/修改测试之后，运行会看到红色，这会令我紧张，于是我不得不"去到"产品代码中，做尽可能快、尽可能小的修改，刚好令测试能通过就好。一旦测试恢复到绿色的状态，我就迫不及待地"回到测试类，在这个安全的状态下慢慢思考，下一步我要许什么愿。

六、重构初体验

紧凑的"红-绿"节奏很容易让人疲劳，现在 FizzBuzz 的主体逻辑已经完成，可以稍作休息。在起身去打一杯咖啡之前，我们再回头看看刚写下的代码，看有没有什么不妥之处。

```java
    @Override
    public String toString() {
        if (rawNumber % 3 == 0 && rawNumber % 5 == 0) {
            return "FizzBuzz";
        }
        if (rawNumber % 3 == 0) {
            return "Fizz";
        }
        if (rawNumber % 5 == 0) {
            return "Buzz";
        }
        return String.valueOf(rawNumber);
    }
```

程序员练功房

细看之下，这段代码中是有重复的：`rawNumber % 3 == 0` 这个条件判断被重复了两遍，同样，`rawNumber % 5 == 0` 也被重复了两遍。而且如果我们再细想的话，这两个条件表达式本身也是重复：它们都表达了“`rawNumber` 能被某个数整除”的意思，只不过“某个数”分别是 3 和 5 的区别。我们应该想办法消除这一点点重复。

> **代码的坏味道**
> Martin Fowler 在《重构》书中形象地把代码中不够良好的结构称为“坏味道”。这些坏味道会给理解和维护代码增加困难。一处处不起眼的坏味道累积起来，就会让整个代码库臭不可闻。所以我们应该防微杜渐，及时发现每一点细微的坏味道，及时重构消除它们，将代码的腐化掐灭在萌芽状态。
>
> 在《重构》的第三章里列出了 22 种（第二版是 24 种）坏味道，每一种都有对应的重构手法。有空的时候，可以去翻一翻这章内容，想想自己的代码中是否还有别的坏味道。

消除重复代码的基本操作是提炼函数（Extract Method）。你可以选中想要提炼的代码片段，按 `Command+Option+m` 快捷键，跟着提示即可完成这个重构操作。不过我正好借这个机会，介绍一点点重构的基本原则。所以我们暂时不用快捷键，手工来做一遍这个重构操作。

关于怎么重构，我在 ThoughtWorks 时的同事王健有个精辟的十六字口诀：

> 旧的不变
> 新的创建
> 一步切换
> 旧的再见

我们运用这个口诀来重构上面的重复代码。“旧的不变”，我们先留着原来的重复代码不动。“新的创建”，把想要提炼的这段逻辑复制出来，为它新建一个函数，并做必要的调整，使之通过编译。

```java
private boolean isDivisibleBy(int denominator) {
    return rawNumber % denominator == 0;
}
```

程序员练功房

运行测试。此时测试应该仍然是通过的绿色状态——我们新增的函数还没有被任何代码使用，因此它不会影响程序的行为。

接下来是口诀的第三句：“一步切换”，原来重复的代码共有 4 处，我们把其中的第一处替换为调用这个新建的函数：

```java
@Override
public String toString() {
    if (isDivisibleBy(3) && rawNumber % 5 == 0) {
        return "FizzBuzz";
    }
    if (rawNumber % 3 == 0) {
        return "Fizz";
    }
    if (rawNumber % 5 == 0) {
        return "Buzz";
    }
    return String.valueOf(rawNumber);
}
```

程序员练功房

运行测试，仍然是绿色。这一步切换足够简单，应该不会出错，但为防万一，我们还是要运行测试。

然后我们重复这个过程，将原来 4 处重复代码全部替换为调用新建的函数，每次替换之后都运行测试并看见绿色。最后我们就得到了下面的代码，原来的重复代码已经“旧的再见”了：

```java
@Override
public String toString() {
    if (isDivisibleBy(3) && isDivisibleBy(5)) {
        return "FizzBuzz";
    }
    if (isDivisibleBy(3)) {
        return "Fizz";
    }
    if (isDivisibleBy(5)) {
        return "Buzz";
    }
    return String.valueOf(rawNumber);
}
```

程序员练功房

这就是重构应该发生的时机和方式。和很多人的理解不同，重构不是一件惊天动地的大事。如果你写上一整天代码然后说“我现在要花一个小时重构”，你已经开始跑偏了一当然我们知道，还有很多团队先不管不顾地写上几个月代码，然后专门安排一个星期时间重构，这就完全是瞎搞。

重构应该在编程的整个过程中随时发生：你写上 10 分钟代码，你回头看看，刚才写的代码里有没有一点点坏味道，如果有，你就马上重构它。并且，在进行重构时，你必须严格遵循重构手法，小步前进，随时保持软件的行为不变——按照我的经验，每对代码做**一分钟**修改，你就应该运行一次测试，并看到绿色的成功提示。如果测试失败，你顶多需要撤销前面一分钟的工作，这不是什么大不了的损失。

我们在前面说过两条 TDD 的编程原则，第一条就是“没有失败的测试就不允许修改产品代码”。可是现在，我们在测试一直为绿色的情况下，对产品代码做了修改。我们违背自己的编程原则了吗?

并没有。重构的定义是“在不改变软件可观察行为的前提下，调整其结构”。也就是说，重构只是对代码结构做调整优化，完全不改变软件的行为，因此测试也应该始终保持绿色不变。所以，我们对 TDD 的编程原则修订如下：

> 编程原则
> 1.没有失败的测试就不允许修改软件的行为。
> 2.只允许做恰好让测试通过的修改。
> 3.在保证不改变软件行为的前提下，可以对代码进行重构，消除其中的坏味道。

好了，代码的坏味道已经消除了，你也对重构有了一个正确的初步理解。现在我就放你一马，去倒杯咖啡休息一下吧。我们下个任务见。

七、完成 FizzBuzz

> 今天的任务
> 完成 Fizz Buzz 整个游戏的需求，并实现新增的需求。
> 要求：严格遵循 TDD 的“红－绿－重构节奏，时刻留意代码是否有尚可改善的地方。

FizzBuzz 的主体逻辑（“能被 3 整除的数替换成`”Fizz ”`，能被 5 整除的数替换成`”Buzz ”`，既能被 3 整除又能被 5 整除的数替换成`”FizzBuzz ”`”）实现之后，我们就可以很轻松地实现“把游戏从 1 玩到 100 的逻辑－–把从 1 到 100 的数挨个拿去创建 `GameNumber` 对象，再挨个调用 `toString（）`方法获得转换之后的字符串就可以了。

是不是已经跃跃欲试要开始写代码了？别急。**别忘了测试驱动开发的节奏：先写测试**。请读者先自己尝试写一遍，再来看我的讲解。

既然我们现在要处理“整个游戏”的逻辑，这个测试顺理成章地就叫做 `GameTest` 好了。和前面一样，我们首先愿“能指定游戏轮数（`size`）创建一个 `Game` 对象”。至于这个对象能干什么，先不着急，慢慢想。

```
public class GameTest {
    @Test
    public void should_create_game_object_with_given_size() {
        Game game = new Game(100);
        assertThat(game.size(), is(100));
    }
}
```

程序员练功房

运行测试，红色。让测试能通过的最简单的实现，完全不用动脑子，直接把构造函数传入的size 给返回回去就好了。

```
public class Game {
    private final int size;

    public Game(int size) {
        this.size = size;
    }

    public int size() {
        return size;
    }
}
```

程序员练功房

运行测试，绿色。

这个看起来近乎弱智的操作步骤，其实是一个常用的手法：对于要处理的问题，我首先创建一个对象，并验证这个对象“包含某些东西”——在这个例子里，我的测试是在验证“size为100的Game对象包含100个某个东西”。有了这第一个成功的测试，接下来我就可以停下来认真想想，这100个“某个东西到底是什么。显然在我们这里，这100个“某个东西其实应该是100个GameNumber对象。所以我把实现代码又修改成：

```
public class Game {
    private final List<GameNumber> gameNumbers;

    public Game(int size) {
        gameNumbers = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            gameNumbers.add(new GameNumber(i));
        }
    }

    public int size() {
        return gameNumbers.size();
    }
}
```
程序员练功房

运行测试，仍然是绿色。

“错了错了！”我听见细心的读者大喊道，我们的游戏应该是从 1 开始，而不是从 0 开始！”

完全正确。但不要着急，先写测试，用测试来许愿描述我们想要的东西。我们希望 Game 对象能给我们一个经过转换之后的字符串列表，其中包含了每个小朋友要报的数（或者要说的词）：

```
    @Test
    public void should_provide_words_to_be_spoken() {
        Game game = new Game(100);
        List<String> wordsToBeSpoken = game.words();
        assertThat(wordsToBeSpoken.size(), is(100));
        assertThat(wordsToBeSpoken.get(0), is("1"));
    }
```
程序员练功房

运行测试，红色。实现这个测试也很容易：

```
    public List<String> words() {
        List<String> result = new ArrayList<>();
        for (GameNumber gameNumber : gameNumbers) {
            result.add(gameNumber.toString());
        }
        return result;
    }
```
程序员练功房

再次运行测试……哎呀，还是红色！原来是在自信地写这个循环逻辑的时候，我忘了修改前面读者发现的那个错误：游戏应该从 1 开始，而不是从 0 开始。于是我把 Game 的构造函数修

改成：

```
public Game(int size) {
    gameNumbers = new ArrayList<>();
    for (int i = 1; i <= size; i++) {
        gameNumbers.add(new GameNumber(i));
    }
}
```

程序员练功房

运行测试，终于看到了绿色。咻~

就我们刚写完的这一点代码，也不是没有优化空间的。我个人很不喜欢用 for 循环去遍历一个列表的写法。首先因为使用循环的代码写起来比较麻烦：我们这里实现 words 方法用了 5 行代码，在开始循环之前要先创建一个列表来容纳最终的结果，循环之后要返回这个结果列表，这一前一后的两行代码，几乎每次循环都得这么写，颇有点讨厌。

而不喜欢写 for 循环更重要的原因是，这个语法不能很清晰地告诉我们，**这个循环到底是在干什么**。其实我们真正在做的事是“拿着包含 N 个 GameNumber 对象的列表在手，对其中的每个元素调用 toString（）方法，将其结果放入包含 N 个 String 对象的列表”。或者用更精炼的语言来描述，我们做的事是：

对 List<GameNumber>列表，调用每个元素的 toString（）方法，进行“映射”操作。

“映射”是列表的几种基本操作之一。在后面的章节我们还会更深入地讨论这几种基本操作。目前，我们只需要知道，Java 的 **stream API** 可以很直观地实现列表的映射操作。用 stream API 改写后，words 方法变成了这样：

```
public List<String> words() {
 return gameNumbers.stream()
    .map(GameNumber::toString).collect(toList());
}
```

程序员练功房

这个写法不仅更短，而且更清晰地描述了我们究竟在对这个列表做什么操作（上面这行代码几乎就是逐字在说“**用 GameNumber 的 toString 方法进行映射操作**”）。我比较喜欢这样的写法，所以我会建议读者也去花些时间研究 Java 的 stream API，看看它可以如何帮助你改善列表操 FizzBuzz 游戏的实现到这里其实已经完成了：拿到映射后的列表，我们可以直接调用 System.out. println 将其打印出来，效果就是这样：

> [1，2，Fizz，4，Buzz，Fizz，7，8 Fizz，Buzz，11，94，Buzz，Fizz，97，98，Fizz，Buzz]

当然我们也可以增加一些断言，多做一些验证，

> assertThat（wordsToBeSpoken. get（），is（”1”））

> assertThat（wordsToBeSpoken. get（2），is（”Fizz”））；
> assertThat（ wordsToBeSpoken.get（4），is（”buzz”））；
> assertThat（wordsToBeSpoken.get14），is（”FizzBuzz”））；
> ......

但其实这些断言并不会进一步增加我们对程序的信心，写还是不写，区别都不太大。如果你没有想到需要增加的新功能或者需要修复的缺陷，那么就没必要继续增加新的测试。

> **测试有多完备?**
> 常有人纠结于一个问题：**TDD 写出来的测试有多完备**?
>
> “测试能覆盖所有可能的情况吗？能保证软件没有缺陷吗？如果测试写错了该怎么办？”这都是我经常遇到的关于 TDD 的疑问。
>
> 要我说，这些疑问，都是想多了。
>
> TDD 是一种软件设计和开发的方法，而不是一种软件测试方法。通过编写单元测试，程序员是在表达“我想要实现什么”，然后实现自己想要实现的逻辑。至于单元测试套件，那是 TDD 顺带的收益：它只能保证软件实际的行为跟程序员想要的行为一致，而不能保证程序员想得完备，甚至不能保证程序员想得正确。
>
> 设计的正确性和完备性，只能通过设计者的思考来得到。没有任何设计工具能帮不会做良好设计的人做出良好设计，正如没有任何一支笔能让不会画画的人画出漂亮的图画。

八、FizzBuzz 拓展需求

玩了几轮 FizzBuzz 游戏以后，数学课的老师可能会嫌游戏太简单了，想给它增加一点难度。比如说：

> ·如果一个数能被 3 整除，或者包含数字，那么这个数就是“Fizz
> ·如果一个数能被 5 整除，或者包含数字 5，那么这个数就是“Buzz”

在以前的练功房中，每次说到这个新需求，马上就会有同学提出一系列的问题：

“既包含 3 又包含 5 的数应该怎么办？”“包含 3 又能被 3 整除的数应该怎么办？”“既包含 3 又能被 5 整除的数应该怎么办？”“包含和整除哪个优先级高？”......

其实这是挺常见的情况：客户提出了新的需求，但是并没有提得非常清晰、非常完备。这时候作为程序员的我们该怎么办呢？有些同学就会说：那我就去问客户，问他应该怎么处理这些逻辑。但这真的是最好的处理办法吗？

如果我们直接拿上面这一系列问题去问客户，很可能客户的感受不会太好。首先，客户很可能并没有把所有这些具体的逻辑想得非常清楚，所以当你把这一堆问题丢给他，他很可能一脸懵，不知道该怎么回答。更重要的是，客户会觉得，这些具体逻辑的、“实现细节”的问题，本来就应该是程序员来考虑的问题。如果程序员不去思考这些问题，直接把具体逻辑的问题丢给客户，会让客户觉得程序员没有主动性、不关心业务，以后就更不愿意跟程序员讨论业务问题了。

对于不清晰、不完备的需求，我有一个应对的口诀：

> 大胆猜想
> 合理假设
> 谨慎求证

软件的需求，最终肯定要由客户来决定但在客户做决定之前，程序员应该先结合对业务的理解，**大胆猜想**这个需求究竟是要达成什么目标，**合理假设**实现这个需求有哪些上下文和约束条件、有哪些客户没有明言的情况，最后把自己的猜想与假设整理清楚，向客户**谨慎求证**。

例如在这个例子里，我们可以按这个口诀来操作：

- **大胆猜想**。我们猜想，客户之所以提出这个新的需求，目的是为了给游戏增加一点难度与乐趣。由于这是个考快速反应的游戏，特殊的报数应该还是“Fizz、“Buzz ”和“FizzBuzz”这几种，只是这几种特殊报数出现的情况变得更多了。但不会出现“Fizz Fizz ” 或者“BuzzFizz 这样的情况，因为在快速的报数中做精确的逻辑计算和排序太困难了。
- **合理假设**。基于上面的猜想，我们做出一个假设：客户之所以没有很详细地说明各种具体情况，是因为在他的理解中，数字的特殊情况是两种，即“Fizz ” 和“Buzz ”，而“FizzBuzz”则是两者的组合－－既是“Fizz”又是“Buzz。至于新增的需求，则是增加了“Fizz ” 和“Buzz”两种数字的范围：不仅是“能被 3/5 整除”，而且是“包含 3/5 的数字”。
- **谨慎求证**。当然，最后我们必须把我们的猜想和假设讲给客户，听取他的反馈。

幸运的是，这次客户开心地说：“没错，我就是这样考虑的。”我们的猜想和假设正确，没有花费客户太大精力就弄明白了他真正想要的东西，而不用耗费大量时间思考那些具体的逻辑问题。但即使我们的猜想和假设有错，当我们采用“猜想假设－求证”的套路来澄清需求时，客户能帮我们发现那些猜想和假设有误，这会帮助我们更好地理解业务，下一次提出更好的猜想和假设。

弄明白了需求之后，还是老规矩，我们先写个测试来许愿：

```
@Test
public void should_say_fizz_when_raw_number_is_13() {
    assertThat(new GameNumber(13).toString(), is("Fizz"));
}
```

程序员练功房

运行测试，红色。现在再回头去看实现代码，我们欣喜地发现，之前做的重构使得代码逻辑清晰易读，因此也很容易看出新需求到底改变在哪里：

```
@Override
public String toString() {
    if (isDivisibleBy(3) && isDivisibleBy(5)) {
        return "FizzBuzz";
    }
    if (isDivisibleBy(3)) {
        return "Fizz";
    }
    if (isDivisibleBy(5)) {
        return "Buzz";
    }
    return valueOf(rawNumber);
}
```

程序员练功房

在原来的代码中，我们关心的是“能被某个数整除”（ isDivisibleBy）的逻辑，而现在我们要对这个逻辑加以扩展，我们现在要关心的是“能被某个数整除或包含某个数的逻辑。于是我们先将 isDivisibleBy 函数改名，然后加上新的逻辑：

```
private boolean isDivisibleByOrContains(int number) {
    return rawNumber % number == 0 ||
        valueOf(rawNumber).contains(valueOf(number));
}
```

程序员练功房

多亏之前做了重构，现在我们只需要做一处代码改动，就可以完美实现整个新需求。干净简洁的代码总是更容易维护，这也正是为什么我们应该随时重构代码。如果还不够确定的话，我们可以增加几个断言来检查新需求是否被完美实现。

```
assertThat(new GameNumber(13).toString(), is("Fizz"));
assertThat(new GameNumber(51).toString(), is("FizzBuzz"));
assertThat(new GameNumber(52).toString(), is("Buzz"));
assertThat(new GameNumber(53).toString(), is("FizzBuzz"));
```

程序员练功房

运行测试，应该仍然是绿色。51 是一个比较特殊的情况：它能被 3 整除，并且包含数字 5，因此它应该是“FizzBuzz ” 很多人一开始都没有想清楚，但程序已经能运行出正确的答案。这就是充分理解业务逻辑的价值：如果你充分理解并正确实现了业务逻辑，代码会很清晰简洁，而且会自然地覆盖到各种复杂的边边角角。

在结束今天的任务之前，我们最后再看看这段代码。isDivisibleByOrContains 这个方法名我觉得不好，不仅因为这个名字很长，而且因为它是在描述方法内部**做了什么**，而不是这个方法的用途。方法名是给方法的使用者看的，所以它应该讲出方法的**用途**，而不是把方法的**实现**再讲一遍——方法的实现应该由方法内部的代码自己来讲。所以我们对这个方法改名，让它讲出方法的用途。

```
    private boolean isRelatedTo(int number) {
        return rawNumber % number == 0
                || valueOf(rawNumber).contains(valueOf(number));
    }
```
程序员练功房

从方法名就可以看到，这个方法的用途是与某个数相关”（isRelatedTo）。如果用户好奇什么叫“相关”，他可以再看实现代码：原来与某个数相关”的意思就是“能被这个数整除、或者包含这个数字”。好的代码就应该这样，让阅读者一望便知它的用途、也能清晰读懂它的实现。

到这里，FizzBuzz 这道题我们总算是完成了。休息一下，明天你将迎来更残酷的训练。我们下个任务见。

九、任务四：速度！速度！

> 今天的任务
> 反复多练几遍 FizzBuzz，找出开发速度的短板，有意识地提升速度，争取练到 5 分钟以内完成。
> 要求：严格遵循 TDD 的“红－绿－重构节奏，主动练习提高速度的环节。

FizzBuzz 是一道超级简单的题目。经过前面的讲解，你应该已经能很有把握地按照 TDD 的步伐将它实现出来，代码质量也很好。对于 DD 的“红－绿－重构”的节奏，你应该已经有了一定的信心。

现在，把前面的代码都删掉，从头再做一遍 FizzBuzz。这次在开始之前，先给自己计个时，看看你需要多长时间能完成这个题目。

FizzBuzz 竞速 ” 是一个流传甚广的小竞赛，测试驱动开发的爱好者们经常会用这个题目来比拼一下。在玩竞速的时候，一般只实现核心逻辑（也就是我们前面介绍的 GameNumber 类的逻辑），包含起始的需求（能被 3/5 整除）和扩展的需求（包含数字 3/5)。据我的观察，比较熟练的程序员大约能在 15 分钟内完成这道题目。然而如果是对开发工具、编程语言、TDD 的节奏、甚至是自己的键盘不熟悉，拖到半小时乃至一小时才能完成的程序员，也并不罕见。

那么，优秀的程序员完成这道题的速度有多快呢?

- 在 Agile22008 大会上，Michael Feathers（他是《修改代码的艺术》一书的作者）和 Emily Bache 结对，用了 4 分钟完成；
- 我完成的时间也差不多是 4 分钟。感兴趣的读者可以看我的全程录像视频；
- 我以前的同事佘镇实现起始需求（能被 3/5 整除）用了 1 分 45 秒。考虑到扩展需求相当简单，估计他完成整个题目的时间大约在 2 分钟左右。

照我看来，至少应该在 5 分钟内完成 FizzBuzz 题，才算是合格程序员应该有的速度。

听我说这个话，很多人马上就会提出反对意见：

“这个题目太简单了，根本不说明什么问题。”

“编程又不是搬砖，不应该太强调速度。”

“重要的是思考，不是敲键盘的速度。”

然而这些反对意见完全没有抓到重点。的确，能在 5 分钟之内完成 FizzBuzz，这不能说明你有多优秀、掌握面向对象设计的技能、熟悉微服务架.反过来，如果这么简单的一道题目花了你超过 10 分钟来解决，很可能说明你在日常工作中有一些效率短板。当你在处理更复杂的问题时，这些效率短板会更严重地妨碍你的开发速度。

更重要的是，软件开发并不是像很多人想象的那样，可以将“思考”与“编码”割裂。实际上，大多数人都没有那么强的抽象思维能力，能不写代码就把一个问题的边边角角都思考清楚。大多数人需要思考一点、写一点代码、在此基础上再深入思考一点，如此循环往复、小步前进。所以，如果编码的速度太慢，就会导致问题迟迟不能展开，会拖长整个思考的过程，使程序员更容易疲劳、注意力涣散。

说得直白一点，同样一个小任务，别人用 15 分钟完成，你用 5 分钟就能完成，那你就多出 10 分钟时间来思考。这样一个个 10 分钟日积月累，你不就比别人多了很多学习和反思的机会。

所以，编码的速度绝非不重要。在简单的题目上及早发现自己的速度短板，及早做必要的调整和训练，在未来的编程生涯中会受益匪浅。

如果你想知道自己的速度短板在哪里，可以参照我的录像视频，对比你自己的编程动作。据我观察，常见的速度短板有这么几个：

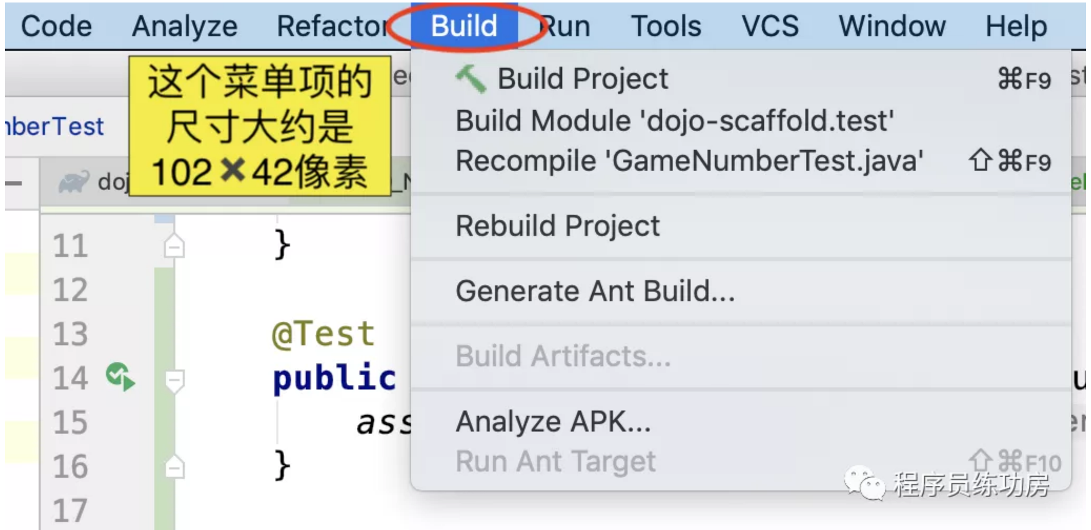
1. 滥用鼠标。
2. 对开发工具不熟。
3. 单纯的产出代码速度慢。

接下来，我们就来挨个看看这几个速度短板应该怎么克服。

十、停止使用鼠标

鼠标是一项伟大的发明，它让业余的计算机用户也能快速上手操作：符合直觉的点点按按拖拖拽拽，就能完成很多任务。但鼠标的问题也恰好在这里：它是为业余的计算机用户设计的，因此**对于很多专业任务而言，鼠标是一种效率低下的工具**。而编程，恰好是这样的一门专业任务。

鼠标的根本问题在于，它是一种需要耗费注意力的工具。本质上，“点按一个菜单项这个操作，需要用户在 2560x1600 像素（我的屏幕分辨率）的屏幕上找到并点中大约 102x42 像素大小的区域，后者的面积大约是整个屏幕面积的千分之一（准确说，大约 1/956)。因此，这个操作永远都需要额外的注意力，用户必须看着鼠标准确移动到菜单项上，然后按下按键。熟练的用户会操作得更快，但这个操作永远不可能“盲操作”。



实际上，为了编译构建整个工程，你需要做两次鼠标点击：先点击“Build”，然后在下拉菜单中点击“Build Project”。而同样的操作如果用键盘的话，只需要一个快捷键组合（Command+F9，在下拉菜单里有提示）。不仅需要的操作更少，而且稍加练习就可以盲操作——你不需要看键盘就可以准确地按到这两个键。

另一个典型的例子是“找到需要编辑的源文件”。在前面的“实现 FizzBuzz 主体逻辑”（任务二）中，我们已经讨论过：如果你想去到 GameNumber 类用鼠标在左侧的项目导航栏里点选 GameNumber.java 文件是最慢也最耗费注意力的一种。工程越大、源文件越多，通过导航找到需要编辑的源文件就会越耗时。更合理的操作有：

- 按 Command+n(同上，Command+O) 快捷键，在弹出的搜索框中输入 GamNumber 的首字母缩写“GN”，找到并打开 GameNumber 类；
- 按 Command+shift+t 快捷键，从测试类 GameNuberTest 直接跳转到 GameNumber 实现类；
- 按 Command+e (Command+Shift+退格键)快捷键，切回最近编辑的文件。

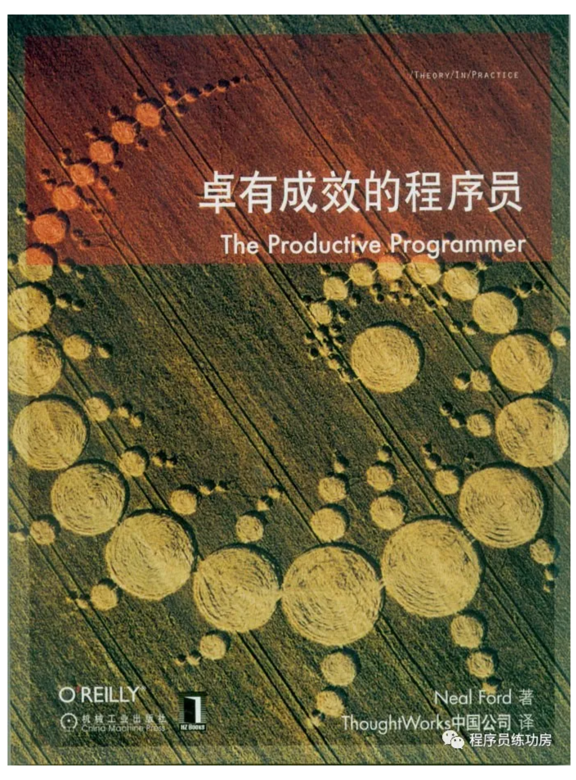
这三种操作更高效，不仅是因为它们都是纯键盘操作（并且你的手不会离开主键位区），而且因为执行这几个操作时，你都是直接表达自己想要的东西（“首字母缩写为 GN 的类”、“当前测试类对应的实现类、“最近编辑的文件”），而不是在导航区里一边浏览一边寻找自己想要的东西。也就是说，这些操作都是搜索操作，因为你知道自己在寻找什么、并明确地寻找目标，而不是一边浏览一边寻找。

我以前的同事 Neal Ford 在《卓有成效的程序员》一书中用了整整一章篇幅讲解程序员应该如何加速自己的工作，其中几段内容值得与大家分享：

> “永远不要将你的双手从字符按键上移开即使是下移到键盘上的箭头按键都会使你慢下来，因为你必须再次回到主排键来输入字符.真正有用的编辑器会使你的手保持在最佳位置，同时进行输入和导航。”
>
> “一种记忆快捷键的好方法是让某人（或某物）提醒你这一点 IntelliJ 就有这样一个很妙的插件叫做‘按键提示器（Key Prompter）‘，每次你使用菜单进行选择时，一个对话框就会弹出来告诉你可以用的快捷键，以及你已经做错了多少次。”
>
> “现代的 Java IDE 允许你在当前工程中快速找到任意 Java 源文件输入完整名称（甚至名称的一部分）很麻烦你只需输入大写字母，不必输入文件名，它［IDE］就会查找有着相同

> 大写字母模式的名字。比如说，如果你要查找文件 ShoppingCartMemento，可以输入 SCM，IDE 会忽略中间的小写字母找到与大写字母匹配的模式，”
>
> “项目变得越来越庞大，包和命名空间越来越多。层次结构变大以后，在其中导航就会变得劲，因为需要进入的层次太深了文件系统就像一个硕大的干草堆，而我们总是需要从中找出一根又一根的针四下寻找文件，这个费时费力的操作会使你分心，无法集中精力考虑真正重要的问题。还好，在新的搜索工具帮助下，你几乎可以完全抛弃麻烦的文件系统导航。”
> “不要文件树。要搜索。”

感兴趣的读者可以去读 Neal 的书了解更多相关信息。



> 编程原则
> 1. 键盘操作优于鼠标操作
> 2. 搜索优于导航

十一、用好快捷键

说到“键盘操作优于鼠标操作”，就会自然地引出第二个问题：很多程序员不是不想尽量用键盘操作，只是他们不知道常用的操作应该用什么快捷键来实现。举个毫不夸张的例子：我认识的一位设计师，她用 Mac 至少有十年了，上个星期她还在问我，在不同应用之间切换的快捷键是什么（我告诉她，是 `Command+Tab`）。在这之前，她一直用鼠标在屏幕下端的应用图标中点选来切换应用。很多人都是这样：即便是自己每天都在使用的工具，也并没有花什么精力去琢磨怎么能用得更高效。即使已经用了十年，他们仍然以业余用户的姿态在使用这个工具。

但是，可不要小看这位设计师。一旦进入 Photoshop，那就是她的天下了。她会熟练地用快捷键进行各种操作，速度可谓出神入化。在她的专业领域里，她在职业生涯的早期就花了精力熟练掌握最常用的工具，这些投入在之后的很多年中一直让她保持高效，节省她的宝贵时间。每

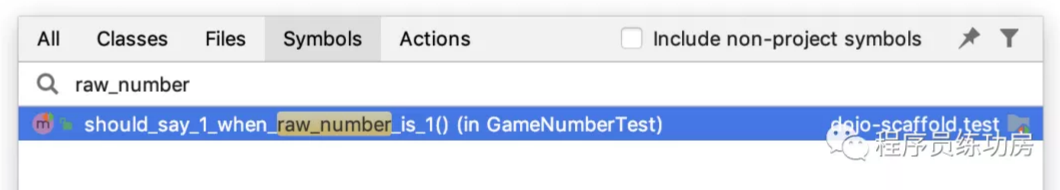
个专业人士都应该像这位设计师一样，花一些精力去熟练掌握自己最常用的工具，而不是像业余用户一样慢慢悠悠地抓着鼠标左点右点。

在 Java 和 Web 开发的领域里，IntelliJ IDEA 可能是最出色的集成开发环境（IDE）没有之一。所以对于在这个领域工作的读者，我强烈建议你使用这款 IDE。

IntelliJ IDEA 最出色的特点之一，就是它几乎对所有任务都有快捷键支持。根据我的经验，你完全可以用 IDEA 编程一整天，完全不用碰一下鼠标——这是件好事，因为这样你的手指会始终保持在主键位区，而一旦你伸出手去抓鼠标，再把手放回键盘时又需要额外的一点精力来找到主键位区。我看到很多人实际上放弃了在鼠标和键盘之间的切换，把右手始终放在鼠标上，只用左手敲打键盘。你可以想象，这样的编程速度会有多慢。所以，一个优秀的 IDE 不仅仅是“为很多任务提供快捷键”，而是应该为几乎所有任务都提供快捷键，让用户可以完全避免使用鼠标。在这一点上，IDEA 做得特别到位。

IDEA 常用的快捷键有以下几类：

- **定位类**。准确定位到需要编辑的位置，接着只需要用方向键微调光标位置就可以直接开始编码。这类快捷键大多是围绕着 `Command` 键展开的。除了前面介绍过的 `Command+n`、`Command+Shift+t`、`Command+e` 等快捷键之外（你还记得这些快捷键的用途吗？），还有一个很强大的快捷键 `Command+Option+Shift+n`（查找符号）。另外，如果确实需要到左侧导航栏里浏览查看，你始终可以用 `Command+1` 进入导航栏，或者用 `Option+F1` 在左侧导航栏里找到当前正在编辑的文件。

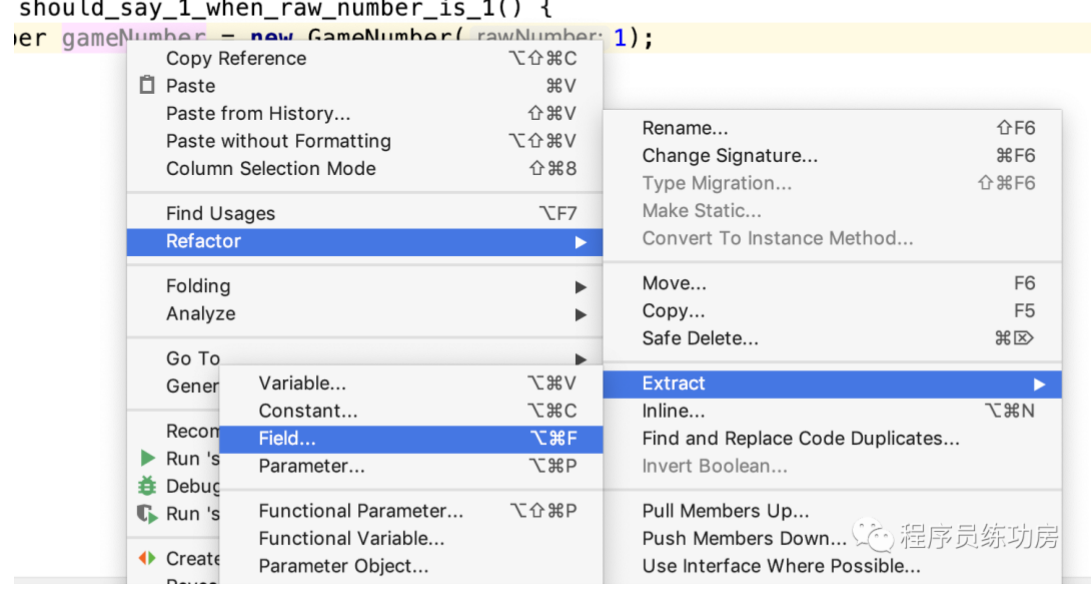


- **代码处理类**。有很多基本的代码处理，都可以用快捷键完成。例如 Command+w 可以逐级扩大反选范围，Command+d 可以复制当前行，Command+x 可以删除当前行，Option+Shift+方向键可以把一行代码上下移动，Command+Shift+方向键则可以把整个语句或方法上下移动。

- **运行类**。在 TDD 的节奏中，你需要频繁运行所有测试，很可能每小时要运行数十次。“运行”的快捷键大多是围绕着 `F10` 键展开的。你可以首先用 `Command+1` 进入导航栏，选中整个 `test` 目录，然后按 `Option+Shift+F10`，运行该目录下的所有测试。在这之后，你就只需要每次按 `Shift+F10`，就可以立即执行所有测试。另外，照理来说你应该不需要使用调试器（debugger），但如果因为什么奇怪的原因你真的需要调试，按 `Shift+f9` 就可以用调试模式运行所有测试。

- **重构类**。重构也是每隔几分钟就要发生一次的事，所以常用的重构手法都有快捷键支持。你可以随时在一段代码中调出“Refactor ” 键菜单，查看有哪些可选的重构手法。常用的重构类快捷键又可以分为两个主要家族：
  - 第一个家族是“提炼型重构手法，快捷键基本上是 `Command+Option` 与重构手法关键字首字母的组合。例如“提炼函数”（Extract Method）的快捷键是 `Command+Option+m`，“提炼变量”（Extract Variable）的快捷键是 `Command+Option+v`，“提炼字段”（Extract Field）的快捷键是 `Command+0ptio+f`，等等。你可以在代码中选中一个变量，在右键弹出的“Refactor

”菜单中找到更多的“提炼型操作。另外，与“提炼”相对应的“内联”（Inline）手法也有相似的快捷键：按 `Command+Option+n` 就可以内联一个函数、变量、字段或者参数。

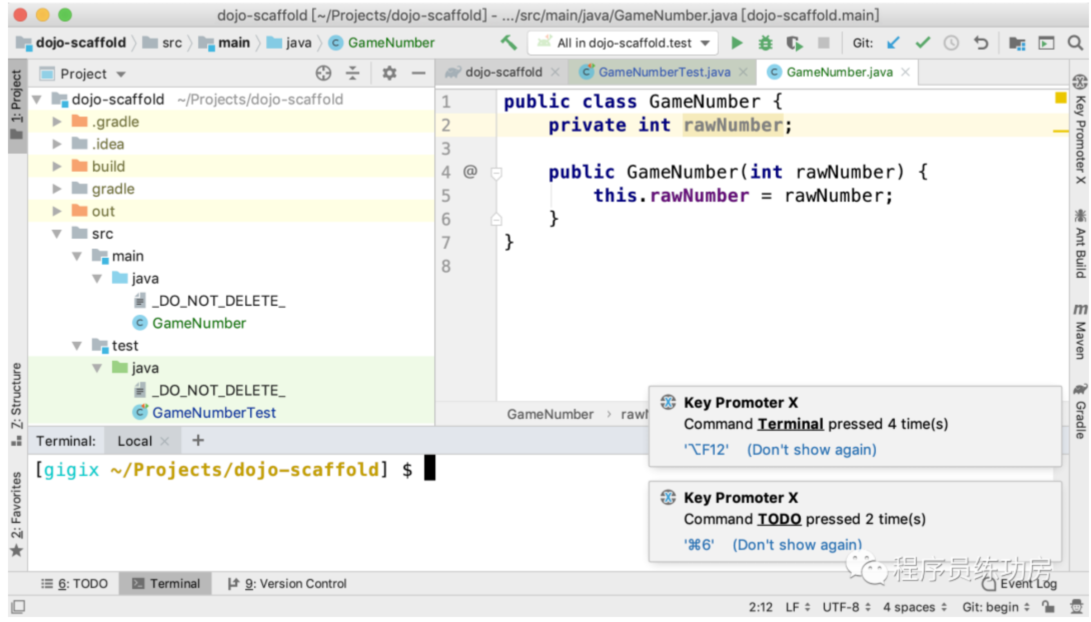


- 第二个家族是“改变”型重构手法，快捷键基本上是围绕着 F6 键展开。例如 F6 可以搬移（Move）函数、字段和常量，Shift+F6 可以给函数字段、常量、变量、参数等元素改名（Rename），Command+6 可以给函数改变签名（Change Signature）——熟悉《重构》的读者会知道，这几个重构手法乍一看用途各异，实现手法其实相似。

- **万能类**。这类其实就一个快捷键 `Option+回车`，但这一个快捷键使用的频率或许比得上其他所有。在所有编译报错的地方，你可以试试这个组合。在所有 IDE 告警的地方，你可以试试这个组合。即便没有报错也没有告警，你也可以不时试试这个组合，看看它会给你什么惊喜。我就把这份探索的喜悦留给读者自己发现了。

如果一下子记不过来这么多快捷键，别着急，有两个办法可以帮你逐步记住它们。首先，你可以在“Help-> Keymap Reference 菜单项中找到完整的快捷键列表，把它打印出来，贴在你随时能看到的地方，每次想动鼠标的时候你可以抬头看看，看你想要的操作应该用什么快捷键来完成。

另外，有一个叫做“Key Promoter X”的插件，可以随时提醒你应该用什么快捷键。每当你在 IDEA 中使用鼠标做一个操作，这个插件就会在右下角弹出提示，告诉你可以用哪个快捷键来完成同样的操作。



Neal Ford 在《卓有成效的程序员》这本书里讲了一个很重要的原则：**键盘输入总比导航快**。他在这本书的第 2 章专门介绍了各种提高工作效率的“加速技巧”，其中大量的技巧都与快捷键有关不仅仅是 DE 中的快捷键，还包括在操作系统中的各种加速操作。每位有志于打磨自己技巧的程序员都应该阅读他的这本书。

十二、更快地产出代码

最后，你可能已经对 IntelliJ IDEA 的快捷键足够熟悉，也基本上不再碰鼠标了，但你仍然可能面临一个速度短板：产出代码的速度就是不够快。

要想以更快的速度产出代码，有两个办法：

1. 你得尽量让开发工具帮你产出代码；
2. 你得提升击键的速度。

如果你认真看了我做 FizzBuzz 的视频录像，你可能会注意到：我在测试类里输入“test”，

```
public class GameNumberTest {
    test
}   test                         New Junit test case
    Press ^. to choose the selected (or first) suggestion and insert a d... afterwards >>
```

然后突然间，一个 JUnit 测试方法的骨架就蹦了出来，而且光标自动跳到了“should__”的后面，让我可以立即开始输入测试的名字：

```
public class GameNumberTest {
    @Test
    public void should_say_1_when_raw_number_is_1() {

    }
}
```

这就是一个典型的例子：开发工具帮我产出了一大堆代码。我只敲了“test”4 个字母，按了一下 Tab 键，就得到了一个测试方法的骨架。这样我就可以把注意力集中在写我真正需要写的东西。

这是 DEA 的一个功能，叫做“活动模板”（Live Template）。你可以进入“Preferences 对话框，搜索“Live Template ”，不难找到它们。一些常用的代码片段，IDEA 已经提供了对应的活动模板。例如你可以试试在代码中输入“fori ” 按 Tab 键，就会得到一个 for 循环的骨架代码：

```
for (int i = 0; i < ; i++) {

}
```

程序员练功房

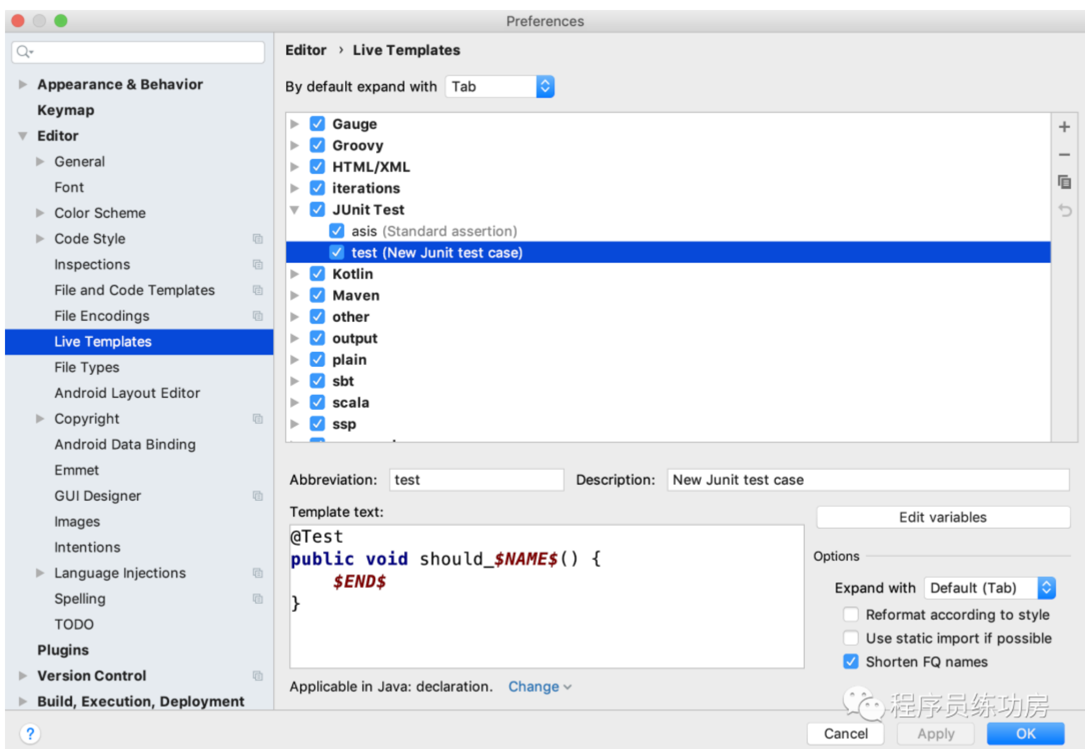
再比如你定义了一个列表变量，在其后输入“iter”，按 Tab 键，就会得到遍历这个列表的骨架代码：

```
List<String> texts = new ArrayList<>();
for (String text : texts) {

}
```

程序员练功房

当然还有每个 Java 程序员都熟悉的 System.out. println，你只要写“sout 然后按 Tab 键就行了。

除了 IDEA 缺省自带的模板之外，我自己又定义了两个 JUnit 常用的活动模板“test”（生成测试方法）和“asis”（生成 assertThat 断言）。看看下图，读者应该就知道怎么创建自己的活动模板了。



最后的最后，不管工具为你提供多少便利，代码还是得靠人来写。击键越准确、速度越快，你就越能够及时把脑子里的想法变成实际可运行的代码，你需要在脑子里暂存的东西就越少，你能有效处理的问题复杂度也就越高。根据我的经验，如果你能一边嘴上说着自己的思路，一边手上就把代码写出来，那么你的键盘速度基本上可以不阻碍你的思考；如果手上出代码的速度完全跟不上嘴里说思路的速度，那你就需要好好训练一下手指了。

手指对键盘的熟练度是最容易训练的项目。你可以访问 www.typing.io 网站，选择你习惯的编程语言，试一下你自己的速度。我最近一次尝试，在“Java Guava ” 这个科目上取得了 57 分（单位是“WPM”，“词每分钟”）的成绩，并有 5%的无效击键。这个成绩不算好，勉强可以支撑代码流出的速度跟上我思维的速度。很多年轻一些的优秀程序员，成绩都在 60 分以上。

Lesson Summary

typeable characters 334

typed characters 352

334 typeable characters
+ 6 incorrectly typed
+ 3 collaterally typed before backspacing
+ 9 backspaces

unproductive keystroke overhead 5%

352 typed characters – 334 typeable characters / 334 typeable characters

elapsed time 1:10

wpm 57

334 typeable characters / 5 characters per word / 1:10

程序员练功房

如果你的成绩还在 40 分以下，很可能代码流出的速度会对思维的速度造成阻碍。这时你就需要稍微花点时间练习了。我从过去的学员那里看到的经验，每天练上 3 次，坚持两个星期，速度就会有大幅度的提升。

千万别小看了键盘速度这件小事。几年前我在 ThoughtWorks 负责一届毕业生的入门培训时写过这样一段话：

也许你会嗤之以鼻：我是知识工作者，又不是搬砖的码农，比键盘速度有什么意义？错！大错特错！为什么？我们都知道，编程时很难把所有细枝末节都想清楚，大家都是想好一个大概方向，然后一边做一边展开细节也就是说，你对一个编程问题的思考，是在编码的过程中不断进行的，细节展开到什么程度，你的思维就延伸到什么程度所以，键盘速度会决定你思维的速度你思维敏捷？对不起，没有用，因为问题细节还没展开，你没法对它进行思考，你的脑子就只能等着

速度一慢，直接的结果就是时间要拉长，就以这几个毕业生来说，我敲键盘的速度就有他们两倍、三倍快，如果再考虑我对快捷键的熟悉程度，出代码的速度能有他们七八倍甚至十倍——和知识、经验都无关，纯粹就是出代码的速度。换句话说，同样一个简单的问题，大家都理解了应该怎么解决，我5分钟之内把代码写完，这些同学们就要写40~50分钟.这四五十分钟他们要一直保持对这个问题的专注思考，而同样程度的专注思考我只需要保持五分钟这就是为什么小菜鸟们经常说工作一天很累、经常在解决问题解决到一半时找不到方向：因为速度太慢键盘速度慢就会成倍地拉长大脑专注于一个问题的时间，就会成倍地增加大脑的负荷本来小菜鸟们知识、经验就缺，还用这种方式让大脑疲劳，当然很快就倒下啦。

键盘的速度会决定思维的速度，从而决定你的人生节奏，这绝不是危言耸听想想看，我们把每天最好的八个小时用在工作上，而在这八个小时里，一个人每小时能思考和解决两个问题，另一个人只能思考和解决一个问题那么一年过去，这两个人的差距会有多大？答案是，后者要比前者少思考和解决将近两千个问题！你想想这两个人工作一年之后会有多大差距、而造成这种差距的，很可能就是因为键盘速度的差距。

当你把环境和工具都准备好，熟悉了常用的快捷键，再花了一点时间来训练对键盘的熟练度，这时再回头来从头做一遍 FizzBuzz，很可能你会发现，自己编程的速度有了明显的提高。所谓专业，从来只会出自不断的训练。要相信，训练不说谎，只有通过反复训练，才能超越业余水平，进入职业水平。

那么，把代码删掉，再来一遍吧。你可以再掐表计一下时，与任务一记录下来的成绩做个对比。你编程的速度是否有明显的提升？是否逐渐进入了 TDD 的编程节奏？

十三、FizzBuzz 项目小结

用了这么长的篇幅来讨论 FizzBuzz 这么简单的一道题目，我希望读者从中有所收获。回顾一下我们的练功之路：

- 我们看到了 FizzBuzz 的需求
- 我们讨论了软件的设计是什么
- 我们用测试驱动开发的方式开始起步了
- 我们完成了 FizzBuzz 的主体逻辑
- 我们做了一点重构
- 我们完成了 FizzBuzz 整个游戏
- 然后又遇上了需求变更……好吧，扩展
- 我们看到有些人编程的速度飞快
- 于是我们开始学习如何尽量避免鼠标
- 以及如何用好 IDE 的快捷键
- 还有如何让自己用键盘编程的速度更快

你以为 FizzBuzz 真的就是看上去那么简单么？

你以为**编程真的就是看上去那么简单么？**

跑个题讲个故事。“鲍勃大叔 ” Robert Martin 曾经写了一篇博客，说有人跟他讲自己使用 TDD 的体验不好，感觉 TDD 并不灵光，然后鲍勃大叔细问了一下这人实践 TDD 的步骤之后，暗自感叹：“其实你做的根本就不是 TDD 呀。”

这已经成了一个常见现象了。我听很多人说过类似的话，说 TDD 有这样那样的问题，然而一旦细问下去，几乎每次都会发现：他们并没有先写测试后写代码；他们并没有每隔一两分钟就经历一遍“红－绿 ” 的循环；他们并没有随时检查自己的代码是否需要重构……简而言之一句话，他们并没有在实践 TDD。他们只是以为自己做的就是 TDD，然后说 TDD 有这样那样的问题、TDD 的效果不好，等等。

我不希望我的读者和学员走进这样的歧途。即使到最后你还是要批评 TDD——这当然是有可能的，没有什么方法是不可批评的——我希望你真的掌握这种方法，真的实践它，然后再批评它

。

所以我希望帮你打下一个扎实的基础，让你站在和我一样的起跑线上，我们接着一步步去探索 TDD 的方方面面。这个扎实的基础包括：

- 看清楚每一行代码是如何严格遵循“红绿”的节奏写出来的；
- 了解什么程度的“坏味道就需要引起关注；
- 看清楚一个重构手法是如何一步步以绝对安全的方式开展的；
- 配置好自己的工作环境；
- 对产出代码的速度做有针对性的训练。

老话说“磨刀不误砍柴工”，有了这些基础的准备之后，我就可以更放心地带你进入稍微复杂一点的 TDD 练习了。不过在此之前，我需要再检查一遍：还记得我们在第一个项目里学到的编程原则吗？

> 编程原则
> 1.没有失败的测试就不允许修改软件的行为。
> 2.只允许做恰好让测试通过的修改。
> 3.在保证不改变软件行为的前提下，可以对代码进行重构，消除其中的坏味道。
> 4.键盘操作优于鼠标操作。
> 5.搜索优于导航。

好的，默念这几条简单的原则，我们准备出发去下一个项目吧。

十四、第二个项目：单位换算

相信 FizzBuzz 已经给了你充分的信心。经过多次反复的练习，你对 TDD 的节奏感应该已经熟悉。接下来，我们稍微提高一点点难度，再来做一个很简单的项目，帮助你温习这些基础。古语云“温故而知新”，刚学到的功夫，一定要换着姿势多练几次才会熟练。

下面就是第二个项目的问题描述：

问题描述

美国人习惯使用很古怪的英制度量单位。英制度量单位的换算经常不是十进制的，比如说：

- 1 英尺（foot）= 12 英寸（inch）
- 1 码（yard）= 3 英尺
-

任务：写一个程序，用于处理英寸、英尺、码之间的换算。

你的程序应该能很好地处理下面这些换算关系：

- 1 英尺 应该等于 12 英寸
- 1 码 应该等于 3 英尺
- 1 码 应该等于 36 英寸
-

Kent Beck 在他的《测试驱动开发》书中用了一个类似的题目，不过不是长度单位的换算，而是货币之间的换算。看来“不同单位之间的换算”这类问题可能确实有一定的代表性。
现在，抓起键盘，默念 TDD 的节奏，开始编程吧。

先写测试，后写代码
没有失败的测试就不能写代码
只能写恰好让测试通过的代码

十五、实现单位换算逻辑

> 今天的任务
> 写一个程序，用于处理英寸、英尺、码之间的换算：1 英尺等于 12 英寸，1 码等于 3 英尺。
> 要求：严格遵循 TDD 的节奏，小步前进。

万事开头难。采用 TDD 方法开发软件时，最难的问题总是“第一个测试是什么”。

每次我在课堂上提这个问题，总会有同学迫不及待地说：“我的第一个测试是 1 英尺可以换算成 12 英寸。”然后我就会说：“你太急了。”

还记得我们在前一个项目里学到的吗？测试是个神奇的许愿池。如果你每次都许很小很小的愿望，你的愿望就会一个接一个实现。然而如果你太贪心，一下子许一个很大的愿望，它实现起来就会比较困难，甚至有时候你会发现实现不了所以我们不要着急，再动脑筋想一想，你能许的最小的愿望是什么？

请读者先自己动手写一遍，再来读我的讲解，效果会更好。

我能想到的最小的愿望是：我希望可以创建代表“1 英寸”的对象。

```
public class InchTest {
    @Test
    public void should_create_one_inch_object() {
        Inch inch = new Inch(1);
    }
}
```

程序员练功房

非常简单：你想要什么，你就直接在测试里说你想要什么。在前一个项目里我们也是以类似的方式起步的。这是一个好习惯，希望这次重复之后你能记住它：

> 编程原则
> 对于你要处理的数据，首先创建一个对象来表示它。

只要写最简单的代码让编译通过就行了实际上你几乎不用写代码，IDEA 可以帮你生成所需的代码。Is

```
public class Inch {
    public Inch(int amount) {
    }
}
```

程序员练功房

运行测试，绿色。我们继续许愿。仍然保持谨小慎微的步伐，我们的第二个愿望是：我希望 1 英寸等于 1 英寸。

```
    @Test
    public void should_1_inch_equal_to_1_inch() {
        assertThat(new Inch(1), is(new Inch(1)));
    }
```

程序员练功房

运行测试，红色。

之所以此时测试会失败，因为我们没有覆写（override）`Inch` 类的 `equals` 方法，所以断言中出现的是两个不同的对象，没人知道它们其实是“相等”的。在测试失败的报错信息中就可以看到：

这个出错信息很不直观：它只告诉我们“这里有两个 `Inch` 对象，它们不相等”，我们完全不知道这两个对象里面到底是什么。我们可以拿个便利贴把这个问题先写下来，等一会儿一等我们回到绿色状态的时候－再来优化这个问题。在我们编程的过程中不时会冒出一些新的想法、或是有一些新的发现，但是，如果你当前正处于红色的状态，千万不要马上开始做另一件事。先拿个便利贴把你的新想法、新发现写下来贴在电脑边上。等完成当前的任务、测试回到绿色状态时，再去考虑要不要处理便利贴上的任务。

> 便利贴
> 优化 `Inch` 对象，在出错时能看到其内部的信息。

我们把便利贴贴在电脑边上，先去覆写 `equals` 方法，用最简单的代码让测试通过：

```
public class Inch {
    private final int amount;

    public Inch(int amount) {
        this.amount = amount;
    }

    @Override
    public boolean equals(Object obj) {
        return amount == ((Inch)obj).amount;
    }
}
```

程序员练功房

别着急，别着急。我知道覆写 equals 方法其实是件不太简单的事。一个教科书式的 equals 方法应该首先检查传入的对象是否为 null、是否跟 this 对象同一类型、然后再做类型转换、再比较内部数据。并且在覆写 equals 方法的同时也应该覆写 hashCode 方法。这些我都知道。不过为了简单起见，在这里我偷懒用最简单、相当欠考虑的方式来实现 equals 方法。在生产代码中，你不应该这样做。现在出于教学目的，我们先将就一下。

运行测试，绿色。现在可以拿起我们刚写的便利贴，来解决“Inch 对象在出错时信息不直观”的问题。当然，我们也要先写一个测试：

```
@Test
public void should_display_internal_information_friendly() {
    assertThat(new Inch(1).toString(), is("1 (Inch)"));
}
```

程序员练功房

运行测试，红色。实现它很简单，覆写 toString 方法，返回数量和单位。

```
@Override
public String toString() {
    return format("%d (Inch)", amount);
}
```

程序员练功房

运行测试，绿色。刚才的便利贴可以撕掉了接下来我们终于可以开始考虑许愿“1 英尺等于 12 英寸”了。自信的读者可能已经写出了这样一段测试：

```
@Test
public void should_1_foot_equal_to_12_inches() {
    assertThat(new Foot(1), is(new Inch(12)));
}
```

程序员练功房

等等这个写法，Foot 和 Inch 不是同一个类型对吧？不是同一个类型的两个对象怎么比较相等性呢？

当然了，聪明的读者会马上给出一串可能的办法。比如说，有人会建议：可以分别覆写 Foot 和 Inch 的 equals 方法，并辨别彼此的类型；还有人会建议：可以让 Foot 和 Inch 集成同样的父类，然后在父类中覆写 equals 方法，从而兼容两个类。

我仍然提出那个老调重弹的问题：步子太大了吧？你有多大的信心，能在两分钟内写完这些代码，让上面这个测试通过？我一想到要修改至少两个函数、或者要创建至少两个类，就觉得有点头皮发麻。我没有信心能在两分钟内完成这一步。

有没有什么办法，能让我们以最简单的方式表达“1 英尺等于 12 英寸”的愿望，并能在两分钟之内就完成实现呢？

答案是有的。我们可以首先对 Inch 类做一次改名，将它改为 Length，代表“长度”。“给类改名”（Rename Class）这个重构手法，可以用 Shift+F6 快捷键完成。IDEA 还会贴心地问你：“要不要顺便把 InchTest 也改名成 LengthTest 呀？”那当然很好，谢谢你，IDEA。不过接着 IDEA 又问了：“你有个变量叫 inch 要不要顺便也改名成 length？ ”那就不用了，它还是叫 inch 比较合适。

```
@Test
public void should_1_inch_equal_to_1_inch() {
    assertThat(new Length(1), is(new Length(1)));
}
```

程序员练功房

运行测试，绿色。不过这个测试看着有点不对劲—所有 3 个测试都有点不太对劲。我们创建了一个数量为 1 的 Length 对象，但没有说它是什么单位，那我们怎么能说这是“1 英寸”呢？所以我们得把单位信息加上。我经常说，在 IEA 里面编程，你想要什么就写什么。比如我现在想要在构造 Length 对象的时候多传入一个“单位”的信息（类型是字符串），我就直接在调用构造函数的地方写上这个新的参数，然后在编译报错的地方按 Option+Enter 键，让 IDEA 帮我想办法：

```
@Test
public void should_1_inch_equal_to_1_inch() {
    assertThat(new Length( amount: 1, "Inch"), is(new Length( amount: 1)));
}
                                   Add 'String' as 2nd parameter to method 'Length'
                                   Create constructor
                                   Inject language or reference
@Test
public void should_display_internal_information_friendly() {
    assertThat(new Length( amount: 1).toString(), is( value: "1 (Inch)"));
}
```

程序员练功房

IDEA 提的建议很不错：给 Length 的构造函数添加一个类型为 String 的参数。我们就照它的建议来。别忘了把这个新加的参数改名叫 unit：

```
public Length(int amount, String unit) {
    this.amount = amount;
}
```

程序员练功房

挺好，这个参数完全没被使用，所以它绝对不会破坏程序现有的行为。我们运行测试，果然是绿色。已有的测试现在看着也对劲了：

```
@Test
public void should_1_inch_equal_to_1_inch() {
    assertThat(new Length(1, "Inch"), is(new Length(1, "Inch")));
}

@Test
public void should_display_internal_information_friendly() {
    assertThat(new Length(1, "Inch").toString(), is("1 (Inch)"));
}
```

程序员练功房

做这么一个小小的调整之后，我们再来许“1 英尺等于 12 英寸”的愿望就简单多了：

```
@Test
public void should_1_foot_equal_to_12_inches() {
    assertThat(new Length(1, "Foot"), is(new Length(12, "Inch")));
}
```

程序员练功房

运行这个测试，红色。测试会失败，这我们并不感到意外，但错误日志里的信息有点令人在意：

```
java.lang.AssertionError:
Expected: is <12 (Inch)>
     but: was <1 (Inch)>
```

程序员练功房

我们没有在 toString 方法里处理“不同的单位”这回事，所以代表“1 英尺”的对象被打印出来却成了“1 英寸”。这算是我们偶然发现的一个小 bug。不过现在测试还是红色，所以我们先拿个便利贴把这件事记下来，等会儿再来处理：

> 便利贴
> 在 `toString` 方法里处理不同的单位，使“1 英尺”能正确显示。

贴好便利贴，现在我们回去处理失败的测试，现在这个逻辑实现起来就超级简单，我们只要把所有长度都兑换成英寸再做对比就行了：

```
public Length(int amount, String unit) {
    if (unit.equals("Foot")) {
        amountInInch = amount * 12;
    } else {
        amountInInch = amount;
    }
}

@Override
public boolean equals(Object obj) {
    return amountInInch == ((Length) obj).amountInInch;
}
```

程序员练功房

（这一步实现还包含了对一个字段的改名。我相信你已经足够熟练，就不再详细讲解了。）运行测试，绿色。现在拿起刚才的便利贴，我们增加一个断言来描述这个 bug：

```
    @Test
    public void should_display_internal_information_friendly() {
        assertThat(new Length(1, "Inch").toString(), is("1 (Inch)"));
        assertThat(new Length(1, "Foot").toString(), is("1 (Foot)"));
    }
```

程序员练功房

运行测试，红色。这只是个小 bug，一分钟就能修好。完成之后的整个 Length 类长得这样：

```
public class Length {
    private final String unit;
    private final int amount;
    private final int amountInInch;

    public Length(int amount, String unit) {
        this.amount = amount;
        this.unit = unit;
        if (unit.equals("Foot")) {
            amountInInch = amount * 12;
        } else {
            amountInInch = amount;
        }
    }

    @Override
    public boolean equals(Object obj) {
        return amountInInch == ((Length) obj).amountInInch;
    }

    @Override
    public String toString() {
        return format("%d (%s)", amount, unit);
    }
}
```

程序员练功房

运行测试，绿色。可以撕掉刚才的便利贴了。虽然这段代码并不完美，但它至少是正确的。我们可以站起身来，先去打杯咖啡，休息一下，再接着处理更多的逻辑。

> 编程原则
> 发现一个 bug 时，首先编写一个失败的测试来描述这个 bug

 Length 的构造方法看着不够漂亮，长度也超过了 5 行（一般而言，我坚持每个函数不超过 5 行），不过用《重构》第三章的坏味道列表来比对一番的话，我还真是说不出这里有什么坏味道。所以，我们可以先忍一忍，继续往下写，实现对“码”（Yard）的支持。有时候你需要让代码的坏味道稍微发酵一点，这样你能更清晰地闻到这个坏味道到底是什么。

> else 关键字经常会导致代码质量迅速下降，因为它会让程序员不知不觉间写出过于复杂的分支逻辑。我经常会看到，一旦开始使用 else 关键字，很快就会酝酿出“长函数或“switch 语句”等坏味道。在《软件开发沉思录》一书的《对象健身操》文中，我以前 ThoughtWorks 的同事 Jeff Bay 提出“拒绝使用 else 关键字的练习规则。这是一个很好的练习，感兴趣的读者可以尝试一下。

我们接着许下一个愿望：1 码等于 3 英尺。

```
@Test
public void should_1_yard_equal_to_3_feet() {
    assertThat(new Length(1, "Yard"), is(new Length(3, "Foot")));
}
```

程序员练功房

运行测试，毫不意外地，我们看到了红色。相信这时候你已经进入了节奏，红色不再让你紧张，只需要按部就班地实现愿望就好了：

```
public Length(int amount, String unit) {
    ...
    if (unit.equals("Foot")) {
        amountInInch = amount * 12;
    } else if (unit.equals("Yard")) {
        amountInInch = amount * 36;
    } else {
        amountInInch = amount;
    }
```

程序员练功房

运行测试，绿色。现在我们顺利完成了“单位换算”这道题的功能需求，而且代码的坏味道也逐渐清晰起来。在明天的任务中，我们就来聚焦解决这里的坏味道。我们下个任务见。

十六、任务二：重构 switch 语句

> 今天的任务
> 识别代码中的坏味道，运用重构手法消除坏味道。
> 要求：严格遵循重构的十六字口诀，小步前进，随时保持测试处于绿色状态。

在上一个任务中，我们实现了“英制长度换算”的逻辑，并得到了这样一段代码：

```
    public Length(int amount, String unit) {
        ...
        if (unit.equals("Foot")) {
            amountInInch = amount * 12;
        } else if (unit.equals("Yard")) {
            amountInInch = amount * 36;
        } else {
            amountInInch = amount;
        }
```

程序员练功房

一开始，只有一对 if...else...的时候，还不太容易看出究竟这里的坏味道是什么。等我们实现了整个需求，有了两对 if...else...，眉目就清晰多了：原来，这里的逻辑随着 unit 变量的值在改变。虽然没有明确使用 switch 语句，这其实就是《重构》第三章里讲的“switch 语句”坏味道。

在《重构》第二版里，MartinFowler 已经放宽了对 switch 语句的态度，将这条坏味道改为“重复的 switch ”。然而我个人而言仍然对 switch 语句抱持着严苛的态度。我会尽可能地消除 switch 语句，因为中国行业的现状，正如 Martin Fowler 所描述的美国在 1990 年代末期的情况：“程序员们太过于忽视多态的价值，我们希望矫枉过正。”

现在问题来了：你打算怎么重构这段代码?

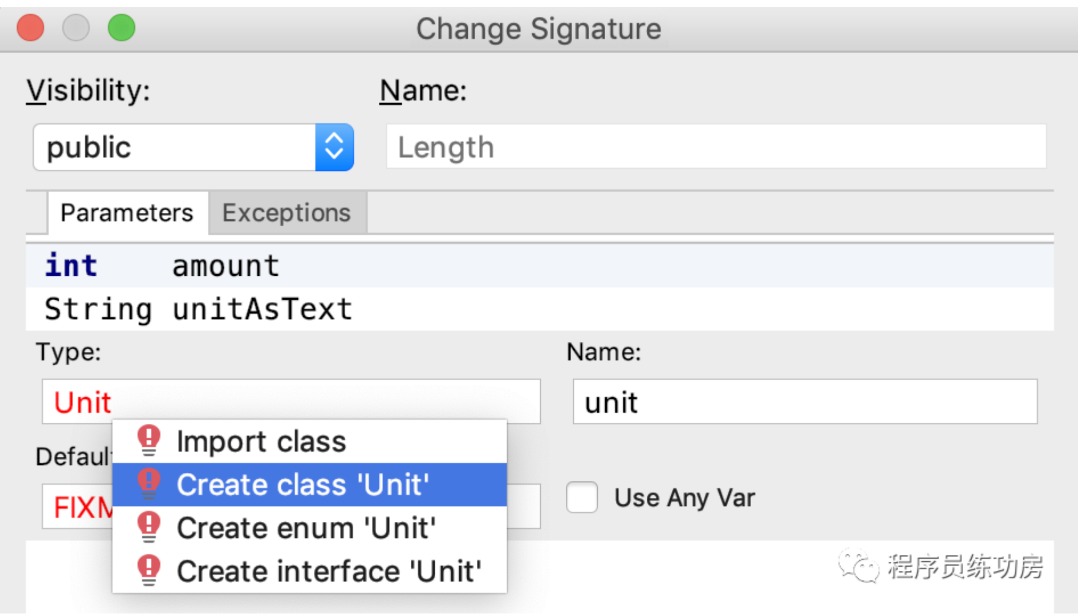
读者不妨自己重构一下试试看。你可以先 git commit 一下，在此基础上开始重构。如果重构效果不好，你随时可以用 git stash 回到现在的状态。

所谓重构，不是凭着美好的愿望随便乱改代码。重构的过程必须严守纪律、小步前进，每一步修改都必须保证代码行为不变。回忆一下前面提到过的重构十六字口诀：

> 旧的不变
> 新的创建
> 一步切换
> 旧的再见

我们要用这个思路来指导重构操作，保持步伐小而平稳。

《重构》第三章解释了为什么 switch 语句是一种坏味道：switch 语句常常是在根据类型码进行选择，因此一个 switch 就意味着本应该存在一个“与该类型码相关的类”。在我们这里，很明显，这个缺失的类名字就应该叫 Unit。不用着急去想这个类应该是什么样，还是跟往常一样，想要什么，我们就写什么。既然我们认为长度”（ Length）应该有“单位”（Unit），我们就对 Length 的构造函数运用“变更函数签名”（Change Signature）的重构手法（快捷键是 Comamnd+F6），加上这个参数：



在邀请 IDEA 帮我实施这个重构手法时，有几个小技巧可以分享：

1.我顺便把原来的 unit 参数（String 类型）改名为 unitAsText，以便给新的参数留下 unit 这个得体的名字；

2.我可以直接在显示红色编译错误的 Unit 类名上按 Option+Enter 键创建这个类；

3.我把新参数的缺省值设置为 FIXME，稍后你就会看到这样做的意义所在。

按下回车，完成这个重构，再尝试运行测试我们马上就会看到测试代码中出现了一堆编译错误：FIXME 是个不存在的符号，所以凡是使用 Length 构造函数的代码都无法编译。

```
public class LengthTest {
    @Test
    public void should_create_one_inch_object() {
        Length inch = new Length( amount: 1,  unitAsText: "Inch", FIXME);
    }

    @Test
    public void should_1_inch_equal_to_1_inch() {
        assertThat(new Length( amount: 1,  unitAsText: "Inch", FIXME), is(ne
    }

    @Test
    public void should_display_internal_information_friendly() {
        assertThat(new Length( amount: 1,  unitAsText: "Inch", FIXME).toStr:
        assertThat(new Length( amount: 1,  unitAsText: "Foot", FIXME).toStr:
    }
```

程序员练功房

刚才说了，这是个小技巧，我是故意为之的。把一个不存在的符号设置为新参数的缺省值，这会强迫我挨个检查所有调用这个函数的地方，给新参数赋上正确的值。这一步并不麻烦，因为完全不需要思考：之前以字符串形式传入的单位是 ” Inch ”，现在就把 FIXME 改成 unit.Inch 当然了，这个写法还是通不过编译，不过没关系，按 Option+Enter 创建一个字段就好：

```
d_create_one_inch_object() {
 new Length( amount: 1,  unitAsText: "Inch", Unit.Inch);
```

Create class 'Inch'
Create enum 'Inch'
Create field 'Inch' in 'Unit'
Create inner class 'Inch'
Create interface 'Inch'
Rename reference
Split into declaration and assignment

```
d_1_inch_equal_to_1_inch() {
w Length( amount: 1,  unitAsText: "Inch", FIXME), i
```

程序员练功房

其他的 FIXME 也如法炮制：原来以字符串形式传入的单位是 " Foot " 的，我们就把 FIXME 改
成 Unit.Foot；原来传入的是 " Yard " 的我们就把 FIXME 改成 Unit.Yard。完全不用动脑子思考，一分钟就改完。这时 Unit 类就变成了这样：

```
public class Unit {
    public static Unit Inch;
    public static Unit Foot;
    public static Unit Yard;
}
```

程序员练功房

运行测试，绿色。我们继续前进，给 Unit 类赋予更多的行为。首先可以让它做一点简单的事，比如说在 Length 的 toString 方法里说明自己代表哪个单位。这个行为已经有测试覆盖了，所以我们可以直接重构。首先，“**旧的不变，新的创建**”，给 Unit 类加上“知道自己代表哪个单位”的行为：

运行测试，仍然是绿色。然后，“**一步切换,旧的再见**”，在 Length 的 toString 方法中使用 Unit 的新行为：

```
    public Length(int amount, String unitAsText, Unit unit) {
        this.unit = unit;
        ...

    @Override
    public String toString() {
        return format("%d (%s)", amount, unit.text);
    }
```

程序员练功房

运行测试，仍然是绿色。

下一个可以让 Unit 类发挥作用的地方，就是 Length 构造函数里的 if...else...。我们可以根据传入的 Unit 参数来控制分支逻辑。

```java
public Length(int amount, String unitAsText, Unit unit) {
    ...
    if (unit == Foot) {
        amountInInch = amount * 12;
    } else if (unit == Yard) {
        amountInInch = amount * 36;
    } else {
        amountInInch = amount;
    }
```

程序员练功房

运行测试，绿色。这时候名为 unitAsText 的字段和构造函数参数都没人使用了，可以放心删掉。于是 Length 的构造函数签名就变成了这样：

```java
Length inch = new Length(1, Inch);
```

程序员练功房

这很好，比以前使用字符串字面量来代表“单位”要更干净。我们运行测试，仍然是绿色。继续观察上面这段 if...else...逻辑，我们发现其实它说的就是“每个单位都有换算到英寸的比率”——Foot的比率是12，Yard的比率是36，最后一个分支的意思其实就是Inch的换算比率为1。所以我们把“换算到英寸的比率”这个行为也放进Unit类去。还是老规矩，**“旧的不变，新的创建”**：

运行测试，绿色。接着，**“一步切换，旧的再见”**：

```java
public Length(int amount, Unit unit) {
    ...
    if (unit == Foot) {
        amountInInch = amount * unit.transferRateToInch;
    } else if (unit == Yard) {
        amountInInch = amount * unit.transferRateToInch;
    } else {
        amountInInch = amount * unit.transferRateToInch;
    }
```

程序员练功房

运行测试，绿色。当然了，上面这段代码其实就是下面这一句话：

```java
public Length(int amount, Unit unit) {
    ...
    amountInInch = amount * unit.transferRateToInch;
```

程序员练功房

运行测试，仍然是绿色。我们把这一句话提炼成一个方法（Extract Method，Command+Option+m），更清晰地表达它的意图“将长度换算成英寸”：

```java
public Length(int amount, Unit unit) {
    ...
    amountInInch = getAmountInInch(amount, unit);
}

private int getAmountInInch(int amount, Unit unit) {
    return amount * unit.transferRateToInch;
}
```

程序员练功房

运行测试，绿色。我们又注意到：`getAmountInInch` 这个方法只用到了 `Unit` 的数据，根本没有用到 `Length` 的数据。这是一个很明显的标志，说明这个方法很可能更应该属于 `Unit` 类而非 `Length` 类。所以我们把它直接搬移到 `Unit` 类去（Move Method，`F6`）。这时候 `Length` 的构造函数成了这样：

```java
public Length(int amount, Unit unit) {
    ...
    amountInInch = unit.getAmountInInch(amount);
}
```

程序员练功房

运行测试，绿色。纯粹出于个人偏好，我又把 `amountInInch` 这个字段给内联了（Inline Field，`Command+Option+n`），并顺手把"以字符串形式显示带单位的长度这个功能也提炼并搬移到了 `Unit` 类。于是 `Length` 类变得超级干净：

```java
public class Length {
    private final Unit unit;
    private final int amount;

    public Length(int amount, Unit unit) {
        this.amount = amount;
        this.unit = unit;
    }

    @Override
    public boolean equals(Object obj) {
        Length another = (Length) obj;
        return getAmountInInch() == another.getAmountInInch();
    }

    private int getAmountInInch() {
        return unit.getAmountInInch(amount);
    }

    @Override
    public String toString() {
        return unit.toString(amount);
    }
}
```

程序员练功房

运行测试，绿色。再去 `Unit` 类做一些最后的整理：所有的字段都可以声明为 `private`，构造函数也声明为 `private`，并把三个静态常量声明为 `final`，这样就没人能对这个类的任何对象做任何修改了。不可变的对象是最好的对象。

```java
public class Unit {
    public final static Unit Inch = new Unit("Inch", 1);
    public final static Unit Foot = new Unit("Foot", 12);
    public final static Unit Yard = new Unit("Yard", 36);

    private final String text;
    private final int transferRateToInch;

    private Unit(String text, int transferRateToInch) {
        this.text = text;
        this.transferRateToInch = transferRateToInch;
    }

    int getAmountInInch(int amount) {
        return amount * transferRateToInch;
    }

    String toString(int amount) {
        return format("%d (%s)", amount, text);
    }
}
```

程序员练功房

重构完成。我们最后一次运行测试，仍然是绿色。现在看看这两个类，它们是不是更清晰地讲述了“长度单位换算”这件事的逻辑？而且从纯粹代码审美的角度来说，两个类都很小，都是不可变（immutable）的类，没有不必要的信息暴露，所有函数都非常短（不超过 2 行），完全没有条件分支逻辑。对这个程序，我感到很满意。

比起最终结果这两个漂亮的类，我觉得更满意的是：我并没有刻意去设计这样的两个类，只是跟着坏味道的指引，遵守重构的纪律，一点点重构下来，就得到了这样清晰又简洁的代码。重构使我们可以不必做大量的预先设计，最终也能得到极其优雅的设计，这极大地减轻了软件开发的难度和负担。在后面的章节里，我们还会更深入地讨论重构的话题。

怎么样，单位换算这题够简单吧？要不，把代码删掉，你再来自己从头练一遍？

十七、单位换算项目小结

第二个项目“单位换算”很简单，在本书中所占的篇幅也是几个项目中最短的。在两天的任务中，我们：

- 严格遵循“红–绿”的节奏实现了英寸、英尺、码的换算逻辑；
- 观察代码中的坏味道，并遵循重构十六字口诀、运用多个重构手法，将其重构成相当干净的两个类。

这个任务的讲解，我明显加快了速度，因为我认为经过 FizzBuzz 的训练，这些基本操作你做起来应该得心应手。单位换算这道题也非常简单，照理说应该也就是几分钟时间

能做完的。如果你发现自己还不能完全跟上讲解的节奏，或者关上书独立练习的时候还不流畅，我建议你再多做几遍练习，把这些操作练熟。

顺便，再重温一下我们在前面两个项目里学到的编程原则：

> **编程原则**
> 1. 没有失败的测试就不允许修改软件的行为。
> 2. 只允许做恰好让测试通过的修改。
> 3. 在保证不改变软件行为的前提下，可以对代码进行重构，消除其中的坏味道。
> 4. 键盘操作优于鼠标操作。
> 5. 搜索优于导航。
> 6. 对于你要处理的数据，首先创建一个对象来表示它。
> 7. 发现一个 bug 时，首先编写一个失败的测试来描述这个 bug。

这里我给各位读者推荐一本课外读物：《敏捷中国史话》，作者是我本人。在这本书中，我尝试从行业史的角度，梳理敏捷传入中国及发展变迁的历程。今天推荐大家读的是这本书的第二章《对软件工程的渴望》。各位可以带着这两个问题去读这一章内容：

- 软件开发的基本能力有哪些?
- 以 CMM 为代表的软件工程给中国软件业建立这些基本能力了吗?

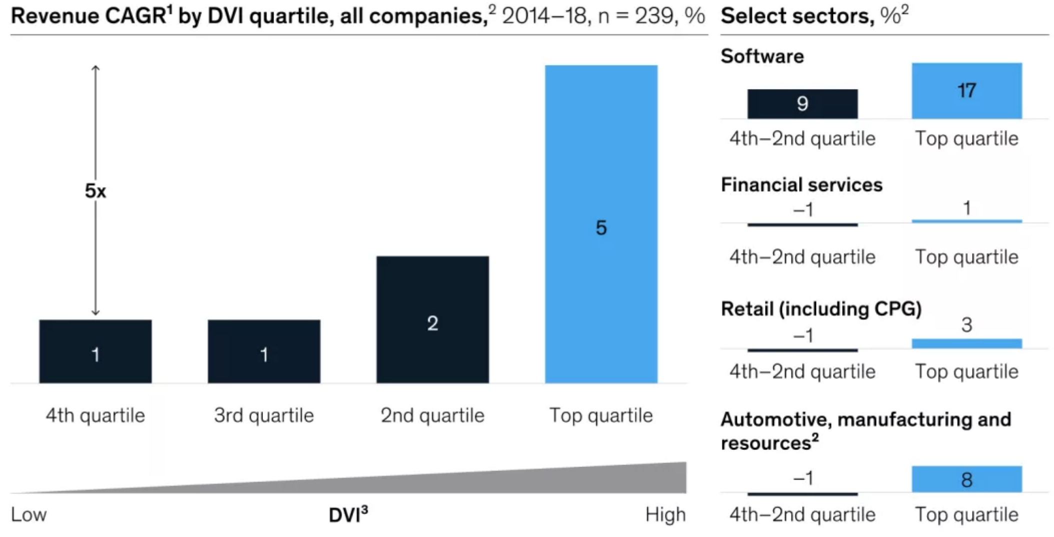
“敏捷宣言”的第一句就讲：**人和交互重于 流程和工具**。只有具备扎实基本功的程序员，以有效的方式交互协作，才有可能做出好的软件。想要绕开基本功的问题，靠流程和工具让能力不足的程序员也能写出好程序，这就是痴人说梦。然而，这样的美梦，在我们这个行业里有很多人在做，以至于整个行业都把基本功不扎实视为常态了。

要改变中国软件业的乱象，要提升中国软件业的整体水平，就非得从软件开发的基本功开始抓起不可，就非得从每个测试、每次重构开始练起不可。

十八、当我们谈软件卓越我们在谈什么

我们这个“TDD 练功房”的连载，到这里就要先告一段落了，在结束连载之前，咱们来聊点高视角的话题吧。

麦肯锡最近有两篇文章，是我很喜欢的调调。第一篇是今年 4 月的《开发者速度：软件卓越如何助推商业绩效》。这篇文章讲，软件开发者速度指数（Developer Velocity Index，DVI）较高的企业与较低的企业相比，业务营收复合增长率可以差出 5 倍。



什么是开发者速度指数呢？这篇文章说，这是由 13 个维度、46 项因素组成的一个复合指标，涉及了软件团队使用的技术、工作方法和组织赋能。其中，**测试自动化、测试驱动开发、技术债管理、CI/CD 等源于极限编程的工程技术实践都赫然在列。**

Developer Velocity involves 46 different drivers across 13 dimensions.

Technology

Architecture
- Software architecture
- Data architecture

Infrastructure and platform
- Public cloud adoption (IaaS, PaaS[1])
- Infrastructure as code

Testing
- Test automation
- Test-driven development

Tools
- Planning tools
- Collaboration tools
- Development tools
- DevOps tools
- Low- or no-code tools
- AI assistance in development

Working practices

Engineering practices
- Technology debt management practices
- Coding guidelines
- Code reviews
- CI/CD practices[2]

Security and compliance
- Security practices
- Compliance practices

Open source and InnerSource
- Open-source usage and contribution
- InnerSource adoption

Agile team practices
- Work-in-progress management
- Agile ceremonies
- Definition of done

Organizational enablement

Team characteristics
- Autonomous scope
- Limited context switching
- Cross-functional teams
- Colocation of teams

Product management
- Product management capabilities
- Product telemetry
- Product vision
- Linkage between strategy and team metrics
- Rapid prototyping

Organizational agility
- Dependency management
- Funding mechanisms
- Portfolio management

Culture
- Psychological safety
- Collaboration and knowledge sharing
- Continuous improvement culture
- Servant leadership
- Culture of customer obsession

Talent management
- Incentives
- Capability building
- Recruiting
- Team health management
- Employee value proposition
- Engineering career paths

程序员练功房

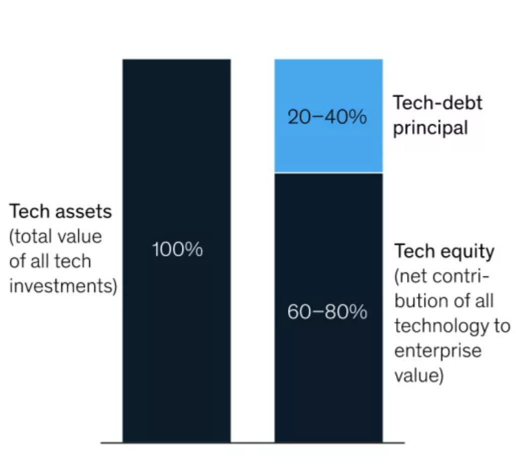
[1] Infrastructure as a Service, Platform as a Service.
[2] Continuous integration/continuous development.

另一篇文章是最近的《技术债：收回技术资产》。这篇文章说，**IT 投资里有 20~40%变成了技术债，这些债务又继续造成后续项目至少付出 10%的“复杂度税”**。战略、架构、人才、流程的问题，都有可能导致技术债。

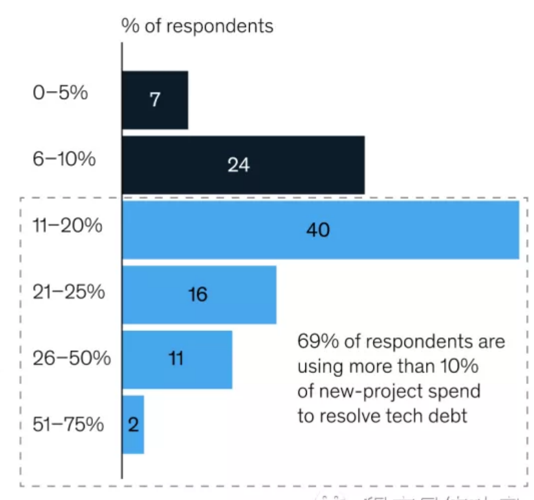
Tech-debt principal accounts for up to 40 percent of IT balance sheets, while most companies pay more than 10 percent interest on projects.

CIO estimates of spend on technology debt

Principal: Relative share of debt and equity on tech balance sheets



Interest: Estimated share of new-project spend allocated to resolving tech debt



程序员练功房

Source: McKinsey survey of tech debt among 50 CIOs, July 2020

所以，当我们谈软件卓越，我们是在谈什么呢？我们谈的不是程序员的技术自嗨。我们谈的是实实在在的商业价值。卓越的软件技术团队，能为企业带来 5 倍的增长率；糟糕的软件技术团队，能浪费 40%的 IT 投资。

这些数字在行业野蛮生长、猪都能飞的年代，或许显得不那么重要；但在增长放缓、企业都在强调提质增效的年代，是否具备卓越的技术能力，就会成为企业的关键竞争力。这就是我在《 To B投资的逻辑和软件卓越 》文中说的，

> 中国软件业的历史造就今天这个行业普遍不重视软件卓越……如今To B的风口……将造就一个新的蓝海市场，而这个蓝海市场是为过去二十年坚信软件卓越、扎实苦练软件开发基本功的人和企业准备的。那些过去二十年在软件行业里招摇撞骗、无视软件开发基本规律的人，在这个蓝海市场中注定被大浪淘沙。
>
> To B投资的逻辑和软件卓越

我们谈的软件卓越是通过刻苦的练习而获得的扎实技能。技术债的消除，最终都离不开重构、持续集成、自动化测试、测试驱动开发，而**这些工程技术实践只有通过刻意练习才能获得**。于是我们会发现，传统的、以老师讲述为主的培训形式，对获得软件卓越完全没有帮助——老师讲得再天花乱坠，那也是老师自己的事，**不能让学员开展有效刻意练习的培训，统统都是无用功。**

我们的“练功房”这种形式则成功地推动了学员们挽起袖子、抄起键盘，开始自己的刻意练习。然后从这些学员这儿，我听到了一些不常见的反馈。有人跟我说，这是他上过所有培训中收获最大、变化最大的；有人对自己练功后的编程速度连呼“恐怖”；还有人跟我一起喊出“信 TDD 得永生”的口号。

所以，当我们谈软件卓越，背后不可缺的一个基础是成规模、快节奏的工程技术能力建设。**有效的工程技术能力建设，必须是基于刻意练习理论的，必须能促使学员开展大量练习，必须能对学员的能力进行有效的评估。**缺乏工程技术能力的组织，不可能获得软件卓越。不基于刻意练习的工程技术能力建设，注定浮于表面。

一切改变都源于行动。没有追求卓越的行动，就不会有卓越的工程技术能力，当然也就不会有组织的软件卓越。个体和交互重于流程和工具，软件卓越必定从每个个体的能力提升开始。不积跬步无以至千里，这就是我想谈的软件卓越。