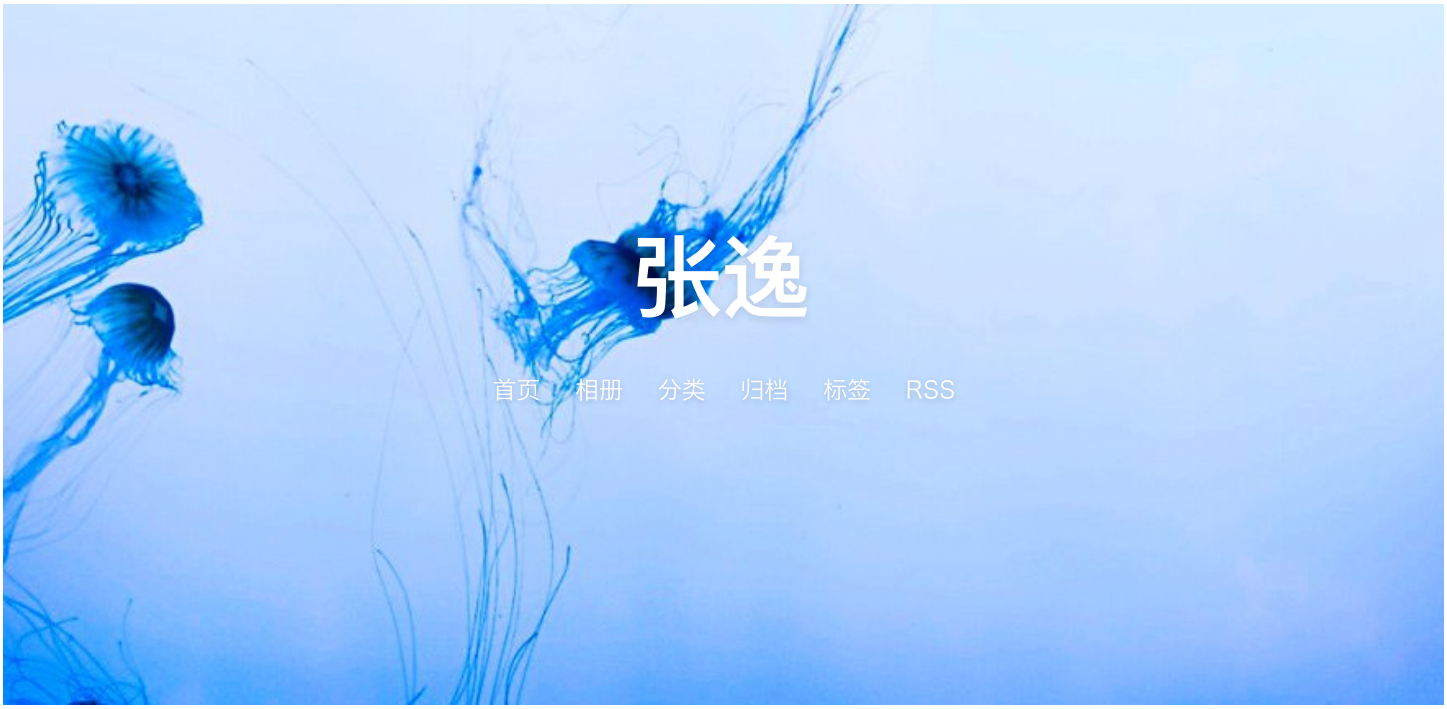

高质量代码的特征

回想起来，我觉得我们似乎在误读Uncle Bob的Clean Code，至少我们错误地将所谓Clean与可读性代码简单地划上了等号。尤为不幸的是，在Clean Code一书中，从第二章到第五章都围绕着可读性代码做文章，于是加深了这种错误的印象。

许多具有代码洁癖的程序员将代码可读性视为神圣不可侵犯的真理，并奉其为高质量代码的最重要特征，封上了“神坛”。殊不知，Uncle Bob在Clean Code的第一章就通过别人之口对所谓“Clean Code”进行了正名：所谓整洁代码并非仅仅是“清晰”这么简单。

按照Kent Beck的简单设计规则，排在第一位的其实不是可读性，而是“通过所有测试”。其中潜藏的含义是满足用户正确的需求，因为测试可以看做是用户提出的需求。这个需求不仅仅是业务上的，还包括质量属性的需求，例如性能、安全等属性。

消除重复和**提高表达力**这两点，有时候会互相促进，去除了冗余的代码，会让代码变得更加清晰；然而，有时候却又互相冲突，消除重复的成本可能会比较高，导致提取了太多细碎微小的实体，反而增加了阅读障碍。

故而我常常将Uncle Bob提出的“函数的第一规则是要短小。第二条规则是还要更短小。”看做是一种矫枉过正的强迫。对于那种喜欢编写大函数的程序员而言，确实需要时刻铭记这一原则，但切记不要将其视为最高准则。保证函数短小是有前提的，仔细阅读Kent Beck的简单设计原则，依其重要顺序：

- 能通过所有测试；
- 没有重复代码；
- 体现设计者的意图；
- 若无必要，勿增实体（方法、函数、类等）。

如果程序满足了客户需求，没有重复代码，函数的表达已经足够清晰地体现设计者意图，为何还要不断地提取函数，使得函数变得极为短小呢？真正有意义的原则是“让函数只做一件事情”。

正因为此，在Clean Code书中，Uncle Bob展示的对FitNesse中HtmlUtil.java的第二次重构并无必要。在经过第一次重构后，代码如下所示：

```
    public static String renderPageWithSetupsAndTeardowns(PageData pageData
    ) throws Exception {
        boolean isTestPage = pageData.hasAttribute("Test");
        if (isTestPage) {
            WikiPage testPage = pageData.getWikiPage();
            StringBuffer newPageContent = new StringBuffer();
            includeSetupPages(testPage, newPageContent, isSuite);
            newPageContent.append(pageData.getContent());
            includeTeardownPages(testPage, newPageContent, isSuite);
            pageData.setContent(newPageContent.toString());
        }
        return pageData.getHtml();
    }
```

这段代码的结构与层次已经非常清晰，也对实现细节做了足够合理的封装与隐藏。若要说不足之处，就是可以将如下代码再做一次方法提取，以满足SLAP原则（单一抽象层次原则）：

```
newPageContent.append(pageData.getContent());

//提取为：
includeTestContents(testPage, newPageContent)
```

而Uncle Bob做的第二次重构，除了将方法变得更加短小，隐藏了太多细节从而引入更多层次之外，究竟给代码的清晰带来了什么呢?

```
public static String renderPageWithSetupsAndTeardowns(PageData pageData, bo
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

过犹不及啊!

有时候，为了去除重复，就必须要从相似代码中寻找到一种模式或者某种抽象，进而对其进行提取。过分的提取反而会让代码变得很难阅读，这是因为提取的手段常常会引入“间接”。正如Martin Fowler所说：“间接性可能带来帮助，但非必要的间接性总是让人不舒服”。不必要的间接常常妨碍代码的直截了当和干净利落。倘若去除重复带来的唯一好处仅仅是避免一个类中少许的私有重复，去除这样的重复其实意义真的不大。

我喜欢清晰的代码，但在同时我认为保持代码的正确、健壮与高效同样重要。因为代码洁癖的缘故，我曾经将大量的非空判断、非法检查与异常处理视为干扰清晰代码的洪水猛兽，但如果不做这些“脏活累活”，代码就可能变得不健壮。在Java中，若真要避免这些判断，可以考虑转移职责，通过定义Checked Exception，将异常处理的职责转移给方法的调用者。然而，职责的盲目转移始终是不负责任的。实现每个方法和每个类的程序员应该保证自己的代码是**自治**的。

如下代码:

```
@Override
public void run() {
   if (isFromFile) {
        if (hasQuery) {
            throw new RuntimeException("both --execute and --file specified
        }
        try {
            query = Files.toString(new File(clientOptions.file), UTF_8);
            hasQuery = true;
        }
        catch (IOException e) {
            throw new RuntimeException(format("Error reading from file %s: 
        }
    }
}
```

这样的代码确实谈不上优雅，然而足够充分的判断保证了代码的正确性与健壮性。我只能说，在满足了这两点的前提下，可以聪明地利用诸如防御式编程、Optional来规避多余的嵌套或分支，从而提高代码的可读性。

Effective Java总结了高质量代码的几个特征：清晰、正确、可用、健壮、灵活和可维护。我认为这个总结非常中肯。写代码真的不要太偏执，不分任何场景一味地追求代码的可读（清晰），一味地重申DRY，我觉得都是不负责任的态度。

或许是我老了的缘故，我变得不再理想主义；但更多的原因是因为我看到太多追求所谓“整洁代码”的程序，不去考虑复杂繁琐的异外情况带来程序的不健壮；因为去除重复带来的不必要间接影响了代码的简洁与干净，甚至影响了代码运行的性能。

整洁代码是必须的，但不是衡量代码质量的唯一标准！

2017-07-07 21:03 1570

< 剖析大数据平台的数据存储

败了，不要找理由，要找原因 >