

架构设计的悖论，复用是美好的还是邪恶的

原创 聂晓龙(率鸽) 阿里技术 2024年09月16日 08:30 浙江



微信扫一扫
关注该公众号



这是2024年的第69篇文章
(本文阅读时间：15分钟)

01
第二十二条军规

我一直有一个疑惑，「*Duplicated code fragment 10 lines long*」这是IntelliJ IDEA中对重复代码的标记。我们从学习面向对象开始就被告诫，*duplicated code is a bad smell*。但我真正进入复杂工程项目后，我发现往往让我栽跟头的，却是那些被各种所谓“复用”的“抽象”后的“*abstracted code*”。

软件工程设计，上至远古大神阿兰·图灵、冯·诺依曼，下至20世纪互联网高度发达的今天，诞生的各种框架与工具，本质上只在朝着2个方向在发展：可复用、可扩展，一对天然矛盾而又长期共存的2个方向。

“你的代码复用性不强”，“如果未来XXX你这里怎么支撑”，“能不能做到新场景0成本接入”，可复用的设计理念像军规一样督导我们编码和开发，可谁又知道它是不是那第二十二条。（*美国海军·第二十二条军规*）

02
中台视角看复用的魔力

提到中台就不得不提中台的鼻祖-芬兰游戏公司Super Cell。相传2015年，马云带领阿里巴巴集团高管，拜访了位于北欧的游戏公司Super Cell。该公司总员工数不到200人，但一年利润却有惊人的15亿美金，平均每人的产值高达750万美元。

Super Cell将游戏开发过程中公共和通用的游戏素材和算法整合起来，并积累了非常科学的研发工具和框架体系，构建了一个功能非常强大的中台。一套工具能支持所有团队，开发速度与工作效率都得到了质的提升。

2.1 坚实可靠的技术中台

受Super Cell中台模式的启发，阿里开始实行“大中台小前台”战略。全链路Eagleeye采集，高速稳定的HSF-RPC服务，极少配置就能实现水平扩展的高性能TDDL分库分表框架等等。问到从阿里出去到其他公司的小伙伴，对方公司的技术基建如何时，无不对阿里的技术中台竖大拇指。

中台战略在阿里的落地，至少从技术中台来看无疑是成功的。为技术同学提供了先进的武器装备，在通用工具建设上，提升了整个技术体系的研发效率。

2.2 饱受争议的业务中台



天然的不确定性

技术基建不可或缺，但作为驱动业务增长的技术团队，绝大多数时间一定都是投在业务项目上。这也是为什么业务中台难做，但还是一直在尝试的主要原因。2000亿的市场但凡啃下5%，那也是百亿级的规模。

曾经我的一位主管说过，业务技术团队比中间件技术团队挑战更高。纯技术团队解决的问题是确定性的，如何减少RT、如何提高吞吐，这一整年甚至好几年都只有这一个命题。但业务团队不一样，今天要打渗透，明天要做拉新，后天又有新打法出来。并且也没人能保证这样做了业务就一定会成功，业务是极具不确定性的。

对现实世界的抽象

任何一个业务中台的失败都可以归咎于“我们没抽象好，并非这条路不对”。如同盲人摸象，昨天我们收集的信息认为这是一堵墙，今天我们获取到的知识又觉得像是一堵墙加一根绳子，明天我们总结的经验又告诉我们，应该是一堵墙加一根绳子加四个柱子。我们可以不断接近真相，但每一次模型的升级却并非如我们自己知识的更新这般容易。

高昂的成本与未知的稳定性

在中台已有的扩展能力范围内，调整业务逻辑相对是容易的。但业务的特点就是不确定性，当业务的变化需要中台团队改变来做出适配时，往往这个合作过程会相当痛苦。业务团队想用简单的if解决这个业务场景的变化，中台团队则倾向于再抽象一层。

而抽象性与其本身的表达能力是向左的，抽象层次越高，表达能力越低，越难以理解。最终业务团队越来越难用好中台系统，而中台系统离业务也越来越远。

中台的抽象往往不是针对某一单一的业务场景，而是适配所有已接入的业务系统以及面向未来可能的扩展性。这会导致任何一处的改动，产生的影响范围都具有极高的认知成本。随着人员的不断更迭，现有系统的维护者，可能不认识系统里90%的代码，于是有那张图：「系统非常稳定，请不要随意改动代码」

// 03 //

软件工程的成本在哪里？

In software engineering, cost estimation is often a more difficult task than in other engineering disciplines due to the intangibility of software.
— Grady Booch 《Object–Oriented Analysis and Design with Applications》

译：在软件工程中，成本估算往往比其他工程学科更困难，因为软件是无形的。

复用本身最直接的目的就是降低成本，同样的事只用于一次，那软件工程的成本到底在哪里？Grady Booch在 *Object-Oriented Analysis and Design with Applications* 中表示「软件是无形的，软件的成本不易评估」。类似我们平常评估需求工时，到底是3人日还是还是3.5人日，其实很难有一个准确的测算。

3.1 确定性的开发成本

虽然成本无法准确测算，但我们依然给出了一个相对“确定”的工时人天，囊括从需求分析、方案设计、编码开发、协调测试、发布上线等各个流程。无论准确与否，我们也确实在按照这种模式运作。3人日或5人日，整体差异不会太大。往往我们也在追求，如何以最少的开发成本来支撑更多的业务需求，无论是在当下还是基于以后。但软件的成本，真的就是等同于开发成本吗？

3.2 无上限的维护成本



软件工程成本在于维护

Software maintenance consumes a significant portion of the total lifecycle cost, making it the most expensive phase of software engineering.
— L. Peter Deutsch Designing for Change: A Software Engineering Perspective.

译：软件维护消耗了总生命周期成本的很大一部分，使其成为软件工程中最昂贵的阶段。

一行代码当时写下去只花了1分钟，但只要它还在线上运行，我们就需要投入无形的维护成本。当年只是连了一下数据库查数据，后面数以万计的工程师加入到去ORACLE的大军中。软件工程最大的成本在于维护，你可以很轻松的在雷区里再加一个更精致的警示牌，你也很难把这篇雷区的雷都排光，甚至你都不知道到底还有哪些是雷区。

复用的代码更难维护

我们总是嘲笑有些代码写得像面条代码 *Spaghetti code*，缺乏设计缺乏美感，但当你被线上的问题搞得晕头转向时，往往这样的代码是高度“抽象”高度“复用”，运用各种设计模型、流程引擎的“高雅代码”。面条代码看似不够优雅，但它对系统的损害远比那些错误的抽象，低得多得多。

多数面条代码都有一个特征，那就是松耦合与隔离。你可以很清楚的知道改动这行代码，带来的变化是什么。但复用的代码正好与之相反，它拥有高度的复用性与收敛性，一行代码的调整，就可以把你想改的那8个场景都改过来，当然，还有你没想到的另外20个场景，它也改了。

认知负荷是高成本根因

曾经我在一家中小公司任职时，我们需要在公司现有商城系统基础上再做一个旅游类系统。当时我每想到一个设计方案，总被告知这样会影响到原有商城逻辑。于是我跟CTO反馈，能不能让我自己单干。CTO告诉我，推到重来总是最简单的，你以后会知道，以后你遇到的从0到1的事情会很少，绝大多数都是从10000到10001。

为什么大家觉得老系统难维护，因为99%的代码都是祖传代码 *Legacy Code*，你可以把警示牌做出花来，但不敢去往雷区伸一步，因为那些代表着未知 *unknown unknowns*。

以Github Copilot为代表的类copilot代码工具在确定性场景下表现优秀，但它并没有在本质上帮我们提升生产力。如果一个需求评审完你就知道怎么做，那成本一般不会太高，但如果你得到的是一堆不确定性的改动-“逻辑和XX一样”，这种看似“清晰”实则完全不确定的需求，才是成本黑洞。

// 04 //

重复代码是可耻的吗？

| 4.1 真重复代码与意外重复代码

两段代码可能非常相似甚至一模一样，但将出于截然不同的原因而被修改，我们称之为意外重复，而意外重复不应该被消除。应该允许重复持续存在，随着需求的变化，两段代码各自演化，这种重复将被消除。

— Robert C.Martin 《Clean Craftmanship》

在定义重复代码前，我们需要先区分真实重复与意外重复。Robert C.Martin在 *Clean Craftmanship* 中阐述，如果因为某些意外，碰巧导致它们代码一样，这种重复应该被允许。并随着需求变化，它们的重复会逐渐消除。

其实这种理念类似我们在创建用户时，我们需要让前端传递UserDTO，我们保存数据库时，传给ORM框架是UserDO，它们的代码可能一模一样，但我们一般不会将两个对象合成一个对象，虽然当下代码是一样的。

那如何鉴别真重复代码与意外重复代码，本质上这取决于代码想表达的意图。意外重复的两段代码所表达的意图是不一样，虽然他们目前的实现是一致；而真实重复的代码意图是一致，当某种意图发生变化时，他们需要同步变化。

| 4.2 DRY是一种应该谨慎使用的哲学

God, grant me the serenity to accept things that I cannot change. Grant me courage to change the things I can, and wisdom to know the difference.

— Reinhold Niebuhr Serenity Prayer

译：请赐予我平静，去接受我无法改变的。给予我勇气，去改变我能改变的。赐我智慧，分辨这两者的区别。

如果是真实重复代码，那一定需要进行避免吗？从理论上回答-是，但理论和实际往往有条鸿沟。美国神学家 Reinhold Niebuhr 在 Serenity Prayer 中写下了这段经典的祈祷文「赐我智慧，分辨这两者的区别」。其实我们也一样，有时我们并不能完全确定它到底是不是真实重复。

Don't Repeat Yourself

DRY原则出自 Andrew Hunt *The Pragmatic Programmer*，指在程序设计以及计算中避免重复代码，因为这样会降低灵活性、简洁性，并且可能导致代码之间的矛盾。系统的每一个功能都应该有唯一的实现，如果多次遇到同样的问题，就应该抽象出一个共同的解决方法，不要重复开发同样的功能。

如果我们真的能确定它确实确实是一段唯一的代码块，我们可以这样去做消除重复。过早优化是万恶之源-premature optimization is the root of all evil，有时我们对未来是缺少预知的。所以我们需要更谨慎的对待它，**DRY**是一种应该被谨慎使用的哲学。

You Aren't Gonna Need It

YAGNI原则出自 Ron Jeffries *Extreme programming*，指的是你自以为有用的功能，实际上都是用不到的。相比把DRY奉为圣经，我更倾向YAGNI原则。很多时候问题并没有那么复杂，但被YY出来的所谓“面向未来的扩展”加持后，最终变成了一坨精美的粑粑~

Rule Of Three

Rule Of Three出自 Martin Flower 大名鼎鼎的 *Refactoring*，第一次做某件事时只管去做；第二次做类似的事会产生反感，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。

"三次原则"可以看作是DRY原则和YAGNI原则的折衷，是提前优化和代码冗余的平衡点，可以比较好的在编码开发中指导我们工作，值得我们在"抽象化"时遵循。

// 05 //

我眼中的卓越工程

What experience and history teach is this: that nations and governments have never learned anything from history or acted upon any lessons they might have drawn from it.

— Friedrich Hegel The Philosophy of History.

译：人类从历史中学到的唯一的教训，就是没有从历史中吸取到任何教训。

卓越工程倡导的今天，也说明前人留给我们的并不是一个卓越工程。20年的风沙淘尽一代又一代的研发人，大家已经不会再关注当初这个需求花了多少人日，而是如今我们需要多少成本去维护它。

我们作为后人的“前人”，用着3人日的开发成本留下数不尽的维护成本给后人，那我们也很难叫做在践行卓越工程。如果当初花了更多的开发成本，但换来了后续更清晰的代码表达、更直观的业务逻辑、更简单的迭代维护。那它，就是卓越工程。

// 06 //

最后一点点思考

错误的抽象、错误的代码复用，所引发的复杂性无限蔓延，对系统的危害比面条代码强大一百倍。复用与扩展，业务与技术，到底哪些该复用哪些不该复用，好像变成了一个哲学问题。如果说“正确的抽象”是一个100分的美丽乌托邦，那面向复杂性隔离的整洁架构，会不会是一个稳定的80分。面向复杂性隔离的整洁架构，我好像有了一些新的想法... To Be Continue

参考链接

- [01] 第二十二条军规
<https://baike.baidu.com/item/%E7%AC%AC%E4%BA%8C%E5%8D%81%E4%BA%8C%E6%9D%A1%E5%86%9B%E8%A7%84/32770>
- [02] supercell
<https://baike.baidu.com/item/supercell/19452566>
- [03] Legacy Code
<https://enkonix.com/blog/legacy-code/>
- [04] Clean Craftmanship
<https://www.goodreads.com/book/show/56631473-clean-craftsmanship>
- [05] Refactoring
<https://refactoring.com/>



 微信公众号

 InfoQ

 CSDN

 今日头条

 微博

 哔哩哔哩

 抖音

 微信视频号

 钉钉视频号



欢迎搜索： 阿里技术



分享

欢迎留言一起参与讨论~