9/25/2019

[关于 'Worse is better' 的总结与看法]

[homework1]

[林晨]
[17302010021]

1

目录

1.1 文本总结.....3

1.2 个人理解.....5

（1） Worse is better 在大型软件系统设计中可能具有普适型..... 5

（2） Worse is better 的命题最初可能并没有扩展至整个软件领域的想法... 6

（3） Worse is better 注重软件设计的简单性..... 6

（4） Worse is better 并没有完全抛弃系统的正确性、一致性和完整性..... 6

（5） 市场环境下的 Worse is better.....6

2.1 教材第一章基本内容.....8

2.1.1 设计原则..... 8

2.1.2 知识概要..... 9

a.....9

b..... 9

c.....9

d..... 10

e..... 10

3.1 联系.....10

3.2 区别.....11

4.1 我对通用软件设计原则的看法.....11

Part 1 如何理解'Worse is better'

1.1 文本总结

被称为"越差越好"的概念认为，在软件制作中，最好从最小的创建开始并根据需要进行扩展。最初提出这个概念的是一家 Lisp 公司老板 Richard P. Gabriel，面对 Lisp 做 AI 的困境，他思考为什么人们认为 C 和 Unix 要比 Lisp 好。

Lisp 是一种表处理语言，用表结构来处理问题，似乎应该广受推崇。1980 年到 1991 年 Lisp 语言从无标准化逐渐发展，性能提升，能够与其他语言和软件更好的集成。而早期的 Lisp 公司面对着 AI 公司和市场的多重压力。从 1991 年从 Richard P. Gabriel 的《Lisp: Good News, Bad News, How to Win Big》以及《Worse is better》这两篇文章中我们可以看出，作者本人对于"Is worse better?"这个问题并不能给出明确的回答，甚至存在矛盾的看法。但是有一个有趣的例子可以给我们一些启示：

MIT 和 New Jersey 方法分别采取不同的方式来实现操作系统。使用 MIT 方法的人认为应该要完全保证程序的正确性（例如 PC 失败的问题），大致可以按照**正确性=一致性>完整性**来表征，而简单性被模糊地表述为：简单性-设计必须在实现和界面上都简单。而当简单性和正确性和一致性冲突时，往往被两者掩盖而变得复杂。而另一方则表示简单性比正确性更加重要，大致可以认为是**简单性>正确性>一致性>完整性**。前者把简单、准确和一致当成硬性要求，后者则明确指明相互冲突时的选择。两者的不同态度带来了不同的结果。无论是 1991 年还是如今 2019 年，我们都不能否认 C 与 Unix（采用 New Jersey 一方）取得了胜利。

作者对于这种现象进行了说明。MIT 方法选择做正确的事情，要求实现几乎 100%的功能，而要实现全部功能，往往意味着一个庞大复杂的系统。花费巨大精力后，可能发现系统只能运行在一个同样极其复杂的硬件设备上。而 New Jersey 方法可以选择一个相对简单的实现，即使不能完全正确，其高效的性能和便捷的使用将会很快吸引人们的使用，在此基础上获得反馈，弥补不足（例如从 50%弥补到 90%）。在这种方法中，一个最终完全正确的方案可能并不存在，而只是不断接近更高的正确率百分比。[1]

[1] 《Lisp: Good News, Bad News, How to Win Big》，Richard P. Gabriel，1991

但矛盾的 Richard P. Gabriel 在 1991-1992 年又写了一篇叫做《Worse is better is worse》的文章。这篇文章以笔名 Nickieben Bourbaki 的身份从九个方面反驳了”Worse is better”的论述，并且认为他（其实就是自己）的文章腐蚀年轻人的思想，给他们上了不好的一课。其九个反驳理由大致如下：[2]

①不存在 Worse is better 的哲学，有些程序由于设计过于天马行空或者硬件限制，不能不显示出“越差越好”的假象。

②没有人能够实现正在正确的设计，我们只能在设计与实现中做权衡，因而不存在纯粹的例子来证明。

③在 PC 失效的例子中，正确的解决方案是漫长的是可能的。而 Unix 采用让用户自己解决问题的方式来实现并不是更好的方案，而且带来了不一致性。

④Unix 是 PDP-11 的 RT-11 的替代品，而在 RT-11 却比 Unix 差得多。

⑤Lisp 被用于 AI 等研究领域，大部分计算机不使用 Lisp 语言，因而该结论没有普适型。

⑥电脑主机的作用被忽略了。不同的电脑拥有不同的性能和价格，因而不能一概而论。

⑦Unix 和 C 都没有对应使用领域的竞争对手。

⑧C 语言不是一种更差的语言:它拥有所有常见的数据类型、现代控制结构并且有漫长的发展。

⑨从来没有真正的、重要的、“越坏越好”的对峙，没有实际的例子能真正证明一者强于另一者。

1992 年作者又发表了《Is worse really better》，以 C++为例，把它的良好性能和面向对象的结合作为反驳“越差越好”的案例。[3]

2000 年，Richard 启动了一轮自问自答，先写了《Back to the Future: Is Worse (Still) Better?》，又写了《Back to the Future: Worse (Still) is Better!》。（我不知道他这样会不会赚双份稿费？）在后一篇中作者做了如下解释：想 C，C++和汇编这样相对低级的语言相对于模块化的 Lisp 和 Smalltalk 对于微小的算法、数据结构的组织和

[2] 《Worse is Better is Worse》，Nickieben Bourbaki，1991-1992

[3] 《Is Worse Really Better?》，Richard P. Gabriel，1992

一些细节更具有好处。模块需要关注大小、性能、可用性，而在高级些的语言中这些的控制不容易。[4][5]

1.2 个人理解

人们将”Worse is better”总结为“简单之美”。原因在于其中涉及的原则把简单性放在首位。结合 Richard 的几篇文章，个人认为可能”Worse is better”在软件系统设计中具有普适型，但是并非适用于任意情形和领域。

正如 Richard 在自己的矛盾一样，C++比 C 在平衡性能和面向对象上更好，而这个过程确实进行了改进。有 Lisp 引出的结论也可能不是一个通用的案例。可以知道，Richard 的报告并没有完整形成一个体系化的想法，在我个人看来确实有点片面。

正如作者自身论述的九个反驳理由，Worse is better 无论在语义上还是实际操作者都有些难以让人信服。但是它的合理性在于引发人们针对一些情况下 Worse is better 案例的思考。

（1）Worse is better 在大型软件系统设计中可能具有普适型

我在阅读老师提供的文章时也在思考，真的是越差的就是越好的吗？我们学习高效的算法，某方面能力比较强的语言（C，C++，Java 等）这些难道并不是越加分析讨论越发进步的吗？于是，我打算把自己从一个更大的视角脱离出来。“Worse is better”这个概念可能要被限制在软件制作中，尤其是大型软件制作中。小型的软件系统本身可能就不具备设计的复杂性，或许可以绕开“Worse is better”的围栏。大型的软件系统，比如 Unix，本身需要实现很多复杂的功能，如果仍然一味追求完整性】一致性、正确性，你们期简单性可能深受影响，实现也会变得异常复杂。在某些领域可能也有类似的现象，但却不能将它随意扩展开来。

在作者自我反驳的 9 个理由中，第 4 个理由可能也可以看做一个论证--Unix 是 PDP-11 的 RT-11 的替代品，而在 RT-11 却比 Unix 差得多。然而我们是否可以把 RT-11 看成是“Worse is better”实践的第一步呢？随着用户被吸引使用，人们发现问题需要弥补，再不断以尽量简单的方式加以改进。而其他几个理由似乎也没有从根本上反驳“Worse is better”（至少没有让我信服）。

---

[4] 《Back to the Future: Is Worse (Still) Better?》，Richard P. Gabriel，2000

[5] 《Back to the Future: Worse (Still) is Better!》，Richard P. Gabriel，2000

（2）Worse is better 的命题最初可能并没有扩展至整个软件领域的想法

作者在《The risk of worse is better》中写道：“A wrong lesson is to take the parable literally and to conclude that C is the right vehicle for AI software. The 50% solution has to be basically right, and in this case it isn't.But, one can conclude only that the Lisp community needs to seriously rethink its position on Lisp design. I will say more about this later....”[6]

尽管我们现在通常以软件开发的世界理解这个说法，但是上面一段话我们可以看出，最初这个说法只是为了给 Lisp 相关的 AI 软件一些参考，更不用说更多的其他领域。即使我们把这种说法放在软件制作领域，也应该视情况而定。

（3）Worse is better 注重软件设计的简单性

大型软件设计可能有需要实现多种多样的功能，这些功能全部都有精准的实现可能会导致整个软件系统成为一个混乱复杂的“大泥球”。这一点我们在软件工程课程学习过，即软件的设计要遵循尽量简单的原则，否则可能会形成“泥球不断翻滚变大”的效应。正如作者的 Lisp 公司在 20 世纪遇到的困境一样，一开始处处都考虑（即一开始就作出正确的设计）可能并不是一个好主意。

（4）Worse is better 并没有完全抛弃系统的正确性、一致性和完整性

简单性是大型系统设计的首要原则，但是不能抛开其他谈简单，这样会使得系统的实现没有意义。简单性依然要保证一些基础情形的正确，使用作者原文的数据，即要超过 50%（但这只是一个假设）。在尽量保证简单性和准确性下，依旧要适当考虑一致性、完整性。所以，“Worse is better”并没有将这些系统完全与“简单性”挂钩。

（5）市场环境下的 Worse is better

在软件开发市场环境下，一种叫“万福玛丽亚”模式，就是，当整个作品完全出来时，再隆重地推向市场。还有一种是，“worse is better”，即快速拿出原型，听取用户意见，然后继续增加功能，修复 bug。前者与计划模型相似，后者与基于增量的开发模型相似。在市场环境下，要抢占市场，时机很重要，掌握用户的使用韧性有必要。这种情况下，后者在快速提高一个原型后便抢占了市场，用

[6]《The Rise of Worse is Better》，Richard P. Gabriel，1989

户习惯于使用当前原型的方式，而不断的更新迭代在每次过程中并不会大幅度影响用户的个人感受。

无疑，交出完整作品的方式在市场环境下比较难以脱颖而出。（在非市场环境下也不一定好用）在市场环境下，Worse is better 的方式确实是有着优势。

Part 2 The content of chapter 1

2.1 教材第一章基本内容

2.1.1 设计原则

①避免过度通用。很多时候，我们追求让人家设计能够满足尽可能多的场合，追求通用，这样可能反而使得对于任何情形人家使用起来都很不便。
②不断升级的复杂性。随着需求数量的增加，人家设计的主观复杂性是呈现指数增长的。复杂性来源于许多方面，其中，通用性会带来复杂性，因而需要做好设计平衡。
③收益递减率。一旦人家系统设计某方面的优势已经较为明显，要进一步改善它便会越发不容易。用课本的话说--某种好处获得的越多，下次获得这种好处所需付出的努力越大。
④不变的基础。修改模块相比修改模块化更容易。接口一旦被另一个模块使用，修改接口至少要替换两个以上的模块，而修改接口对于包装的模块来说对外是不可见其修改了，所需的改动也较小。
⑤鲁棒性原则。宽于输入，严于输出。这一点很重要。对于输入的数据要尽可能解析，对应的不合法数据，模块应该进行相应的处理与提示。这样，外部使用者便可以更加方便地使用模块。
⑥安全边界原则。课本中描述为：原理悬崖，以防坠落。安全边界原则对应着“震荡模式”和“生产模式”两者模式，在前者中，需要严格检查输入，拒绝任何不符合规定的数据。而后者则对所有输入进行处理，对于不合法数据进行相应处理返回。
⑦利用间接将模块去耦合。在模块与模块进行连接的过程中可以加入一个中间层来解耦，一旦模块需要改动，所需要的其余变动也会减少。所以课本认为：“间接支持替换”。
⑧扩展不相称原则。一个全新的设计在交付使用过程中通常会经历调整。扩展带来其他方面要求的提高--不相称则需要相应的设计调整弥补，最终可能整个方案都需要重新设计。
⑨面向迭代的设计。这类似于面向增量的工程方法。通常我们可以认为设计不可能一次性成功，所以最后一开始就设计使得它易于修改。
⑩不断挖掘。在组织内，存在坏消息被封锁，好消息迅速传播的现象。复杂的系统如果需要修正错误，此时已经很难入手。由第④条

原则可知，此时的修改模块化设计很多模块，技术人员可能也不愿意返工。

(11) **全面实施简单性。**这与“Worse is better”的说法有异曲同工之处。简单的设计易于改动、易于理解，设计师对于系统能够有全面的把握。保持简单性是控制复杂的最大的希望之一。

2.1.2 知识概要

a.

突生属性：指系统的单个组件中并不明显，但吧多个组件组合起来就会暴露出来的属性。

传播效应：复杂系统中，随着分析的深入，许多隐蔽的，难以察觉的效应会逐渐浮出水面。

扩展不对称：随着系统规模和速度的提升，系统各个组成部分不一定遵守相同的缩放规则，从而引发问题。

简单来说，上述三种现象就是说，在复杂的系统中**小问题会变成大问题，小扩展会变成大改动。**

b.

系统：互连在一起的组件的集合，在环境中通过其接口表现出预期的行为。

组件的选取需要根据**目的和粒度**。

目的不同对系统的认识就不同，就会把不同部分看出组件。课本举了飞行器的例子--看成飞行器（飞行器：机身、机翼、控制台、引擎；环境：地球引力、引擎推力、空气阻力），看成交通工具（看团购价：座椅、空乘人员、空调系统、厨房；环境：所有乘客）...

粒度不同也会大大影响对系统组件的确定。粒度可以理解为宏观/微观这样的视角。引擎制造商以引擎为系统，而飞机设计师以飞机整体为系统。

系统分析之前首先要**明确分析系统的角度**，包括如何划分组件、组件划分的**粒度、系统边界、已经系统与外部环境的接口**。

c.

复杂性：其标志有组件数量大、组件间互连数量大、不规则情况多、描述冗长、设计实习和维护团队这五个方面。复杂性主要来源于**需求的数量和高利用率。**其中需求的数量可能来源于供应商、通用性、扩展不对称、需求变化。

处理复杂性：模块化、抽象化、层次化、分级化。
模块化：由课本推算可知，将一个系统分成 K 个模块，可以将调试时间减少约 K 倍。即对于 N 个错误，每个模块出现为 N/K，，调试时间为（N/K）^{2}/K 的调试时间。
抽象化：根据相对分割的，交互较少地分割来实现模块间传播效应的减少。这一就可以将模块看成一个整体而不必关系细节。
层次化：新的完整功能基于低层实现。
分级化：减少模块间的互联。

d.
计算机系统飞复杂性不受物理定律限制。计算机技术的发展速度与时间成指数关系。
计算机系统的发展速度前所未有的快。与此而来的，最原始简单的方案往往是正确的选择。

e.
减少复杂性的法宝：
①面向迭代。首先构建一个实现部分功能的、简单的、可工作的系统，然后不断改进，逐渐实现所有功能。
②保持简单。这是处理复杂性最有效但最难实施的方法。

Part 3 Connections between two reading materials

3.1 联系

①都强调简单性。在 Richard 的主张“Worse is better”的文章中提到参与 Unix 制作的工程师在设计 Unix 的过程中关注简单性。“这位主张 New Jersey 的人说，Unix 解决方案是正确的，因为 Unix 的设计理念很简单，而正确的事情太复杂了。”同样，课本在分析复杂性的种种来源和标志后，也强调解决复杂性最大的希望之一是设计的尽量简单。
②都涉及迭代开发的思想。Richard 认为，“正确的事情需要永远设计，但是在进行过程中的每个环节都非常小。要使其快速运行是不可能的，或者超出大多数实现者的能力。”因而，他推崇通过首先开发一个功能超过 50%的系统，放入使用后不断实现其他设计。课本则提到迭代式开发，以增量的形式弥补错误、满足新需求、修改需求或者适应市场。

3.2 区别

①课本材料还涉及模块化、抽象化、分级化和层次化来控制复杂度。而 Richard 的材料不关注这些。

Part 4 My opinions

4.1 我对通用软件设计原则的看法

① **设计简单目前仍很重要。**从计算机发展的先验来看，设计简单对于一个软件系统来说确实至关重要。目前，我们仍然没有更好的方式和机制来实现不需要设计师直接控制简单性就能带来良好系统设计。

② **学会舍弃。**简单性说起来很容易，但是如果暂时舍弃一些实现或者忽略一些暂时的错误，对我们来说可能是一个难事，因为它像一个疙瘩一样让人难受，但是我们起初需要适应，在形成简单且合理的架构设计后继续迭代地完善。

③ **避免通用。**这也是我们的一些通病。老是希望自己写的方法或者函数尽量适用于任何情况。在一些情况下这是为了宽于输入，而另一些情况则可能只是“觉得”通用比较好，而在实际的系统中可能并不需要，此时就有改善的空间。

④ **合理的模块化。**模块划分有助于我们查找错误。但是不合理的划分可能会使得系统设计变得复杂，模块的抽象封装差，相应的耦合比较强。