**软件哲学**

软件更像是一门艺术，没有什么特别量化的评价标准。大家怎么看呢？

1. 变化之本：

如果不关注变化呢？（可能会项目失败；失去占领市场的先机；。。。。。）

为什么会变化？变化不外乎下面几种（外因和内因）：

1. 开始的时候与客户沟通需求不彻底，没有完全弄清楚软件的需求和目标；
2. 客户自己开始也不清楚自己到底想要做什么；
3. 客户在软件开发周期中，要临时修改某些已有需求或增加新的需求；
4. 就目前的需求实现有难度，想变通一下；
5. 刚开始的系统设计在软件迭代过程中需要局部变化；
6. 从设计的角度来进行重构；

Tips：

1. 设计之初就应该考虑到将来的变化，也就是说系统框架要设计的比较有弹性（就像AWS的很多服务那样弹），对于将来的需求的改动不会付出很高代价；
2. 面向变化编程：永远不要假设需求不变，现实中需求大大小小经常变。
3. 变化是创新的必经之路，永恒不变的东西只有变化。

常犯的错误：

硬编码？

僵尸设计？

仅仅满足当前需求？

成本约束所以放弃？

1. 权衡之道：

现实生活中我们也需要经常权衡某件事情的利弊，做软件同样如此。

天下没有免费的午餐？

软件有很多的品质：性能（时间性能，空间性能），可用性，易用性，可扩展性，兼容性，可靠性，安全性，健壮性，可维护性。。。。。。。

这些特性经常是不可兼得的：性能与可维护性经常有抵触？安全性与性能呢？

Tips：

1. 细致的评估这些软件品质，尤其是客户关注的更细粒度的特性（比如吞吐量？）。然后根据评估结果做出取舍（一定要结合实际情况，不要盲目自大的做决定）；
2. 要明确哪个/哪些品质是客户和我们自己最看重的，在软件工程的整个生命周期中一定要把该品质作为首要目标。

常犯的错误：

无底线的追求效率？----现在性能很多情况下都不是作为首位来考虑。

为了炫耀而华而不实的设计？----真的有必要吗？

为了愚弄上司？

权衡了两个压根不需要权衡的软件品质？---也就是说在当前这个项目中，这两个软件品质本来可以同时满足，但是实施人员错误的为了实现一个而牺牲掉另一个。

1. 简单之美：

大家都怎么理解简单？精简短小？简朴？重复的简单？可读性好？

简单的标准可能都不一样，但是一个通用的规则是您在设计及实施软件的时候尽量多想着点别人。也就是说尽量不要杀鸡用牛刀，尽量想着以后可能需要别人给您收拾残局。

Tips：

1. 不要过度设计：有些人喜欢用设计模式来体现自己的能力，经常事与愿违，反而会弄得设计乌七八糟，成为反模式的典型（注：有本书是关于反模式的，是软件系列丛书的bible之一，大家有兴趣可以阅读）。
2. 切勿简单问题复杂化：很多人在学习了一些新的技术（尤其是设计模式）都想在实践中用一下，而不管这技术是否对这个项目来说是否真的最适合。
3. 尽量化繁为简：把复杂的问题分解为简单的多个小问题，然后分别来处理。这是软件工程中用的比较多的分治归并思想。代码量线性增加会导致复杂性以几何级数增加，因此尽量把关注点的范围缩小。
4. 简单易读的代码对于后续人员的维护以及后一个版本的开发，甚至重构都是大有好处的。

试想，您写的代码绝大部分人都看不懂，以后代码出现bug可怎么是好？甚至，可能您在一年后都不知道自己当初写的代码是哪个意思。

常犯的错误：

为了偷懒，拷贝粘贴的代码？

废话太多的实现？

花哨的术语以及新潮的技术，您自己真的搞懂了吗?

反正不是我家的东西，能运行就行？

1. 把握好“度”：

很多东西/事物都有尺度？有些时候我们喜欢尺度大的，有时候喜欢小的。软件也一样，也有尺度的问题。

虽然很多东西都有度，但是软件的度还真是不好度量。您没有办法说一个项目用某种语言开发要把代码量限制在多少以内；问题的切分粒度多少为好；软件分层多少层合适。。。。。。。。。

Tips：

1）“过犹不及“在任何时候对软件都适用：问题切分的越小越好吗？太小的粒度反而会增加操作的复杂性，因为关注点太多了 。可以根据语义结合规模来切分，比如从语义上来看，哪些内容是调用方关心的，哪些是它不需要也不应该关心的。我们就可以根据这些信息粗略来划分模块，而对于逻辑复杂的模块在进一步细分。

2） 多少算合适呢？（还真有一个经验值）

比如做一个中间件项目，所谓中间件就是说他是一个大的夹层，它需要提供API供它的上层调用，同时它也要调用它下层的API。因此您准备把这个中间件的实现分为几层呢？

如果要想把该中间件运行在不同的OS上（类unix，windows）？

1. 相对来说，尺度小容易变大，但是尺度大变小就不容易了。为什么呢？软件角度来说，如果您暴露给外面的太多，外面可能对您的依赖就越大，耦合度也越紧，想要撇清关系太难了。这里就体现出软件工程中谈到的封装性的作用了。

常犯的错误：

为了简单，把实现做得很糙？（比如用一个互斥锁对一段很长的代码做保护，其实真正的需要互斥访问的临界区就几行代码------要理解互斥访问的真正的原因是数据被共享了而不是代码）

行政干预把工作切分很细，以至于光是互相沟通的代码就很大？互相“推球”？

1. 对症下药：

生病了我们都要去看病，去看对应的科室。以前听过一个真人真事，中国某医院的手术大夫把需要做阑尾炎的病人给人切除了一个肾。很荒唐是吗？软件中比这个荒唐的事情太多了，只是我们经常会忽略这个。

一群人讨论问题，客户提出了问题A，其他人讨论了一个上午，但是争论的是问题B。（客户应该投诉他们！）

Tips：

1. 首先找到症状是什么？客户最关心的问题是什么？

不要因为看小病耽误了看大病；故意绕开问题更不可取。

1. 症状对应的药方都有哪些？

您有哪些药可能会对该症状有用？

1. 下药前的思考？

比如一个应用程序调用了一个商用很久的第三方的API，而该第三方实现也是基于GLIBC库的。然后应用程序出了某疑难问题比如内存泄漏，那么您会怀疑谁（应用程序，商用第三方，glibc）更可能有问题呢？

一般来讲，更底层的和商用的软件经历了更多的测试和时间的洗礼，出问题的可能性要小，您应该尽量先考虑您自己写的程序的问题。

1. 您的药下的是否合理？

不要被症状的表面所迷惑，尽量深入进去，用逆向工程（这里是用反向推导）的方法来测试自己的想法是否靠谱。反向推导出可能的结果后，在正向来分析是否可能会出这样的结果。

常犯的错误：

尽信书？

权威说的就是对的？

怀疑一切？（那还混个什么）

1. 知其所以然：

我们经常是知其然，但是所以然知道了吗？可能有些感兴趣的东西我们会去了解其深层的原因，但是软件的问题我们去研究了吗？

软件是枯燥的，很多时候我们都是作为谋生的手段来应付之。因此不知所以然也就很正常了。但是如果您是要真正的学习东西，最好还是”再深一点“

Tips：

1. 想清楚问题的来龙去脉？您希望是知识面更广一点，还是更深一点？（在一个人的精力范围内，二者很难兼得）
2. 您经常google吗？google的答案您相信吗？
3. 您会浪费时间对牛弹琴吗？
4. 我们要加强口活还是思维能力呢？
5. 做永远比说要难？
6. 说自己不会/不知道有那么难吗？

常犯的错误：

经验说？（没有用某个部位思考把？）

跟风论？

本本主义？

假设错误不会发生？假设是安全的？

绝对论？

1. 上下文环境：

软件中经常有某些术语达到了疯狂被使用的地步：会话（session），事务（transaction），系统，平台。大家在平时听到这些词的时候害怕吗？我是经常害怕，因为这些词能用在很多不同的语境中及上下文环境。

Tips：

1. 任何问题都是在固定的场景下发生的，我们难道不应该搞清楚吗？
2. 您都清楚这个问题都有哪些可能的上下文环境吗？
3. 时序的问题经常都是难搞的问题，不同的上下文出来的结果会有很大不同？
4. 为了减少沟通的误解，尽量把上下文环境一起商量好？

常犯的错误：

根本是在不同的层面讨论问题？

我这里可以的啊，您为什么不行？不行重启把？

时髦论？

生产环境与测试环境？