第02课：深入分析软件的复杂度

软件复杂度的成因

Eric Evans 的经典著作《领域驱动设计》的副标题为“软件核心复杂性应对之道”，这说明了 Eric 对领域驱动设计的定位就是**应对软件开发的复杂度**。Eric 甚至认为：“领域驱动设计只有应用在大型项目上才能产生最大的收益”。他通过 Smart UI 反模式逆向地说明了在软件设计与开发过程中如果出现了如下问题，就应该考虑运用领域驱动设计：

- 没有对行为的重用，也没有对业务问题的抽象，每当操作用到业务规则时，都要重复这些业务规则。
- 快速的原型建立和迭代很快会达到其极限，因为抽象的缺乏限制了重构的选择。
- 复杂的功能很快会让你无所适从，所以程序的扩展只能是增加简单的应用模块，没有很好的办法来实现更丰富的功能。

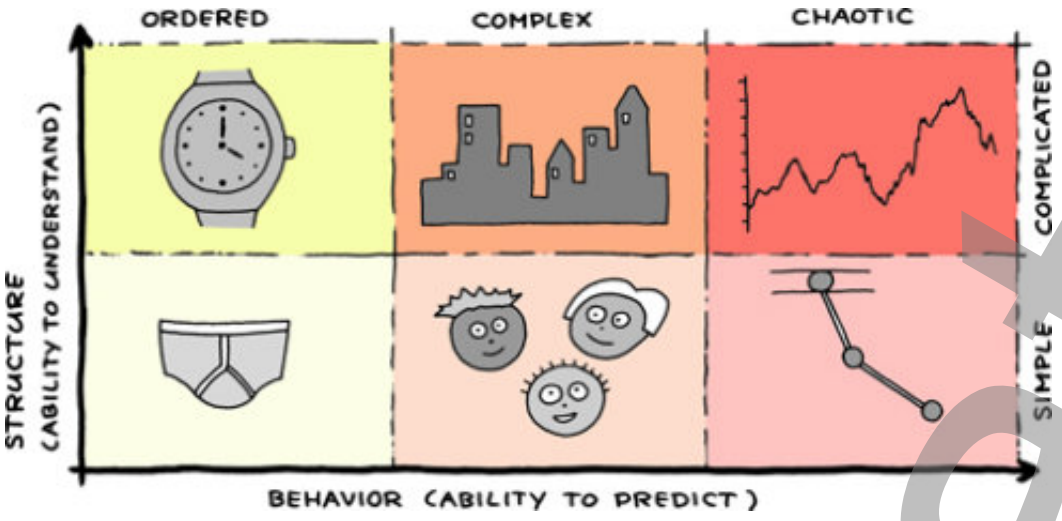
因此，选择领域驱动设计，就是要与软件系统的复杂作一番殊死拼搏，以降低软件复杂度为己任。那么，什么才是复杂呢?

什么是复杂

即使是研究复杂系统的专家，如《复杂》一书的作者 Melanie Mitchell，都认为复杂没有一个明确得到公认的定义。不过，Melanie Mitchell 在接受 Ubiquity 杂志专访时，还是“勉为其难”地给出了一个通俗的复杂系统定义：由大量相互作用的部分组成的系统，与整个系统比起来，这些组成部分相对简单，没有中央控制，组成部分之间也没有全局性的通讯，并且组成部分的相互作用导致了复杂行为。

这个定义庶几可以表达软件复杂度的特征。定义中的组成部分对于软件系统来说，就是我所谓的“设计单元”，基于粒度的不同可以是函数、对象、模块、组件和服务。这些设计单元相对简单，然而彼此之间的相互作用却导致了软件系统的复杂行为。

Jurgen Appelo 从理解力与预测能力两个维度分析了复杂系统理论，这两个维度又各自分为不同的复杂层次，其中，理解力维度分为 Simple 与 Comlicated 两个层次，预测能力维度则分为 Ordered、Complex 与 Chaotic 三个层次，如下图所示：



参考复杂的含义，Complicated 与 Simple（简单）相对，意指**非常难以理解**，而 Complex 则介于 Ordered（有序的）与 Chaotic（混沌的）之间，认为**在某种程度上可以预测，但会有很多出乎意料的事情发生**。显然，对于大多数软件系统而言，系统的功能都是难以理解的；在对未来需求变化的把控上，虽然我们可以遵循一些设计原则来应对可能的变化，但未来的不可预测性使得软件系统的演进仍然存在不可预测的风险。因此，软件系统的所谓“**复杂**”其实覆盖了 Complicated 与 Complex 两个方面。要理解软件复杂度的成因，就应该结合**理解力**与**预测能力**这两个因素来帮助我们思考。

理解力

在软件系统中，是什么阻碍了开发人员对它的理解？想象团队招入一位新人，就像一位游客来到了一座陌生的城市，他是否会迷失在阡陌交错的城市交通体系中，不辨方向？倘若这座城市实则是乡野郊外的一座村落，不过只有房屋数间，一条街道连通城市的两头，还会疑生出迷失之感吗？

因而，影响理解力的第一要素是**规模**。

规模

软件的需求决定了系统的规模。当需求呈现线性增长的趋势时，为了实现这些功能，软件规模也会以近似的速度增长。由于需求不可能做到完全独立，导致出现相互影响相互依赖的关系，修改一处就会牵一发而动全身。就好似城市的一条道路因为施工需要临时关闭，此路不通，通行的车辆只能改道绕行，这又导致了其他原本已经饱和的道路，因为涌入更多车辆，超出道路的负载从而变得更加拥堵，这种拥堵现象又会顺势向这些道路的其他分叉道路蔓延，形成一种辐射效应的拥堵现象。

软件开发的拥堵现象或许更严重：

- 函数存在副作用，调用时可能对函数的结果作了隐含的假设；
- 类的职责繁多，不敢轻易修改，因为不知这种变化会影响到哪些模块；
- 热点代码被频繁变更，职责被包裹了一层又一层，没有清晰的边界；
- 在系统某个角落，隐藏着伺机而动的 bug，当诱发条件具备时，则会让整条调用链瘫痪；
- 不同的业务场景包含了不同的例外场景，每种例外场景的处理方式都各不相同；
- 同步处理与异步处理代码纠缠在一起，不可预知程序执行的顺序。

当需求增多时，软件系统的规模也会增大，且这种增长趋势并非线性增长，会更加陡峭。倘若需求还产生了事先未曾预料到的变化，我们又没有足够的风险应对措施，在时间紧迫的情况下，难免会对设计做出妥协，头疼医头、脚疼医脚，在系统的各个地方打上补丁，从而欠下技术债（Technical Debt）。当技术债务越欠越多，累计到某个临界点时，就会由量变引起质变，整个软件系统的复杂度达到巅峰，步入衰亡的老年期，成为“可怕”的遗留系统。正如饲养场的“奶牛规则”：奶牛逐渐衰老，最终无奶可挤；然而与此同时，饲养成本却在上升。

结构

不知大家是否去过迷宫？相似而回旋繁复的结构使得本来封闭狭小的空间被魔法般地扩展为一个无限的空间，变得无穷大，仿佛这空间被安置了一个循环，倘若没有找到正确的退出条件，循环就会无休无止，永远无法退出。许多规模较小却格外复杂的软件系统，就好似这样的一座迷宫。

此时，**结构**成了决定系统复杂度的关键因素。

结构之所以变得复杂，在多数情况下还是因为系统的质量属性决定的。例如，我们需要满足高性能、高并发的需求，就需要考虑在系统中引入缓存、并行处理、CDN、异步消息以及支持分区的可伸缩结构。倘若我们需要支持对海量数据的高效分析，就得考虑这些海量数据该如何分布存储，并如何有效地利用各个节点的内存与 CPU 资源执行运算。

从系统结构的视角看，单体架构一定比微服务架构更简单，更便于掌控，正如单细胞生物比人体的生理结构要简单数百倍；那么，为何还有这么多软件组织开始清算自己的软件资产，花费大量人力物力对现有的单体架构进行重构，走向微服务化？究其主因，不还是系统的质量属性在作祟吗？

纵观软件设计的历史，不是分久必合、合久必分，而是不断拆分、继续拆分、持续拆分的微型化过程。分解的软件元素不可能单兵作战，怎么协同、怎么通信，就成为了系统分解后面临的主要问题。如果没有控制好，这些问题固有的复杂度甚至会在某些场景下超过因为分解给我们带来的收益。

无论是优雅的设计，还是拙劣的设计，都可能因为某种设计权衡而导致系统结构变得复杂。唯一的区别在于前者是主动地控制结构的复杂度，而后者带来的复杂度是偶发的，是错误的滋生，是一种技术债，它可能会随着系统规模的增大而导致一种**无序设计**。

在 Pete Goodliffe 讲述的《两个系统的故事：现代软件神话》中详细地罗列了**无序设计**系统的几种警告信号：

- 代码没有显而易见的进入系统中的路径；
- 不存在一致性、不存在风格、也没有统一的概念能够将不同的部分组织在一起；
- 系统中的控制流让人觉得不舒服，无法预测；
- 系统中有太多的“坏味道”，整个代码库散发着腐烂的气味儿，是在大热天里散发着刺激气体的一个垃圾堆；
- 数据很少放在使用它的地方，经常引入额外的巴罗克式缓存层，目的是试图让数据停留在更方便的地方。

我们看一个无序设计的软件系统，就好像隔着一层半透明的玻璃观察事物一般，系统中的软件元素都变得模

糊不清，充斥着各种技术债。细节层面，代码污浊不堪，违背了“高内聚、松耦合”的设计原则，导致许多代码要么放错了位置，要么出现重复的代码块；架构层面，缺乏清晰的边界，各种通信与调用依赖纠缠在一起，同一问题域的解决方案各式各样，让人眼花缭乱，仿佛进入了没有规则的无序社会。

预测能力

当我们掌握了事物发展的客观规律时，我们就具有了一定的对未来的预测能力。例如，我们洞察了万有引力的本质，就可以对我们能够观察到的宇宙天体建立模型，较准确地推测出各个天体在未来一段时间的运行轨迹。然而，宇宙空间变化莫测，或许因为一个星球的死亡产生黑洞的吸噬能力，就可能导致那一片星域产生剧烈的动荡，这种动荡会传递到更远的星空，从而干扰了我们的预测。坦白说，我们现在连自己居住的地球天气都不能做一个准确的预测呢。之所以如此，正是因为未知的变化的产生。

变化

未来总会出现不可预测的变化，这种**不可预测性带来的复杂度**，使得我们产生畏惧，因为我们不知道何时会发生变化，变化的方向又会走向哪里，这就导致心理滋生一种仿若失重一般的感觉。变化让事物失去控制，受到事物牵扯的我们会感到惶恐不安。

在设计软件系统时，变化让我们患得患失，不知道如何把握系统设计的度。若拒绝对变化做出理智的预测，系统的设计会变得僵化，一旦变化发生，修改的成本会非常的大；若过于看重变化产生的影响，渴望涵盖一切变化的可能，一旦预期的变化不曾发生，我们之前为变化付出的成本就再也补偿不回来了。这就是所谓的“过度设计”。

从需求的角度讲，变化可能来自业务需求，也可能来自质量属性。以对系统架构的影响而言，尤以后者为甚，因为它可能牵涉到整个基础架构的变更。George Fairbanks在《恰如其分的软件架构》一书中介绍了邮件托管服务公司 RackSpace 的日志架构变迁，业务功能没有任何变化，却因为邮件数量的持续增长，为满足性能需求，架构经历了三个完全不同系统的变迁：从最初的本地日志文件，到中央数据库，再到基于 HDFS 的分布式存储，整个系统几乎发生了颠覆性的变化。这并非 RackSpace 的设计师欠缺设计能力，而是在公司草创之初，他们没有能够高瞻远瞩地预见到客户数量的增长，导致日志数据增多，以至于超出了已有系统支持的能力范围。俗话说：“事后诸葛亮”，当我们在对一个软件系统的架构设计进行复盘时，总会发现许多设计决策是如此的愚昧。殊不知这并非愚昧，而是在设计当初，我们手中掌握的筹码不足以让自己赢下这场面对未来的战争罢了。

这就是变化之殇！

如果将软件系统中我们自己开发的部分都划归为需求的范畴，那么还有一种变化，则是因为我们依赖的第三方库、框架或平台、甚至语言版本的变化带来的连锁反应。例如，作为 Java 开发人员，一定更垂涎于 Lambda 表达式的简洁与抽象，又或者 Jigsaw 提供的模块定义能力，然而现实是我们看到多数的企业软件系统依旧在 Java 6 或者 Java 7 中裹足不前。

这还算是幸运的例子，因为我们尽可以满足这种故步自封，由于情况并没有到必须变化的境地。当我们依赖的第三方有让我们不得不改变的理由时，难道我们还能拒绝变化吗？

许多软件在版本变迁过程中都尽量考虑到 API 变化对调用者带来的影响，因而尽可能保持版本向后兼容。我

亲自参与过系统从 Spring 2.0 到 4.0 的升级，Spark 从 1.3.1 到 1.5 再到 1.6 的升级，感谢这些框架或平台设计人员对兼容性的体贴照顾，使得我们的升级成本能够被降到最低；但是在升级之后，倘若没有对系统做全方位的回归测试，我们的内心始终是惴惴不安的。

对第三方的依赖看似简单，殊不知我们所依赖的库、平台或者框架又可能依赖了若干对于它们而言又份属第三方的更多库、平台和框架。每回初次构建软件系统时，我都为漫长等待的依赖下载过程而感觉烦躁不安。多种版本共存时可能带来的所谓**依赖地狱**，只要亲身经历过，就没有不感到不寒而栗的。倘若你运气欠佳，可能还会有各种古怪问题接踵而来，让你应接不暇、疲于奔命。

如果变化是不可预测的，那么软件系统也会变得不可预测。一方面我们要尽可能地控制变化，至少要将变化产生的影响限制在较小的空间范围内；另一方面又要保证系统不会因为满足可扩展性而变得更加复杂，最后背上**过度设计**的坏名声。软件设计者们就像走在高空钢缆的技巧挑战者，惊险地调整重心以维持行动的平衡。故而，变化之难，在于如何平衡。