为什么要让我们的“领域模型”充血裸奔？

做不完的应用软件

我爸是个乡村小学教师，对我所从事的软件行业一无所知，但是他对我的工作稳定性表示怀疑：“你这做软件的，要是有一天软件做完了，你岂不是要失业了？”也许他想起了他作为老师的情况，教完一批学生，下一批又上来了，一茬一茬的。于是又问我：“你们是不是一个软件接着一个软件做？”我回答他：“不是，就一个软件，好几十个人得做好几年呢。”解释了很多次仍旧没有消除他的疑问：“你们做软件怎么会一直做下去？怎么没有个做完的时候呢？”。

如果他在通往张江的地铁上，知道有那么多我伤不起的IT同类们，他也许会更加迷惑。为什么如此庞大的程序员大军，日复一日年复一年地敲着代码，生产出无数的软件，可是他们不用担心失业？为什么需要那么多看上去类似的软件？为什么这些软件永远没有做完的那天？

应用软件独一无二的地方

答案其实很简单，我们做的每一套软件，都是为了解决某个领域的业务需求。**而业务需求永远没有停止变化的一天，这就是为什么应用软件永远也做不完的原因。**

想想我们为了构建一个应用软件，需要做哪些事？古老的三层架构里，我们要实现业务层（Domain Business Logic）、展现层（Presentation）、存储层（Persistence）。还有Authentication、Authorization、Performance、Security等等。再复杂的系统还包括很重要的系统集成。

稍微思考一下就会发现，在这些要做的事情里面，**只有Business Logic是独一无二的。**，而其它都有框架。

Presentation层。从古老的Delphi，Winform，WPF到现代化的AngularJS、ReactJS，演化得风生水起。

Persistence层，从关系型数据库、存储过程，到ORM框架，以及各种NoSql……同样种类繁多。

缓存，安全，系统集成，莫不如是。所有的东西都可以找到框架，这就是为什么写应用软件，跟写游戏，或者写操作系统等比起来，让人觉得是最没有技术含量的。

然而唯独“业务逻辑”没有框架。正是这“业务逻辑”，让每个应用软件区别于其它应用软件。**因此我们决定要做一个应用软件，我们要做的就是实现客户的业务逻辑，这是唯一**

真正的目的。持久化，UI，Cache……，都是手段。

技术与设计

上面提到技术含量的论断，当然是值得商榷的。在软件行业浸淫十多年后，我越来越倾向于把技术能力和设计能力分开。

什么是技术？对RabbitMQ很精通，我们说这人技术不错。什么是设计？什么时候应该用RabbitMQ？什么时候不应该用？以及如何使用？对于所需的场景，用里面的什么模式？这些都是设计。另外一个例子，技术好的人对C#语言本身很精通，设计好的人则知道应该怎么写代码，代码应该封装到哪一层，哪个类中，类与类之间怎么协作，等等。

我把技术称为**术**，把设计称为**道**。道与术是互相促进的，对RabbitMQ越精通，才知道各种情况下如何使用它。反过来，越是熟悉各种情况下使用它的套路，换一个ZeroMQ，也越容易上手。

我一直鼓吹道术双休，不是因为其中一个比另一个更重要，而是因为大部分的程序员，对道的关注都太少了。只关注术的结果，相信每个人都见识过，身边一定有无数这样的人：年纪越来越大，对术的学习能力越来越赶不上年轻人，所以要么转行做管理做业务，要么在技术这条路上慢慢被淘汰。

为什么需要设计？

先退一步，为什么需要设计？两个原因：

- 需求会变。我们的设计就是为了适应可能的变化，我们不能低估需求变化的可能性，导致设计很难适应变化，又不能高估需求变化的可能性而引入过度设计，所以**设计是一种平衡的艺术**，而且设计的输入就是需求。

- 这是这是程序员的自我需要。写代码的过程有点像写文章的过程，要前后连贯，后面的段落跟前面的段落要有逻辑关系。10分钟后写的代码，10分钟前写的代码能够呼应，我们需要重构10分钟前写的代码，来让现在的代码能按新的思路写下去。**代码不是写出来的，代码是重构出来的。**就像去看作家们的手稿，哪个不是修修改改无数次，曹雪芹一部红楼梦，是于悼红轩中披阅十载，增删五次而成。当然不排除有下笔即成的天才，但我们还是别认为自己是天才为好。我们需要好的设计是因为我们需要读昨天的代码来写今天的代码，我们不希望我们自己都看不懂昨天的代码了。

架构与设计

在技术这条路上一直走下去，终极目标基本是架构师，很酷的名字。然而，什么是架构？

想想为什么需要架构？因为需求会变，因为这是程序员的自我需要。答案跟为什么需要设计是一样的。所以**架构就是设计**，我很喜欢Uncle Bob对架构的定义：

> 架构是设计里不可逆的部分。

一般的设计，如果错了，大不了重构，代价可以承担。但大的设计，比如决定了用AngularJS来写前端，并且建立了一套的框架，如果发现错了，重构的代价就大了。架构就是不可逆的设计，所以架构的要求高得多。反过来说，要做架构师，从注重设计开始。

建筑与软件

说了这么多，那么什么是软件开发中的设计？

有一种把用建筑来隐喻软件开发的说法，建筑的设计师是指那些设计图纸的人，建筑工人依据这些设计图纸来实施。换到软件行业，架构师写出设计文档，程序员根据设计文档用代码实现出来。这是错误的隐喻。正确的隐喻是：写代码的程序员才是建筑里的设计师，MsBuild等编译工具才是建筑里的建筑工人。

就是说，软件开发中，**代码才是设计，MsBuild把代码build成exe才是实现。**

程序员是多么幸福，比较一下大楼的设计人员，当他们设计好的方案（设计图纸）一旦被建筑工人们开始“实现”的时候，他们的设计几乎就不能再改了，因为“实现”的成本太昂贵了。而作为软件设计师，我们的“建筑工人”MsBuild包工头以及它的团队（CPU，RAM等小兵）是多么的廉价和高效，几秒钟就把我们的设计给实现了。**这就是我们能够利用“重构”技术的理由。**（想象一下如果大楼设计人员也这么说：“你们先按照这方案盖起来，我看效果，然后再调整（重构）”……）

与传统行业的设计师相比，我们软件设计师能得到的反馈更快更多（因为我们面对的是电脑），这就是我们幸福的地方，也是我们应该利用的地方。

需求与架构

刚刚讲了，需求是设计或者架构很重要的输入。我们来看一下有哪些需求？

1. 业务需求。
2. UI需求，现在前端火的一个重要原因是手持终端的多样性突然爆发，完全吞噬了市场上可用的前端开发。
3. 存储需求，大数据时代，对数据的格式，数据的volume，都提出了新的挑战。

4. 性能需求。

5. 安全需求。

6. 等等…

架构的一个重要使命，就是用好的设计来应对各种需求的各种可能变化。而第一步，就是要隔离这些不同的需求，架构里的每一块砖头，只负责实现一种需求，这样当某种需求变化时，只需要改其中一块砖头，而不是改整面墙。（这不就是面向对象里SOLID原则里的Single Responsibility Principal吗？再一次佐证架构就是设计。）

如果对上面的需求进行分类，第一个我们成为功能性需求，其它则是非功能性需求。话题拉回来，满足功能性需求是软件的根源目的，满足非功能性需求则是为了让软件可用好用而衍生出来的其它目的。

我们做的每一个系统，都在为客户交付独一无二的业务价值。同时，我们也在进化出不同的框架，来满足各个领域的非功能性需求。这些就解决了程序员们“存在的意义”的哲学问题。

在哪里实现业务逻辑？

以上都是铺垫。至此，我们厘清了两点：一是业务需求是软件的根源目的；二是代码是唯一的设计。现在让我们戴上架构师的帽子，想想如何用代码在哪里实现业务逻辑，有很多选择：

1. 前些年很常见如今被人很鄙视的一种是，存储过程。这种曾经非常流行的技术，自然有它产生的原因。存储过程是什么？是数据库里的东西，而且是关系型数据库里的东西。很多人的思维是这样的：当他试图理解一个业务逻辑时，他心里想的是表以及表与表之间的关系，这就是Database-Driven逻辑，在这种逻辑下，把业务逻辑写在存储过程里，是很自然的事情。如果把思维切换到Domain-Driven的模式中：业务逻辑是我的核心，持久化只是一个辅助的手段，我可以用关系型数据库，也可以用NoSql，而NoSql根本没有存储过程，如此，你把业务逻辑写在存储过程中让人情何以堪啊？

2. 写在UI里，这就要提到当年的RAD之王Delphi了。并不是说在Delphi里只能这么做，而是说Delphi里很多人就这么做，UI直接绑定DataSet，用户点击了某按钮，直接在IDE里双击该按钮，生成Btn1_Click方法，把业务逻辑通通写在那。这么做有一万种缺点，但有一个优点，就是RAD中的R（Rapid）。

3. 写在MVC的Controller里，其实等同于2。MVC/MVP/MVVM是类似的一组模式，但是要知道，它们属于Presentation Layer的pattern，不是属于Domain Layer的。

4. 最好的方式，当然是写在一个独立的Domain Layer里。别忘了业务逻辑是一个应用系统唯一独一无二的地方。

如何实现Domain Layer？

我做过几次技术面试，一般都会有个问题：“你能说说你对架构的理解吗？”得到的回答，第一句往往是：“关于架构，一般是分成3层，Presentation，Business Logic，Persistence……”。这句话即使是很Junior的人也能说得上来，可是再往下问就能问出有意思的东西了：3层之间的依赖关系是怎样的？

一般的回答是：Presentation依赖于Business Logic， Business Logic依赖于Persistence。

可是**既然每个应用系统的“业务逻辑”才是应用系统存在的理由，才是开发它的目的所在。而UI展现、数据库存储、Cache等都是为了实现“业务逻辑”这个目的所提供的手段，都有成熟的框架、模式可用，都可以是雷同的。**

那么为什么“业务逻辑”要依赖于“存储技术”？为什么“目的”要依赖于“手段”？

逻辑依赖与物理依赖

其实“目的”依赖于“手段”并没有什么问题，但更准确的说法应该是“目的”受约束于“手段”，具体说就是“业务逻辑层”受约束于“数据存储层”，举个例子，如果使用NHibernate作为ORM框架，设计的“领域模型”一定是把所有属性都设置为virtual，为了迁就于NHibernate的LazyLoad实现技术。这种迁就或者依赖是无法消除的，然而这里说的是概念上或逻辑上的依赖。

如果到了具体实现上，仍然存在这种依赖，就成了物理上的依赖，简单地说就是BLL这个assembly/package会对DAL这个assembly/package有个引用。物理依赖有什么问题？可以有很多答案，本文余下部分从单元测试和自动化测试角度去佐证。

反馈延迟带来的伤害

先离题一下说说反馈。举个例子，我们拿着杯子去饮水机接水，随着水位的上升，我们知道何时应该停止，这就是眼睛看到水位后，大脑给出的反馈。如果反馈延迟（哪怕只延迟2秒）甚至根本没有反馈，会有什么后果？水溢出来了，大脑才反应过来，后果一定是手被烫到。

简单的例子可以说明反馈被延迟带来的危害。**然而在软件开发中，很多团队不断地被延迟的反馈所反复蹂躏伤害。**此话怎讲呢？

举个例子吧，“代码即设计”，如果代码就是我们的设计，那么如何保证我们的设计正确？很多团队最常见的办法是人肉测试。把代码打包成软件，然后丢给测试人员甚至客户。在我经历过的一个瀑布式软件过程里，今天写好的代码，也许要一个月后才会到测试人员手中，半年后到客户手中，也就是说，外界对我们设计（代码）的验证和反馈周

期，需要几个月之久。这是多么大的延迟，2秒延迟就会烫伤我们的手，几个月，我们伤的起吗？

如何加速反馈

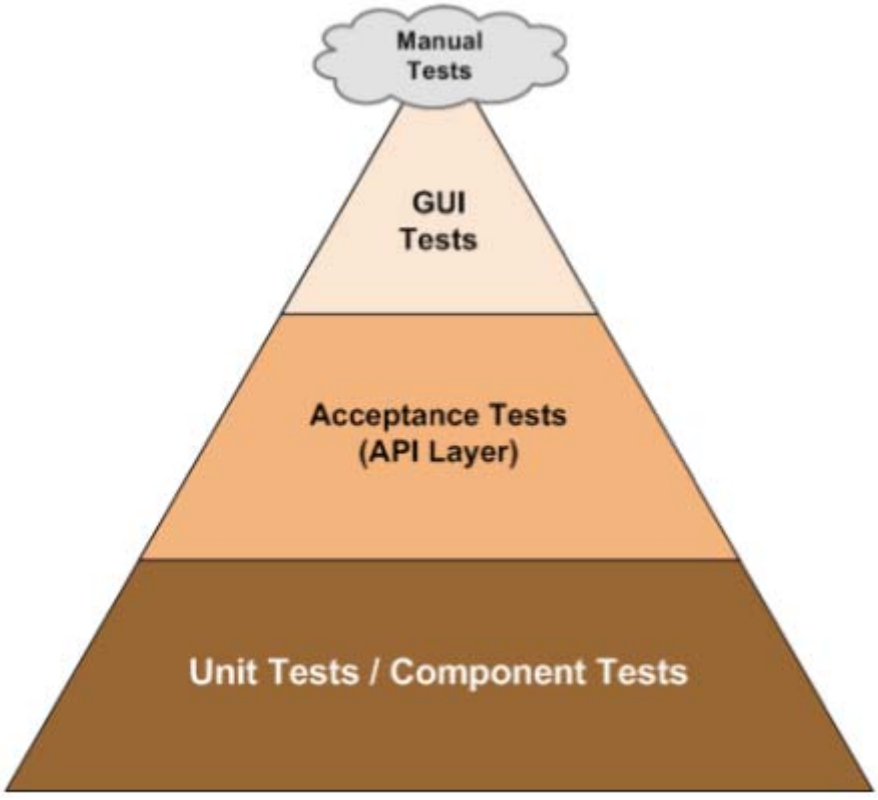
这就是“迭代开发”被引入的一个理由：缩小反馈周期。一个迭代（常见的是2周）内必须把反馈圈给结束掉，也就是2周内完成一个Feature的需求分析、设计、代码、测试等所有环节。从这个角度出发，如果一个迭代里不能getting things done（比如有些团队会在下一个迭代里测试上个迭代开发的用户故事），那不叫迭代，那就叫“两周”。

对于一个Feature来说，两周的反馈周期是可以接受的，毕竟每两周有个功能点给客户看看，确保我们do the right thing，很不错了。

然而如何保证我们do things right（比如，设计和可维护性等等足够好）呢？还有，这两周做的正确的东西，如何保证随着功能的不断增加而不会在将来被破坏呢（答案：回归测试）？如果每两周都人肉回归以前做过的所有功能，那就需要太多QA了。

答案就是**自动化测试。Unit Test保证do things right；验收测试/集成测试来保证do right things。**

自动化测试金字塔



如图，意思是什么呢？如果一个项目的所有自动化测试用例是100，那么最下面的Unit Test应该占80个左右，中间的集成测试占15个左右，上面的UI驱动的验收测试占5个左

右。（还有个最上面的人肉测试，那是浮云:)）为啥呢？因为Unit Test的ROI（投资回报率）最高，它上手容易、运行快，UI驱动的验收测试的ROI最低，运行慢、维护成本高（因为UI是很易变的，UI一变，UI测试脚本就得改。）

但是要注意，这个金字塔不代表下面的测试比上面的重要。它们都重要，处在“浮云”的人工测试也是很重要的，很大的比例是探索性测试。只不过最下层的Unit Test，实现起来是成本最低的，所以一个团队如果要开始自动化测试，最好从Unit Test开始。而最应该写Unit Test的地方是哪个地方呢？毫无疑问，是我们的“目的层”——“领域模型层”。

多说一个题外话：在有些人眼中（比如我），Unit Test不属于自动化测试的范畴，因为**Unit Test首先是一种设计，其次才是测试。**这里不展开了。

Persistence Ignorance

回到我们的问题，“领域模型层”对“数据存储层”有物理上的依赖，导致的不好的结果就是，很难写Unit Test。想象一下，有个Customer类，它的AddOrder（）方法里面调用了DAL层的东西，也就是连接了数据库，那我跑我的UT时也一定要连数据库。连数据库的UT那不叫UT。

怎么办呢？“依赖反转”，Inversion Of Control，IOC。具体做法是：本来BLL依赖于DAL，现在抽一个接口IDAL，让BLL依赖于IDAL，DAL从IDAL继承。从Assembly上来说，BLL和IDAL放到一个Assembly里，DAL放到另一个Assembly，那么DAL这个Assembly现在对BLL那个Assembly有个依赖了。**——这样，就把依赖给反转了。**然后通过Dependency Injection，在运行时把DAL作为IDAL的运行时实例，注入到BLL中。这就是IOC和DI的关系，他们其实不是一个东西，只不过很相关，有时就用IOC或DI泛指这项技术了。

BLL对DAL的依赖，从编译期延迟到了运行期，编译期对DAL没有依赖，只对IDAL有依赖，这就是Persistence Ignorance。

Unit Test

总结下思路：领域模型裸奔的其中一个好处是可以写丰富的Unit Test，Unit Test不连接database，是为了让它跑起来更快，带来的反馈也就更快。

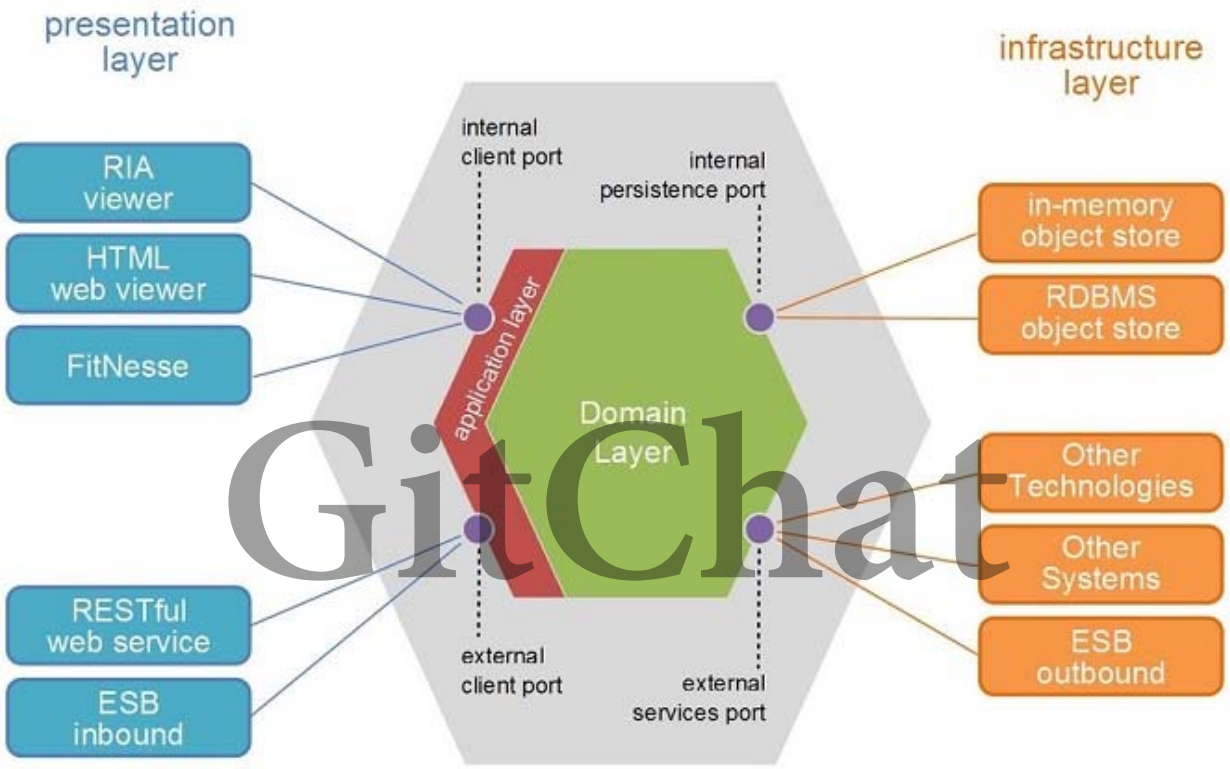
每个unit test都用其方法名说明了我们的设计意图，甚至小片业务逻辑，比如有个测试用例，方法名叫“should_promote_to_VIP_when_customer_buying_platinum_card”，如果让你接手一个别人留下的代码，你不是很清楚里面的业务逻辑，你是愿意去看文档？还是愿意去看他留下的存储过程、或者100行又臭又长的方法？还是愿意看这样的一句话：“当客户买了白金卡后，应该把它提升为VIP”？

unit test的覆盖率足够高时，我们**读完所有一个类的所有unit test方法名（只是名字），我们就知道这个类是干嘛的了。**

事实上，一个项目的维护成本往往是开发成本的四五倍甚至几十倍（越差的代码，这个比例越高）。另外大家也深有体会：读代码比写代码难，我们90%的时间都花在读代码上而不是写代码上了。让代码可读，这是对人对己功德无量的事情。

丰满的领域模型裸奔着向我们呼啸而来

下图是敏捷宣言签署者之一的Alistair Cockburn的Hexagonal Architecture，很精彩的图，留作参考资料了。



后记

想看DDD的人可能会后悔，本文其实没怎么讲DDD，但如同我说的道与术，我们的思考模型、心智模型，我们对软件开发的理解，是道中之道，在这个级别，DDD本身也属于术的范畴了。比如DDD中的Repository模式，就是依赖反转的一个应用。

本文提到了Unit Test和测试金字塔，但对于TDD、BDD没有展开；另外，Unit Test是为了提高代码可读性和可维护性的，可是很多人都有困惑，他们的Unit Test只是让需要维护的代码Double了，本来只要维护生产代码，现在多了一套测试代码要维护，每次做点改动，都要两边改。为什么会这样？因为你的Unit Test写错了。应该怎么写？下篇分解。