## 运用领域场景分析提炼领域知识（下）

运用领域场景分析提炼领域知识（下）

朗读人: 东急之花00′00′′ | 4.4M

### 用户故事

敏捷开发人员对用户故事（User Story）绝不陌生，不过很多人并未想过为何极限编程的创始人 Kent Beck 要用用户故事来代替传统的“需求功能点”。传统的需求分析产生的是冷冰冰的需求文档，它把着重点放在了系统功能的精确描述上，却忽略了整个软件系统最重要的核心—— **用户** 。一个软件系统，只有用户在使用它的功能时才会真正产生价值。传统的功能描述忽略了在需求场景中用户的参与，因而缺乏了需求描写的“身临其境”。用户故事则站在了用户角度，以“讲故事”的方式来阐述需求；这种所谓“故事”其实就是对领域场景的描述，因而一个典型的用户故事，无论形式如何，实质上都是领域场景 6W 模型的体现。

一种经典的用户故事模板要求以如下格式来描述故事：

As a（作为）<角色>   
I would like（我希望）<活动>   
so that（以便于）<业务价值>

格式中的角色、活动与业务价值正好对应了 6W 模型的 Who、What 与 Why。形如这样的模板并非形式主义，而是希望通过这种显式的格式来推动需求分析师站在用户角色的角度，去挖掘隐藏在故事背后的“ **业务价值** ”。需求分析师要做一个好的故事讲述者，就需要站在角色的角度不停地针对用户故事去问为什么。

针对如下用户故事：

作为一名用户，  
我希望可以提供查询功能，  
以便于了解分配给我的任务情况。

我们可以询问如下问题：

* 到底谁是用户？需要执行这一活动的角色到底是谁？
* 为什么需要查询功能？
* 究竟要查询什么样的内容？
* 为什么需要了解分配给我的任务情况？

显然前面给出的用户故事含糊不清，并没有清晰地表达业务目标。 **这样的用户故事并不利于我们提炼领域知识** 。倘若我们将用户识别为项目成员，则这个角色与项目跟踪管理这个场景才能够互相呼应。从角色入手，就可以更好地理解所谓的“业务价值”到底是什么？——项目成员希望跟踪自己的工作进度。如何跟踪工作进度？那就需要获得目前分配给自己的未完成任务。于是，前面的故事描述就应该修改为：

作为一名项目成员，  
我希望获取分配给自己的未完成任务，  
以便于跟踪自己的工作进度。

我以“获取”代替“查询”，是不希望在用户故事中主观地认定该功能一定是通过查询获得的。“查询（Query）”这个词语始终还是过于偏向技术实现，除非该用户故事本身就是描述搜索查询的业务。

显然，在这个用户故事中，“项目成员”是行为的发起者，“跟踪工作进度”是故事发生的“因”，是行为发起者真正关心的 **价值** ，为了获得这一价值，所以才“希望获取分配给自己的未完成任务”，是故事发生的果。通过这种深度挖掘价值，就可以帮助我们发现真正的业务功能。业务功能不是“需要提供查询功能”，而是希望系统提供“获取未完成任务”的方法。至于如何获取，则是技术实现层面的细节。

Dean Leffingwell 在《敏捷软件需求》一书中对这三部分做出了如下阐释：

**角色** 支持对产品功能的细分，而且它经常引出其他角色的需要以及相关活动的环境； **活动** 通常表述相关角色所需的“系统需求”； **价值** 则传达为什么要进行相关活动，也经常可以引领团队寻找能够提供相同价值而且更少工作量的替代活动。

敏捷实践要求需求分析人员与测试人员结对编写用户故事，一个完整的用户故事必须是可测试（Testable）的，因此验收标准（Acceptance Criteria）是用户故事不可缺少的部分。所谓“验收标准”是针对系统设立的一些满足条件，因此这些标准并非测试的用例，而是对业务活动的细节描述，有时候甚至建议采用 Given-When-Then 模式结合场景来阐述验收标准，又或者通过实例化需求的方式，直接提供“身临其境”的案例。例如，针对电商的订单处理，需要为订单设置配送免费的总额阈值，用户故事可以编写为：

作为一名销售经理  
我希望为订单设置合适的配送免费的总额阈值  
以便于促进平均订单总额的提高  
  
验收标准：  
\* 订单总额的货币单位应以当前国家的货币为准  
\* 订单总额阈值必须大于0

如果采用 Given-When-Then 模式，并通过实例化需求的方式编写用户故事，可以改写为：

作为一名销售经理  
我希望为订单设置合适的配送免费的总额阈值  
以便于促进平均订单总额的提高  
  
场景1：订单满足配送免费的总额阈值  
Given：配送免费的总额阈值设置为95元人民币  
And：我目前的购物车总计90元人民币  
When：我将一个价格为5元人民币的商品添加到购物车  
Then：我将获得配送免费的优惠  
  
场景2：订单不满足配送免费的总额阈值  
Given：配送免费的总额阈值设置为95元人民币  
And：我目前的购物车总计85元人民币  
When：我将一个价格为9元人民币的商品添加到购物篮  
Then：我应该被告知如果我多消费1元人民币，就能享受配送免费的优惠

第一个例子的验收标准更加简洁，适合于业务逻辑不是特别复杂的用户故事；Given-When-Then 模式的验收标准更加详细和全面，从业务流程的角度去描述，体现了 6W 模型的 hoW，但有时候显得过于冗余，编写的时间成本更大，这两种形式可以根据具体业务酌情选用。

编写用户故事时，可以参考行为驱动开发（Behavior-Driven Development，BDD）的实践，即强调 **使用 DSL（Domain Specific Language，领域特定语言）描述用户行为，编写用户故事** 。DSL 是一种编码实现，相比自然语言更加精确，又能以符合领域概念的形式满足所谓“活文档（Living Document）”的要求。

行为驱动开发的核心在于“ **行为** ”。当业务需求被划分为不同的业务场景，并以“Given-When- Then”的形式描述出来时，就形成了一种范式化的领域建模规约。使用领域特定语言编写用户故事的过程，就是不断发现领域概念的过程。这些领域概念会因为在团队形成共识而成为统一语言。这种浮现领域模型与统一语言的过程又反过来可以规范我们对用户故事的编写，即按照行为驱动开发的要求，将核心放在“领域行为”上。这就需要避免两种错误的倾向：

* 从 UI 操作去表现业务行为
* 描述技术实现而非业务需求

例如，我们要编写“发送邮件”这个业务场景的用户故事，可能会写成这样：

Scenario: send email  
  
Given a user "James" with password "123456"  
And I sign in  
And I fill in "mike@dddpractice.com" in "to" textbox   
And fill in "test email" in "subject" textbox  
And fill in "This is a test email" in "body" textarea  
  
When I click the "send email" button  
  
Then the email should be sent sucessfully  
And shown with message "the email is sent sucessfully"

该用户故事描写的不是业务行为，而是用户通过 UI 进行交互的操作流程，这种方式实则是让用户界面捆绑了你对领域行为的认知。准确地说，这种 UI 交互操作并非业务行为，例如上述场景中提到的 button 与 textbox 控件，与发送邮件的功能并没有关系。如果换一个 UI 设计，使用的控件就完全不同了。

那么换成这样的写法呢？

Scenario: send email  
  
Given a user "James" with password "123456"  
And I sign in after OAuth authentification  
And I fill in "mike@dddpractice.com" as receiver  
And "test email" as subject  
And "This is a test email" as email body  
  
When I send the email  
  
Then it should connect smtp server  
And all messages should be composed to email  
And a composed email should be sent to receiver via smtp protocal

该用户故事的编写暴露了不必要的技术细节，如连接到 smtp 服务器、消息组合为邮件、邮件通过 smtp 协议发送等。 **我们在编写用户故事时，应该按照行为驱动开发的要求，关注于做什么（what），而不是怎么做（how）** 。如果在业务分析过程中，纠缠于技术细节，就可能导致我们忽略了业务价值。 **在业务建模阶段，业务才是重心，不能舍本逐末。**

那么，该怎么写？

编写用户故事时，不要考虑任何 UI 操作，甚至应该抛开已设计好的 UI 原型，也不要考虑任何技术细节，不要让这些内容来干扰你对业务需求的理解。如果因为更换 UI 设计和调整 UI 布局，又或者因为改变技术实现方案，而需要修改编写好的用户故事，那就是不合理的。 **用户故事应该只受到业务规则与业务流程变化的影响。**

让我们修改前面的用户故事，改为专注领域行为的形式编写：

Scenario: send email  
  
Given a user "James" with password "123456"  
And I sign in  
And I fill in a subject with "test email"  
And a body with "This is a test email"  
  
When I send the email to "Mike" with address "mike@dddpractice.com"  
  
Then the email should be sent sucessfully

只要发送邮件的流程与规则不变，这个用户故事就不需要修改。

### 测试驱动开发

测试驱动开发看起来与提炼领域知识风马牛不相及，那是因为我们将测试驱动开发固化为了一种开发实践。测试驱动开发强调“测试优先”，但实质上这种“测试优先”其实是 **需求分析优先** ，是 **任务分解优先** 。测试驱动开发强调，开发人员在分析了需求之后，并不是一开始就编写测试，而是必须完成 **任务分解** 。对任务的分解其实就是对职责的识别，且识别出来的职责在被分解为单独的任务时，必须是可验证的。

在进行测试驱动开发时，虽然要求从一开始就进行任务分解，但并不苛求任务分解是完全合理的。随着测试的推进，倘若我们觉察到一个任务有太多测试用例需要编写，则意味着分解的任务粒度过粗，应对其进行再次分解；也有可能会发现一些我们之前未曾发现的任务，则需要将它们添加到任务列表中。

例如，我们要实现一个猜数字的游戏。游戏有四个格子，每个格子有 0~9 的数字，任意两个格子的数字都不一样。玩家有 6 次猜测的机会，如果猜对则获胜，失败则进入下一轮直到六轮猜测全部结束。每次猜测时，玩家需依序输入 4 个数字，程序会根据猜测的情况给出形如“xAxB”的反馈。A 前面的数字代表位置和数字都对的个数，B 前面的数字代表数字对但位置不对的个数。例如，答案是 1 2 3 4，那么对于不同的输入，会有如下的输出：

|  |  |  |
| --- | --- | --- |
| 输入 | 输出 | 说明 |
| 1 5 6 7 | 1A0B | 1 位置正确 |
| 2 4 7 8 | 0A2B | 2 和 4 位置都不正确 |
| 0 3 2 4 | 1A2B | 4 位置正确，2 和 3 位置不正确 |
| 5 6 7 8 | 0A0B | 没有任何一个数字正确 |
| 4 3 2 1 | 0A4B | 4 个数字位置都不对 |
| 1 2 3 4 | 4A0B | 胜出 全中 |
| 1 1 2 3 | 输入不正确，重新输入 |  |
| 1 2 | 输入不正确，重新输入 |  |

答案在游戏开始时随机生成，只有 6 次输入的机会。每次猜测时，程序会给出当前猜测的结果，如果猜测错误，还会给出之前所有猜测的数字和结果以供玩家参考。输入时，用空格分隔数字。

针对猜数字游戏的需求，我们可以分解出如下任务：

* 随机生成答案
* 判断每次猜测的结果
* 检查输入是否合法
* 记录并显示历史猜测数据
* 判断游戏结果。判断猜测次数，如果满 6 次但是未猜对则判负；如果在 6 次内猜测的 4 个数字值与位置都正确，则判胜

当在为分解的任务编写测试用例时，不应针对被测方法编写单元测试，而应该 **根据领域场景进行编写** ，这也是为何测试驱动开发强调测试优先的原因。由于是测试优先，事先没有被测的实现代码，就可以规避这种错误方式。

编写测试的过程是进一步理解领域逻辑的过程，更是 **驱动** 我们去寻找领域概念的过程。由于在编写测试的时候，没有已经实现的类，这就需要开发人员 **站在调用者的角度去思考** ，即所谓“ **意图导向编程** ”。从调用的角度思考，可以驱动我们思考并达到如下目的：

* 如何命名被测试类以及方法，才能更好地表达设计者的意图，使得测试具有更好的可读性；
* 被测对象的创建必须简单，这样才符合测试哲学，从而使得设计具有良好的可测试性；
* 测试使我们只关注接口，而非实现；

在编写测试方法时，应遵循 Given-When-Then 模式，这种方式描述了测试的准备、期待的行为以及验收条件。Given-When-Then 模式体现了 TDD 对设计的驱动力：

* 当编写 Given 时，“驱动”我们思考被测对象的创建，以及它与其他对象的协作；
* 当编写 When 时，“驱动”我们思考被测接口的方法命名，以及它需要接收的传入参数；考虑行为方式，究竟是 **命令式** 还是 **查询式** 方法；
* 当编写 Then 时，“驱动”我们分析被测接口的返回值。

例如，针对任务“判断每次的猜测结果”，我们 **首先要考虑由谁来执行此任务** 。从面向对象设计的角度来讲，这里的任务即“职责”，我们要找到职责的承担者。 **从拟人化的角度去思考所谓“对象”** ，就是要找到 **能够彻底理解（understand）该职责的对象** 。基于这样的设计思想，驱动我们获得了 Game 对象。进一步分析任务，由于我们需要判断猜测结果，这必然要求获知游戏的答案，从而寻找出表达了 **猜测结果** 这一领域知识的概念：Answer，这实际上就是以测试驱动的方式来帮助我们进行领域建模。

编写 When 可以帮助开发者思考类的行为， **一定要从业务而非实现的角度去思考接口。** 例如：

* 实现角度的设计：check()
* 业务角度的设计：guess()

注意两个方法命名表达意图的不同，显然后者更好地表达了领域知识。

编写 Then 考虑的是如何验证，没有任何验证的测试不能称其为测试。由于该任务为判断输入答案是否正确，并获得猜测结果，因而必然需要返回值。从需求来看，只需要返回一个形如 xAxB 的字符串即可。通过 Given-When-Then 模式组成了一个测试方法所要覆盖的领域场景，而测试方法自身则以描述业务的形式命名。例如，针对“判断每次猜测的结果”任务，可以编写其中的一个测试方法：

@Test   
public void should\_return\_0A0B\_when\_no\_number\_is\_correct() {  
 //given  
 Answer actualAnswer = Answer.createAnswer("1 2 3 4");   
 Game game = new Game(actualAnswer);   
 Answer inputAnswer = Answer.createAnswer("5 6 7 8");   
  
 //when   
 String result = game.guess(inputAnswer);   
  
 //then   
 assertThat(result , is("0A0B"));   
}

测试方法名可以足够长，以便于清晰地表述业务。为了更好地辨别方法名表达的含义，我们提倡用 Ruby 风格的命名方法，即下划线分隔方法的每个单词，而非 Java 传统的驼峰风格。建议测试方法名以 should 开头，此时，默认的主语为被测类，即这里的 Game。因此，该测试方法就可以阅读为：Game should return 0A0B when no number guessed correctly。显然，这是一条描述了业务规则的自然语言。

这三种方法各有风格，驱动领域场景的力量也各自不同，甚至这些方法在开发实践中并非处于同一个维度，然而在领域场景分析这个大框架下，又都直接或间接体现了场景的 6W 模型。当然，这里展现的仅仅是这些方法的冰山一角，讲解的侧重点还是在于通过这些方法来帮助我们提炼领域知识。同时，借助类似用例、用户故事、任务等载体，可以更加有效而直观地帮助我们理解问题域，抽象领域模型，从而为我们建立统一语言奠定共识基础。

### 提炼领域知识

**提炼领域知识需要贯穿整个领域驱动设计全过程** ，无论何时，都必须重视领域知识，并时刻维护统一语言。在进行领域场景分析时，这是一个双向的过程。一方面，我们已提炼出来的领域知识会指导我们识别用例，编写用户故事以及测试用例；另一方面，具体的领域场景分析方法又可以进一步帮助我们确认领域知识，并将在团队内达成共识的统一语言更新到之前识别的领域知识中。

这种双向的指导与更新非常重要，因为我们提炼的领域知识以及统一语言是领域模型的重要源头。“问渠那得清如许，为有源头活水来。”，只有源头保证了常新，领域模型才能保证健康，才能更好地指导领域驱动设计。

通过前面对用例、用户故事与测试驱动开发的介绍，我们发现这三个方法虽然都是领域场景分析的具体实现，但它们在运用层次上各有其优势。用例尤其是用例图的抽象能力更强，更擅长于对系统整体需求进行场景分析；用户故事提供了场景分析的固定模式，善于表达具体场景的业务细节；测试驱动开发则强调对业务的分解，利用编写测试用例的形式驱动领域建模，即使不采用测试先行，让开发者转换为调用者角度去思考领域对象及行为，也是一种很好的建模思想与方法。

在提炼领域知识的过程中，我们可以将这三种领域场景分析方法结合起来运用，在不同层次的领域场景中选择不同的场景分析方法，才不至于好高骛远，缺乏对细节的把控，也不至于一叶障目，只见树木不见森林。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持