

# 领域驱动设计实践

张逸

www.agiledon.com

领域驱动设计的关注重心是领域，尤其在面对复杂的领域逻辑时，它总能够帮助我们很好地分析领域。领域驱动设计的基础是领域建模。Eric 认为需要和领域专家良好地合作，从交谈中发现通用语言，找到领域的关键词。领域建模是迭代的过程，根据逐渐深入的领域知识来精化模型。不过，领域驱动设计并不排斥其他的分析技术，例如分析模式，或者通过测试驱动开发来引导我们找到问题域的领域模型。

领域建模并非领域驱动设计所独有。在 RUP 中，领域建模就是一个非常重要的环节。它是一种用例驱动的开发方法，通过获得的用例来帮助分析和设计人员寻找对象，以及对象之间的关系。根据我个人的经验，我喜欢采用两种截然不同的方式来获得模型。一种是用例驱动，一种则是测试驱动。在得到初步的领域模型中，我会尝试利用领域驱动设计的思想为对象分类，找到实体、值对象、聚合以及服务对象，并通过分析对象的生命周期，为不同类型的对象建立资源库和工厂对象。

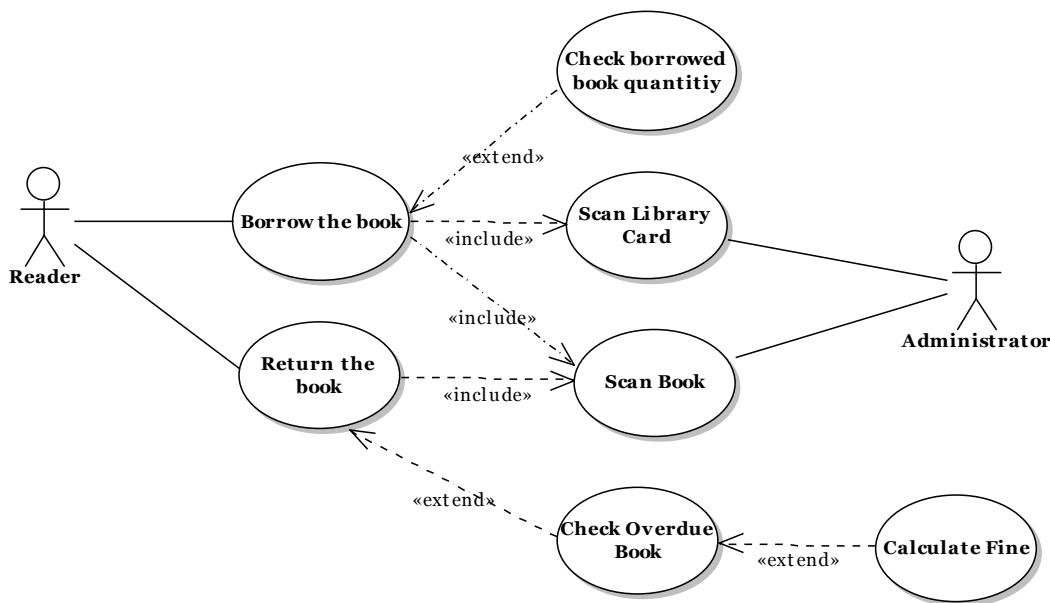
本文将以一个读者耳熟能详的图书馆管理系统作为我们要分析的领域，尝试讲解如何在项目开发中应用领域驱动设计。我将选择用例驱动的方式来获得我最初的领域模型。简单起见，我先给出分析领域的用例以及用例图。

**借书:** 读者携带要借书籍到借书处。图书馆工作人员首先扫描读者的借书卡，获得读者信息，然后扫描书籍的条形码。如果借阅多本，则扫描多本书籍。扫描时，需要判断当前读者的类型，获得读者可借书籍数。如果借阅书籍超出，则提示。如果扫描失败，允许工作人员手工输入编号。借阅的期限为 1 个月。

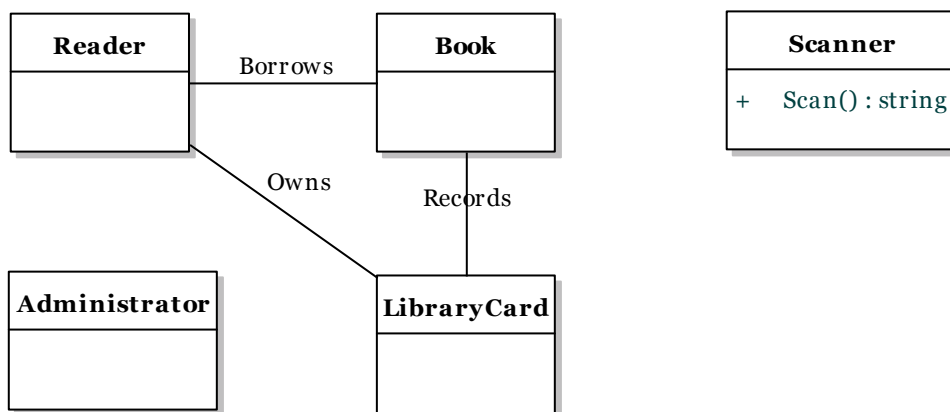
**还书:** 读者携带要还书籍到还书处。图书馆工作人员扫描书籍的条形码，进行还书操作。如果借阅的书籍超期，则提示，并计算出应收罚款的数额。如果扫描失败，允许工作人员手工输入编号。

我采用了摘要方式来描述用例。我喜欢这样一种简洁的方式，它实际上等同于 XP 中的用户故事。在需求并不复杂的时候，或者在对文档要求并不严格的时候，都可以采用这种方式来编写用例。

以下是表达上述两个用例的用例图展现：

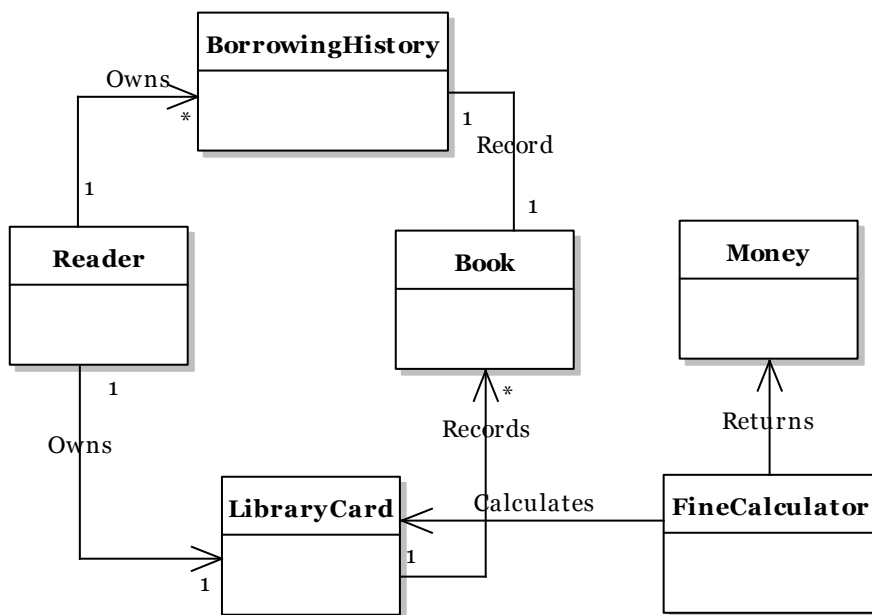


可以首先利用名词/动词法找到模型中的领域对象。这种方法虽然极度地简单与低级，然后在建立领域模型之初，是非常有效的手段。通过对用例的分析，大致可以获得如下对象：Reader, Administrator, Book, Library Card 以及 Scanner。也许还有我们未曾发现的领域对象，这可以通过深入领域或与客户交谈来进一步获得。我们可以尝试着先获得一个最简单的领域模型，如下所示。



我们发现 Administrator 对象是一个孤立的对象，它与其他领域对象没有产生任何关系。至少在借书、还书用例中，我们并不需要管理这个对象，可以考虑删除它。模型中的 Scanner 对象非常特殊，表面上它会对 Book 与 LibraryCard 进行操作，然而对于 Scanner 而言，它并不关心操作的是什么对象，而只需要扫描条形码，返回一个字符串。这是一种行为的体现。在整个系统中，Scanner 对象可以只拥有一个，没有属性和状态，仅提供扫描功能，或者说是服务，因此可以考虑将其定义为服务对象。

Reader 与 Book 之间的关系非常直接，可是在引入 LibraryCard 之后，这个关系就显得有些尴尬了。仔细阅读用例，我们发现识别读者的信息，是通过借书卡来获取的。无论是借书还是还书，都可以通过借书卡来获得读者当前借阅的书。此时，读者与书之间就不存在任何关系了，它已经进行了转嫁。既然借书卡已经实现了对借书关系的管理，我们还有必要保留 Reader 对象吗？阅读用例，我们知道在扫描借书卡时，会获得读者的信息。虽然我们可以在借书卡中保留这些信息，但根据单一职责原则（SRP），将其专门封装为一个对象仍有必要。



目前，借书卡仅仅维护了读者当前借阅的书籍，那么，还需要维护借阅和返还的历史记录吗？从用例的描述来看，并没有这一功能。我们感到疑惑，因为保留历史记录是大多数系统所必备的。此时，客户的答案就显得格外重要。“哦，是的，我们需要查看历史记录！”这是客户给我们的肯定答复。显然，查看历史记录属于另一个用例，它甚至可能属于另外一个上下文（Context），例如关于“查询”的上下文。然而，这一信息的来源却来自于借阅与返回用例，我们应该将其识别出来。如果其他用例需要用到，我认为这个对象是需要共享的。细化后的领域模型如下：

通过对扫描行为的分析，我认为 Scanner 提供的扫描行为与领域无关，而是一种基础设施，因此我将其定义为基础设施层的服务。模型增加了 FineCalculator 对象，用以完成对超期读者的罚款金额计算。显然，它是一个服务对象。注意，BorrowingHistory 与 Book 是一一对应的关系，因为我们需要为每一本书建立一条借阅历史记录。

现在，我们需要识别领域模型中的实体和值对象，以及可能的聚合。我们需要一个唯一的标识来区别读者，且这一标识具有连续性，因此 Reader 是一个实体对象。同样，Book 对象

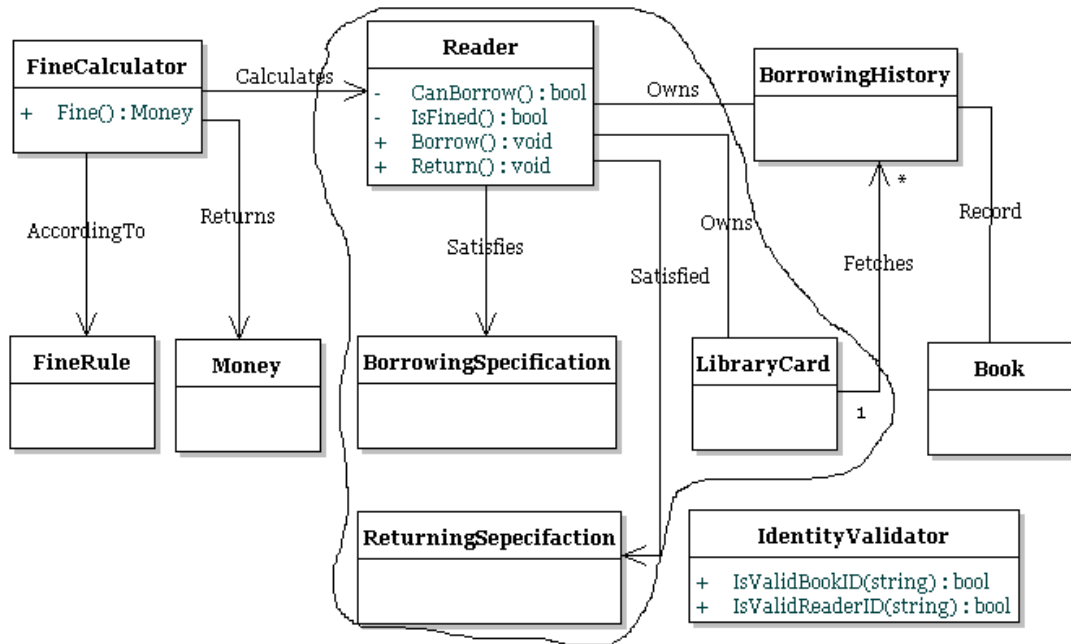
也是一个实体对象，因为我们需要一个唯一标识来完成对书籍的跟踪。注意，在这个模型中的 Book 实体，其实例代表的是具体的某一本书，而不是指同一种书。因为图书馆可能就同一种书购买多本，而读者借阅的是真实的书本，而不仅仅是书的属性。此时，Book 的标识 ID 就显得尤为重要，甚至不能用书籍的 ISBN 来标识。

从表面上看，BorrowingHistory 同样属于实体对象，它的每一条记录都是唯一的，即使存在两条历史记录，具有相同的读者 ID 与书籍 ID，我们仍将其视为不同的记录，因为它们的借阅时间并不相同。不过，对于系统的调用者而言，通常不会去关注所有的借阅记录，而是查询某位读者的借阅记录，因此，我们可以将其作为与 Reader 放在一起的聚合。然而，随着对需求的深入分析，我们发现定义这样的聚合存在问题，因为我们可能还需要查询某本书的借阅记录（例如，希望知道哪本书最受欢迎，跟踪每本书的借阅情况等）。由于 Reader 和 Book 应该分属于不同的聚合，BorrowingHistory 就存在无法划定聚合的问题。既然如此，我们应该将其分离出来，作为一个单独的聚合根。

让人感觉疑惑不解的是 LibraryCard 对象。一方面，它的 ID 来源于 Reader，且存在一对一的关系，因此它可以作为 Reader 聚合的一部分。根据模型图来看，它实际上又记录了读者与书之间的关系。仔细分析，LibraryCard 所维护的这样一种读者与书的关系，事实上正是 BorrowingHistory 的一种体现，区别仅在于一个记录了当前的借书信息，一个还包括过去的借书信息。BorrowingHistory 可以进行信息的持久化，LibraryCard 则完全可以在内存中维持一个当前借阅信息的集合。因此，可以将 LibraryCard 定义在 Reader 聚合中。这样既可以减少对象之间的关联，又能保证对象之间的一致性。

我们还需要深入分析 Reader 对象和 Book 对象的标识 ID，因为这两者的标识 ID 都是通过基础设施的 Scanner 服务获得的。Scanner 并没有能力知道二者之间的区别。而在借阅书籍时，根据需求规定的流程，必须是先扫描借书卡，获得读者信息，然后再扫描书。此外，当扫描出现错误时，系统需要支持操作人员手工输入，因此对手工输入的内容也需要进行 ID 的验证。我们需要有专门验证 ID 的对象。

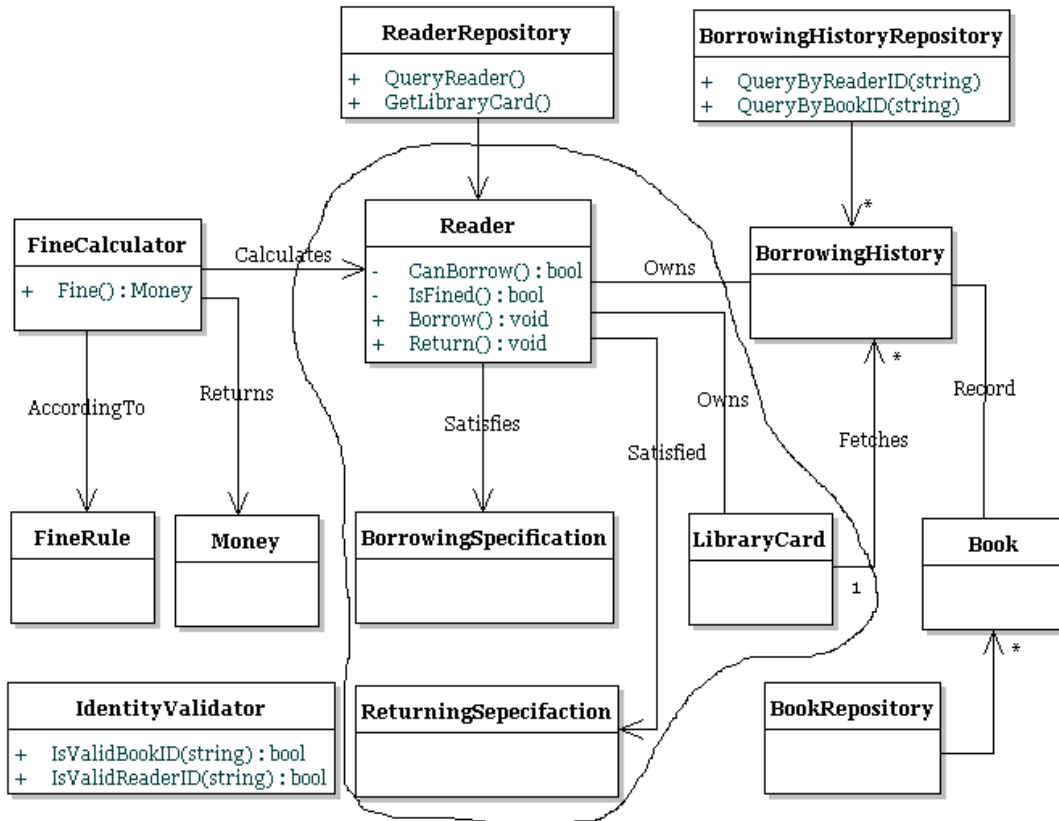
我们还要考虑许多业务规则，例如是否允许读者借书的规则，是否超期的规则，计算罚款额度的规则。如果这些规则极为简单，且不具有变化的可能，可以放在领域对象中。然而，一旦规则变得复杂，就会严重干扰相关领域对象的职责。根据职责分离的原则，我们可以提供专门的规则对象，即领域驱动设计中规格模式的应用。如果可能变化，我们甚至可以引入策略模式，对这些规则进行抽象。经过分析后得到的领域模型如下所示：



Reader 实体对象和 LibraryCard 实体对象处于同一个聚合中，其中 Reader 为聚合根。BorrowingSpecification 和 ReturningSepecification 均为值对象，并放在 Reader 聚合中。FineCalculator 是一个服务对象，它会调用 FineRule 值对象获得罚款规则，通过计算后返回 Money 值对象值。由于聚合的原因，原来 FineCalculator 与 LibraryCard 之间的关系已经修改为计算 Reader 的罚款。

BorrowingHistory 和 Book 均为实体对象，而 IdentityValidator 则为服务对象，负责验证扫描码。

接下来需要为领域对象选择资源库 (Repository)。在领域模型中，只有 Reader、BorrowingHistory 和 Book 三个实体为聚合根对象，因此只需要为这三个对象建立资源库对象即可。



由于需求较为简单，建立的领域模型已经比较完善，我们可以着手编码，对这些模型进行验证。本文没有考虑限定上下文的情况，我希望未来的文章能够以真实的案例对此进行表述。整体而言，根据这个案例，我们已经能够初步领略领域驱动设计的基本步骤。