图书信息系统

陈俊达 161250010

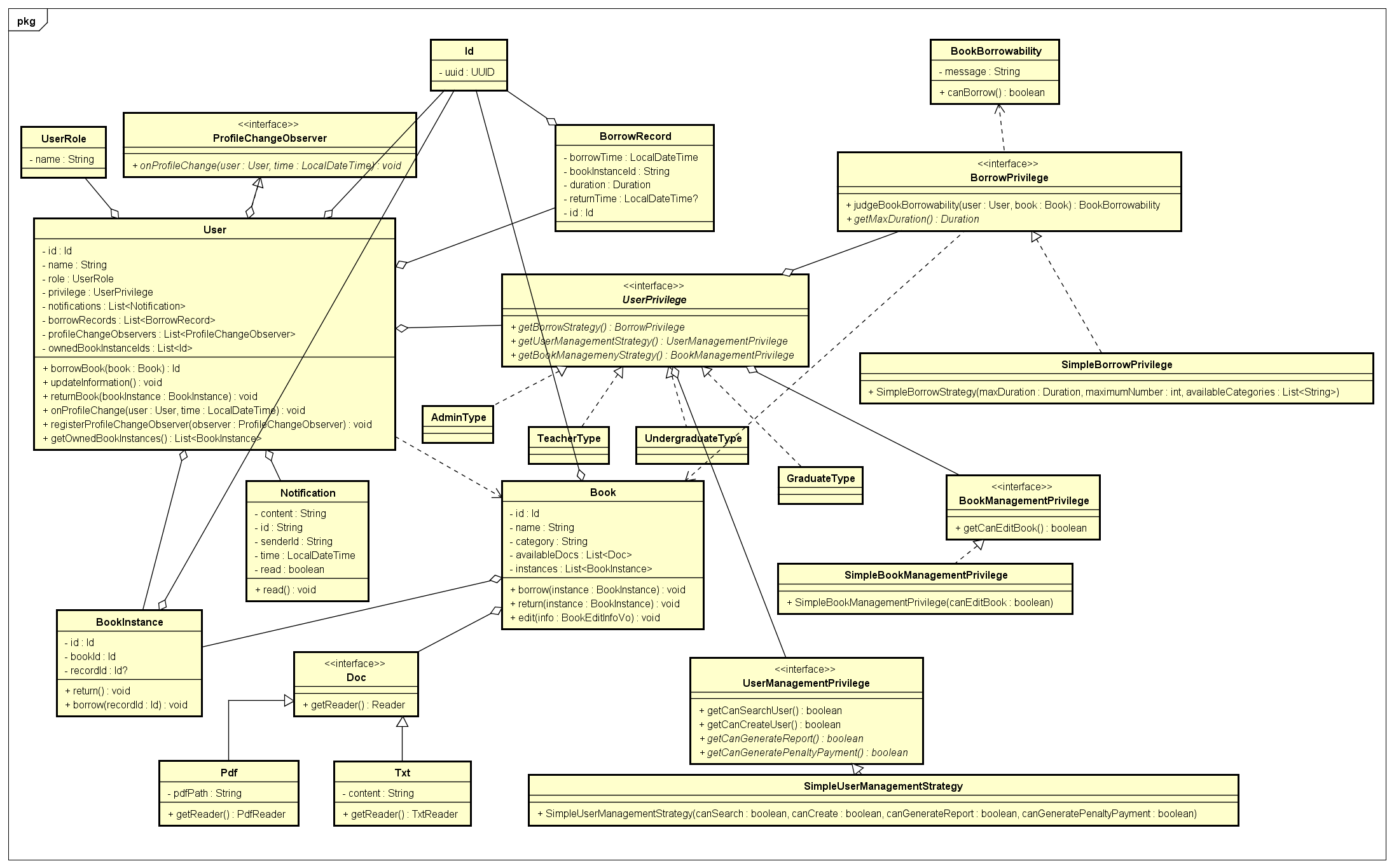
陈俊宇 161250011

2019年1月26日

# 设计

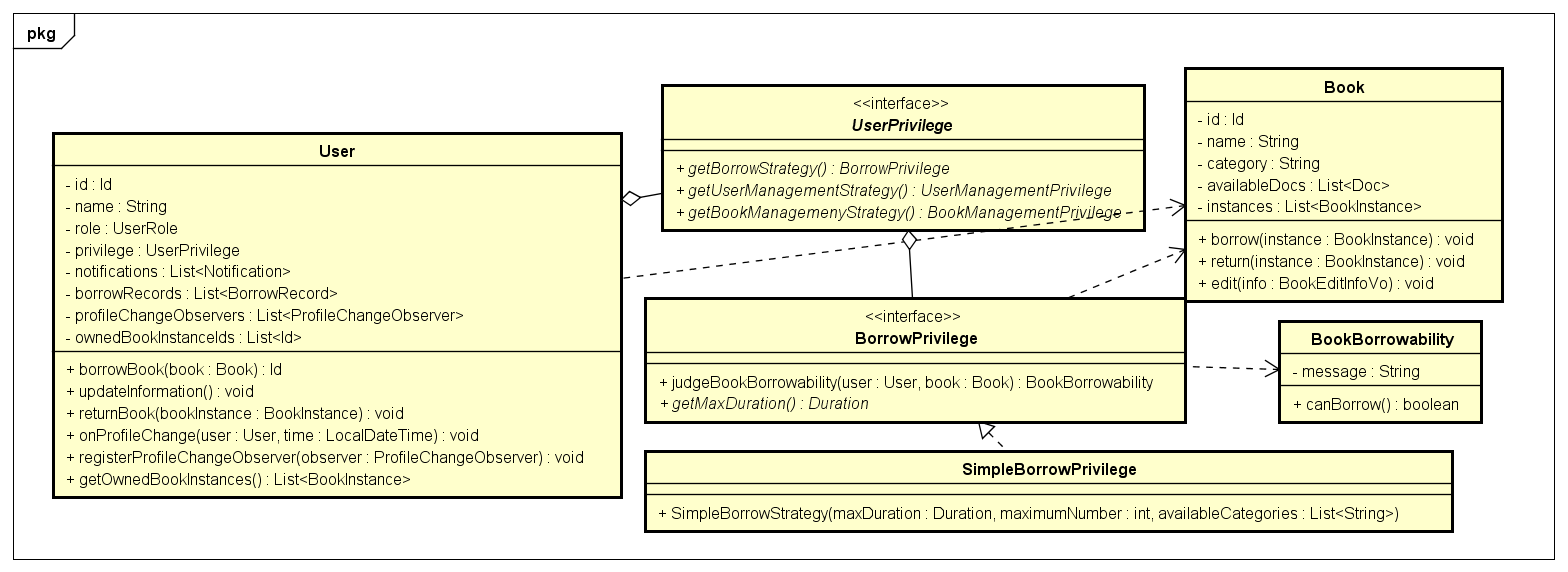
## 总体设计

我们认为，本次作业中的4个需求虽然不同，但是总体来说是相互联系的，所以我们使用领域驱动设计（Domain Driven Design）的方法，针对四个需求进行了整体的设计，其中与业务逻辑相关的**领域层**的类图如下。



## 需求1

需求1要求对不同类型的用户提供不同的借书方法。最终设计如下。



我们根据需求的描述和实际经验分析得出，本需求的核心是**不同用户对书的可借性（borrowability）不同**（distinct access to…）。**可借性**定义为一个用户是否可借一本书，若不可借，需提供错误信息。

所以，我们采用了**策略**模式和**桥接**模式对用户（User）和用户权限（UserPrivilege）进行设计。每个用户（User）对象会包含一个用户权限（UserPrivilege）对象，用户权限（UserPrivilege）对象实现UserPrivilege接口，需要实现接口中的getBorrowPrivilege()方法，此方法会返回一个BorrowPrivilege对象。BorrowPrivilege接口是**借书判断权限相关方法**的总接口，其对象需要实现getMaxDuration()和judgeBookBorrowability(user: User, book: Book): BookBorrowability方法，前者用来**取得用户最长借书期限**，后者为**确定一本书的可借性**的方法。

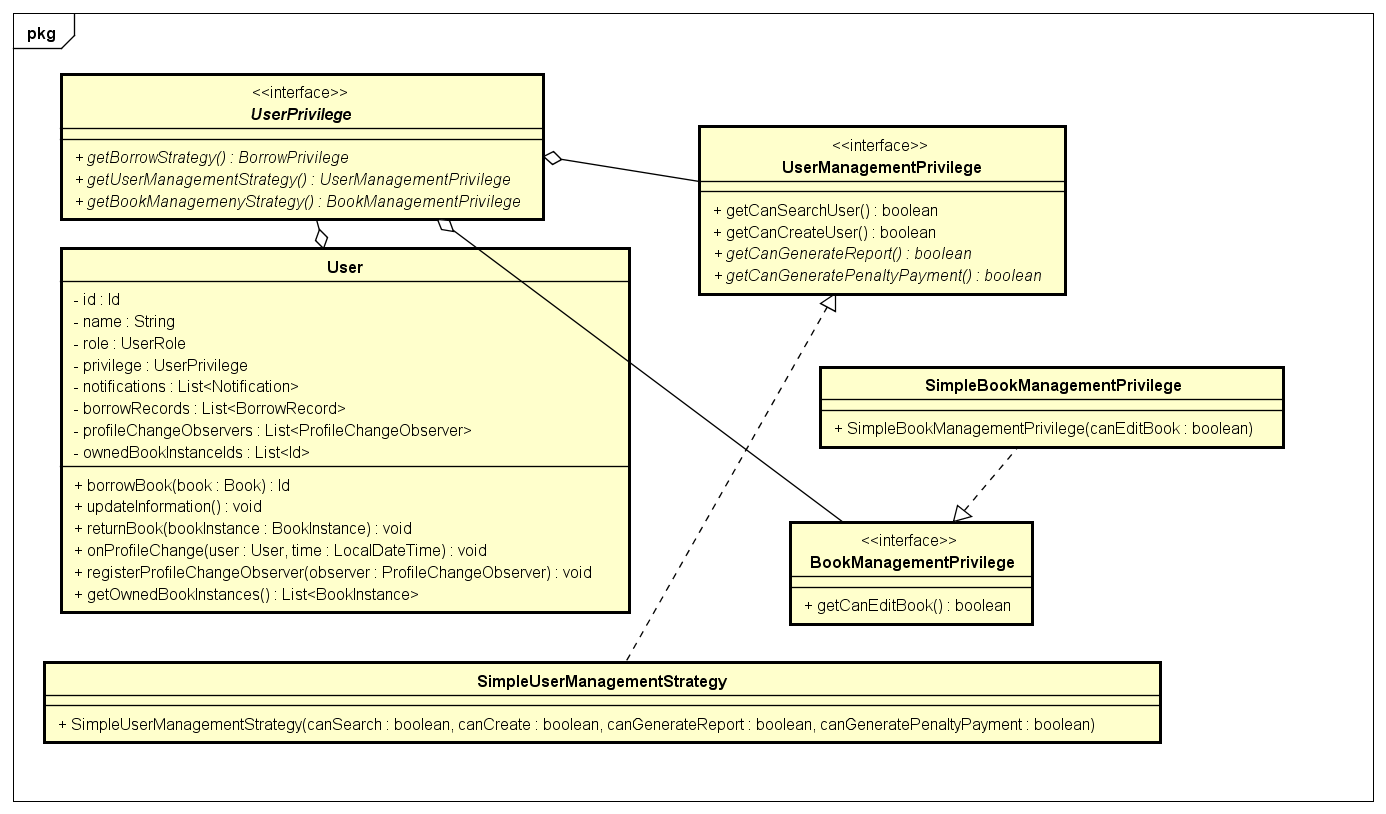
为了方便编程，系统中提供了一个SimpleBorrowPrivilege的BorrowPrivilege默认实现，通过在构造时传入最长时长(maxDuration)，最大数量（maximumNumber）和可以借的书的种类(availableCategories)，在judgeBorrowBorrowability的实现中，根据传入的三个参数、用户信息和书的信息，来判断书的可借性，并在不可借的时候设定对应错误信息（在本实现中，为超过**最大数量**和**种类不可借**）。

这样的设计，不仅可以**方便本原型系统中代码的编写（构造SimpleBorrowPrivilege对象即可，不需要继承）**，也**提供了在系统中动态修改权限的可能性（修改用户的Privilege对象属性值即可），**还**为以后增加更复杂的判断一本书是否可借的逻辑打好了基础（即当SimpleBorrowPrivilege只根据以上所述三个条件不可判断可借性时，可以通过实现BorrowPrivilege定义新的Privilege来实现这些逻辑，不需要修改借书相关实现方法）**。

最后，根据BorrowPrivilege，**用户（User）**对象提供了借书方法（borrowBook），在这个方法中，首先会使用上述**BorrowPrivilege来判断一本书是否可借，若可借再进行进一步操作；若不可，则会返回错误信息以在UI上显示。**

## 需求2

需求2要求在初始设计中，管理也可以修改书信息和用户信息，后者包括生成用户借书记录和逾期款。并在未来设计中，允许新增新的用户类型，且管理员也可以把管理用户信息的**一些权限**给予其他部门或者学校。总体设计如下。



和需求1中的设计相似，我们**使用策略模式和桥接模式**，给UserPrivilege接口新增了getBookManagementPrvilege和getUserManagementPrivilege方法，用来获得此用户类型对应的**图书管理策略**和**用户管理策略**。对于这两种策略，系统也像需求1一样提供了**默认实现**（SimpleUserManagementPrivilege和SimpleBookManagementPrivilege），通过在构造时传入四个boolean代表**可否执行四种操作的权限**（canCreate是否能创建用户，canSearch是否能搜索用户，canGenerateReport是否能生成借书记录，canGeneratePayment是否能生成逾期款）来**方便代码的编写**、**提供动态修改权限的可能性**以及**为以后增加更复杂的判断逻辑打下来基础**。

这样的设计也满足了需求中“**给予一些权限**”的需求，即可以输入不同的参数以允许对应的权限。这样的设计也**使增加新的用户类型变得简单**，**甚至对于简单的新的用户类型（即不需要新增判断逻辑的新类型），可以完全不修改逻辑层的代码，而完全使用UI层动态生成SimpleUserManagementPrivilege和/或SimpleBookManagementPrivilege对象来实现。例如说需要一个只能搜索用户，不能创建用户、生成借书记录和生成逾期款的新用户类型，只需要使用new SimpleUserManagementPrivilege(true, false, false, false)生成对应的UserManagementPrivilege对象即可。**

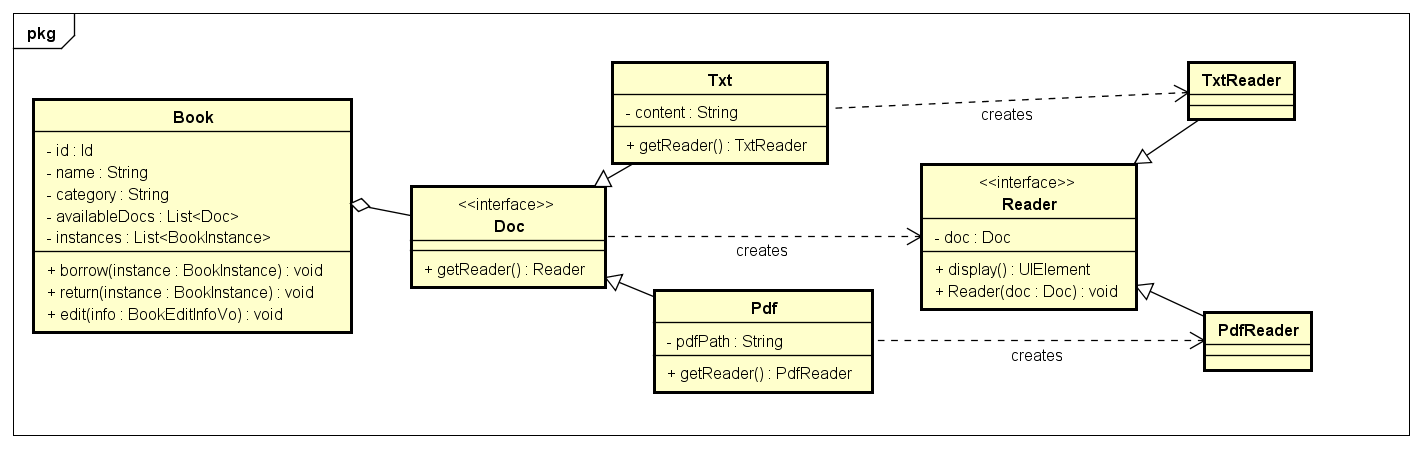
目前我们只知道书管理和用户信息管理各个功能的权限判断比较简单（简单的true或者false），所以接口和默认实现的内容是一致的（均为获得各个功能的权限的boolean值）。但是我们仍然设计成接口和默认实现，其目的时当未来权限判断变得复杂之后，这样设计可以只需要更改接口和默认实现，不需要更改原有生成策略的代码即可适应需求变更。

对于本需求的后半部分“给予其他部门或者学校“，我们认为其**含义不明确**，目前认为给予其他部门或者学校的方式是给系统内部建立新的用户，给予对应的UserPrivilege。在这样的前提下，目前的设计完全可以满足此需求。

## 需求3

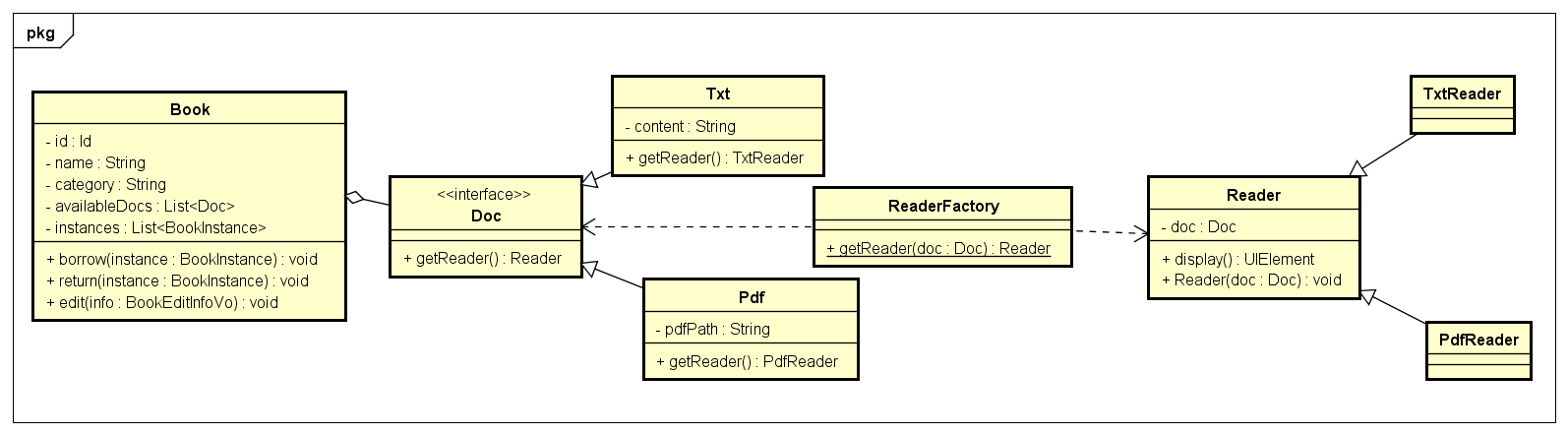
需求3要求对于每种文件类型设计对应的阅读器。

本需求可以使用**策略模式**和**工厂模式**实现。使用策略模式的类图如下：即Doc对象（即一份电子文档，由Book聚合）具有一个生成Reader的方法，一个Reader方法即为一个UI组件的父类，PdfReader和TxtReader为UI组件，可以调用Reader的display方法获得对应的UI组件并显示在屏幕上。



这样的设计**使用多态完全避免了switch-case**，可以直接通过Doc对象拿到对应的Reader并可以直接显示在UI上，但是却出现了**领域层依赖展示层（UiElement为展示层的组件）**的问题，这显然是不合理的。

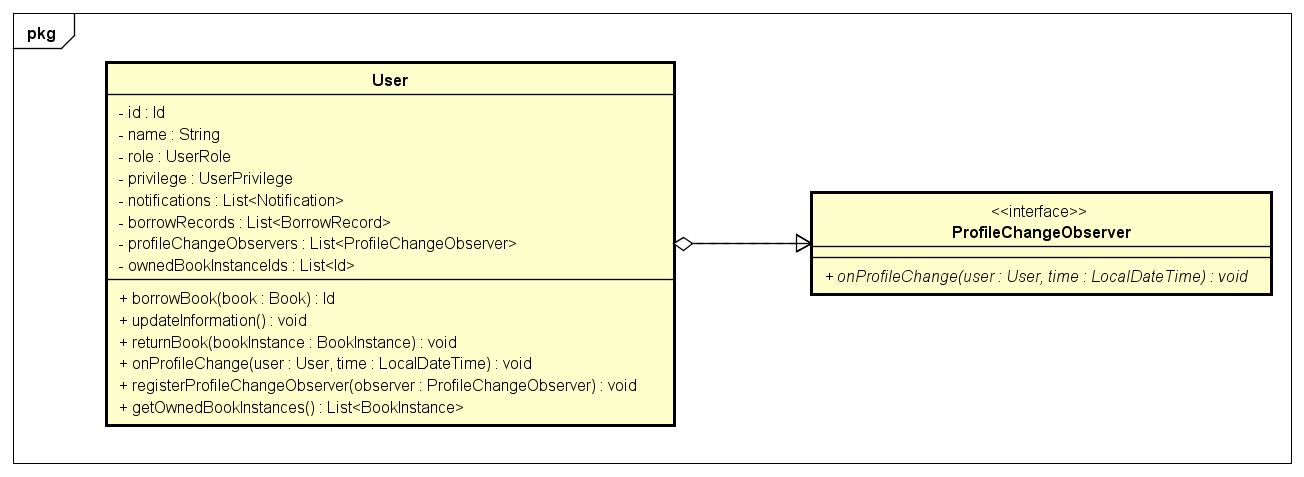
所以，我们最终采用了**简单工厂模式**，即在展示层有一个ReaderFactory，它有一个静态地getReader方法，会根据传入Doc对象的具体类型实例化对应的Reader对象。



这样设计，虽然无法避免if-else/switch-case/状态表等方式判断Doc对象的具体类型，造成新增新文档类型时需要同时修改领域层（增加新的Doc实现类）和展示层（新的Reader以及在工厂中增加新的分支），但是**消除了领域层对展示层的依赖**，使层次间依赖关系变得更加合理和可扩展，我们认为这样的权衡是合理的。

## 需求4

需求4要求普通用户可以修改自己的个人信息，并在修改之后通知所有的管理员。



这个需求非常明显地提示了需要使用**观察者模式**，所以我们采用了观察者模式进行了设计。User对象存放有所有关注自己信息变化地观察者（ProfileChangeObserver），并接受注册观察者（registerProfileChangeObserver），在自己的信息被修改时（updateInformation方法被执行时，下称**信息修改事件**），会对这个列表中的所有观察者调用onProfileChange方法以通知所有的观察者。在这个例子中，一个role为Admin的User（即管理员用户）同样也是一个这样的一个观察者，当其被通知时，会通知自己通知者的信息被修改了。在系统初始化的时候，系统会自动把所有的管理员注册成为每个非管理员用户**信息修改事件**的观察者。这样就实现了这个需求。

在需求实现后，我们出乎意料的发现，这样的设计为我们UI更新提供了方便：我们的UI界面中会显示用户的姓名，当用户的姓名被修改时需要同步更新UI中的一个Label的显示。因为有了观察者模式，我们可以在登录后，将更新UI的方法注册为当前登录用户的**信息修改事件**的观察者。这样，当用户信息更新时，UI对应的地方也会自动更新。

# 贡献和自我陈述

## 161250010 陈俊达

我在这个项目中进行了

## 161250011 陈俊宇

### 贡献

在项目中我主要参与了前期设计的讨论，绘制最初的类图，entity模块的编写以及presentation模块中管理员部分的编写，最后负责测试功能以及编写用户文档。

### 自我陈述

这次项目主要提升了对设计模式的一些理解。就比如策略模式，虽然已经在课上听老师讲过很多次，但最开始设计类图时还是在使用策略模式的同时引进了多态，造成了设计上的冗余。

同时通过领域驱动设计理念的引入，使得之前只包含状态的model以及只包含行为的service融合成为更加饱满的entity，更加符合面向对象的思维方式，对今后项目中面向对象的设计也有所启发。

更深刻体会到了设计的重要性，在整个项目的时间段中前半部分几乎都是在设计，不断调整设计的过程。基于前期详细的设计，后期写代码的过程也就很顺畅。但也不可能在最开始就完全设计正确，就比如我们最开始对book的设计就忽略的系统中的一类书与图书馆中的每一本具体书之间的关系，在后来编写借书模块时才发现缺少了与实体书一一对应的概念，只能重新设计，重构部分代码。