|  |
| --- |
| CSCI 45200 – IUPUI spring 2014 |
| The Healthy People Pharmacy System |
| Assignment 3 – Software Design |
|  |
| **Naeem Tai** |
| **3/31/2014** |

|  |
| --- |
| This document is prepared during the Software Design phase of the Agile Development process of the Healthy People Pharmacy system. This document contains the list of software classes implemented for the first system prototype, design timeline for the rest of the project, discussion about the design patterns and principles, and a screenshot of the running system. |

# Software Classes

Software Classes Implemented:

* ProgressiveTest: temporary classes for testing; progressively test implementation of new classes and interactions between classes
* Item
* RentalItem
* Sale
* Register
* Employee

Language: Java

These classes are contained within the domain model of the layered architecture we chose in the Elaboration Phase. These classes are implemented to produce a working prototype of the system. They are not finalized classes. Their implementation details may change as we further develop the system; however their purpose will remain the same.

# Design Timeline

Gantt chart

# Design Patterns and Principles

* Model View Controller (MVC) pattern, Model View Separation principle

We have applied the Model View Controller pattern and Model View Separation principle to design the Register for our system. This aids our approach to develop the system in a layered architecture. Register is a source of user interaction with the system. However we do not want the users to interact directly with the Register class in the domain layer of the system. We will develop a user interface (UI) in the register in the UI layer of the system. This UI layer will allow the users to interact with the Register class of the system (i.e. operate the register). We will also have a controller for the register which will facilitate the communication between the UI of the register in UI layer and the Register class in the domain layer. This will allow us to easily change the UI of the register without disrupting other layers.

* Façade

We have also applied the Façade pattern in designing the system. In our system, register is the only source of user interaction, therefore also the façade. Users will tell the register what to do, and the register will communicate with other classes in the domain (i.e. sale, productcatalog) to fulfill what users expect.

* Information Expert

We apply the Information Expert principle in designing the system. At the end of a transaction, we need to calculate the subtotal, tax, and total. By the Information Expert principle, the Sale class qualifies to fulfill this responsibility. During the transaction, Sale keeps track of what items are scanned, gets price from Item class, and calculates subtotal. Therefore at the end of the transaction, it has all the needed information to calculate tax and total. Therefore the Sale class is our information expert to calculate subtotal, tax, and total at the end of a transaction.

We can also apply the Information Expert principle to the ProductCatalog class. When an item is scanned, the Register class can call getProductDescription(itemID) method in the ProductCatalog class to get the item’s price, description, etc. The ProductCatalog class has not yet been implemented in our current prototype of the system.

* Factory(Creator)

We apply the Factory (Creator) pattern in designing the Register class and the Sale class. When starting a new transaction from the register, we make the Register class responsible for creating a new instance of the Sale class. When an item is scanned from the register, the Register class passes it off to the Sale class. The Sale class is then responsible for creating a new instance of an Item class and adding it to its list of items. For our current prototype of the system, we are passing the whole Item instance when an item is scanned. This is because we have not yet implemented the ProductCatalog class. Upon its implementation, we will only be passing itemID through our methods instead of the whole item.

* Information Hiding

We have also considered the Information Hiding principle in designing our system. In every class, we have made the attributes of the class private. Additionally, we did not provide any setters. Thus we have restricted other classes from modifying this data. Similarly we also specified some private methods which other classes cannot call. We use this private attributes and methods inside of the class only to keep track of data and fulfill its responsibilities. We do not let other classes see it.

# Screenshots/Mock-ups of the System Prototype

