Developer Manual

1. introduction: The purpose of this manual is to outline the development of The Sudoer’s POS System. This manual is designed for developers looking to modify the POS system functionality.
2. system summary: This system has five general use cases. It can process sales, returns, and rentals. It can startup and shutdown, each dealing with database information retrieval, and storage appropriately.
3. getting started

3.1 system requirements (software & hardware): The system is meant to operate without a barcode scanner. A keyboard is needed for input, and a monitor is needed for output. Right now, it requires at least java 7 to run.

3.2 introduction to the components of your system's installation folder (table format is preferred)

No special setup or third party applications required. Installation folder contains a jar file and some text files. The Users.txt file holds user information. The Product\_Description.txt file holds the product information. The Sales.txt file contains all of the suspended and finished sales. The Rentals.txt file contains all of the suspended and finished rentals. The Coupon.txt file contains all of the coupon information.

3.3 introduction to the third party components

1. We have no third party components.

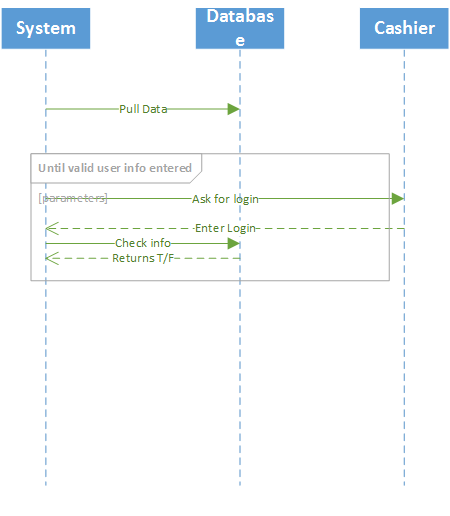
4.1 about your POS system: We have multiple singletons that can be accessed with getInstance methods. They technically are not global variables, but function like them. Our RentalManager, ReturnManager, RentalReturnManager, and SaleManager each keep track of the finished and suspended Transactions of their respective types. The RegisterController is what controls the POS system and what action we are doing.

4.1.1.1 New Use Case: Product Management. Product Management would provide the

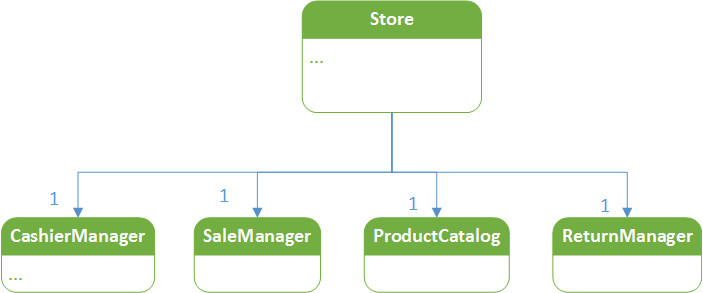
4.2 system startup and shutdown:

4.2.1 On system startup, CashierManager, SaleManager, ProductCatalog, and ReturnManager are all initialized in the store class. All of these classes read from their respective textfiles to initialize system information. Because these textfiles are updated after every transaction or user change, nothing special needs to happen on shutdown.

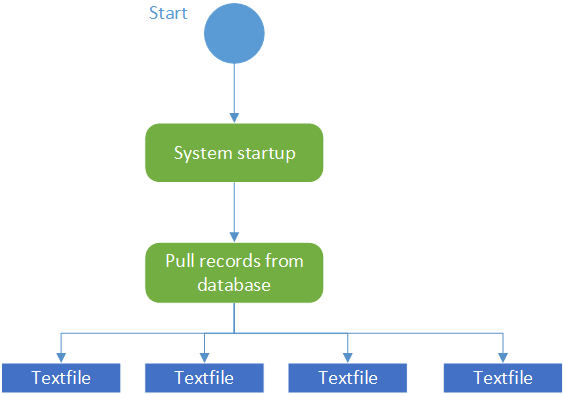
4.2.1.1 architecture view -



Our system on bootup automatically retrieves the necessary data ranging in order to function properly and accurately. No manual commands are needed to load the proper data. The only data the cashier must enter outside of normal transaction 4.2.1.2 class diagrams



4.2.1.3 data view



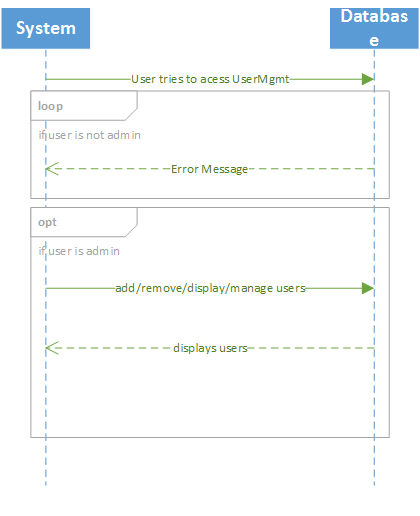
4.2.2 Right now, we don’t have an official credit/debit card validation system. We could add one.

4.2.2.1 On startup in the store class, we would need to connect to the third party system and make sure we disconnect before shutdown. This would all occur in the register controller where we initialize and log out.

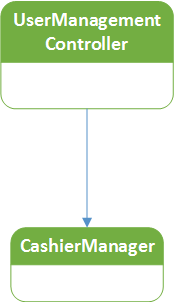
4.3 user management:

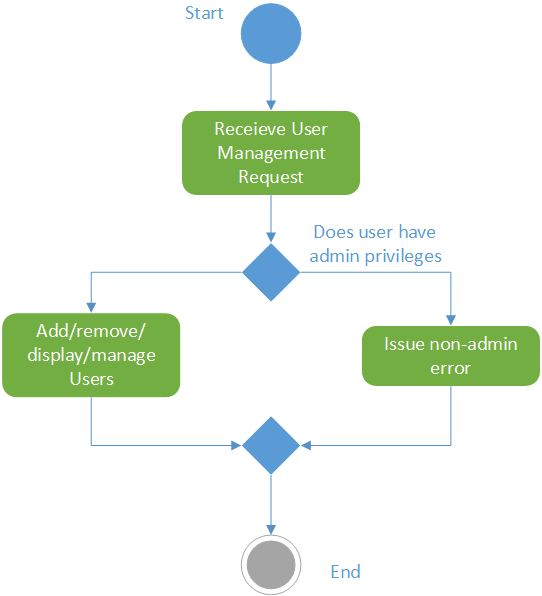
4.3.1 In order to access user management, the user MUST have admin access level according to the users file we read in at the start of the transaction. When logged in as an admin, you can choose user management and add, remove, or display all current users. Each add or remove uses CashierManager to update the text file.

4.3.1.1 architecture view of user management



4.3.1.2 class diagrams of user management



4.3.1.3 data view of user management

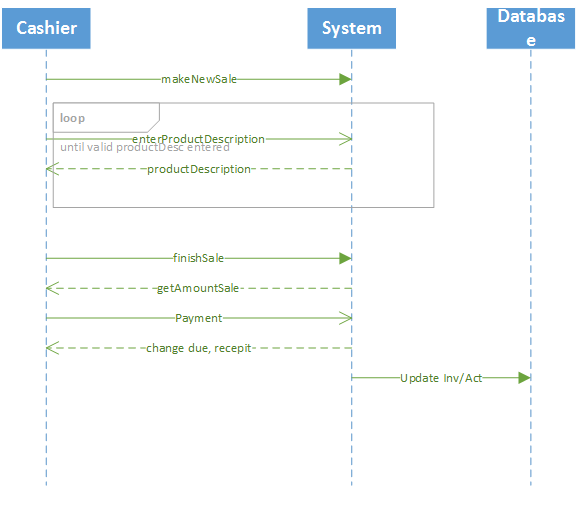
4.3.2 We could add more access levels for larger companies. Right now, we have three access levels, Cashier, Admin, and Manager. Cashiers can old process transactions. Managers can act as Cashiers and provide item overrides. Admins can act as Managers and access user management. We could add a new role like a Store Manager that is able to add or remove items from the product catalog.

4.3.2.1 We use numeric representions for roles, so Store Manager would be represented as 3 in the text file. We would then need to add a ProductController class that would allow us to take input and add or remove items from the product array list in product catalog. After doing this, we would need to update the product text file.

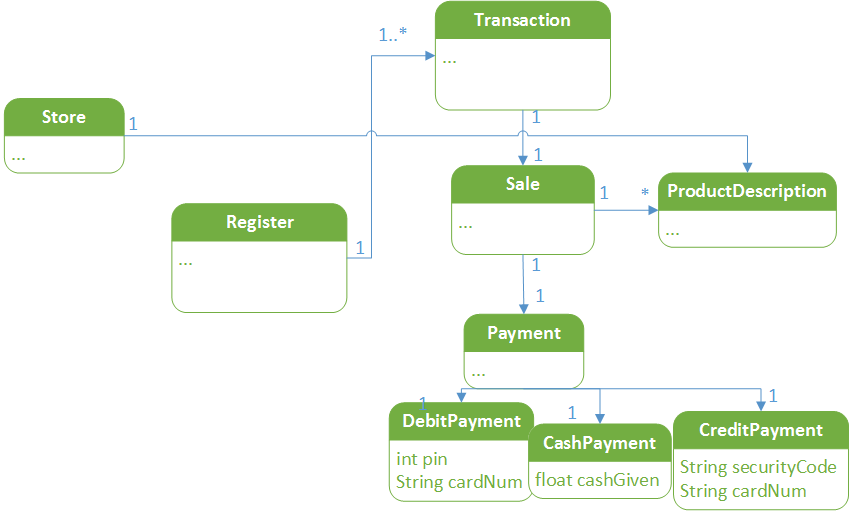
4.4 process sale:

4.4.1 A RegisterControllercreates a SaleController when prompted to start a sale. This sale controller has a Sale domain object. The controller takes care of the higher functionality and and most of the user interaction. it also interacts with the SaleManager to store the sale when necessary.

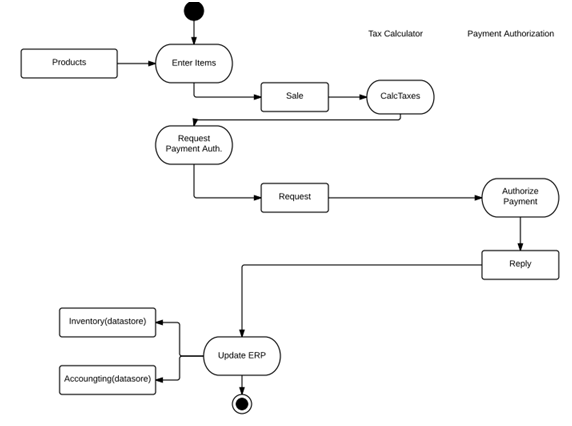
4.4.1.1 architecture view of process sale



4.4.1.2 class diagrams of process sale



4.4.1.3 data view of process sale



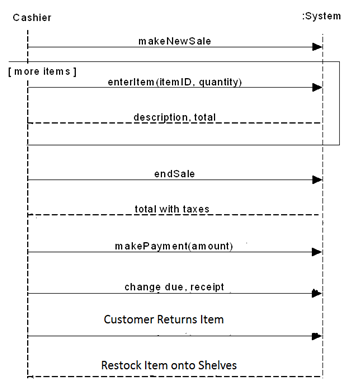
4.4.2 customize your process sale or extend your functionalities of your process sale

4.4.2.1 If we wanted to be able to add a fourth payment method and allow users to pay with checks, we would need some kind of check validation system from a third party. We could connect to this on startup and disconnect on shutdown. Other than that, we would need extend Payment to make a check class similar to CashPayment and add checks as a possible form of payment in the SaleController when we ask for payment.

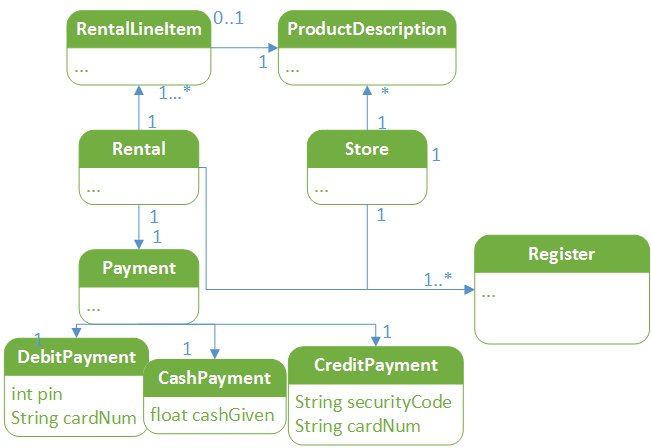
4.5 process rental :

4.5.1 A RegisterController creates a RentalController when prompted to start a return. This controller has a Rental domain object. The controller takes care of the higher functionality and and most of the user interaction. it also interacts with the RentalManager to store the rental when necessary.

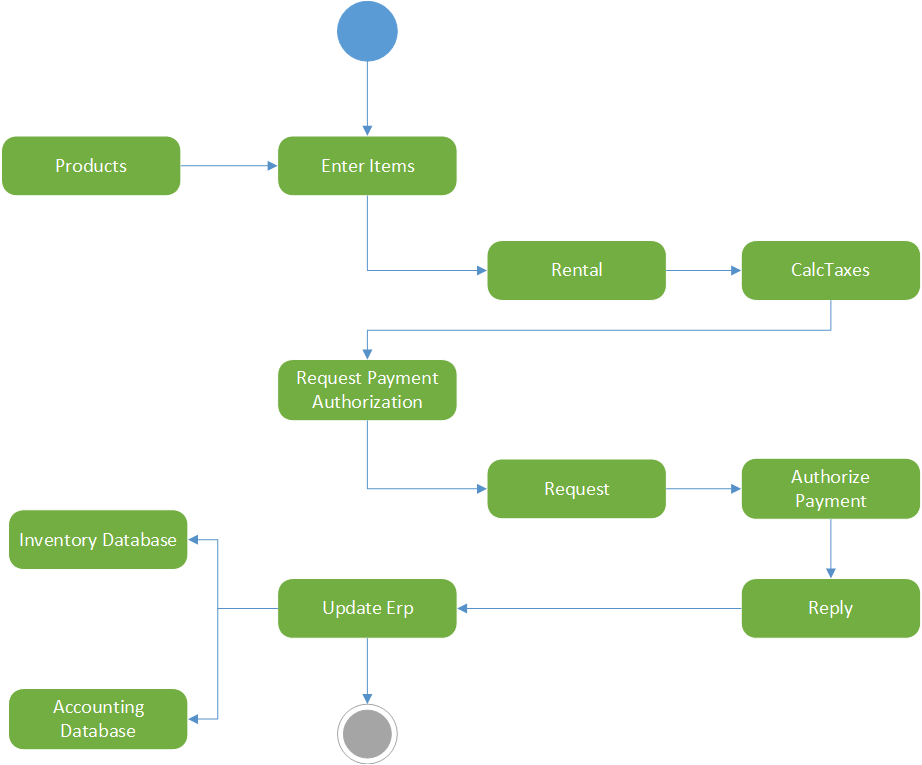
4.5.1.1 architecture view



4.5.1.2 class diagrams



4.5.1.3 data view



4.5.2 customize your process rental or extend your functionalities of your process rental

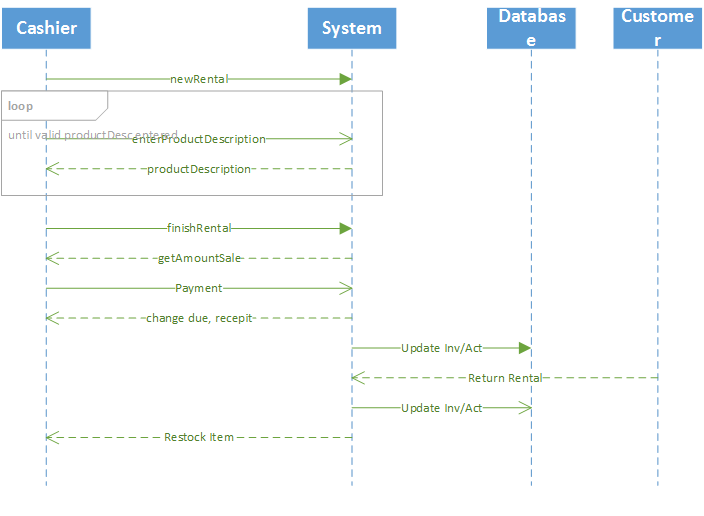
4.5.2.1 If we wanted to give a dicount based on the length of rentals we could store an arrayList of prices instead of a single price. this would hold the price at certain intervals of days. Depending on the length of time the user would like to rent an item, we can charge them a different price per day depending on where it falls in the price range arrayList.

4.5.2.2 the changing of rental rules: illustrate on the components or functions that need to be modified if customer wants to customize their rental rules: If customers wanted to customize their rental rules, we could provide an override function for Rentals like we have for sales. Currently, rentals can be rented for any number of days for a fixed rate each day. If customers wanted a customized discount because they are renting many items, they could do this with an override like we do with sale item prices. We would just have to add the same options to the RentalController.

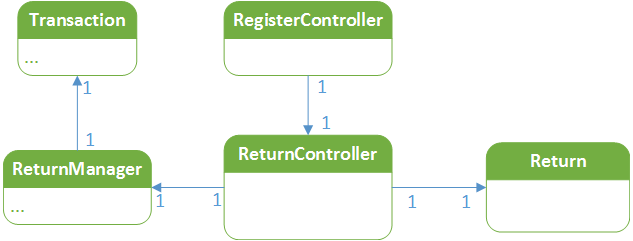
4.6 process return :

4.6.1 A RegisterController creates a ReturnController when prompted to start a return. This controller has a Return domain object. The controller takes care of the higher functionality and and most of the user interaction. it also interacts with the ReturnManager to store the return when necessary and update the sale.

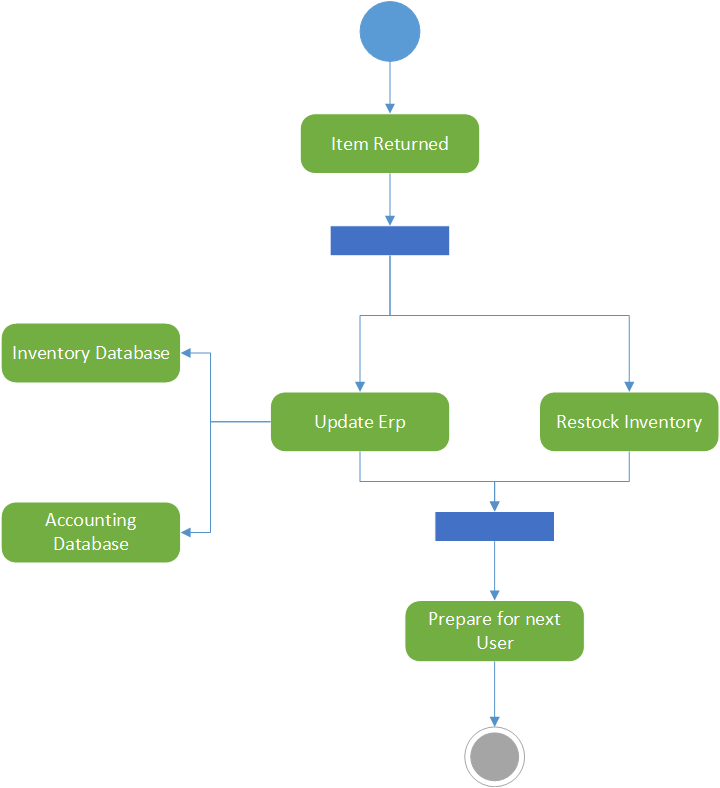
4.6.1.1 architecture view



4.6.1.2 class diagrams



4.6.1.3 data view



4.6.2 We could make rentals extendable.

4.6.2.1 When doing rental returns, we could offer a renew option when we enter the item to be returned, when this happens, we update the number of days in the RentalLineItem in the Rental domain object that corresponds to that item to give them extra time.