*<Component Name>* Design Document

*NOTE: Text in italics should be replaced with your own content.*

Date:

Author:

Reviewer(s):

# Introduction

*A high-level description of this document, for example, “This document defines the design for the Canonical Model”.*

Overview

*Overview of the problem to be solved. What is the problem and why is it being solved? How will the resulting solution provide business value?*

*Consider adding a diagram that explains how this component fits into the overall System with some descriptive text explaining the diagram.*

# Requirements

*This section provides a summary of the requirements for the <Component Name>.*

*Provide your understanding of the requirements, both functional and nonfunctional. Reference the provided Requirements and System Architecture documents. Do not cut and paste from the requirements document.*

*Product Manager and others can read this to understand what requirements your design will support. There is already a requirements doc, so keep this brief and to the point, highlighting the important requirements that the design is addressing. Structure in a way to provide a requirements checklist for your design.*

# Use Cases

*Enumerate the use cases supported by the design,*

*This design supports the following use cases:*

*Include a Use Case Diagram.*

*Include descriptions of each of the actors and use cases.*

# Implementation

*This section of the document will describe the implementation details for ...*

*The implementation section should cover the following topics:*

* *What are the classes, and their properties, associations and methods?*
* *What are the important interfaces and how they will be implemented?*
* *How are the requirements addressed?*

# Class Diagram

*The following class diagram defines the classes defined in this design. Remember to include exception classes.*

*CLASS DIAGRAM GOES HERE*

# Class Dictionary

*This section describes the class dictionary under ‘com.cscie97.store.controller’. The modifications made to the previously created ‘com.cscie97.store.model’ are also discussed.*

## *StoreControllerService*

*The Store Controller Service is an observer that listens to changes in the state of the Store Model Service. The changes can be detected by sensors and appliances which are emulated by commands coming from the command processor for the purposes of this implementation. Once changes are detected, the Store Controller Service is notified and acts upon the events by leveraging the command design pattern. The events are propagated to a command factory which creates appropriate commands. The commands are then stored in a queue for later execution to be triggered by occasional commands that force to empty the queue. The Store Controller Service Keeps a queue of commands and a reference to the Store Model Service to be able to register itself to listen to interesting events.*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| update | (event : Event) : void | An inherited method from the Observer interface that gets called whenever the state of the Store Model Service changes |
| interestedToListen | () : void | This method expresses interest to listen to changes in SMS and registers itself for upcoming notifications |
| stopListening | ():void | Store Controller Services deregisters itself from the list of observers using this method |
| addCommands | (command : AbstractCommand) : ICommand | Adds commands to the queue for later execution |
| invokeCommands | () : void | Forces the execution of commands |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| controllerName | String | An identifier name for the Store Controller Service |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| storeModelService | StoreModelService | This association enables SCS to register and deregister itself to the list of observers contained in SMS |
| commands | ArrayDeque<AbstractCommand> | This is a queue for commands that get executed and cleared when invokeCommands() method from above is called |

## *ICommand*

*This is a top-level interface for all of the commands. It has one method defined common for al of the commands*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : void | Takes appropriate actions during change of state |

## *AbstractCommand*

*This abstract class defines associations like SMS and Ledger to be used by concrete commands to execute their tasks. It also implements the Callable interface from java.util.concurrent to enable asynchronous execution. It defines a constructor to initialize a ledger and gets a singleton instance of SMS during instantiation.*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| call | () : Event | An inherited method from the Callable interface to define what has to happen during asynchronous execution |
| createLedger | (ledgerName, ledgerDesc,  ledgerDesc) : Ledger | Makes an instance of the ledger service from assignment to be leveraged by implementing classes |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| authKey | String | A placeholder for authentication purposes for future assignments |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| ledger | Ledger | A ledger instance is to be used by some of the commands like for checkout or account balance inquiry. |
| storeModelService | IstoreModelService | This association will be used by concrete commands to respond back to SMS with appropriate actions |

## *CommandFactory*

*This class leverages the factory design pattern to parse the payload of an event that is passed in and initializes appropriate commands. The factory delegates to the Store Controller Service the execution of the commands and saves commands in a collection in SCS. It also echoes acknowledgement that commands have been submitted and “We will get back to you” for every command.*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| createCommand | (event : Event) : AbstractCommand | Parses event payload makes appropriate command instances and saves to the command queue in SCS. It also echoes acknowledgement for command received |
| createController | (command : String) : StoreControllerService | Creating a controller must be one of the earlier commands since other commands depend on it. This method creates an SCS instance to be used for orchestrating other commands |

## *AssistCustomerCommand*

*This class is applicable when a customer’s basket weighs more than 10lbs and a customer needs assistance. However, to provide greater customer experience for the disabled and elderly, there is no basket weight restriction required to request for assistance. Any customer can request for assistance according to the design. An extension from the requirements is turnstiles will open for both the customer and the robot to exit. The assumption is that customer has enough balance to purchase the items and is successfully checked out when requesting a robot to get escorted to his car. This class extends AbstractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Sends a robot to where the customer is located, opens turnstiles, and assumes that the customer has successfully checked out |
| calculateBasketWeight | (basket : Basket) : double | Computes the total weight of customer’s basket |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| customerId | String | The customer’s id is used to locate the customer and send him a robot |

## *CheckBalanceCommand*

*A customer may inquire about his available balance in his blockchain account. This class also computes the total value of in the customer’s basket and informs the customer if he has enough funds to purchase the items. This class extends AbstractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Calculates the total value of customer’s basket, checks customer’s balance in his blockchain account and informs customer if he has sufficient funds through the speaker |
| calculateTotal | (Map<Product, Integer>) : int | Computes the amount due for a customer based on items in the basket |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| customerId | String | The customer’s id is used to determine the customer’s ledger account |

## *CheckoutCommand*

*This computes the amount due for a customer based on items in his basket, processes transaction to charge the customer and upon successful transaction echoes confirmation number to the customer. The amount plus fee is transferred from customer’s account to the store’s account. For the purposes of this implementation every store has its own account but having one universal account for all of the stores might be a viable option as well. It also opens turnstile and echoes a goodbye message through the speakers. This class extends AbsractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Computes amount due, charges customer, echoes confirmation number, opens turnstiles and echoes a goodbye message |
| calculateTotal | (Map<Product, Integer>) : int | Computes the amount due for a customer based on items in his basket |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| customerId | String | The customer’s id is used to lookup his blockchain and process transactions |
| storeId | String | Used to identify the store’s account then transfer amount due from customer to store’s account |
| aisleNumber | String | Used to locate closest turnstile to be opened |
| turnstileId | String | This is the initial turnstile the customer needs approached to exit but surrounding turnstiles maybe opened for accompanying guests |

## *CleanStoreCommand*

*This class may be triggered when a camera detects products getting dropped to the floor or when a microphone detects the sounds of items being broken. Robots are instructed to clean the mess. An extension to the requirements is that inventory count will be updated when products are dropped to reflect dropped products can no longer be for sale. This class extends AbstractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Robot is sent to clean up and inventory count gets updated when products are dropped |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| mess | String | Description of what needs to be cleaned |
| storeId | String | Location of the mess |
| aisleNumber | String | Location of the mess |
| shelfId | String | Exact location of the mess |

## *CreateAccountCommand*

*Creates a new account for a customer or the store based on the ledger service. Since addresses must be unique in the ledger service, this constraint must be enforced while trying to create an account. For the purposes of this implementation, the master account in ledger funds all accounts with initial balances so that they can make purchases. This class extends AbstractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Creates a new account using the ledger service |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| customerId | String | The customer’s id is used to locate the customer and send him a robot |

## *AssistCustomerCommand*

*This class is applicable when a customer’s basket weighs more than 10lbs and a customer needs assistance. However, to provide greater customer experience for the disabled and elderly, there is no basket weight restriction required to request for assistance. Any customer can request for assistance according to the design. An extension from the requirements is turnstiles will open for both the customer and the robot to exit. The assumption is that customer has enough balance to purchase the items and is successfully checked out when requesting a robot to get escorted to his car. This class extends AbsractCommand*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : Event | Sends a robot to where the customer is located, opens turnstiles, and assumes that the customer has successfully checked out |
| calculateBasketWeight | (basket : Basket) : double | Computes the total weight of customer’s basket |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| customerId | String | The customer’s id is used to locate the customer and send him a robot |

...

# Implementation Details

*Explain details of the implementation.*

*How do the various parts fit together or interact?*

*How does the design address the requirements? Justify your design decisions and how they address the requirements.*

*Some implementation details may be addressed in the class dictionary, but for things that are not, describe them here.*

*Remember to reference the requirements from the body of the design document to show how your design is addressing the requirements.*

# Exception Handling

*Provide details on your exception handling. What types of exceptions are expected and how are they handled by the design? Describe your exception classes and their properties.*

# Testing

*Provide a testing strategy for testing the component.*

* *Functional*
* *Performance*
* *Regression*
* *Exception Handling*

# Risks

*Document any risks identified during the design process.*

*Are there parts of the design that may not work or need to be implemented with special care or additional testing?*