Store Controller Service Design Document

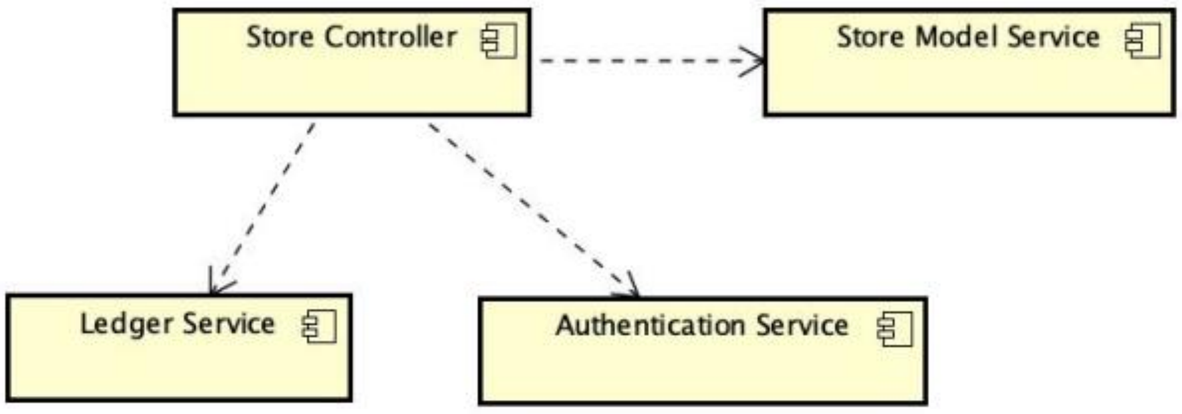
Date: 10/30/2019
Author: Tofik Mussa
Reviewer(s): Taylor Meyer, Derek Kinzo

Introduction

The Store24X7 software system is a next generation fully automated store that consists of four modules to orchestrate the workflow of a retail store. This iteration of the implementation covers the Store Controller Service which mitigates the communication between modules to maintain state and to perform actions. The Store Controller Service can be thought of as an example of the mediator pattern at a high level since it encapsulates complex interactions and acts as a medium of communication. The Store controller Service accomplishes its task by using sensors and appliances to collect information and by using appliances to act upon the information collected.

Overview

This design document is mostly about the Store Controller Service even though a snapshot of the Ledger Service and the Store Model Service are mentioned to illustrate interservice communications. The Store Controller Service detects changes in the stateful objects of Store Model Service, commands the SMS API to update its state, carries out transactions using the Ledger Service and performs authentication with the Authentication Service which will be discussed later. Using the services illustrated below, the Store24X7 system operates indefinitely without human intervention. This leaves error prone and tedious activities when performed manually to a robust machinery.



Requirements

This section provides a summary of the requirements for the Store Controller Service.

The Store Controller Service is where most of the business logic lives in the Store24X7 System. In this iteration of the assignment the Store Model Service implements a Subject/Observable interface which makes it so that one of the observers, the Store Controller Service, can listen to changes and take appropriate action. The Store Controller Service also implements the Observer interface which the Store Model Service knows about and notifies. "Favoring interfaces to implementation" and the dependency inversion principle are leveraged in this design. The Store Model Service doesn't know about the concrete implementations of the Observer interface and notifies however many observers are registered to listening to changing events. Using the command pattern, this implementation encapsulates actions into their own self-contained classes maintaining state and behavior. All the steps when performing actions are logged and output to a file.

Here are some of the interesting actions that the Store Controller Service performs

1. *Create accounts, inquire balances and process transactions using the ledger service*
2. *Manage baskets and inventory keeping the counts and capacity in sync*
3. *Find products to customers and carry items for them using robots when their basket exceeds a 10lbs limit or when they are elderly/disabled*
4. *Perform fully automated checkout activities*
5. *Locate missing customers and keep track of location of customers*
6. *In case of emergency, escort customers out of the store and move robots to address the emergency*

Customers

1. May be registered/guests. Only registered customers can checkout their basket
2. The system keeps track of their location and saves the timestamp of when they were last seen upon leaving the store.
3. May inquire about a missing companion and the system will find it for them
4. May leave the store during emergency without getting charged for purchased items. This may pose some risk, but we value our customers' safety more than any potential loss
5. May inquire about their blockchain accounts' balance at anytime and will get a prompt response
6. If they have enough balance on their accounts, customers can purchase and checkout items of their choosing

Sensors

1. Can be microphones or cameras
2. May gather information about location of customers through voice or image recognition
3. Microphones may listen to customers' inquiries and propagate to the system for further actions
4. Cameras may alert the system about basket activities, emergency situations or activities that require cleaning up after customers

Appliances

1. Appliances are like sensors but are not passive and perform actions. Robots can be considered as both sensors and appliances
2. Appliances may perform maintenance activities like restocking shelves when the store is not busy
3. May respond to emergency using robots
4. Speakers may make the customer experience more interactive by giving prompt response to inquiries. A customer may inquire about an account balance, a missing person or where a product is located, and the system responds through a speaker
5. Turnstiles may open and close based on the customer's need. Opening a turnstile triggers assigning of a basket to a registered customer while closing of a turnstile triggers checkout activity

Store Model Service

1. The Store Model Service is by design a subject that notifies any changes to the Store Controller Service. SCS may be interested in the event and act upon it or ignore the event if no action is necessary
2. The Store Model Service has 3 methods added from the last implementation register/deregister/notify which belong to the subject interface. It also has some convenience methods added for activities like opening/closing turnstiles and for moving robots

Ledger Service

1. Maintains accounts, processes transactions and exposes a public API that SCS uses to make any account inquiries

Commands

1. There is a handful of commands implementing the command interface which maintain state of the event and execute actions on the Ledger and SMS services.A lazy adapter class, AbstractCommand, exists that provides default implementation for common behaviors among commands

Command Factory

1. This class parses the payload in incoming events and makes an instance of the appropriate command setting all of the state needed to perform actions

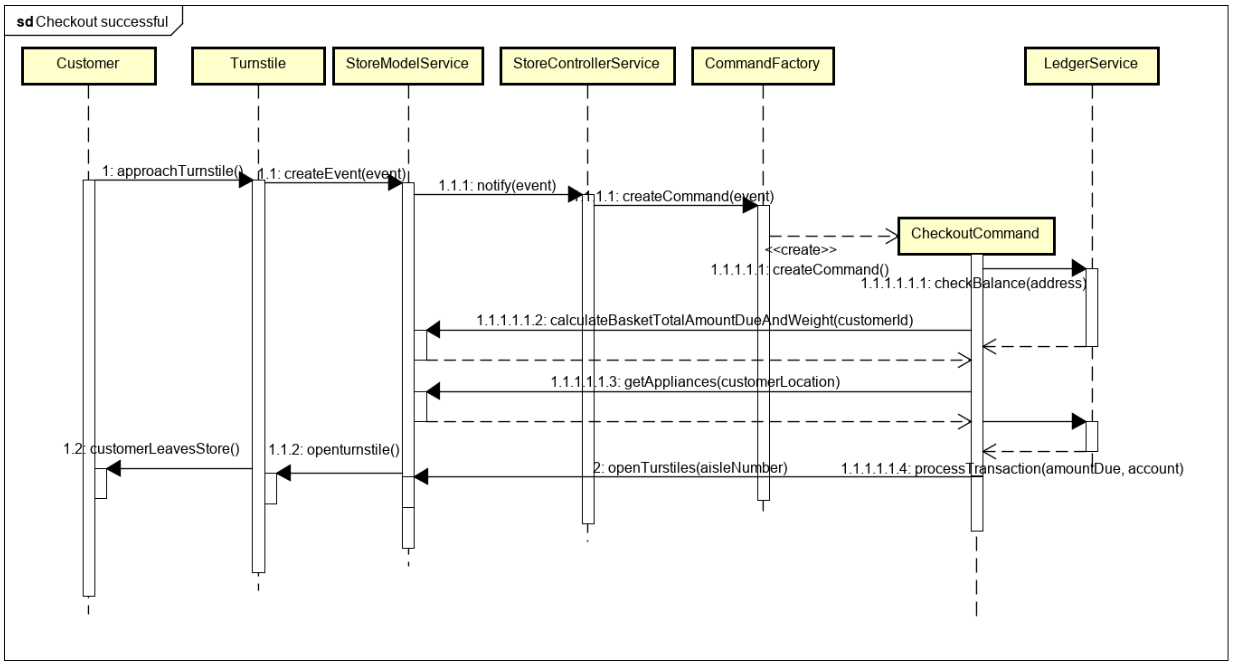
Queuing Mechanism

1. Commands that are created are not executed immediately and are stored in a queue. There is another command that runs occasionally to force execution of commands

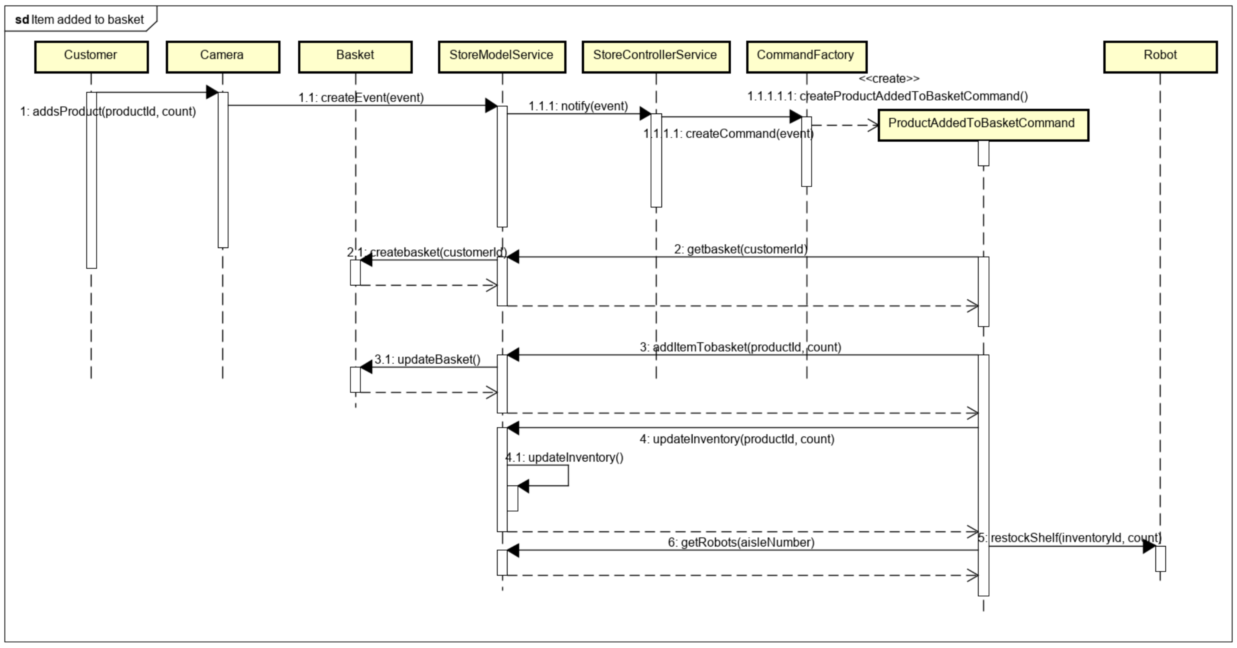
Commands to run the program

```
javac  com/cscie97/ledger/*.java com/cscie97/ledger/test/*.java

javac  com/cscie97/store/controller/*.java

javac  com/cscie97/store/model/*.java com/cscie97/store/model/test/*.java

java -cp . com.cscie97.store.model.test.TestDriver store.script
```

Sample sequence diagram for checkout

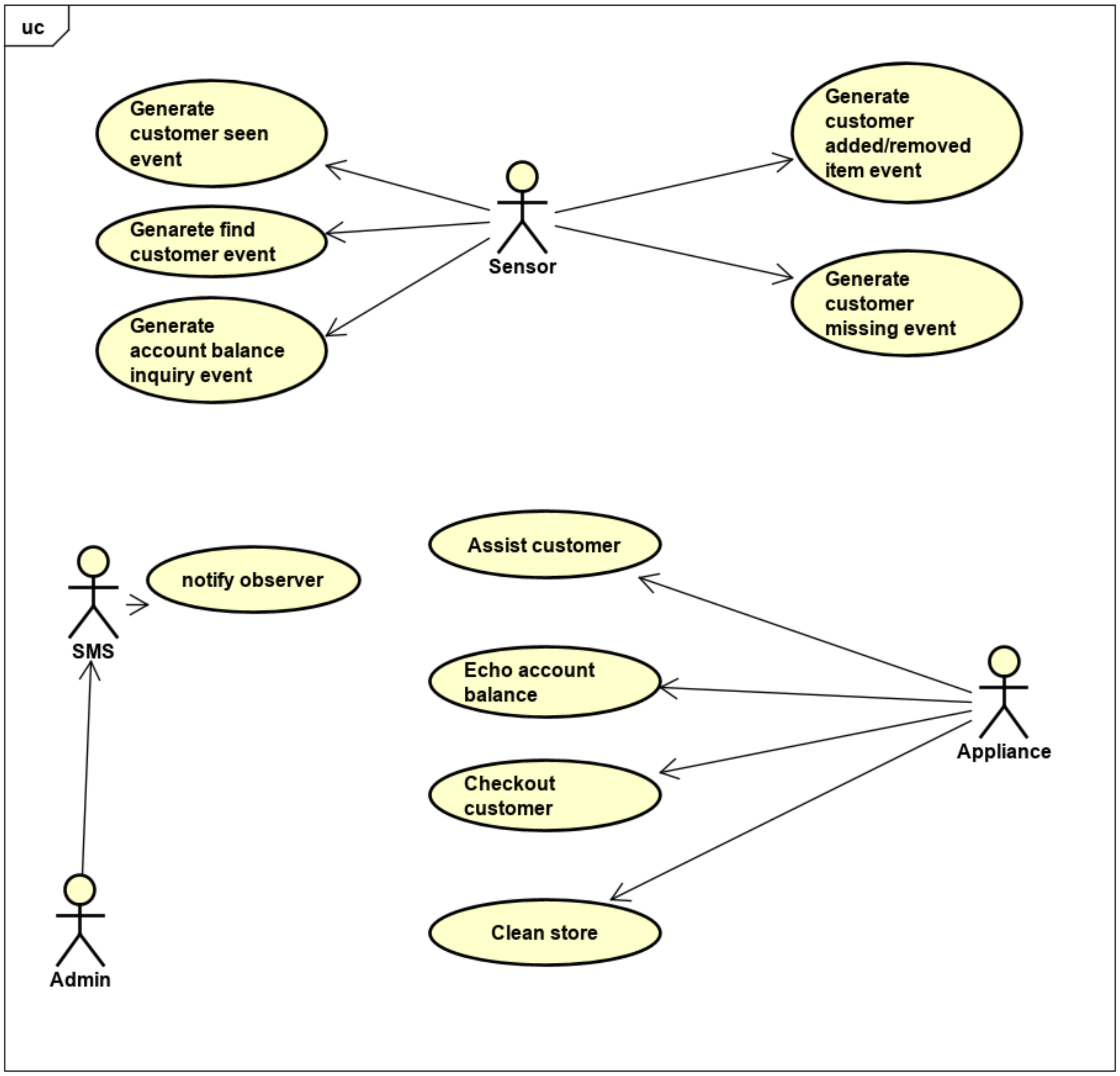

sd Checkout successful
Customer
Turnstile
StoreModelService
StoreControllerService
CommandFactory
LedgerService
1: approachTurnstile()
1.1: createEvent(event)
1.1.1: notify(event)
1.1.1.1: createCommand(event)
CheckoutCommand
<<create>>
1.1.1.1.1: createCommand()
1.1.1.1.1.1: checkBalance(address)
1.1.1.1.1.2: calculateBasketTotalAmountDueAndWeight(customerId)
1.1.1.1.1.3: getAppliances(customerLocation)
1.2: customerLeavesStore()
1.1.2: openturnstile()
2: openTurstiles(aisleNumber)
1.1.1.1.1.4: processTransaction(amountDue, account)

Sample sequence diagram for customer adding items to basket


sd Item added to basket
Customer
Camera
Basket
StoreModelService
StoreControllerService
CommandFactory
Robot
<<create>>
1.1.1.1.1: createProductAddedToBasketCommand()
1: addsProduct(productId, count)
1.1: createEvent(event)
1.1.1: notify(event)
1.1.1.1: createCommand(event)
ProductAddedToBasketCommand
2.1: createbasket(customerId)
2: getbasket(customerId)
3: addItemTobasket(productId, count)
3.1: updateBasket()
4: updateInventory(productId, count)
4.1: updateInventory()
5: restockShelf(inventoryId, count)
6: getRobots(aisleNumber)

Use Cases

The design supports the following use cases are supported by the Store Controller Service


uc
Generate customer seen event
Genarete find customer event
Generate account balance inquiry event
Sensor
Generate customer added/removed item event
Generate customer missing event
SMS
notify observer
Admin
Assist customer
Echo account balance
Checkout customer
Clean store
Appliance

Actors

Store Model Service

The primary actor in this use case is the Store Model Service. SMS consists of stateful entities that approximately translate to real world objects like stores, aisles and appliances. Any change on these entities triggers events that the Store Controller Service takes appropriate actions on.

Admin

The admin has some indirection in this case. The Command Processor which interacts with SMS API can be thought of as an admin for the purposes of this implementation. Command Processor doesn't have direct interactions with SCS but since it also simulates appliances, it can be thought of as an actor.

Appliances and Sensors

Appliances respond to commands and generate events. Sensors generate events but they are passive actors. For example, a camera may trigger a customer moving between aisles and generate a customer seen event

Use cases

- Assist customer
  Robots help customers carrying more than 10lbs or the disabled/elderly
- Check account balance
  Inquire balances of a customer from the ledger service
- Check out
  1. Weight basket items
  2. Check account balance
  3. Process transactions using ledger service
  4. Help customer with basket exceeding 10lbs
  5. Open turnstile
  6. Echo goodbye message
  7. Close turnstile
  8. Move robot back to an aisle where it belongs
- Clean store
  Send robots to clean store and update inventory count if an item has been dropped
- Create account
  Create a new account for a customer or the store itself using the ledger
- Create leger
  Create a ledger that will be able to handle creating accounts and maintaining balances

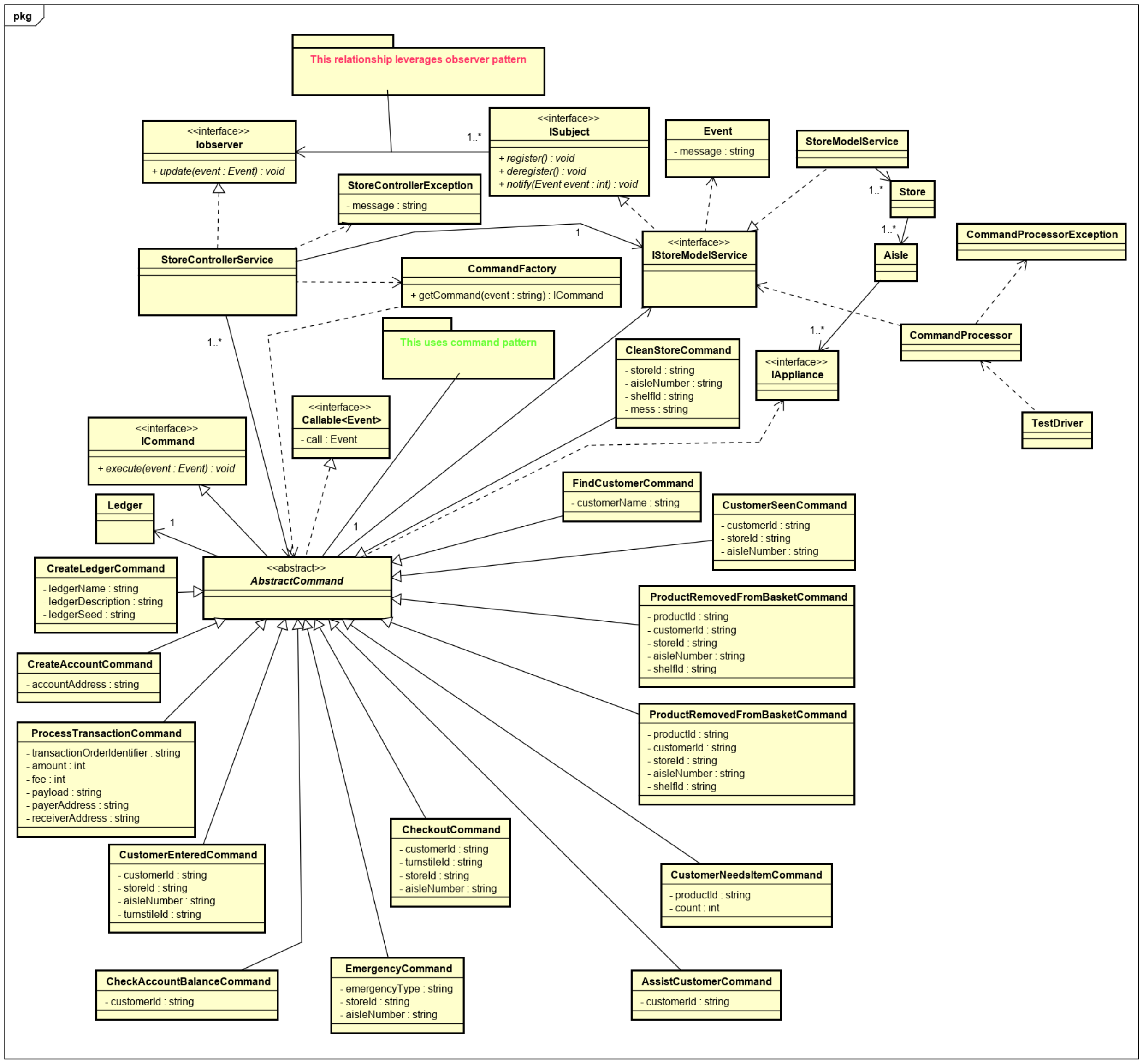
- Enter store
    1. Open turnstile for customer
    2. Registered customers are assigned baskets
- Find item
    1. Microphone listens to customer looking for an item
    2. Speaker echoes back the category and aisle number
- Find customer
    1. Microphone listens to customer complaining about losing a companion
    2. Camera detects where his companion is
    3. Speaker echoes back the location of his companion
- Process transaction
    1. An inquiry is made to the ledger service to check the account balance of the customer
    2. Confirmation of transaction is sent for valid accounts and balances
- Invoke commands in the queue
  Forces commands in the queue to execute
- Add/remove items from basket
    1. Customer picks item
    2. Basket is updated with the count and product id
    3. Inventory count is updated
    4. Event is generated to send robots to restock shelf

Implementation

This section describes designing at the class level. Isubject is an interface that got added to the Store Model Service to enable the Store Controller Service to adhere to the observer pattern. The commands also have a top-level interface ICommand and a lazy adapter AbstractCommand. CommandFactory makes an instance of every command populating its state and stores them in an ArrayDeque defined in StoreControllerService class. The execution of the commands is logged as well.

Class Diagram

The following class diagram mainly shows classes under *'com.cscie97.store.controller'. Since there were some modifications to the existing implementations of the Store Model Service and to amplify the interactions with the Ledger service, a minimal snapshot of those two services is included.*



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
calculateBasket Weight	(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**
accountAddress	String	This must be unique, and it will be used by the ledger service to create accounts

CreateLedgerCommand

This must come before other commands that depend on the ledger and this instance will be used to execute commands that use the ledger. This class extends AbstractCommand

Methods

Method Name	**Signature**	**Description**
execute	() : Event	Creates a new ledger instance

Properties

Property Name	**Type**	**Description**
ledgerName	String	Used to make a new ledger instance
ledgerDesc	String	Used to make a new ledger instance
ledgerSeed	String	Used to make a new ledger instance

CustomerEnteredCommand

Guests may come with a registered customer so all turnstiles near the aisle are opened. The first turnstile will be delegated to the primary customer. Basket is assigned to a customer if there is not one associated with him already. Since the customer is entering a store, his location is updated to the current store he is entering. The customer's balance is looked up from his block chain account. A turnstile is opened for the customer and additional turnstiles may be open for guests coming with the customer. A welcome message is echoed by the speaker. This class extends AbstractCommand

Methods

Method Name	**Signature**	**Description**
execute	() : Event	Allows the customer to enter store

Properties

Property Name	**Type**	**Description**
customerId	String	The customer’s id is used to locate the customer’s blockchain account and to identify guests from registered customers
storeId	String	Identifies the store a customer is visiting
aisleNumber	String	Used to locate and prompt turnstiles in that aisle to open for however many customers
turnstileId	String	Option field to locate the turnstile of the primary customer

CustomerNeedsItemCommand

This extends the requirement to make it more realistic. The steps are
1 - customer requests an item
2 - The nearest robot gets assigned to find item for customer
3 - Robot moves to the aisle where the product is located
4 - Robot moves back to the customer and hands in the product of however many was requested
5 - Customer's basket gets updated with the product and the corresponding count
6 - Inventory count gets updated to reflect the item the customer picked up
This class extends AbstractCommand

Methods

Method Name	**Signature**	**Description**
execute	() : Event	Finds item for customer

Properties

Property Name	**Type**	**Description**
customerId	String	The customer’s id is used to locate the customer and send him a robot
productId	String	Used to look up the item customer is looking for
count	Int	Specifies the quantity of the item customer is looking for

CustomerSeenCommand

Update location of a customer when camera detects it or through voice recognition. This class extends AbstractCommand

Methods

Method Name	**Signature**	**Description**
execute	() : Event	Updates customer’s location

Properties

Property Name	**Type**	**Description**
customerId	String	Identifies the customer
storeId	String	Associates customer with a store
aisleNumber	String	Associates customer with a particular aisle

EmergencyCommand

The sequence of events than happen during emergency are as follows
1 - All of the turnstiles around the area are opened
2 - The speaker urges customers to leave the store
3 - A randomly picked robot from nearby addresses the emergency
4 - The rest of the robots assist customers to leave the store
This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Evacuates customers from store and addresses emergency

Properties

Property Name	Type	Description
storeId	String	Locates where the emergency is
aisleNumber	String	Locates where the emergency is

FIndCustomerCommand

Locates customer by name. The assumption is that it is highly unlikely to have two customers with the same given name at any given time. If it turns out that there is more than one person with the given first name and the customer looking for his mate is dissatisfied, another command shall be sent to find missing person until he is found
This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Finds missing person

Properties

Property Name	Type	Description
customerName	String	Used to search customer by name

ProcessTransactionCommand

Finds payer and receiver then commits transaction. The receiver in this case will be one of the stores. This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Processes transactions using the ledger service

Properties

Property Name	Type	Description
transactionOrder Identifier	String	Incrementing number for each transaction
amount	int	Value of items in basket
fee	int	An amount payable to the master account of the ledger
payload	String	Payload required by ledger
payerAddress	String	Used to lookup payer's account in the ledger
receiverAddress	String	Used to lookup the store's address in the ledger

ProductaddedtoBasketCommand

It is assumed that the customer picks one item at a time. The inventory gets updated to reflect for the count that got taken away and then a robot restocks the shelf to bring it back to initial count.This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Updates customer's basket and inventory count

Properties

Property Name	Type	Description
productId	String	Identifies the product
customerId	String	Identifies the customer
storeId	String	Locates where the product was picked from
aisleNumber	String	Locates where the product was picked from
shelfId	String	Locates where the product was picked from

ProductremovedfromBasketCommand

It is assumed that the customer removes one item at a time. The inventory gets updated to reflect for the count that got put back and then a robot takes the item from floor to store room because it is assumed that the shelf was stocked already
This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Updates customer's basket and inventory count

Properties

Property Name	Type	Description
productId	String	Identifies the product
customerId	String	Identifies the customer
storeId	String	Locates where the product was put back
aisleNumber	String	Locates where the product was put back
shelfId	String	Locates where the product was put back

InvokeCommandsCommand

Forces execution of commands in the queue. This class extends AbstractCommand

Methods

Method Name	Signature	Description
execute	() : Event	Forces execution of commands in queue

Implementation Details

I have had tiny bits of information added to the behaviors to make the implementation mimic a real-world system.

Some examples are

1. Items get dropped to the floor. Since those items can no longer be for sale, the inventory count gets updated
2. A customer puts items to his basket one at a time triggering an event for each item, but those events don't get processed immediately and get stored in a queue. By the time customer approaches turnstile to exit, the items in a basket will be available for computation since commands in the queue may have executed.
3. Restocking of inventory commands also get generated in real time but since they don't get processed immediately, the robot maybe able to stock items in the same area together by the time the commands in the queue execute
4. I have tried to keep the steps in every command as granular as possible, but some steps may be implied. For example, I have fixated to robots to every aisle, but I am able to move them. Once the robot helps customer, the robot may need to go back to the aisle it is responsible for but that is implied by design

The Store Controller Service has enabled me to realize the full picture. Each command is an illustration of the command pattern. The one-way interaction between SMS and SCS is modeled by the observer pattern. Command factory is a convenience class for making instances of the commands and populating their state. Processing transactions and checking out from the requirements spans all the 3 service; SMS, SCS and Ledger. That is a good example of reusability and the powerful implication of leveraging design patterns where they best fit.

Exception Handling

Most of the exception handling has been implemented in previous iterations already since the Store Controller Service is subordinate to the Store Model Service and the Ledger service. However, the StoreException and LedgerException that get thrown by SCS and Ledger respectively are logged then output to file. Exceptions are handled gracefully and failure to execute any of the commands does not collapse the whole system. Execution of the remaining commands will continue while the user is notified that command has failed. StoreControllerServiceException is also part of the implementation and that is being thrown when a payload is arriving from the Store Model Service and is caught under SMS itself.

Testing

The system needs more thorough testing. I was thinking about all the edge cases that system failed to pay attention to. This implementation will be a good candidate for Test Driven Development with Junit. While the test scripts do a good job of orchestrating workflows, enforcing stricter constraints can be achieved by Junit testing.

To make testing more rigorous, the implementor may need to do automated testing with Cucumber/Selenium as well. If this service had a UI component, an end to end regression testing will provide better quality as well.

Risks

*In an event-based programming, the verbose implementation details of listening for events with pus*hing instead of polling can be abstracted away from the user by a message broker like RabbitMQ or Kafka. A publish/subscribe model technology would fit well into this implementation.

While modularity and separation of concerns is obtained with this implementation, there is extra added complexity and moving pieces when performing interservice operations like checkout. Testing may pose inherent problems like when a customer with no blockchain account tries to checkout. The system needs to perform checks and balances to ensure consistency as well.

A queuing mechanism may lead to some important events not getting executed on time and may result in frustration of the customer. The sequence of events may also matter, and the queuing mechanism may sometimes have unexpected outcome.

The store should operate at least with minimal human intervention. There must be some alerting system not mentioned in this implementation that should notify management about potential improvements. The store may be too busy, or the hardware may have worn out and need more appliances/sensors which a human must manage.