Store Controller Service Design Document

Date: 9/23/2021
Author: Austin High
Reviewer(s): Truman Biro, Jiajia Chen

Introduction

Improvements in sensors, robots and electronic payment capabilities have made the fully automated store a reality.

This document outlines the implementation of a Controller that is part of a larger back-end service for a grocery store that operates without employees. Robots stock shelves, clean floors, check inventory, fetch items for customers, and respond to customer questions. Sensor-embedded turnstiles check out customers and monitor for fires. Cameras monitor customers' location throughout stores.

Overview

The Store Controller Service is designed to listen to specific events that occur within stores that belong to the StoreModelService. When an event is created, the Controller service is notified, and the event is passed to the Controller. The Controller determines what the event is, and responds accordingly. That could mean announcing an emergency within the store and opening all turnstiles when a fire is detected. Or it could mean updating the contents of a customer's basket and a shelf's inventory when a camera senses that a customer has added or removed an item from their basket. The Controller service manages a variety of events and ensures that the appropriate actions are taken within the StoreModelService.

Organization:

1. Requirements
2. System Architecture
3. Use Cases
4. Class Diagram
5. Sequence Diagrams
6. Class Dictionary
7. Design Details
8. Testing
9. Risks

Requirements

This section defines the requirements for the Store Controller Service. Implementation of the service must fulfill these requirements:

Observer Pattern:

1. The Controller and Model service implement the Observer pattern.
    a. The Device interface in the StoreModelService will act as the subject, and must contain methods attach(), detach() and notify().
    b. The StoreController will implement the Observer interface which contains the update() method.

Attach / Detach Observer:

Adds an Observer to the Subject (device)'s list of observers.

Notify Observer:

Device class maintains a list of all observers, the Device class has a method, notifty() calls the update(event) method on all observers.

Create Event:

Creates an event object and notifies all observers that a new event has been created.

Command Pattern

1. The Controller implements the Command pattern.
    a. When the update() method is called on the StoreController, the event that is passed is used to create a Command object of the correct type. The StoreController does not know what logic is contained within the Command classes, it simply knows how to instantiate and execute them.

Command Interface:

The Command interface ensures that all command classes implement the same execute() method.

Store Controller:

Acts as an Observer on devices and listens as events are added to devices. When an event is added, the StoreController's update(event) method is performed which instantiates the appropriate type of Command. It then runs the event's execute() method to perform the event's logic.

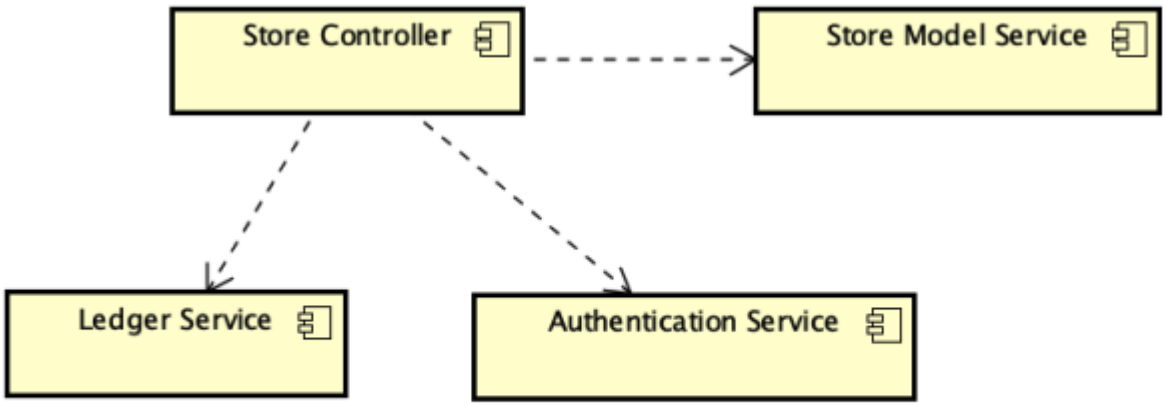
Event Handling:

The event class parses events that are passed as strings to the command processor. The event class reads the first word to determine what type of event is contained. Then, it verifies that the appropriate number of arguments are given in the appropriate format, and stores them as attributes. If an event is not of a valid type, the event class should throw an error.

System Architecture:

The Store Model Service is part of the larger Store 24X7 Software System.

This UML Component diagram shows all of the system's components. This document outlines implementation details for the Store Model Service.



Caption: UML Component diagram describing the high-level services for the Store 24X7

Store Controller Service

The Store Controller Service monitors the sensors and controls all of the appliances in the store. The Controller Service is responsible for monitoring the events received from the sensors located in the store and responding by appropriately controlling the appliances. In addition, the Controller Service is responsible for listening for voice commands and responding by controlling the appropriate appliance.

The Controller Service can also respond to general questions about the state of the store. The Controller Service is responsible for checkout and processing transactions. It uses the Ledger Service to manage transactions and account balances for customers.

Ledger Service

The Ledger Service specifies an implementation of a blockchain service as a means of accepting payment from customers. The Ledger Service manages the transactions, accounts, and blocks that make up the Blockchain. Once a transaction has been submitted and added to the blockchain, its contents become immutable.

Store Model Service

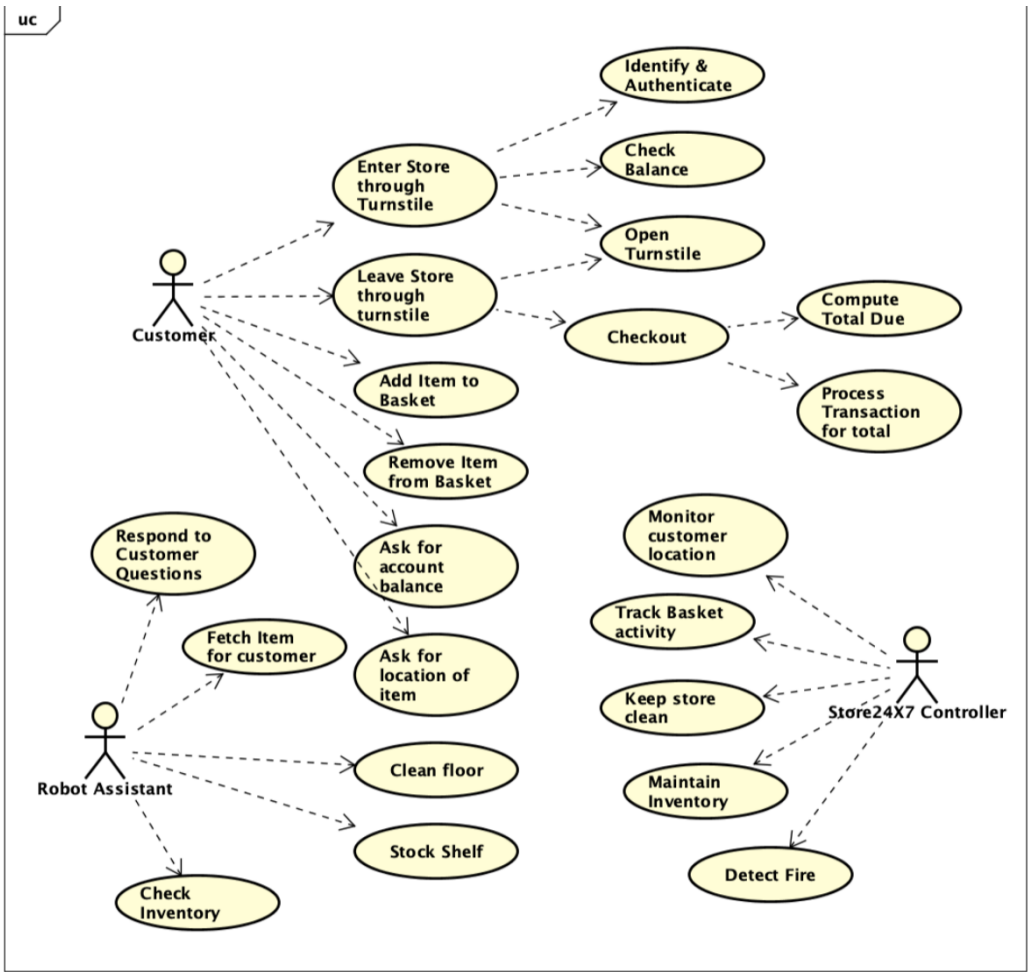
The Store Model Service is responsible for managing the domain entities of the Store24x7 System. Domain entities include the store, inventory, products, customers, baskets, turnstiles, robot assistants, tasks, and sensors. The Store Model Service provides an API for interacting with those objects. The API supports querying the state of the entities, as well as updating the state. The Model Service maintains the state of all domain objects.

Authentication Service

The Authentication Service manages the authentication of users and controls access to the store. The Entitlement Service first identifies the user through the face and/or voice recognition. Once identified, the Authentication Service is used to gate access to control appliances.

Use Cases:

The following Use Case diagram describes the use cases supported by the Store Model System.



Actors:

Actors of the Store Model System include Robot Assistants, the Store Controller and Customers.

Robot Assistants:
Responsible for stocking shelves, cleaning spills, checking inventory, fetching items for customers, and responding to customer questions.

Store Controller:
Responsible for monitoring customer location, tracking basket activity, keeping store clean, maintaining inventory, and detecting fires.

Customer:
Customers can enter the store through the turnstile, add items to their basket, remove items from their basket, ask for their account balance and ask for the location of an item.

Use Cases:

Fetch Product:

Microphone event, triggered when customer says “Please get me <number> of <product>”. The Controller passes the following command to the robot closest to where the product exists in inventory for the store, “fetch <number> of <product> from <aisle> and <shelf> and bring to customer <customer> in aisle <customer_location>”.

Missing Person:

Microphone event, triggered by customer phrase “can you help me find <customer_name>”. Locates the customer by name by iterating through all customers marked as currently in the store, and then announces via speaker in Store, “<customer_name> is in <aisle>.

Emergency:

Camera event, triggered when a fire, flood, earthquake, or armed intruder is detected. Controller opens all turnstiles, announces “There is a <emergency> in <aisle>, please leave <store> immediately”. Then commands a robot to “address <emergency> in <aisle>”, and commands remaining robots to “Assist customers leaving the <store>”.

Basket Event:

Camera event, triggered when a customer is spotted adding or removing products from a shelf. Controller adds/removes the product to/from the customer’s basket. Then performs the appropriate add/remove action on the shelf. Finally, if the stock is below a threshold of 50%, a robot is tasked with restocking the shelf.

Customer Seen:

Camera event, triggered when a customer is spotted in an aisle. Controller makes an API call to StoreModelService to update the customer's location to their current aisle.

Check Account Balance:

Microphone event, triggered when customer says "What is the total basket value?". Controller calls API to compute basket total. Then checks the user's account balance. Creates speaker event to instruct customer "total value of basket items is <value> which is (more | less) than your account balance of <balance>.

Assist Customer to Car:

Turnstile event, triggered when customer's total basket weight exceeds 10lbs. Controller commands a robot to assist the customer to their car.

Broken Glass:

Microphone event, triggered by the sound of breaking glass. Controller commands the closest robot to "clean up the broken class in <aisle>".

Enter Store:

Turnstile event, triggered when a customer is waiting to enter the turnstile. First the customer is looked up by id. Then their wallet balance is checked, a positive balance is required for store entry. If the customer's wallet balance is positive, they are assigned a basket, the turnstile is opened, and a welcome message is announced "Hello <customer_name>, welcome to <store name>!"

Cleaning Event:

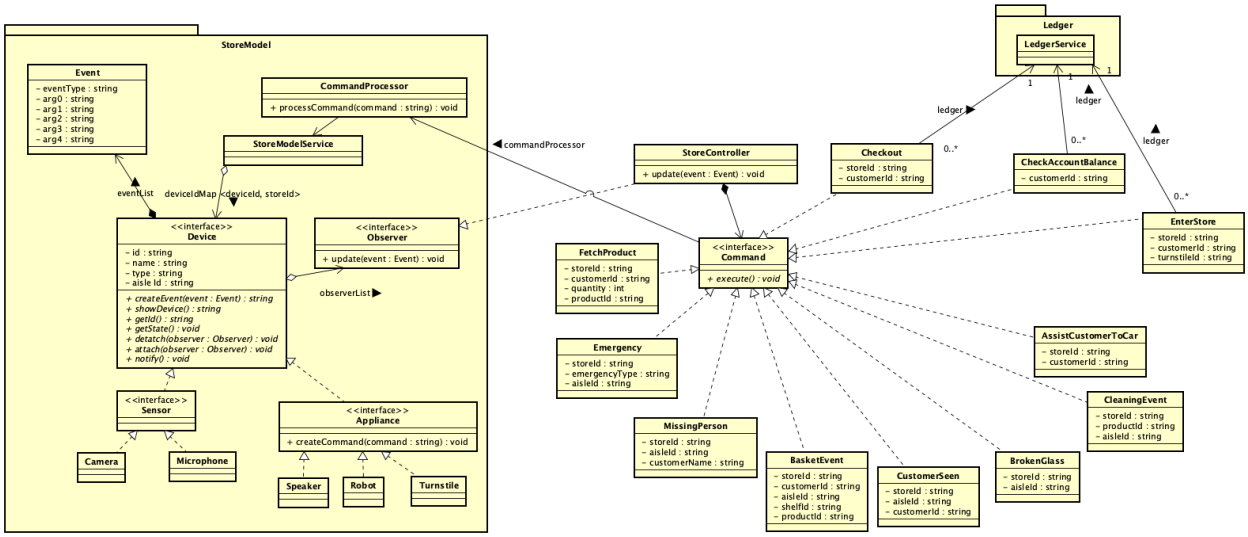
Camera event triggered when a product is spotted on the floor in an aisle of the store. Controller instructs a robot to "clean-up <product> in <aisle>".

Checkout:

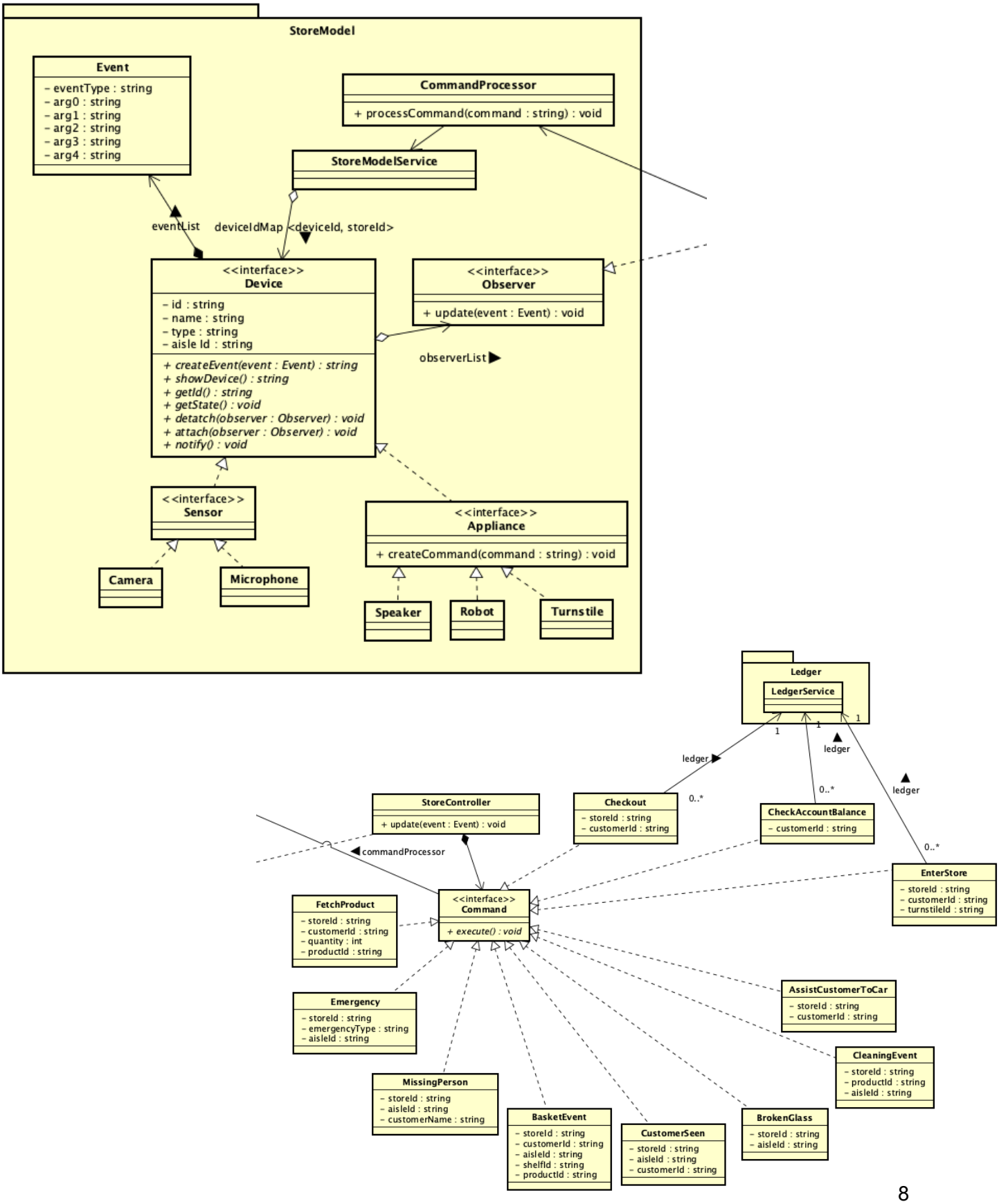
Turnstile event, triggered when customer approaches a turnstile. Controller first identifies the customer. Then computes the basket total. A transaction is created in the ledger, and the transaction is submitted to the blockchain. Turnstiles are opened, and a checkout message is announced "goodbye <customer_name>, thanks for shopping at <store_name>!"

Class Diagram:

The following class diagram defines the classes defined in the design.

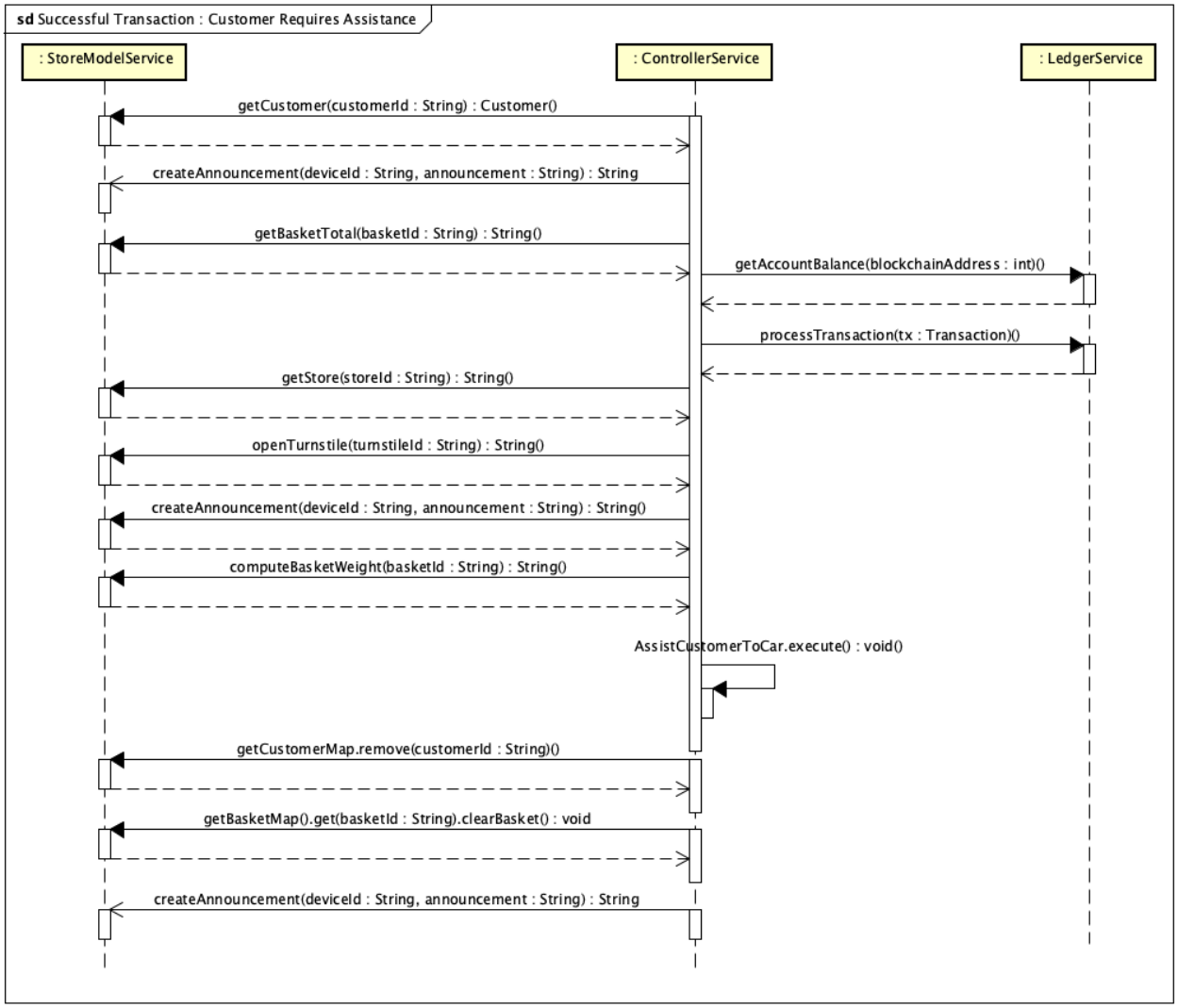


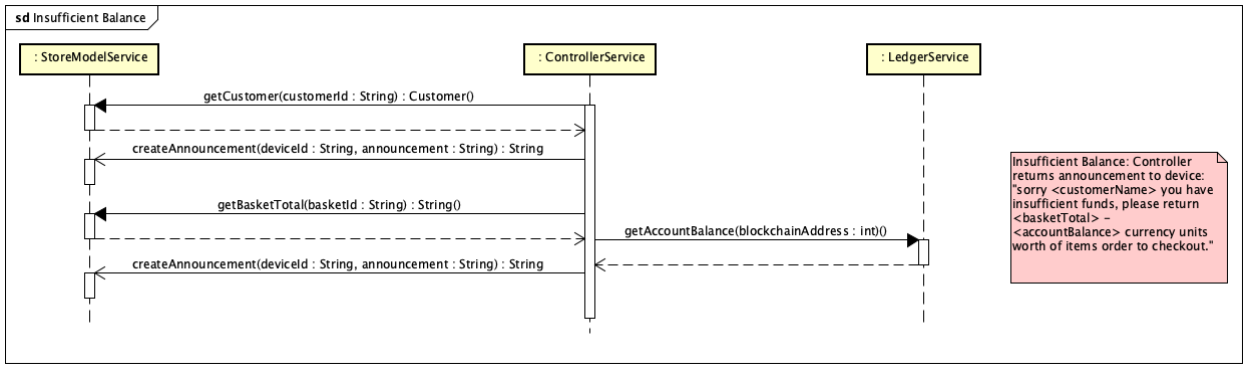
Class Diagram Enlarged:

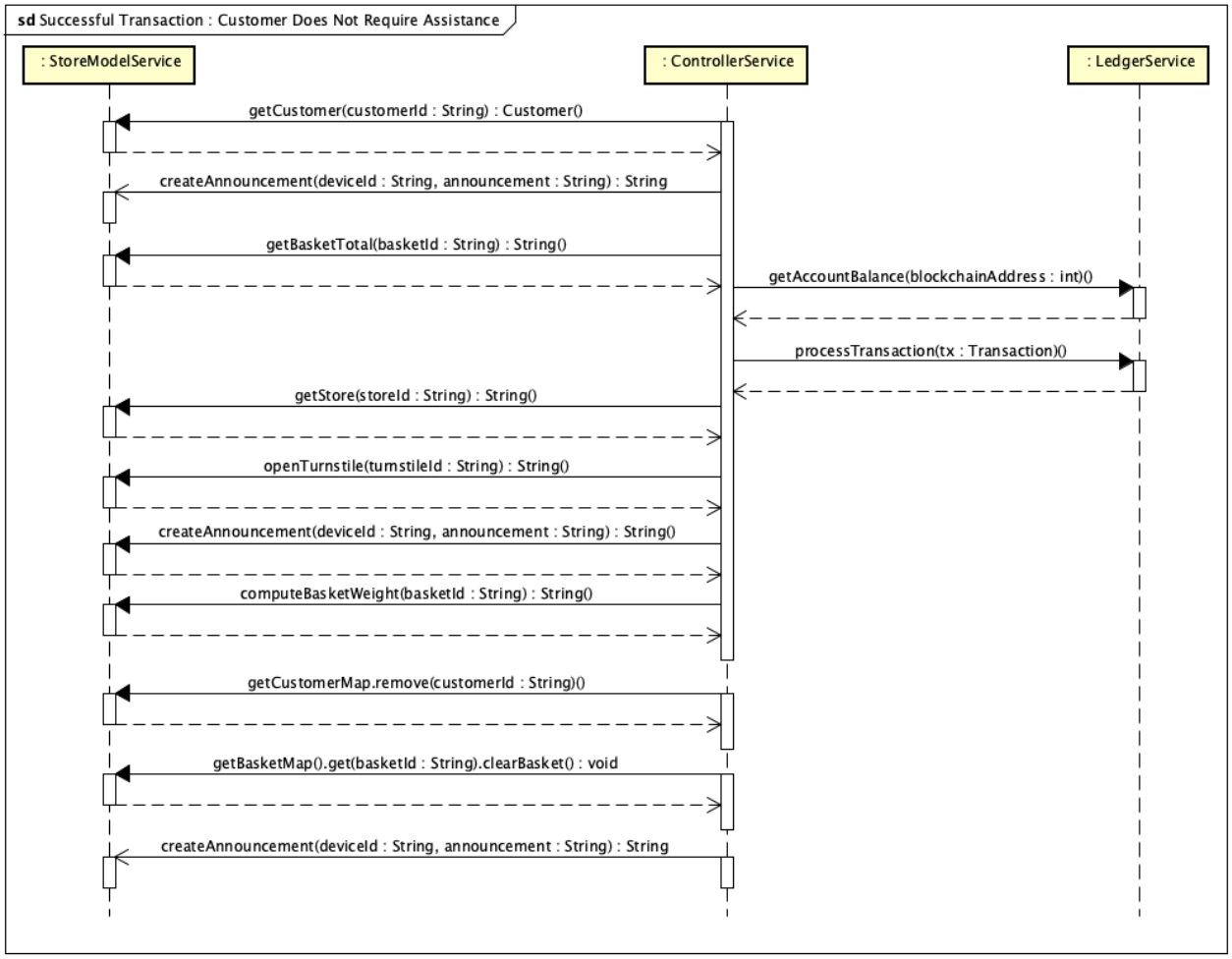


Sequence Diagrams:

The following diagram outlines the possible sequences that may occur when a customer attempts to checkout of the store.




sd Insufficient Balance
: StoreModelService
: ControllerService
: LedgerService
getCustomer(customerId : String) : Customer()
createAnnouncement(deviceId : String, announcement : String) : String
getBasketTotal(basketId : String) : String()
getAccountBalance(blockchainAddress : int)()
createAnnouncement(deviceId : String, announcement : String) : String
Insufficient Balance: Controller returns announcement to device: "sorry <customerName> you have insufficient funds, please return <basketTotal> - <accountBalance> currency units worth of items order to checkout."


sd Successful Transaction : Customer Does Not Require Assistance
: StoreModelService
: ControllerService
: LedgerService
getCustomer(customerId : String) : Customer()
createAnnouncement(deviceId : String, announcement : String) : String
getBasketTotal(basketId : String) : String()
getAccountBalance(blockchainAddress : int)()
processTransaction(tx : Transaction)()
getStore(storeId : String) : String()
openTurnstile(turnstileId : String) : String()
createAnnouncement(deviceId : String, announcement : String) : String()
computeBasketWeight(basketId : String) : String()
getCustomerMap.remove(customerId : String)()
getBasketMap().get(basketId : String).clearBasket() : void
createAnnouncement(deviceId : String, announcement : String) : String

Class Dictionary

This section specifies the class dictionary for the Store Model Service. Device and Observer are defined within the package “com.cscie97.store.model” The remaining classes should be defined within the package “com.cscie97.store.controller”.

Device	**Interface**	
Method Name	Signature	Description
createEvent()	(event : Event) : string	creates a new event to the specifications provided in the API interface, calls the notify method to update observers.
attach()	(observer : Observer) : void	Adds an observer to the observer list
detatch()	(observer : Observer) : void	Removes an observer from the observer list
notify()	() : void	Calls the updateEvent() method on all observers when a new event is created.
Association Name	Association Type	Description
eventList	Event	List of events, added as events are called. Documents executed events.
observerList	Observer	List of observers, all observers are notified when notify method is called.
Observer	**Interface**	

update()	(event : Event) : void	Retrieves the eventType of the passed event. Instantiates the appropriate command object, calls the execute() method on the new command.
StoreController	**Class**	Implements Observer Interface
Command	**Interface**	
Method Name	Signature	Description
execute()	() : void	This allows for encapsulation of each command's logic. The StoreController can execute a command's logic, knowing only its type.
FetchProduct	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where fetch is called
customerId	string	ID of customer requesting product
quantity	int	quantity of products that customer desires.
productId	string	ID of product that customer desires
Method Name	Signature	Description
execute()	() : void	Command robot to: fetch <quantity> of <productId> from <aisleId> and shelfId> and bring to customer <customerId> in aisle <customer_location>
MissingPerson	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where customer location is requested
customerName	string	Name of customer to be located
Method Name	Signature	Description

execute()	() : void	Locates customer by name, creates speaker event to announce "<customerName> is in aisle <aisle>"
Emergency	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where camera spotted emergency
aisleId	string	ID of aisle where camera spotted emergency
emergencyType	string	{fire, flood, earthquake, armed intruder}
Method Name	***Signature***	***Description***
execute()	() : void	Opens turnstiles, announces "There is a <emergency> in <aisle>, please leave <store> immediately", Sends command to robot "address <emergency> in <aisle>"
BasketEvent	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where basket event occured.
customerId	string	ID of customer who performed event
aisleId	string	ID of aisle where event was performed
shelfId	string	ID of shelf that was modified
productId	string	ID of product that was added or removed
Method Name	Signature	Description
execute()	() : void	Adds or removes product to or from the specified customer's basked. Adds or removes product from appropriate shelf, and commands robot to perform restock on specified shelf if product is below 50% stock.

CustomerSeen	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where customer was seen
aisleId	string	ID of aisle where customer was seen
customerId	string	ID of customer who was seen
Method Name	Signature	Description
execute()	() : void	Updates customer location
CheckAccountBalance	**Class**	**Implements Command**
Property Name	Type	Description
customerId	string	ID of customer who is asking for account balance check
Method Name	Signature	Description
execute()	() : void	Computes the value of items in the basket, checks the account balance for the customer, creates speaker event: "total value of basket items is <value> which is (more | less) than your account balance of <balance>
AssistCustomerToCar	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where customer needs assistance
customerId	string	ID of customer who needs assistance
Method Name	Signature	Description
execute()	() : void	Checks the weight of the basket, if the basket weighs over 10lbs, command a robot to assist the customer to their car
BrokenGlass	**Class**	**Implements Command**
Property Name	Type	Description

storeId	string	ID of store where glass broke
aisleId	string	ID of aisle where glass is broken
Method Name	Signature	Description
execute()	() : void	Commands robot to "clean-up broken glass in <aisle>"
EnterStore	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store that customer is trying to enter
customerId	string	ID of customer trying to enter
turnstileId	string	ID of turnstile that customer is waiting at
Method Name	Signature	Description
execute()	() : void	Lookup customer by id, check for positive account balance, assign the customer a basket, open the turnstile, send command to turnstile "Hello <customer_name>, welcome to <store name>!"
CleaningEvent	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where event occured
productId	string	ID of product on floor
aisleId	string	ID of aisle where product is on floor
Method Name	Signature	Description
execute()	() : void	Commands robot to "clean-up <product> in <aisle>"
Checkout	**Class**	**Implements Command**
Property Name	Type	Description
storeId	string	ID of store where checkout is requested
customerId	string	ID of customer checking out
Method Name	Signature	Description

execute()	() : void	Identifies customer by id, computes the total cost of the customer's basket, creates a transaction, submits the transaction to the blockchain, opens the turnstile, commands turnstile to send message "goodbye <customer_name>, thanks for shopping at <store_name>!"
Association Name	Association Type	Description
ledger	LedgerService	LedgerService association used to check customer's account balances, create transactions and commit transactions to the blockchain.
Event	**Class**	
Property Name	Type	Description
EventType	string	This attribute is queried to determine what type of event is being passed, the remaining object attributes are read accordingly.
arg0	string	Queried as either storeId, or customerId depending on type of event.
arg1	string	Optional value holding argument depending on event type
arg2	string	""
arg3	string	""
arg4	string	""

Design Details

The core component for the Store 24x7 System is the StoreModelService class. The StoreModelService provides an API for interacting with the Stores and implements the API methods that manage the Customers, Products, Devices, Baskets, Aisles, and Shelves that make up the Stores.

The Store Controller is responsible for responding to events that occur within the stores with actions. The Store Controller maintains a reference to the CommandProcessor class in order to give commands to the StoreModelService, as well as the Ledger service to access payment information. In future iterations, the Controller will interact with the StoreModelService using an API with token verification.

The Controller Service implements the Observer interface and acts as an observer to the Devices in stores. When a new event is triggered via the CLI, the device's observers are notified. The notification process consists of iterating through all of the device's observers and passing them the event. The Controller Service takes this event, categorizes it as a command object, instantiates the appropriate command, and executes the command's logic.

The Controller provides automation for the Sensor and Appliance behavior within all stores handled by the Store Model Service. The Controller can handle multiple Store Model Services if the need arises.

Testing

Implement a test driver class called TestDriver that implements a static main() method. The main() method should accept a single parameter, which is a command file. The main method will call the CommandProcessor.processCommandFile(file:string) method, passing in the name of the provided command file. The TestDriver class should be defined within the package "cscie97.store.test".

Three test CLI files will be included. They contain the API commands included in the "Assignment3_Script_Syntax.pdf" document.

Command to build program:

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

Command to run tests:

java -cp . com.cscie97.store.test.TestDriver store.script
java -cp . com.cscie97.store.test.TestDriver store_b.script
java -cp . com.cscie97.store.test.TestDriver store_c.script

Risks

The model currently has no authentication system. An authentication system should be implemented to control which methods users are allowed to call.

The in-memory implementation leaves the system vulnerable to losing store, customer, product, inventory, and device information. This should be corrected as soon as possible.

In the event of an emergency, customers baskets are all cleared, but products are not returned to their shelves. This is possibly a fair assumption to make, as if all turnstiles open shoppers are likely to walk out with a few inventory items, but it would be better to incorporate some more accurate accounting procedure.

Currently, if a customer removes 12 products from a shelf containing 10, a sensor error is printed, but there is no action taken to resolve the action. Adding a command to automatically email a sensor maintenance provider would be a nice step to mitigate store accuracy issues owing to sensor failure risk.