Store Controller Service Design Document

Date: 10/19/19 (revised 10/30/19)
Author: Matthew Thomas
Reviewer(s): Antony Gavidia, Trisha Singh

Introduction

This design specifies the implementation of a Store Controller Service, one component of the Store 24X7 Software System. The Store Controller Service is the “brains” behind the system and ties multiple components together, including the Ledger Service (assignment 1) and the Store Model Service (assignment 2). The Store Controller listens for events that take place in a Store located within the system and responds accordingly. A response can consist of one of many predefined actions to update the state of the store, respond to a customer query, or to command an appliance to perform a task. Responses are flexible and can be added to or modified by a Store 24X7 Administrator.

Overview

As the idea of a Store 24X7 System comes to fruition, a necessary component is a Controller Service to communicate with all the pieces of the system. The Controller solves the problem of management: what should happen in response to an event? What happens when a Customer enters a store and adds items to their basket, there is a spill in an Aisle, or there is an emergency that requires an evacuation? With the advent of the internet of things (IoT) technology, and a lack of human employees, the Store 24X7 System needs a way to monitor, process, and respond to a multitude of scenarios.

In earlier assignments the Store Model Service was designed to define and update a physical store: how the Store is laid out, what Products and Devices a Store contains, and how Inventory shifts from Shelf to Basket. The Ledger Service was designed as the payment processor, keeping track of Customer balances and a history of transactions made in the Store. The Store Controller Service brings both pieces together and fulfills the promise of the Store 24X7 System to be maintained autonomously.

For full automation, the Store 24X7 System needs a way to monitor physical stores and all events that occur within. To forgo human employees, and provide a cost benefit to the store owners, the Controller Service must be able to act on events according to a prescribed set of rules. In this way, it can replace the need for store managers to delegate responsibilities, shelf stockers to replace products, and janitors to clean up spills in an aisle. The Store Controller Service provides the system administrator with centralized control over the behavior of individual stores to enable this autonomous behavior.

Document Organization

This design document provides a list of requirements the Store Controller will need to address, and use case scenarios about how actors will interact with the service. This document also includes a class diagram and dictionary to assist a programmer with an implementation, and sequence diagrams to demonstrate how interactions between the Controller, Ledger, and Store Model Service take place. Furthermore, the design touches upon exception handling, testing, and known risks in the Store Controller that should be taken into consideration when the system is deployed.

Requirements

This section defines the requirements for the Store Controller.

Monitor Store Model Service Stimuli

1. Sensors and Appliances located within a Store detect events (controller requirements, page 1).
2. Events are broadcast from the Store Model Service to registered Observers.
3. Observers acknowledge notifications and respond if applicable.

Respond to Stimuli

1. The Controller parses an event message.
2. If an event is recognized by the Controller, a Command object is created.
3. Each Command contains related Store objects and a list of actions to take in response to the event (controller requirements, pages 2-4).
4. The Controller executes Commands once created.

Process Customer Transactions

1. Validate that a Customer is registered with the Store 24X7 System and has a positive account balance with the Ledger Service. If valid, the turnstile a Customer arrives at is opened to let them into the Store (controller requirements, page 5).
2. Calculate the total cost of items in a Customer's basket. Query the account balance and compare with the total cost. If valid, a Transaction is created and submitted to the Ledger Service and added to the blockchain (controller requirements page 1). The Customer is then allowed to leave the Store (controller requirements, page 4).

Not required

Authentication

To read and update the state of the Store via the public API requires that an authentication token be included in the request. This will be implemented as part of assignment 4. For now, the Controller does not need to deal with authentication tokens and authorized access; rather, the

Store Model Service includes an opaque String that was implemented in assignment 2 and does not affect the operations.

Persistence

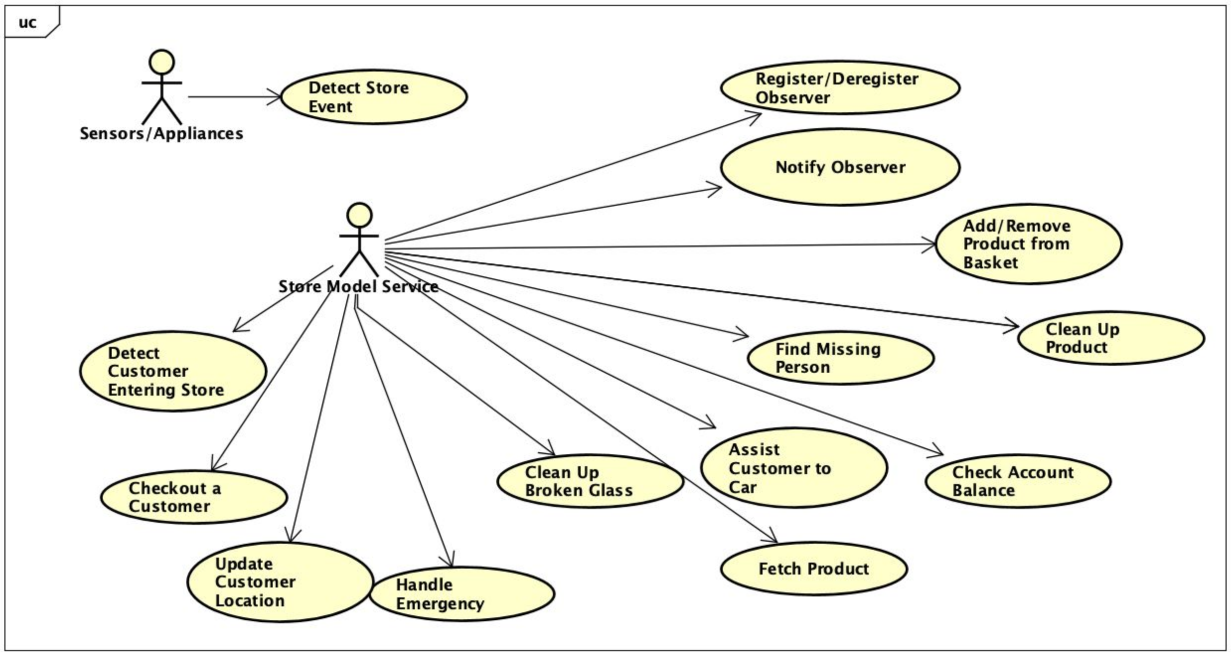
As in assignments 1 and 2, the Store Controller Service, all objects, and transactions are maintained in memory. Saving the state of the Store Model Service, Ledger Service, and the Controller Service is not required beyond keeping the objects in memory.

Handling Duplicate Events

In a physical store that contains multiple Sensors and Appliances, a given event may be detected by more than one Device. Each Device that detects a given stimulus would emit an event that is broadcast by the Store Model Service to all Observers; in this case the Store Controller Service. For the purposes of this assignment, it is assumed that the same event is only reported to the Controller Service once (as this is provided via the test scripts). For example, the Controller Service does not have to handle cameras A and B in both detecting and reporting intruder X. Intruder X will be simulated by a single command in the test script.

Use Cases

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



Actors

The actors of the Store Controller include Sensors/Appliances, Store Model Service, and the Store 24X7 Administrator.

Sensors/Appliances

The Sensors/Appliances are responsible for autonomously monitoring their location within a Store. These AI-powered devices (system architecture, page 1) detect all audio or visual stimuli present in their domain and report the presence of a stimulus through the Store Model Service.

Store Model Service

The Store Model Service is responsible for tracking all Devices located within Stores. The Store Model Service also maintains a list of registered Observers to notify when an event is detected by one of the Devices.

Use Cases

Detect Store Event

Physical Sensors and Appliances autonomously detect events that occur within a store. These physical devices then emit a software event that is broadcast by the Store Model Service to all registered Observers.

Register/Deregister Observer

Following the Observer pattern, the Store Model Service implements an interface that allows another object to register (sign up) or deregister (unsubscribe) from notifications.

Notify Observer

When an event is detected by the physical devices, the software generates a digital event that the Store Model Service broadcasts to all the Observers currently registered.

Add/Remove Product from Basket

When a Camera detects a Customer moving a Product to/from their basket, update the status of the Basket and Inventory as appropriate.

Assist Customer to Car

During checkout, if a Customer's basket is detected to weigh over 10 lbs, command a Robot to help them carry their to their car.

Check Account Balance

When a Customer asks a microphone for their balance, calculate their basket total, compare with their Ledger account, and report back the balance.

Checkout a Customer

When a Customer approaches a turnstile to leave, calculate their basket total and create a transaction to submit to the Ledger. If the transaction is approved, open the turnstile and emit a goodbye message.

Clean Up Broken Glass

When a microphone detects the sound of breaking class, dispatch a Robot to clean it up.

Clean Up Product

When a camera detects a Product on the floor, dispatch a Robot to clean it up.

Detect Customer Entering Store

When a Customer approaches a turnstile entering the Store, look up their account balance to ensure they have funds to cover a purchase. If they do, open the turnstile and welcome them into the store.

Fetch Product

When a Customer asks a store microphone for a Product, dispatch a Robot to fetch the product and bring it to the customer.

Find Missing Person

When a Customer asks a store microphone to locate another Customer, query the Store Model Service to find the Customer's location.

Handle Emergency

When a Camera detects an emergency situation, open all turnstiles, announce the emergency and task Robots to address the emergency and assist customers leaving the store.

Update Customer Location

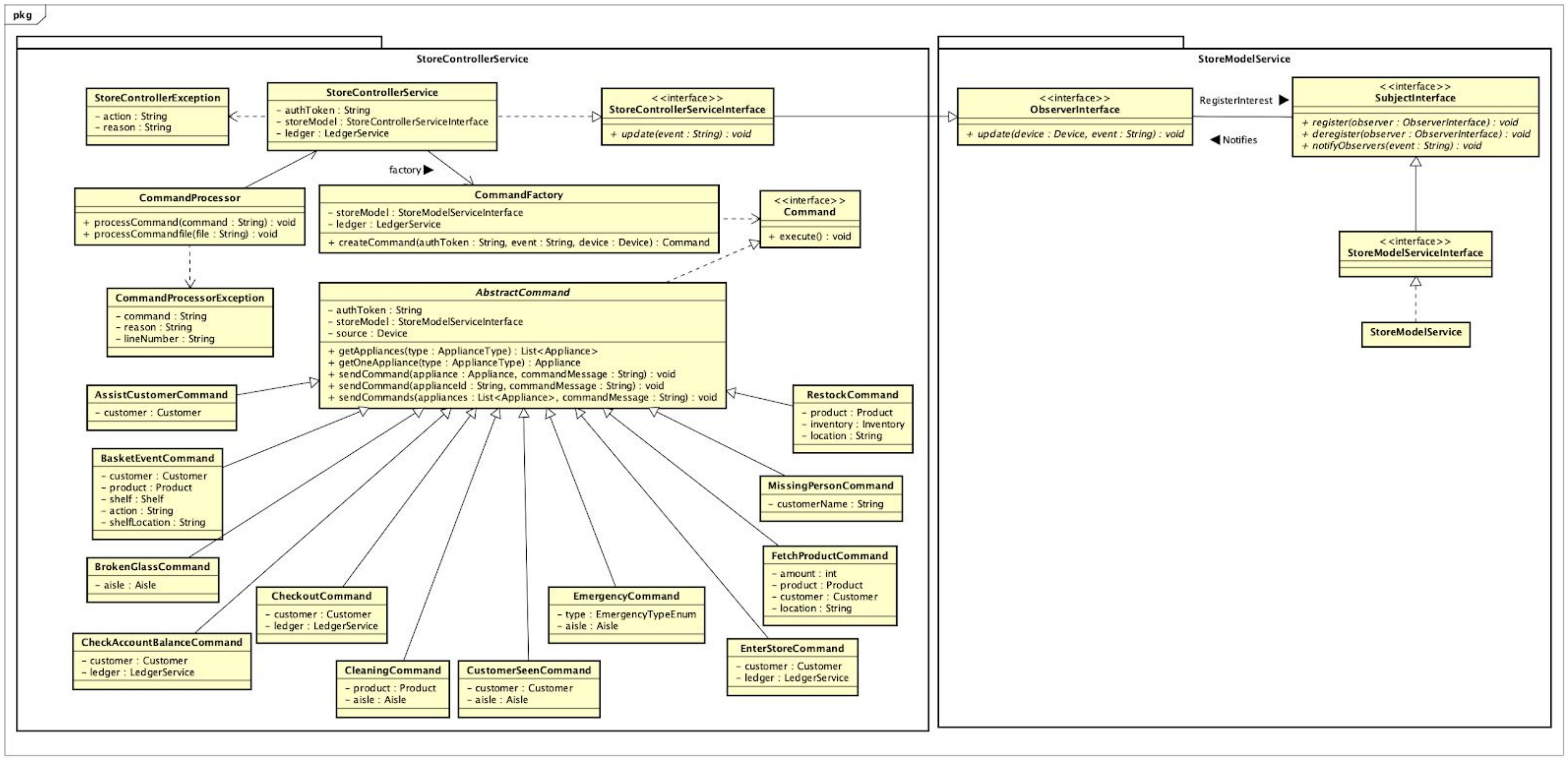
When a camera detects a Customer moving throughout the store, update their location in the Store Model Service representation.

Implementation

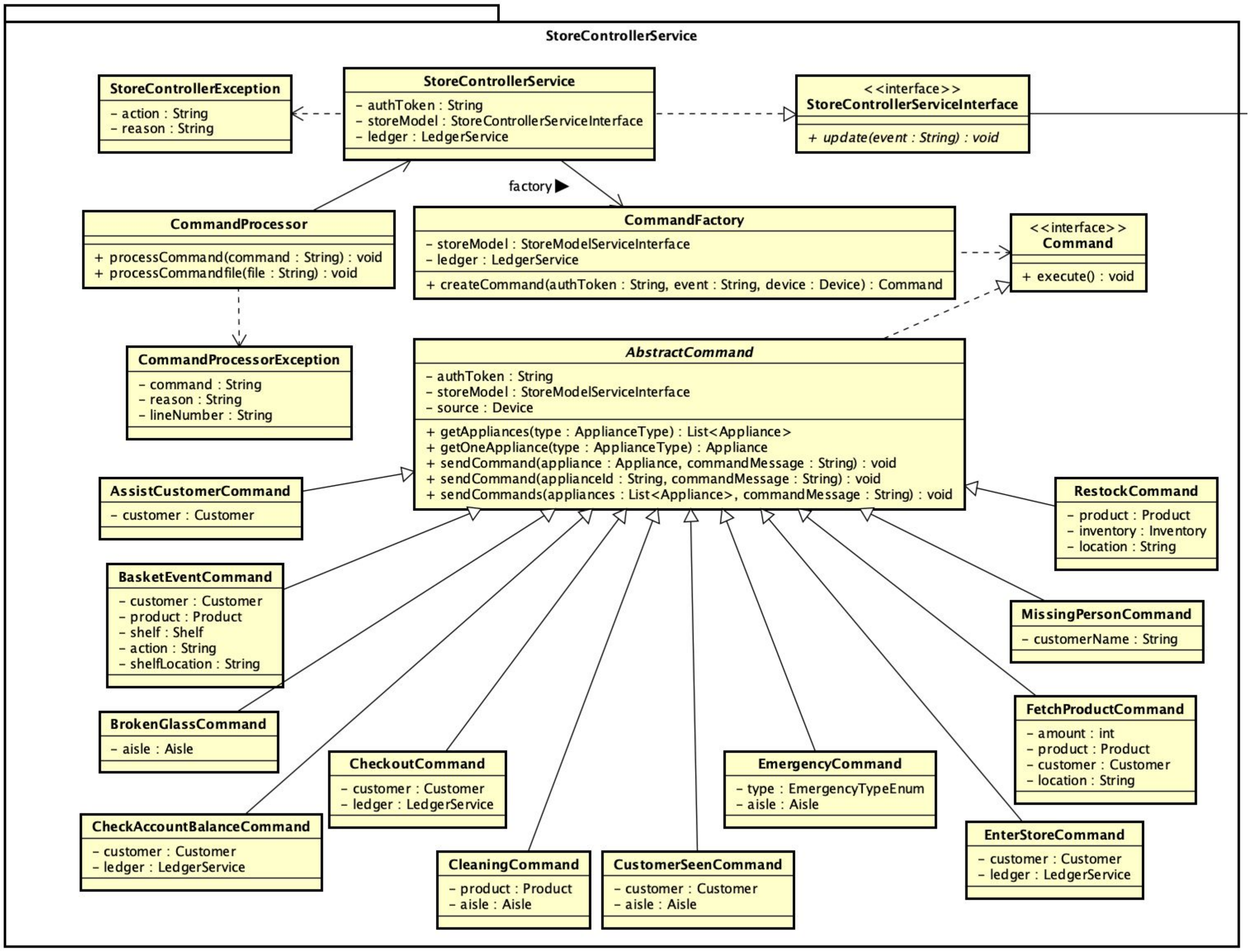
Class Diagram

The following class diagram shows the interaction between the Store Controller Service and Store Model Service packages to illustrate the Observer Pattern. The Ledger association is omitted for clarity (details on how the Ledger interacts with the Store Controller Service are outlined in the Implementation Details section below). Additional details about the Store Model Service can be found in the Store Model Service design document. Classes shown here are new classes implemented for assignment 3 to facilitate the Observer pattern.

Note: full-resolution images can be found in the zipped submission folder under the “Diagrams” directory.



The following class diagram defines the Store Controller Service implementation classes contained within the package "com.cscie97.store.controller".



Class Dictionary

This section specifies the class dictionary for the Store Controller Service. The classes are defined within the package "com.cscie97.store.controller".

StoreControllerServiceInterface

The main interface that is implemented by a concrete Store Controller Service. It follows the Observer pattern as described in *Head First Design*, Chapter 2. The StoreControllerService is notified of Store events when the Store Model Service (the Subject), calls the update() method in the Observer interface (see: Implementation Details section below for more details).

StoreControllerService

A concrete Store Controller Service to use with the CommandProcessor. This implements the StoreControllerServiceInterface (described above) and the Observer (see: Implementation Details section below) interfaces. When a concrete Store Controller Service is constructed, it stores associations to a Store Model Service, a Ledger, and a CommandFactory.

The Controller is responsible for parsing event notifications (Observer pattern) it receives and generating Command objects to perform a set of actions in response to the event. It creates the Commands through its CommandFactory association and executes them following the Command Pattern (as described in *Head First Design*, Chapter 6).

Associations

Association Name	**Type**	**Description**
storeModel	StoreModelServiceInterface	The Subject in the Observer pattern. The Controller keeps track of this object to either register/deregister from notifications.
ledger	Ledger	The Ledger which processes transactions and customer's Ledger accounts.
factory	CommandFactory	A Factory interface for generating Command objects to execute (see: *Head First Design*, Chapter 4 for a description of the Factory pattern).

Properties

Property Name	**Type**	**Description**
authToken	String	Authentication token to make requests to the StoreModelService; further details coming in assignment 4.

CommandFactory

A CommandFactory works in conjunction with a StoreModelServiceInterface and a Ledger to parse Store events and generate Commands to run sets of actions. The Factory is run in a StoreControllerService and is invoked anytime the Controller (an Observer) is notified of an event by the Store Model (Subject).

Methods

Method Name	**Signature**	**Description**

createCommand	(authToken:String, event:String, device:Device) : Command	Parse and event and create a corresponding Command. The event is checked against a list of known events, and if recognized, a private helper method is called to retrieve necessary store objects from the event parameters (see: Command syntax section).

Properties

Property Name	Type	Description
storeModel	StoreModelServiceInterface	The Store Model that provides a public API to call to retrieve and update state in Store objects.
ledger	Ledger	The Ledger which processes transactions and customer's Ledger accounts.

Command

The main interface that is used to implement the Command Pattern (as described in *Head First Design*, Chapter 6). A Command is created by the Store Controller Service (through a CommandFactory) in response to being notified of a Store event. Once the Command is created, its execute() method is called to run the set of actions defined within.

Methods

Method Name	Signature	Description
execute	() : void	Runs the set of actions that are defined within the Command class. The actions vary depending on which subclass of Command is created.

AbstractCommand

An abstract command class that defines properties and methods for all Command classes. Each command contains an authentication token, a Store Model Service, and the source Device. Each command comes with methods to retrieve Appliances and send command(s) to Appliance(s). The getter methods are described here to highlight the differences between getAppliances() and getOneAppliance().

Methods

Method Name	Signature	Description
getAppliances	(type:ApplianceType):List<	Retrieve a list of all Appliances of a given

	Appliance>	type, located within the Store where the event occurred (as determined by the source Device).
getOneApplianc e	(type:ApplianceType):Appli ance	Retrieve a single Appliance of a given type, located within the Store where the event occurred (as determined by the source Device).
sendCommand	(appliance:Appliance, commandMessage:String): void	Overloaded method. Send a command to a specified Appliance via the Store Model Service API. Use this method when the Appliance object is present.
sendCommand	(applianceId:String, commandMessage:String): void	Overloaded method. Send a command to a list of Appliances via the Store Model Service API. Use this method when only the ID of the Appliance is known.
sendCommands	(appliances:List<Appliance >, commandMessage:String): void	Send a command to a list of Appliances via the Store Model Service API.

Properties

Property Name	**Type**	**Description**
authToken	String	Token to authenticate with StoreModel API.
storeModel	StoreModelServiceInterface	The Store Model Service which provides the API to interface with.
source	Device	The Device which detected the event.

AssistCustomerCommand

An AssistCustomerCommand is created by a Turnstile during the CheckoutCommand process if the weight of Products in the basket exceeds a certain limit, in this case, 10 lbs (controller requirements, page 3). When triggered, the first available Robot is commanded to assist the Customer bring the items to their car.

Properties

Property Name	**Type**	**Description**
customer	Customer	The Customer who is checking out of the Store.

BasketEventCommand

A BasketEventCommand is created when a Camera detects that a Customer moves a product to/from their basket. This command then updates the state of the StoreModelService to increment/decrement the Customer Basket and Shelf Inventory. After the update is performed, a Robot is commanded to add more of the Product to the Shelf. Cameras detect movement of a single object at a time. Should the Customer remove two of the same Products from a Shelf, two BasketEventCommands would be created and executed.

Properties

Property Name	**Type**	**Description**
customer	Customer	The Customer who was detected moving a Product.
product	Product	The Product that was moved by the Customer.
shelf	Shelf	The Shelf where the Product is located.
action	String	The action the customer took (either ‘adds’ or ‘removes’).
shelfLocation	String	The fully qualified shelf location (of form <store>:<aisle>:<shelf>

BrokenGlassCommand

A BrokenGlassCommand is created when a Microphone detects the sound of glass breaking. A Robot is commanded to the Aisle where the glass broke.

Properties

Property Name	**Type**	**Description**
aisle	Aisle	The Aisle where the broken glass was detected.

CheckAccountBalanceCommand

A CheckAccountBalanceCommand is created when a Customer asks a Microphone for their basket’s current value. The Controller Service then calculates the Basket’s cost, queries the Ledger Service for the Customer’s account balance, and tells a Speaker to report back the result.

Properties

Property Name	Type	Description
ledger	Ledger	The Ledger which stores Account information.
customer	Customer	The Customer who asked the question.

CheckoutCommand

A CheckoutCommand is created by a Turnstile when a Customer approaches on the way out of the Store. When the customer approaches, the total cost of the basket is calculated, a Transaction is created and then submitted to the Ledger Service to be added to the blockchain. Once processed, and the Transaction was validated, the Turnstile opens and the closest Speaker emits a goodbye message.

Properties

Property Name	Type	Description
ledger	Ledger	The Ledger which stores Account information and processes transactions.
customer	Customer	The Customer leaving the Store.

CleaningCommand

A CleaningCommand is created when a Camera detects Product on the floor of an Aisle. When detected, the first available Robot is commanded to clean up the Product.

Properties

Property Name	Type	Description
product	Product	The Product that was spilled in the Aisle.
aisle	Aisle	The Aisle the mess is located in.

CustomerSeenCommand

A CustomerSeenCommand is created when a Camera detects that a Customer has moved in the Store to a new Aisle.

Properties

Property Name	Type	Description
customer	Customer	The Customer that was detected.
aisle	Aisle	The Aisle the Customer is seen in.

EmergencyCommand

An EmergencyCommand is created when a Camera detects an emergency in a store Aisle. When execute() is called, the Command opens all turnstiles, announces the emergency through all speakers in the Store, selects the first unoccupied Robot to send to the Aisle in which the emergency was detected, and commands all other Robots to assist Customers.

Properties

Property Name	Type	Description
emergency	EmergencyTypeEnum	The type of emergency that is detected. Options include: fire, flood, earthquake, armed intruder.
aisle	Aisle	The Aisle where the emergency was detected.

EnterStoreCommand

An EnterStoreCommand is created by a Turnstile when a Customer approaches on the way in to a Store. When detected, the Store Controller Service looks up the Customer in the Store Model Service and checks their account balance in the Ledger Service. If the Customer is registered and has a positive balance, a Basket is assigned to the Customer, the Turnstile opens, and the closest Speaker emits a welcome message.

Properties

Property Name	Type	Description
customer	Customer	The Customer entering the Store.
ledger	Ledger	The Ledger which tracks Accounts.

FetchProductCommand

A FetchProductCommand is created when a Customer asks a Microphone for some amount of a Product. When created, the first available Robot is tasked to go to the Aisle where the Product is located, remove some number of the Product from the Inventory, and bring it to the Customer in the Aisle where it was asked for, and add it to their Basket.

Properties

Property Name	Type	Description
amount	int	The amount of the Product to retrieve.
product	Product	The Product to fetch.

customer	Customer	The Customer who made the Product request.
location	String	The fully qualified Inventory location (store:aisle:shelf:inventory).

MissingPersonCommand

A MissingPersonCommand is created when a Customer asks a Microphone to locate a Customer. When the query is asked, the Store Controller searches the Store Model Service for the Customer’s location and responds through the closest Speaker in the Store to the Customer who asked.

Properties

Property Name	Type	Description
customerName	String	The name of the Customer to look for. This can either be the customer ID, or the Customer’s full name — first and last name, separated by a space.

RestockCommand

A RestockCommand is created after a BasketEventCommand if the Inventory where a Product was removed contains less than half of its capacity. If the Inventory is at less than half of its capacity when a Product is removed, the BasketEventCommand will trigger a Store Model Event that the Inventory needs to be restocked, back to its full capacity.

The RestockCommand is not specified in the Store Controller requirements, but follows logically from a BasketEventCommand. By separating out this action, restocking an inventory now becomes independent from a BasketEvent and can be performed at any time (e.g. at midnight, when there are few Customers in the store, cameras can survey inventory and see if any are approaching half of their capacity and preemptively restock shelves) This keeps with the Store Controller requirements as the BasketEventCommand will include a RestockCommand.

Properties

Property Name	Type	Description
product	Product	The Product to restock the Inventory with.
inventory	Inventory	The Inventory that needs to be restocked.
location	String	The fully qualified Inventory location. Takes the form <store>:<aisle>:<shelf>:<inventory>.

Event Syntax

Physical devices generate events by constructing Strings according to the following syntax: Events are received by the Store Model Service, which then notifies all Observers of the event. All commands follow the general syntax outlined in the Store Model Service requirements (page 9):

```
create event <device_id> event <event>
```

The following describe how the <event> portion of the syntax is formatted for each of the specified commands, as outlined in the Store Controller requirements (pages 2-4). Refer to the Use Case section for what actions each Command performs.

AssistCustomerCommand

```
customer <customer> assistance
```

BasketEventCommand

```
customer <customer> (adds | removes) <product> from <aisle:shelf>
```

BrokenGlassCommand

```
sound of breaking glass in <aisle>
```

CheckAccountBalanceCommand

```
customer <customer> says 'What is the total basket value?'
```

CheckoutCommand

```
customer <customer> approaches turnstile
```

CleaningCommand

```
product <product> on floor <store:aisle>
```

CustomerSeenCommand

```
customer <customer> enters <aisle>
```

EmergencyCommand

```
emergency (fire | flood | earthquake | armed intruder) in <aisle>
```

EnterStoreCommand

```
customer <customer> waiting to enter at the turnstile <turnstile>
```

FetchProductCommand

```
customer <customer> says please get me <number> of <product>
```

MissingPersonCommand

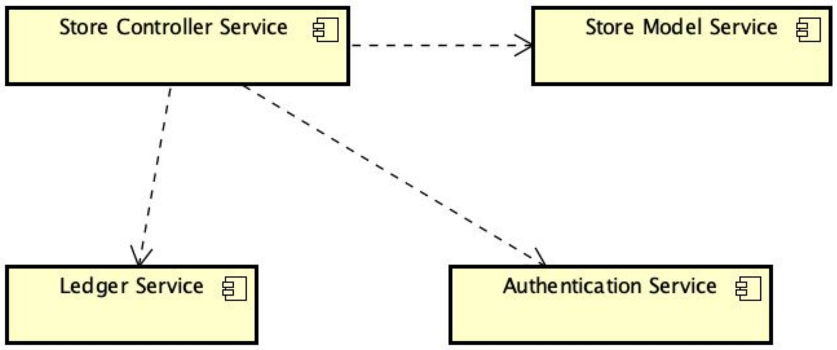
```
can you help me find <customer name>
```

RestockCommand

```
product <product> inventory <inventory> restock
```

Implementation Details

The StoreControllerService fits into the larger Store 24X7 System and integrates with two other components — the StoreModelService and Ledger (an Authentication Service is forthcoming in assignment 4). The following diagram from the System Architecture document (page 4) shows how the different services connect:



For simplicity, the Store Controller Service class diagram above leaves out the dependency between the Store Controller Service and the Ledger Service. This relationship can still be inferred through the diagram via the ‘ledger’ property which references a Ledger Service.

For more information about the StoreModelServiceInterface, and associated classes, reference the Store Model Service design document created for assignment 2.

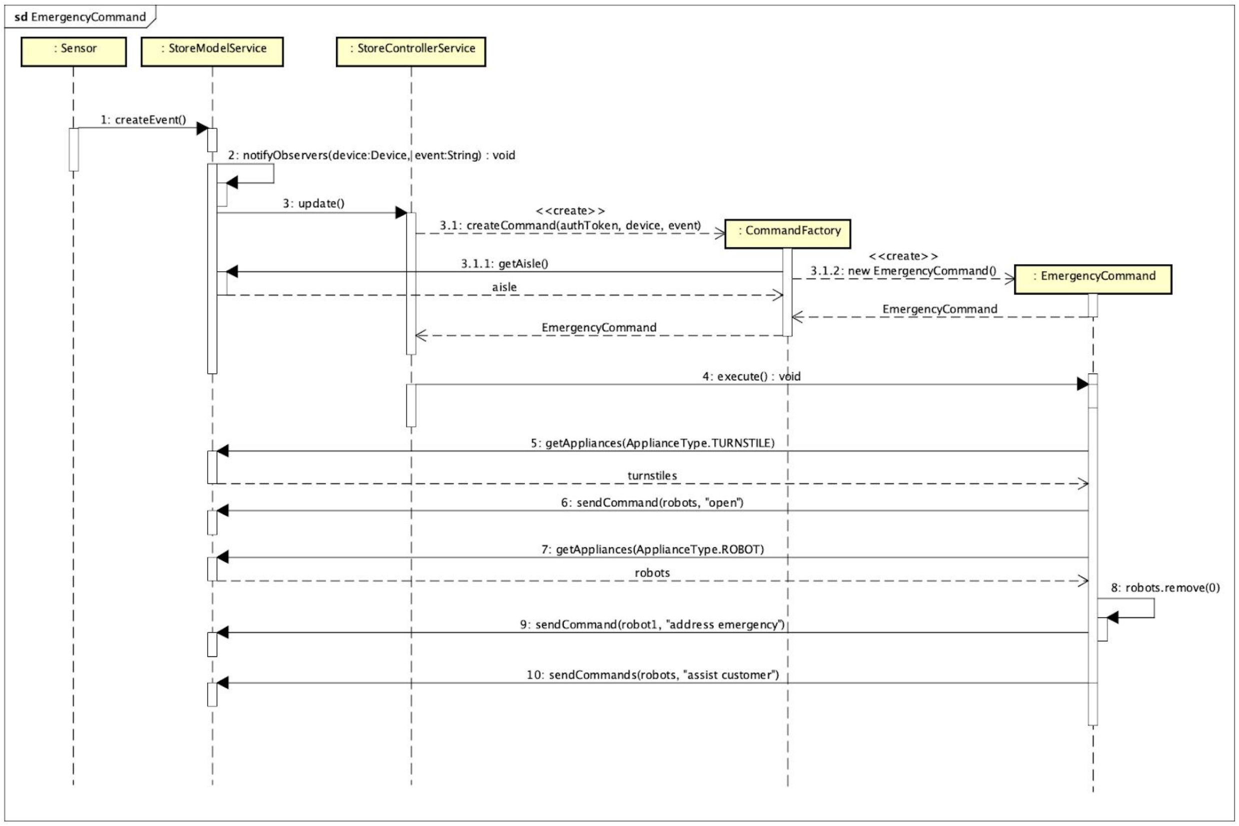
While no Class Dictionary is provided for the Observer and Subject classes in this document (as these are part of the Store Model Service), the Store Controller does implement the Observer interface, following the Observer Pattern (as described in *Head First Design*, Chapter 2). The Observer can register (or deregister) with a Subject, in this case a Store Model Service, to be notified of any events. The Observer contains a single method, update(), which is called by the Subject whenever a notification occurs. A Subject will notify all registered Observers when notifyObservers() is invoked, but in this implementation, the only Observer is the Store Controller.

The Store Controller Service in turn utilizes the Command Pattern (as described in *Head First*

Design, Chapter 6), as mentioned in the requirements (page 2). The Command Pattern describes creating Command objects that contain a single execute() method to perform a predefined set of actions. These commands and actions are outlined in the Controller requirements (page 2-4) and further specifics are described in this design document.

As a further abstraction, this design document also outlines how the Store Controller utilizes the Factory Pattern (as described in *Head First Design*, Chapter 4). By abstracting Command generation away from the Controller, the only responsibilities that remain are registering with a Subject (Store Model Service), passing the Event it receives to the Factory, and then calling execute() on the Command that is created. This allows for easy further expansion by creating more Command classes with no additional work needed on the part of the Controller.

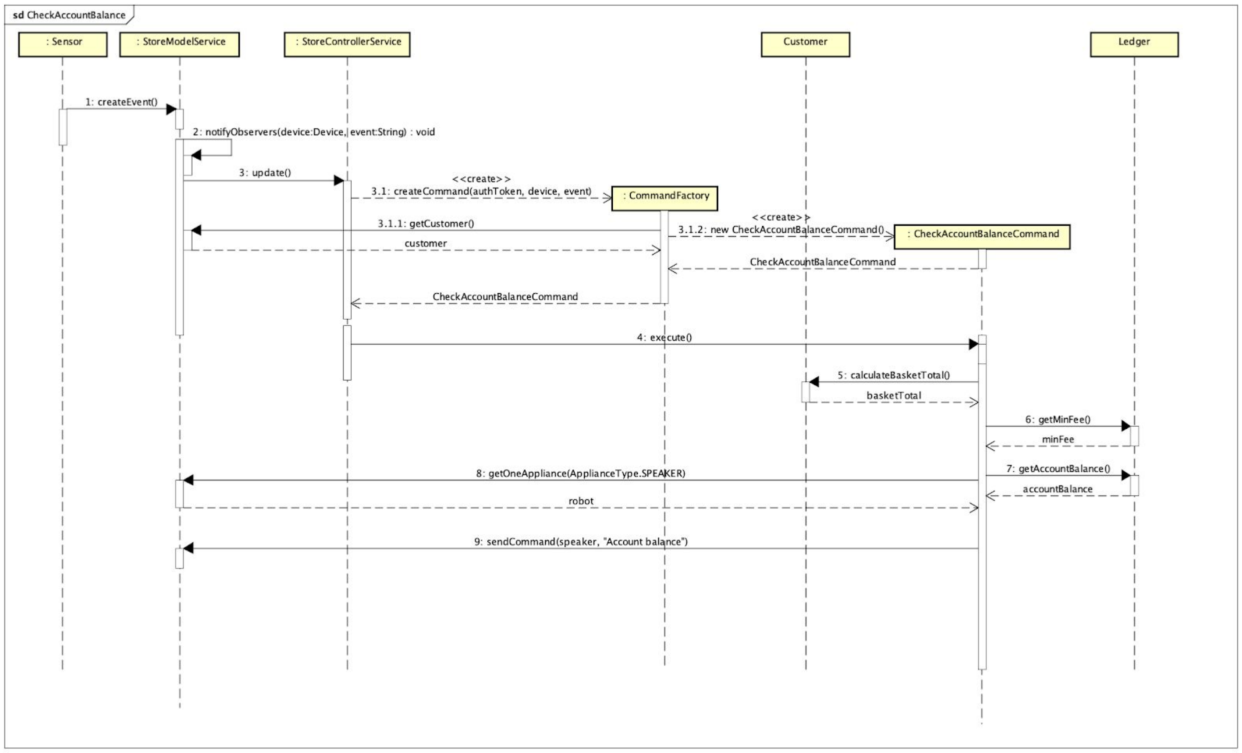
These steps are outlined in the following sequence diagrams which describe an EmergencyCommand and a CheckAccountBalanceCommand. (Note: full-resolution images can be found in the zipped submission folder under the “Diagrams” directory.)


sd EmergencyCommand
: Sensor
: StoreModelService
: StoreControllerService
1: createEvent()
2: notifyObservers(device:Device, event:String) : void
3: update()
<<create>>
3.1: createCommand(authToken, device, event)
: CommandFactory
<<create>>
3.1.1: getAisle()
3.1.2: new EmergencyCommand()
: EmergencyCommand
aisle
EmergencyCommand
EmergencyCommand
4: execute() : void
5: getAppliances(ApplianceType.TURNSTILE)
turnstiles
6: sendCommand(robots, "open")
7: getAppliances(ApplianceType.ROBOT)
robots
8: robots.remove(0)
9: sendCommand(robot1, "address emergency")
10: sendCommands(robots, "assist customer")

This sequence diagram walks through the steps of handling an EmergencyCommand request. The diagram depicts a Device invoking createEvent() as a way to simulate a physical device. In practice, this event is provided to the StoreModelService via the test script. When the Store Model receives the event, it notifies all Observers by calling the update() method. In this instance, the StoreControllerService is the only Observer.

The StoreControllerService then passes the event off to the CommandFactory which parses the event and detects it is an EmergencyCommand. Before instantiating a Command object, it retrieves an Aisle from the StoreModelService through the public API getAisle() method. An EmergencyCommand object is created and returned back to the StoreControllerService which then calls execute() on the Command.

The EmergencyCommand then communicates with the StoreModelService to retrieve lists of Turnstiles and Robots. Commands are constructed and sent to the Turnstiles and Robots in accordance with the set of actions described in the controller requirements (page 2) and this design document.


sd CheckAccountBalance
: Sensor
: StoreModelService
: StoreControllerService
Customer
Ledger
1: createEvent()
2: notifyObservers(device:Device, event:String) : void
3: update()
<<create>>
3.1: createCommand(authToken, device, event)
: CommandFactory
3.1.1: getCustomer()
customer
<<create>>
3.1.2: new CheckAccountBalanceCommand()
: CheckAccountBalanceCommand
CheckAccountBalanceCommand
CheckAccountBalanceCommand
4: execute()
5: calculateBasketTotal()
basketTotal
6: getMinFee()
minFee
7: getAccountBalance()
accountBalance
8: getOneAppliance(ApplianceType.SPEAKER)
robot
9: sendCommand(speaker, "Account balance")

This sequence diagram walks through the steps of handling an CheckAccountBalance request.

Steps 1-4 in the diagram are mostly the same as the EmergencyCommand diagram above. Step 3 differs slightly in that a Customer object is queried and returned from the StoreModelService instead of an Aisle.

During execution, the Command refers to the Customer to calculate the basket total. The Command then calls two public Ledger methods, getMinFee() and getAccountBalance() to compare to the basket total. A message is composed to send to the Customer via a Speaker command.

Exception Handling

StoreControllerServiceException

The StoreControllerServiceException is returned from the Store Controller Service in response to an error condition. The StoreControllerServiceException captures the action that was attempted and the reason for the failure.

The main condition where this Exception will be thrown is if the “event command” passed through via the Subject, is not recognized by the Controller. In other cases, where an “event command” is recognized but one or more components is not recognized or does not exist, a StoreModelServiceException or LedgerException, will be thrown.

Properties

Property Name	**Type**	**Description**
action	String	Action that was performed (e.g., “emergency event…”).
reason	String	Reason for exception (e.g. “That event is not supported.”).

CommandProcessorException

The CommandProcessorException is returned from the CommandProcessor methods in response to an error condition. The CommandProcessorException captures the command that was attempted and the reason for the failure. In the case where commands are read from a file, the line number of the command should be included in the exception. (Note: this was implemented in assignment 1 and is copied verbatim for assignments 2 and 3.)

Properties

Property Name	**Type**	**Description**
command	String	Command that was performed (e.g. “emergency event”).
reason	String	Reason for exception (e.g. “”).
lineNumber	int	Line number of the command in the input file.

Testing

A TestDriver class will be defined within the package “com.cscie97.store.test” and will handle reading in a test script (controller.script and others are provided), and handing off instructions to the CommandProcessor. The CommandProcessor instantiates a StoreModelService and

Ledger Service and passes these into the constructor for a StoreControllerService. The CommandProcessor accepts input relating to all StoreModelService and Ledger Service public API methods to load a pre-defined store configuration and account information.

Functional testing is performed by validating that executed commands output correct results (e.g., If a “cleaning event” is issued, a Robot responds and notes the ‘product’ and ‘aisle’ that needs to be cleaned). Negative testing is also performed to validate that the Controller Service ignores unrecognized commands as well as if invalid data is passed through the command (e.g., If a ‘missing person’ event is issued but customer ‘foobar’ does not exist, an error message should be logged).

Performance testing is not handled for this assignment as we do not have all the tools to handle doing so. As noted in the design document for assignment 2, performance testing could be achieved by, “writing a long test script or adjusting the CommandProcessor to accept multiple script files in succession to achieve a kind of scale we might anticipate in production.” However, for the purposes of this assignment, provisioning a Store and a small number of related store entities, Customers, and Ledger Accounts will be sufficient to test the requirements outlined and ensure the Store Controller Service works as intended.

For regression testing, since parts of the Store Model Service are updated to incorporate the Observer pattern, the same test script that was submitted in assignment 2 should be run again with the output compared to the previous implementation. Any differences or variations that are detected should be addressed and modified to produce identical output. Since the Ledger Service is not being modified, and instead is simply utilized by the Controller Service, regression testing does not need to be performed in this case.

Exception handling is described in its own section above.

Risks

As in the first two assignments, security is a big risk that should be taken into account. While assignment 2 incorporated an authentication token into API calls to the Store Model Service, the Controller Service has not yet incorporated authentication. That will be forthcoming in assignment 4. For now, the Store 24X7 System is open to attacks by hackers or other unauthorized users, as anyone is allowed to make a call to the public API to access the state of the store or account balances, and to update store objects and Ledger accounts.

Another recurring risk is that of losing store layout and transaction information. Since the software is maintained in memory and not written to disk, the Store Controller Service and its subcomponents are liable to data loss should the application stop running due to power loss, user error, or other unforeseen circumstances. Future designs and implementations should incorporate methods to save Store and Account states.

A new risk presented in the Store Controller Service is that of duplicate events. Handling duplicate events is not a requirement for this design, but it still poses a risk to the operation of the Store. As described above, since multiple Sensors and Appliances could track the same events occuring in the Store, this could lead to a Product being added to a customer's basket multiple times, among other scenarios. To address this, the Store Controller Service should implement de-duplication of events and detect whether or not a Device has already reported on the same stimulus happening within the store.