Store Controller Service Design Document

Date: 10/21/23
Author: Eric Margolis
Reviewer(s): Hunter Ward

Table of Contents

Introduction

This document describes the Store Controller Service, which is the autonomous control center of the Store 24X7 system.

Overview:

The goal of the 24X7 Store is to provide an automated shopping experience for customers, saving customers time and reducing the operational costs of the grocery store model by minimizing the amount of human labor required to run the stores. Store appliances such as robots, speakers, and turnstiles are used in place of human staffers to perform the actions required to maintain the store in an operational state. The Store Controller Service will direct the appliances using rule-based logic in response to events that occur within the stores, allowing the 24X7 system to run autonomously.

Requirements

- The Store Model Service and Controller Service must implement the Observer pattern to allow the Model Service to provide updates to which the Controller dictates a response.
- The Controller must follow a rule-based response system.
- The Controller must be able to interact with the Blockchain Ledger through the Ledger API.
- The system must scale to an arbitrary number of stores. Location information must be provided in event reports and command creation.

- Logging should be performed when an event, command, transaction, or response action is carried out.

Use Cases

Store Event

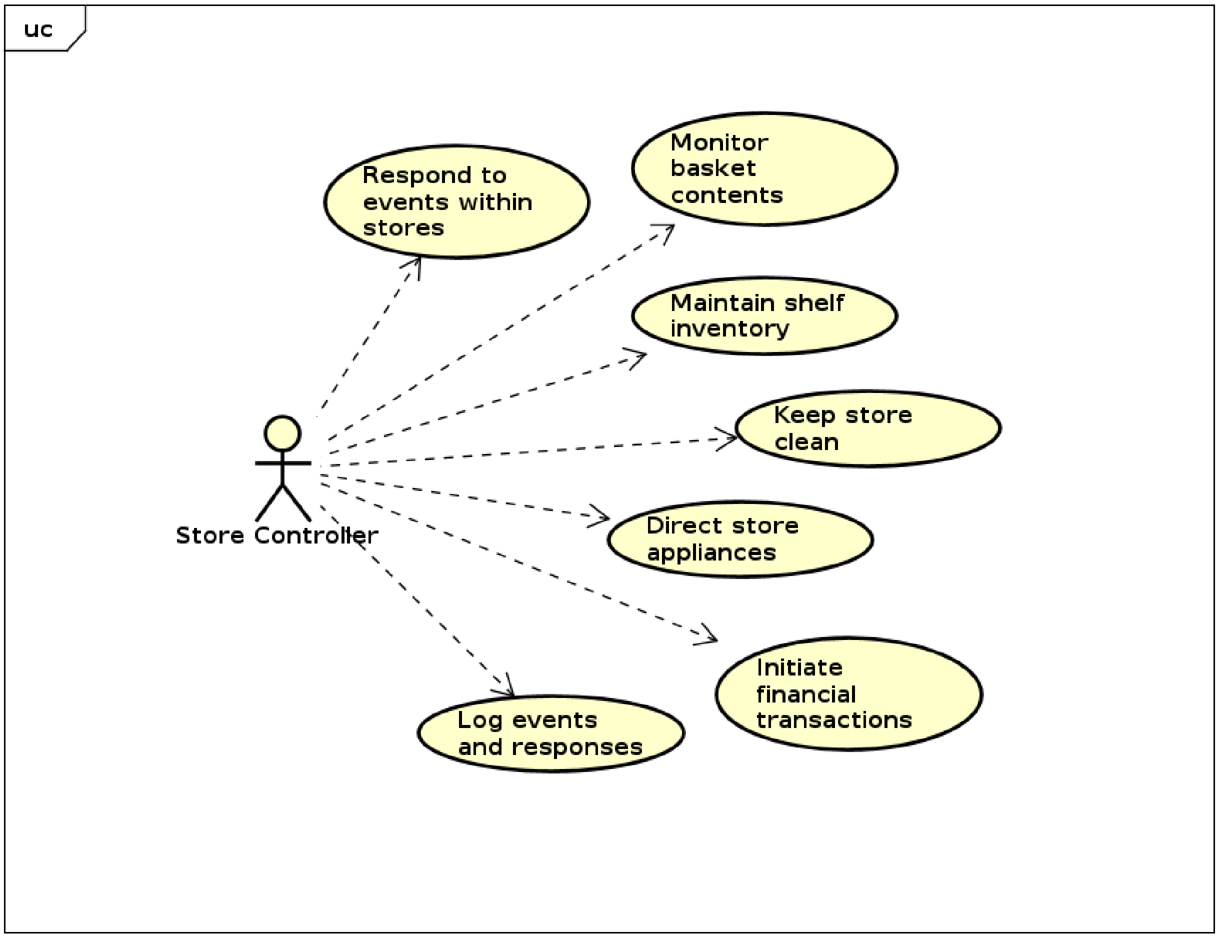
Main Success Scenario

1. An event occurs within a store
2. A sensor or appliance detects the event
3. The Store Model Service notifies its observers
4. The Store Controller Service is notified of the event type and location
5. The Controller decides how to respond to the event
6. The Controller creates a command
7. The Controller executes the command and logs it
8. The command is issued via the Store Model Service API
9. The Store Model Service forwards the command to the appliance(s)
10. The appliance(s) carry out the action and log(s) it

Extensions:

6a. The event is invalid or the Controller encounters an error
 .1 The Controller throws a StoreControllerServiceException
. .2 The failure is logged

10a. The appliance encounters an issue preventing it from carrying out the action
 .1 The appliance throws an ApplianceException
 .2 The exception is encapsulated in a new event
 .3 The Store Controller Service is notified of the event



Use case diagram displaying Store Controller use cases.

Actors

The Store Controller Service is the parent of the other elements of the overall 24X7 System architecture. As such, the only external actor influencing the Controller is the administrator or executive function that instantiates the service.

Design

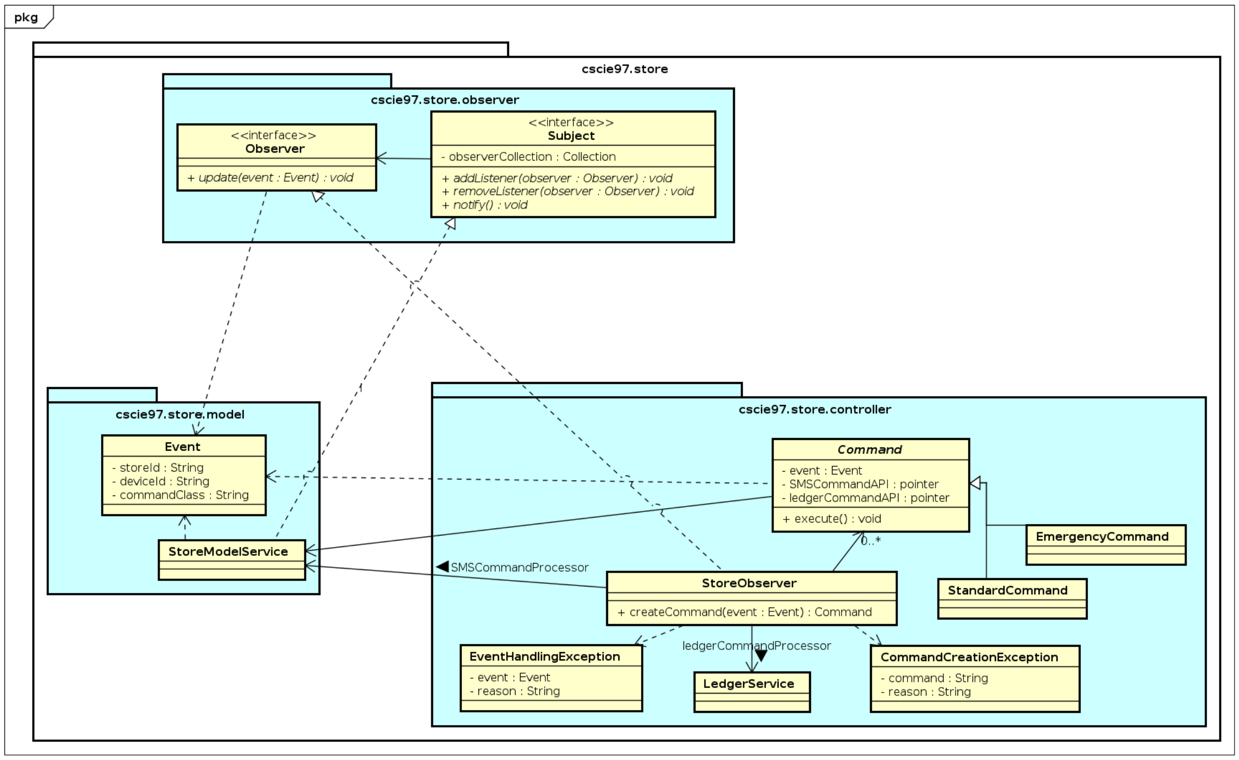
In accordance with the requirements, we utilize the Observer and Command patterns. We implement our own Observer and Subject interfaces rather than the ones provided by Java.util library.

We implement interfaces for the Subject, in this case utilized by the StoreModelService, and the Observer, implemented by the StoreObserver class. The Model Service detects an event within a store, represented by the Event class, and notifies the StoreObserver using the update(event) method.

The Store Controller Service uses the Command pattern by receiving the Event and creating a Command in response to the event. The abstract Command class is realized as two broad categories of Commands, EmergencyCommands and StandardCommands. EmergencyCommand is used to respond to emergencies such as a store fire. StandardCommands represent normal operational events within a store, such as customer queries, basket events, checkout events, and unintended but non-emergent events, such as a mess that requires cleaning. These command categories will be implemented as individual concrete classes that extend Command, but it is useful to view them at a higher level as two command types. Guidelines are provided under Design Details for implementing new Event and Command types.

Class Diagram

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



This diagram illustrates the interaction between the StoreModelService and Store Controller Service components. The StoreModelService updates the StoreObserver when an Event takes place, the StoreObserver creates a Command in response and sends it back to the StoreModelService. The StoreObserver holds a reference to the public APIs of the StoreModelService and the LedgerService in order. The LedgerService is required to process transactions and query customer account balances. This fulfills the requirement of utilizing the blockchain ledger system to perform transactions.

Class Dictionary
See Assignment 1 design doc for the LedgerService and Assignment 2 design doc for the StoreModelService classes.

StoreObserver

The StoreObserver is the class that receives Events and creates Commands in response. In the context of the Command Pattern, the StoreObserver is the client. Its associations are the public APIs of the StoreModelService and the LedgerService. The StoreObserver is a Singleton class.

Methods

Method Name	**Signature**	**Description**
createCommand	(Event event) : Command	Uses the Event signature to determine which concrete Command to create an instance of in response to events occurring within stores.
update	(Event event) : void	Calls createCommand and executes the command in response to an event update.

Associations

Association Name	**Type**	**Description**
SMSCommandProcessor	CommandProcessor	The command line processing API of the StoreModelService. Commands are directed to this API.
ledgerCommandProcessor	CommandProcessor	The command line processing API of the LedgerService. Transactions and balance inquiries are sent to this API.

Event

The Event class represents events that occur within the system stores, such as customer activities, spills, and emergencies.

Properties

Property Name	**Type**	**Description**
commandClass	String	The type of event that occurred (a customer adding a product to their basket, for example). This property is used to determine the correct response command.

storeId	String	The unique ID of the store where the event occurred.
deviceId	String	The ID of the device that detected the event.

Command

The Command class allows the Store Controller Service to direct appliances in response to events that occur within stores. It is an abstract class.

Methods

Method Name	**Signature**	**Description**
execute	() : void	Executes the stored behavior of the command

Associations

Association Name	**Type**	**Description**
event	Event	The event that caused the command to be created. Events contain state information such as event location.
commandProcessor	CommandProcessor	The API that the command is sent to which passes the command on to the target appliance.

For concrete Event and Command classes, see the Design Details portion of this document.
For exception classes, see Exception Handling.

Design Details

This design supports the requirements of monitoring and responding to changes within stores by having the StoreModelService create Event objects as events occur. Events contain information which the Store Controller uses to autonomously respond to occurrences within stores.

The StoreModelService passes events to the Controller by implementing the Subject interface and calling the notify() method, which updates the StoreObserver class. The StoreObserver implements the Obserfver interface, fulfilling the requirements of using that interface. The StoreObserver is passed the Event object using the update(event) method.This also fulfils the requirement of monitoring sensors and appliances for updates.

The StoreObserver then utilizes the Command pattern (meeting another design requirement) to create a concrete Command class. Concrete Commands are matched with concrete Events to implement a rule-based response logic system. Commands are passed stateful information from the Event object and are given pointers to the StoreModelService and LedgerService public APIs.

The execute() method of the Command object uses those APIs to implement the desired behavior in response to the event, for example querying a customer's account balance or ordering a robot to clean up a spill. This supports the requirement of generating and sending control messages to appliances.

Events and Response Commands

The following displays the Event commandClass properties and their corresponding concrete Command classes. These pairings provide the functionality to address all the events and responses layed out in the requirements document.

```
Emergency event: EmergencyCommand
Emergency event types {
        Fire:
        flood
        earthquake
        armed intruder
}
Standard events {
        enter-store: CheckInCommand,
        basket-event: UpdateBasketCommand,
        fetch-product: FetchCommand,
        customer-seen: CustomerSeenCommand,
        broken-glass: BrokenGlassCommand,
        product-spill: CleanAisleCommand,
        missing-person: MissingPersonCommand,
        check-acc-bal: CheckBalanceCommand,
        assist-customer: AssistCommand,
        checkout: CheckoutCommand
}
```

See the Store Controller Service Requirements document for details including desired response behavior for each concrete event and command pair.

It is important to consider that other events and commands may be added in the future. The following section provides a general overview of how to implement new concrete Event and Command classes.

Implementing Events

Events will be defined according to the following structure:

```
public class <event class> extends Event {
   <datatype> field1;
   <datatype> field2;
   . . .

   public  <event class> (String deviceId, String storeId, String... eventArgs) {
      super(deviceId, storeId, eventArgs);
      This.field1 = eventArgs[1]
      This.field2 = eventArgs[2]
      . . .
   }
}
```

eventArgs contains relevant information passed to the event factory from the sensors (or test API).

Implementing Commands

Commands shall be implemented according to the following structure:

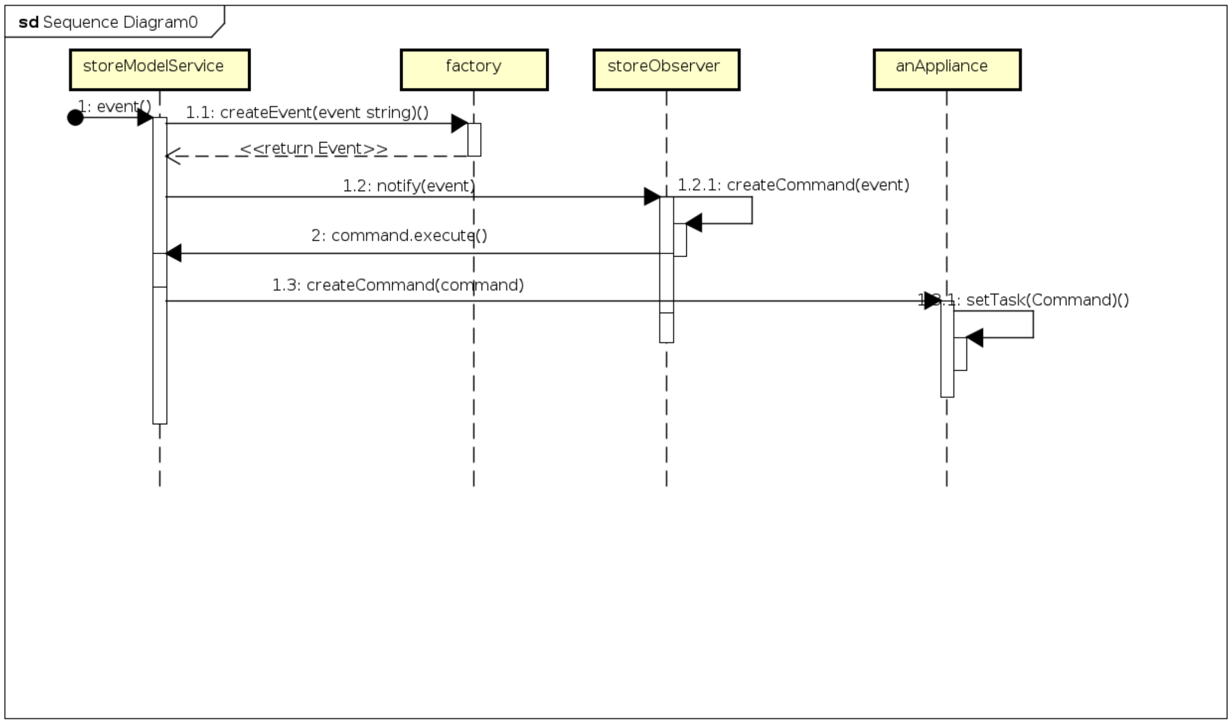
```
public class <command_name> extends Command {
  # relevant fields

   public <command_name> (Event event, CommandProcessor SMSCommandProcessor,
CommandProcessor ledgerCommandProcessor) {
      super(event, storeModel, ledger);
      this.relevantField = event.getRelevantField();
       . . .
   }

   @Override
   public void execute() {
      # implement desired behavior
   }
}
```

Event timeline:

1. A store device detects an event
2. A factory method creates a concrete Event class instance
3. The Event, location, and the deviceID are sent to the Controller
4. The Controller uses its internal rule-based logic to decide on a response
5. The Controller issues the command to the StoreModelService
6. The StoreModelService executes the command, delegating the action to an appliance
7. The appliance carries out the commanded action



Sequence diagram visualizing the event timeline described above.

Exception Handling

Errors are expected to occur during store operation due to the unpredictable nature of the store's customers and the physical environment each store is in. It is crucial that errors are handled gracefully to allow the system to continue to operate in an automated fashion.

EventHandlingException

A StoreControllerException is thrown when a component of the Store Controller Service fails to execute its intended function. The action that was being performed and the reason for the failure is captured.

Properties

Property Name	Type	Description
event	Event	The event that caused the exception.

reason	String	The reason for the exception.

CommandCreationException

This exception is thrown when the StoreObserver encounters an issue with the creation and sending of a Command.

Properties

Property Name	Type	Description
command	String	The command that was being instantiated when the error occurred.
reason	String	The reason that the action was invalid.

Testing

Testing will be conducted using test scripts and the event simulation functionality of the Store Model Service API. A script will be created that will model a new store instance and all the entities within it (customers, products, appliances, and so on). Then, events will be simulated using the following syntax:

```
create-event <device-id> location <location> event <event>
```

The Controller service will treat these scripted events as real events and issue commands to appliances accordingly. Print statements will be used to keep track of the commands that are issued for verification that the system is behaving as intended.

Risks

Special care should be taken to ensure that errors are handled gracefully so that the Controller Service continues running in the face of unexpected events because of its crucial role as the brain of the system.

It is inevitable that some event will occur which the Controller does not have a rule for. As such, care must be taken to ensure that the implementation follows the open-closed principle so that support for new Events and Commands is easily added without losing functionality.