

Advanced Object Oriented Programming
POST Assignment #2

Team Tabs



Eric Groom • Richard Robinson • Michael Winata
Michael Swanson • Peter Le

Table of Contents

1) Introduction	pg. 2
2) Overview	pg. 2
3) Use cases	pg. 20
4) Scope of work	pg. 28
5) Compile	pg. 30
6) Difficulties	pg. 30
7) Assumptions	pg. 32
8) Implementation	pg. 32
9) Results / Conclusions	pg. 35

1) Introduction

This application is a POST (point of sale terminal) with integrated GUI. It is possible to query for items by description or UPC, add a selected item with a given quantity, remove items from the cart, and to pay with cash, credit, or check. The list of items is retrieved from a remote server and when a transaction is completed it sends

the transaction to the server. Any number of transactions can occur until the store is closed by hitting the store close button (red on off icon), which on press prints the list of all transactions to terminal and closes the application.

2) Overview

Terms used:

UPC (universal product code):

Unique code to identify the items.

User:

A user is an entity which is making use of the POST. Subdivided into customer and Manager.

Customer:

Purchaser of goods.

Manager:

Seller/ Merchant, owner of the Store. Has permission to open or close stores. May add or remove inventory or items to the catalog.

Catalog:

Items available for sale

Inventory:

Items currently in stock (not necessarily available from sale)

POST (Point of sale terminal):

Point of access for purchasing items, maintenance, and instantiation of stores.

Item:

Entity representing an item for sale, includes an identification code, price, and description.

Payment:

Means of purchasing an item-- Cash/ Check/ Credit

Transaction:

Exchange of payment method

Includes date of purchase, items, customer information, total cost, payment method, change returned, and if payment was accepted or not.

UML Diagrams:

Packages:

- fileparser (diagram 1-0)
- item (diagram 2-0)
- post (diagram 3-0)
- store (diagram 4-0)
- transaction (diagram 5-0)
- user (diagram 6-0)
- Assignment-2 UML
- gui (diagram 7-0)
- network (diagram 8-0)

Diagram 1-0: fileparser package:

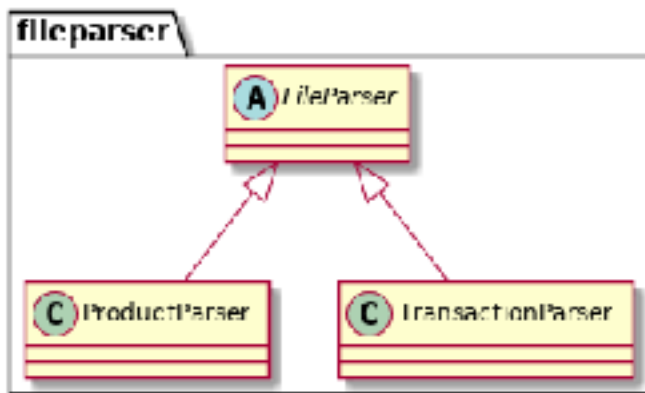


Diagram 1-1: FileParser class:

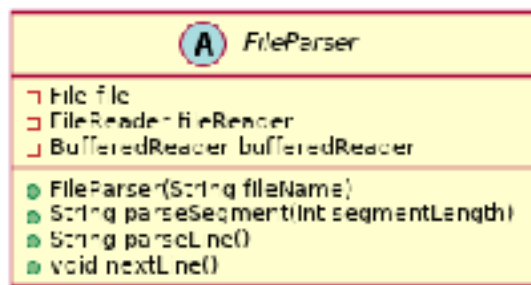


Diagram 1-2: ProductParser class:

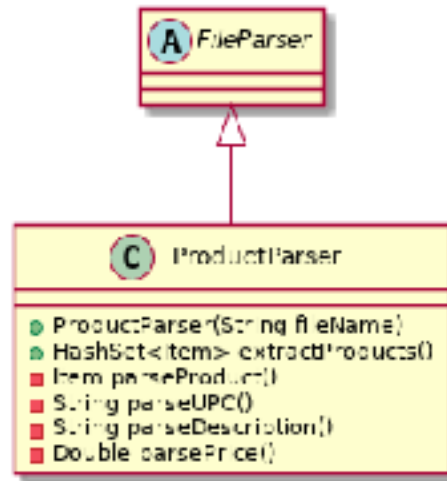


Diagram 1-3: TransactionParser class:

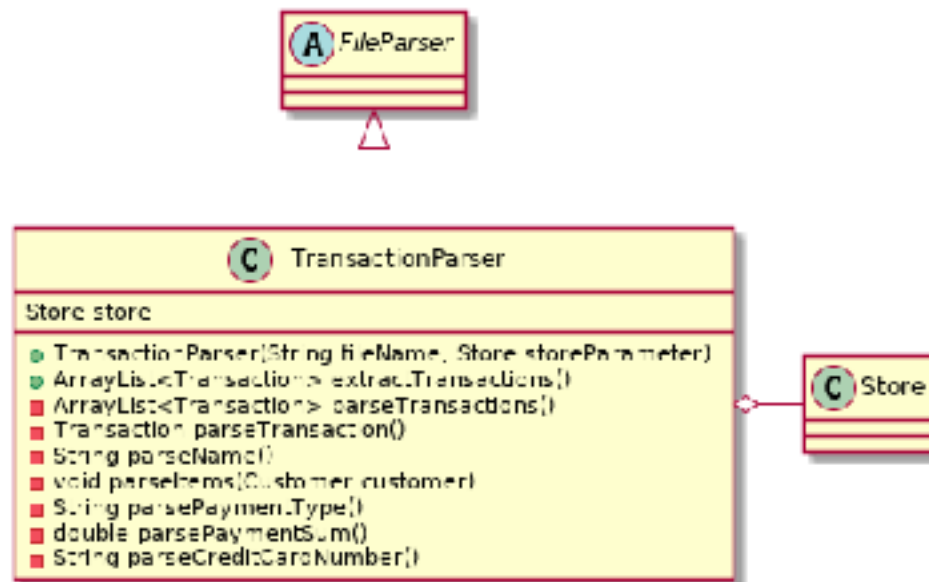


Diagram 2-0: item package:

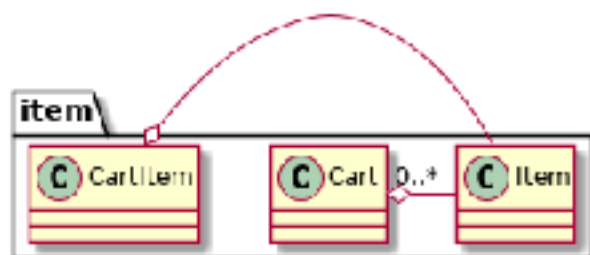


Diagram 2-1: Item class:

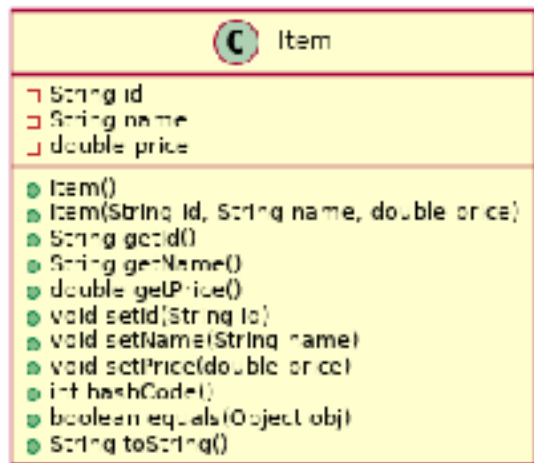


Diagram 2-2: CartItem class:

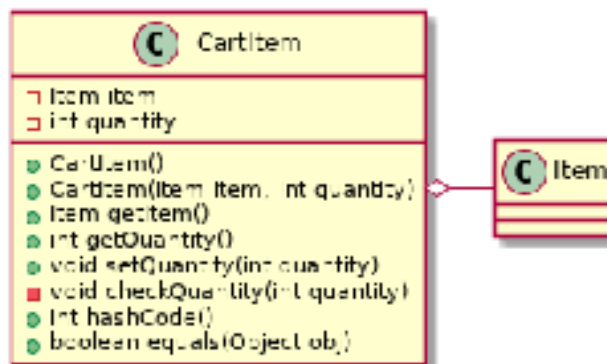


Diagram 2-3: Cart class:

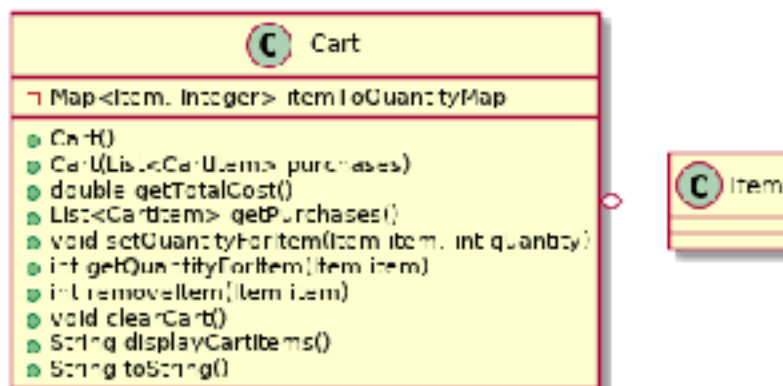


Diagram 3-0: POST package:



Diagram 3-1: POST class:

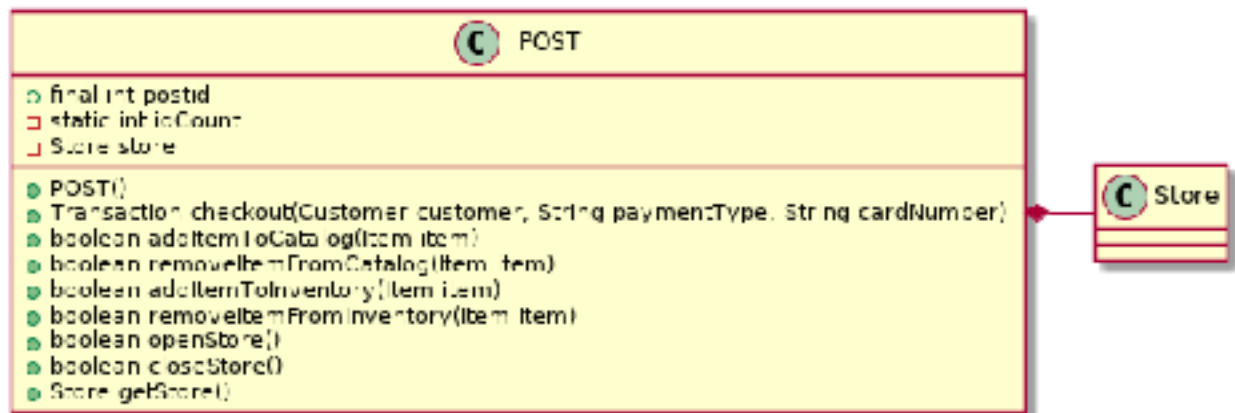


Diagram 4-0: store package:

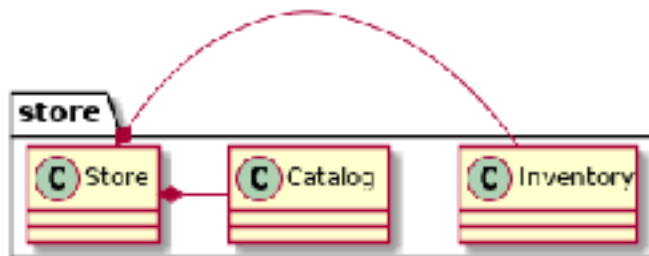


Diagram 4-1: Catalog class:

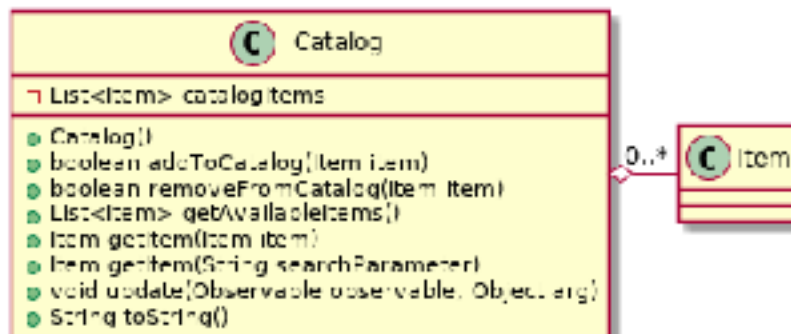


Diagram 4-2: Inventory class:

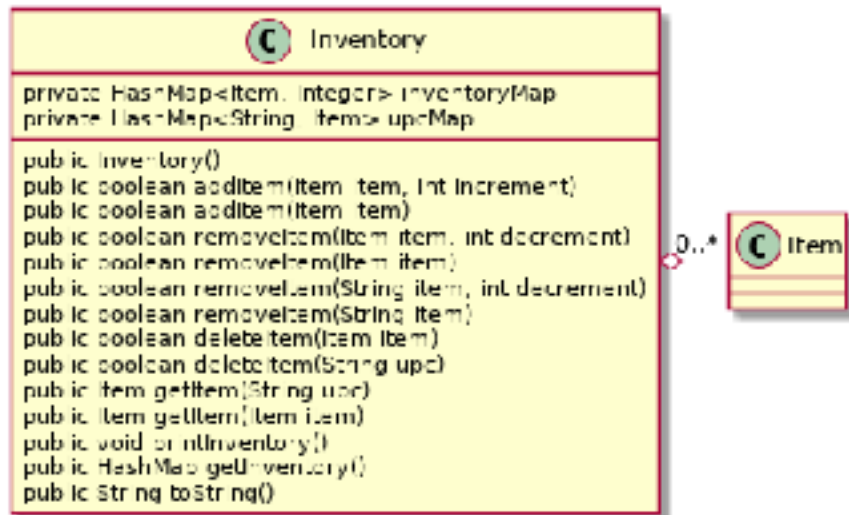


Diagram 4-3: Store class:

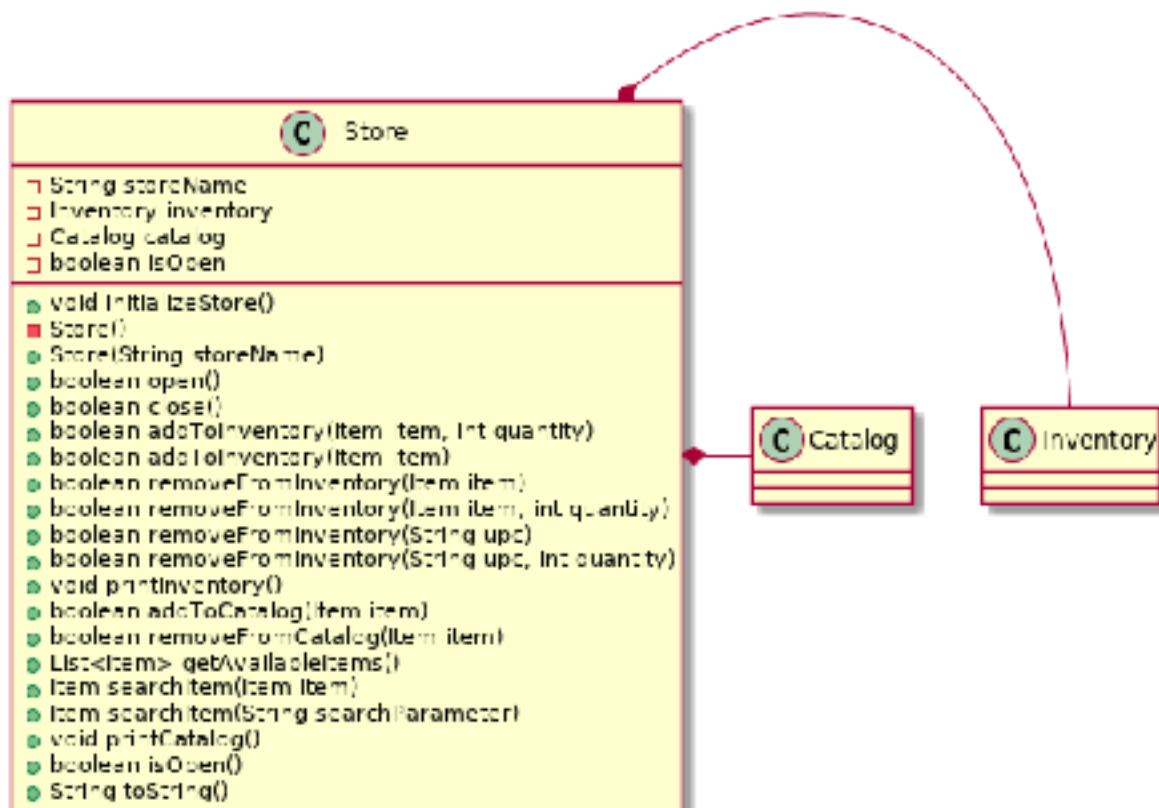


Diagram 5-0: Transaction package:



Diagram 5-1: Invoice class:

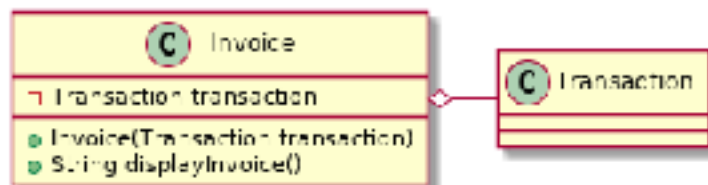


Diagram 5-2: Payment class:

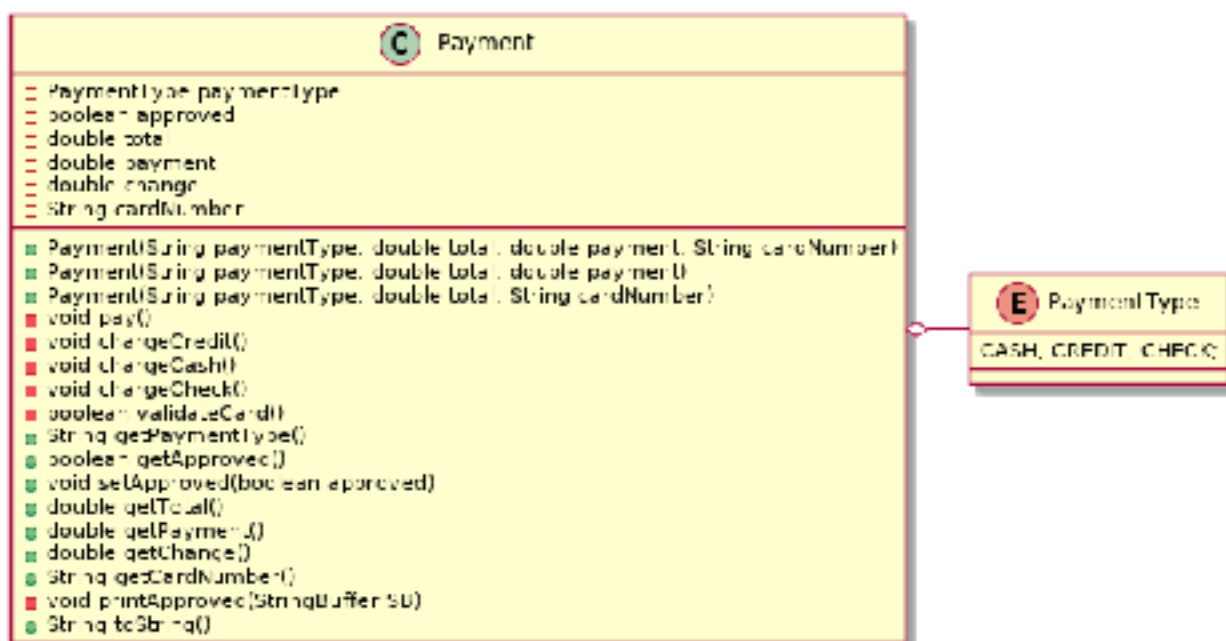


Diagram 5-3: Transaction class:

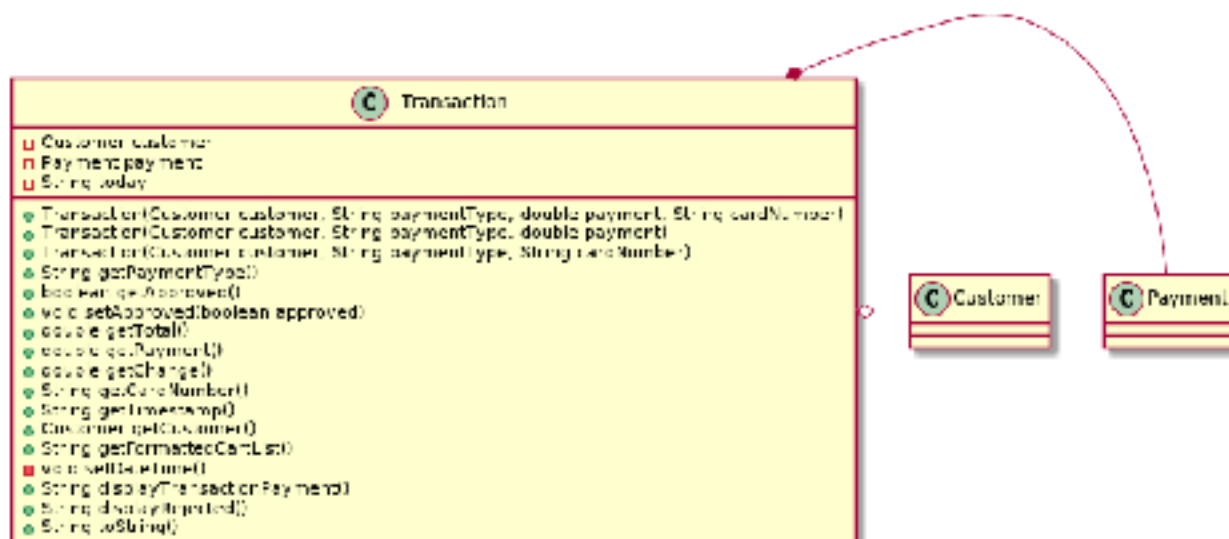


Diagram 6-0: user package:

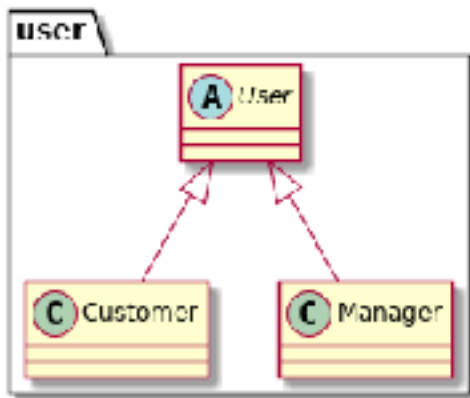


Diagram 6-1: User class:

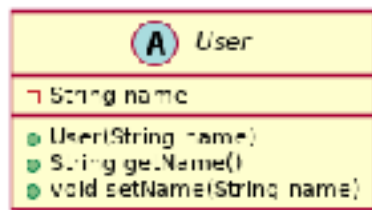


Diagram 6-2: Customer class:

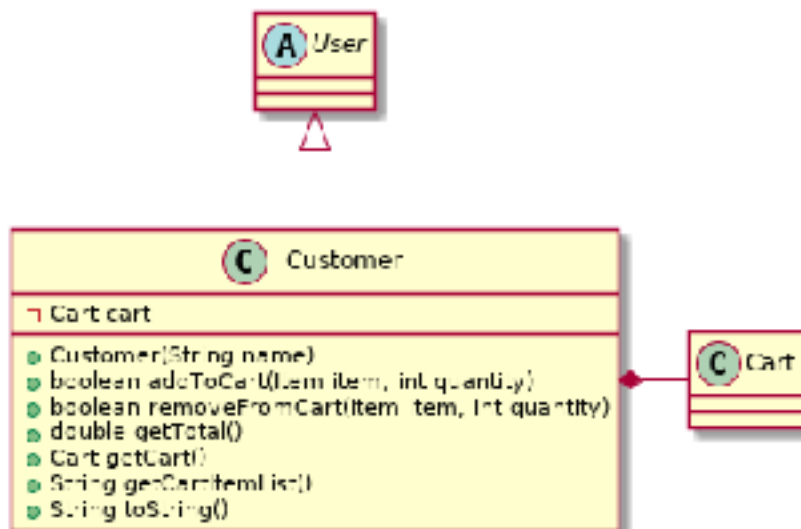


Diagram 6-3: Manager class:

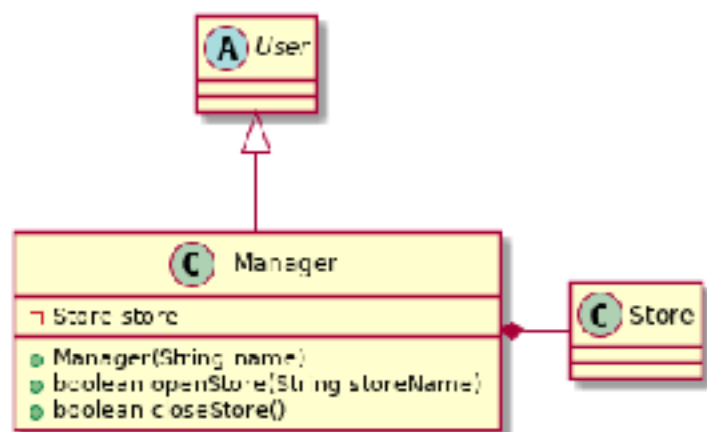


Diagram 7-0: gui package:



Diagram 7-1: gui.panel.bottompanel package:

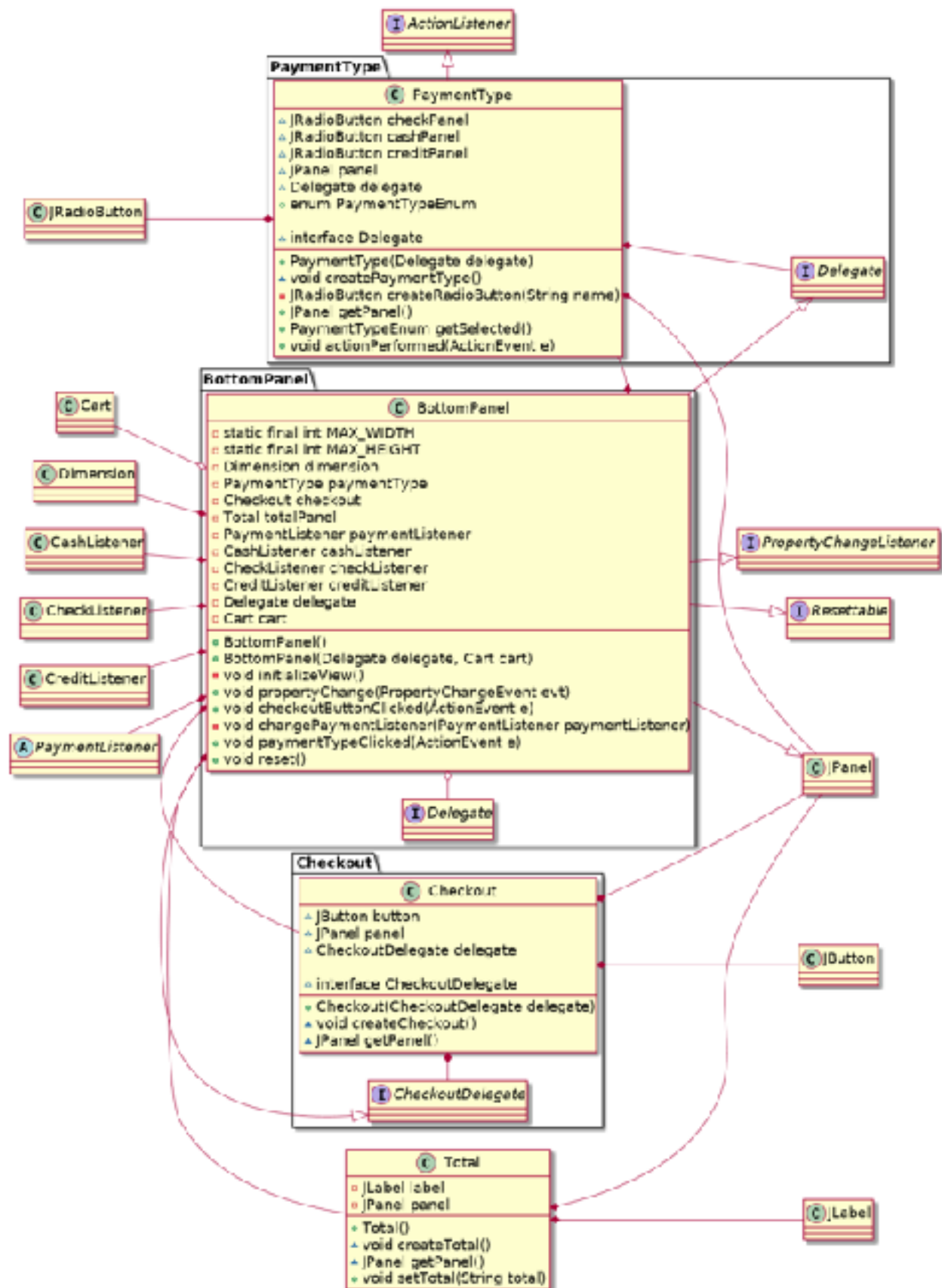


Diagram 7-2: gui.panel.bottompanel.paymentlistener package:

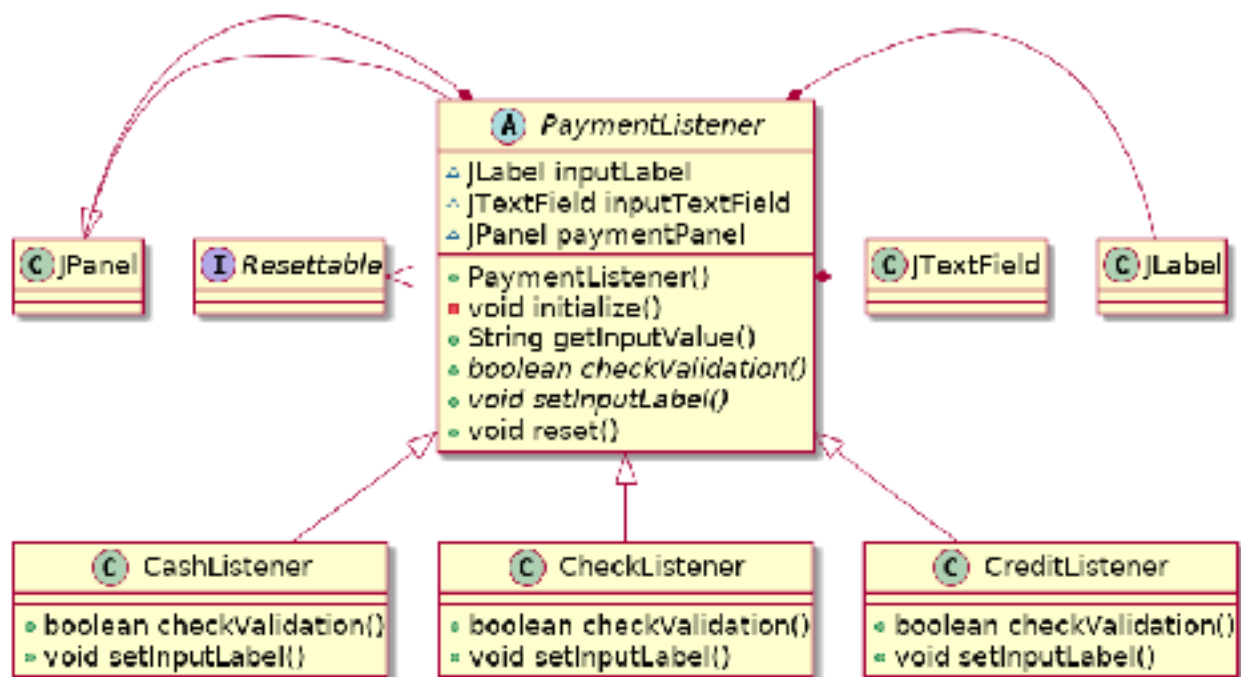


Diagram 7-3: gui.panel.middlepanel package:

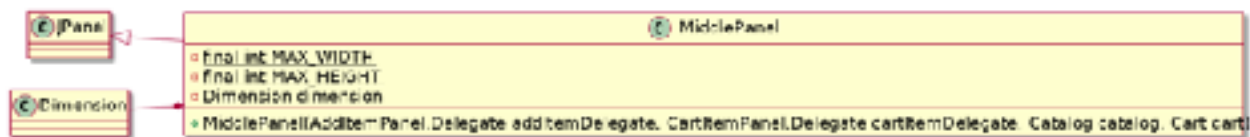


Diagram 7-4: gui.panel.middlepanel.cartitemgui package:

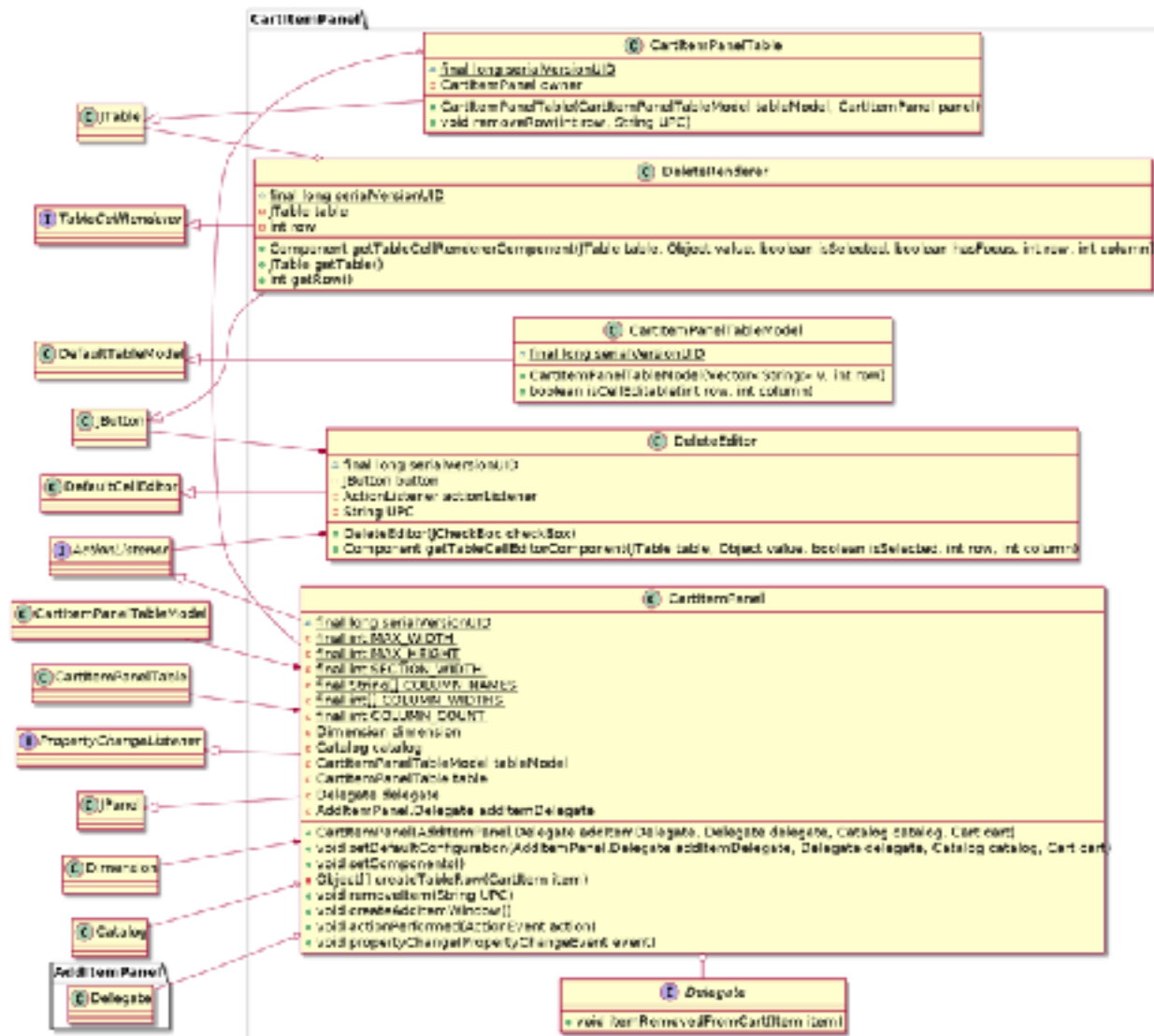


Diagram 7-5: gui.panel.optionspanel package:

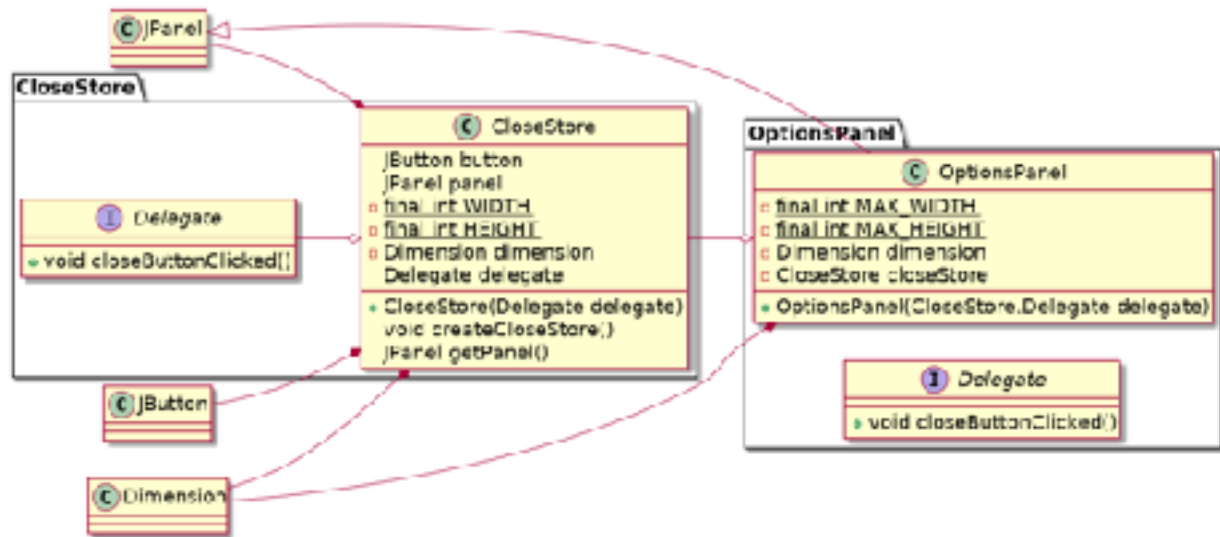


Diagram 7-6: `gui.panel.toppanel` package:

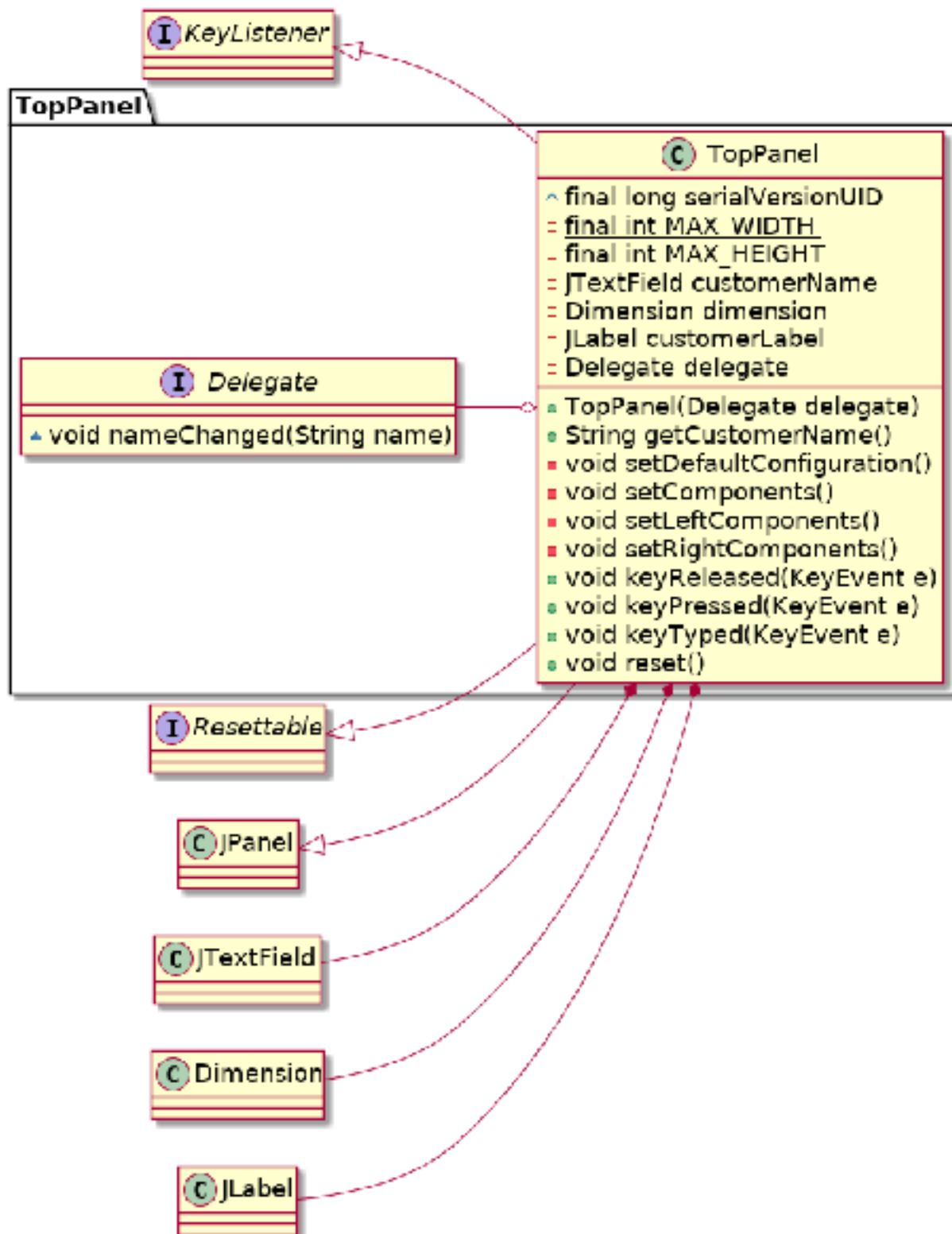


Diagram 7-7: gui.productsearch package:

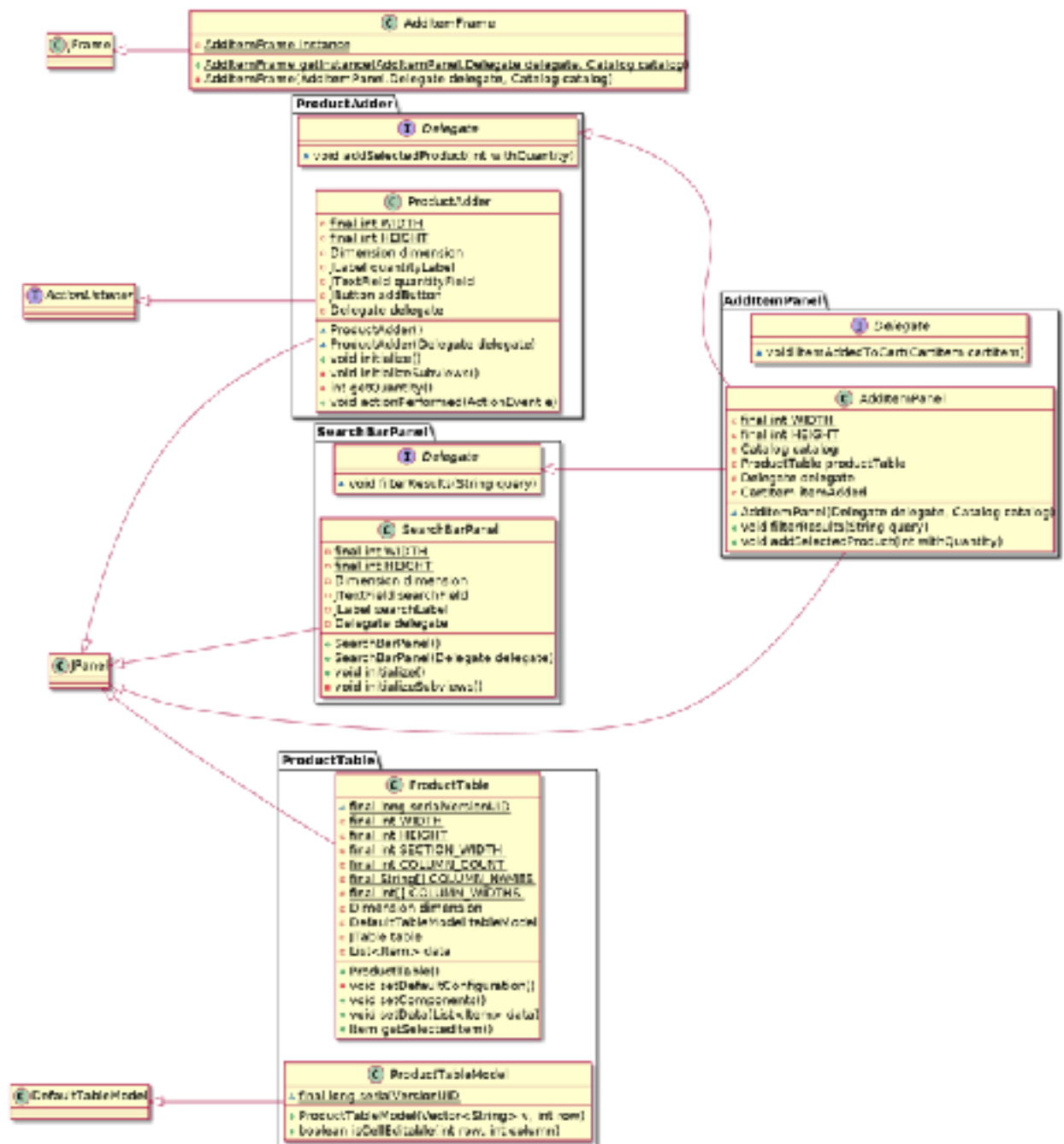


Diagram 8-0: network package:

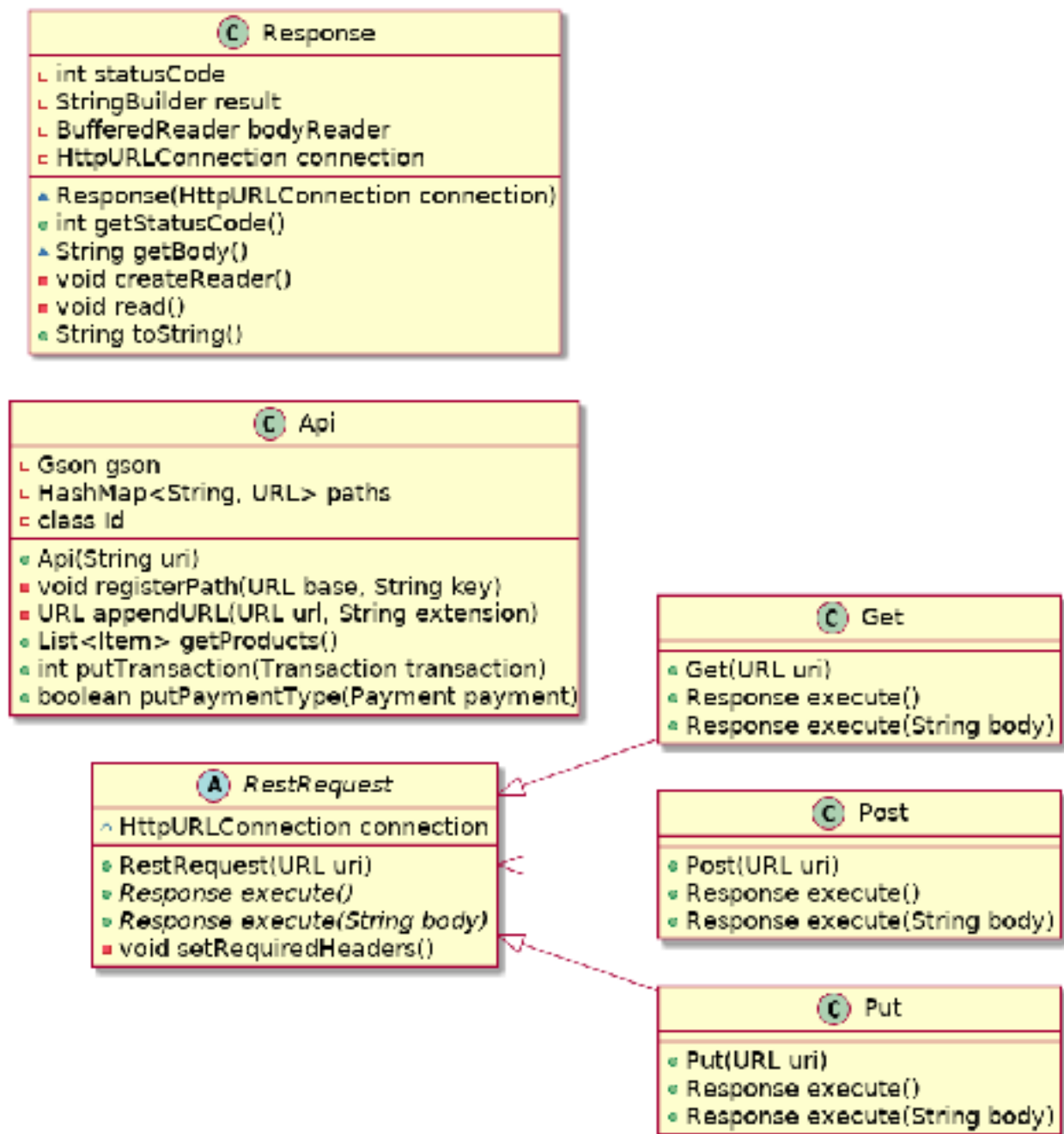


Diagram 8-1: network.adapters package:



3) Use cases

Use Case #	1
Use Case Name	Change Customer Name
Summary	User can change the customer's name
Dependency	
Actor	User of the POST
Precondition	
Description	The user of the POST system can change the current customer's name for logging and invoice reasons. The user of the POST system is expected to be an employee or manager of the store.
Alternative	
Postcondition	Customer's name changes in system

Use Case #	2
Use Case Name	Browse Catalog
Summary	User can browse the catalog for items for sale
Dependency	
Actor	User of the POST
Precondition	
Description	The user of the POST can look through the list of items for sale and see their UPC, description, and price. The user of the POST system is expected to be an employee or manager of the store.
Alternative	

Postcondition	
----------------------	--

Use Case #	3
Use Case Name	Catalog Search
Summary	User can search the catalog for items for sale
Dependency	
Actor	User of the POST
Precondition	
Description	The user of the POST can search for items in the catalog by name or UPC. The user of the POST system is expected to be an employee or manager of the store.
Alternative	
Postcondition	

Use Case #	4
Use Case Name	Add to Cart
Summary	User can add items from the catalog to the cart
Dependency	
Actor	User of the POST
Precondition	User can somehow select an item i.e. by UPC
Description	The user of the POST can add items from the catalog into the current customer's cart. The user of the POST system is expected to be an employee or manager of the store.
Alternative	
Postcondition	The item is added to the customer's cart.

Use Case #	5
Use Case Name	Purchase with Cash
Summary	Customer can purchase item(s) with cash
Dependency	
Actor	User of the POST, Customer
Precondition	The customer has item(s) in their cart.

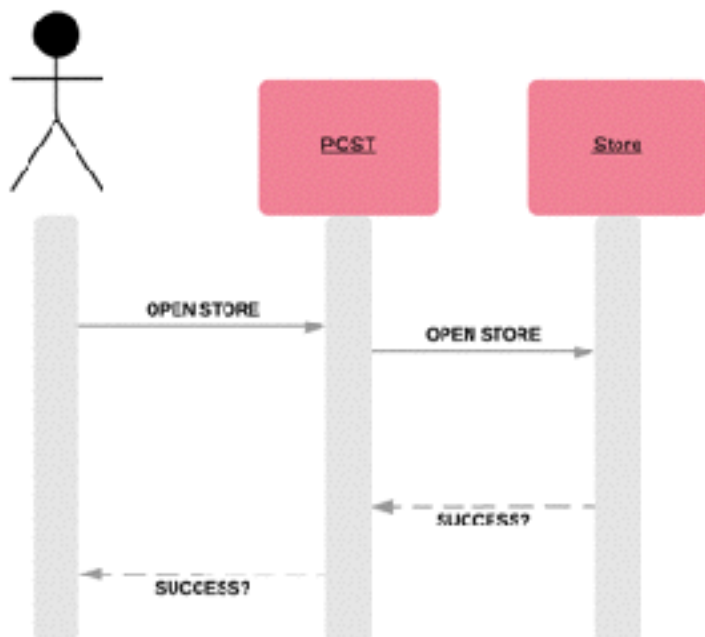
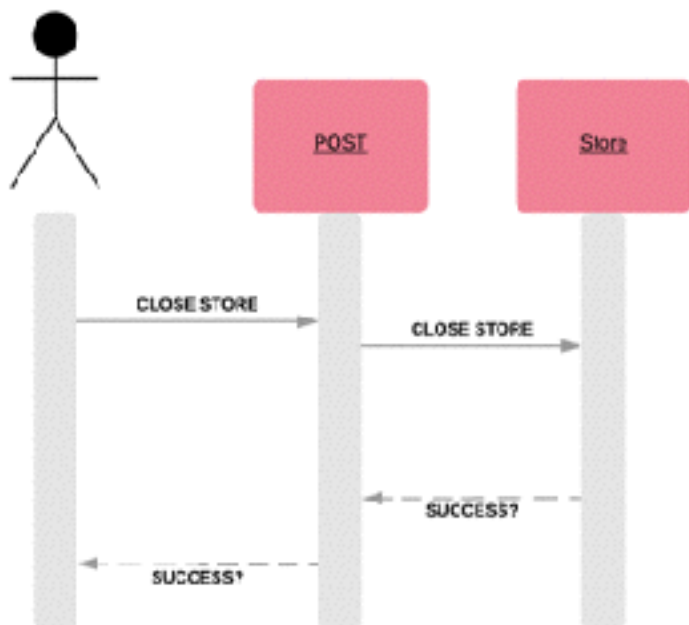
Description	<ul style="list-style-type: none"> • The customer can purchase the items in their cart with cash provided they have enough to cover the cost of the transaction. • The user of the POST is responsible for entering the customer's payment information into the system. • The customer's payment is validated. • If validated, the transaction is logged and the items are considered purchased.
Alternative	
Postcondition	

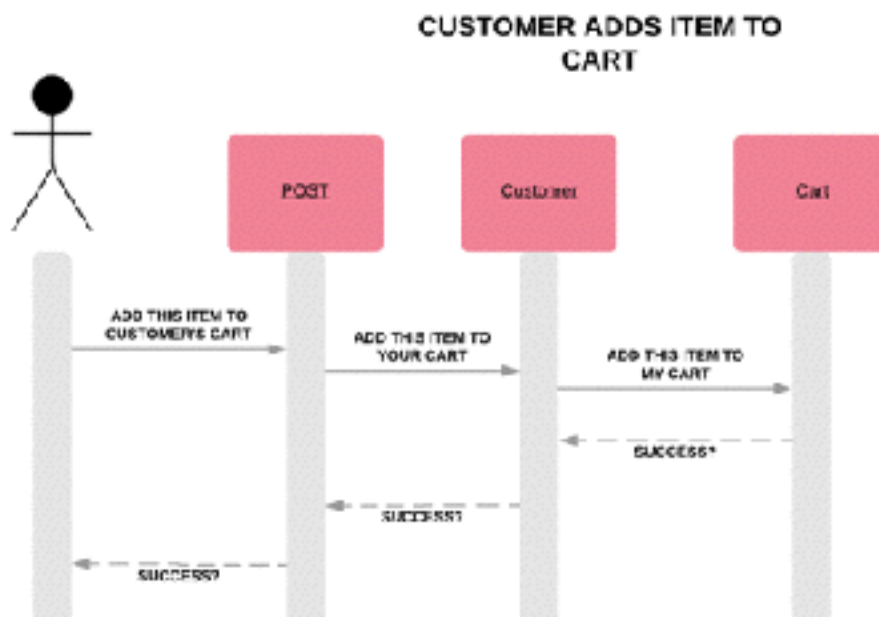
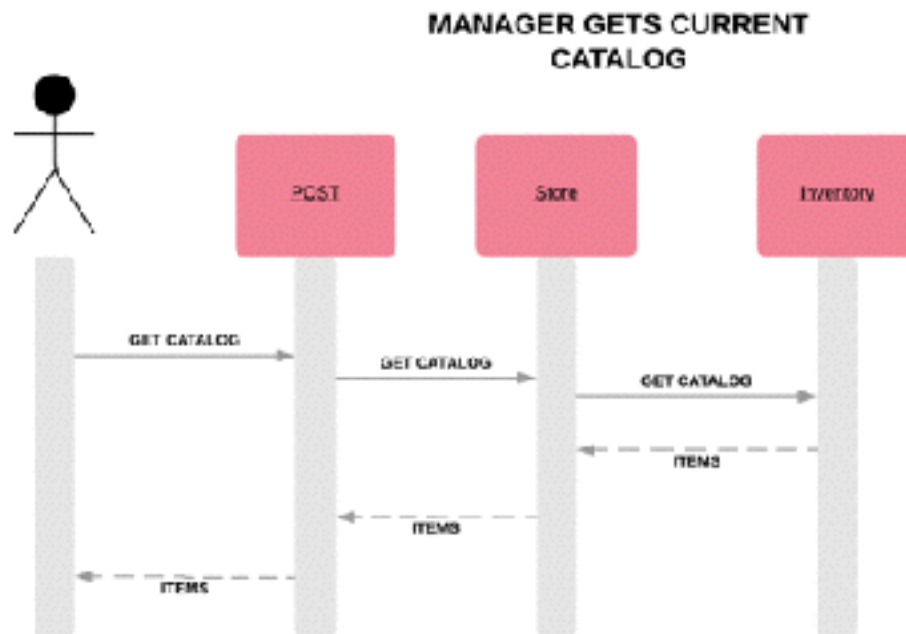
Use Case #	6
Use Case Name	Purchase with Check
Summary	Customer can purchase item(s) with check
Dependency	
Actor	User of the POST, Customer
Precondition	The customer has item(s) in their cart.
Description	<ul style="list-style-type: none"> • The customer can purchase the items in their cart with check. • It is assumed that the customer will write a check for the exact amount of the transaction. • The customer's payment is validated. • If valid, the transaction is logged and the items are considered purchased.
Alternative	
Postcondition	

Use Case #	7
Use Case Name	Purchase with Credit Card
Summary	Customer can purchase item(s) with credit card
Dependency	
Actor	User of the POST, Customer
Precondition	The customer has item(s) in their cart.

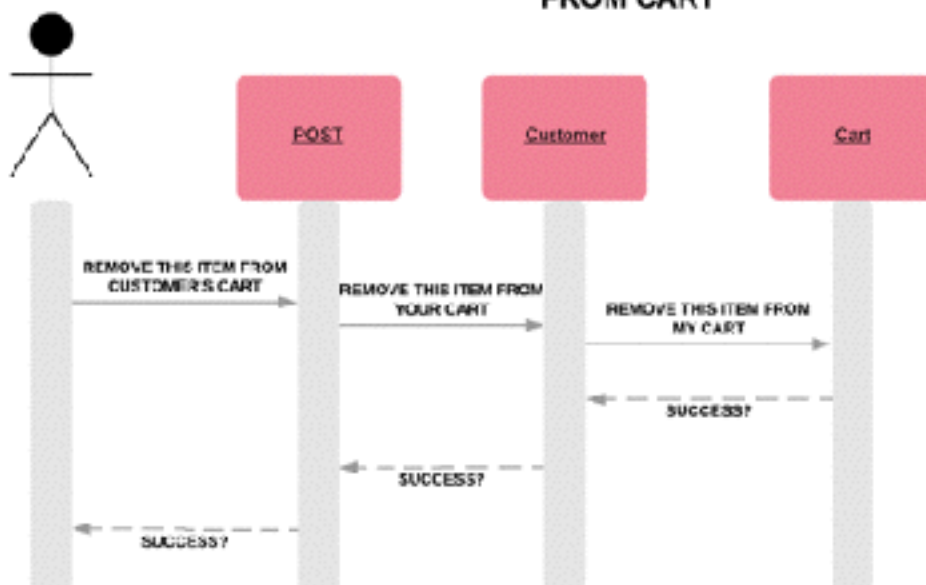
Description	<ul style="list-style-type: none"> • The customer can purchase the items in their cart with their credit card. • The user of the POST will enter the customer's credit card number into the system. • The customer's payment is validated. • If valid, the transaction is logged and the items are considered purchased.
Alternative	
Postcondition	

Use Case #	8
Use Case Name	Close Store
Summary	User can close the store
Dependency	
Actor	User of the POST
Precondition	
Description	The user of the POST can close the store which will close the GUI and print out invoices for the day to the console.
Alternative	
Postcondition	

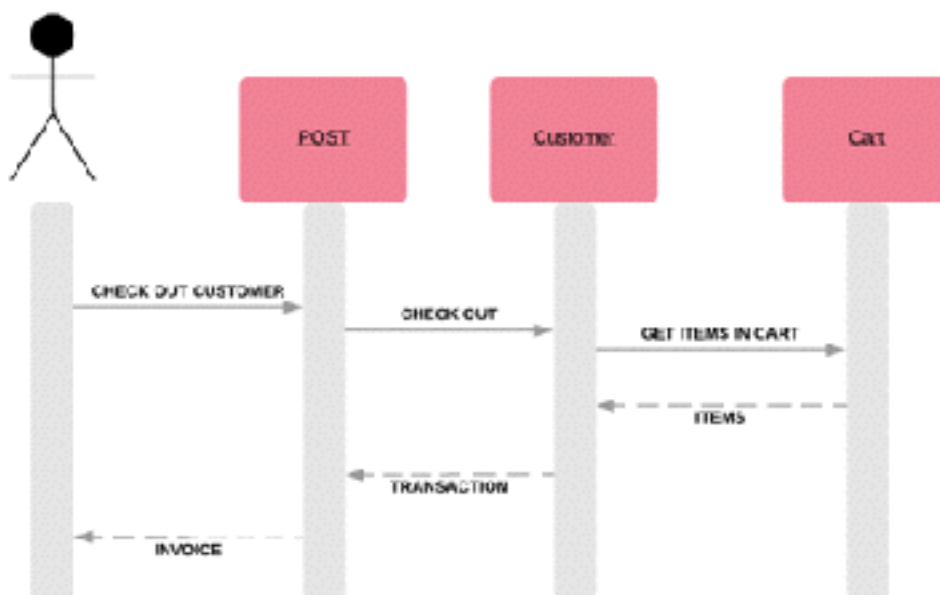
MANAGER OPENS STORE**MANAGER CLOSES STORE**

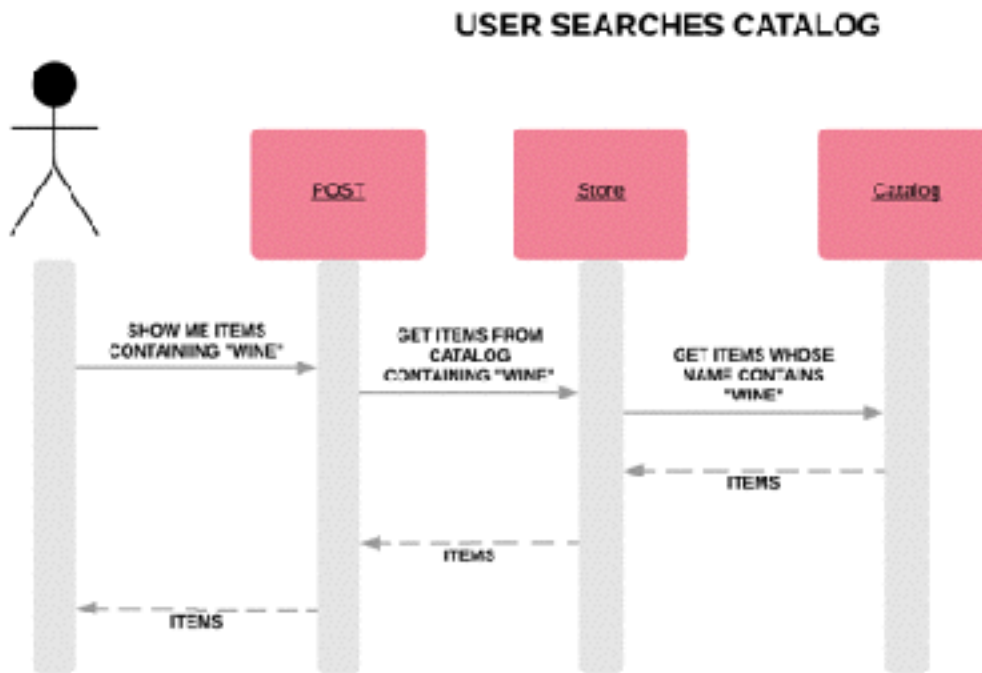


CUSTOMER REMOVES ITEM FROM CART



CUSTOMER CHECKS OUT





4) Scope of work

Tasks

Event	Expectations	Output
Initialize Store	Manager initializes store with a name	Works as intended
Initialize Stock	Manager adds item to stock	Works as intended
Initialize Catalog	Manager adds items to catalog	Works as intended
Initialize POST	Creates store instance	Works as intended
Customer identifies self	Customer name can be inputted and stored	Works as intended

Customer time arrived	Customer arrival time documented	Works as intended
Customer adds items to cart	Add item to customer cart	Works, does not currently check catalog / inventory, but methods are there.
Customer purchases items	Customer chooses to purchase items	Works as intended
Payment method confirmed	Customer enters payment information and amount.	Works as intended, in extensions test edge cases.
Receipt Shown	Receipt is given	Receipt can be shown
Store Closed	Store is closed, cannot purchase until re-opened	Store closes
Invoices Printed	Invoices are printed	Invoices can be read and printed from text file.

5) Compile

Command line:

make install (or place GSON jar in jars/gson-2.8.5.jar)

make build

6) Difficulties

a. UML Design:

- i. Problem: We had issues designing our UML diagrams since we each had our own idea of how the whole program should run and interact with each part. We also thought some classes were doing different tasks from one another which made it confusing when we were coming up with it.
- ii. Solution: Drawing out the diagram and explaining what each class's task was helped us to resolve the confusion that we had about not knowing how they interacted with each other. Once we got the hang of it, everything flowed smoothly for the design aspect of the UML.

b. Scheduling:

- i. Problem: Being that it was most of our last semester at this school, we each had classes and work schedule that we needed to work around. Normally this wouldn't be an issue, but the assignment was due in a week, therefore there wasn't much room to play around and fit everybody's timing that they would like.
- ii. Solution: The way that we worked around this was that we met up earlier and set up a good flow of work that didn't conflict with one another. This means that we each could work on our own once a good base was set up.

c. UML Implementation:

- i. Problem: When we finally started the coding aspect of the assignment, we found out that there were many flaws or gaps in logic that we had in our UML diagrams. We tried to follow it as much as we could since it gave us a good base to work from, but it was missing many parts and that it didn't flow as well as we had anticipated. Much of the design we had needed to be changed to fit with one another as more of the assignment was being built.
- ii. Solution: We used the UML diagram as a basic idea of how our program should work, and not as a solid must follow plan. The reason for this is because as we built the code, we figured out that some of our logic was wrong, or that there were better ways that they could be implemented. One thing that had to be considered with this approach is that we needed to inform the rest of our team on these updates or it will cause confusion.

d. Specification:

- i. Problem: We followed to the best of our abilities to follow the specifications that were given to us in the assignment 1 pdf and information that were mentioned in class, however there were some issues that we did not understand. Some of the specifications were extremely broad and could be taken with many different interpretations.

- ii. Solution: We decided to follow the specification and make assumptions that made sense in the context of our program, in this case a “post”. Another thing that we did when we figure that it was crucial is that we would ask the professor on slack, and or other students who might have had the same issues.

e. Code quality:

- i. Problem: There weren't that much time given for this assignment given that the specification was a bit broad. This means that our code quality is a bit behind because we weren't able to test some of the edge cases in the sense that much of the code kept changing as we were figuring out what we were supposed to do.
- ii. Solution: To deal with this, we made the code as modular as possible allowing us to change portions of the code without it affecting the rest of the program. This allowed us to keep our code quality up while working with code that was always changing.

f. GUI:

- i. Problem: Graphic user interface was difficult to make because Java's GUI api is tedious to build in. It forces us to use panels inside of other panels which made it hard to access other attributes. We also had a problem with connecting our GUI into existing code as well.
- ii. Solution: To fix our problem with not being able to have buttons work across panels, we implemented a delegate pattern which fixed much of the issues that we were having. In order to make the GUI work with our current program, we had to make some minor changes in the implementation of some of our files.

g. Network:

- i. Problem: Integrating GSON into our project was fairly difficult, especially to get working across Operating Systems. Additionally, the library was difficult to work with compared to other languages: the library seems to focus too much on the “happy path” where your classes match the structure of the JSON exactly; we ended up having to write custom serialization/deserialization helpers. Additionally, the library looks at private class properties and attempts to automatically parse them; this was a problem as some of the prices in the API were strings with a dollar sign, and it took a long time to figure out why we were getting parsing errors, as we thought it was calling our classes constructor instead of analyzing the instance properties of the class.
- ii. Solution: We weren't able to create a makefile that worked on both Windows and Unix systems, we instead used the makefile on Unix systems and manually linked the GSON library in IDE's on Windows. As for solving our problems with the library itself, it required looking through the documentation.

7) Assumptions

User Inputs: We assume that the user and only enter in valid inputs with the correct formatted data.

Store / manager interaction: We assume that the store is set up and controlled by the manager.

products.txt / transactions.txt: We assume that these information is in correct format and located in our /db directory.

Credit / check: We assume that the card / credit will fail 10% of the time at random on our machines and that they have an unlimited amount in their accounts. We also assume that the cashier will not take check if it isn't the correct charge.

Inventory: We assume that the store will have a bottomless inventory / stockroom that defies the law of physics.

Item: We assume that items will not have conflicting id number given from the product list.

8) Implementation

This project completed the expected tasks from what we were aware of. Progress moved forward at a steady rate and each meeting more components were completed.

All team members did different tasks and completed them individually according to the design. After this integration went fairly smoothly. As long as the methods and noted components were accounted for, the underlying logic was up to the team member's discretion.

Store

- Store
- Inventory
- Catalog

Store: Holds inventory and catalog, store name, and if it is open or not

Inventory: Keeps track of items and quantities within a store

Catalog: Keeps track of items for sale

Inventory is observable and catalog is an observer so that when an item is deleted from inventory it is automatically removed from catalog, one should not be able to sell items that are not existing in inventory.

Items may be retrieved by passing an item or a upc or description.

Item

Item: An entity representing something for sale.

CartItem: Helper class to represent an Item in a cart, i.e. an Item with a quantity.

Cart: A Customer's list of items they wish to purchase and how much of each item they wish to purchase.

Fileparser

FileParser: an abstract base class for parser classes. It requires a file name with its file path on construct. The FileParser class provides three public methods:

- parseSegment: takes an integer and returns a segment with length based on the input integer.
- parseLine: returns a line.
- nextLine: skip the current line.

ProductParser: a public class extended from FileParser to parse a product text file. It requires a file name with its file path on construct. The ProductParser class provides one public method:

- extractProducts: returns a HashSet of Item objects. The extract method stops extracting item data once it hits end-of-file or a line with improper column setup.

TransactionParser: a public class extended from FileParser to parse a transaction text file. It requires a file name with its file path on construct. The TransactionParser class provides one public method:

- extractTransactions: returns an ArrayList of Transaction objects. The extract method stops extracting item data once it hits end-of-file or a line with improper column setup.

User

User: Abstract class representing a system user with a name.

Customer: User class representing a customer within the system. Extends User. It holds the Cart object that has a map of the items that the user had added to the cart including its quantity. The user is allowed to add and remove from the cart by giving an item and quantity for both actions.

Manager: Manager class representing a manager within the system. Extends User. The manager has the option to close and open the store by storing the store.

Transaction

Transaction: Transaction entity containing items sold, the customer who purchased them, and their payment method.

Payment: An entity representing a payment made with either credit card, check, or cash. On creation, it check whether the given transaction is approved or not.

Invoice: A receipt generated given the transaction.

POST

POST: POST is responsible with sending changes of state to the Store. POST is directly accessible from the Driver, contains methods such as openStore() and closeStore(), and methods for manipulating the store inventory.

Driver

Driver: Driver is responsible with initializing the file parsers and redirecting user input. Driver contains an instance of POST to communicate with the Store to make calls to process transactions and change the state of the Store.

Input: The abstract class Input represents the operations available on a given menu/screen. The classes that extend Input are responsible for taking an input signal and translating that into a destination or action for the Driver.

GUI

The gui stemmed from a main frame into subdivisions for top, bottom, and middle and an additional pop up frame for queries and adding items.

The gui is responsible for taking in user events, displaying relevant information, and the main panel is the controller for actions.

Network

Api: encapsulates all interaction with the REST API to the rest of the system. It's role is to coordinate REST handlers, JSON parsing, and serialization.

RestRequest: a base class containing the logic need to make HTTP requests. It is subclassed by PUT, GET, and POST which set the HTTP verb and minimal logic changes.

Response: Represents an HTTP response from the API which has a status code and body. It is also responsible for reading from the HttpURLConnection's stream.

adapters package: GSON's default serialization and deserialization were insufficient for our use case since our class structures didn't exactly match the API's JSON response format in both data types in naming convention. Each adapter implements a customer serializer and deserializer for various classes in our system. It serves as the translation layer between our system's and the server's data types.

9) Results / Conclusions

Entry Point

Customer:

Feb 20, 2012

UPC	Item	QTY	Unit Price	Total Price	Balance

* Add Item *

Total: \$0.00

☐ Check
☐ Cash
☐ Credit

Checkout

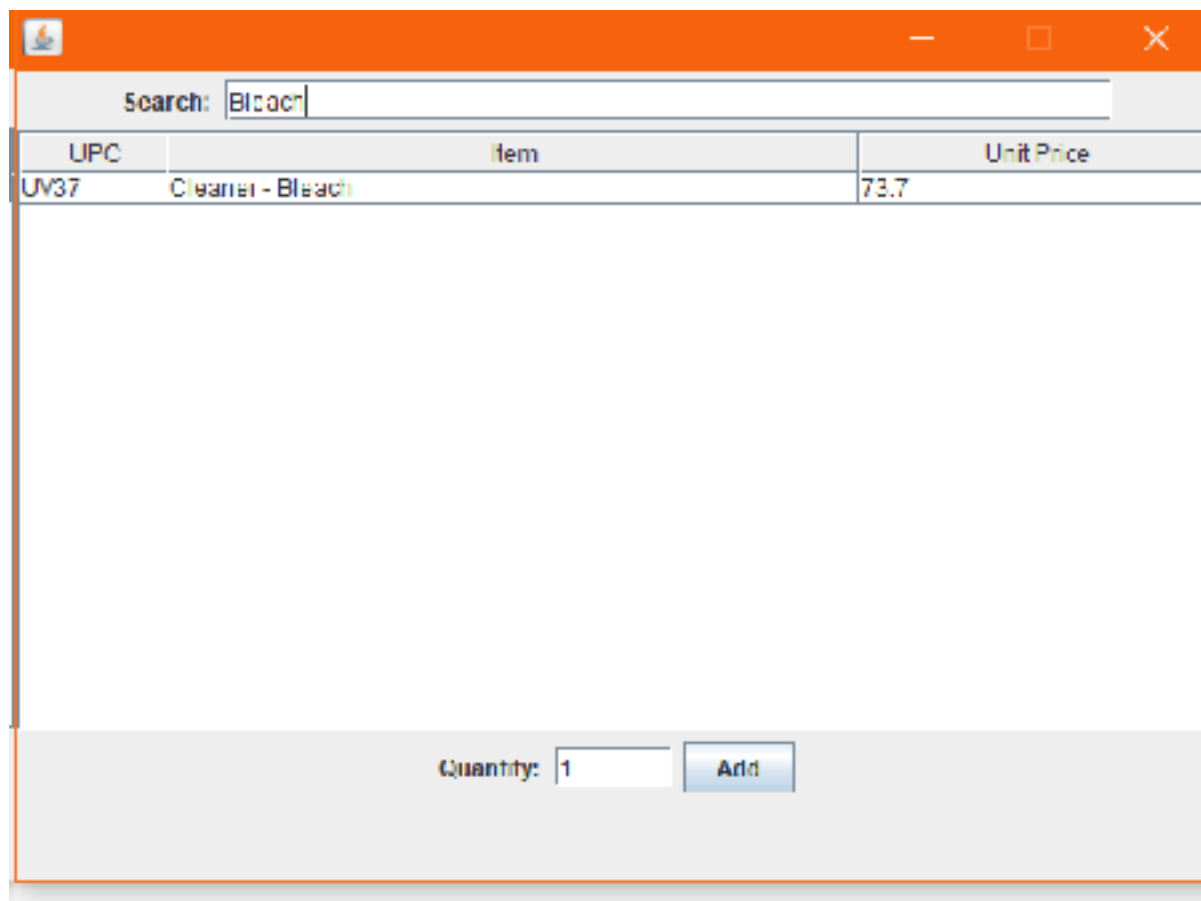
Click <Add Item>

Search:

UPC	Item	Unit Price
CX14	Wine - Niagara vga Redling	79.75
IOY3	Turkey - Oven Roast Breast	79.25
IWU1	Soup - Campbell's, Chix Cumbo	24.83
12P4	Wine - Carmenere Castilero Dcl	96.55
HIB2	Artichoke - Fresh	10.79
DVY3	Soba - Tropical Energy	73.31
7MM1	Cheese - Le Cheve Noir	92.81
4MZ2	Spice - Chili Powder Mexican	6.20
8FQ1	Cheese - Gouda	37.11
NC25	Lid Tray - 12In Dome	47.09
UV37	Cleaner - Bleach	73.7
UCD4	Vermacill - Sprinkles, Assorted	85.06
7V97	Beer - Sleemans Cream Ale	3.34
SCL4	Wine - Magnolia, Merlot Sr Vga	21.75
WOK7	Onions Granulated	39.33
OU03	Lettuce - Iceberg	37.73
42U4	Olives - Black Pitted	65.79

Quantity:

Search item:



Search:

UPC	Item	Unit Price
UV37	Cleaner - Bleach	73.7

Quantity:

Click add, close window

Customer: Janna Feb 20, 2019

QTY	Item	QTY	Unit Price	Total Price	Delete
1/1/37	Cleaner - Bleach	1	73.70	73.70	X

< Add Item >

Total: \$73.70

☒ Check
☐ Cash
☐ Credit

Checkout

Checkout -- insufficient payment

Message

Not enough money

OK

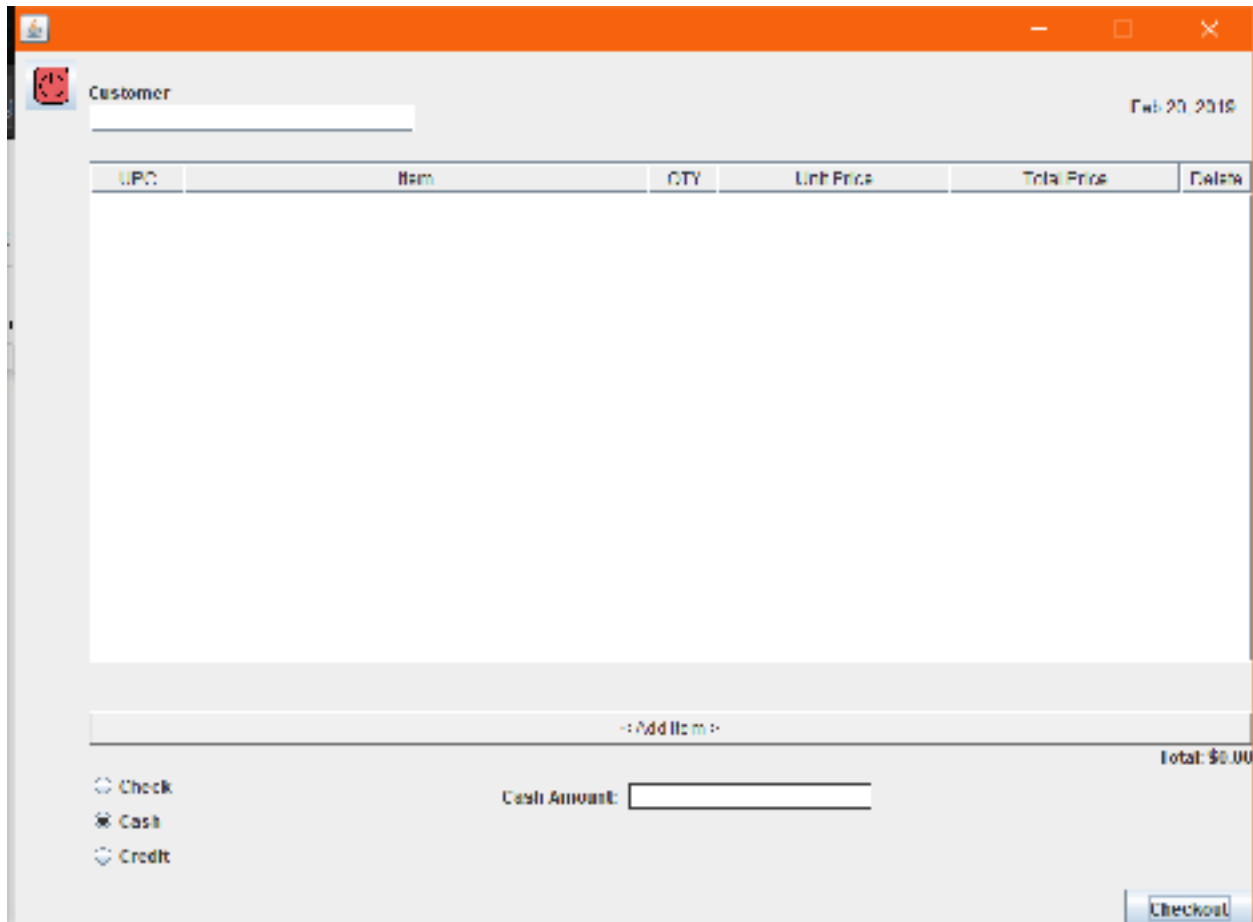
< Add Item >

Total: \$73.70

Cash Amount: 70

Checkout

With correct payment -- clears window saves results, pings server



Conclusion:

This assignment gave practice with designing classes and their relationships before coding began. Agreement was met with the basic needs and structure of the program. A lot of re-use was available from the first part of the project, but some components had to be extended and adapted for more functionality. Using UML diagrams helped us lay out the foundation for us to start building the objects needed. Some of the code could be much cleaner for this assignment, but all functionality asked for was provided within the short development time we had.

Writing clean, readable and flexible GUI code is hard. Different Layout classes (GridLayout vs. GridBagLayout vs. BorderLayout) have different constraints and some of the constraints might break / be removed if we try to manually set a component's positions, resulting to unpredictably undesired results.