Opus Card Machine

Astghik Minasyan

Vanier College

420-SF2-RE: Data Structures And Object Oriented Programming

Yi Wang

May 11, 2025

Table of Contents

Project Description

Scenario:

A person wants to buy bus passes on their opus card or to check how many they have left.

Design Paradigm and Expected Output:

With this machine, the user can choose to check their card balance, to buy tickets or to cancel the latest transaction. When the user checks their card, they can see how many trips they have left or if their card is charged for the month or for the week. When the user wants to buy tickets, they have the option to buy individual trips, to get a monthly pass or a weekly pass. If they regret having done a transaction, they have the option to cancel it. However, they can only cancel the latest transaction.

To be able to do this, the user would have to write their card's ID. If they don't have an ID yet, then they would have to register by putting their name, last name and their status (student or a normal user) and the machine would provide them with an ID for their card. If a user forgets their card ID, they'll have to call someone who will get access to all the accounts by writing a passcode. They'll then be able to see what ID their name is registered under.

Hierarchies:

There are two hierarchies: card and ticket

Card: the user can have a student card (with an applied discount) or a normal card.

Ticket: the user can buy a monthly ticket, a weekly ticket or an individual trip.

Interfaces:

Rechargeable: this interface is implemented by the Monthly and the Weekly classes. It contains a method recharge() that allows the user to activate the monthly or the weekly passes. It needs to be an interface since the IndividualTrip class inherits from the same parent as Monthly and Weekly, but it's not rechargeable.

Runtime-Polymorphism methods:

The recharge() method in the Monthly and the Weekly classes that apply runtime-polymorphism since it's in the Rechargeable interface and it is written in different ways in those classes, the method is overridden.

TextIO:

The Accounts class uses textIO to write the accounts in a file and to read from the file and put it in the TreeSet of cards. It is also used in the Card class to write the next Id to a file, so that the field will be updated every time a card is registered.

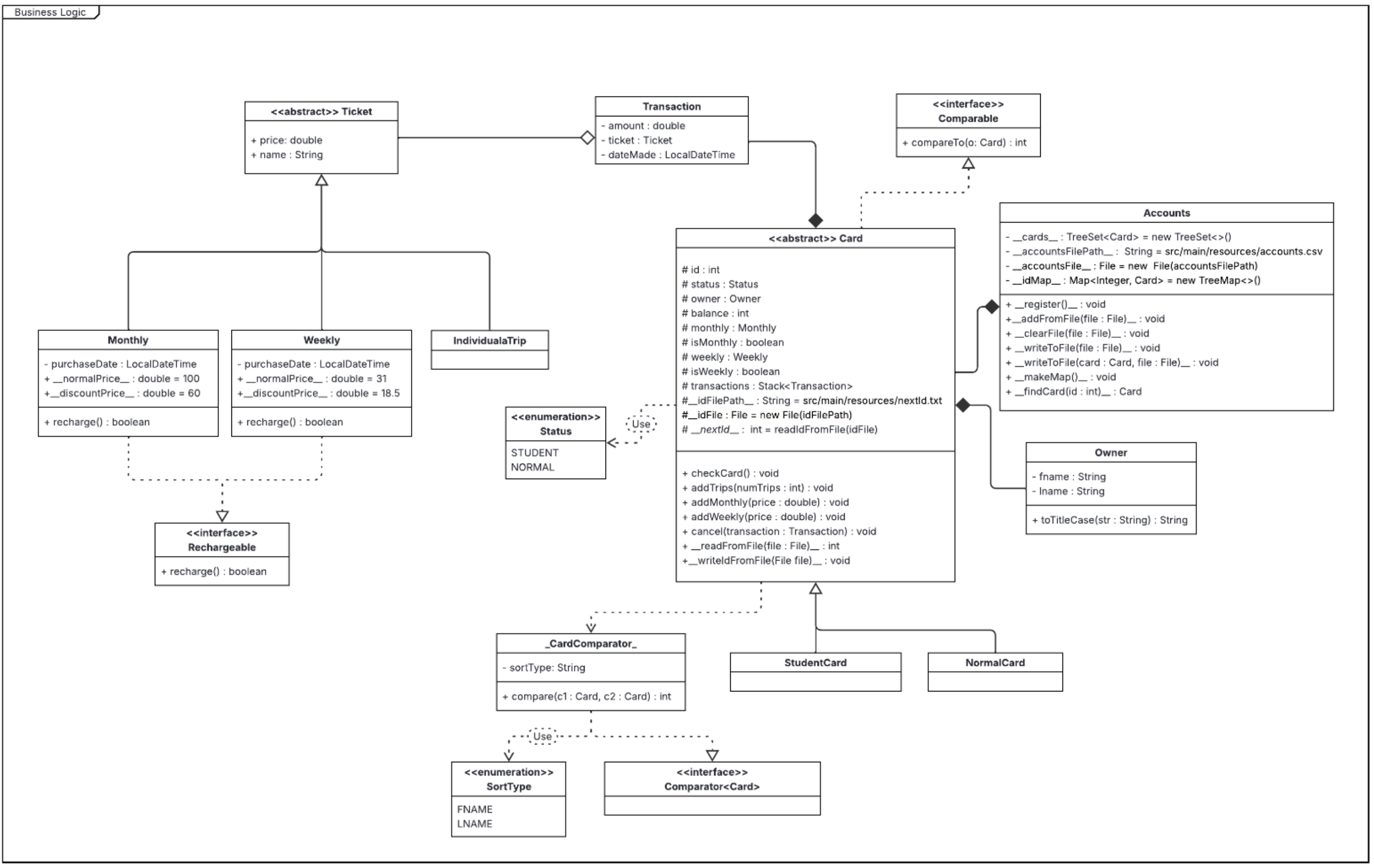
Comparable and Comparator:

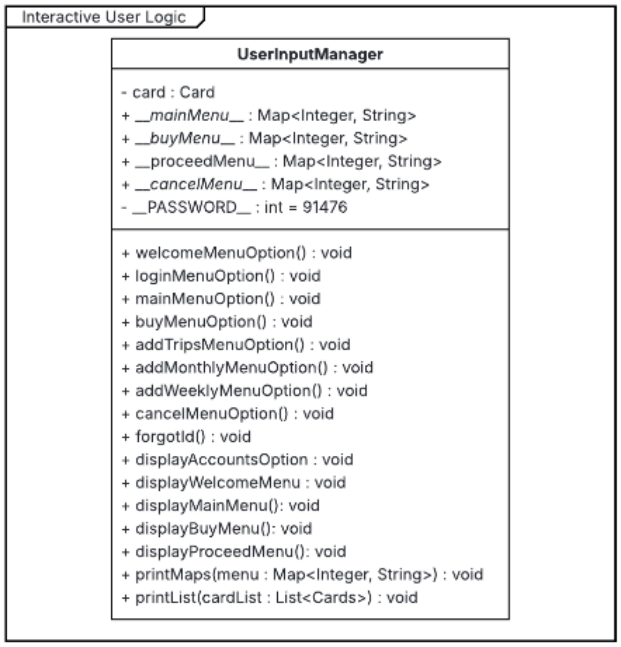
The Card class implements Comparable. Since Cards is an object, the TreeSet in the Accounts class needs to know how to sort the Cards. This class also needs a Comparator to sort and display the accounts the way the person who checks the accounts wants it (by first name or by last name).

Deliverable 2:

All the methods in the Card class (and its children), in the Owner class, in the Ticket, (and its children) and the Rechargeable interface will be implemented for deliverable 2. The methods with scanners and TextIO will be done if time will permit it, or else they will be done for deliverable 3.

UML class diagram


Business Logic
<<abstract>> Ticket
+ price: double
+ name : String
Transaction
- amount : double
- ticket : Ticket
- dateMade : LocalDateTime
<<interface>>
Comparable
+ compareTo(o: Card) : int
Accounts
- __cards__ : TreeSet<Card> = new TreeSet<>()
- __accountsFilePath__ : String = src/main/resources/accounts.csv
- __accountsFile__ : File = new File(accountsFilePath)
- __idMap__ : Map<Integer, Card> = new TreeMap<>()
+ __register()__ : void
+__addFromFile(file : File)__ : void
+ __clearFile(file : File)__ : void
+ __writeToFile(file : File)__ : void
+ __writeToFile(card : Card, file : File)__ : void
+ __makeMap()__ : void
+ __findCard(id : int)__ : Card
<<abstract>> Card
# id : int
# status : Status
# owner : Owner
# balance : int
# monthly : Monthly
# isMonthly : boolean
# weekly : Weekly
# isWeekly : boolean
# transactions : Stack<Transaction>
#__idFilePath__ : String = src/main/resources/nextId.txt
#__idFile : File = new File(idFilePath)
# __nextId__ : int = readIdFromFile(idFile)
+ checkCard() : void
+ addTrips(numTrips : int) : void
+ addMonthly(price : double) : void
+ addWeekly(price : double) : void
+ cancel(transaction : Transaction) : void
+ __readFromFile(file : File)__ : int
+__writeIdFromFile(File file)__ : void
Monthly
- purchaseDate : LocalDateTime
+ __normalPrice__ : double = 100
+__discountPrice__ : double = 60
+ recharge() : boolean
Weekly
- purchaseDate : LocalDateTime
+ __normalPrice__ : double = 31
+__discountPrice__ : double = 18.5
+ recharge() : boolean
IndividualaTrip
<<enumeration>>
Status
STUDENT
NORMAL
Use
Owner
- fname : String
- lname : String
+ toTitleCase(str : String) : String
<<interface>>
Rechargeable
+ recharge() : boolean
CardComparator
- sortType: String
+ compare(c1 : Card, c2 : Card) : int
StudentCard
NormalCard
Use
<<enumeration>>
SortType
FNAME
LNAME
<<interface>>
Comparator<Card>

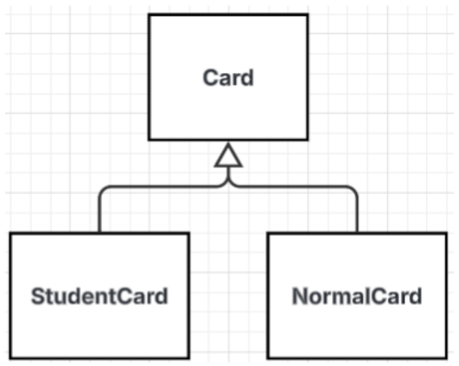

Interactive User Logic
UserInputManager
- card : Card
+ __mainMenu__ : Map<Integer, String>
+ __buyMenu__ : Map<Integer, String>
+ __proceedMenu__ : Map<Integer, String>
+ __cancelMenu__ : Map<Integer, String>
- __PASSWORD__ : int = 91476
+ welcomeMenuOption() : void
+ loginMenuOption() : void
+ mainMenuOption() : void
+ buyMenuOption() : void
+ addTripsMenuOption() : void
+ addMonthlyMenuOption() : void
+ addWeeklyMenuOption() : void
+ cancelMenuOption() : void
+ forgotId() : void
+ displayAccountsOption : void
+ displayWelcomeMenu : void
+ displayMainMenu(): void
+ displayBuyMenu(): void
+ displayProceedMenu(): void
+ printMaps(menu : Map<Integer, String>) : void
+ printList(cardList : List<Cards>) : void

Program Features

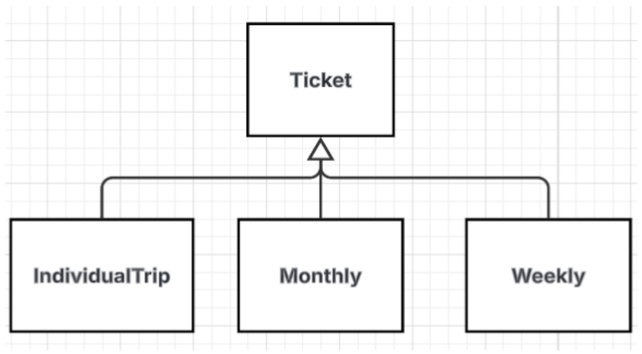
The project meets the requirements by implementing these features:

- A field in the Card class, called transactions, is a Stack of Transaction. A stack is used because the user can only cancel the latest transaction, so only the last element of the stack needs to be accessed. Also, a field in the Accounts class, called cards, is a TreeSet of Card. A TreeSet is used because the same card shouldn't be added twice and I wanted to define the way the elements are sorted (by id).

- There are two hierarchies in this project and each one has two layers. There is StudentCard and NormalCard that extend from Card and there is IndividualTrip, Monthly and Weekly that extend from Ticket.



Card hierarchy

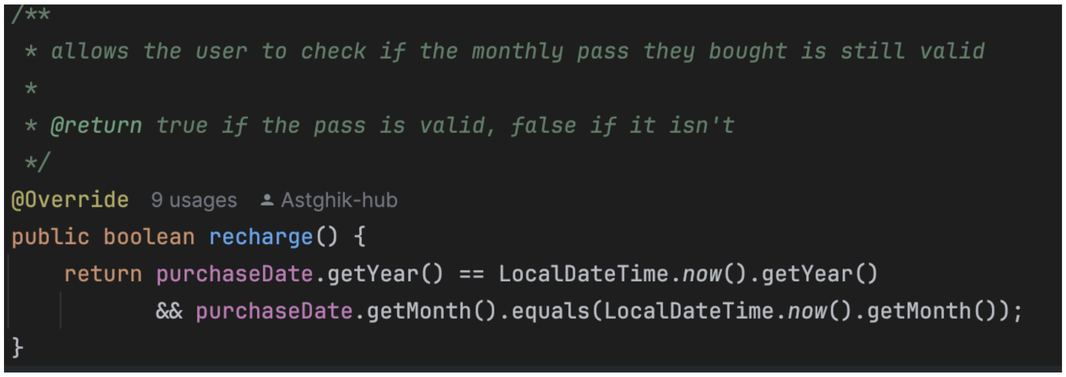


Ticket Hierarchy

- The project has a user defined interface Rechargeable with and abstract method recharge() in it

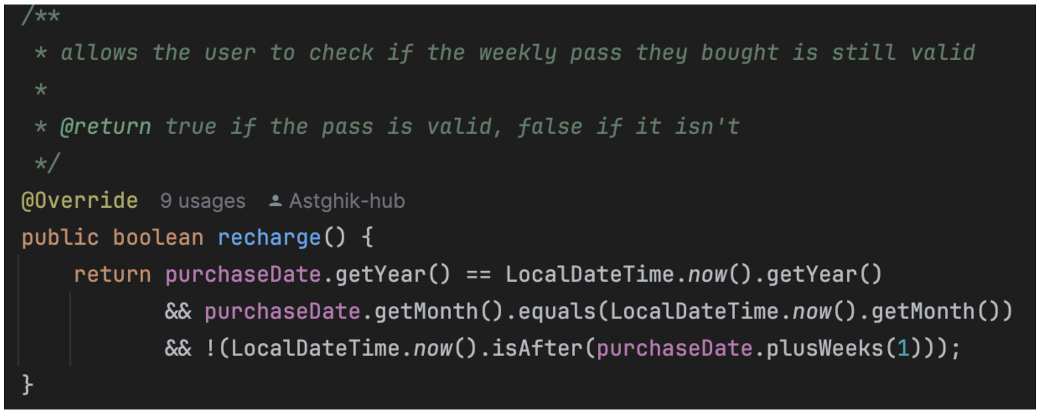
```
public interface Rechargeable {
   boolean recharge();
}
```

- Runtime-polymorphism is applied on the method recharge in the Monthly and Weekly classes. Both implement the same method but have a different body.



```
/**
 * allows the user to check if the monthly pass they bought is still valid
 *
 * @return true if the pass is valid, false if it isn't
 */
@Override  9 usages  Astghik-hub
public boolean recharge() {
    return purchaseDate.getYear() == LocalDateTime.now().getYear()
            && purchaseDate.getMonth().equals(LocalDateTime.now().getMonth());
}
```

The method in the monthly class

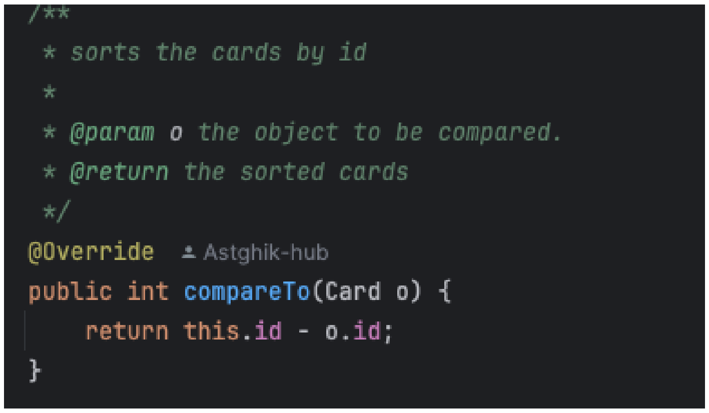


```
/**
 * allows the user to check if the weekly pass they bought is still valid
 *
 * @return true if the pass is valid, false if it isn't
 */
@Override  9 usages  Astghik-hub
public boolean recharge() {
    return purchaseDate.getYear() == LocalDateTime.now().getYear()
            && purchaseDate.getMonth().equals(LocalDateTime.now().getMonth())
            && !(LocalDateTime.now().isAfter(purchaseDate.plusWeeks(1)));
}
```

The method in the Weekly class

- Two classes use testIO (reading and writing): Accounts and Cards. When running the project, the addFromFile method in the Accounts class is called in order to read the cards from the file and add them to the TreeSet of Cards. The writeToFile methods in that same class allow to write a newly registered card to the file or to modify a cards field when necessary (buying bus passes or canceling transactions). In the Card class, when a new instance of StudentCard or NormalCard is created, the readIdFromFile reads the id from the file, allowing to attribute that id to the newly created card. The writeIdToFile writes the next id to the file, allowing the value to be updated.

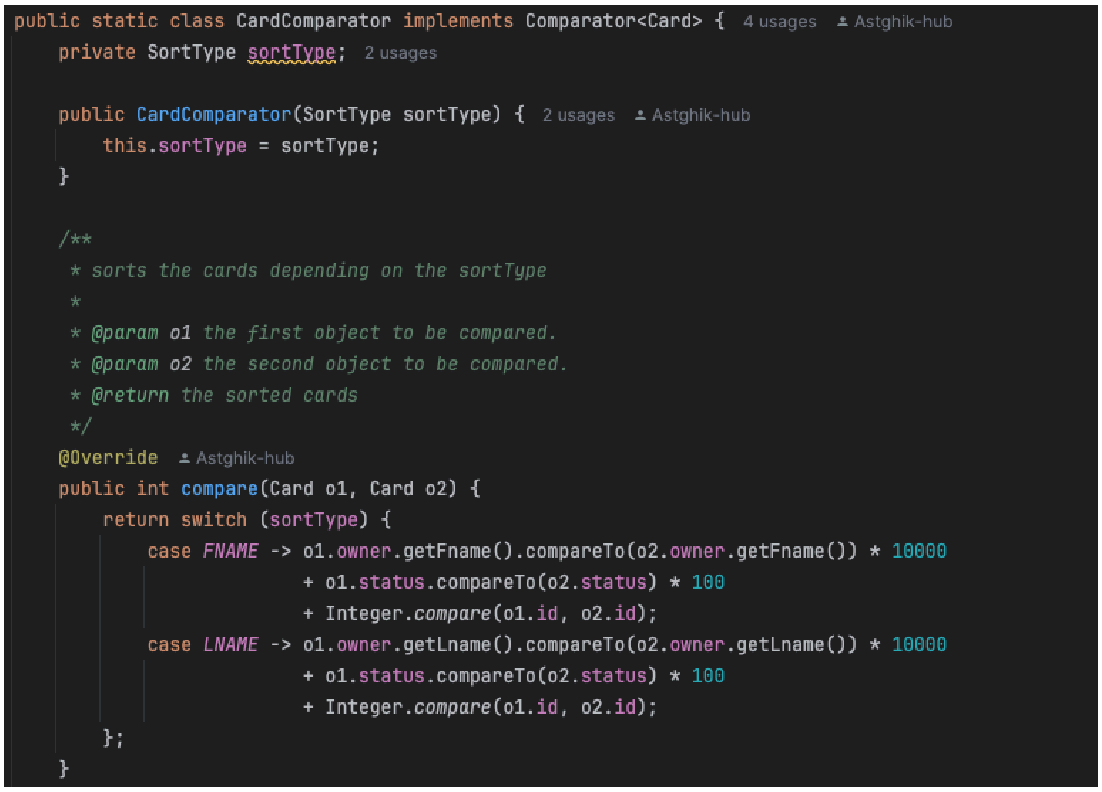
- Since Cards is an object, the TreeSet in the Accounts class needs to know how to sort the Cards. Therefore, it implements Comparable.



```java
/**
 * sorts the cards by id
 *
 * @param o the object to be compared.
 * @return the sorted cards
 */
@Override  Astghik-hub
public int compareTo(Card o) {
    return this.id - o.id;
}
```

Comparable

This class also uses a Comparator class to sort and display the accounts the way the person who checks the accounts wants it (by first name or by last name).



```java
public static class CardComparator implements Comparator<Card> {  4 usages  Astghik-hub
    private SortType sortType;  2 usages

    public CardComparator(SortType sortType) {  2 usages  Astghik-hub
        this.sortType = sortType;
    }

    /**
     * sorts the cards depending on the sortType
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return the sorted cards
     */
    @Override  Astghik-hub
    public int compare(Card o1, Card o2) {
        return switch (sortType) {
            case FNAME -> o1.owner.getFname().compareTo(o2.owner.getFname()) * 10000
                          + o1.status.compareTo(o2.status) * 100
                          + Integer.compare(o1.id, o2.id);
            case LNAME -> o1.owner.getLname().compareTo(o2.owner.getLname()) * 10000
                          + o1.status.compareTo(o2.status) * 100
                          + Integer.compare(o1.id, o2.id);
        };
    }
```

Comparator class

- Unit testing was applied to all non-void user defined methods. In this case, the addFromFile and findCard methods in the Accounts class, the readIdFromFile in the Card class, the recharge methods in the Monthly and Weekly classes and the toTitleCase method in the Owner class were tested using unit tests.

Challenges

The only big issue faced during development was the exception handling. In most cases where I used exception handling, the user was supposed to provide an input to the console. The try block had the Scanner methods in it and the catch block handled input mismatch exceptions. As long as the catch block was triggered, that part of the code was supposed to loop. It was challenging because I haven't had a lot of opportunities to implement exception handling, so I didn't have a lot of practice with that. The code crashed many times but in the end I figured out how to effectively use it.

Learning Outcomes

I gained a lot of experience with object oriented programming. I bettered my understanding of how class hierarchies work and I learned when to integrate interfaces. I also learned the difference between extending from a class and implementing an interface. I effectively demonstrated polymorphism by using both method overloading and overriding and I was able to better visual their difference. Utilizing multiple data structures allowed me to see different ways in managing and accessing data efficiently and when and why I should use them. I also became more confident in handling unexpected program behavior through exception management and improved my ability to read and write data using external files. Streams and lambda expressions helped me write simpler and more readable code. I especially learned a lot on how and why to use git repositories and how to manage code changes, which I hope will make it easier to write collaborative projects with other people in the future. Finally, writing unit tests early helped make sure everything worked as expected. Therefore, I didn't have to wait until I wrote my whole code to know if my methods worked or not and I learned that it is way more efficient than to test them manually.