CSC207: Project Phase 1 Introduction

Introductory Activity

With your project group, do a CRC analysis of the project described here. Make sure that your cards include both responsibilities (written in English rather than code!) and collaborations.

Learning Objectives

- Do a CRC design
- Use Git in a group setting
- Build a graphical user interface using JavaFX, Swing, or something similar(Phase 2)
- Write a "real" application
- Design and work with configuration files
- Implement a logging system
- Work with all Java features seen so far
- Work in an environment where not everything is specified

Overview

This project description is a first introduction to the project. Phase 2 will have a more complete set of requirements, with extra features, a graphical user interface (GUI), and the chance to make your project unique by adding features that are specific to only you project.

Large cities often have a public transit system that includes trains, trolleys, busses, etc. You will be creating a program that tracks and calculates fares for anyone who uses a travel card (called a "cardholder") to enter and exit the system.

Your program should work with a transit network that contains only underground trains (called "subways") and above-ground buses.

Fares are calculated using a combination of time and distance, as measured in minutes and number of stops/stations respectively. (For a definition of "stop" and "station", see the section below called "Transit System".)

Your system will be used by cardholders and transit staff (admin users).

It is a good idea not to hard code values into your program so that it can be used in more than one city, where fares, transit maps, caps, etc. may differ from each other.

Fares:

Each bus ride costs $2. In other words, $2 is deducted from the travel card's balance every time someone enters a bus. Unlike the TTC, this system does not have transfers - fares are deducted from the cards balance every time the cardholder enters a vehicle.

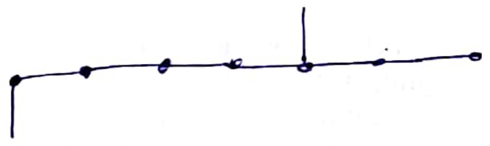
Subway rides cost $0.50 for ever subway station reached by the rider. This will include every station except for the first one. For example, entering at Station A, riding through Station B, and then exiting from Station C will cost $1 because the user travelled to B and to C, which is two stations. Entering a subway station and immediately exiting is free.

Fares are deducted from the card balance every time cardholder exits a station. Nothing is deducted upon entry into a station. This is opposite to the way money is deducted upon entry to (not exit from) a bus.

You can assume that there is a different input when a card taps to enter a station and to exit a station. In other words, the input will tell you whether a person is entering or exiting a station/bus.

It is possible for someone to illegally enter or exit a station without tapping or simply forget to tap on the way out of a bus. Alternatively, the hardware could malfunction due to a power outage, causing non-standard tapping sequences. It is up to your group to decide how you want to handle these pathological cases (e.g., someone appears to tap out of a station without ever tapping into the system, etc.). It is up to you to design a way of handling these situations that will be appealing the your customer (the government who runs the transit system).

Transit system:



Bus routes all include at least one station. They have a starting point and an ending point and do not return to the same location. This information may or may not be useful for your design. You have to decide.

For Phase 1, you can assume that there is only one subway line and that it does not intersect itself.

For phase 1, you can also assume that all bus routes include at least one station and do not intersect themselves.

A continuous trip through the system involves entering and exiting subways and buses at points where their routes intersect.

Points on the subway line are called "stations" and points along a bus route are called "stops". When a cardholder "taps out" (exists) a bus, the time and location name of the stop is recorded, to check if the next entrance is at the same location. If not, the next entrance is considered to be the beginning of a new trip. This is relevant when calculating caps (see Travel Restrictions).

Travel Restrictions:

The system keeps track of the times when a card taps into and out of each subway station and/or bus.

Individual trips are capped at $6. In other words, all entrances and exits from the transit system by an individual cardholder within a 2 hour period are deducted from the card's balance up to a maximum of $6, after which travel is free, as long as all entrances are within 2 hours of the initial entrance that began the trip.

If a person rides the subway for three hours without exiting, the most they can be charged is $6. However, if they exit the subway after riding for three hours and then enter a bus, they will be charged separately for the bus because they entered it more than two hours after beginning their trip.

The two hour cap does not apply to disjointed trips. For example, if a cardholder exits from subway station and then enters a bus that does not stop at that station, the bus entrance is not counted towards the last trip.

Instead, a new 2-hour period has started for this cardholder.

Cardholders:

Each transit user has an account that stores their name, email address, and links to any and all travel cards.

Users can change their name, but not their email address. They can also add and remove as many travel cards as they want.

Cards each have an associated balance that must stay in positive numbers (or $0) with one exception: If the next entrance (into a subway or bus) will move your balance from a positive to a negative number, the system will let you pay, but then prevent any further entrances until you add to your card balance.

Cardholders can add $10, $20, $50 to their card balance at a time. The mechanism for doing this is outside of your program. All you have to do is ensure that there are methods for incrementing the balance by those amounts and that inputting a request to increase the balance does, in fact, increase the balance.

All new cards start with a $19 balance.

Users can also view their three most recent trips, suspend a stolen card's activity, and track their average transit cost per month.

Admin Users:

The people who run the transit system should be able to compare overall revenue to their operating costs. They will do this by comparing all fares collected each day with the total number of stops/stations reached by all cardholders. In Phase 2, you may be asked to keep more detailed statistics, so your system should be extensible in that direction.

For phase 1, the daily report can be printed to the screen or a file.

You can decide how you want to handle the separation from one day to the next. This can involve input from the user. Alternatively, your system can assume that the program runs continuously during the day and then is closed at the end of each day. This last option requires you to have a method of input for ending execution.

System Boundaries:

Each entry and exit of a card from/to a station or bus is considered to be input into your program.

Transit user interactions with their account involves both input and output (viewing/modifying their balance, adding/removing cards, changing their name, viewing their recent trips, etc.)

Any messages to the user are considered to be output. For example, telling a user that they cannot enter because their balance is already negative. On the TTC, this looks like a red X on the screen when a cardholder attempts to tap into a station or onto a bus.

Any confirmation messages to the user is considered to be output. For example confirming that money has been added to a card's balance is considered to be output.

All requests from admin user(s) are input. The resulting information displayed to the admin user(s) is output.

events.txt

You will have a text file called events.txt that contains one line for each individual input. For example, you can have a line that says "1204 enters St. George Station". Alternatively, the same input could be formatted as "St. George Station entry Card 1204". Or you could decide to include more or less information for that particular input. You get to decide on the format for each line of events.txt

In Phase 2, you will have a user interface that will replace Events.txt so that all input will come from the user. Phase 1 is the backend of your completed Phase 2 program.

All output can be written to the screen using text for Phase 1. For example, your program can have println statements saying this like "$20 has been added to the balance of card 12345670987".

README.txt

You will create a file called `README.txt` that will instruct the marker on how to run your program, what format to use for each type of input into `events.txt`, and any other information we need to set up and run your code. If you use other configuration files besides `events.txt` please include detailed instructions so that we know how we can and cannot modify those files in order to get your program to run.

design.pdf

Once you have completed your code for phase 1, create a uml diagram with all of your classes, their variables and methods, and the following relationships: inheritance, aggregation, and association. Further references for uml diagrams will be posted as Readings on the course website's Lectures page.

Getting Set Up

Once all of the groups have signed up on MarkUs, we will send out an email announcing that the group repositories can now be created. At least one person must log into MarkUs and click on `phase1` to create your repository. The following instructions assumes that this has already happened:

You have a shared repository whose name is your group name. For example, if your group name is `group_4321`, then your repository URL will be `https://markus.teach.cs.toronto.edu/git/csc207-2018-05/group_4321`.

Your repo has a `phase1` subdirectory with this file inside. Create a new IntelliJ project inside it. This will automatically create a `src` subdirectory. Create any packages you like. Only the contents of the `phase1` folder will be graded.

Only add and commit your `.java` files and any text files you create. **In particular, do not add or commit the `out` directory, nor any other automatically-generated files created by your IDE.** You can add other subdirectories and files for things like unit tests and a TODO list, if you want.

Citing Code

If you use any code you find, cite it according to the format described in the "Examples of citing code sources" section of the "Writing Code" page of the MIT Academic Integrity handbook.

Further Requirements

The user is likely to add further requirements over the next month or so. The requests might involve expanding your software to include more features or handle more inputs. Be sure to design your code with this in mind.

You will see these further requirements in the phase 2 handout. There may also be clarifications posted to the message board. Make sure that at least one group member is monitoring the message boards on any given day.

In real life, you would be able to ask a contact with the municipal government for further clarification regarding the software they want from you. For the purposes of this project, you can direct such questions to the discussion board. Any response from Lindsey or David is to be taken as the government's response. You are also invited to do your own research regarding transit systems and to ride one. For example, what does "tap into" and "tap out of" a station really mean?

Preview of Phase 2:

In Phase 2, you will be asked to extend your program in a few different directions. You will also be asked to create a user interface with buttons, windows, etc, to eliminate all `println` statements, and any exceptions unless you have a really good reason for keeping them.

Logging and design patterns will be a part of Phase 2, as well.

You will also be asked to add your own extra features to make your program unique and more marketable. It would helpful to brainstorm what those features might be during Phase 1, even if you do not add them.

It is IMPORTANT to finish the Phase 1 functionality before attempting to start Phase 2. We will be marking Phase 1 as a complete program. Adding extra, unfinished pieces of Phase 2 may negatively impact how your design for Phase 1 is graded. Instead we suggest:

1. Finish Phase 1.
2. Look for places where adding extra features or front end may be difficult.
3. Improve the design for the parts identified in the previous step.
4. Then start Phase 2.

You can also start looking at JavaFX, Swing, or any other Java supported GUI framework in advance of Phase 2, if you finish Phase 1 early.

How to Get a Good Mark:

Each group's Phase 1 should have similar functionality, even if each design is different from one group to the next. Try to make your design encapsulated so that a change to one part of the program does not have much impact on other parts of the program. Consider each of the SOLID principles when making design decisions. Likewise, we will be discussing design patterns on July 5 and 12 during lecture. For Phase 2, see if any of the design patterns can improve your design.

There is more than one possible design that will work well with this project. You only have to develop one of these designs. If you are having trouble making major decisions about how the overall design of your project should look, feel free to ask your TA during lab, attend office hours as a group or to book an appointment with an instructor to discuss it.

This is a course called Software Design. Therefore, most of your mark will come from your design.

Labs

All labs for the rest of the semester will be devoted to working on your group project.

Please make sure to register your group on MarkUs so that we can set up your git repository and assign your group to a lab room.

What To Submit

As you work, regularly commit and push your changes. We will be checking the git logs to make sure everyone is making a significant contribution. Try to make your last changes to the code at least one day before the due date. That will give you enough time to finalize your uml diagram, update your `README.txt` file and check that your program still works after everyone's work has been merged for the last time.

We will be looking for:

- A completely functional program that meets the phase 1 specifications
- `events.txt`
- `README.txt`
- `design.pdf`
- Any other configuration files

Your code should be platform agnostic: it must run on any operating system that has the Java 8 VM (or higher) installed. We will be running it on the teach.cs servers, which run Linux.

Cards = enter exit - fill card
Subway Station - ~~[illegible]~~ prev - next
Bus stop
User
Admin User
Main

have a file of constants like fares.