CENG 443 – Object-Oriented Programming Languages and Systems

Spring 2018 - Homework 1

Hitcoin Business

Selim Temizer

Feedback : Between April 16th and April 20th, 2018
Due date : April 22nd, 2018 (Submission through COW by 23:55)

In this homework, we will build an animation about traders sending buy/sell orders to the market to trade ***Hitcoins***, and some con men trying to intercept the orders and steal traders' money. We will use object-oriented design principles when building the animation. Basically, we will need the following classes to implement our application:

- ***Animation Entities***: These entities consist of a dynamic plot of hitcoin prices, traders, orders, con men, and a price ticker. Some entities are stationary and some are mobile. Most animation entities will have various strings (like names, and other related data) displayed aside.
    - ***Hitcoin Price Plot***: A dynamic plot at the top portion of the window will display live hitcoin prices and some recent history.
    - ***Traders***: Stationary. Each trader has a name, a nickname, number of coins, amount of capital, and a dynamic net worth (which is equal to the current price of hitcoin times the number of coins the trader has, plus the capital). Traders randomly generate buy or sell orders, and they shoot their orders to randomly chosen points at the base of the hitcoin price plot.
    - ***Orders***: An order has a random speed, a random size (number of coins to be bought/sold) and a side (buy or sell). Orders are represented by filled circles labelled with the initials of the originating trader and the size information. Following the old tradition, buy orders are green and sell orders are pink. When an order hits the base of the hitcoin price plot, the order is executed for the historic price that was recorded at the hit point.
    - ***Con Men***: They can *CHASE CLOSEST* order around (moving towards the closest order with a random speed), pick a random target position and *GOTOXY* towards that target, *SHAKE* by random dispositions in horizontal and vertical directions, and *REST* for a while occasionally. If they come in contact with an order, they steal **[order size × current hitcoin price]** amount of money from the order's trader. If the order is a buy order, the trader loses the cash amount described, and if the order is a sell order, the trader loses order size amount of hitcoins.
    - ***Price Ticker***: A scrolling ticker at the bottom of the window displays the price levels for Borsa İstanbul (BİST) 100 companies. The companies should appear in a random order.
- ***Common***: Brings together animation parameters, instances of entities, and other utility fields / methods that are required for running the application.

- ***Display***: Extends *JPanel* and consists of the following logical parts: at the top left, the hitcoin logo should appear. At top right, the hitcoin price plot should be placed. At the bottom, the price ticker should run. Above the ticker, the traders should be located. And in the empty middle region, orders and con men roam around.
- ***HitcoinRunner***: A class that contains the main method.
- ***Position***: An optional class for representing entity positions and containing various utility methods that manipulate position data.

However, we will also use the following software design patterns that will require us to extend our basic design:

- ***Factory Method / Abstract Factory***: Specific factories will create different types of orders. Make sure that the part of the application that creates the orders is unaware of the types of orders (i.e., demonstrate proper use of the factory method / abstract factory design pattern in your implementation).
- ***State***: As a minimum, 4 types of states should be prepared: *Rest*, *Shake*, *GotoXY*, and *ChaseClosest*. At any given time during the animation, each con man will be in a state picked from this list, and act accordingly. It should be possible to easily extend the system with additional states. For proper design, a state should not know about other states.
- ***Decorator***: Each con man will be decorated with colored rectangles to denote how much money s/he has already stolen (2000+ will have an orange badge representing *Novice* level, 3000+ will have a magenta badge representing *Master* level, and 5000+ will have a blue badge representing *Expert* level). Make sure that the code fragment that decorates the con men is outside the *ConMan* class implementation, for good object-oriented design.

Using the above design patterns, we should be able to:

- Prepare an environment with some predefined number of various properly initialized entities. As the animation proceeds, con men will be capable of changing their states as described above. When grading, we need to clearly see the changing states during the flow of the application.
- Decorate con men at run time: As con men steal money, rectangle shaped badges should be added when drawing them. Here, for correct usage of the decorator design pattern, make sure that the *ConMan* class is not aware of the fact that it will be decorated (once or more) somewhere else in the application source code.
- As a suggestion, ideally a *stepAllEntities* method in the *Common* class might ask each entity to act for one step, then might check which entities interact with each other, and might decorate the con men that need to be decorated. This *stepAllEntities* method might be called over and over (with a screen refresh and sleeping for a few milliseconds in between) to create the animation.

A screenshot of a sample implementation is provided in the following part of this document as a reference. We are also required to professionally **design** and briefly **document** our application. For this purpose, we will need to use a specific UML tool (StarUML) to first design the class diagram for our application, generate stub code from it, and then fill out the methods and missing implementation details. (You need to submit the UML document from StarUML application together with your source code). A detailed (but not complete) class diagram is given in the following pages to serve as a starting point and a guideline for you.

What to submit? (Use ***only ASCII characters*** when naming all of your files and folders)

1. A single ".uml" file from StarUML v1, or an ".mdj" file from StarUML v2. Name it as "***Design.uml***" (or "***Design.mdj***"). Start with an EMPTY project. Just have a single class diagram (and maybe optionally a statechart diagram and/or an activity diagram for bonuses). Don't first write the code and then reverse engineer it to get the UML. This homework aims to teach you how to go from a carefully designed UML to the code (this is what you will be hired to do for the rest of your software design careers).
2. Any other documentation that you would like to add (in a directory named "***Docs***").
3. Your source code (all in a single directory named "***Source***"). Do NOT use packages. Do NOT submit binary code or IDE created project files. Just submit your ".java" files. We should be able to compile and run your code simply by typing the following:

```
C:\...\Source> javac *.java
C:\...\Source> java HitcoinRunner <any documented parameters that you might have>
```

Zip the 3 items above together, give the name <ID>_<FullName>.zip to your zip file (tar also works, but I prefer Windows zip format if possible), and submit it through COW. For example:

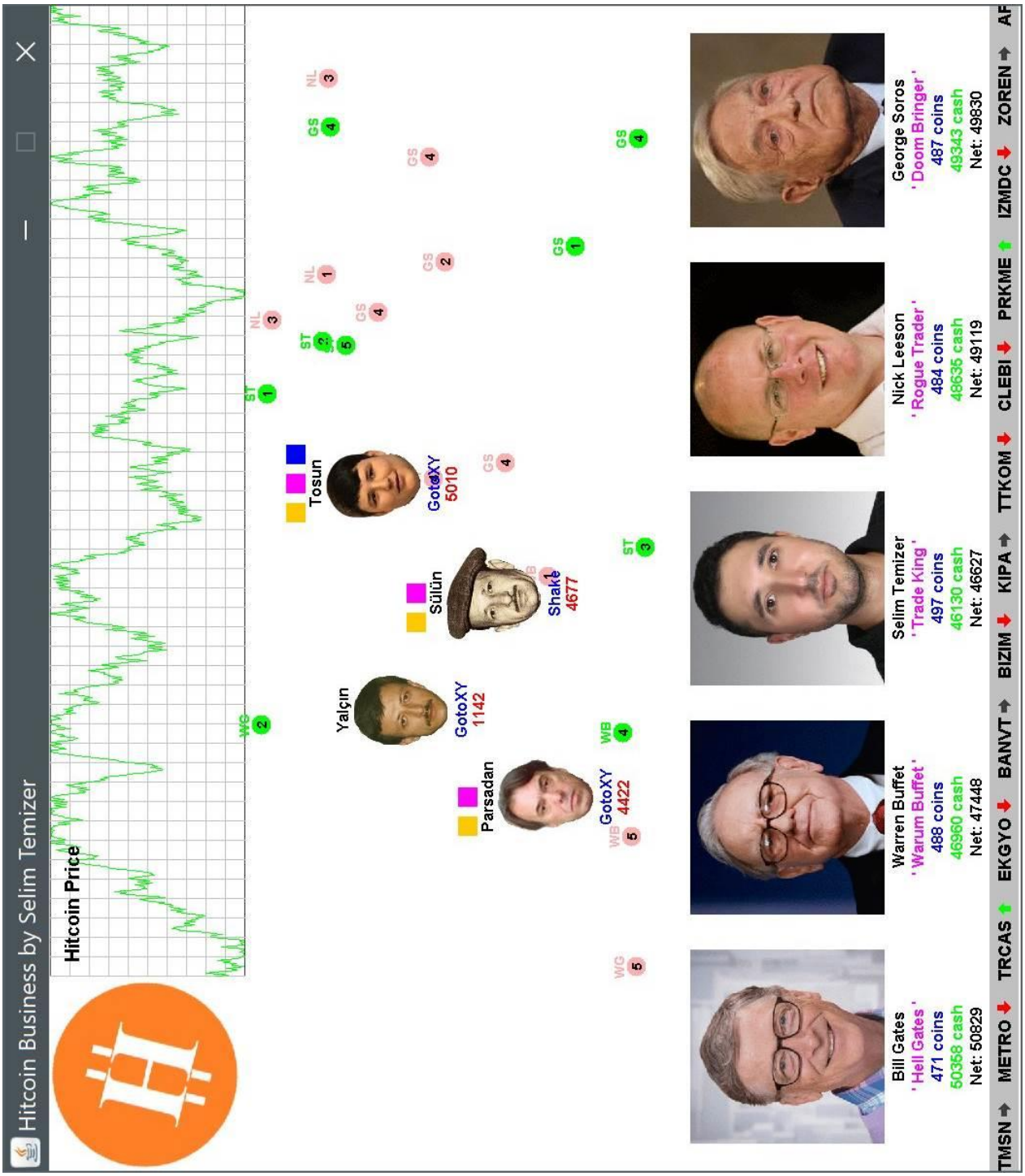
e1234567_SelimTemizer.zip

There are a number of design decisions (example: read all parameter values from editable configuration files) and opportunities for visual improvements (example: draw much nicer figures) and creative extensions (example: additional states) that are deliberately left open-ended in this homework specification. We have enough time until the deadline to discuss your suggestions and make further clarifications as necessary. There will be bonuses awarded for all types of extra effort. Late submissions will NOT be accepted, therefore, try to have at least a working baseline system submitted on COW by the deadline. Good luck.

IMPORTANT: Late submissions (even for 1 minute) will not be accepted!

We will only grade submissions on COW, and system closes automatically at due time!

Sample Screenshot



Sample UML Class Diagram (Draft)