Middle East Technical University Department of Computer Engineering

# CENG 443

Introduction to Object-Oriented Programming Languages and Systems

Spring 2021 - 2022

Homework 1 - Arms Race
version 1.0

Due date: 24 April 2022, 23:59

## 1 Introduction

In this assignment, you will implement a simulation about defence contractors selling arms to countries which buy/sell gold and import goods. When building the simulation, you will employ the object-oriented design principles you learn during the classes. To see the big picture, a video of the sample implementation is provided (link).

## 2 Simulation Entities

You will implement the simulation with the following classes. Do not add new classes or rename/remove the existing ones. You will complete the parts where **TODO** comments are.

1. **SimulationRunner**: Contains the main method. Already filled for you.
2. **Display**: Represents the display on which simulation entities, i.e. countries, orders, and corporations, are repeatedly drawn.
3. **Common**: Brings together simulation parameters, instances of entities, and other utility fields/methods that are required for running the application.
4. **Entity**: Represents a simulation entity. Already filled for you.
5. **Position**: Represents a position (x and y coordinates). Already filled for you.
6. **LivePrice**: Represents the live food/electronics/gold price that changes at random. Already filled for you.

7. **Country**: A country has a name, some gold, some cash, a dynamic worth [cash + gold x current gold price], and percentage of happiness (of its citizens). A country randomly generates gold orders at all times. However, when the happiness is less than 50%, it also generates import orders, i.e., food products orders and consumer electronics orders, to increase its citizens' happiness.

8. **Order**: Base class for **GoldOrder**, **FoodOrder**, and **ElectronicsOrder**. An order has a random amount (between 1 and 5), a random speed, and a random path. An order is a filled circle labelled with the amount information and initials of the originating country. Gold orders are either green or red. While food orders are yellow, electronics orders are blue. When an order hits the horizontal line, just below the live prices, the order is executed for the gold price recorded at the hit time.

9. **GoldOrder**: Base class for **BuyGoldOrder** and **SellGoldOrder**.

10. **BuyGoldOrder** When a buy gold order is executed at the horizontal line, the originating country gains gold by [order amount] and loses cash by [order amount x current gold price]. The color of a buy gold order is green.

11. **SellGoldOrder** When a sell gold order is executed at the horizontal line, the originating country loses gold by [order amount] and gains cash by [order amount x current gold price]. The color of a sell gold order is red.

12. **FoodOrder** When a food order is executed at the horizontal line, the originating country loses cash by [order amount x current food price] and gains happiness by [order amount x 0.2].

13. **ElectronicsOrder** When an electronics order is executed at the horizontal line, the originating country loses cash by [order amount x current electronics price] and gains happiness by [order amount x 0.4].

14. **Corporation**: When a corporation comes in contact with a gold order, it sells arms to order's originating country and gains cash by [order amount x current gold price]. If it is a buy gold order, the country loses cash by [order amount x current gold price]. If it is a sell gold order, the country loses [order amount] gold. Wasting tax money on arms creates some unrest among the citizens and as a result the country's happiness decreases by [order amount x 0.1] as well (for both buy and sell gold orders). Successful corporations are awarded badges. If a corporation's cash exceeds 2000$, it is awarded with a white (rectangular) badge. When the cash exceeds 4000$, it is also awarded a yellow badge. Exceeding 6000$, the corporation is rewarded with a red badge.

15. **State**: Base class for **Rest**, **Shake**, **GotoXY**, and **ChaseClosest**. A corporation has a state which it changes at random.

16. **Rest**: A state where the corporation stays still.
17. **Shake**: A state where the corporation makes random dispositions in horizontal and vertical directions.
18. **GotoXY**: A state where the corporation picks a random position as its destination and moves there with a random speed. When the destination is reached, the above process is repeated.
19. **ChaseClosest**: A state where the corporation picks the closest gold order as its destination and chases it with a random speed. If there are no gold orders roaming around, then the corporation stays still.

## 3 Grading Rubric & Design Issues

- You are supposed to explain your code via comments (15 pts). We will read the comments to understand how your program behaves. You should at least comment on Common.stepAllEntites(), step() of **Corporation** and **Country**, and **State**'s subclasses. You will not get any points for trivial explanations.
- Your simulation implementation should be smooth, similar to the sample implementation (35 pts). Entities should be proportional, i.e., not tiny or giant entities. Entities should not be abnormally positioned. Corporations and orders should move in a normal way, i.e., no instant movements from a position to another position or not moving at all (except for **Rest** state). Orders that hit the horizontal line or absorbed by corporations should not stay in the GUI. Dynamic entities should not move outside the GUI. Your simulation should not crash at runtime and generate any runtime errors. And, obviously, there should not be any compile time errors.
- **State** subclass logic should work correctly (15 pts). A corporation with **ChaseClosest** state should catch the nearest gold order, not randomly moving to a position or chasing a wrong order (i.e., chasing a food/electronics order or non-nearest gold order). This also includes states being randomly changed, i.e., a corporation should not be stuck with the same state at all times.
- **Country** should not know what type of order it has generated, i.e., it just generates an **Order** (5 pts). The other classes should handle the order subclass instantiation logic. This delegation improves the code design since the code of **Country** does not have to change when a new type of order emerges. Similarly, **Corporation** should not know what type of state it has, i.e., it just knows it has a **State** (5 pts).
- Order execution logic should be correct (15 pts). When an order hits the horizontal line or absorbed by a corporation, the related fields of the corresponding country and corporation should be updated accordingly.

- When the happiness of a country exceeds 50%, it should no longer generate food/electronics orders, and vice versa (5 pts).
- Correctly display the badges of a corporation (5 pts).
- How negative gold/cash/happiness is handled is up to you. That is not going to be graded.
- You should read the provided images from "images" folder (so that we can streamline the grading process).
- Your submission will be tested on a computer with a resolution of 1920 x 1080. Either implement a parametric GUI which adjusts itself for all resolutions, or do not change the provided GUI parameters. Also, when testing, we will employ Java 17.
- You can ask your questions related to homework at the discussion forum on ODTUClass.

## 4 Submission

Submission will be done via ODTUClass. You will submit a zip file called **hw1.zip** that contains all aforementioned classes except **Position**, **Entity**, **LivePrice**, and **SimulationRunner**. A penalty of **5 x LateDay x LateDay** will be applied for submissions that are late at most 3 days.