Object-Oriented Programming

Exam Project

Magnus Madsen

magnus@cs.aau.dk

Formalia

You are to solve the programming problems described below. You must write an object-oriented program in Java and document it with program comments. You must describe any simplifications, omissions, and design choices you make. You should include these as program comments, not in a separate report.

Requirements. The program must be written in Java. The program must be well-written, well-structured, and well-explained. You are not allowed to include program parts from other sources (books, papers, from the internet, or from fellow students). You must write your name and AAU email address at the top of every single file.

Individuality. You must write your program alone. You are welcome to *discuss* the project with fellow students. Discussion means talking, not programming.

Workload. Workload of this miniproject: 15 hours, corresponding to two working days.

Submission. You must submit a zip-file that contains the source code of your program. Let me repeat. A *zip-file*, not any other type of archive file.

If you use Intellij IDEA, you are encouraged to include the entire project folder.

Twilight Imperium

Pax Magnifica, Bellum Gloriosum
Magnificent Peace, Glorious Warfare!

Twilight Imperium is a board game of intergalatic conflict, politics, trade, and warfare. In Twilight Imperium, each player controls a race which vies for dominance of the Galaxy. At the center of the Galaxy lies the former imperial seat and mega-world *Mecatol Rex*. Desolate since the fall of the *Lazax empire*, who shall be the first to reclaim its throne?

In this project, you will implement a "baby" version of Twilight Imperium. The rules of baby Twilight are significantly simplified and fully described within this document. For inspiration and flavour, you are welcome to look into the official rule book:

`https://www.fantasyflightgames.com/en/products/twilight-imperium-fourth-edition/`

The structure of this project is as follows: I will present concepts and rules of the game, and then ask you to implement them. This pattern repeats. The programming problems start out easy and increase slightly in complexity.

If a problem is unclear, you should make your own design choice(s). It is expected that you will be able to solve the majority (if not all) of the problems.

Players. A game of baby Twilight is played between two to six players. Each player has a name, a unique race, and a unique color. The game has sixteen races such as: *The Barony of Letnev*, *The Clan of Saar*, *The Emirates of Hacan*, and *The Federation of Sol*.

Problem 1. Write a class to represent a player. The class should have an appropriate constructor and appropriate getters. Implement the `equals`, `hashCode`, and `toString` methods for the class.

Units. In baby Twilight, each player commands a fleet of spaceships. A spaceship has a type, a combat value, a resource cost, a movement speed, and capacity. Every spaceship, in the game, belongs to a specific player. The spaceships are:

Type	Resource Cost	Combat Value	Movement Speed	Capacity
Destroyer	1	9	2	0
Cruiser	2	7	2	0
Carrier	3	9	1	6
Dreadnought	5	5	1	0

Briefly, the attributes of a spaceship can be understood as follows: the *resource cost* is how much the ship costs to produce, the *combat value* is a measure of how good the ship is at fighting, the *movement speed* is a measure how many systems the ship can move, and finally the capacity is a measure of how many troops can be aboard.

Problem 2. Write an interface for units. The interface should have getters to retrieve the resource cost, combat value, movement speed, capacity, and the player who owns it.

Problem 3. Write classes, which extend the interface, for each type of unit.

Baby Twilight is played on a hexagonal grid of systems that contain planets and units.

Planets. A planet has a name and a resource production. Examples of planets are: *Velnor*, *Mirage*, *Perimeter*, *Vega Minor*, *Vega Major*, *Hope's End*, *Rigel II*, and *Industrex*. A planet can produce between zero and six resources.

Problem 4. Write a class to represent a planet.

Systems. A system is a hexagonal region of space that contains planets and ships. A system has up to six neighboring systems. A system may have less than six neighbors if it is on the outer rim. We can identify the neighbors by their position: We have a system to the north, to the north-east, to the south-east, to the south, to the south-west, and finally to the north-west. A system has between zero and three planets. The planets in a system never change. Ships, on the other hand, may enter and leave a system. Figure 1 shows an example of a Galaxy with systems and planets (ships not shown).

Problem 5. Write a class to represent a system. Add an appropriate constructor and methods for ships to enter and leave the system. Add a method to retrieve all the ships currently in the system.

Galaxy. A collection of systems, a grid of hexagons, make up the Galaxy.

Problem 6. Write a class to represent the Galaxy. Add methods to find all (a) systems, (b) ships, and (c) planets in the Galaxy.

Problem 7. Write a method which returns a Galaxy with the following configuration. The Galaxy has two players: *Crassus* playing *The Emirates of Hacan* and *Pompey* playing the *Federation of Sol*. Crassus is blue, Pompey is red. The Galaxy has the systems:

- At the center is a system that contains the planet *Mecatol Rex*.
- To the north is system that contains the planets *Vega Minor* and *Vega Major*.
- To the north-east is an empty system.
- To the south-east is a system that contains the planet *Industrex*.
- To the south is a system that contains the planets *Rigel I* and *Rigel II*.
- To the south-west is an empty system.

- To the nort-west is a system that contains the planet *Mirage*.

The Galaxy contains the following ships:

- In the *Mecatol Rex* system there are two Dreadnoughts and a Destroyer owned by the blue player.
- In the *Vega Minor* and *Vega Major* system there are two Cruisers and a Carrier owned by the red player.

You decide on the unspecified properties of the Galaxy, e.g. planet resources.

Problem 8. Write unit tests to ensure that your above implementation is correct.

Legal Galaxies. A Galaxy is said to be *legal* if it satisfies the following properties:

- The center system must have exactly one planet named *Mecatol Rex*.
- Every planet belongs to at most one system.
- Every system has at most three planets.
- If system A is to the north of system B then the system B is to the south of A. Similarly for the other five compass directions.

Problem 9. Write a method to verify that the above properties are satisfied by a Galaxy. For each property, introduce an exception to be thrown, if that property is violated.

Problem 10. Write a method that returns all ships owned by a player in the Galaxy. The ships must be returned in sorted order by combat value, e.g. a Dreadnought must appear before a Carrier in the returned list. If two ships share the same combat value, the more expensive one must appear first.

Planetary Control. A planet is said to be under the *control* of a player if he or she has a ship in the system that contains the planet, and moreover no other player has a ship in the same system.

Problem 11. Write a method that creates a text file containing a list of players and the planets they control. For example, the output could be:

```
Blue Player (Emirates of Hacan)
    Mecatol Rex

Red Player (Federation of Sol)
    Vega Minor
    Vega Major
```

Problem 12. Write a method that constructs a random Galaxy with the *Mecatol Rex* system in the center and with one layer of systems around it. The systems should randomly have between zero and three planets. At least two systems should contain ships belonging to at least two players.

Space Battles. A space battle occurs when ships of two different players occupy the same system simultaneously. A space battle is only ever fought between two players. Without loss of generality, we may consider these players to be red and blue. A space battle is resolved through a series of *rounds*. At each round, ships from both players gets to fire and score hits. For example, red may score three hits and blue may scores one hit. If so, the blue player loses three ships and red player loses one ship. This continues until one or both players have lost all their ships in the system.

That is, repeat the following until only one player has ships left in the system:

- For each blue ship with combat value v roll a 10-sided die d. If $d \geq v$ then a hit is scored.
- For each red ship with combat value v roll a 10-sided die d. If $d \geq v$ then a hit is scored.
- For each hit scored by blue, remove a red ship from the system.
- For each hit scored by red, remove a blue ship from the system.

The ships are removed in order from lowest resource costs to highest, i.e. a Cruiser is destroyed before a Dreadnought.

Problem 13. Write a method to resolve space combat in a system. The method should return the player who won the space combat.

Problem 14. Prepare your hand-in. Ensure that:

- Each source file contains your name and AAU email address.
- The code is well-formatted (`Code -> Reformat Code`).
- The code compiles (`Build -> Rebuild Project`).
- The code is well-tested and well-documented.
- Your code is contained within one zip file.

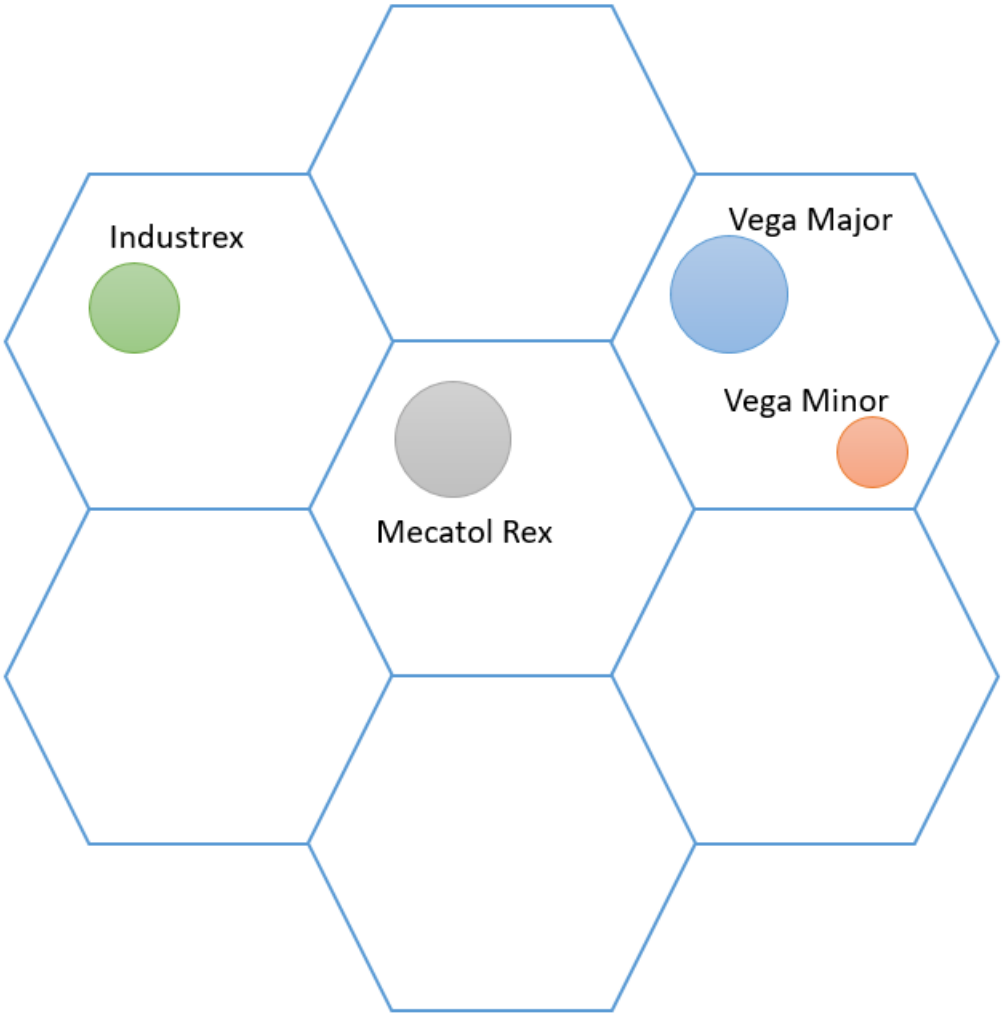

Figure 1: Example of the Hexagonal Galaxy.

Movement. A player may move one or more ships between systems provided that they have sufficient movement speed. For example, a ship with a movement speed of one is only able to reach adjacent systems. A ship with a higher movement speed can reach systems further away.

Problem 15 (OPTIONAL). Write a method to move a ship from one system to another. The method should take the origin system, the destination system, and the ship to move as parameters. The method should return true and move the ship, if the move is legal. Otherwise it should return false.

Hint: Determine the distance between the two systems. Once the distance is determined, it is easy to determine if a move is legal.

Hint: To determine the distance between two systems, perform a breadth first search. The key idea is as follows: Maintain a set of already visited systems and a queue of systems to visit. Initially place the origin system in the queue. The distance to the origin system is zero. While the queue is non-empty, extract a system from the queue, update the distance to all adjacent systems to be one more than the distance to the system itself, and add all non-visited systems to the queue.

This problem is optional. You are eligible for the highest grade without completing it.