Programming Languages Lab – Lab 05

Statement of Purpose

This document helps students complete, by themselves, the mission of this session of the programming languages course.

The student should follow the steps included in this document during the practical session. They can refer to the demonstrator or lecturer regarding any ambiguous points through the session.

For any inquiry about how to write a specific instruction or what is known as *Syntax*, the student must review the book first before referring to the demonstrator or lecturer.

The student should read and learn the *session basics* before starting to solve the *session exercises*.

Session Procedure

The student should work independently within the session. They have to go through the session basics, implement any code and try first hand to solve the exercises found at the end of this document. The student can refer to the teacher when needed. The student can leave the session after completing the exercises and registering their attendance.

If the student has any question regarding the contents of this document, they can immediately ask the teacher by raising their hand or writing their name on the board. The teacher periodically checks the students for questions and helps them complete the tasks needed.

Either the teacher or one of the students on the teacher's request must write the students' names who have attended the session. A student must register their attendance before leaving the session, and it's his/her responsibility. A student is correctly registered as present in their own session only. Any attendance in the wrong session will not be counted towards the total number of attended classes.

Contents

Destroy or Defend break down

We can't start coding the problem before taking high level look about the game environment, players' interactions and game rules. The lack of vision at the beginning may result in limitations in consistency. A systemic look at the problem must first take place. **Worry about implementation details later**. We should start by abstracting the problem using the problem statement.

In the assignment's case, it must be clear that there are concepts about:

- Game;
- Team;
- Player;
- Unit;
- Unit Strategy;
- Unit Property.

The ***Game*** class is responsible for setting up the game properly and progressing the game smoothly amongst players as well as managing roles and ***Unit***s. In addition to decide which ***Team*** won the game.

The ***Player*** class is responsible for representing enough information about the player such as their name. If the player is a human user, then the *Player* class manages I/O from the user and communicates the user's wishes to the ***Game*** class. These wishes are the player's ***Unit***s and actions and the communications could be via console or GUI or maybe mobile or web interface in the future. Since the ***Player*** class will be the game interface to communicate with the external world, it will appropriate to implement the **auto player** as a child class of ***Player***.

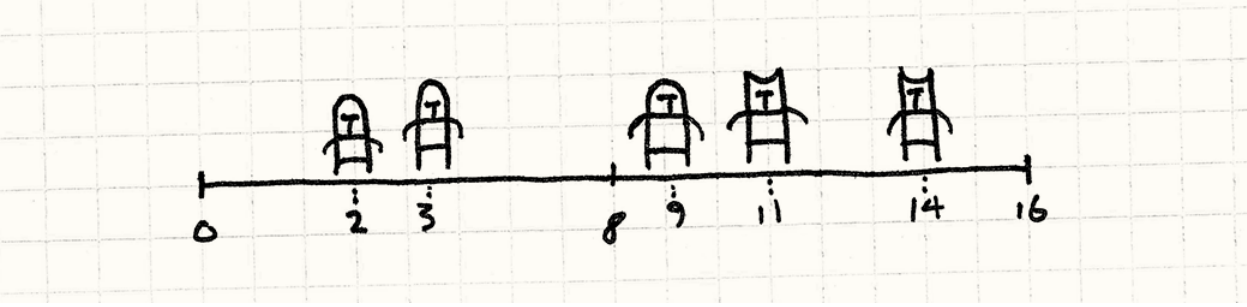
DoD's map is very large and might be 1M X 1M square and that makes it impossible to represent the map as two-dimensional array (10^{12}) square and perform any process by these units. Something like the below unaccepted code!

```
void handleUnitsAttack(Unit[] units, int numUnits)
{
  for (int a = 0; a < numUnits - 1; a++)
  {
    for (int b = a + 1; b < numUnits; b++)
    {
      if (units[a].position() == units[b].position())
      {
        handleAttack(units[a], units[b]);
      }
    }
  }
}
```

Actually, we don't care about all the map's squares, but rather we need to deal with the "*active squares*" only which are the squares that have units on it. This change of thinking leads us to consider the map as **Grid** of **Unit**s.

Grid explanation

The problem we're running into is that there's no underlying order to the array of units. To find a unit near some location, we have to walk the entire array. Now, imagine we simplify our game a bit. Instead of a 2D battle*field*, imagine it's a 1D battle*line*.

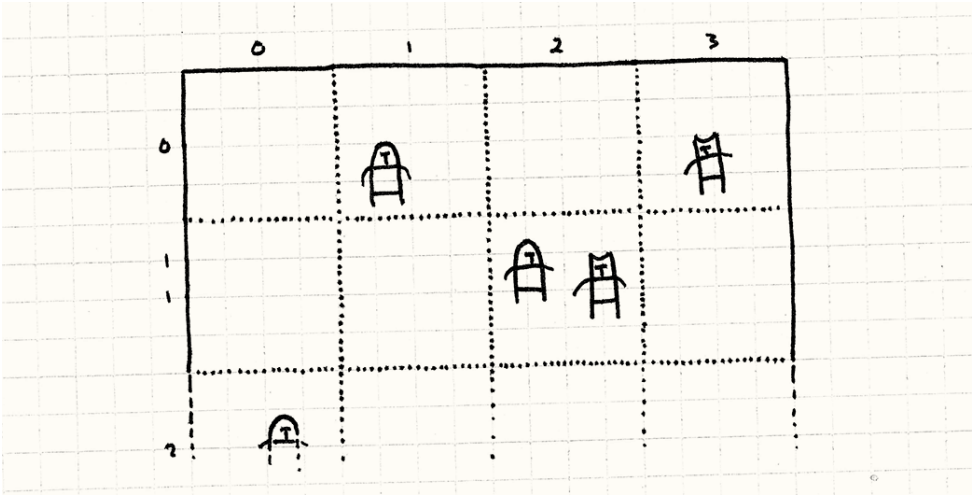


In that case, we could make things easier on ourselves by *sorting* the array of units by their positions on the battleline. Once we do that, we can use something like a binary search to find nearby units without having to scan the entire array. A binary search has *O(log n)* complexity, which means find all battling units goes from $O(n^2)$ to *O(n log n)*.

The lesson is pretty obvious: if we store our objects in a data structure organized by their locations, we can find them much more quickly. This pattern is about applying that idea to spaces that have more than one dimension.

So:

For a set of **objects**, each has a **position in space**. Store them in a **spatial data structure** that organizes the objects by their positions. This data structure lets you **efficiently query for objects at or near a location**. When an object's position changes, **update the spatial data structure** so that it can continue to find the object.



When we handle combat, we only consider units **within the same and nearby cells**. Instead of comparing each unit in the game with every other unit, we've *partitioned* the battlefield into a bunch of smaller mini-battlefields, each with many fewer units.

```
class Unit {
    private Unit _next;
    private Unit _previous;
}

class Grid {
    private static int NUM_CELLS;
    private static int CELL_SIZE;
    private Unit[][] units;

    public Grid(int num_cells, int cell_size) {
        this.NUM_CELLS=num_cells;
        this.CELL_SIZE=cell_size;
        units = new Unit[num_cells][num_cells];
    }
}
```

And when we need to add new unit to the grid we can do something like:

```
class Grid {
    public void addUnit(Unit unit)
    {
        // Determine which grid cell it's in.
        int cellX = (int)(unit.position.centerX / CELL_SIZE);
        int cellY = (int)(unit.position.centerY / CELL_SIZE);

        // Add to the front of list for the cell it's in.
        unit._previous = null;
        unit._next = units[cellX][cellY];
        units[cellX][cellY] = unit;
        if (unit._next != null) unit._next._previous = unit;
    }
}
```

Unit Explanation

Units are the main objects in our game. They attack or defend the base. They move on the map and follow a strategy. If the defender base destroyed then the attackers are the winners. And if the attacker units destroyed or they failed to destroy the defender base within the time, then the defenders are the winner.

We have two important aspects here:

1. Creating these units;
2. Running these units (their movements on the map and their attacks).

Creating these Units

Currently we have 10 types of units and this number might increase any time. It's easy to create a new type of units, so we need to have a dynamic was to extend the units types. And that means we need to deal with large number of classes if go with the option of creating a new class for each unit.

What we know about the unit? All units share some properties (size, attack range, health, armor…etc.)

And some differences like the attack strategy and the movement method (the attacks' units will go straight to the defender base while the defenders' units will go in a patrol or stand at their position)

Both the strategy and the movement method differ from unit to another and will be determine at runtime right before calling the unit's object constructor. So, **we can INJECT these methods to the unit.**

We can use a **Factory** to create these objects. This factory knows all the units' movements and strategies methods.

Running these Units

After creating the units, we need to manage their behavior inside the game (traveling through the map and attacking each other). We can do that by creating a "Units Manger" that goes through the units and call the *move* and the *attackUnit* method sequencelly. Obviously, this solution doesn't achieve the attack speed concept and we can't calculate the remaining game time.

Therefore, we need to put each unit on a separate thread and the unit can work independently on its own thread. The unit will try to move to another square and check there is no unit occupies it. And attack the enemies' units whenever it can.

The current version of game has only a single type of attacks, but what will happen if we added another type of attacks? Like if unit destroyed 3 targets, then will be promoted and it will attack two units at a time instead of one or if the unit will be able to use different types of attacks for different types of targets? Or what if the attack will reduce the health of the targeted unit and increase the attacker armor or anything. That's why we need to make higher abstraction about the units' interactions with each other.

The attacker unit will use an object from **UnitAttack** (build using the decorator design pattern in case our single attack will perform different operations) and apply different **AttackResults** on the targeted unit properties (health, armor, attack speed…). When a unit attacks a unit for the first time, it will subscribe itself on the targeted unit and it will keep attacking it unit it got destroyed.

When this unit's health reach zero, we need to notify the **Game** about this to check what's the impact of this unit destruction on the game state (is it the defender base or the last attack units?).

Notes:

1. Used design patterns in the diagram:
   - **Singleton**: All AttackStrategy's classes, all Movement's classes and Team's classes in addition to Grid and DoDGameManager;
   - **Observer**: **DoDGameManger** observes all the **Unit**s and the **Unit** observes what it attacks;
   - **Decorator**: For **UnitAttack** and **AttackResult**;
   - **Dependency Injection**: The Movement and Strategy methods are injected into the **Unit**;
   - **Factory**: for the units' creation.
2. All attacker units will attack the defender base when it's within their range.
3. Please note that there is a flag on the **GameManager** to tell the game status and all the units should stop their actions when this flag is referring to Pause.

Game Exceptions Model

In this game, there are a few possible exceptions which may occur during input and output. Those exceptions can be dealt with locally within the I/O methods.

However, there are two exceptions which are game related and must be thought of and used properly as a way of carrying messages from one part of the program to another:

- ***IllegalAttack***: which is raised whenever a unit tries to attack another unit but it's already dead (by another attack) happened right before this unit decided to attack it. Or when a unit's AcceptAttack is invoked while the Game is in pause.
- ***IllegalSquare***: which is raised when the **Grid** is trying to move a unit into new position but it finds that it's already occupied buy another unit or it's outside the map.

As a final note, it must be clear that all of the above abstraction steps are mere suggestions to the students. They **do not have to** follow the resultant class diagram. It is up to the students to determine what they use and how. In the assignment handed-in report, students must justify their design decisions.

Destroy or Defend first change

1. The map will contain a valley, river, bridges which will affect the units' movements and attack range.
    - Valley: units can't pass through it;
    - River: units movement speed is reduced when passing through it;
    - Bridges: normal square but might be destroyed.
2. New types of units:
    - **Air Force**. These units travel from their base to the target location to attack, then they go back to the base to reload. These units can attack structures only;
    - **Air Defense**: These units attack the "**Air Force**" units only and they have a huge attack range (details in the Appendix A).
3. These new units will break the concept of movements block (so it's possible one unit on the ground and many in the air have the same position).

4. What's the point of buying bomb cars when the defender will focus it first! So, let's give the players the ability to make some initial plans (for each unit or group of units). The plan is just a series of simple commands: 1- Move to square x, y; 2- Wait in your position for N seconds

After this initial plan completion (for this unit or group of units), the game will continue executing the default strategy.

The first millstone is postponed until December 6th but this first change should be implemented. 7 marks

Session Exercises

1- You may wish to check out the factory, decorator, singleton, observer and dependency injection design patterns;
2- Start implementing the assignment solution.