COMP2500 과제 3

목차

본 과제는 컴퓨터에서 해야 하는 과제입니다. 코드 작성이 끝났다면 실습 1에서 만들었던 깃 저장소에 커밋 및 푸시를 하고 슬랙을 통해 자동으로 채점을 받으세요.

테란 영토에 시빌 워(Civil War)가 시작되고 있습니다. 테란 영토를 침략한 반란군들이 이 지역의 안위를 위협하고 있네요. 멩스크 황제는 테란 군의 최고 군사 전략가인 여러분에게 반란군을 재빨리 처치할 수 있는 군사 유닛(unit)들을 발명하라고 지시했습니다. 다행히도 여러분은 오래전부터 이런 상황에 준비해오고 있었기에 한 달 안에 실전 투입이 가능한 후보 유닛 목록을 가지고 있었습니다.

1. 해병(Marine, 마린)
2. 전차(Tank, 탱크)
3. 망령(Wraith, 레이스)
4. 미사일 포탑(Turret, 터렛)
5. 지뢰(Mine, 마인)
6. 똑똑한 지뢰(SmartMine, 스마트 마인)
7. 파괴자(Destroyer, 디스트로이어)

하지만 그러기 전에 각 유닛의 강점, 약점, 상성 등을 알기 위해 배틀 로열 시뮬레이션을 돌리기로 결정했습니다. 이 시뮬레이션에서 얻은 결과를 이용해 각 유닛의 훈련 프로그램을 개선하려는 게 목적이죠. 하지만 이 새로운 유닛들을 시뮬레이션할 수 있는 프로그램은 아직 존재하지 않습니다. 이 유닛들을 전장에 투입할 수 있도록 빨리 시뮬레이션 프로그램을 만들어 주세요.

1. 프로젝트를 준비한다

1. `Assignment3` 프로젝트를 엽니다.

2. `SimulationManager.java` 파일을 만들어 다음 내용을 추가하세요.

```
package academy.pocu.comp2500.assignment3;

import java.util.ArrayList;

public final class SimulationManager {
    public static SimulationManager getInstance() {
        return null;
    }

    public ArrayList<Unit> getUnits() {
        return null;
    }

    public void spawn(Unit unit) {

    }

    public void registerThinkable(Unit thinkable) {

    }

    public void registerMovable(Unit movable) {

    }

    public void registerCollisionEventListener(Unit listener) {

    }

    public void update() {

    }
}
```

3. IntVector2D.java 파일을 만들어 다음 내용을 추가하세요.

```
package academy.pocu.comp2500.assignment3;

public class IntVector2D {
    private int x;
    private int y;

    public IntVector2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return this.y;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

4. AttackIntent.java 파일을 만들어 다음 내용을 추가하세요.

```
package academy.pocu.comp2500.assignment3;

public class AttackIntent {

}
```

5. Unit.java 파일을 만들어 다음 내용을 추가하세요.

```java
package academy.pocu.comp2500.assignment3;

public class Unit {
    public IntVector2D getPosition() {
        return null;
    }

    public int getHp() {
        return -1;
    }

    public AttackIntent attack() {
        return null;
    }

    public void onAttacked(int damage) {

    }

    public void onSpawn() {

    }

    public char getSymbol() {
        return ' ';
    }
}
```

6. SimulationVisualizer.java 파일을 academy.pocu.comp2500.assignment3.app 패키지 안에 만들어 다음 내용을 추가하세요.

```java
package academy.pocu.comp2500.assignment3.app;

import academy.pocu.comp2500.assignment3.Unit;

import java.util.ArrayList;

/**
 * DO NOT MODIFY. YOU DON'T HAVE TO!
 */
public class SimulationVisualizer {
    private static final int NUM_COLUMNS = 16;
    private static final int NUM_ROWS = 8;

    private ArrayList<Unit> units = new ArrayList<>();
    private ArrayList<Boolean> aliveStatus = new ArrayList<>();

    public SimulationVisualizer(ArrayList<Unit> units) {
        for (Unit unit : units) {
            this.units.add(unit);
            this.aliveStatus.add(true);
        }
    }

    public void visualize(int frameNumber, ArrayList<Unit> units) {
        markDeadUnits(units);

        ArrayList<ArrayList<ArrayList<Unit>>> unitPositions = new ArrayList<>();

        for (int i = 0; i < NUM_ROWS; ++i) {
            unitPositions.add(new ArrayList<>());

            for (int j = 0; j < NUM_COLUMNS; ++j) {
                unitPositions.get(i).add(new ArrayList<>());
            }
        }

        for (int i = 0; i < this.units.size(); ++i) {
            if (this.aliveStatus.get(i)) {
                Unit unit = this.units.get(i);

                int x = unit.getPosition().getX();
                int y = unit.getPosition().getY();

                unitPositions.get(y).get(x).add(unit);
            }
        }

        StringBuilder sb = new StringBuilder();

        sb.append(String.format("Frame: %d", frameNumber));

        sb.append(System.lineSeparator());
        sb.append(System.lineSeparator());

        for (int i = 0; i < this.units.size(); ++i) {
            sb.append(String.format("%s(%c)", Integer.toHexString(i).toUpperCase(), this.units.get(i).ge

            if (this.aliveStatus.get(i)) {
                sb.append(String.format("%02d", this.units.get(i).getHp()));
            } else {
                sb.append("XX");
            }

            sb.append(' ');

            if ((i + 1) % 4 == 0) {
                sb.append(System.lineSeparator());
            }
        }

        sb.append(System.lineSeparator());
        sb.append(System.lineSeparator());

        sb.append("  ");
        for (int i = 0; i < NUM_COLUMNS; ++i) {
            sb.append(Integer.toHexString(i).toUpperCase());
        }

        sb.append(System.lineSeparator());

        addHorizontalBorder(sb);
```

```
        ArrayList<ArrayList<Unit>> overlappedTiles = new ArrayList<>();

        for (int i = 0; i < NUM_ROWS; ++i) {
            sb.append(Integer.toHexString(i).toUpperCase());
            sb.append('|');

            for (int j = 0; j < NUM_COLUMNS; ++j) {
                ArrayList<Unit> uList = unitPositions.get(i).get(j);

                if (uList.isEmpty()) {
                    sb.append(' ');
                } else {
                    sb.append(Integer.toHexString(getUnitId(uList.get(0))).toUpperCase());

                    if (uList.size() > 1) {
                        overlappedTiles.add(uList);
                    }
                }
            }

            sb.append('|');
            sb.append(System.lineSeparator());
        }

        addHorizontalBorder(sb);
        sb.append(System.lineSeparator());

        for (int i = 0; i < overlappedTiles.size(); ++i) {
            ArrayList<String> ids = new ArrayList<>();

            for (Unit unit : overlappedTiles.get(i)) {
                ids.add(Integer.toHexString(getUnitId(unit)).toUpperCase());
            }

            sb.append(String.join("-", ids));
            sb.append(System.lineSeparator());
        }

        System.out.println(sb.toString());
    }

    private void addHorizontalBorder(StringBuilder sb) {
        sb.append(' ');
        sb.append('+');
        for (int i = 0; i < NUM_COLUMNS; ++i) {
            sb.append('-');
        }
        sb.append('+');
        sb.append(System.lineSeparator());
    }

    private int getUnitId(Unit unit) {
        for (int i = 0; i < this.units.size(); ++i) {
            if (this.units.get(i) == unit) {
                return i;
            }
        }

        return -1;
    }

    private void markDeadUnits(ArrayList<Unit> units) {
        assert (this.units.size() >= units.size());

        for (int i = 0; i < this.units.size(); ++i) {
            if (!units.contains(this.units.get(i))) {
                this.aliveStatus.set(i, false);
            }
        }
    }
}
```

7. 자, 이제 자리에서 일어나서 국민체조 한 번 하시고... 설계와 프로그래밍 시작하겠습니다. :)

2. 전쟁 시뮬레이터 설계 및 구현하기

2.1 전반적인 규칙

- 이 과제에서 제공된 파일들에 있는 클래스 이름과 메서드 시그내처를 바꿀 수 없습니다. 단, 다음 메서드들의 매개변수 자료형(param type)은 바꿔도 됩니다.
  - `SimulationManager.registerThinkable()`
  - `SimulationManager.registerMovable()`
  - `SimulationManager.registerCollisionEventListener()`
- 다른 생성자, 메서드, 멤버 변수는 자유로이 추가해도 됩니다.
- 아래 제공된 명세에 맞게 자유로이 클래스를 추가하고 설계하세요. 그러나 빌드봇이 여러분의 설계가 적절하지 못하다 생각하면 점수를 깎을 것입니다. :P
- 모든 클래스는 `academy.pocu.comp2500.assignment3` 패키지에 넣어주세요. 그렇지 않으면 빌드봇이 이 파일들을 무시할 것입니다.
- `instanceof`, `Object.getClass()` 그리고 `Class<T>.cast()` 메서드를 사용할 수 없습니다. 사용하면 곧바로 0점이 뜨니 시도조차 하지 마세요. :)
- `SimulationManager.getInstance()` 메서드 외에 `public static` 메서드는 사용할 수 없습니다.
- 정적 (`static`) 변수는 이 두 가지 경우를 제외하고는 사용할 수 없습니다:
  1. 상수. 단, 이 경우에는 `final` 이기도 해야 합니다.
  2. 싱글턴 인스턴스
- 내포(inner) 클래스 및 내포 열거형의 사용을 금합니다.
- 클래스 내부에 데이터를 저장할 때 사용하는 자료형은 직접 만들어 사용하세요. (예외: String과 HashMap, ArrayList, HashSet와 같은 컨테이너, 그리고 기본 자료형)
- 각 유닛(unit)에 대한 자세한 명세와 시뮬레이션의 규칙은 이 문서 마지막에 있는 부록 A를 읽어 주세요.
- 유닛을 고유하게 식별하는 기준은 그 개체의 참조(레퍼런스)입니다. 따라서 유닛을 깊게 복사하지 마세요.
- 어떤 클래스 대신 사용할 수 있는 기본 자료형이 존재한다면(예: Boolean/boolean, Integer/int) 그걸 대신 사용하세요. 기본 자료형 대신 클래스 버전이 허용되는 경우는 제네릭(generic) 클래스의 타입(type) 매개변수로 사용할 때 뿐입니다.

2.2 `Unit` 클래스를 구현한다

- `getPosition()` : 유닛의 현재 위치를 반환합니다.
- `getHp()` : 유닛의 현재 HP를 반환합니다.
- `attack()` : 공격 의도(`AttackIntent`) 개체를 반환합니다. `AttackIntent` 안에는 공격 위치, 피해치, 공격자 등 공격에 필요한 정보가 담겨 있어야 합니다.
- `onAttacked()` : 유닛에 피해치(damage)를 적용합니다.
- `onSpawn()` : 유닛이 월드에 추가될 때(spawn) `SimulationManager` 가 이 메서드를 호출해야 합니다.
- `getSymbol()` : 유닛을 나타내는 표식(symbol)을 반환합니다. 각 유닛의 표식은 부록 A.3 참고해주세요.

2.3 `Marine` 클래스를 구현한다

- `Marine` 에는 `IntVector2D` 를 유일한 인자로 받는 public 생성자가 있어야 합니다.
- 해병에 대한 명세는 부록 A.3.1을 참고하세요.

2.4 `Tank` 클래스를 구현한다

- `Tank` 에는 `IntVector2D` 를 유일한 인자로 받는 public 생성자가 있어야 합니다.
- 전차에 대한 명세는 부록 A.3.2를 참고하세요.

2.5 `Wraith` 클래스를 구현한다

- `Wraith` 에는 `IntVector2D` 를 유일한 인자로 받는 public 생성자가 있어야 합니다.
- 망령에 대한 명세는 부록 A.3.3을 참고하세요.

2.6 `Turret` 클래스를 구현한다

- `Turret` 에는 `IntVector2D` 를 유일한 인자로 받는 public 생성자가 있어야 합니다.
- 미사일 포탑에 대한 명세는 부록 A.3.4를 참고하세요.

2.7 `Mine` 클래스를 구현한다

- `Mine` 에는 다음의 인자들을 받는 public 생성자가 있어야 합니다.
  - 지뢰의 처음 위치를 나타내는 `IntVector2D`
  - 이 지뢰가 최소 몇 번을 밟혀야 터지는지를 나타내는 `int`
- 지뢰에 대한 명세는 부록 A.3.5를 참고하세요.

2.8 `SmartMine` 클래스를 구현한다

- `SmartMine` 에는 다음의 인자를 받는 public 생성자가 있어야 합니다.

    - 스마트 지뢰의 처음 위치를 나타내는 `IntVector2D`
    - 최소 몇 번을 밟혀야 터지는지를 나타내는 `int`
    - 최소 몇 명의 적을 감지해야 터지는지를 나타내는 `int`
- 스마트 지뢰에 대한 명세는 부록 A.3.6을 참고하세요.

2.9 `Destroyer` 클래스를 구현한다

- `Destroyer`에는 `IntVector2D`를 유일한 인자로 받는 public 생성자가 있어야 합니다.
- 파괴자에 대한 명세는 부록 A.3.7을 참고하세요.

2.10 `SimulationManager` 클래스를 구현한다

- `SimulationManager`는 반드시 싱글턴이어야 합니다.
- `getInstance()`: `SimulationManager`의 인스턴스를 반환합니다.
- `getUnits()`: 시뮬레이션 월드에서 아직도 살아있는 유닛의 목록을 반환합니다.
- `spawn()`: 시뮬레이션 월드에 유닛 하나를 추가합니다.
- `registerThinkable()`: 생각할 수 있는(thinkable) 유닛을 등록합니다.
- `registerMovable()`: 움직일 수 있는(movable) 유닛을 등록합니다.
- `registerCollisionEventListener()`: 유닛을 충돌 이벤트의 구독자(subscriber)로 등록합니다.
- `update()`: 이 메서드를 호출하여 시뮬레이션의 한 프레임을 진행합니다. 이 메서드에서 해야 하는 일은 순서대로 다음과 같습니다.
    1. 각 유닛들이 이번 프레임에서 할 행동(선택지: 공격, 이동, 아무것도 안 함)을 결정
    2. 움직일 수 있는 각 유닛에게 이동할 기회를 줌
    3. 이동 후 충돌 처리
    4. 각 유닛에게 공격할 기회를 줌
    5. 피해를 입어야 하는 각 유닛에게 피해를 입힘
    6. 죽은 유닛들을 모두 게임에서 제거함

3. 본인 컴퓨터에서 테스트하는 법

`academy.pocu.comp2500.assignment3.app` 패키지에 들어있는 `SimulationVisualizer` 클래스를 보면 프레임 번호와 유닛 목록을 매개변수로 받는 `visualize()` 메서드가 있을 것입니다. 디버깅을 할 때 이 메서드를 사용해서 현재 시뮬레이션 상태를 확인해보세요. 다음과 같이 말이죠.

```
SimulationManager simulationManager = SimulationManager.getInstance();

Unit u0 = new Mine(new IntVector2D(12, 1), 2);
Unit u1 = new Marine(new IntVector2D(0, 5));
Unit u2 = new Turret(new IntVector2D(5, 6));
Unit u3 = new Tank(new IntVector2D(2, 4));
Unit u4 = new Marine(new IntVector2D(2, 4));
Unit u5 = new Wraith(new IntVector2D(2, 7));

ArrayList<Unit> units = new ArrayList<>();

units.add(u0);
units.add(u1);
units.add(u2);
units.add(u3);
units.add(u4);
units.add(u5);

for (Unit unit : units) {
    simulationManager.spawn(unit);
}

SimulationVisualizer visualizer = new SimulationVisualizer(units);
visualizer.visualize(0, simulationManager.getUnits());
```

명세에 맞춰 유닛들을 다 구현한 뒤, 위 코드를 실행하면 다음의 내용이 콘솔 창에 출력될 것입니다.

```
Frame: 0

0(N)01 1(M)35 2(U)99 3(T)85
4(M)35 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4|  3             |
5|1               |
6|     2          |
7|  5             |
 +----------------+

3-4
```

- Line 1: 프레임 번호
- Line 3 - 4: 시뮬레이션 중인 각 유닛의 상태. 상태는 다음의 포맷을 따릅니다.

  `<유닛 번호>(<표식>)<HP>`

  - `유닛 번호`: 유닛의 번호. 시뮬레이션에 추가된 순서에 따라 자동으로 번호가 매겨짐
  - `표식`: 유닛의 표식
  - `HP`: 유닛의 현재 HP

  예: `0(N)01` 은 유닛 번호가 0이고 HP가 1인 지뢰, `4(M)35` 은 유닛 번호가 4고 HP가 35인 해병을 의미. 죽은 유닛은 HP에 'XX`를 출력합니다. 따라서 `4(M)XX`는 4번 유닛인 해병이 죽었다는 의미입니다.
- Line 6 - 16: 시뮬레이션 중인 월드(world). 월드 안에 있는 16진수는 유닛 번호를 나타냅니다.
- Line 18: 한 타일 안에 여러 유닛이 있을 경우 등장합니다. 여기서 3-4는 3번 유닛이 있는 타일에 4번 유닛도 있음을 의미합니다.

다음은 10 프레임까지 시뮬레이션을 하는 코드입니다.

```java
package academy.pocu.comp2500.assignment3.app;

import academy.pocu.comp2500.assignment3.App;
import academy.pocu.comp2500.assignment3.SimulationManager;
import academy.pocu.comp2500.assignment3.IntVector2D;
import academy.pocu.comp2500.assignment3.Marine;
import academy.pocu.comp2500.assignment3.Mine;
import academy.pocu.comp2500.assignment3.Tank;
import academy.pocu.comp2500.assignment3.Turret;
import academy.pocu.comp2500.assignment3.Unit;
import academy.pocu.comp2500.assignment3.Wraith;
import academy.pocu.comp2500.assignment3.registry.Registry;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class Program {

    public static void main(String[] args) {
        SimulationManager simulationManager = SimulationManager.getInstance();

        Unit u0 = new Mine(new IntVector2D(12, 1), 2);
        Unit u1 = new Marine(new IntVector2D(0, 5));
        Unit u2 = new Turret(new IntVector2D(5, 6));
        Unit u3 = new Tank(new IntVector2D(2, 4));
        Unit u4 = new Marine(new IntVector2D(2, 4));
        Unit u5 = new Wraith(new IntVector2D(2, 7));

        ArrayList<Unit> units = new ArrayList<>();

        units.add(u0);
        units.add(u1);
        units.add(u2);
        units.add(u3);
        units.add(u4);
        units.add(u5);

        for (Unit unit : units) {
            simulationManager.spawn(unit);
        }

        SimulationVisualizer visualizer = new SimulationVisualizer(units);
        for (int i = 0; i < 10; ++i) {
            clearConsole();
            visualizer.visualize(i, simulationManager.getUnits());
            simulationManager.update();
            continueOnEnter();
        }
    }

    public static void continueOnEnter() {
        BufferedReader reader =
                new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.println("Press enter to continue");
            reader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void clearConsole() {
        try {
            new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

커맨드 라인에서 다음과 같이 `java` 커맨드를 사용하여 이 프로그램을 실행한 것을 권장합니다.

```
java academy.pocu.comp2500.assignment3.app.Program
```

그러면 각 프레임을 다음과 같은 애니메이션을 볼 수 있거든요.

```
Frame: 0

0(N)01 1(M)35 2(U)99 3(T)85
4(M)35 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4|  3             |
5|1               |
6|     2          |
7|  5             |
 +----------------+

3-4

Press enter to continue
```

프레임 0, 1, 2, 3의 결과를 아래에도 텍스트로 복사해 놓았으니 디버깅 중에 유용하게 사용하세요.

```
Frame: 0

0(N)01 1(M)35 2(U)99 3(T)85
4(M)35 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4|  3             |
5|1               |
6|     2          |
7|  5             |
 +----------------+

3-4


Frame: 1

0(N)01 1(M)35 2(U)99 3(T)73
4(M)35 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4|1 3             |
5|                |
6|  5  2          |
7|                |
 +----------------+

3-4


Frame: 2

0(N)01 1(M)31 2(U)99 3(T)61
4(M)35 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4| 13             |
5|  5             |
6|     2          |
7|                |
 +----------------+

3-4


Frame: 3

0(N)01 1(M)25 2(U)99 3(T)37
4(M)23 5(W)80

  0123456789ABCDEF
 +----------------+
0|                |
1|            0   |
2|                |
3|                |
4| 13             |
5|  5             |
6|     2          |
7|                |
 +----------------+

3-4
```

4. 클래스와 메서드를 등록한다

본인 컴퓨터에서 테스트를 마쳤다면 `academy.pocu.comp2500.assignment3.registry` 패키지 안에 있는 `Registry` 클래스에 여러분의 메서드를 등록하세요. 그래야 빌드봇이 어떤 public 메서드를 호출하여 테스트를 해야 하는지 알게 됩니다. `Registry` 클래스에는 총 7개의 `registerXXX()` 메서드가 있답니다.

1. `registerMarineCreator`: 해병 유닛을 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
2. `registerTankCreator`: 전차 유닛을 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
3. `registerWraithCreator`: 망령을 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
4. `registerTurretCreator`: 미사일 포탑을 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
5. `registerMineCreator`: 지뢰를 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
6. `registerSmartMineCreator`: 스마트 지뢰를 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
7. `registerDestroyerCreator`: 파괴자를 만드는 생성자나 메서드를 등록합니다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.

`academy.pocu.comp2500.assignment3.registry` 안에 들어있는 클래스들을 **변경하지 마세요**. 전혀 그럴 필요 없고 그냥 사용하시기만 하면 되요!

`Registry` 클래스를 사용하는 방법을 예를 들어 설명하겠습니다. 여러분이 지뢰를 나타내는 `Foo` 라는 클래스를 만들었다고 합시다. 그러면 `App` 클래스 생성자에서 다음과 같이 `registerMineCreator()` 를 호출해 주세요.

```
package academy.pocu.comp2500.assignment3;

import academy.pocu.comp2500.assignment3.registry.Registry;

public class App {
    public App(Registry registry) {
        ...

        registry.registerMineCreator("Foo");

        ...
    }
}
```

그게 아니라 `Foo` 클래스에 지뢰를 생성하는 `bar()` 메서드를 만들었다면 다음과 같이 해주세요.

```
package academy.pocu.comp2500.assignment3;

import academy.pocu.comp2500.assignment3.registry.Registry;

public class App {
    public App(Registry registry) {
        ...

        registry.registerMineCreator("Foo", "bar");

        ...
    }
}
```

이러면 빌드봇이 새로운 지뢰 유닛을 만드는 법을 알게 됩니다.

모든 메서드를 등록한 후에는 다음 코드를 `Program.java` 에 추가한 뒤, 빌드봇에 채점 요청을 하기 전에 실행해 보세요.

```java
package academy.pocu.comp2500.assignment3.app;

import academy.pocu.comp2500.assignment3.App;
import academy.pocu.comp2500.assignment3.registry.Registry;

public class Program {

    public static void main(String[] args) {
        Registry registry = new Registry();
        App app = new App(registry);
        registry.validate();
    }
}
```

`validate()` 메서드는 `Registry` 클래스에 메서드를 등록할 때 오타 등을 내지 않았는지 확인해줍니다. 빌드봇이 메서드들을 찾지 못해 어쩔 수 없이 0점을 주기 전에 미리 확인해보는 게 좋겠죠? :) 참고로 이 메서드는 `assert` 키워드를 사용하니 적절한 VM 옵션을 지정해주는 것도 잊지 마세요.

5. 커밋, 푸시 그리고 빌드 요청

이건 어떻게 하는지 이제 다 아시죠? :)

A. 부록

A.1 용어집

다음은 모든 유닛들에게 적용되는 용어와 규칙입니다.

- 시야(vision): 시야는 유닛이 현재 위치에서 볼 수 있는 거리를 나타내는 값입니다. 이 값이 높을수록 유닛은 더 넓은 영역을 볼 수 있습니다. 시야가 0인 유닛은 현재 자기 위치만 볼 수 있습니다.

  예>

  `*`: 유닛
  `o`: 유닛이 볼 수 있는 타일

  ```
  시야: 1

  o o o
  o * o
  o o o
  ```

  ```
  시야: 2

  o o o o o
  o o o o o
  o o * o o
  o o o o o
  o o o o o
  ```

  ```
  시야: 3

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o * o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  ```

- 공격 지점: 어떤 유닛이 공격을 하는 타일을 의미합니다.
- 공격 구역: 유닛이 공격할 수 있는 타일들을 의미합니다. 이 타일들은 유닛의 위치가 바뀜에 따라 바뀌며, 각 유닛의 공격 구역은 다양합니다.
- 범위 효과(AOE, Area Of Effect): AoE 값이 1 이상인 공격은 공격 지점 근처 타일에 있는 적들에게도 피해를 입힙니다. AoE 값이 클수록 피해를 받는 범위도 커집니다. AoE 값이 0인 경우 이 공격은 공격 지점 타일에 있는 적들에만 영향을 미칩니다.

  예>

  `*`: 공격 지점
  `o`: 공격의 영향을 받는 타일

```
AoE: 1

o o o
o * o
o o o


AoE: 2

o o o o o
o o o o o
o o * o o
o o o o o
o o o o o


AoE: 3

o o o o o o o
o o o o o o o
o o o o o o o
o o o * o o o
o o o o o o o
o o o o o o o
o o o o o o o
```

AoE로 인해 발생하는 피해치는 공격 지점에서 거리가 멀어짐에 따라 줄어듭니다. (x, y) 위치에서 받는 AoE 공격의 피해치를 계산하는 공식은 다음과 같습니다.

```
피해치(x, y) = (공격 지점에서의 피해치) * (1 - 공격 지점으로부터의 거리 / (공격의 AoE 값 + 1))
```

`거리`는 다음과 같이 정의합니다. 다음 그림에서 숫자는 공격 지점으로부터의 거리값을 나타냅니다.

```
3 3 3 3 3 3 3
3 2 2 2 2 2 3
3 2 1 1 1 2 3
3 2 1 * 1 2 3
3 2 1 1 1 2 3
3 2 2 2 2 2 3
3 3 3 3 3 3 3
```

계산 중에는 `double` 형을 사용하고 최종 계산 뒤에는 소수점 이하는 버리세요.

예>

```
공격 지점에서의 피해치 = 10
공격 지점 = (5, 6)
AoE = 3

피해치(4, 6) = 10 * (1 - 1 / (3 + 1))
             = 7

피해치(7, 5) = 10 * (1 - 2 / (3 + 1))
             = 5
```

- 생명(HP): HP는 유닛이 버틸 수 있는 최대 피해치를 나타냅니다. HP가 0이 된 유닛은 죽습니다.
- 공격력(AP): AP는 다른 유닛을 공격할 때 입히는 피해치를 나타냅니다.

A.2 시뮬레이션 규칙

- 시뮬레이션에 사용하는 월드는 16 x 8 크기의 2D 그리드(grid)입니다.
- 원점은 (0,0)이며 +x는 오른쪽, +y는 아래쪽입니다. 또한 북쪽은 -y, 동쪽은 +x 입니다
- 시뮬레이션 시작 전에 모든 유닛을 월드에 추가합니다.
- 본 시뮬레이션에서는 어느 유닛도 팀을 이루지 않는 없는 배틀 로열 방식입니다.
- 한 프레임에서 유닛이 할 수 있는 행동은 이동, 공격, '아무것도 안 함' 중 하나입니다. 유닛은 한 프레임에서 복수의 행동을 **할 수 없습니다.** (즉, 한 프레임이 이동과 공격을 다 할 수 없음)
- 유닛은 현재 위치에 바로 인접한 타일로만 이동할 수 있습니다. 즉, 동서남북 중 한 방향으로만 이동할 수 있으며 대각선 이동은 허용하지 않습니다.
- 모든 이동 가능한(movable) 유닛은 동등한 이동 기회를 받습니다. 즉, 해당 프레임에서 각 유닛의 이동 순서가 달라져도 시뮬레이션 결과는 바뀌지 않습니다.
- 유닛은 공격 구역에 속한 타일만 공격할 수 있습니다.

- 유닛은 공격 구역에 있는 적을 발견하면 반드시 그 타일을 공격해야 합니다. 이동 가능한 유닛은 공격 구역에 있는 적을 발견하지 못한 경우에만 이동할 수 있습니다.
- 유닛이 어떤 타일을 공격하면, 그 타일 안에 있는 다른 모든 유닛이 공격을 받습니다. 단, 명세에 다른 규칙이 있는 경우는 예외입니다.
- 모든 유닛은 동등한 공격 기회를 받습니다. 즉, 한 프레임에서 각 유닛의 공격 순서가 달라져도 시뮬레이션 결과는 바뀌지 않습니다.
- 유닛은 자기 자신에게 피해를 입힐 수 없습니다.
- 유닛은 각 유닛 명세에 나와 있는 (A.3 참고) 교전/이동 규칙을 따라야 합니다. 교전/이동 규칙의 각 단계를 실행한 후에도 적이 한 명으로 특정되지 않고 여럿이 남아있다면 남은 적들에 대해 다음 단계들을 실행합니다. 그게 아니라 어떤 단계를 실행 후, 적이 하나로 특정되었다면 그 적을 공격/따라가며, 그 후 단계들은 실행하지 않습니다.

A.3 유닛 명세

이 시뮬레이션에는 6종류의 유닛이 있습니다.

A.3.1 해병(Marine, 마린)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
M	지상	2	0	6	35	지상, 공중

- 공격 구역:

  `*`: 해병 `o`: 해병이 공격할 수 있는 타일. 자기 위치를 공격할 수도 있음 `.`: 해병이 공격할 수 없는 타일

```
. . o . .
. o * o .
. . o . .
. . . . .
```

- 해병은 자기 시야 안에 있는 지상 유닛과 공중 유닛을 다 볼 수 있습니다.
- 다음은 해병의 교전규칙입니다. (우선순위 순)

  1. 가장 약한 유닛(HP가 가장 낮은 유닛)이 있는 타일을 공격
  2. 자신의 위치에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 북쪽(위쪽)에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 시계 방향으로 검색하다 찾은 유닛의 타일을 공격
- 다음은 해병이 시야 안에서 적을 발견했을 때 따르는 이동 규칙입니다. (역시 우선순위 순)

  1. 가장 가까이 있는 유닛 쪽으로 이동. 가장 가까운 유닛은 맨해튼 거리를 사용하여 판단합니다.
  2. 가장 약한 유닛 쪽으로 이동
  3. 북쪽에 있는 유닛 쪽으로 이동, 북쪽에 유닛이 없다면 시계 방향으로 검색하다 찾은 유닛 쪽으로 이동

  이동할 때는 언제나 y축을 따라 다 이동한 뒤 x축을 따라 이동합니다.
- 해병이 시야 안에서 적을 찾지 못한 경우, 현재 타일에서 움직이지 않습니다.

A.3.2 전차(Tank, 탱크)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
T	지상	3	1	8	85	지상

- 공격 구역:

  `*`: 전차
  `o`: 전차가 공격할 수 있는 타일 `.`: 전차가 공격할 수 없는 타일

```
. . . . . . .
. . o o o . .
. o . . . o .
. o . * . o .
. o . . . o .
. . o o o . .
. . . . . . .
```

- 전차는 자기 시야 안에 있는 지상 유닛만 볼 수 있습니다.
- 전차에는 두 가지 모드(mode)가 있습니다.

  - 전차(tank) 모드 (기본 모드): 전차 모드인 경우, 이동할 수 있으나 공격할 수는 없습니다.
  - 공성(siege) 모드: 공성 모드인 경우, 이동할 수 없으나 공격할 수는 있습니다.

  전차는 모드를 바꿀 때 1 프레임을 소모합니다.
- 공성 모드인 전차는 탱크 모드일 때보다 2배의 피해를 받습니다.

- 다음은 전차의 교전규칙입니다. (우선순위 순)

  1. 현재 공성 모드가 아닌 경우 공성 모드로 변경
  2. 가장 약한 유닛이 있는 타일을 공격
  3. 북쪽에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 시계 방향으로 검색하다가 찾은 유닛의 타일을 공격
- 전차가 시야 안에서 적을 찾으면 공성 모드로 변환하여 공격할 준비를 합니다.

- 전차가 시야 안에서 적을 찾지 못하면 다음의 이동 규칙을 따릅니다. (역시 우선순위 순)

  1. 이동하던 방향 끝까지 이동. 한번도 이동한 적이 없다면 오른쪽으로 이동
  2. 반대 방향 끝까지 이동
  3. 시야 안에서 적을 발견할 때까지 1 - 2를 반복

A.3.3 망령(Wraith, 레이스)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
W	공중	4	0	6	80	지상, 공중

- 공격 구역

  `*`: 망령
  `o`: 망령이 공격할 수 있는 타일. 자기 위치를 공격할 수도 있음 `.`: 망령이 공격할 수 없는 타일

```
. . . . .
. . o . .
. o * o .
. . o . .
. . . . .
```

- 망령은 자기 시야 안에 있는 지상 유닛과 공중 유닛을 다 볼 수 있습니다.

- 망령은 공격을 받으면 즉시 가동되는 특수 방어막을 가지고 있습니다. 한 번 가동된 방어막은 현재 프레임이 끝날 때까지 지속되어 망령은 피해를 입지 않습니다. 다음 프레임부터는 공격을 받으면 피해를 입습니다.

- 다음은 망령의 교전규칙입니다. (우선순위 순)

  1. 공중 유닛들을 공격할 후보로 선택. 선택할 공중 유닛이 없다면 지상 유닛들을 선택
  2. 가장 약한 유닛이 있는 타일을 공격
  3. 자신의 위치에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 북쪽(위쪽)에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 시계 방향으로 검색하다 찾은 유닛의 타일을 공격
- 다음은 망령이 시야 안에서 적을 발견할 경우 따르는 이동 규칙입니다. (역시 우선순위 순)

  1. 공중 유닛들을 따라갈 후보로 선택. 선택할 공중 유닛이 없다면 지상 유닛들을 선택
  2. 가장 가까이 있는 유닛 쪽으로 이동
  3. 가장 약한 유닛 쪽으로 이동
  4. 북쪽에 있는 유닛 쪽으로 이동. 북쪽에 유닛이 없다면 시계 방향으로 검색하다 찾은 유닛 쪽으로 이동

  이동할 때는 언제나 y축을 따라 이동하는 게 우선입니다.
- 망령이 시야 안에서 적을 찾지 못한 경우, 자기의 처음 위치 쪽으로 이동해야 합니다. 이때 역시 y축을 따라 먼저 이동합니다.

A.3.4 미사일 포탑(Turret, 터렛)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
U	지상	2	0	7	99	공중

- 공격 구역

  `*`: 미사일 포탑
  `o`: 미사일 포탑이 공격할 수 있는 타일. 자기 위치를 공격할 수도 있음 `.`: 미사일 포탑이 공격할 수 없는 타일

```
. . . . .
. o o o .
. o * o .
. o o o .
. . . . .
```

- 미사일 포탑은 이동할 수 없는 유닛입니다.

- 미사일 포탑은 자기 시야 안에 있는 공중 유닛만 볼 수 있습니다.

- 다음은 미사일 포탑의 교전규칙입니다. (우선순위 순)

1. 가장 약한 유닛이 있는 타일을 공격
2. 자신의 위치에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 북쪽(위쪽)에 유닛이 있다면 그 타일을 공격. 그렇지 않을 경우 시계 방향으로 검색하다 찾은 유닛의 타일을 공격

A.3.5 지뢰(Mine, 마인)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
N	지상	0	0	10	1	지상

- 공격 구역

  *: 지뢰. 자기 타일만 공격할 수 있음 .: 지뢰가 공격할 수 없는 타일

```
. . . . .
. . . . .
. . * . .
. . . . .
. . . . .
```

- 지뢰는 움직일 수 없는 유닛이며, 다른 유닛은 지뢰를 볼 수 없습니다.
- 지뢰는 다른 유닛이 일정 횟수만큼 밟으면 그때 터집니다. 지뢰 위치에 있는 다른 유닛들은 모두 피해를 입습니다.
- 이 횟수는 지뢰마다 다르게 지정할 수 있습니다.
- 터진 지뢰는 파괴됩니다.

A.3.6 스마트 지뢰(SmartMine, 스마트 마인)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
A	지상	1	1	15	1	지상

- 공격 구역

  *: 스마트 지뢰. 자기 타일만 공격할 수 있음 .: 스마트 지뢰가 공격할 수 없는 타일

```
. . . . .
. . . . .
. . * . .
. . . . .
. . . . .
```

- 스마트 지뢰는 지뢰의 상위 호환 버전입니다. 스마트 지뢰는 지뢰와 동일하게 작동하는 것 외에도 초특급 울트라 레이더를 탑재한 덕분에 근처에 있는 유닛들을 감지하는 능력을 갖추고 있습니다. 만약 시야 안에서 몇 명 이상의 적 유닛이 감지되면, 스마트 지뢰가 폭발합니다. 폭발에 필요한 최소 적 유닛 수는 스마트 지뢰마다 다르게 지정할 수 있습니다.
- 초특급 울트라 레이더 기술 덕분에 스마트 지뢰는 시야 안에 있는 모든 지상 유닛을 감지할 수 있습니다. 하지만 공중 유닛과 볼 수 없는 유닛은 감지할 수 없답니다.

A.3.7 파괴자(Destroyer, 디스트로이어)

표식	유닛 종류	시야	AoE	AP	HP	공격 대상
D	?	?	?	?	?	?

- 이 유닛은 알려진 것이 별로 없습니다. 이 유닛을 개발한 군사 과학자에 따르면 이 유닛은 망령을 제외한 모든 유닛을 한 프레임 만에 모두 파괴할 수 있다고 합니다. (망령을 죽이지 못하는 이유는 방어막 때문)
- 파괴자는 공격자의 공격력과는 상관없이 1의 피해만 받습니다.