Homework #2

RELEASE DATE: 04/30/2021 - 3 DAYS

GREEN CORRECTION: 05/11/2021 10:00

BLUE CORRECTION: 05/04/2021 22:40

RED CORRECTION: 04/29/2021 04:30

DUE DATE: 05/28/2021, BEFORE 10:00 ON GITHUB CLASSROOM

QUESTIONS ARE WELCOMED ON THE NTU COOL FORUM.

You need to upload your submission files to the github repository under the exact guidelines that will be provided on NTU COOL.

Any form of cheating, lying, or plagiarism will not be tolerated. Students can get zero scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.

Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from.

Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

You should write your solutions in English. We do not accept solutions written in any other languages.

This homework set comes with 200 points and 20 bonus points. In general, every homework set would come with a full credit of 200 points, with some possible bonus points.

Overview

A Role Playing Game (RPG) is a game in which players assume the roles of characters. Many RPGs are based on a turn-based battle system. Under the turn-based battle system, two troops (i.e., teams) are fighting against each other in a mode that every unit in the troop takes turns to perform an action. The actions could be committing a basic attack to one enemy, or using a special skill with various effects at the expense of the Magic Point (MP) of the unit.

Developing such game is really a huge challenge for programmers as there are many variations from the client's requirement! For instance, assume that you are a special unit of the world called the hero, what kind of skills would you like to be equipped with? A skill that helps protect your allies, a skill that can heal yourself, or a skill that can harm more enemies? Those requirement variations are called **Behavioral Variations**, because **all skills have different effects** that act on different units within the RPG!

In this homework, you are asked to implement a "fictional" RPG that comes with **10 skills** in total. We ask you to carefully think about how to design your classes to encapsulate and abstract the behavioral variations in different skills. Also, you will enjoy "contract programming" with OOP by letting your code integrate with some written libraries that represent the artificial intelligence of the units. The "elegance" of your design will be part of the grading criteria—in particular, we challenge you to try to beat The TAs' design in mind. Enjoy!

Glossary

1. RPG: The role-play game based on a turn-based battle system.
2. Unit: An entity in the RPG that comes with the following properties—HP (Health Point), MP (Magic Point), STR (Strength), state, name, and a list of skills.

3. Troop: A list of Units that are teamed up under a battle.
4. Hero: One special unit within one of the Troops that represents the player, and there will only be one Hero in the RPG.
5. Battle: The activity where two Troops fight against each other.
6. Skill: A special action that causes some effects to some of the Units.
7. Ally: Another Unit who is in the same Troop to a Unit.
8. Enemy: Another Unit who is in a different Troop to a Unit.
9. Damage a Unit: The step of decreasing the HP of the unit by some value.
10. Dead Unit: A Unit with HP ≤ 0.

Game Flow

1. In our RPG, there are two troops T_1, T_2 fighting against each other. For $i \in \{1, 2\}$, denote $u_{i,1}, u_{i,2} ..., u_{i,n_i}$ to be the units in T_i, where n_i is the number of units in T_i. The first unit in T_1 will be called the hero h. That is, $h = u_{1,1}$.
2. In the beginning of the battle, all units are in their default HP, MP and STR values, and are set to the `Normal` state.
3. In every round, the RPG wants every non-dead unit (i.e. with HP > 0) to perform an action, using the order $(u_{1,1}, u_{1,2}, ...u_{1,n_1}, u_{2,1}, u_{2,2}, ..., u_{2,n_2})$ and skipping any dead units. The round ends and the next round starts when every non-dead unit has taken its turn to performed its action. During the turn of the action unit $u_{i,j}$, the unit is asked to
   - (S_1) Select an action: The action is either a `BasicAttack`, or a skill from its list of skills. Some actions may cost MP from $u_{i,j}$. The RPG will check if $u_{i,j}$ has enough MP to conduct the action. If not, the action is considered illegal, and the RPG will keep asking $u_{i,j}$ until it selects a legal action.
   - (S_2) Decide the target: If the selected action in (S_1) requires m specific targets and the number of legitimate candidates n is greater than m, then the RPG will ask $u_{i,j}$ to decide m targets from the list of n legitimate candidates that the action can be performed on. Otherwise (i.e. if the action does not require specific targets, or if $m \geq n$), the RPG will automatically decide the targets and skip (S_2), but the following step (S_3) will still be executed.

   (S_3) After (S_1) and (S_2), the RPG will perform the selected action from $u_{i,j}$ by deducting the needed MP from $u_{i,j}$, and making the necessary property updates to the target unit(s) of the action.
4. After the selected action is performed, the RPG checks if the battle is over. The battle is over if either (1) the hero h is dead, or (2) either of the two troops is annihilated (i.e., all of the units in the troop are dead). Otherwise the battle continues.
5. When the battle is over, the player wins if the hero h survives. Otherwise the player loses.

States

Except for the `Normal` state, the other states change the unit's behavior in its turn to perform an action.

1. `Petrochemical`: The unit cannot perform an action in this turn (i.e., the steps (S_1), (S_2) and (S_3) will be skipped).
2. `Poisoned`: The HP of the unit is decreased by 30 in its turn, before (S_1). If the decrease causes the unit to be dead, the unit cannot perform an action.
3. `Cheerup`: For every damaging action, which decreases the HP of some target unit(s), each target unit will be damaged by 50 more.

Actions

An action that marks * in its target requires specific target units. That is, (S_2) will be performed if the action is selected in (S_1) and there are enough legitimate candidates.

1. `BasicAttack`: (MP: 0 — Target: 1 specific enemy*) - damage the target unit by the STR of the action unit.
2. Skills:
   - [a] `Waterball`: (MP: 50 — Target: 1 specific enemy*) - damage the target unit by 120.
   - [b] `Fireball`: (MP: 50 — Target: All enemies) - damage every target unit by 50.
   - [c] `SelfHealing`: (MP: 50 — Target: the action unit itself) - increase the HP of the target unit (which is the action unit) by 150.
   - [d] `Petrochemical`: (MP: 100 — Target: 1 specific enemy*) - turn the target unit into the `Petrochemical` state for 3 rounds, including the current round.
   - [e] `Poison`: (MP: 80 — Target: 1 specific enemy*) - turn the target unit into the `Poisoned` state for 3 rounds, including the current round.
   - [f] `Summon`: (MP: 150 — Target: None) - Summon a Slime. The Slime is a special unit with (HP: 100, MP: 0, STR: 50, skills: none, state: `Normal`, name: `Slime`). Create the Slime and put it at the end of the same troop as an ally, which allows the Slime to perform an action right within the current round.
   - [g] `SelfExplosion`: (MP: 200 — Target: All units) - Commit suicide (decrease the action unit's HP to 0), and damage every unit in both troops of the battle by 150.
   - [h] `Cheerup`: (MP: 100 — Target: 3 specific allies*) - turn every target unit into the `Cheerup` state for 3 rounds, including the current round.
   - [i] `Curse`: (MP: 100 — Target: 1 specific enemy*) - When the cursed target unit dies in the future, the action unit's HP will increase by the remaining MP of the target unit. The same target unit can be cursed by multiple action units during different turns. When the same action unit curses the same target unit, the action unit gets only one copy of the remain MP; when different action units curse the same target unit, each action unit gets one copy of the remaining MP. For instance, if a unit has been cursed by units $u_{i,j_1}, u_{i,j_2}, u_{i,j_1}$, when it dies with 60 MP, u_{i,j_1} will get 60 HP, and u_{i,j_2} will also get 60 HP. The increase in HP is effective immediately after the target unit dies (i.e. with HP ≤ 0) in step (S_3).
   - [j] `OnePunch`: (MP: 180 — Target: 1 specific enemy*) - damage the target unit by a random amount. You do not need to implement this skill by yourself. The `OnePunch` class is already compiled and provided. You only need to integrate it in your code.

   For skills that are state-changing, after the three rounds, the state of the unit should go back to `Normal`. Also, the state-changing skills override the original state. That is, they nullify any remaining rounds that the original state should last, and re-start counting the (three) rounds.

Input Format

```
#START-TROOP-1
<troop-1>
#END-TROOP-1
#START-TROOP-2
<troop-2>
#END-TROOP-2
<Hero's decision 1>
<Hero's decision 2>
<Hero's decision 3>
...
```

- `<troop-1>` and `<troop-2>` are composed of several `<Unit>`, each of which a line like

```
<Unit> = <name> <HP> <MP> <STR> <skill-1> <skill-2> <skill-3> ... <skill-K>
```

- `<HP>`, `<MP>` and `<STR>` are the unit's HP, MP and STR, respectively, guaranteed to be non-negative integers.
- `<name>` is the unit's name which contains alphanumeric characters (A-Z, a-z, or 0-9, case-sensitive). The first `<Unit>` in `<troop-1>` is guaranteed to start with the name: `Hero`, as $u_{1,1}$ is always the hero.
- For $k = 1, \ldots, K$, `<skill-k>` is the k^{th} skill in the unit's list of skills, `<skill-k>` is limited to `Waterball`, `Fireball`, `SelfHealing`, `Petrochemical`, `Poison`, `Summon`, `SelfExplosion`, `Cheerup`, `Curse`, `OnePunch`.
- The decisions of all non-hero units are made by the TAs' AI and you do not need to worry about them. But your program should parse the player's (a.k.a. hero's) decisions. Each `<Hero's decision>` is a separate line that consists of a decision made from the hero, which is either for selecting an action during (S_1) with a number, or specifying specific target units during (S_2) with several numbers separated by ", " (a comma followed by a space).

  For simplicity, we assume that each `<Hero's decision>` only contains valid choices within the range of candidate selections, and the decision for (S_2). contains the correct number of targets that need to be selected. Nevertheless, as illustrated in the Game Flow, `<Hero's decision>` in (S_1). may contain illegal selections (because of insufficient MP) that need to be skipped. When the decision is illegal for (S_1), the next line of `<Hero's decision>` would remain to be for (S_1) until a legal selection is made.

Output Format

- In order to make the units distinguishable between the two troops. Every unit's name is prefixed with `[1]` if it is in T_1, or `[2]` if it is in T_2. For example: `[1]Hero`, `[2]Boss`.
- At every non-dead unit's turn, first output a line of

  ```
  <unit>'s turn (HP: <HP>, MP: <MP>, STR: <STR>, State: <state>).
  ```

  where `<unit>`, `<HP>`, `<MP>`, `<STR>` and `<state>` are the unit's name, HP, MP, STR and state, respectively. The `<state>` is limited to `Normal`, `Petrochemical`, `Poisoned`, `Cheerup`. Note that a `Petrochemical` unit, as long as it is not dead, still needs this line to be outputted.
- During the step (S_1), to ask the action unit to select an action, output a line of:

  ```
  Select an action: (0) Basic Attack (1) <skill-1> (2) <skill-2> ... (<K>) <skill-K>
  ```

  where `<K>` is the number of skills of the unit, and for $k = 1, \ldots, K$, `<skill-K>` is the k^{th} skill's name. The order of the skills in the output line should be the same as the unit's list of skills in the input.
- During the step (S_2), if it is the hero's turn, ask the hero to select the target units from a list of candidates, output a line of

  ```
  Select <m> targets: (0) <candidate-1> (1) <candidate-2> ... (<n-1>) <candidate-n>
  ```

  where `<m>` is the number of target units required by the action, and for $j = 1, \ldots, n$, `<candidate-j>` is the j^{th} candidate's name. The order of the candidate units in the output line should follows the order $(u_{1,1}, u_{1,2}, ...u_{1,n_1}, u_{2,1}, u_{2,2}, ..., u_{2,n_2})$.
- During the step (S_3), the RPG let the action unit perform an action. If the action is a `BasicAttack`, output a line of:

  ```
  <unit> attacks <target>.
  ```

If the action is `Summon` or `SelfHealing`, output a line of:

`<unit> uses <skill>.`

Otherwise, output a line of:

`<unit> uses <skill> on <target list>.`

where `<unit>` is the unit's name, `<skill>` is the used skill's name and `<target list>` is a list of target's names separated by ", ", which a comma followed by a space, e.g.,

`[1]Hero, [1]Catwoman, [1]Robin`

- If the action damages a unit, then for each target unit that is damaged (following the order $(u_{1,1}, u_{1,2}, ...u_{1,n_1}, u_{2,1}, u_{2,2}, ..., u_{2,n_2})$), output the following message in a line:

  `<unit> causes <action's damage> damage to <target>.`

  The `<action's damage>` is an integer that indicates the decrease of the target unit's HP. Then, if an unit dies this damage, output the following message in another line:

  `<target> dies.`

- In (S_1), if an action cannot be performed because there is not enough MP, output a line of:

  `You can't perform the action: insufficient MP.`

- When the game is over, if the player wins, output a line of:

  `You win.`

  Otherwise output:

  `You lose.`

Java Programming

The Java Programming environment is exactly like Homework 1. We hope to clarify here that your program will be graded under Linux (though Java is supposed to be cross-platform and there should not be many issues if you developed the program under Windows or Mac). Also, the compile command is changed to

```
javac -sourcepath src/ -classpath provided.jar -d out/ src/*.java
```

where `provided.jar` contains the classes provided by the TAs.

Working with Other Classes

Our test cases' input only contains the hero's decisions. The decisions of other non-hero units' are made by AI. The AI's implementation is already compiled and provided in `provided.jar`. This implementation uses pseudo-random numbers with fixed seeds to make every decision ensuring that the result is always deterministic. The assumption of these AI units is that the `selectAction` and `selectTarget` methods are called only *once* in the unit's turn. In addition, the `OnePunch` skill class is also compiled and provided in `provided.jar`. You can find the guide in the `src/tutorials` package.

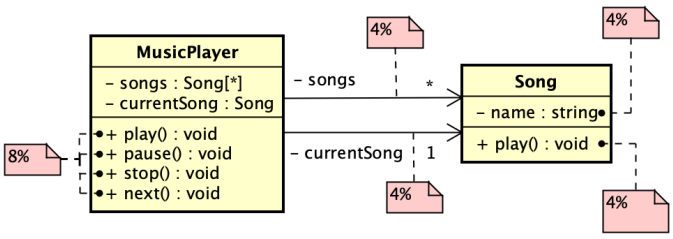
Grading Criteria

- program correctness (each 10%):
  - public test case 1: only-basic-attack (only `BasicAttack`)
  - public test case 2: waterball-to-fireball-1v2 (`Waterball` ∼ `Fireball`)
  - public test case 3: waterball-to-self-healing-1v2 (`Waterball` ∼ `SelfHealing`)
  - public test case 4: waterball-to-petrochemical-2v2 (`Waterball` ∼ `Petrochemical`)
  - public test case 5: waterball-to-summon-2v2 (`Waterball` ∼ `Summon`)
  - public test case 6: waterball-to-self-explosion-2v2 (`Waterball` ∼ `SelfExplosion`)
  - public test case 7: waterball-to-cheerup-2v2 (`Waterball` ∼ `Cheerup`)
  - public test case 8: waterball-to-Curse-2v2 (`Waterball` ∼ `Curse`)
  - public test case 9: waterball-to-one-punch-2v2 (`Waterball` ∼ `OnePunch`)
  - public test case 10: waterball-to-one-punch-3v3 (`Waterball` ∼ `OnePunch`)
  - public test case 11: waterball-to-one-punch-3v10 (`Waterball` ∼ `OnePunch`)
  - public test case 12: waterball-to-one-punch-10v10 (`Waterball` ∼ `OnePunch`)
  - public test case 13: waterball-to-one-punch-10v10 (`Waterball` ∼ `OnePunch`)
  - ~~private test case 14: anything-10v10 (`Waterball` ∼ `OnePunch`)~~
  - ~~private test case 15: anything-30v30 (`Waterball` ∼ `OnePunch`)~~
- Software design
  - Object-Oriented design (30%)
  - Follow the Open-Close Principle on removing/adding the implementations of AI units (10%)
  - Follow the Open-Close Principle on removing/adding the skill `SelfHealing` (10%)
  - Follow the Open-Close Principle on removing/adding the skill `Summon` (10%)
  - Follow the Open-Close Principle on removing/adding the skill `Petrochemical` (10%)
- Bonus software design
  - Follow the Open-Close Principle on removing/adding the skill `Curse` (Bonus 5%)
  - Follow the Open-Close Principle on removing/adding the skill `Cheerup` (Bonus 10%)
  - Follow the Open-Close Principle on removing/adding the skill `OnePunch` (Bonus 5%)

In general, we expect your code to be "plugin-like" so that removing or adding any skills will not require erasing or modifying any of your existing codes except the *initialization procedure* (instantiation/dependency-injection).

Design Report

The purpose of the design report is to help the TAs grade the software design (and bonus) part by human. The TAs will try to grade by checking whether your design satisfies some "ideal" design choices using an internal graph of class relations like this.



In this design report, **you DO NOT need to illustrate your design of every class**. You only need to write down how do you achieve the Open-Close principle **on those requirements which ask you to follow the Open-Close principle under certain cases**.

We are not strictly asking you to produce this kind of graph (as we have not taught you about how to do so). But please feel free to illustrate your classes with a similar graph if you want to. Other than those required above, please feel free to add anything that helps the TAs understand your design. But **please keep your design report as concise as possible**.

While the TAs grade with the "ideal" design choices, there is always a possibility that your design choices are better than the "ideal" ones. In this case, you can certainly check/argue with the TAs to get the points that you deserve afterwards. So please do not be too worried about whether your current design choices are "ideal" or not.

Submission Files

- your source code in the `src/` directory of your repository.
- a design report `Design-Report.md` in the root directory of your repository.

Tips from TAs

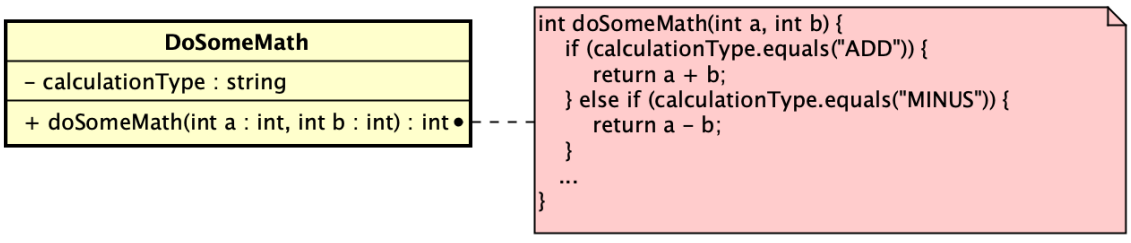
1. Try to declare the responsibility (in a form of the method's signature) of every class in your design very carefully. Do not rush into a design. Otherwise you may suffer from lots of refactoring later.
2. Refactoring steps: To accommodate the behavioral Variations following OCP, you may refactor your design incrementally with the following three steps:

   [a] **Encapsulate what varies**: Create a class for each behavior variant with a method encapsulating its behavior.

   [b] **Abstract common behaviors**: Abstract those variants by extracting an abstract class or an interface with well-defined abstract methods that represent the common behaviors.

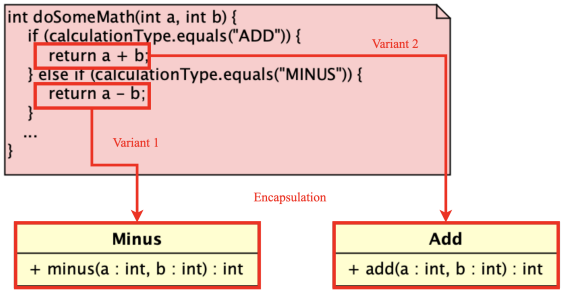
   [c] **Delegation/Composition**: Delegate the behavior to the abstract class or the interface that is extracted in the previous step.

   For example, you have a class `DoSomeMath` which has a variation in its math calculation method:

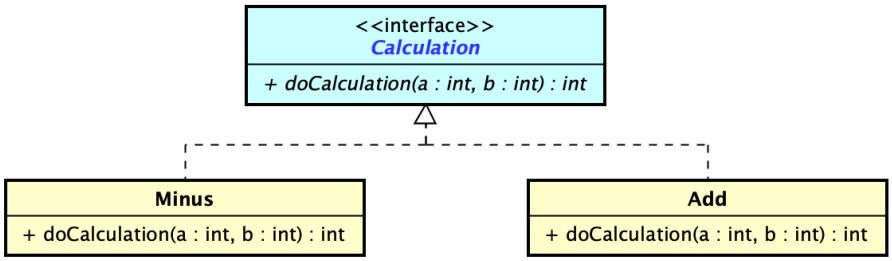


   Then, you can conduct the three refactoring steps to model such variation.

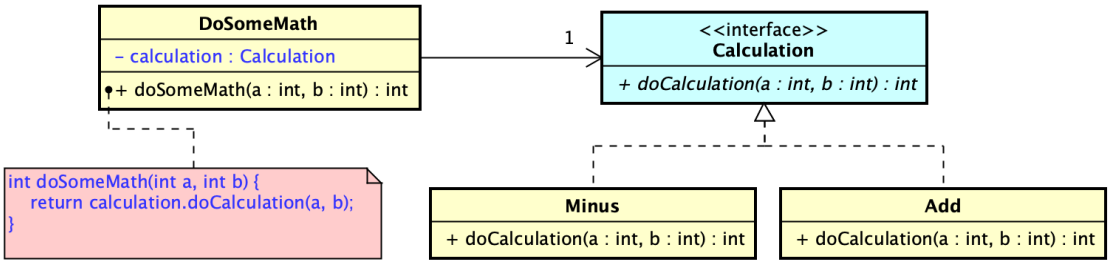
   [a] Encapsulate what varies



[b] Abstract common behaviors



[c] Delegation / Composition



3. If you have an excellent OOD model, all of your *skill* classes can only contain 2–4 lines of code in its behavioral method.

Other References

1. Homework Submission Guide: `https://hackmd.io/zAsrNRprQWqfI77msYQ5Bg`
2. Course Policy: `https://www.csie.ntu.edu.tw/~htlin/course/foop21spring/doc/policy.pdf` The TAs will compute the optimal usage of your gold medals after all the homework scores are announced. That is, you do not need to do anything on your side (except for remembering how many you have on hand).
3. Clarification of the Open-Close principle: `https://hackmd.io/rFlV4kBGRROFkyI1yuo6JA`