German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2015*
Yu Gi Oh: **Milestone 2**

Deadline: Fri 27.3.2015 @23:59

This milestone is an *exercise* on the concepts of **abstraction** and **polymorphism**. The *objective* of this milestone is to build the actual game play. Refer to the (**Game Description**) document for more details about the rules.

By the **end of this milestone**, you should have:

- Completed your implementation of the game engine
- Handle all the special cases and scenarios in the game

Requirements

Please note that you are not allowed to change any of the signatures provided in milestone 1. However, you are free to add more helper methods and instance variables.

General modifications

1. **Modify** the `action` method in (`Card`) class to be abstract. Apply all the needed modifications to accommodate the changes.
2. **Update** the (`SpellCard`) class to prevent instantiating it without changing its access modifier.
3. **Implement** the `action` method in the `MonsterCard` class. The `action` of a `MonsterCard` is his attack. The action method is used to attack the opponent, calculate the score and check if there is a winner. All the constraints on the attacking actions should be handled in `declareAttack` method (It will be discussed later in the milestone).

   Example 1 *Constraint (1): Each monster can attack only once per turn.*
4. **Overload** the action method with a version that takes no parameters. This version is similar to its sister method. However, it is used whenever the opponent has *no monsters* in the field.
5. **Extend** the adding of the monsters to check that you have an exact number of the required sacrifices. If you have more or less sacrifices, you should not add the monster.

   Hint: Modify `addMonsterToField` method in the `Field` class.

6. **Modify** the system to keep track of the players' scores. In the context of the game, the score is the *life points of each player*.

7. **Detect** when the game is over and determine the winner according to the described game rules. Once the `winner` is set (found in `Board` class), the game ends and you should not allow any other actions by any of the players. You are free to decide where and how you implement this feature in the code.

8. **Extend** your code to enforce all the phases' rules discussed in the (**Game Description**) document.Make sure to modify your code such that no action is done outside its appropriate phase. To achieve this goal, you will need to implement the interface discussed in the following section.

Create the `Duelist` interface

This component serves as a playing interface for the game. It has all the main functions needed to automate the game play. It also wraps all the low level functions implemented previously into logical high level functions related to the game play.

Name : `Duelist`

Package : `eg.edu.guc.yugioh.board.player`

The interface should have the following methods:

1. `public boolean summonMonster(MonsterCard monster)`: Adds a monster that does not need any sacrifices to the field in **ATTACK** mode if it does not violate any of the game regulations. It returns `true` if the monster was added to the field successfully and `false` otherwise. Please check the card visibility rules from the game description.

2. `public boolean summonMonster(MonsterCard monster, ArrayList<MonsterCard> sacrifices)`: An *overloaded version* of its sister method. However, it handles the monsters that need sacrifices. Please check the card visibility rules from the game description.

3. `public boolean setMonster(MonsterCard monster)`: Adds a monster that does not need any sacrifices to the field in **DEFENSE** mode if it does not violate any of the game regulations. It returns `true` if the monster was added to the field successfully and `false` otherwise. Please check the card visibility rules from the game description.

4. `public boolean setMonster(MonsterCard monster, ArrayList<MonsterCard> sacrifices)`: An *overloaded version* of its sister method. However, it handles the monsters that need sacrifices. Please check the card visibility rules from the game description.

5. `public boolean setSpell(SpellCard spell)`: Adds a spell card to the field if it does not violate any of the game rules without activating its action. Please check the card visibility rules from the game description. It returns `true` if the spell was added to the field successfully and `false` otherwise.

6. `public boolean activateSpell(SpellCard spell, MonsterCard monster)`: Depending on the spell location, the method will *either* add it to the field if it does not violate any of the game rules then activates it *or* directly activate an already set card in the field. After the card is activated, it should be sent to the graveyard. If the spell card does not need any monsters to apply its action, pass `monster` as `null`. It returns `true` if the activation was successful and `false` otherwise.

7. `public boolean declareAttack(MonsterCard activeMonster, MonsterCard opponentMonster):` Checks all the constraints from the game rules that should be handled before you can attack with a monster. Afterwards, it performs the attack and decides if there is a winner. It returns `true` if the attack was successful and `false` otherwise.

8. `public boolean declareAttack(MonsterCard activeMonster)`: An *overloaded version* of its sister method. It is used whenever the opponent player has no monsters in the field.

9. `public void addCardToHand()`: Allow the *active player* to draw one card and add it to his/her hand.

10. `public void addNCardsToHand(int n)`: Allow the *active player* to draw `n` cards and add them to his/her hand.

11. `public void endPhase():` Handles the shift to the next phase. After *Main (2) phase*, it should shift the turn.

12. `public boolean endTurn():` Switches the player and passes the turn to the opponent. *Remember that* the player can request ending the turn at the beginning of **any** phase. You must also check that the player who calls this method is the *active player*.

13. `public boolean switchMonsterMode(MonsterCard monster):` Switches the mode of the monster (**ATTACK** or **DEFENSE**), if it does not violate any of the game rules. It returns `true` if the mode switching was successful and `false` otherwise.

Implement the `Duelist` interface

1. You should **implement** the interface in the `Player` class.

2. The interface methods should ensure that all the **game constraints** are applied. You are highly encouraged to refer to the game description as a reminder for the terminology and the rules.

   Hint: Use the methods you implemented in *milestone (1)* as your *building blocks* to fill the interface methods.

3. All the implemented interface methods must perform their actions *if and only if* the player calling them is the **active player**.