CS5351 Software Engineering
2024/2025 Semester B
Tutorial on Code with Quality

Instructions

- **Q1** is an **individual** exercise.
- **Q2** is a **team-based** exercise like what we did in our exercise on Requirement Elicitation.
  1. Form a team of 2 students. Find a partner team for your team: Team A and Team B.
  2. Each team completes Question 2(a)-(c) in **30 minutes:** Plan your design revision and evaluate what design principles you apply.
  3. A member of your team elaborates on your design plan and explains the design principles to a member of your partner team in **10 minutes**. Your partner team member may raise questions on design rationales for clarification and suggestions.
  4. Your team takes feedback from your partner teams and revises the design in **5 minutes**.
  5. Your team writes a short plan of your design revision.

Important note: Please submit your answers in Canvas (the Discussion section). Ensure to include the names of both students.

Q1. Abstraction, Information Hiding, and Encapsulation

We exercise how to model data structure by inter-relating its operators. This style of abstract data type is known as the functional style (or you can think of the Lisp programming language).

Stack Example. The following illustrates an example to represent the behavior of Stack by their interfaces in the functional style.

- `new`: create a new Stack.
- `pop`: return the Stack identical to the given Stack but with the top element removed.
- `top`: return the first element of the stack. If the Stack is empty, it returns an `error`.
- `empty`: return whether the stack is empty (`true` for empty, `false` otherwise).
- `push`: return the Stack with an additional and given element at the top.
- Note that we do not create "local variables." All variables are the arguments of the corresponding operators.

Style 1	Style 2
`pop(new()) = new()`	`pop(new()) = new()`
`pop(push(S,i)) = S`	`pop(push(S,i)) = S`
`top(new()) = error`	`top(S) = if empty(S) = true then error`
`top(push(S,i)) = i`	`top(push(S,i)) = i`
`empty(new()) = true`	`empty(new()) = true`
`empty(push(S,i)) = false`	`empty(push(S,i)) = false`

Example 1:
```
  push(pop(push(push(new(),4),7)),3)
= push(           push(new(),4)     ,3)
```

Example 2:
```
  top(pop(push(push(pop(new()), x), y))))
= top(pop(push(push(     new() , x), y))))
= top(          push(     new() , x)       )
= x
```

How do we implement a stack accordingly?

- `S = new Stack()`
- `pop(.)` and `push(.)` maintain the stack (e.g., `S.pop(), S.push(i)`)

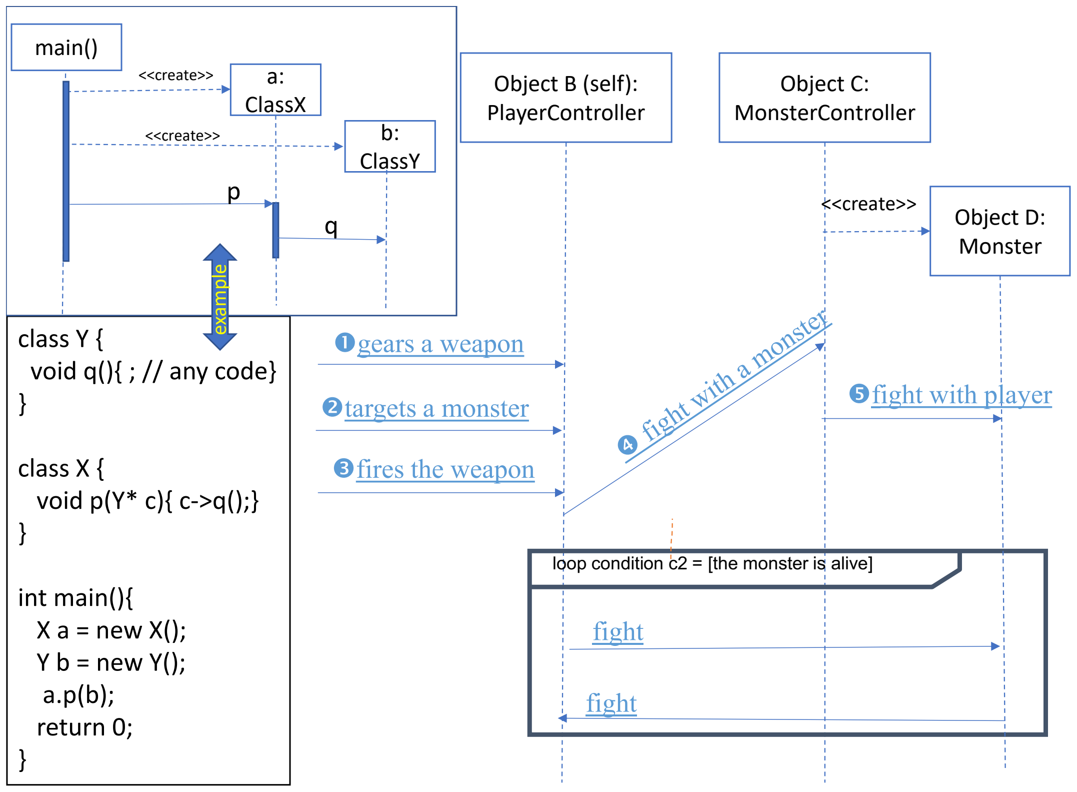
- `top(.)` and `empty(.)` return the stack state info (e.g., `S.top(), S.empty()`)

Your Action: Define a `Set` in the functional style by inter-relating the following operators.

- `empty:` create a new empty `Set`.
- `insert:` return a `Set` with one more element than the given `Set`, and the additional element is the given one.
- `delete:` return a `Set` without the given element from the given `Set`.
- `member:` return whether the `Set` contains the element.
- Note: A condition "if `a` = `b` then `c` else `d`" can be used. Ensure that all variables are used on both sides of the = operator.

Q2. Design Principles

The C++ code mockup on page 3 implements the following sequence diagram for a player to fight with a single monster in a computer game.



```
class Y {
  void q(){ ; // any code}
}

class X {
  void p(Y* c){ c->q();}
}

int main(){
  X a = new X();
  Y b = new Y();
   a.p(b);
  return 0;
}
```

https://www.onlinegdb.com/online_c++_compiler

```
#include<iostream>
using namespace std;
class Player { };
class PlayerController;
class MonsterController;
class Monster;

class Monster {
protected:
        PlayerController* opponent = NULL;
        bool alive = true; // true - alive,  false - killed
        int lifespan = 3;  // if 0 => alive = false, otherwise true
public:
        void fightWithPlayer(PlayerController* o) { opponent = o; }
        bool isAlive() { return alive; }
        void damage(int point) {
                lifespan = lifespan - point;
                cout << "monster is hurt (" << lifespan << ")" << endl;
                if (lifespan <= 0) {
                        cout << "monster is killed" << endl;
                        alive = false;
                }
        }
        void fight();
};

class MonsterController {
        Monster* m;
public:
        MonsterController() { m = new Monster(); }
        Monster* fightWithMonster(PlayerController* pc) { m->fightWithPlayer(pc);
return m; }
```

Console window output

```
player is hurt (4)
monster is hurt (2)
player is hurt (3)
monster is hurt (1)
player is hurt (2)
monster is hurt (0)
monster is killed
```

```
};

class PlayerController {
        MonsterController* mc;
        int lifespan = 5;   // lifespan of Player if 0 then alive = false, else true
public:
        PlayerController(MonsterController* mc_init) { mc = mc_init; }
        void UC2MainPath() { gear(); target(); fire(); }
        void gear() {};
        void target() {};
        void fire() { fireAWeapon(); };
        void fight(Monster* m) { m->damage(1); };
        void damage(int point) {   lifespan = lifespan - point;
                cout << "player is hurt (" << lifespan << ")" << endl;}
        void fireAWeapon()
        {
                Monster* m = mc->fightWithMonster(this);
                while (m->isAlive()) { m->fight(); fight(m); }
        }
};

void Monster::fight() { opponent->damage(1); }


int main() {
        MonsterController* mc = new MonsterController();
        PlayerController* pc = new PlayerController(mc);
        pc->UC2MainPath();
        return 0;
}
```

Your action: The code has several design problems (note that there is no unique answer). Evaluate the **current** program design against the design principles (e.g., SRP, OCP, LSP, ISP, DIP, Abstraction, Information Hiding, Encapsulation, Code Smells) **to support the following anticipated changes.** *Note that although many other changes are required, let's **primarily focus** on issues of (the code smells and) violations of design principles* for these anticipated changes.

(a). The game will support one player controller serving a list of players one by one. When the player controller serves a particular player, it will kill a monster on behalf of the player. Individual players are responsible for keeping their own states.

(b). The game will support several monsters. Like the player controller, the monster controller will serve all monsters.

(c). A special type of monster is a dark magician. When a player fights with a dark magician, the magician does not hurt the player's lifespan. Instead, the magician will cast a spell sometimes. The spell will temporarily convert the player into a monster for a time period only known to the player. While a player is being affected by such a spell, another player may "target" this "player-based monster" and then "fire." Note that if this player-based monster is fighting with another player, the message for this player-based monster should be "player is hurt" instead of "monster is hurt."

(a)

Issue (description, violating which design principles):

Possible solutions:

(b)

Issue (description, violating which design principles):

Possible solutions:

(c)

Issue (description, violating which design principles):

Possible solutions: