Lab: Object Communication and Events

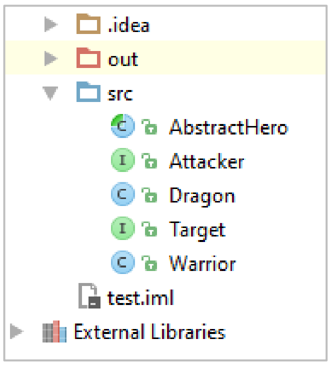
Problems for exercises and homework for the ["Java OOP Advanced" course @ SoftUni](https://softuni.bg).

You can check your solutions here: https://judge.softuni.bg/Contests/537/Object-Communication-and-Events-Lab .

Part I: Chain of Responsibility, Command Design Pattern

0. Resources

You are given a file with some classes. Place them in a new project and get familiar with them.



```
▶ .idea
▶ out
▼ src
    AbstractHero
    Attacker
    Dragon
    Target
    Warrior
  test.iml
▶ External Libraries
```

1. Logger - Chain of Responsibility

Create a **Chain of Responsibility** Logger and provide:

- `enum LogType`
  - `values - ATTACK, MAGIC, TARGET, ERROR, EVENT`
- `interface Handler`
  - `void handle(LogType, String)`
  - `void setSuccessor(Handler)`
- Concrete loggers that log messages to console:
  - `CombatLogger`
  - `EventLogger`

Log messages in format (**"TYPE: message"**)

Solution

Create enum LogType

```java
public enum LogType {
    ATTACK, MAGIC, TARGET, ERROR, EVENT
}
```

Create **`Handler`** interface

```java
public interface Handler {
    void handle(LogType type, String message);
    void setSuccessor(Handler handler);
}
```

You can create an **`abstract`** logger, so you can abstract some of the functionalities

```java
public abstract class Logger implements Handler {

    private Handler successor;

    public void setSuccessor(Handler successor) { this.successor = successor; }

    protected void passToSuccessor(LogType type, String message) {...}

    public abstract void handle(LogType type, String message);
}
```

Create a concrete logger that **`extends Logger`**

```java
public class CombatLogger extends Logger {

    @Override
    public void handle(LogType type, String message) {
        if (type == LogType.ATTACK || type == LogType.MAGIC) {
            System.out.println(type.name() + ": " + message);
        }

        super.passToSuccessor(type, message);
    }
}
```

Test the logger through you client

```java
public static void main(String[] args) {
    Logger combatLog = new CombatLogger();
    Logger errorLog = new ErrorLogger(); // implement by yourself

    combatLog.setSuccessor(errorLog);

    combatLog.handle(LogType.ATTACK, "some attack");
    combatLog.handle(LogType.ERROR, "some error");
    combatLog.handle(LogType.EVENT, "some event"); // should not log
}
```

Don't forget to **inject the logger** into any model that needs to log/print messages

2. Command

Create a **Command Pattern** Executor and provide:

- **`interface Command`**
    - **`void execute()`**
- **`interface Executor`**

    - **void executeCommand(Command command)**

- Concrete Executor named **CommandExecutor implements Executor**
- Concrete Commands
    - **TargetCommand** with constructor **(Attacker, Target)**
    - **AttackCommand** with constructor **(Attacker)**

Hints

Create the interfaces

Each new command should implement **Command**, so it can be executed by the **Executor**

```
public class AttackCommand implements Command
```

Create as many commands as you like

Test your commands

```
public static void main(String[] args) {
    Logger combatLog = new CombatLogger();
    Logger eventLogger = new EventLogger();

    combatLog.setSuccessor(eventLogger);

    Attacker warrior = new Warrior("Warrior", 10, combatLog);
    Target dragon = new Dragon("Dragon", 100, 25, combatLog);

    Executor executor = new CommandExecutor();
    Command target = new TargetCommand(warrior, dragon);
    Command attack = new AttackCommand(warrior);
}
```

Part II: Mediator, Observer Design Pattern

3. Mediator

Implement a Mediator Pattern groups and provide:

- **interface AttackGroup**
    - **void addMember(Attacker)**
    - **void groupTarget(Target)**
    - **void groupAttack()**
- Concrete class **Group** that implements **AttackGroup**
- Concrete Commands:
    - **GroupTargetCommand** with constructor **(AttackGroup, Target)**
    - **GroupAttackCommand** with constructor **(AttackGroup)**

Hints

Implement **Group implements AttackGroup** - this will be the concrete mediator

```java
public class Group implements AttackGroup {

    private List<Attacker> attackers;

    public Group() { this.attackers = new ArrayList<>(); }

    @Override
    public void addMember(Attacker attacker) { this.attackers.add(attacker); }

    public void groupTarget(Target target) {...}

    public void groupAttack() { attackers.forEach(Attacker::attack); }

    public void groupTargetAndAttack(Target target) {...}
}
```

Create some group commands, following the logic from the previous problem

Test the mediator

```java
public static void main(String[] args) {
    Logger combatLog = new CombatLogger();
    Logger eventLogger = new EventLogger();

    combatLog.setSuccessor(eventLogger);

    AttackGroup group = new Group();
    group.addMember(new Warrior("Warrior", 10, combatLog));
    group.addMember(new Warrior("ElderWarrior", 13, combatLog));

    Target dragon = new Dragon("Dragon", 100, 25, combatLog);

    Executor executor = new CommandExecutor();

    Command groupTarget = new GroupTargetCommand(group, dragon);
    Command groupAttack = new GroupAttackCommand(group);
}
```

4. Observer

Implement the **Observer Design Pattern** by providing the following:

- interface **`Subject`**
  - **`void register(Observer)`**
  - **`void unregister(Observer)`**
  - **`void notifyObservers()`**
- interface **`Observer`**
  - **`update(int)`**

If a **Target** dies, it should **send reward** to all of its **Observers**

Hints

Create the interfaces

`Attacker` should be the `Observer`

* `Dragon` should be the `Subject` - (the easiest way is to make `Target extends Subject`, but this is violation of the **Interface Segregation Principle**). The better solution is to create a new interface `ObservableTarget` and `implement` both `Target` and `Observer`.