07 - Command, Memento

Credits: Askar Gafurov

Memento

Pouzivame, ked chceme ulozit vnutorny stav objektu a neskor vratit objekt do povodneho stavu, pricom chceme skryt pred klientom implementaciu tohto procesu (napriklad chceme ulozit stav hry a neskor ho vratit naspat).

Zakladna struktura:

```
public class Originator {
    // internal state is stored in the object's attributes
    // we can store all or only some of the variables in Memento object
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public int getCounter() {
        return counter;
    }

    public Memento saveToMemento() {
        return new Memento(this);
    }

    public void restoreFromMemento(Memento memento) {
        this.counter = memento.getCounter();
    }

    // alternatively, we may let Memento restore the state of Originator
    public void restoreFromMemento2(Memento memento) {
        memento.restoreOriginator(this);
    }

    public static class Memento {
        private final int counter;

        private Memento(Originator originator) {
            counter = originator.counter;
        }

        private int getCounter() {
            return counter;
        }
```

```
        // in alternative mode, the Memento takes care of restorement
        private void restoreOriginator(Originator originator) {
             Originator.counter = this.counter;
        }

        // also, we may allow or forbid to interchange Mementos between
        // different objects (how?)
    }
}
```

Pouzitie:

```
class Demo {
    void demo() {
        Originator o = new Originator();
        o.increment();
        o.increment();
        Originator.Memento s1 = o.saveToMemento();
        o.increment();
        o.increment();
        System.out.println(o.getCounter()); // should print 4
        o.restoreFromMemento(s1);
        System.out.println(o.getCounter()); // should print 2
    }
}
```

Ulohy:

U: Vytvorte triedu `MonitorConfiguration` s gettermi a settermi pre atributy `contrast`, `brightness` a `frequency` (t.j. s metodami `getContriast`, `setContrast` atd). Pomocou navrhoveho vzoru Memento pridajte moznost ulozit nastavenia a vratit ich neskor.

U*: Upravte triedu `MonitorConfiguration` tak, aby Memento ukladal ukladal vysledky nie do operacnej pamate, ale do pomocneho suboru (napriklad `monitor.cfg`). Nasledne by pri volani metody `restoreFromMemento` data boli nacitane z pomocneho suboru.

Command

Pouzivame, ked chceme zabalit do objektu vsetky potrebne informacie pre vykonanie nejakej cinnosti s moznostou vykonania danej cinnosti v neskorsom case. Mozne pouzitia:

1. Oddelenie aplikacneho rozhrania od logiky (napr. tlacitka v aplikacii s metodami typu `onClick`)
2. Moznost vykonavania uloh specifickym sposobom (napr. podla priority, paralelne, s ohladom na zavislosti medzi nimi, atd)

3. Operacie ‘Undo/Redo’

Zakladna struktura I:

```
// e.g. a TV set
public class Receiver {
    public void actionA() {}
    public void actionB() {}
}

// e.g. “Turn TV On”
public interface Command {
    void execute();
}

// e.g. “Turn TV On”
public class ActionACommand implements Command {
    private final Receiver receiver;
    public ActionACommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        // maybe something more interesting than this
        receiver.actionA();
    }
}

public class ActionBCommand implements Command {
    private final Receiver receiver;
    public ActionBCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        receiver.actionB();
    }
}

// e.g. a TV Controller with buttons
public class Invoker {
    private final HashMap<String, Command> commandMap = new HashMap<>();


    // creating a new “button”
    public void register(String name, Command command) {
        commandMap.put(name, command);
    }
```

```
    // pushing a “button”
    public void execute(String name) {
        commandMap.get(name).execute();
    }
}
```

Pouzitie I:

```
class Demo {
    void demo() {
        Receiver r = new Receiver();

        Invoker invoker = new Invoker();
        invoker.register("action A", new ActionACommand(r));
        invoker.register("action B", new ActionBCommand(r));

        // the client doesn't have to know how
        // to operate `Receiver` object
        // he uses only `Invoker` interface
        // imagine the `Invoker` as a remote controller for a TV
        invoker.execute("action A");
        invoker.execute("action B");
    }
}
```

Zakladna stuktura II - Priority Queue:

```
public class Planner {
    private final Queue<PriorityCommand> queue = new PriorityQueue<>();

    public void addCommand(Command command, int priority) {
        queue.add(new PriorityCommand(command, priority));
    }

    public void executeHighestPriority() {
        Command highest = queue.peek();
        highest.execute();
        queue.remove();
    }

    // the sorting by priority may be achieved in other fashions as well
    private class PriorityCommand implements Command,
Comparable<PriorityCommand> {
        private final Command command;
        private final int priority;

        private PriorityCommand(Command command, int priority) {
```

```
            this.command = command;
            this.priority = priority;
        }

        @Override
        public int compareTo(PriorityCommand o) {
            // command with higher priority will be executed sooner
            return o.priority - this.priority;
        }

        @Override
        public void execute() {
            command.execute();
        }
    }
}
```

Zakladna stuktura III - Undo:

```
public interface Command {
    void execute(); // store the current state and then apply changes
    void undo(); // restore Receiver to the state before the execution
    // this may be done with or without Memento pattern
}

public ConcreteUndoableCommand implements Command {
    private Receiver.Memento memento;
    private final Receiver receiver;

    public ConcreteUndoableCommand(Receiver receiver) {
         this.receiver = receiver;
    }

    @Override
    public void execute() {
         memento = receiver.saveToMemento();
         // … do stuff …
    }

    @Override
    public void undo() {
         receiver.restoreFromMemento(memento);
    }
}



public class CommandHistory {
    private final Stack<Command> history = new Stack<>();
```

```
    public void doCommand(Command command) {
        command.execute();
        history.add(command);
    }

    public void undoLastCommmand() {
        Command lastCommand = history.peek();
        lastCommand.undo();
        history.pop();
    }
}
```

Ulohy:

U: Vytvorte triedu “MockTask” pod rozhranim Command s konstruktorom `MockTask(int duration, int id)`, ktorej metoda `execute()` pocka prislusny pocet milisekund a nasledne vypise svoj identifikator na standardny vystup.

U: Vytvorte Planner (podla zakladnej struktury II), ktoreho prioritou bude priamo dlzka vykonavania ulohy. Implementujte metodu `executeAll()`, ktora vykona postupne vsetky nahrate ulohy v poradi podla priority. Pridajte 50 nahodnych uloh do Planneru (staci napriklad `planner.addCommand(new MockTask(dur, i), dur)`). Vyskusajte metodu `executeAll()`.

U**: Implementujte moznost pridavania prerekvizit pre jednotlive ulohy (t.j. Ak uloha X ma ako prerekvizitu ulohu Y, tak sa najprv ma spustit uloha Y a az po jej dokonceni uloha X) napriklad pridanim metody `addCommand(Command command, int priority, int id, int … prerequisiteIds)`. Upravte metody `executeAll()` a `executeHighestPriority()` tak, aby vykonavali ulohy respektujuc ich prerekvizity. Predpokladajte, ze cyklicke zavislosti neexistuju.

U: Implementujte triedu Calculator (*“receiver” part*) s nasledovnym rozhranim:

- getAcc() - vrati aktualnu hodnotu premennej `accumulator`
- setAcc(int newAcc) - nastavi hodnotu premennej `accumulator` na hodnotu `newAcc`
- add(int x) - pripocita ku akumulatoru hodnotu x
- multiply(int x) - vynasobi akumulator hodnotou x

U: Implementujte pomocou vzoru Command operácie Add, Subtract, Multiply, Repeat nad triedou Calculator. Operácia Repeat pri inicializácii dostane operáciu, ktorú má vykonať, a počet opakovaní.

U: Implementujte triedu ConsoleCalculator (*“invoker” part*), ktorá po zavolaní metódy run() bude prijímať zo štandardného vstupu nasledovné príkazy:

- + n
- - n
- * n

- R n : zopakuj predošlý príkaz n krát
- EXIT : ukončí beh metódy run()

Po každej operácii sa vypíše aktuálny stav počítadla.

U: Pre každú operáciu implementujte metódu undo(), po zavolaní ktorej sa počítadlo vráti do stavu pred vykonaním danej operácie.

U: Implementujte podporu operácie Undo ("U" na vstupe).

U*: Implementujte podporu operácie Redo ("Y" na vstupe). Pre ukladanie histórie operácií použite zásobník (java.util.Stack).