08 - Bridge, State, Observer

Credits: Askar Gafurov

Bridge

Pouzivame, ked chceme oddelit triedy od implementacii konkretnych metod za ucelom ich nezavisleho vyvoja. Jeden zo sposobov, ako si to predstavit, je napriklad standardna systemova kniznica operacneho systemu (prikazy typu `openFile`, `print`, `listDirectory`), kde kazda z metod ma standardne rozhranie napriec roznymi operacnymi systemami, ale ich implementacie su rozlicne. Konkretne implementacie pre jednotlive systemy su od nas skryte, pouzivame len standardne rozhranie.

Zakladna struktura:

```
class Remote { // Abstraction
    protected Device device; // Implementor

    public Remote(Device device) {
        this.device = device;
    }

    void togglePower() {
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    void volumeUp() {
        int currentVolume = device.getVolume();
        if (currentVolume < 100) {
            device.setVolume(Math.min(100, currentVolume + 5));
        }
    }

    void volumeDown() {
        // analogically
    }
}

class AdvancedRemote extends Remote {
    public AdvancedRemote(Device device) {
        super(device);
    }
```

```
    void mute() {
        device.setVolume(0);
    }
}

interface Device { // Implementor
    boolean isEnabled();
    void enable();
    void disable();

    int getVolume();
    void setVolume(int percentage);
}

class TV implements Device {
    // ...
}

class Radio implements Device {
    // ...
}
```

Pouzitie:

```
public class BridgeDemo {
    static void demo() {
        Device tv = new TV();
        Remote remote = new Remote(tv);

        remote.togglePower();
        remote.volumeDown();

        Device radio = new Radio();
        Remote remote2 = new Remote(radio);
        remote2.togglePower();
        remote2.volumeDown();

        AdvancedRemote advancedRemote = new AdvancedRemote(tv);
        advancedRemote.mute();
    }
}
```

Ulohy:

U: Implementujte (pomocou navrhoveho vzoru Bridge) rozhranie `TablePrinter` s metodami `openOutputFile(String filename)`, `closeOutputFile()`, `printHeader(List<String> header)` a `printRow(Map<String, Object> row)`, kde argument `row` bude obsahovat dvojice (nazov_stplca -> hodnota). Implementujte dve konkretne implementacie `CSVPrinter` a `TSVPrinter` (v prvej sa ako delimiter pouzije ciarka, v druhej tabulator \t). Vytvorte dalej

triedu `TextLogger`, ktora bude nacitavat zo standardneho vstupu text a bude ho spolu s poradovym cislom riadku priebezne ukladat do urceneho textoveho suboru v urcenom formate.

U: TODO jednoduchsia uloha

State

Pouzivame, ked chceme dynamicky menit spravanie objektu (t.j. jeho metody) na zaklade jeho vnutorneho stavu. Da sa to vnimat Strategy pattern, ktory si sam moze zmenit strategiu. Je to dalsi sposob ako oddelit data od algoritmov na ich spracovavanie.

Diagram prechodov medzi stavmi do bolesti zubov pripomina prechodovu funkciu konecneho automatu (respektive riadiacej jednotky lubovolneho ineho formalneho stroja).

Zakladna struktura:

```
public interface State {
    void actionA(Context context);
    void actionB(Context context, int param);
}


public class ConcreteStateX implements State {
    @Override
    public void actionA(Context context) {
        // stuff
    }

    @Override
    public void actionB(Context context, int param) {
        // stuff
        // ...
        // if we want to change the behaviour to `ConcreteStateY`
        context.setState(new ConcreteStateY());
    }
}

public class ConcreteStateY implements State {
    @Override
    public void actionA(Context context) {
        // stuff
        // if we want to change the behaviour to `ConcreteStateX`
        context.setState(new ConcreteStateX());
    }

    @Override
    public void actionB(Context context, int param) {
```

```
        // stuff
    }
}

public class Context {
    // some internal variables describing the context
    // ...
    // if we want `ConcreteStateX` as an initial behaviour
    private State state = new ConcreteStateX();

    // we cannot make this method private
    // the best we can do is the package-level access (no modifier)
    void setState(State newState) {
        this.state = newState;
    }

    public void actionA() {
        state.actionA(this);
    }

    public void actionB(int param) {
        state.actionB(this, param);
    }
}
```

Pouzitie:

```
class Demo {
    public void demo() {
        Context context = new Context();
        context.actionA();
        context.actionB(47); // after this call the state is switched to
                             // `ConcreteStateY`
        context.actionA();   // after this call the state is switched to
                             // `ConcreteStateX`
    }
}
```

Ulohy:

U: Implementujte kavomat s tlacitkami (t.j. verejnym rozhranim kontextu):

- switchPower() - prechadza zo stavu OFF do stavu IDLE a z lubovolneho ineho stavu do stavu OFF
- pressEspresso() - prechadza zo stavu IDLE do stavu MAKING_ESPRESSO a v inych stavoch to tlacitko nerobi nic
- pressCappuccino() - <...> stavu MAKING_CAPPUCCINO <...>

Stav MAKING_ESPRESSO vypise na standardny vystup “Errrrrrrr!” a po dvoch sekundach vypise “Take your espresso!” a prejde do stavu IDLE (cely proces sa moze odohrat priamo v konstruktore daneho stavu). Obdobne nech funguje aj stav MAKING_CAPPUCCINO.

U: Implementujte pomocou navrhoveho vzoru State konecny automat nad abecedou {a, b} so stavmi A, B, C, D (A je pociatocny, C je akceptacny) s nasledovnou prechodovou funkciou:

	a	*b*
A	B	D
B	D	C
C	D	B
D	D	D

Aky jazyk akceptuje dany konecny automat? (riesenie: ab(bb)*)
Aky jazyk by bol akceptoval konecny automat, ak by jedinym akceptacnym stavom bol stav D? (riesenie: (b | ab*a)[ab]*)
Rozhranie triedy Context ma mat dve verejne metody:

1. void nextSymbol(char symbol) - nacita dalsi symbol
2. boolean isAcceptingState() - vrati true alebo false podla toho, ci dany stav je akceptacny

U*: Vidite, ze v danom pripade triedy jednotlivych stavov su velmi podobne. Implementujte tovaren, ktora bude generovat objekty prislusneho typu, t.j. triedu StateFactory s verejnym rozhranim `State getStateA()`, `State getStateB()`, etc.

U**: Vidite, ze v danom pripade aj tato tovaren je velmi jednoducha. Implementujte tovaren tovarni, ktora pri inicializacii dostane popis konecneho automatu vo vhodnej reprezentacii a z toho vytvori potrebnu tovaren pre predchadzajucu ulohu (pre konzistentnost s predchadzajucou ulohou predpokladajte fixne stavy A, B, C, D).

Observer

Pouzivame, ked chceme notifikovat objekty o zmenach v inych objektoch (event handling).

Zakladna struktura I:

```
// Observer observes the changes in Subject
public interface Observer {
    void update(); // add some arguments if necessary
}

public class Subject {
```

```
    // list of registered observers
    private final List<Observer> observers = new ArrayList<>();
    // internal state, e.g. a single integer
    private int state = 0;

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    private void notifyObservers() {
        for (Observer o: observers) {
            o.update();
        }
    }

    public int getState() {
        return state;
    }

    public void setState(int newState) {
        this.state = newState;
        notifyObservers();
    }
}


public class ConcreteObserver implements Observer {
    private Subject subject; //set e.g. by initialization

    @Override
    public void update() {
        int newState = subject.getState();
        // now do something useful with this information
    }
}
```

Pouzitie I:

```
class Demo {
    void demo() {
        Subject subject = new Subject();
        Observer o1 = new ConcreteObserver();
        Observer o2 = new ConcreteObserver();
        subject.addObserver(o1);
        subject.addObserver(o2);
```

```
        subject.setState(47);
        subject.removeObserver(o2);
    }
}
```

Zakladna struktura II (pomocou Observable):

```
public class Subject extends Observable {
    private Integer state;

    public void setState(Integer newState) {
        this.state = newState;
        setChanged();
        notifyObservers(this.state);
    }
}

public class ConcreteObserver implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        Integer state = (Integer) arg;
        // do what you need to do with this information
    }
}
```

Pouzitie II:

```
class Demo {
    void demo() {
        Subject subject = new Subject();
        Observer o1 = new ConcreteObserver();
        Observer o2 = new ConcreteObserver();
        subject.addObserver(o1);
        subject.addObserver(o2);

        subject.setState(42);
        subject.deleteObserver(o2);
    }
}
```

Ulohy:

U: Implementujte triedu Burza s metódou setState(double kurz), ktorá bude predstavovať zmenu vymenného kurzu eura ku doláru. Implementujte triedy Bull a a Bear, ktoré po každej zmene kurzu budú vypisovať svojú reakciu na to (medvede sa tešia, keď kurz klesá, a býky naopak).

U: Upravte príklad na kalkulačku (cvičenie 07) tak, aby každá zmena v hodnote bola propagovaná všetkým zaregistrovaným Observerom. Implementujte dve triedy Observerov, ktoré budú implementovať štandardný Java interface Observer.
LoggingObserver - Vždy po zmene hodnoty vypíše zmenu na konzolu, napr. 2 -> 8
HistoryObserver - pamätá si všetky hodnoty, po zavolaní vlastnej metódy print() ich vypíše, napr. 2 -> 8 -> 24 -> 22 -> 2 -> 22 ...
Triedu typu Observable implementujte rozšírením štandardnej triedy Observable: Vyskúšajte aj alternatívny spôsob, kedy všetky metódy dedené z Observable implementujete sami.