ŠTRUKTURÁLNE NÁVRHOVÉ VZORY
(Composite, Decorator, Facade, Flyweight)

Tomáš Vavro, Tamara Tučková, Emma Čavojová, Martin Černý

29.9.2022

Obsah

1 Úvod

V dokumentácií sa v kapitole 2 zaoberáme témou návrhových vzorov. V kapitole 3 bližšie rozoberáme štrukturálne návrhové vzory **Composite**, **Decorator**, **Facade** a **Flyweight** 3. V kapitole 4 predstavíme vytvorený softvér, ktorý implementuje spomenuté návrhové vzory.

2 Návrhové vzory

Návrhové vzory reprezentujú stratégie na riešenie problémov, ktoré sa často vyskytujú pri návrhu softvéru. Tieto stratégie sú nezávislé od programovacieho jazyka a preto neposkytujú konkrétnu implementáciu, ale **ideu**, pomocou ktorej je možné vytvoriť flexibilný, prepoužiteľný a udržateľný kód.

Na rozdiel od algoritmov, ktoré taktiež popisujú riešenia pre známe problémy, návrhové vzory predstavujú high-level riešenie, ktorého konkrétna implementácia závisí od programátora. Dôsledkom toho môže byť kód 2 programov, ktoré implementujú rovnaký vzor, odlišný.

2.1 Formálny popis vzoru

Každý návrhový vzor má svoju špecifikáciu, alebo množinu pravidiel. Formálne je možné vzory popísať na základe:

- **Zámeru** - definícia vzoru, stručný popis problému a riešenia
- **Motivácie** - popisuje príklady použitia, tzn. konkrétne riešenia, ktoré vzor dokáže priniesť
- **Štruktúry tried** - znázorňuje jednotlivé časti vzoru a ich vzťahy (UML)
- **Ukážky kódu**

Formálny popis budeme využívať na zadefinovanie vzorov Composite, Decorator, Facade, Flyweight.

2.2 História vzorov

Koncept vzorov prvý raz popísal **Christopher Alexander** v jeho knihe *A Pattern Language: Towns, Buildings, Construction*. Kniha popisuje návrh mestského prostredia prostredníctvom jazyka, ktorého jednotkami sú vzory. Vzory popisujú, ako by mali vyzerať budovy, parky, susedstvo (, atď.) v meste.

Tento koncept bol v roku 1994 aplikovaný v programovaní ako návrhový vzor v knihe *Design Patterns: Elements of Reusable Object-Oriented Software*, ktorý bola publikovaná autormi **Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm**. Kniha predstavila 23 vzorov na riešenie problémov objektovo-orientovaného návrhu softvéru a rýchlo sa stala best-sellerom. [2] Odvtedy bolo zavedených veľa ďalších vzorov.

2.3 Motivácia pre používanie vzorov

- Návrhové vzory môžu napomôcť k urýchleniu vývojového procesu softvéru, nakoľko predstavujú odskúšané a otestované paradigmy.
- Použitím návrhových vzorov sa dá predísť problémom, ktoré môžu byť viditeľné až v neskoršom štádiu implementácie.
- Použitie vzorov môže docieliť lepšiu čitateľnosť a prehľadnosť kódu pre vývojárov a architektov.

2.4 Základné delenie

Vzory môžu byť rozdelené na základe zámeru, alebo účelu do 3 skupín:

- **Vytvárajúce (angl. *creational*) vzory** - zabezpečujú mechanizmy na vytváranie objektov, ktoré zvyšujú flexibilitu a prepoužiteľnosť existujúceho kódu.
- **Štrukturálne (angl. *structural*) vzory** - podrobne rozoberáme v kapitole 3.
- **Vzory zamerané na správanie objektov (angl. *behavioral*)** - zabezpečujú efektívnu komunikáciu a rozdelenie zodpovedností medzi objektmi.

3 Štrukturálne návrhové vzory

Štrukturálne návrhové vzory popisujú spôsoby, ako zlučovať objekty a triedy do väčších štruktúr tak, aby bola zachovaná flexibilita a efektívnosť. Existuje 7 štrukturálnych návrhových vzorov, avšak naším cieľom bolo detailne predstaviť 4 z nich, nakoľko zostávajúce 3 boli prezentované na prednáške.

Pokiaľ nie je uvedené inak, všetky formálne informácie z tejto kapitoly pochádzajú zo zdrojov [1], [3] a a[2].

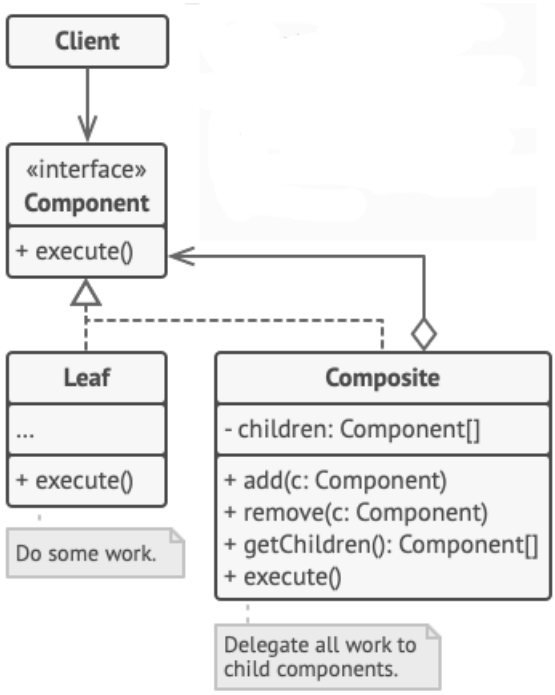
3.1 Composite

Zámer: Composite umožňuje skladať (angl. *compose*) objekty do stromových (hierarchických) štruktúr a s vytvorenými štruktúrami (jednoduchými objektmi a zoznamami objektov) pracovať ako s individuálnymi objektmi. Napriek tomu, že jednoduché a zložené objekty sú rozdielne spracúvané v kóde, je možné s nimi pracovať prostredníctvom jedného rozhrania. Vzor je možné kombinovať s Decorator-om, alebo Flyweight-om, ktorým je možné reprezentovať listy hierarchickej štruktúry.

Motivácia: Stromové štruktúry sú v informatike veľmi časté, preto sa vzor Composite dá aplikovať na mnoho problémov. Prostredníctvom Composite-u dokážeme modelovať súborový systém, v ktorom je splnený predpoklad narábania

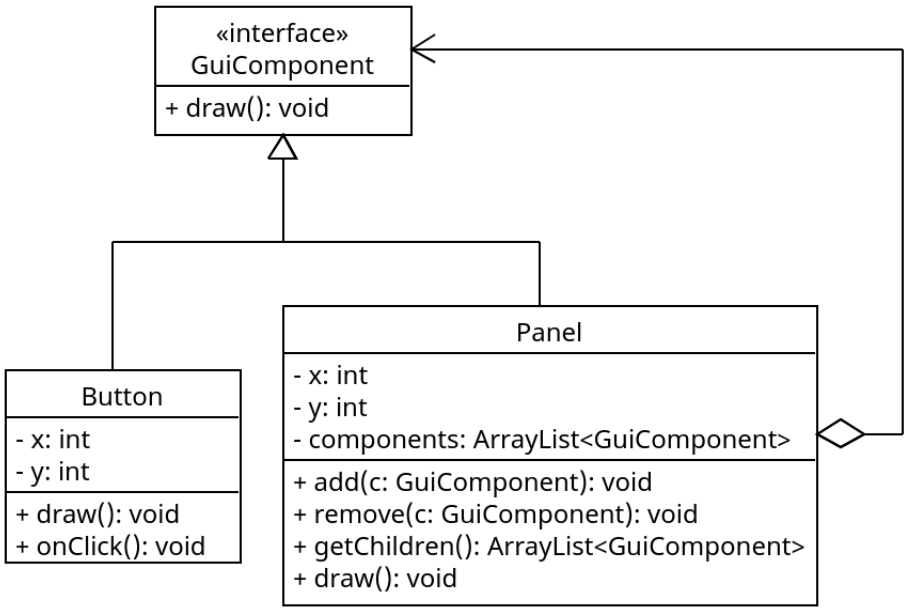
s kolekciou jednoduchých a zložených objektov, t. j. súborov a priečinkov. Ďalším príkladom môže byť hierarchia grafických objektov v používateľskom rozhraní.

Štruktúra a UML: Interface *Component* popisuje operácie, ktoré sú spoločné pre jednoduché a zložené objekty, a teda klientovi umožňuje pristupovať jednotne k jednoduchým, aj zloženým objektom. Trieda *Leaf* predstavuje jednoduchý objekt, ktorý neobsahuje ďalšie štruktúry. Trieda *Composite* prestavuje zložený objekt, ktorý obsahuje ďalšie objekty (kolekcie objektov). K objektom pristupuje prostredníctvom interface-u *Component* tak, že po prijatí požiadavky prehľadá podstrom detí, spracuje výsledok a vráti ho klientovi. Obrázok 1 zachytáva všeobecný UML diagram tried pre vzor Composite.



Obr. 1: Štruktúra vzoru Composite

Príklad: Klient vytvára GUI. V GUI používa kontajner *Panel*, ktorý z hľadiska vzoru Composite zastáva úlohu zloženého objektu (trieda Composite). *Button* je jednoduchý grafický objekt, ktorý vo vzore Composite zastáva triedu Leaf. Obrázok 2 popisuje UML diagram tried pre tento príklad.



Obr. 2: Príklad použitia vzoru Composite

Implementácia príkladu:

Výpis 1: Vzorový príklad implementácie návrhového vzoru Composite

```
public interface GuiComponent {
    void draw();
}

public class Button implements GuiComponent {
    int x, y;

    @Override public void draw() { ... }
    public void onClick() { ... }
}

public class Panel implements GuiComponent {
    ArrayList<GuiComponenet> components;

    public void add(GuiComponent c) { ... }
    public void remove(GuiComponent c) { ... }
    public Component[] getChildren() { ... }
    public void draw() {
        for (GuiComponent child:children) child.draw();
        // draw panel itself
```

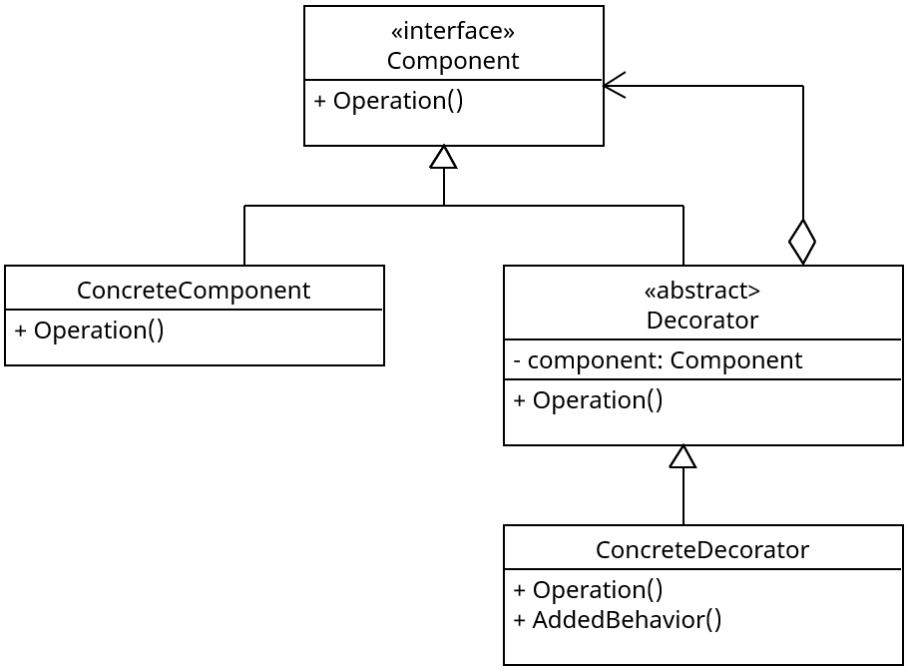
```
    }
}

public class Application {
    GuiComponenl panel;

    public static void main(String[] args) {
        Button b1 = new Button();
        Button b2 = new Button();
        Panel leftPanel = new Panel();
        Panel rightPanel = new Panel();
        leftpanel.add(b1);
        rightpanel.add(b2);
        panel = new Panel();
        panel.add(leftPanel);
        panel.add(rightPanel);
        panel.draw();
    }
}
```

3.2 Decorator

Zámer: Decorator (prípadne *Wrapper*) obaľuje existujúce objekty a dynamicky im pridáva dodatočnú funkcionalitu. V prípade použitia viacerých decorator-ov sú výsledné objekty štruktúrované do stack-u.

Motivácia: Máme triedu na načítanie a zápis dát. Predstavme si situáciu, že klientsky kód vytvorí inštanciu tejto triedy a použije ju na manipuláciu s citlivými dátami. Decorator umožní tieto dáta zašifrovať nezávisle od kódu, ktorý tieto dáta využíva, ako aj od presnej implementácie triedy načítavania, resp. zapisovania dát.

Štruktúra a UML: Máme komponent *ConcreteComponent*, ktorý implementuje interface *Component*. Tento komponent poskytuje určitú funkcionalitu prostredníctvom metódy *Operation*. Abstraktný Decorator tiež implementuje tento interface, ale naviac si ukladá referenciu na objekt rohrania *Component*. Decorator implementujeme tak, že vytvoríme inštanciu *ConcreteComponent* a obalíme ju inštanciou decorator-a *ConcreteDecorator*. Pri zavolaní metódy *Operation* na decorator-e *Decorator* je zavolaná táto metóda aj na referencovanom objekte Component-u. Obrázok 3 zachytáva všeobecný UML diagram tried pre vzor Decorator.

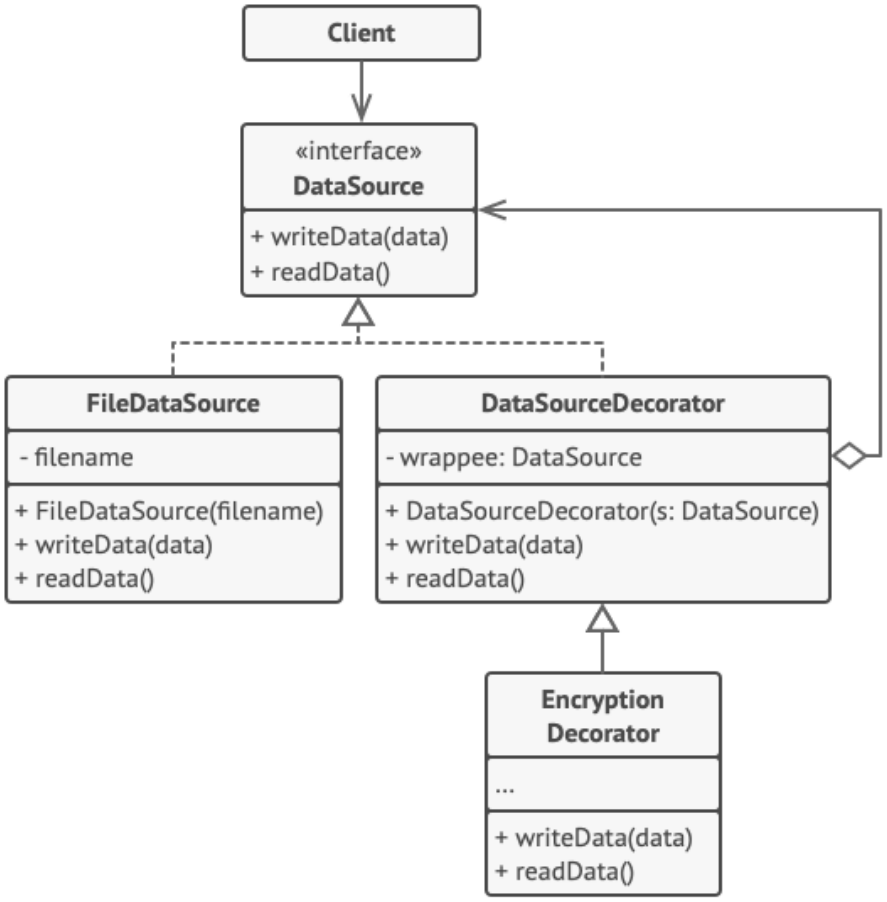


Obr. 3: Štruktúra vzoru Decorator

Príklad: Aplikácia obalí objekt *DataSource* decorator-om, ktorý zabezpečí šifrovanie dát nasledovným spôsobom:

- pred tým, ako sú dáta zapísané na disk sú zašifrované, a
- po tom, ako sú dáta prečítané z disku, decorator zabezpečí dešifrovanie.

Trieda *FileDataSource* nerozlišuje medzi objektmi *FileDataSource* a *Ecryption Decorator*, nakoľko implementujú rovnaký interface. Dôsledkom toho zapíše dáta do súboru bez toho, aby si „uvedomila“ pozmenenie dát. Obrázok 4 zachytáva UML diagram tried pre tento príklad.



Obr. 4: Príklad použitia vzoru Decorator

Implementácia príkladu:

Výpis 2: Vzorový príklad implementácie návrhového vzoru Decorator

```
public interface DataSource {
    void writeData(data);
    String readData();
}

public class FileDataSource implements DataSource {
    String filename;
```

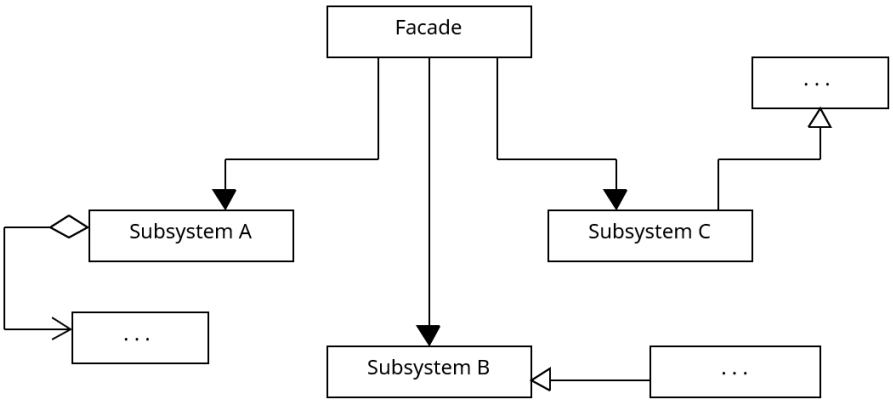
```
    public FileDataSource(String filename) { this.filename = filename; }

    @Override public void writeData() { ... } // to file
    @Override public String readData() { ... } // from file
}

public abstract class DataSourceDecorator implements DataSource {
    protected DataSource wrappee;

    public DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    public void writeData(data) { wrappee.writeData(data); }
    public String readData() { return wrappee.readData(); }
}

public class EncryptionDecorator extends DataSourceDecorator {
    @Override public void writeData(String data) {
        String encrypted = encryptData(data)
        wrappee.writeData(encrypted)
    }

    @Override public String readData() {
        String data = wrappee.readData();
        try {
          data = decryptData(data)
          return data;
        }
        catch(Exception e) {
          return data;
        }
    }

    private void encrypt(DataSource source) { ... }
    private void decrypt(DataSource source) { ... }
}

public class Application {
    public static void main(String[] args){
        String[] salaryRecords = ...;
        // target file
        source = new FileDataSource("file.xls")
        source.writeData(salaryRecords)

        source = new EncryptionDecorator(source)
        source.writeData(salaryRecords)
        // Encryption > FileDataSource
    }
}
```

3.3 Facade

Zámer: Facade poskytuje zjednodušený interface pre komplexný systém, knižnicu, alebo framework. Znížením komplexity sofistikovaného systému umožňuje udržať prehľadný kód a zároveň dodať požadovaný výstup klientovi.

Motivácia: Majme situáciu, v ktorej do aplikácie prirábame modul slúžiaci na úpravu kontrastu a veľkosti obrázkov. Chceme použiť komplexnú knižnicu na manipuláciu s obrázkami, pomocou ktorej môžeme vytvoriť rýchly a efektívny *ImageEditorModule*.

Štruktúra a UML: Máme viacero subsystémov *SubsystemA*, *SubsystemB*, atď. Každý zo subsystémov sprostredkováva klientovi ním požadovanú funkcionalitu, avšak nevýhodou je, že klient musí interagovať so všetkými. Trieda *Facade* poskytuje jeden interface, ktorý poskytuje klientovi zjednotený a zjednodušený prístup k funkcionalitám. Klient posiela požiadavky tomuto interface-u, ktorý ich spracuje a prepošle ďalej. Obrázok 5 zachytáva schematický UML diagram tried pre vzor Facade.

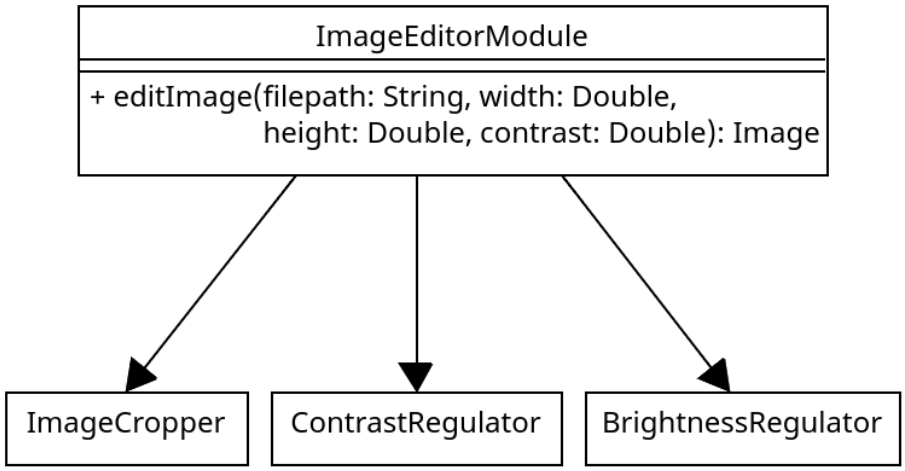


Obr. 5: Štruktúra vzoru Facade

Príklad: Máme knižnicu s množstvom tried, ktoré poskytujú rôzne funkcionality. Namiesto manipulácie s mnohými triedami knižnice priamo v kóde aplikácie vytvoríme triedu *ImageEditorModule*, v ktorej pre udržanie modulárnosti, prehľadnosti a jednoduchosti našej aplikácie enkapsulujeme požadovanú funkcionalitu. Klientska aplikácia preto získa požadovaný výstup zavolaním jednej metódy.

V našom príklade fasádu predstavuje **ImageEditorModule** s metódou editImage(filepath, width, height, contrastParam), ktorá enkapsuluje funkcie slúžiace na spracovanie obrázka (zmenu kontrastu a veľkosti). Triedy *ImageCropper*,

ContrastRegulator, *BrightnessRegulator*, atď., sú triedy knižnice na manipuláciu s obrázkami. Obrázok 6 zachytáva UML diagram tried pre tento príklad.



Obr. 6: Príklad použitia vzoru Facade

Implementácia príkladu:

Výpis 3: Vzorový príklad implementácie návrhového vzoru Facade

```
// classes of 3rd party library for image
public class ImageCropper{ ... }

public class ContrastRegulator{ ... }

public class BrightnessRegulator{ ... }
// ... and more classes

public class ImageEditorModule {
    public Image editImage(String filepath, Double width,
    Double height, Double contrastParam) {
        image = ImageIO.read(new File(filepath));
        if (image) {
            cropper = new ImageCropper();
            contrastRegulator = new ContrastRegulator(contrastParam);
            image = cropper.cropp(image, width, height);
            image = contrastRegulator.adjust();
            return image;
        } else {
            throw new Exception();
        }
```

```
    }
}

public class Application {
    public static void main(String[] args){
        editorModule = new ImageEditorModule();
        image = editorModule.editImage("sunset.jpg", 300, 500, 50);
        image.save();
    }
}
```

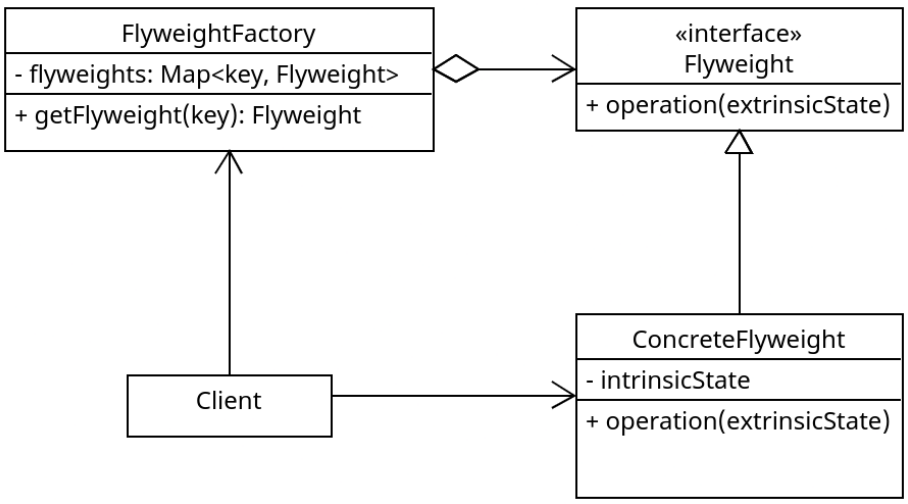
3.4 Flyweight

Zámer: Flyweight, tiež známy ako *Cache*, je návrhový vzor, pomocou ktorého dokážeme znížiť pamäťovú náročnosť ukladania tisícov/miliónov objektov do dostupnej RAM. Každý objekt pozostáva z dvoch stavov:

- **intrinsický stav** je uložený vo Flyweight objekte, podľa best practice by mal byť nemenný.
- **extrinsický stav** je uložený/počítaný klientom a podsunutý Flyweightu po ukončení výpočtov.

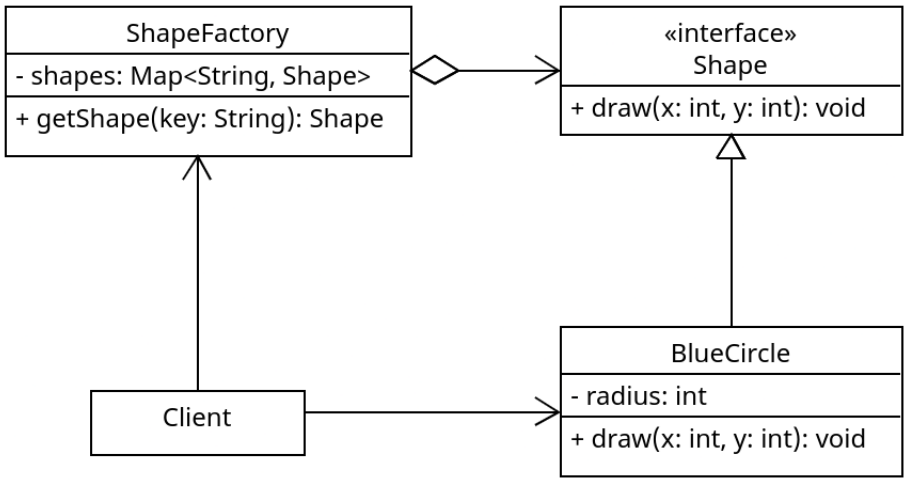
Motivácia: Majme situáciu, v ktorej program pracuje s veľkým množstvom podobných objektov, ktorými zapĺňa RAM, príkladom môže byť veľké množstvo grafických objektov na plátne. Tieto nezdieľané objekty vieme nahradiť malým množstvom zdieľaných objektov.

Štruktúra a UML:



Obr. 7: Štruktúra vzoru Flyweight

Príklad: Vykresľujeme objekty *BlueCircle* na plátno. Tieto objekty vytvára a spravuje *ShapeFactory*, v ktorej dochádza k prepoužívaniu už vytvorených inštancií. Klient nevytvára inštancie sám, ale musí si ich vyžiadať od *ShapeFactory*. Obrázok 8 zachytáva UML diagram tried tohto príkladu.



Obr. 8: Príklad použitia vzoru Flyweight

Implementácia príkladu:

Výpis 4: Vzorový príklad implementácie návrhového vzoru Flyweight

```
public interface Shape {
    void draw(int x, int y);
}

public class ShapeFactory {
    private Map<String, Shape> shapes;

    public ShapeFactory() {
        shapes = new HashMap<>();
    }

    public Shape getShape(String key) {
        Shape s = shapes.get(key);
        if (s == null) {
```

```
            s = new BlueCircle(50);
            shapes.put(key, s);
        }
        return s;
    }
}

public class BlueCircle {
    private int radius;
    public BlueCircle(int radius) { this.radius = radius; }
    public draw(int x, int y) { ... }
}

public class Main {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();
        BlueCircle circle = factory.getShape("circle");
    }
}
```

4 Softvér

Vytvorili sme jednoduchú aplikáciu na zobrazenie hierarchie predmetov a rozvrhu. Aplikácia inkorporuje všetky štyri prezentované návrhové vzory. Aplikácia má grafické rozhranie aj textové rozhranie.
Aplikácia používa build systém Maven a jazyk Java vo verzii 1.8. Zdrojové kódy aplikácie sú k dispozícii na `https://github.com/tj314/ASOS-prezentacia-2`.

4.1 Architektúra

Používateľ interaguje s aplikáciou pomocou grafického (GUI) alebo terminálového (TUI) rozhrania. Rozhrania sú implementované v triedach *GuiApplication*, respektíve *TuiApplication*. Obe tieto triedy dedia od abstraktnej triedy *AbstractApplication* a so systémom komunikujú pomocou vzoru Facade.

Návrhový vzor Facade je implementovaný pomocou triedy *Facade*, ktorá drží referencie na subsystémy, s ktorými komunikuje (subsystémy bližšie popíšeme v nasledovných podkapitolách). Subsystémy sú vytvorené pomocou triedy *FacadeFactory*. Pri vytvorení subsystémov sú nastavené príslušné údaje pre subsystémy. V našej implementácii nedochádza k načítaniu údajov z databázy, ani z online zdroja, subsystémy sa preto napĺňajú dummy hodnotami. *Facade* je zároveň implementovaná ako návrhový vzor Singleton.

4.1.1 Subsystém na reprezentáciu hierarchie predmetov

Prvý subsystém, s ktorým trieda *Facade* komunikuje, je subsystém na reprezentáciu hierarchie predmetov. Hierarchia predmetov je implementovaná použitím

návrhového vzoru Composite a trieda *Facade* si drží referenciu na koreň stromu tejto hierarchie. Prostredníctvom rozhrania *CourseListing* trieda *Facade* jednotne pristupuje k objektom tried *CourseDirectory* a *Course*. *CourseDirectory* predstavuje zložený objekt, ktorý v sebe ukladá viacero položiek typu *CourseListing* (buď *CourseDirectory*, alebo *Course*). Trieda *Course* reprezentuje samotný predmet.
Trieda *Course* je odekorovaná decorator-om *SeminarDecorator*. Pomocou *SeminarDecorator* v aplikácii rozlišujeme prednášky a cvičenia daného predmetu v rozvrhu, a to úpravou výpisu informácií triedy *Course* tak, aby obsahovala výpis "(SEMINAR)".

4.1.2 Subsystém na reprezentáciu rozvrhu

Druhý subsystém, s ktorým trieda *Facade* interaguje, slúži na reprezentáciu rozvrhu. Rozvrh je reprezentovaný triedou *Schedule*, ktorá spravuje informácie o tom, kedy a kým je daný predmet vyučovaný. Na správu týchto informácií využíva pomocné triedy *ScheduleData* a *TimeSlot*.

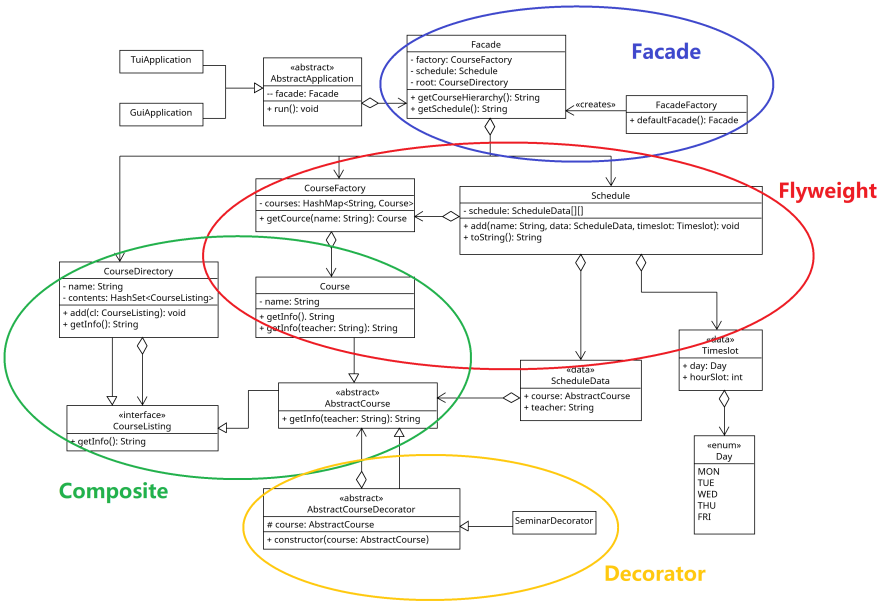
Trieda *Schedule* vytvára inštancie typu *Course* prostredníctvom triedy *CourseFactory*. Trieda *CourseFactory* si ukladá inštancie predmetov podľa ich názvov. Ak dostane požiadavku na vytvorenie predmetu s novým názvom, vytvorí novú inštanciu. Ak však daný názov pozná, volajúcemu vráti už existujúcu, príslušnú inštanciu. Táto trieda je súčasťou vzoru Flyweight.
Course má ako atribút uložený názov predmetu - svoj **intrinsický stav**. Pri výpise rozvrhu sa však ten istý predmet môže nachádzať na viacerých miestach, no s iným časom a vyučujúcim, t. j. líšiacim sa **extrinsickým stavom**. Trieda *Schedule* spravuje tieto nezdieľané informácie pre *Course*.

4.2 Použitie

Po spustení aplikácie sa používateľovi zobrazí hlavné okno. Na ľavej strane okna sa nachádza stromová štruktúra predmetov. Táto stromová štruktúra reprezentuje analógiu k stromovej štruktúre, ktorá sa nachádza v akademickom informačnom systéme. Počet zobrazených predmetov je však menší, kvôli prehľadnosti a názornosti príkladu. Na pravej strane okna sa nachádza samotný rozvrh, rozdelený po dňoch. Každý predmet je zobrazený vo formáte "názov-predmetu meno-vyučujúceho čas". Ak ide o cvičenie, naviac sa zobrazí aj "(SEMINAR)".
Ak používateľ nechce používať grafické rozhranie, môže použiť textové rozhranie, ktoré obsahuje identickú funkcionalitu. Prepnutie na textové rozhranie je možné iba cez kód použitím triedy *TuiApplication*. Príklad použitia je ilustrovaný v komentároch vo funkcii *main*.

4.3 Štruktúra softvéru

Na obrázku 9 je vidieť UML diagram tried pre náš softvér. V diagrame sú farebne oddelené triedy podľa príslušných návrhových vzorov, ktorých sú súčasťou.



Obr. 9: UML diagram tried riešenia so zvýraznením aplikovaných vzorov

4.4 Možné vylepšenia

Keďže táto aplikácia bola vytvorená ako ukážka použitia návrhových vzorov, nie je plne funkčná a teda má priestor pre možné vylepšenia:

- Pridanie funkcionality na načítavanie údajov z AIS (v súčasnej verzii sú predmety zadávané manuálne).
- Pridanie funkcionality na vyriešenie kolízií predmetov.
- Poskytnutie možnosti interakcie s používateľom (súčasná verzia slúži len na zobrazenie predmetov a rozvrhu)
- Prepínanie aplikácie do textového módu použitím používateľského rozhrania (súčasná verzia podporuje prepnutie úpravou kódu).

5 Záver

V tejto dokumentácii sme predstavili štrukturálne návrhové vzory Decorator, Composite, Flyweight a Facade. Ku každému sme uviedli názorné príklady. Implementovali sme aplikáciu, ktorá využíva všetky spomenuté vzory a podrobne sme ju popísali. Implementácia softvéru je k dispozícii na `https://github.com/tj314/ASOS-prezentacia-2`.

Referencie

[1] *Design Patterns*. 2. URL: https://sourcemaking.com/design_patterns.

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[3] *Structural Design Patterns*. 1. URL: https://refactoring.guru/design-patterns/structural-patterns.