10. Návrhové vzory

Obsah

- Návrhové vzory - základní pojmy
- Jedináček (Singleton, Multiton)
- Tovární metoda (Factory Method)
- Iterátor (Iterator)
- Událostmi řízené programování
- Vhodné použití a příklady

Návrhové vzory - základní pojmy

Návrhový vzor (design pattern) je obecné řešení opakujícího se problému při návrhu softwaru. Jedná se o popis nebo šablonu, jak řešit problém způsobem, který je použitelný v různých situacích.

Návrhové vzory obvykle rozdělujeme do tří základních kategorií: 1. **Tvořící vzory** (Creational) - řeší problémy s vytvářením objektů 2. **Strukturální vzory** (Structural) - řeší organizaci tříd a objektů 3. **Vzory chování** (Behavioral) - řeší komunikaci mezi objekty

Jedináček (Singleton)

Singleton je návrhový vzor zajišťující, že třída má pouze jednu instanci a poskytuje globální bod pro přístup k této instanci.

Vlastnosti singletonu:

- Garantuje existenci pouze jedné instance třídy
- Poskytuje globální přístupový bod k této instanci
- Zabraňuje vytvoření více instancí omezením konstruktoru

Implementace v Javě:

```
public class Singleton {
    // Statická instance třídy - jediná v celém programu
    private static Singleton instance;

    // Soukromý konstruktor zabraňuje vytvoření více instancí
    private Singleton() {
        // Inicializace
    }

    // Statická metoda pro získání instance (tzv. "lazy initialization")
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Metody singletonu
    public void doSomething() {
        // Nějaká funkčnost
    }
}
```

Thread-safe Singleton:

```
public class ThreadSafeSingleton {
    private static volatile ThreadSafeSingleton instance;

    private ThreadSafeSingleton() {}

    // Thread-safe implementace s dvojitým zamykáním (double-checked locking)
    public static ThreadSafeSingleton getInstance() {
        if (instance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if (instance == null) {
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
}
```

Enum Singleton (nejbezpečnější v Javě):

```
public enum EnumSingleton {
    INSTANCE;

    // Metody singletonu
    public void doSomething() {
        // Nějaká funkčnost
    }
}
```

Výhody Singletonu:

- Kontrolovaný přístup k jediné instanci
- Redukce množství alokované paměti
- Nahrazuje globální proměnné

Nevýhody Singletonu:

- Může porušovat princip jedné odpovědnosti (Single Responsibility Principle)
- Obtížnější testování (vytváří silnou závislost)
- Skrytý stav aplikace

Multiton

Multiton je rozšíření vzoru Singleton, které umožňuje mít omezený počet instancí třídy, kde každá instance je identifikována klíčem.

Implementace v Javě:

```
public class Multiton {
    // Mapa pro uchování instancí podle klíče
    private static final Map<String, Multiton> instances = new HashMap<>();

    // Soukromý konstruktor
    private Multiton() {}

    // Metoda pro získání instance podle klíče
    public static synchronized Multiton getInstance(String key) {
```

```
        if (!instances.containsKey(key)) {
            instances.put(key, new Multiton());
        }
        return instances.get(key);
    }

    // Metody multitonu
    public void doSomething() {
        // Nějaká funkčnost
    }
}
```

Výhody Multitonu:

- Omezuje počet instancí na předem definovanou množinu
- Poskytuje flexibilnější řízení instancí než Singleton

Nevýhody Multitonu:

- Větší složitost
- Stejně jako Singleton může komplikovat testování

Tovární metoda (Factory Method)

Tovární metoda je návrhový vzor, který poskytuje rozhraní pro vytváření objektů v nadtřídě, ale umožňuje podtřídám měnit typ objektů, které budou vytvořeny.

Struktura:

- **Produkt**: rozhraní pro objekty vytvářené tovární metodou
- **Konkrétní produkt**: implementace rozhraní Produkt
- **Tvůrce**: abstraktní třída deklarující tovární metodu
- **Konkrétní tvůrce**: implementace tovární metody pro vytváření konkrétních produktů

Implementace v Javě:

```
// Rozhraní produktu
interface Product {
    void operation();
}

// Konkrétní produkty
class ConcreteProductA implements Product {
    @Override
    public void operation() {
        System.out.println("Operace produktu A");
    }
}

class ConcreteProductB implements Product {
    @Override
    public void operation() {
        System.out.println("Operace produktu B");
    }
}

// Abstraktní tvůrce s tovární metodou
abstract class Creator {
```

```
    // Tovární metoda
    public abstract Product createProduct();

    // Metoda používající produkt
    public void someOperation() {
        // Vytvoření produktu pomocí tovární metody
        Product product = createProduct();
        // Použití produktu
        product.operation();
    }
}

// Konkrétní tvůrci
class ConcreteCreatorA extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}

class ConcreteCreatorB extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
    }
}
```

Výhody Tovární metody:

- Odděluje vytváření objektů od jejich používání
- Umožňuje přidávat nové typy produktů bez změny existujícího kódu
- Dodržuje princip otevřenosti/uzavřenosti (Open/Closed Principle)

Nevýhody Tovární metody:

- Může vést k velkému množství podtříd
- Větší složitost kódu

Iterátor (Iterator)

Iterátor je návrhový vzor, který poskytuje způsob, jak sekvenčně přistupovat k prvkům kolekce bez znalosti její vnitřní struktury.

Struktura:

- **Iterátor**: rozhraní definující metody pro procházení kolekce
- **Konkrétní iterátor**: implementace rozhraní Iterátor pro konkrétní kolekci
- **Kolekce**: rozhraní definující metodu pro vytvoření iterátoru
- **Konkrétní kolekce**: implementace kolekce, která vytváří iterátor

Implementace v Javě:

```
// Rozhraní iterátoru (standardní v Javě je java.util.Iterator)
interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

```java
// Rozhraní kolekce
interface Collection<T> {
    Iterator<T> createIterator();
}

// Konkrétní kolekce
class ConcreteCollection<T> implements Collection<T> {
    private List<T> items = new ArrayList<>();

    public void add(T item) {
        items.add(item);
    }

    @Override
    public Iterator<T> createIterator() {
        return new ConcreteIterator<>(this);
    }

    public List<T> getItems() {
        return items;
    }
}

// Konkrétní iterátor
class ConcreteIterator<T> implements Iterator<T> {
    private ConcreteCollection<T> collection;
    private int position = 0;

    public ConcreteIterator(ConcreteCollection<T> collection) {
        this.collection = collection;
    }

    @Override
    public boolean hasNext() {
        return position < collection.getItems().size();
    }

    @Override
    public T next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return collection.getItems().get(position++);
    }
}

// Příklad použití
public class IteratorDemo {
    public static void main(String[] args) {
        ConcreteCollection<String> collection = new ConcreteCollection<>();
        collection.add("První");
        collection.add("Druhý");
        collection.add("Třetí");

        Iterator<String> iterator = collection.createIterator();
        while (iterator.hasNext()) {
            String item = iterator.next();
            System.out.println(item);
```

```
        }
    }
}
```

Java Collection Framework

V Javě je vzor Iterátor součástí standardní knihovny (java.util.Iterator):

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class JavaIteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("První");
        list.add("Druhý");
        list.add("Třetí");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String item = iterator.next();
            System.out.println(item);
        }

        // Modernější způsob s for-each cyklem, který interně používá iterátor
        for (String item : list) {
            System.out.println(item);
        }
    }
}
```

Výhody Iterátoru:

- Skrývá vnitřní strukturu kolekce
- Poskytuje jednotný způsob procházení různých typů kolekcí
- Umožňuje paralelní procházení stejné kolekce

Nevýhody Iterátoru:

- Může zavést dodatečnou složitost
- V některých případech nemusí být efektivní jako přímý přístup

Událostmi řízené programování

Událostmi řízené programování (Event-driven programming) je programovací paradigma, ve kterém je běh programu určen událostmi (např. akce uživatele, zprávy od jiných programů nebo senzorů).

Základní koncepty:

1. **Události** (Events) - akce nebo výskyty detekované programem
2. **Posluchači událostí** (Event Listeners) - kód, který reaguje na určité události
3. **Zpracování událostí** (Event Handling) - mechanismus zpracování událostí posluchači
4. **Zdroje událostí** (Event Sources) - objekty generující události

Model Observer (Pozorovatel)

Observer je návrhový vzor často používaný v událostmi řízeném programování, který definuje závislost jednoho objektu na druhém tak, že když jeden objekt změní stav, všechny závislé objekty jsou automat-

icky informovány a aktualizovány.

Implementace v Javě:

```java
// Rozhraní subjektu (zdroj událostí)
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Rozhraní pozorovatele (posluchač událostí)
interface Observer {
    void update(String message);
}

// Konkrétní subjekt
class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String state;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }

    // Metoda měnící stav a vyvolávající oznámení
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}

// Konkrétní pozorovatel
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received message: " + message);
    }
}
```

```java
// Příklad použití
public class ObserverDemo {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();

        ConcreteObserver observer1 = new ConcreteObserver("Observer 1");
        ConcreteObserver observer2 = new ConcreteObserver("Observer 2");

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);

        subject.setState("Nová událost!");
    }
}
```

Java Event Model

V Javě je událostmi řízené programování implementováno především v GUI knihovnách:

```java
import java.awt.Button;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class EventDrivenExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Event Example");
        Button button = new Button("Click me");

        // Přidání posluchače událostí na tlačítko
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button was clicked!");
            }
        });

        // Modernější způsob s lambda výrazem
        button.addActionListener(e -> System.out.println("Button was clicked! (lambda)"));

        // Přidání posluchače pro zavření okna
        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Výhody událostmi řízeného programování:

- Oddělení zdrojů událostí od jejich zpracování
- Flexibilní architektura
- Přirozené pro uživatelská rozhraní
- Dobrá škálovatelnost a znovupoužitelnost kódu

Nevýhody událostmi řízeného programování:

- Složitější sledování toku programu
- Možné problémy s laděním
- Riziko "callback hell" (vnořené zpětné volání)

Vhodné použití a příklady

Singleton

Vhodné použití: - Správce konfigurace systému - Připojení k databázi (pool připojení) - Logger (záznamník událostí) - Přístup ke sdíleným zdrojům

Příklady v reálných aplikacích: - `java.lang.Runtime` - každá JVM má pouze jednu instanci Runtime - Správce připojení k databázi - Správce cache - Nastavení aplikace

Multiton

Vhodné použití: - Správa omezených zdrojů podle kategorie - Sdílení objektů s různými konfiguracemi

Příklady v reálných aplikacích: - Připojení k různým databázím podle identifikátoru - Jazykové mutace aplikace (jedna instance na jazyk) - Správce témat uživatelského rozhraní

Tovární metoda

Vhodné použití: - Kdy třída nemůže předvídat třídy objektů, které má vytvořit - Když třída chce, aby její podtřídy specifikovaly objekty, které vytváří - Když je vytváření objektů delegováno specializovaným pomocným třídám

Příklady v reálných aplikacích: - Framework pro vytváření UI komponent podle konfigurace - Parsery dokumentů různých formátů - Vytváření datových konektorů pro různé zdroje

Iterátor

Vhodné použití: - Procházení složitých datových struktur (stromy, grafy) - Poskytnutí jednotného rozhraní pro procházení různých kolekcí - Když chceme skrýt vnitřní implementaci kolekce

Příklady v reálných aplikacích: - Java Collection Framework (`java.util.Iterator`) - Procházení výsledků dotazu na databázi - Sekvenční přístup k elementům XML dokumentu - Lazy loading (načítání dat podle potřeby při procházení)

Událostmi řízené programování

Vhodné použití: - Uživatelská rozhraní - Asynchronní zpracování - Systémy, kde události přicházejí v nepředvídatelném pořadí - Komunikace mezi volně spojenými komponentami

Příklady v reálných aplikacích: - JavaFX, Swing, Android (UI frameworky) - Node.js (asynchronní server) - Systémy zpracování senzorových dat - Herní enginy - Webové aplikace (zachycení kliknutí uživatele)