NYÍREGYHÁZI EGYETEM
MATEMATIKA ÉS INFORMATIKA INTÉZET

Tervezési minták egy Objektum Orientált programozási nyelvben

Programozási technológiák

Beadandó dolgozat

Készítette: Hamvai Sándor
Nyíregyházi Egyetem
Matematika és Informatika Intézet
Programtervező informatikus BSC
szakos hallgató

Nyíregyháza, 2024. december 04.

1. Mi a Design Patterns?

A programtervezési minták (design patterns) olyan általános megoldások, amelyeket ismétlődő szoftvertervezési problémákra alkalmazunk. Ezek a minták lehetővé teszik, hogy a fejlesztők gyorsan és hatékonyan oldjanak meg problémákat, anélkül, hogy újra fel kellene találniuk a *„Spanyolviaszt”*. A programtervezési minták növelik a kód újrafelhasználhatóságát és fenntarthatóságát, mivel bevált gyakorlatokat használnak.

Három fő kategóriába sorolhatók:

- Creational patterns (létrehozási)
- Structural patterns (szerkezeti)
- Behavioral patterns (viselkedési)

A Creational patterns az objektumok létrehozását egyszerűsítik és egységesítik.
Ilyen például a Singleton, Factory Method, Prototype, Builder és Abstract Factory minta.
A Structural patterns az objektumok közötti kapcsolatok és struktúrák szervezésére fókuszálnak.
Ide tartozik az Adapter, Bridge, Proxy, Composite és Decorator minta is.
A Behavioral patterns az objektumok közötti kommunikációt és interakciókat határozzák meg. Példák erre a State, Template, Visitor, Chain of Responsibility, Command, Iterator és Observer minták.

A design patterns segítenek abban, hogy a kód olvashatóbb, karbantarthatóbb és bővíthetőbb legyen. Ezeket a mintákat tapasztalt fejlesztők dolgozták ki, és széles körben elfogadottak az iparágban. A megfelelő minta kiválasztása jelentősen növeli a szoftver minőségét és csökkenti a hibalehetőségeket. Összességében a programtervezési minták nélkülözhetetlen eszközök a szoftvertervezésben, amelyek segítenek a fejlesztőknek hatékonyan és kreatívan megoldani a komplex problémákat.

2. A Builder Design Pattern

A Builder minta egy létrehozási minta, amely célja az összetett objektumok létrehozásának egyszerűsítése és szabályozása. Ez a minta különösen hasznos, ha egy objektum több lépésben és különböző konfigurációkkal hozható létre. A Builder minta lehetővé teszi, hogy a létrehozási folyamat részleteit különválasszuk az objektum belső reprezentációjától. Ezzel elérhető, hogy ugyanazt az építési folyamatot használjuk különböző típusú objektumok létrehozására. A minta általában egy Builder osztályt definiál, amely metódusokat biztosít az objektum különböző részeinek létrehozására, valamint egy Director osztályt, amely irányítja az építési folyamatot.

Ennek köszönhetően rugalmasan konfigurálható a végeredmény anélkül, hogy a kód bonyolultsága megnövekedne. A Builder minta előnye, hogy a létrehozási folyamat jól strukturálható és karbantartható, különösen akkor, ha sok különböző konfigurációra van szükség. Ez a minta különösen hasznos a szoftverfejlesztésben, ahol a komplex objektumokat különböző módon kell

előállítani és kezelni. A Builder minta alkalmazása növeli a kód újrafelhasználhatóságát és csökkenti a hibák előfordulásának valószínűségét.

Rövid leírás: A következő kód a Builder minta segítségével mutatja be egy autó létrehozásának folyamatát különböző jellemzőkkel (motor, sebességváltó, ülések száma, szín). A példában is látható, hogy ez a minta lehetővé teszi az összetett objektumok létrehozását több lépésben és konfigurálható módon.

```
public class Car {
    private String engine;
    private String transmission;
    private int seats;
    private String color;

     public static class Builder {
        private String engine;
        private String transmission;
        private int seats;
        private String color;

        public Builder setEngine(String engine) {
            this.engine = engine;
            return this;
        }

        public Builder setTransmission(String transmission) {
            this.transmission = transmission;
            return this;
        }

        public Builder setSeats(int seats) {
            this.seats = seats;
            return this;
        }

        public Builder setColor(String color) {
            this.color = color;
            return this;
        }

        public Car build() {
            return new Car(this);
        }
    }

    private Car(Builder builder) {
        this.engine = builder.engine;
        this.transmission = builder.transmission;
        this.seats = builder.seats;
        this.color = builder.color;
    }

    @Override
    public String toString() {
        return "Car with " + engine + " engine, " + transmission + " transmission, " +
                seats + " seats, and color " + color;
    }
}

public class BuilderPatternDemo {
    public static void main(String[] args) {
        Car car = new Car.Builder()
                .setEngine("V8")
                .setTransmission("Automatic")
```

```
            .setSeats(4)
            .setColor("Red")
            .build();
        System.out.println(car);
    }
}
```

Magyarázat:

- **Car osztály:** Ez az osztály egy autót reprezentál, amely különböző jellemzőkkel rendelkezik, mint például a motor típusa, a sebességváltó típusa, az ülések száma és a szín.
- **Builder belső osztály:** Ez az osztály biztosítja azokat a metódusokat (setEngine, setTransmission, setSeats, setColor), amelyekkel az autó különböző részeit konfigurálhatjuk. Minden metódus visszaadja a Builder objektumot (return this), így láncolt hívásokkal konfigurálhatjuk az autót.
- **build metódus:** Ez a metódus létrehozza a végleges Car objektumot a Builder objektum alapján.
- **BuilderPatternDemo osztály:** Ebben az osztályban hozzuk létre és konfiguráljuk a Car objektumot a Builder minta használatával. A main metódusban egy Car objektumot hozunk létre, beállítva a motor típusát, a sebességváltót, az ülések számát és a színt, majd kiírjuk az autó tulajdonságait.

3. A Bridge Design Pattern

A Bridge minta egy szerkezeti minta, amely célja az absztrakció elválasztása a megvalósítástól, így lehetővé téve mindkettő független módosítását. Ezzel a mintával elérhető, hogy a rendszert úgy bővítsük új funkciókkal, hogy a meglévő kódot nem kell módosítani. A Bridge minta különösen hasznos olyan helyzetekben, ahol az osztály hierarchiája bonyolult és a változások gyakoriak.

A minta két fő részből áll: az Abstraction és az Implementor interfészekből.

Az Abstraction osztály definiálja az absztrakciós réteget, míg az Implementor interfész a megvalósítási réteget. A konkrét osztályok (ConcreteImplementor) valósítják meg az Implementor interfészt, így a megvalósítás független az absztrakciótól.

Például képzeljük el, hogy egy grafikus rendszerben különböző formákat (például kör, négyzet) kell megjeleníteni különböző platformokon (például Windows, Linux). A Bridge minta lehetővé teszi, hogy a formák és a platformok külön osztályokban legyenek definiálva, így mindkettő függetlenül módosítható. Ezzel a megközelítéssel elérhető, hogy új formákat vagy platformokat vezessünk be anélkül, hogy a meglévő kódot módosítani kellene.

A Bridge minta alkalmazása növeli a rendszer rugalmasságát és csökkenti a kód duplikációját. Ezenkívül a minta segít az objektumok közötti kapcsolatok és interakciók tiszta és jól szervezett kialakításában.

Rövid leírás: Ez a kód a Bridge minta használatát mutatja be egy üzenetküldési rendszerben, ahol különböző csatornákon (Email és SMS) küldünk üzeneteket. A minta elválasztja az absztrakciót (Message osztály) a megvalósítástól (MessageSender interfész), így mindkettő függetlenül módosítható.

```
// Abstraction
public abstract class Message {
    protected MessageSender sender;

    public Message(MessageSender sender) {
        this.sender = sender;
    }

    public abstract void send(String message);
}

// Implementor
public interface MessageSender {
    void sendMessage(String message);
}

// ConcreteImplementor A
public class EmailSender implements MessageSender {
    public void sendMessage(String message) {
        System.out.println("Sending Email with message: " + message);
    }
}

// ConcreteImplementor B
public class SMSSender implements MessageSender {
    public void sendMessage(String message) {
        System.out.println("Sending SMS with message: " + message);
    }
}

// RefinedAbstraction
public class TextMessage extends Message {
    public TextMessage(MessageSender sender) {
        super(sender);
    }

    public void send(String message) {
        sender.sendMessage(message);
    }
}

public class BridgePatternDemo {
    public static void main(String[] args) {
        Message emailMessage = new TextMessage(new EmailSender());
        emailMessage.send("Hello via Email!");

        Message smsMessage = new TextMessage(new SMSSender());
        smsMessage.send("Hello via SMS!");
    }
}
```

Magyarázat:

- **Message (Abstraction) osztály:** Ez az absztrakciós réteg, amely definiálja a send metódust, és tartalmaz egy hivatkozást a MessageSender (Implementor) interfészre.

- **MessageSender (Implementor) interfész:** Ez az interfész definiálja az üzenetküldési metódust, amelyet a konkrét implementációk valósítanak meg.
- **EmailSender és SMSSender (ConcreteImplementor):** Ezek a konkrét implementációk, amelyek különböző csatornákon (Email és SMS) valósítják meg az üzenetküldési folyamatot.
- **TextMessage (RefinedAbstraction) osztály:** Ez az osztály kiterjeszti a Message osztályt, és implementálja a send metódust.
- **BridgePatternDemo osztály:** Ebben az osztályban demonstráljuk a Bridge minta használatát különböző csatornákon (Email és SMS) történő üzenetküldésre.

4. Az Iterator Design Pattern

Az Iterator minta egy viselkedési minta, amely célja az, hogy egy kollekció elemein való bejárást egységes és egyszerű módon valósítsa meg anélkül, hogy a kollekció belső struktúráját ki kellene tenni. Ez a minta lehetővé teszi, hogy különböző kollekciókat egységes módon kezeljünk és járjunk be, növelve ezzel a kód rugalmasságát és karbantarthatóságát.

Az Iterator minta általában egy Iterator interfészt és egy ConcreteIterator osztályt definiál, amelyek biztosítják az elemek közötti lépkedést. Ezen kívül a minta tartalmaz egy Aggregate interfészt és egy ConcreteAggregate osztályt, amelyek a kollekciókat reprezentálják. Az Iterator interfész metódusai általában tartalmazzák a következő elemre lépkedést (next), az aktuális elem elérését (current) és a kollekció végének ellenőrzését (hasNext).

Az Iterator minta előnye, hogy a kollekciók elemein való bejárás jól strukturált és könnyen karbantartható. Ez a minta különösen hasznos olyan rendszerekben, ahol a kollekciók gyakran változnak vagy különböző típusúak. Összességében az Iterator minta egy hatékony és rugalmas módja a kollekciók elemein való bejárásnak, amely növeli a kód olvashatóságát is.

Rövid leírás: Ez a kód az Iterator minta használatát mutatja be egy játékkártya pakli elemeinek bejárására. Lehetővé teszi a kollekció elemeinek egységes és egyszerű bejárását anélkül, hogy a kollekció belső struktúráját felfednénk.

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

// ConcreteAggregate
public class CardDeck implements Iterable<String> {
    private List<String> cards = new ArrayList<>();

    public void addCard(String card) {
        cards.add(card);
    }

    @Override
    public Iterator<String> iterator() {
        return new CardIterator();
```

```
    }

    // ConcreteIterator
    private class CardIterator implements Iterator<String> {
        private int index = 0;

        @Override
        public boolean hasNext() {
            return index < cards.size();
        }

        @Override
        public String next() {
            if (hasNext()) {
                return cards.get(index++);
            }
            return null;
        }
    }
}

public class IteratorPatternDemo {
    public static void main(String[] args) {
        CardDeck cardDeck = new CardDeck();
        cardDeck.addCard("Ace of Spades");
        cardDeck.addCard("King of Hearts");
        cardDeck.addCard("Queen of Clubs");

        for (String card : cardDeck) {
            System.out.println(card);
        }
    }
}
```

Magyarázat:

- **CardDeck (ConcreteAggregate) osztály:** Ez az osztály egy kártyapaklit reprezentál és implementálja az Iterable interfészt, lehetővé téve a kártyák bejárását.
- **addCard metódus:** Ezzel a metódussal kártyákat adhatunk hozzá a paklihoz.
- **iterator metódus:** Ez a metódus visszaadja a CardIterator példányát, amely végrehajtja az iterációt.
- **CardIterator (ConcreteIterator) belső osztály:** Ez az osztály valósítja meg az Iterator interfészt és kezeli a bejárás logikáját.
- **IteratorPatternDemo osztály:** Ebben az osztályban demonstráljuk az Iterator minta használatát a CardDeck példáján keresztül.

BEFEJEZÉS

Tehát amint láthattuk a korábbi példákban a tervezési minták tipikus megoldást jelentenek a szoftverek tervezésében gyakran felmerülő problémákra. Olyanok, mint a tervrajzok, melyeket testreszabhatunk a részben különböző, de ismétlődő tervezési problémák megoldására.

Nem elég megtalálni egy mintát, és bemásolni azt a programunkba, ahogy azt a polcról levehető kész komponensekkel vagy program könyvtárakkal szokás tenni.

A minta nem egy konkrét kódrészlet, hanem egy általános koncepció egy adott probléma megoldására.

A mintát követve készíthetünk egy olyan megoldást, amely megfelel a saját programunk sajátosságainak.