PROGRAMTERVEZÉSI MINTÁK KIDOLGOZVA

Mik a programtervezési minták?

A tervezési minták tipikus megoldások a szoftvertervezésben gyakran előforduló problémákra. Olyanok, mint az előre elkészített tervrajzok, amelyeket a programozó testreszabhat a kódban visszatérő tervezési problémák megoldása érdekében.

1. Bridge

Összetevők:

1. **Abstraction (Absztrakció)**: Az absztrakt osztály vagy interfész, amely definiálja a magas szintű működést.
2. **RefinedAbstraction (Finomított Absztrakció)**: Az Abstraction leszármazottja, amely további specifikus funkciókat nyújt.
3. **Implementor (Implementáció)**: Egy interfész, amely definiálja a funkciókat, amelyeket az absztrakció használ.
4. **ConcreteImplementor (Konkrét Implementáció)**: Az Implementor interfész konkrét megvalósítása.

Előnyök:

- Elkülöníti az absztrakciót az implementációtól.
- Könnyebb karbantartás és bővítés.
- Az absztrakció és az implementáció külön-külön változtatható.

Hátrányok:

- Bonyolultabb struktúrát eredményezhet.
- Az osztályok közötti kapcsolatokat nehezebb követni.

```java
package mintak;

interface Szin {
    void feltolt();
}

class Piros implements Szin {
    public void feltolt() {
        System.out.println("Feltöltés Piros színnel");
    }
}

class Kek implements Szin {
    public void feltolt() {
        System.out.println("Feltöltés Kék színnel");
    }
}


abstract class Alakzat {
    protected Szin szin;

    public Alakzat(Szin szin) {
        this.szin = szin;
    }

    abstract void rajzol();
}

class Kor extends Alakzat {
    public Kor(Szin szin) {
        super(szin);
    }

    public void rajzol() {
        System.out.print("Kör rajzolása");
        szin.feltolt();
    }
}

class Negyzet extends Alakzat {
    public Negyzet(Szin szin) {
        super(szin);
    }

    public void rajzol() {
        System.out.print("Négyzet rajzolása");
        szin.feltolt();
    }
}


public class Bridge {
    public static void main(String[] args) {
        Alakzat pirosKor = new Kor(new Piros());
        pirosKor.rajzol();

        Alakzat kekNegyzet = new Negyzet(new Kek());
        kekNegyzet.rajzol();
    }
}
```

2. Builder

Összetevők:

1. **Product (Termék)**: Az összetett objektum, amelyet a Builder állít össze.
2. **Builder**: Egy interfész, amely meghatározza a termék részeinek létrehozásához szükséges lépéseket.
3. **ConcreteBuilder (Konkrét Builder)**: A Builder interfész implementációja, amely a konkrét termék részeit hozza létre.
4. **Director (Igazgató)**: Koordinálja a Builder metódusait a termék felépítéséhez.

Előnyök:

- Komplex objektumokat könnyen létrehozhat.
- Tisztán elkülöníti az objektum építését és annak megvalósítását.
- Könnyen olvasható és módosítható kód.

Hátrányok:

- Túlkomplikálttá válhat, ha az objektum egyszerű.
- Sok osztályt hozhat létre.

```java
package mintak;

class Kocsi {
    private String model;
    private String szin;
    private int gyartasiEv;

    public Kocsi(String model, String szin, int ev) {
        this.model = model;
        this.szin = szin;
        this.gyartasiEv = ev;
    }

    @Override
    public String toString() {
        return "Kocsi [Model=" + model + ", Szín=" + szin + ",
Gyártás éve=" + gyartasiEv + "]";
    }
}


class KocsiBuilder {
    private String model;
    private String szin;
    private int gyartasiEv;

    public KocsiBuilder setModel(String model) {
        this.model = model;
        return this;
    }

    public KocsiBuilder setSzin(String szin) {
        this.szin = szin;
        return this;
    }

    public KocsiBuilder setGyartasiEv(int gyartasiEv) {
        this.gyartasiEv = gyartasiEv;
        return this;
    }

    public Kocsi build() {
        return new Kocsi(model, szin, gyartasiEv);
    }
}

public class Builder {
    public static void main(String[] args) {
        Kocsi car = new
KocsiBuilder().setModel("Ford").setSzin("Fekete").setGyartasiEv(2024)
.build();
        System.out.println(car);
    }
}
```

3. Factory

Összetevők:

1. **Product (Termék)**: Az interfész vagy absztrakt osztály, amely meghatározza a létrehozandó objektumok közös működését.
2. **ConcreteProduct (Konkrét Termék)**: Az interfész vagy absztrakt osztály konkrét implementációja.
3. **Creator (Gyártó)**: Az interfész vagy absztrakt osztály, amely tartalmazza a createProduct metódust.
4. **ConcreteCreator (Konkrét Gyártó)**: Egy implementáció, amely meghatározza, hogy milyen konkrét terméket hozzon létre.

Előnyök:

- Egyszerűsíti az objektumok létrehozását.
- A gyártási logika elkülöníthető a kliens kódtól.
- Könnyen bővíthető új típusokkal.

Hátrányok:

- Több osztály hozzáadását követeli meg.
- Bonyolultabbá válhat, ha sok konkrét gyártó van

```java
package mintak;

interface Allat {
    void hangKiadas();
}

class Kutya implements Allat {
    public void hangKiadas() {
        System.out.println("Vau!");
    }
}

class Macska implements Allat {
    public void hangKiadas() {
        System.out.println("Meow!");
    }
}

abstract class AllatFactory {
    public abstract Allat letrehoz();
}

class KutyaFactory extends AllatFactory {
    public Allat letrehoz() {
        return new Kutya();
    }
}

class MacskaFactory extends AllatFactory {
    public Allat letrehoz() {
        return new Macska();
    }
}

public class Factory {
    public static void main(String[] args) {
        AllatFactory kutyak = new KutyaFactory();
        Allat kutya = kutyak.letrehoz();
        kutya.hangKiadas();

        AllatFactory macskak = new MacskaFactory();
        Allat macska = macskak.letrehoz();
        macska.hangKiadas();
    }
}
```

4. Iterator

Összetevők:

1. **Iterator**: Az interfész, amely meghatározza a bejárás metódusait (például hasNext, next).
2. **ConcreteIterator (Konkrét Iterátor)**: Az Iterator interfész implementációja, amely a konkrét kollekciót járja be.
3. **Aggregate (Kollekció)**: Az interfész, amely definiálja az iterátor példányának visszaadását.
4. **ConcreteAggregate (Konkrét Kollekció)**: Az Aggregate interfész implementációja, amely konkrét adatokat tartalmaz.

Előnyök:

- Kollekciót be lehet járni anélkül, hogy tudnánk annak belső működését.
- Egyszerűsíti a kollekció kezelését.
- Többféle iterációs módszert támogathat.

Hátrányok:

- Nem minden kollekciós típus esetében egyszerű implementálni.
- Bonyolultabbá válhat, ha többféle iterációs logika szükséges.

```java
package mintak;

import java.util.List;
import java.util.ArrayList;

interface Iterator {
    boolean hasNext();
    Object next();
}

class KocsIterator implements Iterator {
    private List<String> kocsik;
    private int position = 0;

    public KocsIterator(List<String> kocsik) {
        this.kocsik = kocsik;
    }

    public boolean hasNext() {
        return position < kocsik.size();
    }

    public Object next() {
        return hasNext() ? kocsik.get(position++) : null;
    }
}


class KocsiCollection {
    private List<String> kocsik = new ArrayList<>();

    public void hozzaAd(String kocsi) {
        kocsik.add(kocsi);
    }

    public Iterator getIterator() {
        return new KocsIterator(kocsik);
    }
}

public class IteratorMinta {
    public static void main(String[] args) {
        KocsiCollection collection = new KocsiCollection();
        collection.hozzaAd("Tesla");
        collection.hozzaAd("BMW");
        collection.hozzaAd("Audi");

        Iterator iterator = collection.getIterator();
        while (iterator.hasNext()) {
            System.out.println("Kocsi: " + iterator.next());
        }
    }
}
```

5. Prototype

Összetevők:

1. **Prototype**: Az interfész, amely meghatározza a klónozás metódusát.
2. **ConcretePrototype (Konkrét Prototype)**: A Prototype interfész implementációja, amely az objektum klónozását végzi.

Előnyök:

- Gyorsabb objektum létrehozás másolás révén.
- Bonyolult objektumok inicializációja elkerülhető.
- Könnyen testre szabható.

Hátrányok:

- A mély másolás megvalósítása bonyolult lehet.
- Ha az objektum struktúrája megváltozik, a klónozási logikát is frissíteni kell.

```java
package mintak;

interface Prototype {
    Prototype clone();
}

class Jarmu implements Prototype {
    private String model;

    public Jarmu(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }

    @Override
    public Prototype clone() {
        return new Jarmu(this.model);
    }

    @Override
    public String toString() {
        return "Jarmu [Model=" + model + "]";
    }
}

public class PrototypeMinta {
    public static void main(String[] args) {
        Jarmu eredeti = new Jarmu("Tesla Model S");
        Jarmu lemasolt = (Jarmu) eredeti.clone();

        System.out.println(eredeti);
        System.out.println(lemasolt);
    }
}
```