Tervezési minták egy OO programozási nyelvben. MVC, mint modell-nézet-vezérlő minta és néhány másik tervezési minta

Programtervezési minta

Az informatikában programtervezési mintának (Software Design Patterns) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.

A tervminták nem kész tervek, amiket közvetlenül le lehet kódolni. Céljuk az, hogy leírást vagy sablont nyújtsanak és segítik formalizálni a megoldást.

A minták rendszerint osztályok és objektumok közötti kapcsolatokat mutatnak, de nem specifikálják konkrétan a végleges osztályokat vagy objektumokat. A modellek absztrakt osztályai helyett egyes esetekben interfészek is használhatók, de azokat maga a tervminta nem mutatja. Egyes nyelvek beépítetten tartalmaznak tervmintákat. A tervminták tekinthetők a strukturált programozás egyik szintjének a paradigma és az algoritmus között.

A legtöbb tervminta objektumorientált környezetre van kidolgozva. Az objektumorientált minták közül nem mindegyiket lehet és nem mindegyiket érdemes itt használni, van, amit módosítani kell.

Fogalma

A programtervezési minták fogalma Christopher Alexander építész ötlete nyomán született meg. Ő volt az, aki olyan, az építészetben újra és újra felbukkanó mintákat keresett, amelyek a jól megépített házakat jellemzik. Könyvében, a „The Timeless Way of Buliding”-ben olyan mintákat próbált leírni, amelyek segítségével akár egy kezdő építész is gyorsan jó épületeket tervezhet.

1987-ben Kent Beck és Ward Cunningham elkezdett kísérletezni hasonló minták alkalmazásával a programozásban, speciális mintanyelvek segítségével. A következő években kutatásaikhoz mások is társultak.

A programtervezési minták a '90-es évek elején (1994) tettek szert népszerűségre, amikor a négyek bandájaként vagy gammáékként ("Gang of Four" vagy GoF)-ként emlegetett Erich Gamma, Richard Helm, Ralph Johnson és John Vlissides programozó négyes kiadta a Programtervezési minták című könyvüket, amely ma is alapjául szolgál az objektumorientált programozási minták kutatásának. Magát a tervmintát nem definiálták. A fogalom maga évekig formalizálatlan maradt.

Ez a könyv összesen 23 mintát mutat be, és a következő kategóriákba sorolja őket:

- létrehozási minta
- szerkezeti minta

- viselkedési minta

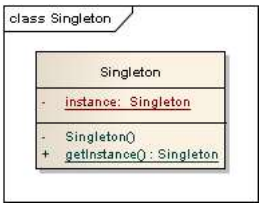
A programtervezési minták a GoF definíciója szerint: „egymással együttműködő objektumok és osztályok leírásai, amelyek testre szabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben”. A szoftvertervezésben egy-egy problémára végtelen sok különböző megoldás adható, azonban ezek között kevés optimális van. Tapasztalt szoftvertervezők, akik már sok hasonló problémával találkoztak, könnyen előhúzhatnak egy olyan megoldást, amely már kipróbált és bizonyított. Kezdő fejlesztőknek viszont jól jön, ha mindazt a tudást és tapasztalatot, amit csak évek munkájával érhetnek el, precízen dokumentálva kézbe vehetik, tanulhatnak belőle és az általa bevezetett kifejezésekkel könnyebben beszélhetik meg egymás között az ötleteiket. A programtervezési minták ilyen összegyűjtött tapasztalatok, amelyek mindegyike egy-egy gyakran előforduló problémára ad általánosított választ. Azonban mindezek mellett még számos előnyük van:

- lerövidítik a tapasztalatszerzési időt
- lerövidítik a tervezési időt
- közös szótárat ad a fejlesztők kezébe
- magasabb szintű programozást tesz lehetővé

1. Létrehozási minták

a. Egyke (Singleton)

Az egyke minta akkor használatos, ha garantálnunk kell, hogy egy objektumpéldányból egy időben egy JVM-ben csak egy létezzen, amelyet alkalmazásunk objektumai közösen használnak. Ez az adott osztály korlátozott példányosításával biztosítható. Az egyke konstruktora az osztályon kívülről nem látható, az egyetlen példány elérésére nyilvános metódus(ok) szolgál(nak).



Az egyke minta megvalósításának osztálydiagramja

A privát láthatóságú `instance` osztályszintű adattag tárolja a példányt, amelyet a nyilvános `getInstance` metódussal kérdezhetünk le.

Példa:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class ConnectionHelper {

    private static final ConnectionHelper
     connectionHelper = new ConnectionHelper();

    private ConnectionHelper() {
    }

    public static ConnectionHelper getInstance() {
        return connectionHelper;
    }

    private Connection conn;
    private String username;
    private String password;

    private String hostname = "servername.inf.unideb.hu";
    private int port = 1521;
    private String sid = "ORCL";

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setHostname(String hostname) {
        this.hostname = hostname;
    }

    public void setPort(int port) {
        this.port = port;
    }

    public void setSid(String sid) {
        this.sid = sid;
    }

    public Connection getConnection() throws
     SQLException, MissingCredentialsException {
        if (conn == null) {
            synchronized (connectionHelper) {
                if (conn == null) {
                    if (   username == null ||
     username.isEmpty()
                        || password == null ||
     password.isEmpty())
                        throw new
     MissingCredentialsException("Missing credentials.");
                    conn =
```

```
     DriverManager.getConnection(new
     StringBuilder("jdbc:oracle:thin:@")
                                     .append(hostname).appe
nd(':')
                                     .append(String.valueOf
(port)).append(':')
                                     .append(sid).toString(
),
                                      username, password);
                }
            }
        }

        return conn;
    }

    public void closeConnection() {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                // Kivétel kezelése
            } finally {
                conn = null;
            }
        }
    }
}
```

Itt a `getInstance` metódusa singleton egy példányát adja vissza, amely bezárja a kapcsolódási adatok megadására szolgáló `set`... metódusokat, valamint a kapcsolat kiépítését lusta inicializációval (lazy init) elvégző `getConnection` metódust, amely, ha már létrejött a globálisan elérhető kapcsolat, azt adja vissza, egyébként pedig létrehozza azt, és úgy adja vissza. A `closeConnection` a globális hozzáférési ponttal rendelkező kapcsolat bezárására szolgál.

Megjegyzés:

A példában a singleton példányt hagyományos módon, míg az általa bezárt `Connection` objektumot lusta inicializációval hoztuk létre.

A fenti példa a singleton klasszikus implementációs megoldása, azonban az 1.5-ös feletti Java verziókban javasolt egy egyelemű felsorolásos típus (`enum`) használata a singleton megvalósítása érdekében.

Példa:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public enum ConnectionHelper {
    INSTANCE;

    private Connection conn;

    private ConnectionHelper() {
    }

    private String username;
    private String password;
    private String hostname = "servername.inf.unideb.hu";
    private int port = 1521;
    private String sid = "ORCL";

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setHostname(String hostname) {
        this.hostname = hostname;
    }

    public void setPort(int port) {
        this.port = port;
    }

    public void setSid(String sid) {
        this.sid = sid;
    }

    public Connection getConnection() throws
     SQLException, MissingCredentialsException {
        if (conn == null) {
            if (   username == null || username.isEmpty()
                || password == null ||
     password.isEmpty())
                throw new
     MissingCredentialsException(Missing credentials.");
            conn = DriverManager.getConnection(new
     StringBuilder("jdbc:oracle:thin:@")

                    .append(hostname).append(':')

     .append(String.valueOf(port)).append(':')

     .append(sid).toString(),
                            username, password);
        }
```

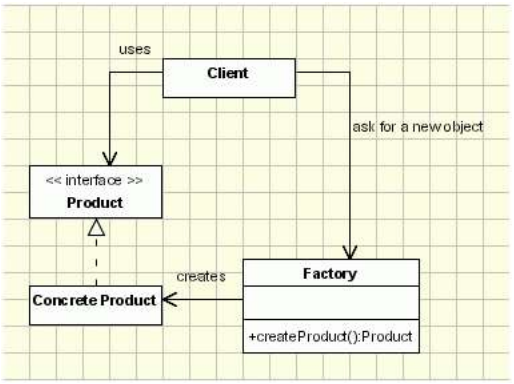
```
        return conn;
    }

    public void closeConnection() {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                // Kivétel kezelése
            } finally {
                conn = null;
            }
        }
    }
}
```

A hívás helyén pedig az alábbi módon használhatjuk fel az `enum` példányt:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConnMain {
     public static void main(String[] args) throws
     SQLException, MissingCredentialsException {

     ConnectionHelper.INSTANCE.setUsername("username");

     ConnectionHelper.INSTANCE.setPassword("password");
          Connection conn =
     ConnectionHelper.INSTANCE.getConnection();
             // Műveletek
          ConnectionHelper.INSTANCE.closeConnection();
     }
}
```

Gyártó minták

A gyártó minták úgy készítenek új objektumokat, hogy közben a létrehozás logikáját elrejtik a kliens elől, amely az újonnan létrehozott példányt egy interfészen keresztül érheti el.

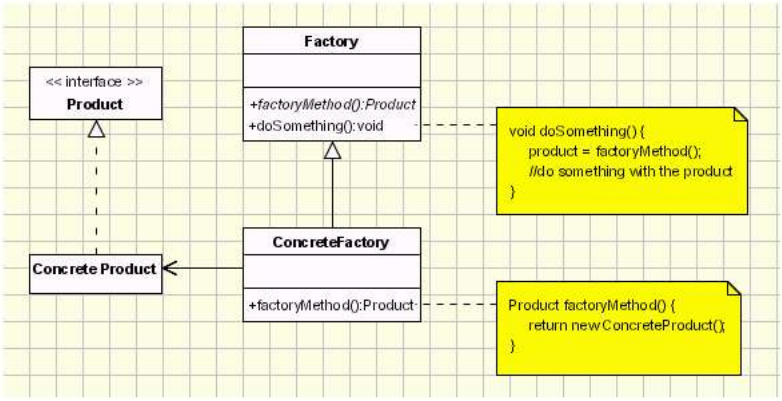


A gyártó minták általános megvalósításának osztálydiagramja

A kliens által igényelt objektum a `new` operátor alkalmazása helyett a megadott paraméterek alapján a gyárban készül. A kliens nincs tisztában a létrehozás és az objektum konkrét implementációjával.

b. Gyártófüggvény (Factory method)

Ezt a tervezési mintát akkor használjuk, rendelkezünk egy szülőosztállyal, amelynek több leszármazottja van, és a bemeneten múlik, hogy mely gyermekosztályra van szükségünk. A minta egy gyártó metódusra teszi az objektumok létrehozásának felelősségét.



A gyártófüggvény minta megvalósításának osztálydiagramja

A `Product` interfész a gyártó metódus által előállított objektum típusa, amelyet a `ConcreteProduct` osztály implementál. A `Factory` osztály definiálja a gyártó metódust, amely egy `Product` objektummal tér vissza. A `ConcreteFactory` osztály implementálja a metódust, amely a konkrét objektumot állítja elő.

Példák:

A `String` osztály `valueOf` metódusa, amely gyártófüggvényként legyártja a paraméterének megfelelő csomagolóosztálybeli objektumot.

A `getInstance` metódusok (sokszor statikusak) legyártanak egy-egy megfelelő objektumpéldányt (erre láttunk példát a singleton esetében is, de a szabványos Java API-ban előforduló statikus gyártófüggvényekre is van példa):

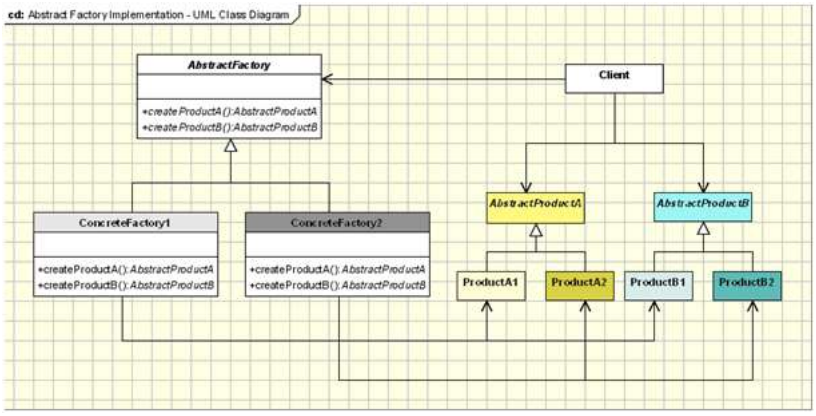
```
Timestamp currentTimestamp = new
     Timestamp(GregorianCalendar.getInstance().getTimeInM
illis());
```

```
NumberFormat nf = NumberFormat.getInstance();
colValues.add(nf.format(book.getPrice()));
```

```
Locale.setDefault(new Locale("hu", "HU"));
ResourceBundle rBundle =
     ResourceBundle.getBundle(bundleName);
```

c. Elvont gyár (Abstract factory)

Az elvont gyár minta hasonló a gyártófüggvényhez, azonban magasabb absztrakciós szintet képvisel. A gyártók gyárának is szokták nevezni. Ebben a mintában nem `if-else` vagy `catch` blokkokkal határozzuk meg, hogy mely termékalosztályt példányosítjuk, hanem minden termékalosztályhoz tartozik egy gyár osztály, amely segítségével az absztrakt gyár elkészíti az objektumot. Absztrakt gyár alkalmazásával elrejthetjük a platformfüggő osztályokat a kliens elől.



Az elvont gyár minta megvalósításának osztálydiagramja

Az `AbstractFactory` osztály absztrakt objektumok előállítására tartalmaz metódusokat, amelyeket a `ConcreteFactory` osztályok definiálnak felül a konkrét objektumok előállítása érdekében. Az `AbstractProduct` típusok a `Product` konkrét típusok közös műveleteit és tulajdonságait írják le. A konkrét `Product` objektumokat a konkrét gyárak állítják elő, a kliens azonban az absztrakt gyárat és az absztrakt terméket használja, így teljesen rejtve maradnak előle a konkrét

implementációk.

Példák (itt az első sorok rendre statikus gyártófüggvényre példák, legyártanak egy absztrakt gyárat, míg az absztrakt gyár segítségével gyártjuk le a megfelelő implementáció XML-feldolgozást segítő objektumát):

```
DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder =
     factory.newDocumentBuilder();
```

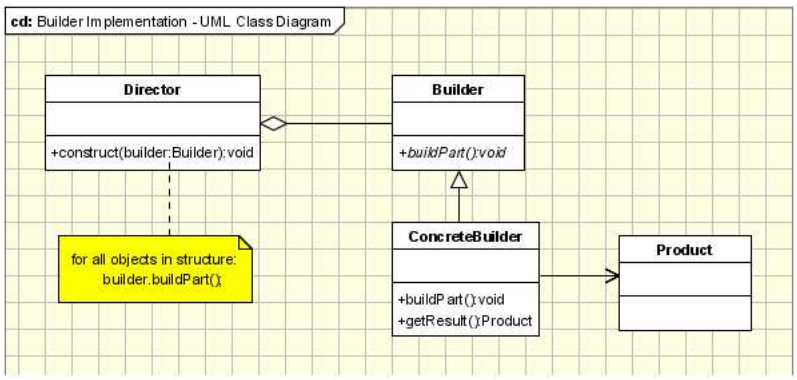
```
TransformerFactory transformerFactory =
     TransformerFactory.newInstance();
Transformer transformer =
     transformerFactory.newTransformer();
```

```
SAXParserFactory factory =
     SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
```

```
XMLInputFactory inputFactory =
     XMLInputFactory.newInstance();
InputStream in = new FileInputStream(xmlFile);
XMLEventReader eventReader =
     inputFactory.createXMLEventReader(in);
```

d. Építő (Builder)

Ez a minta szintén objektumok előállítására szolgál, viszont egyben megoldást is kínál a gyár mintákkal kapcsolatos problémákra. A gyár minták nehezen boldogulnak a sok attribútummal rendelkező objektumokkal. Az építő azonban lépésről lépésre építi fel az objektumot, állítja be az attribútumok értékét, míg végül visszaadja a teljesen elkészült példányt.



Az építő minta megvalósításának osztálydiagramja

A `Builder` osztály egy absztrakt felületet határoz meg, amely egy objektumot épít

fel annak részeiből. A `ConcreteBuilder`, amely implementálja a `Builder` absztrakt metódusát, összeállítja az objektumot annak részeiből, az elkészült példányt pedig eltárolja. A végterméket a `getResult` metódussal kérdezhetjük le. A `Builder` interfész segítségével a `Director` osztály hozza létre az összetett objektumot. A `Product` a konstruálandó objektum típusa.

Példák:

```
DocumentBuilder docBuilder =
     factory.newDocumentBuilder();

Document doc = docBuilder.newDocument();
Element rootElement = doc.createElement("products");
doc.appendChild(rootElement);
```

```
XMLEventWriter eventWriter =
     outputFactory.createXMLEventWriter(new
          FileOutputStream(xmlFile));

StartDocument startDocument =
     eventFactory.createStartDocument();

eventWriter.add(startDocument);eventWriter.add(newLine);
```

```
StringBuilder sb = new StringBuilder();
for (Person person : contribs) {
    sb.append(person.getLastName());
    sb.append("-");
}
```

e. Prototípus (Prototype)

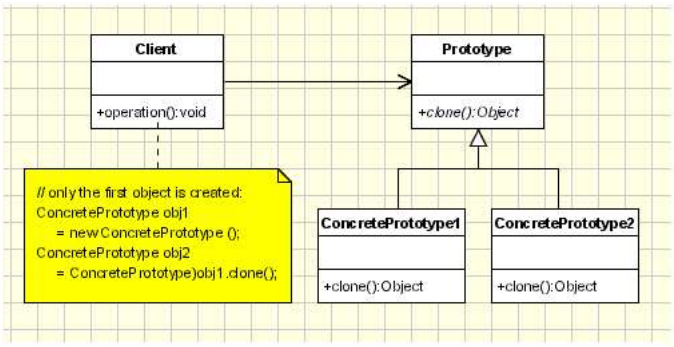
A prototípus minta olyan esetekben használatos, amikor több hasonló objektum előállítására van szükség, és ez az előállítás magas költségekkel jár. A mintának szüksége van egy már létező objektumra, amelyet lemásol, majd módosítja az attribútumait. Ennek megvalósítására klónozást alkalmazhatunk, amelynek implementálásáról a másolandó objektum osztályának kell gondoskodnia. A másolás igény szerint sekély és mély lehet.

Megjegyzés:

A ***sekély másolat (shallow copy)*** készítésekor az eredeti objektum és annak primitív típusú adattagjai másolódnak csak le, a referencia típusú alsóbb szintű (tartalmazott) objektumoknak csak a referenciái másolódnak le, vagyis ha az adott referenciával rendelkező objektum megváltozik, akkor a másolat is a megváltozottat éri majd el.

A ***mély másolat (deep copy)*** készítése során az eredeti objektumnak nem csupán a primitív típusó adattagjairól, de a referencia típusú adattagok mögött álló objektumokról is másolat (méghozzá mély másolat) készül, vagyis a prototípus

alapján legyártott objektum kezdőállapota megegyezik a prototípusáéval, azonban a legyártott objektum által hivatkozott objektumok függetlenek a prototípusobjektum elemeitől.
Az `Object` osztályban a `clone` metódus áll rendelkezésre prototípusok készítésére, ez sekély másolatot készít.



A prototípus minta megvalósításának osztálydiagramja

A kliens létrehoz egy új objektumot a prototípusnak küldött klónozási kéréssel. A prototípus olyan felületet nyújt, amely lehetővé teszi a klónozást. A konkrét prototípusok valósítják meg a klónozási mechanizmust.

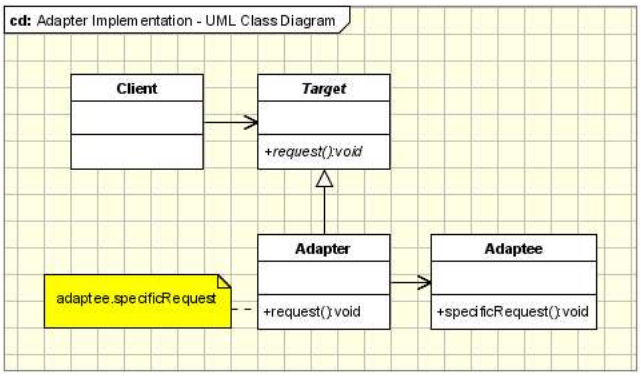
Példa:

```
public class Product implements Cloneable ... {
    @Override
    public Object clone() throws
     CloneNotSupportedException {

        return super.clone();
    }
    ...
}

public class X {
    ...
    public void y(...) {
        try {
            Product prod = (Product) prod.clone();
        } catch (CloneNotSupportedException ex) {
            ...
        }
        ...
    }
}
```

2. Szerkezeti minták

a. Illesztő (Adapter)

Az illesztő minta a nem összeillő interfészek együttműködését teszi lehetővé. Az egyik interfészhez olyan felületet rendel, amely becsatolható a másik interfészbe.



Az illesztő minta megvalósításának osztálydiagramja

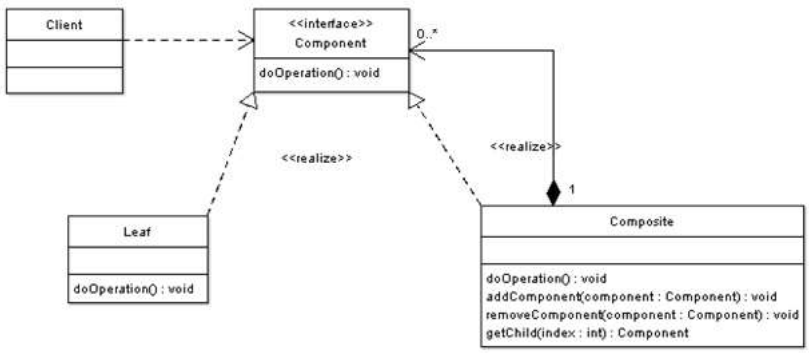
A `Target` egy, a kliens által használt szakterületspecifikus interfész. Az `Adaptee` az adapter által a `Target`-hez alakított már létező interfész.

Példa:

```
public void X(String[] columnNames ...) {
    List<String> simpleTableColNames =
     Arrays.asList(columnNames);
    ...
}
```

b. Összetétel (Composite)

Az összetétel minta egy fastruktúrát kezel. Ez egy olyan hierarchia, amelynek segítségével rész-egész viszonyt fejezhetünk ki. Az egyes részek ugyanolyan módon kezelendők, mint az egész.



Az összetétel minta megvalósításának osztálydiagramja

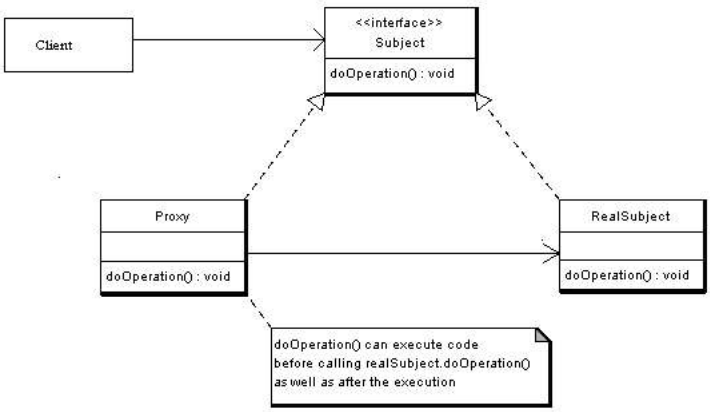
A `Component` egy olyan interfész, amelyet mind az összetett osztálynak, mind a levélelemnek implementálnia kell. A `Leaf` osztály reprezentálja a tovább már nem osztható elemeket, míg a `Composite` osztály objektumai további komponensekből épülnek fel.

Példa:

```
jPanel2Layout.setVerticalGroup(jPanel2Layout
        .createParallelGroup(javax.swing.GroupLayout.Alig
nment.LEADING)
        .addGroup(jPanel2Layout.createParallelGroup(javax
.swing.GroupLayout.Alignment.BASELINE)
                .addComponent(jTextField1,

     javax.swing.GroupLayout.PREFERRED_SIZE,

                    javax.swing.GroupLayout.DEFAULT_SIZE,

     javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(jLabel3)
                .addComponent(jPasswordField1,

     javax.swing.GroupLayout.PREFERRED_SIZE,

                    javax.swing.GroupLayout.DEFAULT_SIZE,

     javax.swing.GroupLayout.PREFERRED_SIZE)
                .addComponent(jButton1)
                .addComponent(jLabel2)
                .addComponent(jButton2)
        )
);
```

c. Helyettes (Proxy)

A helyettes, ahogyan a neve is mutatja, egy másik objektumot helyettesít, illetve a másik objektumhoz való hozzáférést felügyeli. Abban az esetben is használjuk, amikor az eredeti objektum létrehozása magas költségekkel jár. A helyettesítő egy másik objektum funkcionalitását mutatja a külvilág felé.



A helyettes minta megvalósításának osztálydiagramja

A `Subject` interfészt implementáló `RealSubject` osztályt helyettesíti a `Proxy`, így a helyettesítőnek is implementálnia kell a `Subject` interfészt.

Példa:

A `Subject` mintaelemet reprezentáló interfész egy operációs rendszerbeli könyvtár műveleteit ábrázolja:

```
public interface IFolder {
    public void performOperations();
}
```

A `Folder` osztály a `RealSubject` mintaelemet valósítja meg:

```
public class Folder implements IFolder{
    public void performOperations() {
        // különféle műveletek elvégzése a könyvtáron
        System.out.println("Művelet elvégzése");
    }
}
```

A proxyobjektum ugyanúgy viselkedik, mint a `RealSubject` (ez esetben a `Folder`), de többletfeladatként hozzáférés-ellenőrzést valósít meg: a `username` nevű, `password` jelszavú felhasználó számára engedélyezzük a műveletet, mások számára nem. Figyeljük meg, hogy proxyobjektum az ellenőrzést követően delegálja a műveletvégzést a Folder objektumnak!

Megjegyzés:

A User osztály itt egy felhasználói név és jelszó párost bezáró egyszerű objektum, kódját mellőzzük.

```
public class FolderProxy implements IFolder {
    Folder folder;
```

```
    User user;

    public FolderProxy(User user) {
        this.user = user;
    }

    public void performOperations() {
        if
     (   user.getUserName().equalsIgnoreCase("username")

             &&
      user.getPassword().equalsIgnoreCase("password")) {

             folder = new Folder();
             folder.performOperations();
         }
         else {
             System.out.println("Hozzáférés megtagadva!");
         }
    }
}
```

A hívás helyén így egyszerűen egy proxyobjektumot hozunk létre, amely a műveletvégzéskor a konstuktorában megkapott felhasználóra elvégzi a jogosultságellenőrzést:

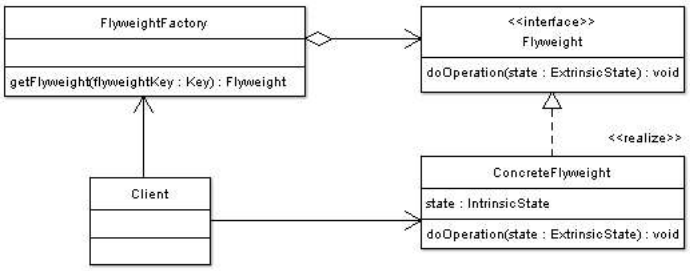
```
String username = ...;
String password = ...;
IFolder folder = new FolderProxy(new User(username,
     password));
folder.performOperations();
```

Megjegyzés:

Természetesen valamely létrehozási minta alkalmazásával lehetőségünk van a példányosítási függőség kimozgatására.

d. Pehelysúlyú (Flyweight)

A pehelysúlyút akkor használjuk, amikor egy osztály több példányára is szükségünk van, és ezeknek az előállítása magas költségekkel jár. A memóriafogyasztás csökkentésére az osztályok osztozhatnak az objektumok külső állapotán.



A pehelysúlyú minta megvalósításának osztálydiagramja

A `Flyweight` interfész biztosítja azt a felületet, amelyen keresztül a pehelysúlyúak külső állapotokat kaphatnak, és ezeken különböző műveleteket végezhetnek. A `ConcreteFlyweight` osztály implementálja a `Flyweight` interfészt és tárolja a belső állapotot. Egy konkrét pehelysúlyú objektumnak megoszthatónak kell lennie, és kezelnie kell mind a belső, mind a külső állapotot. A `FlyweightFactory` osztály felelős a pehelysúlyúak előállításáért és közzétételéért. A gyár egy tárolót tart fenn a különböző pehelysúlyú objektumok számára, ahonnan példányokat ad vissza, ha azok már elkészültek, és ahová az új objektumokat menti el. A kliens feladata a pehelysúlyúak hivatkozása és a külső állapotok kezelése.

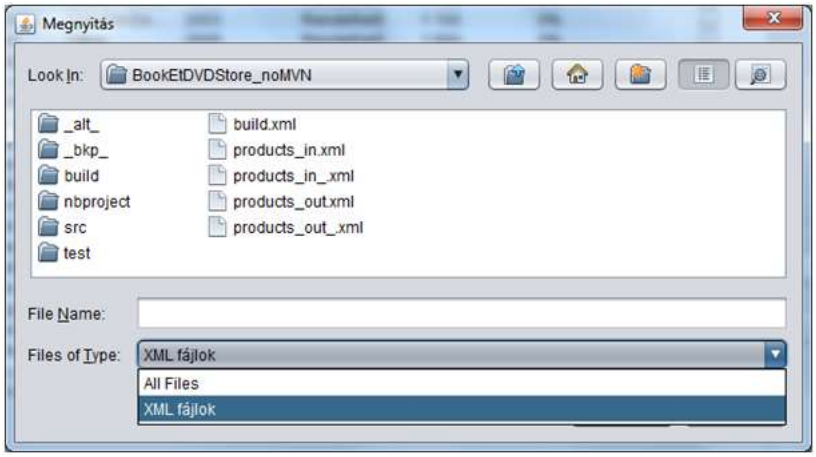
Példák:

A Java sztring-poolozása, valamint

```
Attribute attribute =
     productStartElement.getAttributeByName(QName.valueOf
("selector"));
```

e. Homlokzat (Façade)

A homlokzat minta segítségével a kliens egyszerűbben kommunikálhat a rendszerrel. A háttérben álló osztályok, interfészek és egyéb programozás eszközök bonyolultságának növekedésével nehezedik a rendszer használata. A homlokzat minta egy olyan felületet biztosít, amelyen keresztül a külvilág számára egyszerűen érhetők el a bonyolultabb mechanizmusok.



A homlokzat mintát megvalósító JFileChooser

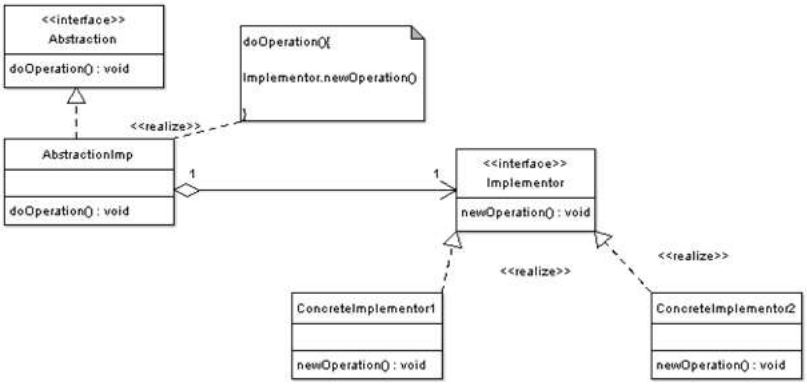
Példa:

```
...
FileFilter filter = new FileNameExtensionFilter("XML
fájlok", "xml");
JFileChooser fc = new JFileChooser();
fc.addChoosableFileFilter(filter);
int retVal = fc.showDialog(this, "Megnyitás");
if (retVal == JFileChooser.APPROVE_OPTION) {
    List<Product> importedProductList =
     xmlReader.readXmlWithDOM(fc.getSelectedFile());
    productsById = new HashMap<>();
    for (Product product : importedProductList) {
        productsById.put(product.getId(), product);
    }
    ...
}
...
```

A fenti kódrészletben egy `javax.swing.JFileChooser` objektumot használunk ahelyett, hogy közvetlenül hivatkoznánk a mögötte álló bonyolult kódot.

f. Híd (Bridge)

A híd minta az absztrakciót és az implementációt választja szét, hogy azok egymástól függetlenül változtathatóak maradhassanak. Ezzel egy időben összetétel segítségével közvetít is a kettő között. Az összetétel előnyt élvez az öröklődés alkalmazásával szemben. A híd felület konkrét osztályok funkcionalitását veszi át ahelyett, hogy interfészeket implementálna.



A híd minta megvalósításának osztálydiagramja

Az `Abstraction` egy interfészt reprezentál. Az `AbstractionImpl` az absztrakciós interfészt implementálja az `Implementor` típus egyik objektumának segítségével. Az `Implementor` egy felületet nyújt az implementáló osztályok számára, amelynek nem szükséges közvetlenül megfelelnie az absztrakciós interfésznek. Az absztrakciós interfész implementációja azok alapján a műveletek alapján történik, amelyeket az `Implementor` interfész biztosít. A konkrét implementorok valósítják meg az `Implementor` interfészt.

Példa:

Az absztrakciót példánkban az egy alakzatot reprezentáló `Shape` interfész jelenti:

```
public interface Shape {
    void colorIt();
}
```

Az absztrakciót implementáló osztály, amely `Implementor` típusú objektumot tartalmaz egy téglalapot ír le:

```
public class Rectangle extends Shape {
    private Color color;

    public Rectangle(Color color) {
        this.color = color;
    }

    @Override
    public void colorIt() {
        color.fillColor();
        System.out.print(" színezett téglalap");
    }
}
```

A megvalósítás absztrakcióját (vagyis a minta `Implementor` elemét) a `Color` interfész írja le:

```
public interface Color {
    void fillColor();
}
```

Két konkrét implementációt készítettünk a `Color` interfészhez, egy piros és egy kék színű alakzat készíthető segítségükkel.

```
public class RedColor implements Color {
    @Override
    public void fillColor() {
        System.out.println("Pirosra");
    }
}
```

```
public class BlueColor implements Color {
    @Override
    public void fillColor() {
        System.out.println("Kékre");
    }
}
```

Alkalmazási példa:

```
Shape s1 = new Rectangle(new RedColor());
Shape s2 = new Rectangle(new BlueColor());
```

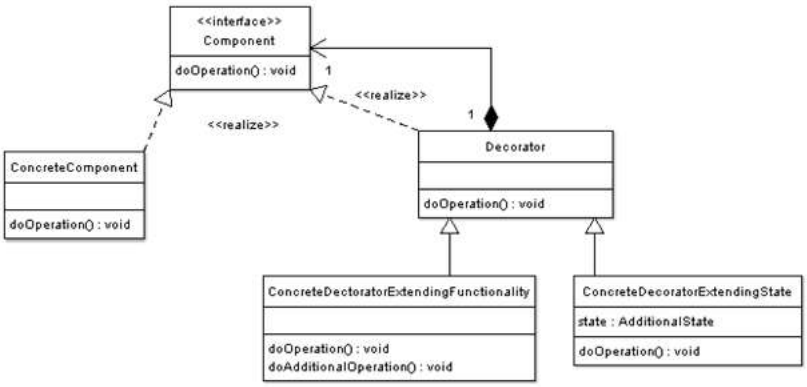
```
A végrehajtás eredménye:

Pirosra színezett téglalap
Kékre színezett téglalap
```

Megjegyzés:

Ez a struktúra könnyen bővíthető mind új konkrét megvalósítással (jelen példa esetében színnel), mind pedig új absztrakciómegvalósításokkal (jelen példában további alakzatokkal, például körrel, stb.). Ezek ráadásul egymástól függetlenül bővíthetőek.

g. Díszítő (Decorator)

A díszítő minta egy objektum funkcionalitását módosítja futási vagy fordítási időben. Másik neve csomagoló (Wrapper). Az új funkcionalitás vagy felelősségi kör megvalósítása nem érinti az eredeti objektumot.



A díszítő minta megvalósításának osztálydiagramja

A `Component` interfész objektumaihoz további funkciók adhatók. A `ConcreteComponent` implementálja a `Component` interfészt. A `Decorator` típus a `Component` objektumokra tartalmaz hivatkozást, és egy olyan felületet biztosít, amely illeszkedik a `Component` felületéhez. A `ConcreteDecorator` terjeszti ki a komponens funkcionalitását új állapot vagy műveletek hozzáadásával.

Példa:

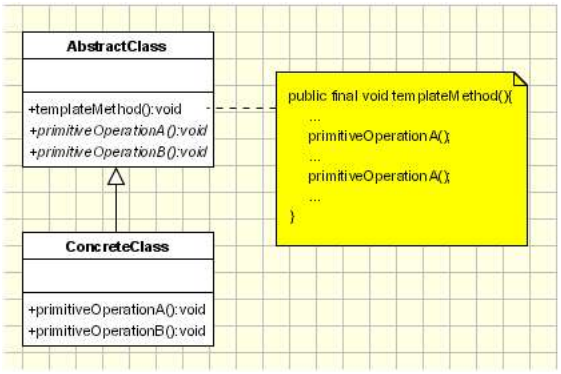
```
AbstractTableModel twoSelTableModel ...;
...
getContentPane().add(new JScrollPane(new
     JTable(twoSelTableModel)));
```

A fenti kódrészletben a tábla komponenst egy görgethető panel díszíti.

3. Viselkedési minták

a. Sablonfüggvény (Template method)

A sablonfüggvény minta egy metóduscsonkot hoz létre, amely néhány lépés implementálását az alosztályokra hagyja. A sablonmetódus meghatározza egy algoritmus lépéseit, amelyek némelyikéhez implementációt is ad. Ez minden leszármazott osztály által közösen használható.



A sablonfüggvény minta megvalósításának osztálydiagramja

Az `AbstractClass` azokat az egyszerű metódusokat határozza meg, amelyeket az algoritmus lépéseiként a leszármazott osztályoknak felül kell definiálniuk. A `templateMethod` sablonmetódus az algoritmus váza, amely a felüldefiniálandó metódusokat hívja. A `ConcreteClass` azokat a metódusokat implementálja, amelyek az algoritmus egyes lépéseit valósítják meg.

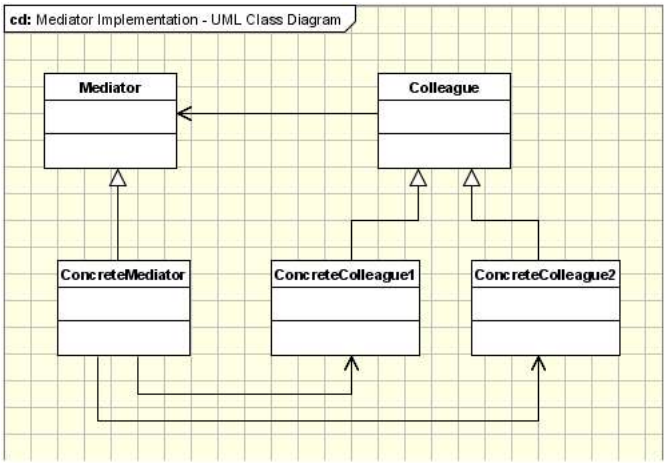
Példa:

A `javax.swing.table.AbstractTableModel` osztály minden nem absztrakt metódusa, például

```
setValueAt(Object ertek, int sorindex, int oszlopindex);
```

b. Közvetítő (Mediator)

A közvetítő minta egy központilag használt médiumot biztosít, amelyen keresztül a rendszer különböző objektumai kommunikálhatnak egymással. Amennyiben az objektumok közvetlenül egymással kommunikálnak, egy szorosan összekötött viszonban vannak, amelynek költséges a karbantartása, rugalmatlan és nehezen bővíthető. A közvetítő arra fókuszál, hogy az objektumok rajta keresztül lépjenek egymással interakcióba, így lazán kapcsoltak maradjanak.



A közvetítő minta megvalósításának osztálydiagramja

A `Mediator` interfész nyújt felületet a `Colleague` objektumok kommunikációjához. A `ConcreteMediator` osztály hivatkozást tartalmaz a `Colleague` objektumokra, és üzeneteket közvetít közöttük. A `Colleague` osztályok hivatkozást tartalmaznak a közvetítőjükre, és azzal kommunikálnak ahelyett, hogy egy másik `Colleague` objektummal tennék azt.

Példa:

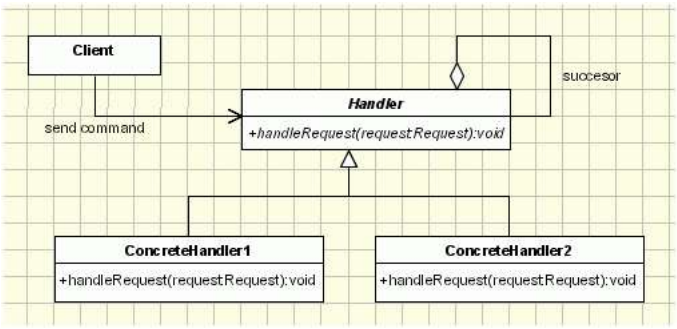
```
public class BrowseAndSearch extends JFrame {
    ...
    private Login login;
    private BookShelf bookShelf;
    private ShoppingCart shoppingCart;
    private OrderDialog orderDialog;
    ...
}

public class BookShelf extends JDialog {
    ...
    private BrowseAndSearch parentFrame;
    ...
}
```

c. Felelősséglánc (Chain of responsibility)

A felelősséglánc a lazán csatoltság megvalósítására szolgál a szoftvertervezésben. A klienstől érkező kérésobjektum objektumok egy sorozatán (láncán) jár végig, amíg feldolgozásra nem kerül. A lánc minden objektuma kezelheti a kérést, továbbíthatja azt, vagy esetleg mindkettőt végezheti. A láncot felépítő objektumok maguk döntik el, hogy melyiküknek kell feldolgoznia a kérést, és hogy az továbbmenjen-e a láncon vagy sem. Ezzel biztosítjuk, hogy a kérést az arra legalkalmasabb objektum dolgozza fel.

Megjegyzés:

Ha valamelyik objektum feldolgozza a kérést, az általában nem halad már tovább a láncon, bár a konkrét megvalósítás dönthet másként.
Köznapi példa erre egy üdítőautomata, ahol bár különféle pénzérmékkel fizethetünk, mégsincs külön nyílás minden pénzérmefajta számára, hanem a bedobott pénzérméket az automata egy felelősséglánc segítségével dolgozza fel: külön ellenőrzés tartozik minden pénzérmefajtához, de ha egy ellenőrzés nem képes meghatározni a pénzérme típusát, akkor továbbítja azt a következő ellenőrzőnek.



A felelősséglánc minta megvalósításának osztálydiagramja

A `Handler` interfész a kérések kezelését biztosítja. A `RequestHandler` kezeli azt a kérést, amelyért ő a felelős. Amennyiben fel tudja dolgozni a kérést, megteszi, ellenkező esetben továbbküldi azt a következő kezelő láncszemnek. A kliens küldi el a kérést a lánc első elemének feldolgozásra.

Példa:

A `Handler` interfészt példánkban az `EmailHandler` interfész adja meg:

```
public interface EmailHandler {
    void setNext(EmailHandler handler);
    void handleRequest(Email email);
}
```

`ConcreteHandler` megvalósításból többet is készítünk: az egyik az üzleti, a másik a gmail-es e-mail címre érkező levelek megfelelő mappába mentését végzi. Ha a kezelő nem ismeri fel az e-mail cím végződését, akkor továbbítja, egyébként elvégzi a megfelelő feldolgozási műveleteket.

```
public class BusinessMailHandler implements EmailHandler
{
    private EmailHandler next;

    public void setNext(EmailHandler handler) {
```

```
        next = handler;
    }

    public void handleRequest(Email email) {
        if(!
     email.getFrom().endsWith("@businessaddress.com") {
            next.handleRequest(email);
        }
        else {
            // a kérés kezelése (megfelelő mappába
     mentés)
        }
    }
}
```

```
public class GMailHandler implements EmailHandler {
    private EmailHandler next;

    public void setNext(EmailHandler handler) {
        next = handler;
    }

    public void handleRequest(Email email) {
        if(!email.getFrom().endsWith("@gmail.com") {
            next.handleRequest(email);
        }
        else {
            // a kérés kezelése (megfelelő mappába
     mentés)
        }
    }
}
```

Az `EmailProcessor` osztály a kezelők menedzselését végzi, segítségével dinamikusan tudunk új kezelőket hozzáadni, vagyis bővíteni tudjuk a felelősségláncot.

```
public class EmailProcessor {
    // az előző kezelő referenciáját nyilvántartjuk, így
könnyebb a következőt hozzáadni
    private EmailHandler prevHandler;

    public void addHandler(EmailHandler handler) {
        if(prevHandler != null) {
            prevHandler.setNext(handler);
        }
        prevHandler = handler;
    }
}
```

Példa az alkalmazásra:

```
Email email = ...;
```

```
EmailProcessor ep = new EmailProcessor();
ep.addHandler(new BusinessMailHandler());
ep.addHandler(new GMailHandler());
ep.handleRequest(email);
```

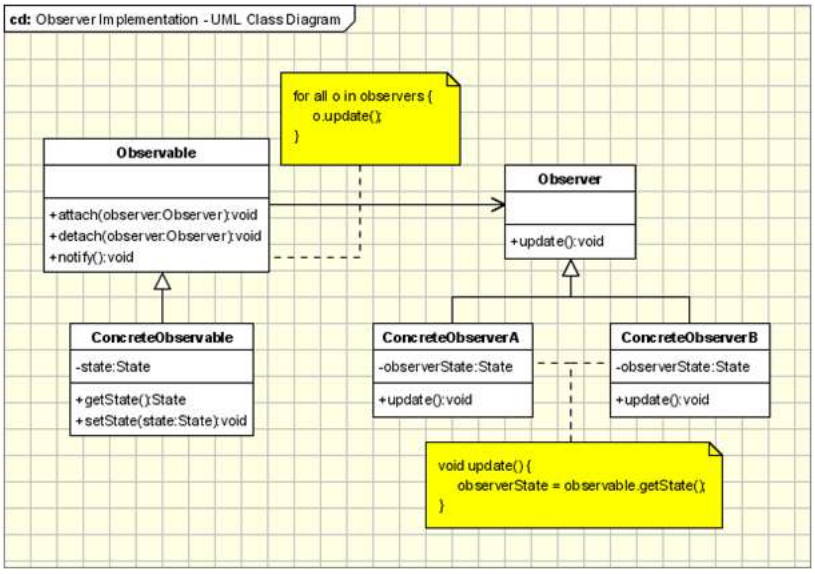
Tipp:

Könnyebben olvashatóvá tehetjük a kódunkat, ha úgynevezett folyékony API-t (fluent API) készítünk. Ehhez módosítsuk az `EmailProcessor` osztály `addHandler` metódusát úgy, hogy ne eljárás legyen, hanem `EmailProcessor` példányt adjon vissza, mert ekkor az eredményobjektumon újabb műveleteket végezhetünk.

```
...
    public EmailProcessor addHandler(EmailHandler
handler) {
        if(prevHandler != null) {
            prevHandler.setNext(handler);
        }
        prevHandler = handler;
        return this;
    }
    ...

Email email = ...;
new EmailProcessor()
    .addHandler(new BusinessMailHandler())
    .addHandler(new GMailHandler())
    .handleRequest(email);
```

d. Megfigyelő (Observer)

A megfigyelő minta akkor hasznos, amikor egy objektum egy másik objektum állapotában érdekelt, és tudatában kell lennie (értesítést kell kapnia) az ebben bekövetkező változásokról. A megfigyelő mintában megfigyelőnek (`Observer`) hívják azt az objektumot, amely ellenőrzi egy másik objektum, a megfigyelt alany (`Subject, Observable`) állapotváltozásait.



A megfigyelő minta megvalósításának osztálydiagramja

Az `Observable` vagy más néven alany (`Subject`) egy interfész vagy absztrakt osztály, amely metódusokat tartalmaz a megfigyelők kapcsolódására és lekapcsolódására. A `ConcreteObservable` osztály a megfigyelt objektum állapotát kezeli, és értesítést küld a felcsatolt megfigyelőknek, amennyiben ebben az állapotban változás következik be. Az `Observer` interfész vagy absztrakt osztály metódusokat tartalmaz a megfigyelt objektumok állapotváltozásainak feldolgozására. A `ConcreteObserver` osztályok az `Observer` interfész implementációi.

Példa:

A Java Swing osztálykönyvtárában számos helyen találkozunk a Megfigyelő mintával. Az alábbi példában a `guiObject` játssza az `Observable` szerepét (a minta `attach` metódusának megfelelője az `addActionListener`, amellyel az egyes eseménykezelők regisztrálhatják magukat a megfigyelhető felhasználóifelület-komponensnél). Az `ActionListener` interfész az `Observer` szerepében tűnik fel (az `update` metódus megfelelője itt az `actionPerformed`), míg maga a `MyActionListener` osztály egy konkrét megfigyelőt ír le.

```
public class MyActionListener implements ActionListener {
    @Override
    public void
     actionPerformed(java.awt.event.ActionEvent evt) {
        //Exportáláshoz használt fájlformátum
     kiválasztása
        JComboBox cb = (JComboBox) evt.getSource();
        String selectedFileFormat = (String)
     cb.getSelectedItem();
        ...
    }
```

```
}
```

A figyelő regisztrációját a megfigyeltnél az alábbi módon végezzük:

```
guiObject.addActionListener(new MyActionListener());
```

Egy másik példa azt mutatja be, hogy hogyan alkalmazhatjuk a Megfigyelő mintát egy billentyűzetről beolvasott egész szám bináris, oktális és hexadecimális értékének a megjelenítésére. Ehhez először is definiálnunk kell egy `Observer` interfészt. Mivel minden konkrét megfigyelő kapcsolatban áll a megfigyelttel (subject-tel), ezért ezt absztrakt osztályként valósítjuk meg, amely rendelkezik egy `Subject` referenciával.

```
public abstract class Observer {
    protected Subject subject;

    public abstract void update();
}
```

A `Subject` a megfigyelt objektum, amely a példában a beolvasott egész érték hordozója. Mindemellett nyilvántartja a nála feliratkozott megfigyelőket is. A `notifyObservers` metódus az összes regisztrált megfigyelőt értesíti. Erre az értesítésre az állapot változásakor (`setState`) van szükség.

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

public class Subject {
    private List<Observer> observers;
    private int state;

    public Subject() {
        observers = new ArrayList<>();
    }

    public int getState() {
        return state;
    }

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

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

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

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

Most már létrehozhatjuk a konkrét megfigyelőket megvalósító `ConcreteObserver`-eket, mint az `Observer` absztrakt osztály leszármazottait:

```
public class BinObserver extends Observer {
    public BinObserver(Subject subject) {
        this.subject = subject;
    }

    @Override
    public void update() {

     System.out.println(Integer.toBinaryString(subject.ge
tState()) + " BIN");
    }
}
```

```
public class OctObserver extends Observer {
    public OctObserver(Subject subject) {
        this.subject = subject;
    }

    @Override
    public void update() {

     System.out.println(Integer.toOctalString(subject.get
State()) + " OCT");
    }
}
```

```
public class HexObserver extends Observer {
    public HexObserver(Subject subject) {
        this.subject = subject;
    }

    @Override
    public void update() {

     System.out.println(Integer.toHexString(subject.getSt
ate()) + " HEX");
    }
}
```

A főprogramban létrehozzuk a `Subject`-et, regisztráljuk nála az egyes megfigyelőket, majd mindaddig, amíg negatív számot nem olvasunk, egészeket olvasunk a billentyűzetről, és frissítjük vele a `Subject` objektum állapotát.

```
import java.util.Scanner;

public class ObserverMain {
    public static void main(String[] args) {
        Subject s = new Subject();
        s.attach(new BinObserver(s));
        s.attach(new OctObserver(s));
        s.attach(new HexObserver(s));
        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.print("Adjon meg egy számot: ");
            int x = sc.nextInt();
            if (x < 0)
                break;
            s.setState(x);
        }
        sc.close();
    }
}
```

A program kimenetén lthatjuk, hogy az állapotbeállításra reagálva automatkusan lefutnak a megfigyelők `update` metódusai:

```
Adjon meg egy számot: 1024
10000000000 BIN
2000 OCT
400 HEX
Adjon meg egy számot: 7
111 BIN
7 OCT
7 HEX
Adjon meg egy számot: 13
1101 BIN
15 OCT
d HEX
Adjon meg egy számot: 0
0 BIN
0 OCT
0 HEX
Adjon meg egy számot: -1
```

Tipp

A Megfigyelő (Observer) minta támogatására a szabványos Java API-ban is találunk eszközöket, ezeket is alkalmazhatjuk!. Ezek a java.util csomagban elhelyezkedő `Observable` osztály (amely a megfigyeltet, vagy más néven a subject-et valósítja meg) és `Observer` interfész (amely a megfigyelők számára biztosít felületet). Ez azt jelenti, hogy nincs szükség saját `Observer` interfész megvalósítására, és a megfigyelők megfigyeltnél történő regisztrációra is alapból adott (ezt az `Observable` osztály biztosítja). A kódunk ekkor az alábbiak szerint alakul:

```
import java.util.Observable;

public class Subject extends Observable {
    private int state;

    public int getState() {
        return state;
    }

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

Látható, hogy a `Subject` osztályból kikerültek a megfigyelők regisztrálására, deregisztrálására illetve értesítésére szolgáló metódusok, mert ezeket a `Subject` osztály az `Observable`-től (implementációval együtt) megörökli. A `setState` metódusban az `Observable`-től örökölt `setChanged` metódussal jelezzük, hogy megváltozott az állapot, és a `notifyObservers`-szel értesítjük a regisztrált megfigyelőket.

A konkrét megfigyelők implementációja is egyszerűsödik: nem kell nyilvántartanunk a figyelőben a megfigyeltet, mert azt mindig megkapjuk az update első paramétereként.

```
import java.util.Observable;
import java.util.Observer;

public class BinObserver implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof Subject)

     System.out.println(Integer.toBinaryString(((Subject)
o).getState()) + " BIN");
    }
}

import java.util.Observable;
import java.util.Observer;

public class BinObserver implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof Subject)

     System.out.println(Integer.toOctalString(((Subject)
o).getState()) + " BIN");
    }
```

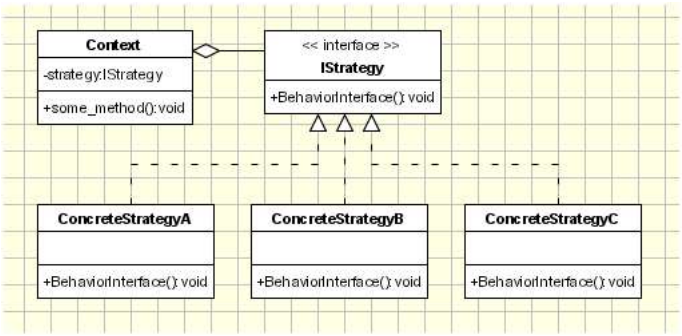
```
}

import java.util.Observable;
import java.util.Observer;

public class BinObserver implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof Subject)

     System.out.println(Integer.toHexString(((Subject)
o).getState()) + " BIN");
    }
}
```

A főprogram is egyszerűsödik: a `Subject` objektum `Observable`-től örökölt `addObserver` metódusával végezzük a figyelők regisztrációját.

```
import java.util.Scanner;

public class ObserverMain {
     public static void main(String[] args) {
          Subject s = new Subject();
          s.addObserver(new BinObserver());
          s.addObserver(new OctObserver());
          s.addObserver(new HexObserver());
          Scanner sc = new Scanner(System.in);
          while (true) {
               System.out.print("Adjon meg egy számot:
");
               int x = sc.nextInt();
               if (x < 0)
                    break;
               s.setState(x);
          }
          sc.close();
     }
}
```

e. Stratégia (Strategy)

A stratégia minta egy feladat elvégzésre több különböző algoritmust biztosít, és a kliens dönti el, hogy ezek közül melyiket használja (például paraméter segítségével).



A stratégia minta megvalósításának osztálydiagramja

A `Strategy` interfész egy közös felületet nyújt a támogatott algoritmusok számára. Azt, hogy ennek mely `ConcreteStrategy` implementációja hajtódjon végre, a stratégia környezete dönti el. Minden konkrét stratégia osztály egy algoritmust implementál. A `Context` osztály hivatkozást tartalmaz egy stratégia objektumra, miközben a konkrét implementációkról semmit nem tud.

Példák:

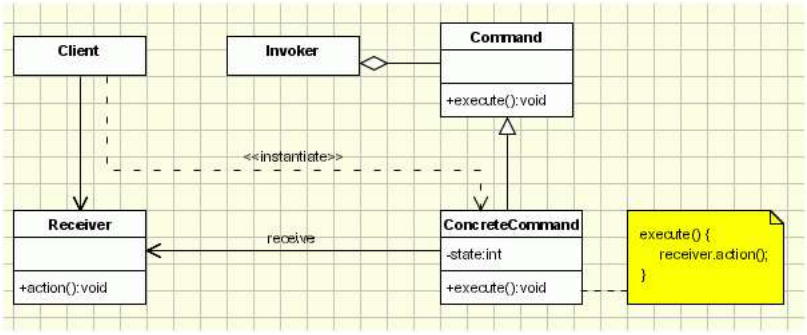
A Java Collections Framework-ben található `Collections.sort` metódus, amelynek tetszőleges `Comparator` implementációt adhatunk meg, valamint

```
public class XmlReader {
    public List<Product> readXml(File xmlFile, String
     withAPI) {

        if ("DOM".equalsIgnoreCase(withAPI)) {
            return new
     DOMReader().readXmlWithDOM(xmlFile);

        } else if ("StAX".equalsIgnoreCase(withAPI)) {
            return new
     StAXReader().readXmlWithStAX(xmlFile);
        } else {
             return new
     SAXReader().readXmlWithSAX(xmlFile);
        }
    }
}

public class DOMReader {
    public List<Product> readXmlWithDOM(File xmlFile)
     {...}
}

public class SAXReader {
    public List<Product> readXmlWithSAX(File xmlFile)
```

```
      {...}
}

public class StAXReader {
    public List<Product> readXmlWithStAX(File xmlFile)
{...}
}
```

f. Parancs (Command)

A parancs minta laza kapcsolat megvalósítására szolgál egy kérés–válasz modellben. Ebben a mintában a kérés a hívónak küldődik, amely továbbítja azt a beágyazott parancsobjektum felé. A parancsobjektum továbbítja a kérést a fogadó megfelelő metódusához, hogy végrehajtódjon a kívánt tevékenység.



A parancs minta megvalósításának osztálydiagramja

A `Command` egy művelet végrehajtásához biztosít felületet. A `ConcreteCommand` a parancs interfészének megvalósítása a megfelelő műveletek meghívásával a fogadó objektumon. A kliens egy konkrét parancsobjektumot hoz létre, beállítva a fogadót. Az `Invoker` kéri meg a parancsobjektumot, hogy hajtsa végre a kérést. Egy parancs végrehajtásának folyamata a kliens kérésére indul el. Az `Invoker` veszi a parancsot, beágyazza, majd a `ConcreteCommand` objektum végrehajtja a kért parancsot és az eredményt visszaszolgáltatja a fogadónak.

Példa:

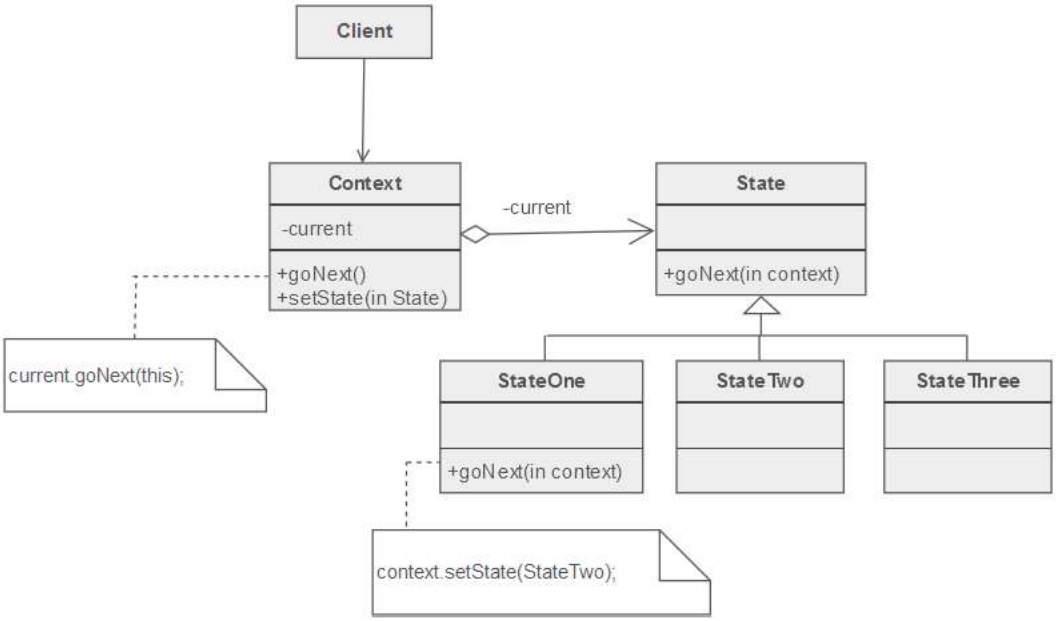
A `javax.swing.Action` minden implementációja. Ezek `ActionListener` objektumként vannak a komponenshez hozzáadva.

```
jComboBox1.addActionListener(new MyActionListener());
```

g. Állapot (State)

Állapot mintát abban az esetben használunk, ha egy objektum megváltoztatja a

viselkedését a belső állapota alapján. Ez a viselkedési mód `if-else` blokkok használata helyett állapotobjektumokkal valósul meg. A különböző állapottípusok mind ugyanannak az általános osztálynak a leszármazottai. Ez a stratégia egyszerűsíti és könnyen érthetővé teszi a kódot.



Az Állapot minta megvalósításának osztálydiagramja

Példa:

Mozifilm-kölcsönző rendszer esetén az árképzés megvalósításánál figyelembe kell vennünk, hogy a filmek árkategóriái futásidőben is változhatnak, hiszen egy új kiadású film idővel kikerül az új kiadásúak számára fenntartott árkategóriából, és valamely más kategóriába kerül át (egyik állapotból a másikba jut). Mivel a Java nyelvben az objektumok nem képesek arra, hogy dinamikusan osztályt váltsanak, ezért az Állapot minta alkalmazásával érhetjük el a kívánt hatást. Ez esetben a környezet, vagyis a `Context` a `Movie` osztály lesz, a minta `State` elemének szerepét pedig a `Price` osztály játssza, amelynek leszármazottjai az egyes konkrét árképzési stratégiák (`NewReleasePrice, ChildrensPrice, RegularPrice`). Mivel a `Movie` osztály nem öröklődést, hanem objektumösszetételt használ, az egyazon mozifilm objektumhoz kapcsolódó `Price` objektum dinamikusan változtatható lesz.

A `Price` egy absztrakt osztály, amely a különféle állapotok ősosztálya lesz. Ez hordozza az állapotfüggő viselkedést is, amely példánk esetén a film kölcsönzési árának kiszámítása a kölcsönzési napok függvényében.

```
package hu.unideb.inf.prt.refactoring;
```

```
public abstract class Price {
    public abstract double getCharge(int daysRented);
}
```

A `Movie` osztály tartalmaz tehát egy `Price` referenciát, amely az adott mozifilm árát reprezentálja. Az állapotváltozásokkor a `setState` metódus meghívására van szükség, melynek paramétere egy konkrét állapot (a `Price` valamely leszármazottja) lesz.

```
package hu.unideb.inf.prt.refactoring;

public class Movie {
    private String title;
    private Price price;

    public Movie(String title, Price price) {
        this.title = title;
        this.price = price;
    }

    public Price getPrice() {
        return price;
    }

    public void setPrice(Price price) {
        this.price = price;
    }

     ...
}
```

A konkrét állapotokat a `Price` alosztályai határozzák meg, az éllapotfüggő viselkedés pedig polimorf megvalósítással rendelkezik:

```
package hu.unideb.inf.prt.refactoring;

public class NewReleasePrice extends Price {
    @Override
    public double getCharge(int daysRented) {
        return daysRented * 3;
    }
}
```

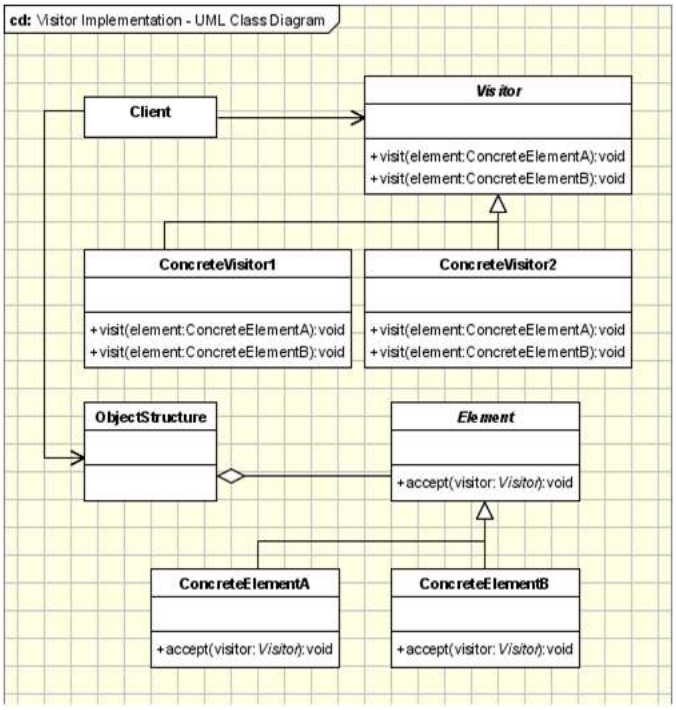
```
package hu.unideb.inf.prt.refactoring;

public class ChildrensPrice extends Price {
    @Override
    public double getCharge(int daysRented) {
        double thisAmount = 0;
        thisAmount += 1.5;
        if (daysRented > 3)
            thisAmount += (daysRented - 3) * 1.5;
```

```
        return thisAmount;
    }
}
```

```
package hu.unideb.inf.prt.refactoring;

public class RegularPrice extends Price {
    @Override
    public double getCharge(int daysRented) {
        double thisAmount = 0;
        thisAmount += 2;
        if (daysRented > 2)
            thisAmount += (daysRented - 2) * 1.5;
        return thisAmount;
    }
}
```

h. Látogató (Visitor)

A látogató minta az ugyanahhoz a típushoz tartozó objektumok csoportján végzendő műveletek elvégzésére szolgál. A minta segítségével a logikát az objektum osztályából egy másik osztályba vihetjük át. Különböző látogatók használatával különböző funkcionalitások rendelhetők a látogatott osztályhoz anélkül, hogy annak szerkezetében változás következne be.



A látogató minta megvalósításának osztálydiagramja

A `Visitor` interfész határozza meg a látogat metódusokat minden látogatható típusra. A metódusok neve ugyanaz, de paraméterük eltér aszerint, hogy milyen látogatható osztályokhoz adnak `visit` metódust. A `ConcreteVisitor` a látogató interfész megvalósítása. Minden látogató más és más műveletekért felel. A `Visitable` interfész egy `accept` metódust definiál, amely belépési pontként szolgál a látogató metódusa számára. A `ConcreteVisitable` osztályok azok a típusok, amelyeken a látogató műveleteket végez. Az `ObjectStructure` osztály tartalmazza az összes látogatható objektumot, és mechanizmust kínál az objektumok bejárására is. Osztálya nem feltétlenül kollekció, lehet összetétel is.

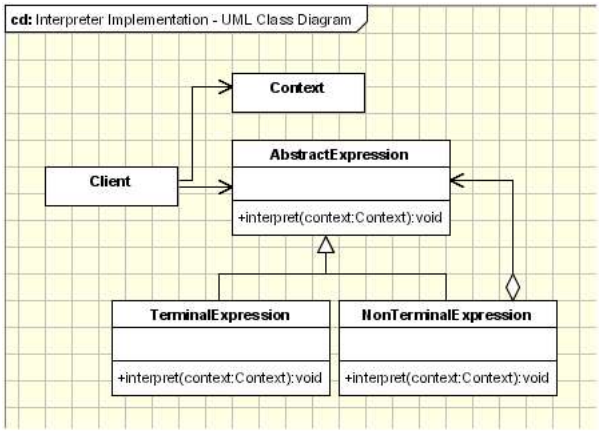
Példa:

```
public interface PriceVisitable {
    public double accept(PriceVisitor visitor);
}

public interface PriceVisitor {
    public double visit(Item item);
    public double visit(Customer customer);
}

public class Item implements PriceVisitable {
    ...
    @Override
    public double accept(PriceVisitor visitor) {
        return visitor.visit(this);
    }
    ...
}

...
public static class BDSPriceCalculator implements
     PriceVisitor {
    @Override
    public double visit(Item item) {
        double discountedTotalPrice =
     item.getProduct().getPrice();
        discountedTotalPrice *= (1 -
     item.getArticle().getDiscounts());
        discountedTotalPrice *= item.getPieces();
        return discountedTotalPrice;
    }
    @Override
    public double visit(Customer customer) {
        double customerDiscount = 0;
        switch (customer.getCategory()) {
            case "NONE": {customerDiscount = 0; break;}
            case "SILVER": {customerDiscount = 0.05;
     break;}
            case "GOLD": {customerDiscount = 0.1; break;}
            ...
```

```
        }
        return customerDiscount;
    }
}

public class Order ... {
    ...
    public void calculateTotalPrice() {
        PriceVisitor priceVisitor =
     new ....BDSPriceCalculator();
        double tp = 0;
        for (Item it : items)  {
            tp += it.accept(priceVisitor);
        }
        this.totalPrice = tp;
    }
    ...
}
```

i. Értelmező (Interpreter)

Ez a minta egy nyelvtanhoz ad értelmezést. Általában fastruktúrában építi fel a nyelv elemeit, amelyhez egy szakterületet, egy nyelvtant és egy OO világra történő leképezést határoz meg.



Az értelmező minta megvalósításának osztálydiagramja

Az `Expression` osztályok az értelmezendő nyelv grammatikájának különböző típusú elemeit reprezentrálják.

Példák:

A reguláris kifejezések feldolgozása (`java.util.regex` csomag), valamint

```
NumberFormat nf = NumberFormat.getInstance();
colValues.add(nf.format(book.getPrice()));
```

Az Értelmező minta egyik klasszikus példája a római számok értelmezése (és arab számokká alakítása). Az értelmezendő kifejezés egy olyan sztring, amelynek a környezetét a még nem értelmezett római szám sztring és a már elemzett résznek megfelelő szám együttesen alkotja. Ezt a környezetet négy értelmezőnek adjuk át: az egyik az ezresek, a másik a százasok, a harmadik a tízesek, míg a negyedik az egyesek értelmezését végzi (ebben a példában csak terminális kifejezések szerepelnek). A környezet kezdeti állapota a teljes átalakítandó római számot tartalmazó sztring és a 0 érték (a kimenő decimális).

```
public class Context {
    private String input;
    private int output;

    public Context(String input) {
      this.input = input;
    }

    public String getInput() {
      return input;
    }

    public void setInput(String input) {
      this.input = input;
    }

    public int getOutput() {
      return output;
    }

    public void setOutput(int output) {
      this.output = output;
    }
}
```

Az `Expression` osztály a kifejezések absztrakt ősosztálya, amely a környezet alapján megvalósítja az értelmezést.

```
public abstract class Expression {
    public void interpret(Context context) {
        if (context.getInput().length() == 0)
            return;
        if (context.getInput().startsWith(nine())) {
            context.setOutput(context.getOutput() + (9 *
multiplier()));

     context.setInput(context.getInput().substring(2));
        }
```

```
        else
            if (context.getInput().startsWith(four())) {
                context.setOutput(context.getOutput() +
(4 * multiplier()));

     context.setInput(context.getInput().substring(2));
            }
            else
                if
     (context.getInput().startsWith(five())) {
                    context.setOutput(context.getOutput()
+ (5 * multiplier()));

     context.setInput( context.getInput().substring(1));
                }
        while (context.getInput().startsWith(one())) {
            context.setOutput(context.getOutput() + (1 *
multiplier()));

     context.setInput(context.getInput().substring(1));
        }
    }

    public abstract String one();
    public abstract String four();
    public abstract String five();
    public abstract String nine();
    public abstract int multiplier();
}
```

Az egyes terminális kifejezéseket megvalósító osztályok megvalósítják az absztrakt osztály metódusait.

```
public class ThousandExpression extends Expression {
    public String one() { return "M"; }
    public String four(){ return " "; }
    public String five(){ return " "; }
    public String nine(){ return " "; }
    public int multiplier() { return 1000; }
}
```

```
public class HundredExpression extends Expression {
    public  String one() { return "C"; }
    public  String four(){ return "CD"; }
    public  String five(){ return "D"; }
    public  String nine(){ return "CM"; }
    public  int multiplier() { return 100; }
}
```

```
public class TenExpression extends Expression {
    public String one() { return "X"; }
    public String four(){ return "XL"; }
```

```
    public String five(){ return "L"; }
    public String nine(){ return "XC"; }
    public int multiplier() { return 10; }
}
```

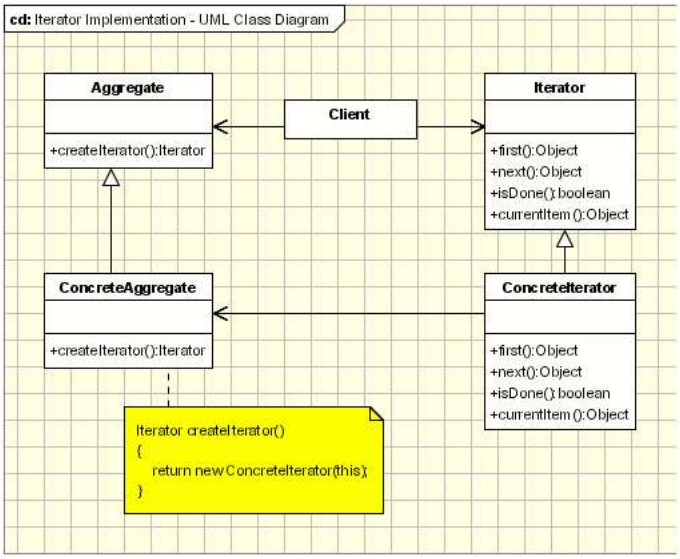
```
public class OneExpression extends Expression {
    public String one() { return "I"; }
    public String four(){ return "IV"; }
    public String five(){ return "V"; }
    public String nine(){ return "IX"; }
    public int multiplier() { return 1; }
}
```

A kliens feladata a környezet létrehozása és a nyelvtan által definiált mondatokat reprezentáló szintakszisfa felépítése. Miután ez felépült, meghívja az interpret metódust a kiértékelés (vagyis a római–arab konverzió elvégzése) érdekében.

```
public class MainInterpreter {
    public static void main(String[] args) {
        String roman = "MCMXXVIII";
        Context context = new Context(roman);
        List<Expression> tree = new ArrayList<Expression>
();
        tree.add(new ThousandExpression());
        tree.add(new HundredExpression());
        tree.add(new TenExpression());
        tree.add(new OneExpression());
        for (Expression exp : tree)
            exp.interpret(context);
        System.out.println(roman + " = " +
     Integer.toString(context.getOutput()));
    }
}
```

j. Bejáró (Iterator)

A bejáró minta szabványos módot kínál objektumcsoportok, -kollekciók szekvenciális bejárására, amely során a kollekció elemei egyesével dolgozhatók fel. A bejárás különböző módokon történhet (minden elemet érintünk-e, milyen irányban haladunk). Az iterátor objektum elrejti a mögötte álló implementációt, és leveszi a kollekcióról a bejárás felelősségét.



A bejáró megvalósításának osztálydiagramja

Az `Iterator` interfész kollekciók bejárására szolgáló metódusokat definiál, amelyeket a `ConcreteIterator` osztály valósít meg. Az `Aggregate` interfész vagy absztrakt osztály reprezentálja a bejárandó kollekciót, és egy iterátor objektum létrehozására alkalmas metódust. A `ConcreteAggregate` osztály az `Aggregate` interfész megvalósítása, amelynek elemeit a konkrét iterátor járja végig.

Példák:

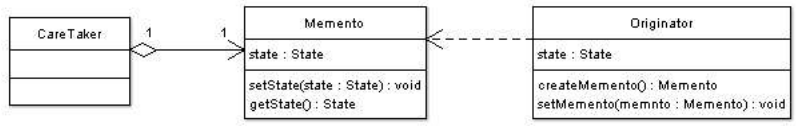
A Java Collections Framework ezt a mintát használja az elemek bejárására, valamint

```
XMLInputFactory inputFactory =
     XMLInputFactory.newInstance();
InputStream in = new FileInputStream(xmlFile);
XMLEventReader eventReader =
     inputFactory.createXMLEventReader(in);
...
while (eventReader.hasNext()) {
    XMLEvent event = eventReader.nextEvent();
    ...
}
...
```

k. Emlékeztető (Memento)

Az emlékeztető minta abban az esetben hasznos, ha el kell tárolnunk és vissza kell állítanunk egy objektum állapotát. Az objektum mentett állapota nem érhető el az objektumon kívülről, így megőrizhető az állapotadatok integritása. A mementó minta

két objektummal dolgozik: egy kezdeményezővel (`Originator`) és egy gondozóval (`Caretaker`). A kezdeményező az az objektum, amelynek állapotát elmentjük, majd visszatöltjük. A mentés egy, a kezdeményező belső osztályaként létrehozott mementó objektumba kerül. A műveletet a gondozó végzi el.



Az emlékeztető minta megvalósításának osztálydiagramja

A `Memento` osztály egy objektuma az `Originator` belső állapotát tárolja. A belső állapotot teszőleges számú adattag képviselheti. Az `Originator` belső osztálya a `Memento` , amelyet a tartalmazó osztály példánya hoz létre. Az `Originator` a `Memento`-t arra használja, hogy saját belső állapotát tárolja benne, majd visszatöltse azt. A `Caretaker` felelős a `Memento` megtartásáért, és azért, hogy az az `Originator` osztályon kívülről ne legyen módosítható. A `Caretaker` nem végezhet műveleteket a `Memento`-n. Az `Originator` egy visszavonási utasítás esetén használja a `Memento`-t előző állapota visszatöltésére.

Példa:

```
...
public class ShelfAndCartMemento {
    private final List<Product> bookShelfContent;
    private final List<Product> shoppingCartContent;

    public    ShelfAndCartMemento(List<Product>    shelf,
List<Product> cart) {
        bookShelfContent = new ArrayList<>(shelf);
        shoppingCartContent = new ArrayList<>(cart);
    }

    public List<Product> getSavedBookShelfContent() {
        return bookShelfContent;
    }

    public List<Product> getSavedShoppingCartContent() {
        return shoppingCartContent;
    }
}

...
public ShelfAndCartMemento saveShelfAndCart() {
    List<Product> productsOnTheShelf = new ArrayList<>();
    List<Product> productsInTheCart = new ArrayList<>();
```

```
    Map<String,        String>        shelfAndCart        =
moreSelTableModel.getSelectionValuesById();
    for (String productId : shelfAndCart.keySet()) {
        if (productsById.containsKey(productId)) {
            if
     (shelfAndCart.get(productId).equalsIgnoreCase("polcr
a")) {


     productsOnTheShelf.add(productsById.get(productId));

            }                    else                    if
(shelfAndCart.get(productId).equalsIgnoreCase("kosárba"))
{

     productsInTheCart.add(productsById.get(productId));

            }
        }
    }
    return   new   ShelfAndCartMemento(productsOnTheShelf,
productsInTheCart);
}

public    void    restoreShelfAndCart(ShelfAndCartMemento
memento) {
    List<Product>  productsOnTheShelfAndInTheCart  =  new
ArrayList<>(memento.getSavedBookShelfContent());

     productsOnTheShelfAndInTheCart.addAll(memento.getSav
edShoppingCartContent());


     moreSelTableModel.setTableDataWithVisitor(productsOn
TheShelfAndInTheCart);

    for         (int         i         =         0;         i
< productsOnTheShelfAndInTheCart.size(); i++) {
        if                                              (i
< memento.getSavedBookShelfContent().size()) {
            moreSelTableModel.setValueAt("Polcra",      i,
moreSelTableModel.getColumnCount() - 1);
        } else { moreSelTableModel.setValueAt("Kosárba",
i, moreSelTableModel.getColumnCount() - 1);
        }
    }
    jTable1.repaint();
}
...
```

Modell-nézet-vezérlő(MNV)

A modell-nézet-vezérlő (MVC-Model-View-Controller) a szoftvertervezésben használatos programtervezési minta. Összetett, sok adatot a felhasználó elé táró számítógépes alkalmazásokban gyakori fejlesztői kívánalom az adathoz (modell) és a felhasználói felülethez (nézet) tartozó dolgok szétválasztása, hogy a felhasználói felület ne befolyásolja az adatkezelést, és az adatok átszervezhetők legyenek a felhasználói felület változtatása nélkül. A modell-nézet-vezérlő ezt úgy éri el, hogy elkülöníti az adatok elérését és az üzleti logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.

Hagyományosan asztali felhasználói felületekhez használt, de manapság már webalkalmazásokhoz is népszerűvé vált. Népszerű programozási nyelvek mint a JavaScript, Python, Ruby, PHP, Java, C# és Swift már külön telepítés szükségessége nélkül rendelkeznek MNV keretrendszerekkel web- és mobilalkalmazások fejlesztésére.

Történet

A mintát Trygve Reenskaug írta le először 1979-ben, miután a Smalltalkon dolgozott a Xerox kutatói laborban. Az eredeti megvalósítás részletesen a nagy hatású Applications Programming in Smalltalk-80: How to use Model-View-Controller című tanulmányban olvasható.

Az MNV minta ezután továbbfejlődött és olyan új változatok jöttek létre mint a hierarchikus modell-nézet-vezérlő (HMVC), modell-nézet-adapter (MVA), modell-nézet-prezenter (MVP), modell-nézet-nézetmodell (MVVM) és mások, amelyek különböző helyzetekhez alakították át az MNV-t.

Gyakori egy alkalmazás több rétegre való felbontása: megjelenítés (felhasználói felület), tartománylogika és adatelérés. Az MNV-ben a megjelenítés tovább bomlik nézetre és vezérlőre. Az MNV sokkal inkább meghatározza egy alkalmazás szerkezetét, mint az egy programtervezési mintára jellemző.

Modell

Az alkalmazás által kezelt információk tartomány-specifikus ábrázolása. A tartománylogika jelentést ad a puszta adatnak (pl. kiszámolja, hogy a mai nap a felhasználó születésnapja-e, vagy az összeget, adókat és szállítási költségeket egy vásárlói kosár elemeihez).
Sok alkalmazás használ állandó tároló eljárásokat (mint mondjuk egy adatbázis) adatok tárolásához. Az MNV nem említi külön az adatelérési réteget, mert ezt beleérti a modellbe.

Nézet

Megjeleníti a modellt egy megfelelő alakban, mely alkalmas a felhasználói interakcióra, jellemzően egy felhasználói felületi elem képében. Különböző célokra különböző nézetek létezhetnek ugyanahhoz a modellhez.

Vezérlő

Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat.

Az MNV gyakran látható webalkalmazásokban, ahol a nézet az aktuális HTML oldal, a vezérlő pedig a kód, ami összegyűjti a dinamikus adatokat és létrehozza a HTML-ben a tartalmat. Végül a modellt a tartalom képviseli, ami általában adatbázisban vagy XML állományokban van tárolva.

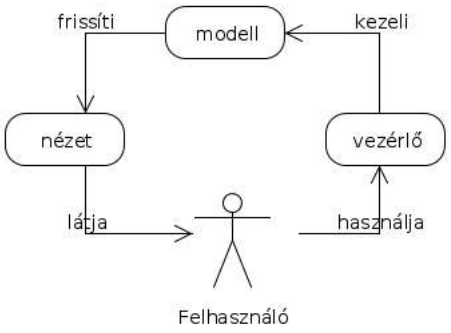
Az MNV-nek sok értelmezése létezik, a vezérlés menete általánosságban a következőképp működik:

1. A felhasználó valamilyen hatást gyakorol a felhasználói felületre (pl. megnyom egy gombot).
2. A vezérlő átveszi a bejövő eseményt a felhasználói felülettől, gyakran egy bejegyzett eseménykezelő vagy visszahívás útján.
3. A vezérlő kapcsolatot teremt a modellel, esetleg frissíti azt a felhasználó tevékenységének megfelelő módon (pl. a vezérlő frissíti a felhasználó kosarát). Az összetett vezérlőket gyakran alakítják ki az utasítás mintának megfelelően, a műveletek egységbezárásáért és a bővítés egyszerűsítéséért.
4. A nézet (közvetve) a modell alapján megfelelő felhasználói felületet hoz létre (pl. a nézet hozza létre a kosár tartalmát felsoroló képernyőt). A nézet a modellből nyeri az adatait. A modellnek nincs közvetlen tudomása a nézetről.
5. A felhasználói felület újabb eseményre vár, mely az elejéről kezdi a kört.

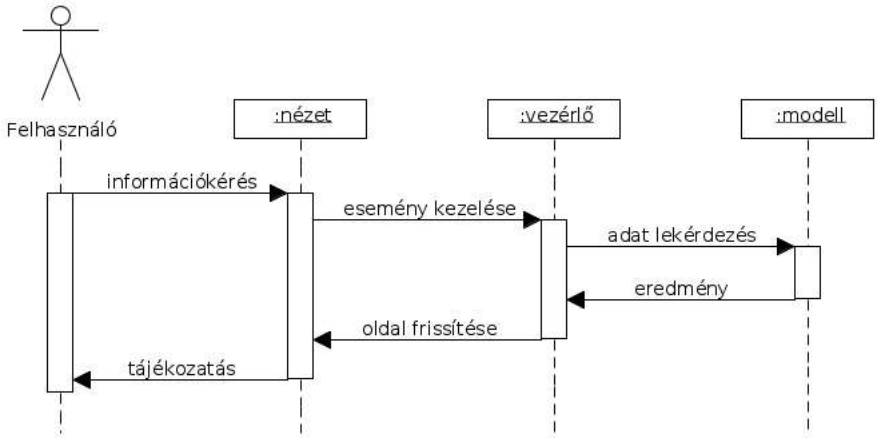
A modell és a nézet kettéválasztásával az MNV csökkenti a szerkezeti bonyolultságot, és megnöveli a rugalmasságot és a felhasználhatóságot.

Szolgáltatás (Service)

A vezérlő és a modell közötti réteg. A modelltől kér le adatokat és a vezérlőnek adja azt. Ennek a rétegnek a segítségével az adattárolás (modell), adatlekérés (szolgáltatás) és adatkezelés (vezérlő) elkülöníthetőek egymástól. Mivel ez a réteg nem része az eredeti MNV mintának, ezért használata nem kötelező.



Modell-nézet-vezérlő minta elve



Modell-nézet-vezérlő minta megvalósítása

Előnyök:

- Egyidejű fejlesztés – Több fejlesztő tud egyszerre külön a modellen, vezérlőn és a nézeteken dolgozni.
- Magas szintű összetartás – MNV segítségével az összetartozó funkciók egy vezérlőben csoportosíthatóak. Egy bizonyos modell nézetei is csoportosíthatóak.
- Függetlenség – MNV mintában az elemek alapvetően nagy részben függetlenek egymástól
- Könnyen változtatható – Mivel a felelősségek szét vannak választva a jövőbeli fejlesztések könnyebbek lesznek
- Több nézet egy modellhez – Modelleknek több nézetük is lehet
- Tesztelhetőség – Mivel a felelősségek tisztán szét vannak választva, a külön elemek könnyebben tesztelhetőek egymástól függetlenül

Hátrányok:

A MNV hátrányait általában a szükséges extra kódból adódnak.

- Kód olvashatósága – A keretrendszer új rétegeket ad a kódhoz ami megnöveli a bonyolultságát
- Sok boilerplate kód – Mivel a programkód 3 részre bomlik, ebből az egyik fogja a legtöbb munkát végezni a másik kettő pedig az MNV minta kielégítése miatt létezik.
- Nehezebben tanulható – A fejlesztőnek több különböző technológiát is ismernie kell a MNV használatához.

Források:

Programtervezési minta -
https://hu.wikipedia.org/wiki/Programtervez%C3%A9si_minta
Létrehozási minta -
https://hu.wikipedia.org/wiki/L%C3%A9trehoz%C3%A1si_minta
Szerkezeti minta -
https://hu.wikipedia.org/wiki/Szerkezeti_minta
Viselkedési minta -
https://hu.wikipedia.org/wiki/Viselked%C3%A9si_minta
Létrehozási, szerkezeti és viselkedési minták és ábrák -
https://gyires.inf.unideb.hu/GyBITT/21/ch04s03.html
Modell-nézet-vezérlő -
https://gyires.inf.unideb.hu/GyBITT/24/ch05s03.html
MVN ábrák -
https://gyires.inf.unideb.hu/GyBITT/24/ch05s03.html