**Programtervezési minták**: Egymással együttműködő objektumok és osztályok leírásai, amelyek testre szabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben.

Közel 30 tervezési mintát fogok ismertetni, és ezek 3 nagy kategóriába sorolhatók:

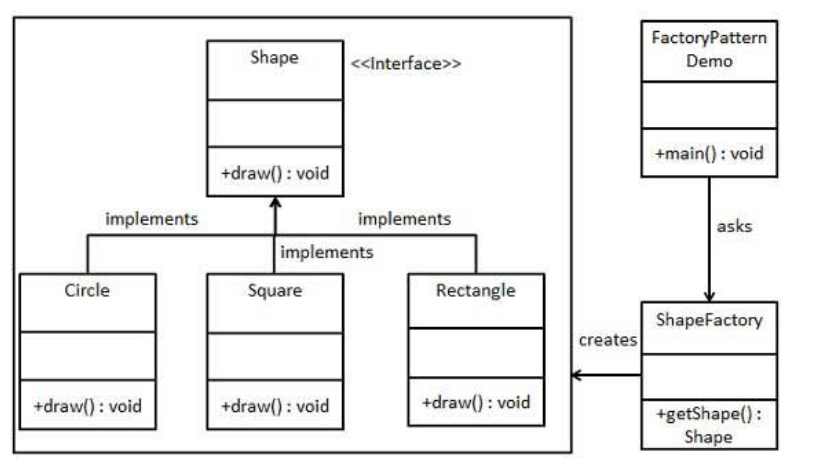
- **Létrehozási minta:** Megoldást biztosít, hogy objektumokat hozzunk létre, miközben elrejti a létrehozási logikát, tehát nem kell közvetlenül létrehoznunk objektumokat a new operátorral.

- **Szerkezeti minta:** A szerkezeti minták lényege az osztályokból és objektumokból nagyobb szerkezetek létrehozása. Amennyiben osztályokkal dolgozunk, örökléssel vagy felületek megvalósításával összetételeket hozunk létre, például több interfész implementálásával egy többféleképpen is viselkedő osztályt. Objektumok esetében dinamikus „összeragasztásukkal” érjük el az olyan összetételek létrejöttét, amelyek kellően rugalmasak.

- **Viselkedési minta:** Objektumok kommunikációjának megoldásait hordozzák magukban.

**Gyártó minta (Factory Pattern):** Ez a leggyakrabban használt tervezési minta Java nyelven. A Létrehozási minták csoportjába tartozik. Ez a minta biztosítja az egyik legjobb módját az objektumok létrehozásának.

A gyártó mintában anélkül hozunk létre objektumokat, hogy a kliens számára elrejtjük a létrehozási logikát (new példányosítást).



A lényeg a következő: Van egy Factory osztály mely, egy interfészen keresztül visszaad példányokat. Definiál egy Shape interfacet, mely interfacet implementál a három fent látható alakzat. A getShape(String) metódusnak paraméterként átadjuk az alakzat nevét, amit létre akarunk hozni.

public interface Shape {

void draw();

}

Az alábbi módon létrehozzuk a többi alakzat osztályt, amely implementálja a Shape interfacet.

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

A következő lépésben a gyártó osztályt hozom létre:

public class ShapeFactory {

//a getShape metódust használjuk arra, hogy Shape-t implementáló osztályt adjunk vissza.

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

Az equalsIgnoreCase() egy olyan metódus ami a Stringek értékét úgy vizsgálja, hogy közben a kis-nagybetű érzékenységre nem figyel.

System.out.println("alma".equalsIgnoreCase("aLMa")); 🡪 Ennek az eredménye true;

Az alábbi megoldásban már a gyártó függvényt használjuk, úgy , hogy egy információt adunk át paraméterként, és ő előállítja nekünk a felparaméterezett objektumot, és vissza is adja.

Shape shape1 = shapeFactory.getShape("CIRCLE");

**- Absztrakt gyártó minta (Abstract Factory Pattern):** Az Abstract Factory lényege, hogy a gyártó szolgáltatásokat is egy gyártó szolgáltatás adja. Így egy Gyártó gyártót hozunk lényegében létre, ami a gyártókat hozza létre, azok pedig már legyártják a rájuk vonatkozókat.

**-Singleton Pattern(Egyke Pattern):**Ennek az a lényege, hogy egy osztály saját magából csak egyetlen példányt enged létrehozni. Ennek megvalósítása, hogy egy statikus referenciája van saját magára, ami nyilván private, illetve a konstruktor is private , és Gyártófügvényben ezt az egy példányt adja mindig vissza annak, aki elkéri.

**-Builder Pattern(Építő Pattern):** Ennek az a lényege, hogy amiket le akar gyártani azokat értelem szerűen interfacen keresztül lehet elérni. Listába belepakolja a példányokat, majd a factory függvényét meghívva visszaadja a lértehozott és bepakolt elemeket.

**-Prototype Pattern(Prototípus Pattern):**Még ez is létrehozási minta.

**When Would I Use This Pattern?**

The Prototype pattern should be considered when

* Composition, creation and representation of objects should be decoupled from the system
* Classes to be created are specified at runtime
* You need to hide the complexity of creating new instance from the client
* Creating an object is an expensive operation and it would be more efficient to copy an object.
* Objects are required that are similar to existing objects.

**-Adapter Pattern():** Ez már a szerkezeti minták körébe tartozik.

Az OOP programozás egyik legnagyobb előnye az újrafelhasználás. Olyan osztályok, komponenseket használhatunk fel többször, melyeket akár nem is mi írtunk. Azonban ez nem mindig ilyen egyszerű, gondoljunk bele, hogy egy korábban megírt osztályt szeretnénk felhasználni, viszont annak az interfésze melyen keresztül elérjük, az nem kompatibilis a többi általam írt osztályéval. Írjam át az összes osztályomat úgy, hogy kompatibilis legyen vele?(Itt az átírás nem metódusnév átírás, hanem strukturális átalakítások.). Oké, akkor inkább változtassuk meg az inkompatibilis osztályt, amit használni akarunk? Ez sem jó megoldás, mivel lehet , hogy nem férünk hozzá a forráskódhoz, vagy annyira bonyolult a szerkezete, hogy nem lenne értelme neki látni értelmezni, sőt valószínűleg azért szeretnénk felhasználni, hogy ne nekünk kelljen megírni azt a funkciót.

Erre megoldás az Adapter tervezési minta. Egy olyan adapter osztályt fogunk alkalmazni, ami az elvárt interfészt nyújtja az alkalmazásom felé, ugyanakkor az inkompatibilis osztály eljárásait implementálja. (Azaz megfelelő formában meghívja a metódusait, esetleg átalakításokat végez, majd továbbítja az általam írt kliens számára.). Ez lényegében egy átalakító (Adapter).

Röviden: Lefordítja az egyik osztály interfészét egy kompatibilis másik interfészre

Amennyiben a felhasználni kívánt osztály(inkompatibilis osztály) semmilyen formában nem támogatná valamelyik műveletet, amelyre a kliensnek szüksége van akkor azt esetleg implementálhatjuk magunk is az adapterben, de persze csak csínján bánjunk ezzel a megoldással: ha azt vesszük észre hogy túl sok hiányzó funkcionalitást kell így saját kezűleg pótolnunk akkor élhetünk a gyanúperrel hogy nem is a megfelelő osztály választottuk ki a céljainkhoz.

Két féle megvalósítása van az Adapter tervezési mintának:

\* Osztály szintű megvalósítás

\* Objektum szintű megvalósítás

Osztály szintű adapter minta: Ez a többszörös öröklődésen alapszik. Javaban nincs többszörös öröklődés, viszont az interfészekkel hasonló funkciót érhetünk el (Interfészeket azért jó használni, mert egy osztály több interfészt implementálhat, így több viselkedést is magába foglalhat 🡪 Többszörös öröklődés-szerű mechanizmus). Az Adapter osztály a külsö inkompatibilis osztály leszármazottja lesz, így minden metódusát ismeri, és azoknak megvalósítását is, ha az Adaptert példányosítjuk, akkor kezelhető lesz, mint a külső osztály egy példánya(Az adapter a külső inkompatibilis osztály példánya). Ez viszont nem elég, hanem implementálnia kell az Adapter osztálynak azt az interfacet, ami közös az én általam írt programmal, melyen keresztül már tudnak kommunikálni.

Ahogy az látszik az Adapter tervezési mintában alapvetően négy résztvevőt kell tudnunk megkülönböztetni:

\* a kliens osztályt, "aki" az egésznek a haszonélvezője, ő használja fel az interfész segítségével a többi osztályt;

\*a közös interfészt, amit mindenkinek implementálnia kell;

\*az adapter osztályt, aki az átalakítást végzi;

\*az átalakítandó "külső" osztályt, aki tartalmazza azokat a roppant bonyolult algoritmusokat, amiket magunktól lusták lettünk volna (újra) lekódolni.

- Objektum szintű adapter minta: Az objektum szintű megvalósításban az Adapter osztály nem leszármazik a külső idegen osztályból, hanem példányt tartalmaz belőle adattag szinten. Tehát itt az Adapter tartalmaz egy külső osztály példányt, és azzal kommunikál, és az eredményt visszaadja a közös interfacen keresztül.(Ez lényegében a kompozíció). Itt jön ki az objektumszintű megoldás előnye, ugyanis egy interfész alá, akár több külső osztály funkcionalitását betehetjük. Itt elég mindegyikből példányt létrehozni, viszont osztály szintűnél az a probléma, hogy leszármazni csak egy osztályból tud. Hasonlóan lehetővé válik, hogy ne csak az eredeti külső osztályt, hanem annak alosztályait is adaptáljuk a közös interfész alá.

Ebből a szempontból tehát az "objektumos" változat egyértelműen jobbnak tűnik, de azért a másiknak is meg vannak a maga előnyei: szükség esetén jóval egyszerűbben felül lehet írni a külső osztály viselkedésének bizonyos elemeit. Megfontolandó továbbá hogy az osztály alapú megvalósításnál csupán egy objektum példányosodik, míg az objektum szintűnél legalább kettő, és a memóriahivatkozások lánca melyen "keresztül" eljut az kérés és az eredmény a feladótól a címzetthez hosszabb. Ez persze a rendszererőforrásokkal kevésbé takarékos (esetenként lassabb) programot eredményez.

Szemléltetés példán keresztül: Ha a külső osztály(inkompatibilis osztály) azt csinálja, hogy híváskor egy flaget ad vissza valamilyen logika alapján. Ha híváskor az idő órában nézve páros, akkor egyest, ha páratlant akkor nullát ad vissza. Ez viszont az én kliens alkalmazásomnak nem jó mert én azt szeretném kiíratni, hogy true vagy false a consolra. Ilyenkor jön a képbe az Adapter osztály, ami átalakítja a külső osztály inputját (0-1), már az én kliensem számára is értelmezhető formátumra(true-false). A kliensemnek csak annyi a dolga, hogy megjelenítgeti az információkat.

Megvalósítás:

A Kliens megvalósítása:

**public** **class** Client {

/\*Ez az osztály ismeri azt az interfacet, amin keresztül ő alérhet egy adott funkciót.

Itt a funkció, hogy valamilyen logika alapján kap egy booleant, és azt kiírja.\*/

**public** **static** **void** main(String[] args) {

/\*Ez a metódus ami kiír egy booleant, és egy interface implementációt vár, hisz ennek az interfacenak ismeri a metódusait , formátumát.\*/

*kiir*(**new** Adapter());

}

/\*A közös interfacet megvalósító elemet vár paraméterként, majd annak meghívja a szolgáltatását.\*/

**public** **static** **void** kiir(KozosIsmertInterface inter){

System.***out***.println(inter.getBoolean());

}

}

A közös interface megvalósítása:

**public** **interface** KozosIsmertInterface {

/\*Ezt ismeri az Adapter és a kliens is.\*/

**public** **boolean** getBoolean();

}

A külső osztály megvalósítása:

**public** **class** KulsoIdegenSzamlalo {

Date date;

**public** KulsoIdegenSzamlalo() {

}

/\*Ez a metódus, ami nem kompatibilis a kliensemmel.Ez számot ad vissza, nekem pedig boolean kellene.\*/

**public** **int** getNumber(){

date = **new** Date();

**if**(date.~~getMinutes~~() % 2 == 0){

**return** 1;

}

**return** 0;

}

}

Az Adapter osztály megvalósítása:

**public** **class** Adapter **extends** KulsoIdegenSzamlalo **implements** KozosIsmertInterface{

/\*Leszármazik a külső osztályból, és implementálja a közösen ismert interfacet, ami

a kliens és az Adapter közötti kommunikációt segíti elő.Itt a közös nyelv a getBoolean

metódus, amiben az adapter átalakítja a számértéket booleanra, és azzal tér vissza.\*/

@Override

**public** **boolean** getBoolean() {

**if**(getNumber() == 1){

**return** **true**;

}

**return** **false**;

}

}

Látható, hogy a megvalósítás nem nehéz. Ha objektum alapú megközelítés lenne, akkor nem leszármazna az Adapter, hanem példánya lenne a külső osztályra, és annak hívná a getNumber() metódusát.

**Bridge Pattern(Híd Pattern):**