Kötelező esszé: példákon keresztül, ábrákkal bemutatni a CC metriket egy legalább 2 oldalas esszében. Nem kézzel írni. Alapbeállításokkal

Szoftverkrízis

PPP(Design Patterns, Principles, Good Pratices)
Szoftverkrízis van, a programok nem fejeződnek be időben. Amíg a szoftverkrízis van addig elégedetlenek az ügyfelek, ezért jobb cégekért kutatnak a rendelők. Emiatt sokat fizető munka a programozás.

A szoftverkrízis miatt vannak jó fejlesztő környezeteink, ezért van a PPP

Alapelveink

A szoftver kódja mindig változik (pl. angular).

- A rendelő változtatja a követelményeit
- Hibák javítása
- Kód szépítése
- Környezet változása

Emiatt kerülni kell a **fragile** kódot, mivel az nem tartós, azt cserléni kell. (más szóval büdös, spaghetti kód)

There is no silver bullet

nincs ezüst golyó
A szoftver technológiának nincs egy üdvözítő megoldása ami legyőzné a szoftver krízist. Szoftvertechnológiának nincs Szent Gralja

80/20-as alapelv

a szoftverfejleszség első 80%-a könnyű, az utolsó 20%-a nehéz

Separation of concerns

Amit szét lehet választani azt érdemes széválasztani, mert csökken a komplexitás.

The mystical man month

Egy késésben lévő projekthez további embereket hozzáadni csak további gondot okoz.
2 ember közt 1 kommunikációs csatorna, 3 ember közt már 3, 4 ember közt 6
Azaz n ember közt $n * (n-1) * 2$ → a kommunikációs utak száma négyzetesen nő

Megoldások:

- Vezető fejlesztő
- Daily SCRUM
  - Mit csináltál előző nap
  - Mit fogsz ma csinálni
  - Milyen akadályokba ütköztél

A szofter is csak egy termék, mint egy doboz

van ára, minősége, forgalma, célközönsége

Obejktum orientált tervezési alapelvek

SOLID alapelvek
Agile manifesto:

- Elfogadjuk, hogy elfogadjuk a változást
- Akár mi kezdeményezzük azt

SRP (Single responsibility principle)

Minden osztálynak, modulnak csak 1 oka legyen változásra. Ne legyen macskuty osztály. Minden osztály csak 1 felelősséget vegyen fel, de azt tisztességesen.

OCP (Open-closed principle/Nyitva zárt alapelv)

Az osztályok legyenek nyitottak a bővítésre de zártak a módosításra, azaz ha már van egy jól kitervelt, tesztelt metódusom, akkor azt ne módosítsuk, mert a többi metódus ami ettől függ, is meg kell változtatni.

LSP (Liskov-féle behelyettesítési alapelv/Liskov substitution principle)

Azt mondja ki, hogy a gyermekosztály azaz ősosztály altípusa legyen, a gyermek tartsa be az ős szerződését. A program viselkedése ne változzon meg, ha ősosztály helyett alosztályt használunk.

ISP (Interface seggregation principle)

Az interface-k max 1 metódusból álljanak. Nagy osztályok sok külön interface-t használjakan. Ha egy interfacet átírunk akkor az összes azt használó osztályt újra kell fordítani.

DIP (Dependency inversion principle/függőség megfordításaának alapelve)

Ne alacsony szintűtűl függjön magas szintű dependencia, hanem absztakciótól függjön. Ha UML ábrára felrajolunk egy osztályt akkor interfacetől függjön.

Gof I

gang of four

- Programozz felületre megvalósítás helyett (program to an interface, not an implementation)

Interface-re programozni annyit jelent, hogy a többi osztálynak csak a felületét ismeret, nem tudjuk belül hogy van implementálva.

```
Kutya k1 = new VadaszKutya();
```

`VadaszKutya k2 = new VadaszKutya();`
`Kutya k3 = new Kutya();`
(Kutya ősosztály, vadászkutya gyermekosztály)
Ezek közül az 1. betartja a **GOF I**-et, a második nem, a harmadik sor valószínűleg absztrakt akkor a harmadik nem is lehetne.

GOF I szerint azt a legősibb osztályt kell típusként használni aminek a felülete él.
Ha csak equals-re van szükségem akkor mégősibb osztályt kell használni, ilyenkor `object k4 = new VadászKutya();`

Gof II

Öröklődés helyett használj objektum összetételt, ha csak lehet. (Favor object composition over class inheritance.)

Osztályt egy téglalappal jelöljük, abba írjuk bele az osztály nevét.

IS-A kapcsolat helyett használj **HAS-A** kapcsolatot ha csak lehet.

IS-A

- 2 fajtája van:
  - Öröklődés (extend)
  - Interface implementáció (implements)
  Minden amivel típust kapsz az IS-A kapcsolat
  Fordításkor jön létre

HAS-A

Rombusz a nyíl végén

- 4 fajtája van:
  - Kompozíció (besatírozott) (repülő szárnya)
  - Aggregáció (gitáros gitárja) (gitáros ← gitár)
  - Barátság (szaggatott - create)
  - Barátság (sima nyíl - átmeneti)
  Kizárólagos tulajdonság: Kompozíció (garbage collector együtt szabadítja fel a 2 osztályt)
- Több referencia: Aggregáció
- Nem egyből van referencia: create
- Mezőben tárolt: barátság
- Futás időben jön létre, és változtatható

Példák:

> 01 Peldak
>
> # Gerinces
>
> Példa 1:
>
> ```
> Class Gerinces{
>     pulbic String fut (){
>         return "fut";
>     }
> }
>
> Class Kutya extends Gerinces{
>     public String GyorsanFut(){
>         return "gyorsan" + fut()
>     }
> }
>
> Main:
>     Kutya k1 = new Kutya();
>     System.out.println(GyorsanFut());
>
> > "Gyorsan fut"
> ```
>
> Példa 2:
>
> ```
> Class Gerinces{
>     Public String Fut(){
>         return "fut";
>     }
> }
>
> Class Kutya{
> ```

```
    Gerinces gerinc = new Gerinces();
    public String GyorsanFut(){
        return "gyorsan" + gerinc.Fut();
    }
}

Main:
    Kutya k1 = new Kutya();
    System.out.println(GyorsanFut());

> "Gyorsan fut"
```

Kacsa

01

```
Class Kacsa{
    primal static int HAZIKACSA = 1;
    primal static int NEMAKACSA = 2;
    primal static int GUMIKACSA = 3;
    int tipus;

    Public Kacsa(int tipus){
        this.tipus = tipus
    }

    public string Hapog(){
        if(tipus == HAZIKACSA){
            return "Hap hap";
        }
        else if(tipus == NEMAKACSA){
            return "";
        }
        else if(tipus == GUMIKACSA){
            return "sip sip";
        }
        else{
            throw new Exception("Rossz tipus")
        }
    }
}
```

02

```
interface HapogasiStrategia{ //Stateless osztály, nincs belső állapota
    string Hapog();
}
```

```
interface RepulesiStrategia{
    string Repul();
}

Class SimaHapogas implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "Hap hap";
    }

}
Class NemHapog implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "";
    }
}
Class RekedtenHapog implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "hjäp hjäp";
    }
}

Class Repules implements RepulesiStrategia{
    @Override
    public String Repul(){
        return "ég";
    }
}
Class NemRepul implements RepulesiStrategia{
    @Override
    public String Repul(){
        return "föld";
    }
}

Class Kacsa /* implements HapogasiStrategia */{
    HapogasiStrategia hs; //HAS-A kapcsolat
    RepulesiStrategia rs;
    public void SetHapogasiStrategia(HapogasiStrategia hs){
        this.hs = hs; //injection
    }
    public void SetRepulesiStrategia( RepulesiStrategia rs){
        this.rs = rs; //injection
    }
    public Kacsa(HapogasiStrategia hs, RepulesiStrategia rs){
        this.hs = hs; //Dependency injection
        this.rs = rs;
```

```
    }
    public string Hapog(){
        return hs.Hapog(); // felelősség átadás, delegation
    }
    public string Repul(){
        return rs.Repul();
    }
}
```

```
Kacsa k1 = new Kacsa(new NemHapog());
System.out.println(k1.Hapog());
//Console:
k1.SetHapogasiStrategia(new SimaHapogas());
System.out.println(k1.Hapog());
//Console: Hap Hap
```

3.

```
abstract class ForroItalFozes{
    public void Foz(){}; //Kötelező közös lépés
    private void VizetForralok(){
    System.out.println("Vizet forralok");
    }

    protected abstract void KoffeinBele(); //Azert abstract mert kötelező kifejteni
    //Inversion of control
    protected void Izesites(){}; //a hooknak van törzse de ez üres
    //Inversion of control
}



Class RumosKaveFozes extends ForroItalFozes{
    @Override
    protected void KoffeinBele(){
        System.out.println("Forro vizbe instant kave port keverek");
    }

    protected void Izesites(){
        System.out.println("rumot teszek bele");
    }
}

Class TeaFozes extends ForroItalFozes{
    @Override
    protected void KoffeinBele(){
        System.out.println("Teafiltert logatok a vizbe");
    }
```

```
}
```

Példa 1-nél öröklődést használtam, azaz IS-A kapcsolatot. Példa 2-ben objektum összetételt használtam, azaz HAS-A kapcsolatot. A példa azt mutatja, hogy az IS-A kapcsolat kiváltható HAS-A kapcsolattal a program lefutásának változása nélkül

Mi a referencia neve ami megvalósítja a kapcsolatot? A `gerinc` referencia. Ezen a referencián keresztül érem el a fut metódust. Egyetlen dolgot nem ad a HAS-A amit az IS-A. A példa 2-ben a kutya nem gerinces. Ha szükségem van a kapcsolatra akkor IS-A kapcsolat a helyes, de ha nem akkor HAS-A.

A példa 2-ben kompozíciót használunk.

LSP

Ez azt mondja ki hoyg a gyermekosztály tartsa be a szerződéseit.
Design by contract/zerződés alapú programozás

Minden osztálynak van elő és utófeltétele, minden osztálynak van ...

- Bank account példa
  - sosem mehetek minúszba
  - `double balance >= 0`
    Ez azt jelenti, hogy minden metódús hívás előtt és után igaznak kell lennie.

A metódus előfeltétele a metódus bemenetétől függ, az utófeltétel epdig a return-től függ.

Pl.: bináris keresés

> meg kell egyezni a return értékre ha nem lenne a listában a keresett érték (pl. -1)

Best practices/OCP

OCP

Meglévő, kitesztelsz metódust az OCP szerint tilos. Szintaxis szintjén azt jelenti, ne használj override kulcsszót cask abstract vagy hook(kampó/akatsztó) metódus fölülírására. A hook metódust a template metódus tervezési mintájánál fogjuk megtanulni.

OCP szintaxis szintjén ne használj if, elsi/if szerkezetet (beleértve a switchet is). Mert van egy fő alapelvünk, a program kódja állandóan változik. A fő alapelv miatt lesz egy

új if else ágad, ami ahbár az if szempontjából bővítés de metódus szempontjából módosítás, és ezt nem engedi az OCP.

Kitesztelt metódus: van sok unit teszt hozzá. Minden metódusnak van CC száma, ami **Cyclomatic complexity**. Minden lehetséges lefutásra 1 unit teszt (Annyi unit teszt amennyi CC szám) . A for *ciklus* is egy *if*. A unit teszt úgy működik mint az életbiztosítás. Ha a kód megváltozik akkor kell regressziós tesztet csinálni. Ez csak annyit jelent hogy az összes unit-tesztet újrafuttatod.

Hoare-logic: egy utasításnak mi az elő és utófeltétele.

Ha nincs unit-teszt akkor elkezd rothadni a kód. Ez annyit jelent, hogy el kezdek hibákat javítani és ez további hibákat hoz, és emiatt nem mersz hozzányúlni a kódhoz.

Másik filozófia unit-teszthez: Itt a unit-teszt a specifikáció. Nem írok követelmény spec.-et vagy hasonlókat, csak unit tesztet. Ez a best-practice a TDD (Test Driven Development). Másnevén piros-zöld-piros.
Működése: először nem az üzemi kódot (metódust) írom, hanem a unit tesztet, ami leírja a metódus működését. Ez piros lesz mert a metódust még nem írtam meg. (Piros = failed unit teszt). A metódusból csak annyit jrunk meg hogy a unit teszt zöld legyen. Ezután írunk új unit-tesztet ami úgy meg-nem írt metódus ágra megy. Ez a unit teszt piros lesz, megíratlan metódus miatt, és ezután megírom a metódus annyira hogy zöld legyen a unit-teszt. Ezt tovább folytatjuk. Ha a class nem publikus akkor nem tesztelhető.
Ezt a módszert érdemes használni.

Unit teszt felépítése

Mindig van egy target, vannak benne lefixált paraméterek és van egy elvárt visszatérési érték. és egy actual visszatérési érték. Az expectedet előre megmondom, az actualt meg összehasonlítom azzal.

Beugró lehet: csináljunk egy getalma metódust, double kiló, ár bemenet: kiló, kimenet mennyibe kerül, 1kg ára

1. 10%, 5kg estén, 10kg-nál 15% kedvezmény
   3kg → 600ft
   target.getalmaár(kg)
   Assert = (actual,expected)
   java-ban JUnit a bevett framwork, dotnetben van beépített

2. 5kg: 900

3. 15kg

4. 0kg

5. -6kg

Pair-programming

Egy gépnél 2 programozó, az egyik írja, a másik figyeli a kódot, és ezt megbeszélik. Ezzel gyorsan észreveszünk hibát, és minél hamarabb vesszük észre annál olcsóbb javítani.
A tapasztalt programozó fel tud húzni egy juniort egy magasabb szintre páros programozással.
Ha az egyik programozóval történik valami akkor a másik tudja helyettesíteni. Ez a kockázamenedzsment egyik része.

Code Review

A vezér programozó megnézi a kódodat. Az egész csapatnak mutatod mega kódodat, és azok elfogadják vagy elutasítják. A code review legmagasabb foka amikor egy külső szakember megnézi a kódot, erre csak akkor hagyatkozunk amikor a kód működik de lassú.

Behaviour: ha egy új else-if ágat hozzáadsz akkor új viselkedést adsz hozzá. Az új viselkedést érdemes egy új alosztályban egységbe zárni.

Az alagzat példa:

> c03

```
Class Alagzat
{
    final static int HAROMSZOG = 1;
    final static int TEGLALAP = 2;
    final static int NEGYZET = 3;
    double a,b,c,d;
    int tipus;
    public void setA(double a){
        this.a = a;
    }
    public void setB(double a){
        this.a = a;
    }
    public void setc(double a){
        this.a = a;
    }
    public void setd(double a){
```

```
            this.a = a;
        }

        public double GetKerulet(){
            if(tipus == HAROMSZOG){
                return a+b+c;
            }
            else if(tipus == TEGLALAP){
                return 2*(a+b);
            }
            else if(tipus == NEGYZET){
                return 4*(a);
            }
            else{
                throw new Exception("Ismeretlen típus");
            }
        }

        public double GetTerulet(){
            if(tipus == haromszog){
                return Heron(a,b,c);
            }
            else if(tipus == TEGLALAP){
                return a*b;
            }
            else if(tipus == NEGYZET){
                return a*a;
            }
            else{
                throw new Exception("Ismeretlen típus");
            }
        }
    }
```

Megoldás: abstract osztály vagy interface

```
public abstract class Alagzat{
    public abstract double GetKerulet();
    public abstract double GetTerulet();
}

public class Haromszog extends Alagzat{
    double a,b,c
    public Haromszog(double a, double b, double c){
        this.a = a;
        this.b = b;
        this.c = c;
    }
    @Override
```

```
    public double GetKerulet(){
        return a+b+c;
    }
    # Mi lesz ha bejön a trapéz?
    # Létrehozunk egy új aloszályt ami egységbe zárja a trapéz viselkedélsét
}
```

OCP kötelező tétel. Tökéletesen kell tudni

Template metódus

stratégia és sablon tervezési minták

Terv minták gyakran előforduló programozási feladatokhoz vannak, rugalmas, könnyen bővíthető, újrahasználható megoldás

STP (Strategy)

Alap probléma: van egy osztályom és abban egy változékony metódus. Egy webshop esetén egy GetPrice metódus változékony az akciók miatt. Ha van egy ilyen, akkor a változékony metódust érdemes kiemelni stratégiába, van egy ősosztály(interface) amiben egyetlen 1 metódus van, és a gyermekosztályok a különböző metódusokat zárják egységbe. Onnan ahonnan kiemeljük kell egy objektum összetétel, azaz referencia, amin keresztül hívni fogjuk a kielemt metódust.

01

```
Class Kacsa{
    primal static int HAZIKACSA = 1;
    primal static int NEMAKACSA = 2;
    primal static int GUMIKACSA = 3;
    int tipus;

    Public Kacsa(int tipus){
        this.tipus = tipus
    }

    public string Hapog(){
        if(tipus == HAZIKACSA){
            return "Hap hap";
        }
        else if(tipus == NEMAKACSA){
            return "";
        }
        else if(tipus == GUMIKACSA){
            return "sip sip";
```

```
        }
        else{
            throw new Exception("Rossz tipus")
        }
    }
}
```

Stratégiai metódus

02

```
interface HapogasiStrategia{ //Stateless osztály, nincs belső állapota
    string Hapog();
}
interface RepulesiStrategia{
    string Repul();
}

Class SimaHapogas implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "Hap hap";
    }

}
Class NemHapog implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "";
    }
}
Class RekedtenHapog implements HapogasiStrategia{
    @Override
    public String Hapog(){
        return "hjäp hjäp";
    }
}

Class Repules implements RepulesiStrategia{
    @Override
    public String Repul(){
        return "ég";
    }
}
Class NemRepul implements RepulesiStrategia{
    @Override
    public String Repul(){
```

```
            return "föld";
        }
    }

    Class Kacsa /* implements HapogasiStrategia */{
        HapogasiStrategia hs; //HAS-A kapcsolat
        RepulesiStrategia rs;
        public void SetHapogasiStrategia(HapogasiStrategia hs){
            this.hs = hs; //injection
        }
        public void SetRepulesiStrategia( RepulesiStrategia rs){
            this.rs = rs; //injection
        }
        public Kacsa(HapogasiStrategia hs, RepulesiStrategia rs){
            this.hs = hs; //Dependency injection
            this.rs = rs;
        }
        public string Hapog(){
            return hs.Hapog(); // felelősség átadás, delegation
        }
        public string Repul(){
            return rs.Repul();
        }
    }
```

```
Kacsa k1 = new Kacsa(new NemHapog());
System.out.println(k1.Hapog());
//Console:
k1.SetHapogasiStrategia(new SimaHapogas());
System.out.println(k1.Hapog());
//Console: Hap Hap
```

A HAS-A kapcsolatot megvalósító referencia neve hs.
Dependency injection - felelősség injektalasa amikor a HAS-A kapcsolat.

Ha van két kacsám és mindkettő hápog, akkor ismételni kell a kódot

Jó megoldás: multiton

Template method

Stratégiát akkor használok amikor ugyanakkor használok de másképp. Template method: ugyanazt csinálom de más eredménnyel.

Template methodnak közeli rokona a *factory method*. Template methodnál nincs megkötés mikor hívom, factory methodnál van ajánlás.

Rögzítem az algo lépéseit, de nem mondom meg mit csinálnak, csak a sorrendjét. Egy algoritmus vázat alkotok meg. A lépések lehetnek kötelező lépések, ebből is kétfajta van:

- Kötelező közös
- Kötelező nem közös
  Emellett még van az opcionális.

Kötelező közös lépéseket ősben kifejtem és ezek private-ok .
Kötelező nem közöst a gyermek fogja kifejteni, ezek protected abstract-ok.
Az opcionálisakat a gyermek vagy kifejti vagy nem. Ezek **hook** metódusok. Ezeknek van törzse de a törzse üres. Ha a hook metódus nem üres akkor muszály return de olyat ad vissza ami nem befolyásol. (pl. nullát)

1 public metódusa van, a **sablon metódus**.

A receptben kell felsorolni lépéseket.

IOC (inversion of control): amikor nem a gyermek hívja az ősét, hanem az ős hívja a gyermekét. Az ős úgy tud hívni a gyermekből, hogy olyan metódust hív ami még benne abstract.

Miért nem lehet a Vizforralas public-us? így zároljuk a lépések sorrendjét.

> # 3.
>
> ```
> abstract class ForroItalFozes{
>     public void Foz(){}; //Kötelező közös lépés
>     private void VizetForralok(){
>     System.out.println("Vizet forralok");
>     }
>
>     protected abstract void KoffeinBele(); //Azert abstract mert kötelező kifejteni
>     //Inversion of control
>     protected void Izesites(){}; //a hooknak van törzse de ez üres
>     //Inversion of control
> }
>
>
>
> Class RumosKaveFozes extends ForroItalFozes{
>     @Override
>     protected void KoffeinBele(){
>         System.out.println("Forro vizbe instant kave port keverek");
>     }
>
>     protected void Izesites(){
> ```

```
            System.out.println("rumot teszek bele");
        }
}

Class TeaFozes extends ForroItalFozes{
        @Override
        protected void KoffeinBele(){
            System.out.println("Teafiltert logatok a vizbe");
        }

}
```

Valamit

Rossz

```
public static void Main(String[]args){
        Shape s1=new Shape(Shape.Square); //eredeti döntést
        s1.setA(4);
        sout(s1.getArea()); //->16
}

class Shape{
        publix static finals int Square = 1;
        //...

        //...
        private int a,b,c,d;
        private int type;
        public Shape(int type){
            this.type = type; //eredeti döntést konzerválom
        }
        public int getArea(){
            if(type == square) //ismételt döntés
                return a*a
            if (type == rectangle)
                return a*b
        }

}
```

Jó

```
public static void Main(String[]args){
        Shape s1=new Square(Shape.Square); // eredeti döntés, s1-ben konzerválom
        sout(s1.getArea()); //->16 //ismtételet döntés
```

```
        //polymorphic method call
    }
    abstract class Shape{
        public abstract int getArea();
    }
    class Square extends Shape{
        int a;
        public Suqare(int a) {this.a = a}
        @Override
        public int getArea() {return a*a}
    }

    class Rectangle extends Shape{
        int a,b;
        public Suqare(int a, int b) {
        this.a = a;
        this.b = b;
        }
        @Override
        public int getArea() {return a*b}
    }
```

A rossz példában van if, a jó példában nincs, hogy lehetséges?
Eredeti döntés: amikor eldöntöm melyik viselkesdésre van szükség
Ismtélelt döntés: mikor újra mek kell hoznom ugyanazt a döntést amit már egyszer meghoztam, hogy a megfelelő viselkedéssel bírjon a progarmom

Késői kötés: late binding
Minden polymorphic metohod mögött egy ismételt dönts van, amit a késői kötés hoz meg
A java a késői kötéshez rendel valószínűséget, ami valószínű azt pipelineba rakja, ezért gyorsabb lesz a kód.
Amiben van garbage collector(C#, Java), abban nem lehet realtime rendszert írni(pl atomerőművet futtatni)

Flyweight

pehelysúlyú tervezési minta
Szövegszerkesztőt írni objektum-orientáltan nem egyértelmű, mert minden betűt külön objektumként lehet használni. Ez egy természetes ötlet, mivel minden betűnek van színe, stílusa, mérete, van pár flag(pl. *italic*, **bold**)
. Minden betűnek van környezete, és a mérete, stílusa a környezetétől függ. Egy dolog van ami nem függ a környezettől, az pedig a karakter ASCII kódja.

AlmA A fA AlAtt → első A betű az új objektum, de a többi A viszont már referencia.
A kiíratás függa környezetről. Az objektum viselkedése függ a belső állapotból, a belső

állapot a mező. The behaviour of an object depends on it's inner state, which is the momentary value of it's attributes. Úgy tünik hogy a viselkedés nem csak a belső hanem a külső állapottúl is függ, amit kontextusnak nevezünk. Ennekj több szintje van, pl.: betű→szó→mondat→bekezdés→szöveg

Hogyan kapom meg a kontextust?
Metódusok paraméterével kapom meg. A betűkiír metódusnak egy paramétere van, a kontextus. A kontextusok egymásba vannak ágyazva

Method proxy

proxy → helyettes
proxy cache server, gyorsítótár
Gábor és karcsi ugyanazt a könyvet akarja letölteni. Amerikai szerveren van tárolva, de amikor Gábor letölti akkor egy európai proxy cache szerveren lesz egy ideig, és amikor karcsi letölti neki gyorsabban és olcsóbban letölti

Proxy-t akkor használunk ha valami drága, érzékeny pbjektumhoz szabályozott hozzáférést biztosítsunk . Programozás technikailag átlátszó becsomagolást használunk, hiszen ha rendelkezik jogosultsággal, erőforrásssal, akkro nem is veszem észre hogy proxyn keresztül férek hozzá az objektumhoz

A method proxy metódusokat segít elérni.

Kép proxy

A kép proxyja csak annyit tud hogy mekkora a kép.
Pl. wordben a docx az egy zip file, a proxy csak annyit tud hol van a zipben a kép

Díszítő tervezési minta (decorator)

Alap probléma: szeretnék karácsonyfát, szeretnék gömbös karácsonyfát, szeretnék szalon cukros karácsony fát, szeretnék egy egy csillag díszes karácsonyfát, és ezek kombinációját. Ha mindegyiknek szeretnék osztályt akkor rengeteg osztály lenne. Not enough DRY. Do not repeat yourself. Legegyszerűbb az ismétlődő kódrészletet kiemelni metódusba, vagy alosztályba

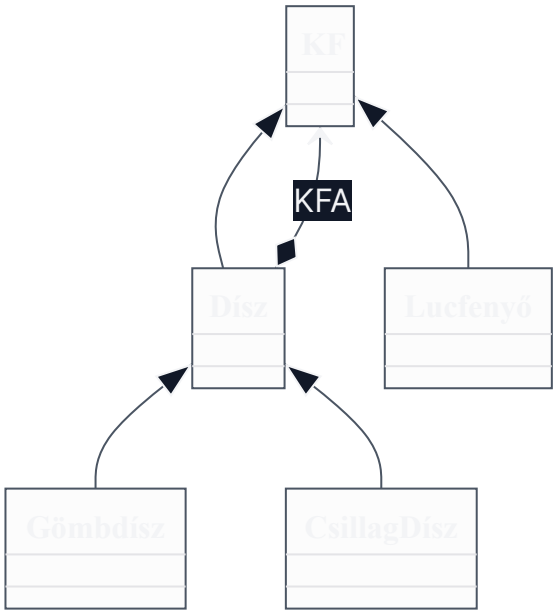
> # Rossz megoldás
>
> ```
> class KF{
>     @Override
>     public string ToString(){
>         return "Lucfenyő"
>     }
>     public double getÁr() {return 5200}
> ```

```
}
class GömbösKF extends KF{
      public string ToString(){
            return "Gömbös lucfenyő"
      }
      public double getÁr() {return 5200 + 1200}
}
class CsillagosKF extends KF{
 ...
}
...
```

Itt sokat kell másolni ezért ez nem eléggé DRY

OCP-t megszegtük, mert konkrét metódust írtam fölül overridal. Az öröklődés az egy white box reuse.

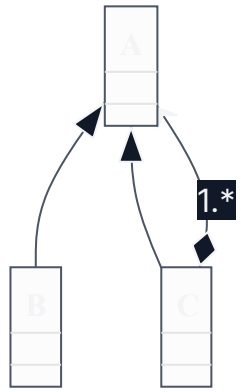
Felépítési minta



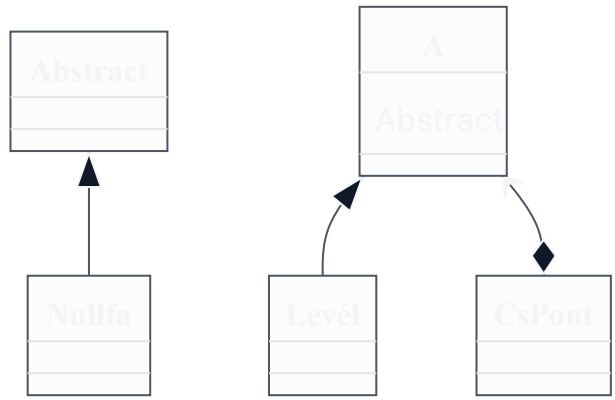
A díszítőfa ötlete az, hogy a diszítés előtt és után is karácsonya farad. Ez az átlászó becsomagolás (wrapping). Nem átlátszó csomagolási minta: pl adapter

A proxy tervezési minta is átlátszó becsomagolás is.

Composite



Binfa



HF: rossz kód kijavítása.

Observer tervezési minta(megfigyelő)

MVC(model view controller) tervezési minta
Pl. van a tőzsde am kb 2 percet késik, és van a a cégeké ami real-time
A control az amikor egy esemény történik és az befolyásolja a
Ha periódikusan frissíted a nézetet azt úgy hívják ,hogy busy waiting.

"Don't call us, we will call you". Amint a model megváltozik az összes viewnak kell változnia. A busy waiting-et csak a watchdog-ban kell használni, mindenhol máshol káros. Ez a Hollywood Principle.

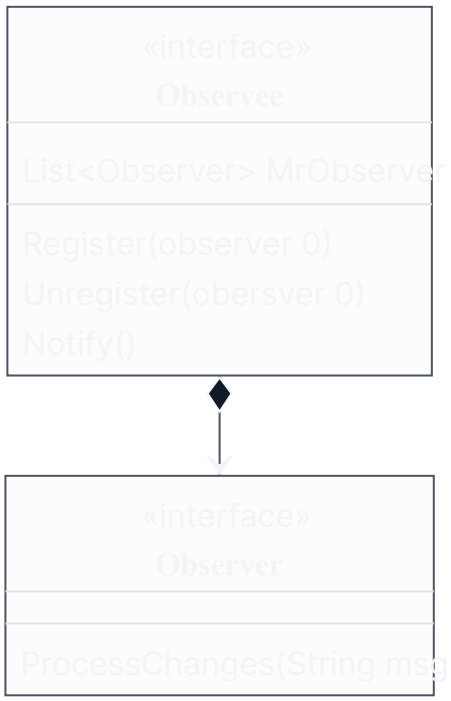
Observer : megfigyelő, Observee: megfigyelt
Ezek általában interfacek. Az observee-hez fel és le lehet iratkozni.
Pl. subscribe-unsubscribe, add listernet-remove listener, attach-deattach

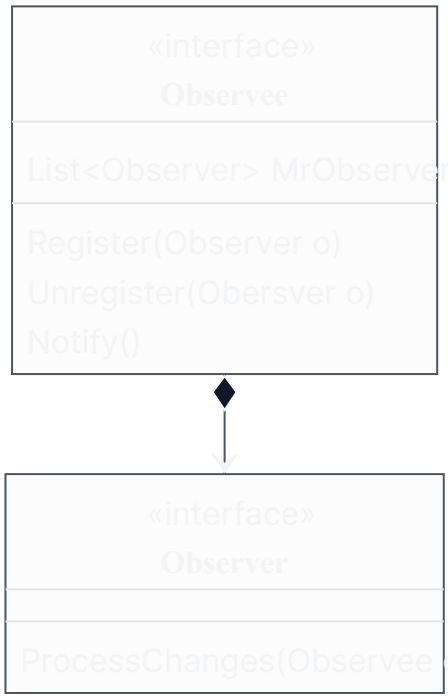
Értesítés: Ez akkor van amikor megváltozik a model és erről a megfigyelőket értesíteni kell. A notify egy foreach. Ebben végigmegyek a megfigyelőkön, és a megfigyelőket értesítem a változásról.

Két fajta observer van: a **pull**-os és a **push**-os
Pull: én veszem el a tortát
Push: a tortát az arcomba nyomják

Az értesítésnél a this-t adom át, … a listen-en keresztül le tudja kérni hogy mi változott. Ha pl. string-et adok át akkor mindent megkapok.

A megfigyeltnek van belső állapota, és ha annak változik a megfigyelt állapota, akkor arról értesíti a megfigyelőt, vagy elküldi a belső állapotát, vagy a megfigyelő lekéri a belső állapotoát.

> c07
>
> # Pull
>
> 
>
>
> # Push
>
> Adok egyreferenciát és azon keresztül a változásokat lekérjük



IS-A kapcsolatnak két fajtája van: Extends(--|>), implements(..|>).
MVP: Model view presenter
Model view view-model

A logka átmegy a presenterbe illetve a viewmodelbe. A presenter előfeldolgozza mit kell ... és a view ... . Az MVC-t gyakran párhuzamba állítják a 3-rétegű (3-tier) programozással. Ez a három réteg:

- GUI (Graphical User Interface)
- Business logic
- Persistency layer (gyakran az adatbázis)
  Sokszor szenvedünk azzal, hogy az üzleti logika és a GUI összekeveredik.

Parancstervezési minta

Akkor használjuk, ha egy tevékenységet később akarok csinálni, mint amikor eldől, mit akarok csinálni.

Egységbe zárom a tevékenységet. Tevékenység felületén egyetlen paracs van: execute.

Objektum orientált nyelvekben 2 nagy ... van: az egyik a metódus hívás (procedure call) a másik az üzenet küldés (message passing).