MILTON FRIEDMAN EGYETEM

PROGRAMOZÁSI TECHNOLÓGIÁK

5. óra – TERVEZÉSI ALAPELVEK

2024.04.08

Szalai Patrik

szalai.patrik@uni-milton.hu

TÉMAKÖRÖK:

Risk management

- Common Criteria

Tervezési Alapelvek

- OOP
  - Egységbezárás
  - Öröklődés
  - Többalakúság
- Alkalmazott OOP
  - Automatikus szemétgyűjtés
  - Lokális-globális változó mező
  - Osztály helyettesítés
  - Csatoltság csökkentése
- Tervezési Alapelvek
  - GOF1
  - GOF2
  - Principles

Feladatok

- Kockázatelemzés
  - Táblázatok kitöltése
- OOP
  - Encapsulation
  - Inheritance
  - Polymorphism
- GOF
  - GOF1
  - GOF2
  - Strategy minta és GOF1

Beadás: Április 19

KOCKÁZATELEMZÉS - CC

Common Criteria

- Common Criteria for Information Technology Security Evaluation
- Informatikai rendszerek tesztelése, annak érdekében, hogy megbizonyosodjunk azok biztonságosságáról
- Széles körűen elfogadott szabvány

Fogalmak

- ÉT – Értékelés Tárgya / TOE – Target of Evaluation
- VP – Védelmi Profil / PP – Protection Profile
- BRT – Biztonsági Rendszerterv / ST – Security Target
- BFK – Biztonsági Funkcionális Követelmények / SFR – Security Functional Requirements
- GBK – Garanciális Biztonsági Követelmények / SAR – Security Assurance Requirements
- ÉGSZ – Értékelési Garancia Szint / EAL – Evaluation Assurance Level

ÉGSZ Szintek

- ÉGSZ 1 - 7 definiálása

KOCKÁZATELEMZÉS - CC

- **ÉT – Értékelés Tárgya / TOE – Target of Evaluation**
  - A szoftver vagy rendszer megjelölése, melyet vizsgálunk a CC segítségével.
  - A CC nem foglalkozik fizikai/környezeti kockázatokkal

- **VP – Védelmi Profil / PP – Protection Profile**
  - Egy dokumentum
  - Felhasználói csoportok alakítják ki
  - Biztonsági követelmények gyűjteménye
  - Nem kötelező ez alapján elkészíteni a BRT-t, de hasznos útmutatás

- **BRT – Biztonsági Rendszerterv / ST – Security Target**
  - Létrejöhet VP-k alapján
  - BFK-kat sorol fel, melyek az ÉT teszt tárgyát képezik majd
  - Nem kötött tartalmú, általában publikusan elérhető (hogy lássa a végfelhasználó, milyen BFK-k esetén tanúsított a szoftver)

- **BFK – Biztonsági Funkcionális Követelmények / SFR – Security Functional Requirements**
  - Általánosan megfogalmazott biztonságra vonatkozó FUNKCIONÁLIS követelmények
  - A CC számos ajánlást és azok közötti összefüggést is listáz, de ezek nem kötelező jellegűek

KOCKÁZATELEMZÉS - CC

- **GBK – Garanciális Biztonsági Követelmények / SAR – Security Assurance Requirements**
  - Fejlesztés és tesztelés során megvalósítandó követelmények
  - Ezek alapján tudja teljesíteni a BRT-ben felsorolt BFK-kat az ÉT
  - Számszerűen mérhető eredményeket követel meg
  - A CC sok BGK-t felsorol és csoportosít

new CC:2022 Release 1

CC:2022 Release 1 consists of five parts. Make sure to download and use these files:

	PDF
Part 1: Introduction and general model	CC2022PART1R1.pdf
Part 2: Security functional requirements	CC2022PART2R1.pdf
Part 3: Security assurance requirements	CC2022PART3R1.pdf
Part 4: Framework for the specification of evaluation methods and activities	CC2022PART4R1.pdf
Part 5: Pre-defined packages of security requirements	CC2022PART5R1.pdf

- **ÉGSZ – Értékelési Garancia Szint / EAL – Evaluation Assurance Level**
  - Az CC alkalmazásának eredményterméke, hogy az ÉT kap egy ÉGSZ szintet 1 – 7-ig
  - Sorban kell haladni a szintek között
  - Magas ÉGSZ nem jelenti azt, hogy a szoftver biztonságosabb, csak a BRT-ben leírtak megbízhatóbban lettek tesztelve!
  - + jellel jelölik, ha teljesítette az adott ÉGSZ-t és magasabb szintekről is néhány GBK-t pl.: Windows XP ÉGSZ4+

KOCKÁZATELEMZÉS - CC

- **GBK – Garanciális Biztonsági Követelmények / SAR – Security Assurance Requirements**
  - Fejlesztés és tesztelés során megvalósítandó követelmények
  - Ezek alapján tudja teljesíteni a BRT-ben felsorolt BFK-kat az ÉT
  - Számszerűen mérhető eredményeket követel meg
  - A CC sok BGK-t felsorol és csoportosít

- **ÉGSZ – Értékelési Garancia Szint / EAL – Evaluation Assurance Level**
  - Az CC alkalmazásának eredményterméke, hogy az ÉT kap egy ÉGSZ szintet 1 – 7-ig
  - Sorban kell haladni a szintek között
  - Magas ÉGSZ nem jelenti azt, hogy a szoftver biztonságosabb, csak a BRT-ben leírtak megbízhatóbban lettek tesztelve!
  - + jellel jelölik, ha teljesítette az adott ÉGSZ-t és magasabb szintekről is néhány GBK-t pl.: Windows XP ÉGSZ4+
    - Vagy EAL Augmented

KOCKÁZATELEMZÉS - CC

- **ÉGSZ – Értékelési Garancia Szint / EAL – Evaluation Assurance Level SZINTEK**
  - **ÉGSz1:** Funkcionálisan tesztelt (EAL1: Functionally Tested)
  - **ÉGSz2:** Strukturálisan tesztelt (EAL2: Structurally Tested)
  - **ÉGSz3:** Módszeresen tesztelt és ellenőrzött (EAL3: Methodically Tested and Checked)
  - **ÉGSz4:** Módszeresen tervezett, tesztelt és áttekintett (EAL4: Methodically Designed, Tested, and Reviewed)
  - **ÉGSz5:** Félformálisan tervezett és tesztelt (EAL5: Semiformally Designed and Tested)
  - **ÉGSz6:** Félformálisan igazolt terv és tesztelt (EAL6: Semiformally Verified Design and Tested)
  - **ÉGSz7:** Formálisan igazolt terv és tesztelt (EAL7: Formally Verified Design and Tested)

Kérdés: Milyen hasonló minősítéseket ismerünk?

TERVEZÉSI ALAPELVEK

OOP

AOP

TERVEZÉSI ALAPELVEK

OOP

- Objektum Orientált Programozás
  - Legnépszerűbb
  - Legjobban támogatott
  - Leginkább kiforrott
  - Rugalmas forráskódot eredményez

„A program kódja állandóan változik”

AOP

- Aspektus Orientált Programozás
  - Programozási Technológiák fejezetnél vesszük
  - OOP limitáció esetén alkalmazandó
  - Segít funkciókat egységbe zárni és a fő üzleti logikától függetlenül kezelni

„Separation of Concerns”

ISMÉTLÉS

- **Szoftverkrízis**
  - A programozási nyelvek válasza: Modulok és Moduláris programozás
  - Az osztály a Modul OOP esetén

- **Az Osztály**

  Első megközelítés:
  - A valóság egy darabkájának ***absztrakciója***
  - Mérete ***granularitástól*** függ
  - Lehet teljesen technikai is, mely nem kapcsolódik a valósághoz

  Második megközelítés:
  - Összetett, ***inhomogén*** adattípus
  - Sokban hasonlít a ***rekordhoz,*** ami szintén ilyen adattípus

Az adatokat és a rajtuk végrehajtott műveleteket egységbe zárjuk. Ezek az egységek az OSZTÁLYOK.

Modul

- Forráskód kis része
- Egy programozó képes átlátni
- ***Fordítási alegység*** is
- Külön állományban található

Osztály és Rekord

- Egy osztály rugalmasabb és több képességgel rendelkezik
- Adat és Viselkedés egységbezárása
- Az osztály megváltoztatható
- Az osztály tartalmazhat metódusokat

AZ OSZTÁLY

- **Felépítése**
  - Mezők – Adattagok
  - Metódusok – Adattagokon értelmezett műveletek

- **Példányok – Objektumok**
  - Első értelmezésben: A világ összes lehetséges példányának absztrakciója
  - Másik értelmezésben: Példányosításkor konstruktorokból és viselkedésekből felépülő homogén adattípus
  - **3 jellemzője van:**
    - Felület (vagy Típus)
    - Viselkedés
    - Belső állapot

```
public class Kutya {
    private String név;
    public Kutya(String név) { this.név = név; }
    public String GetNév() { return név; }
}

Kutya kutya = new Kutya("Bobbikutya");
```

OBJEKTUMOK

```
public class Kutya {                                    // Az Osztály definiálja a példányosítást
    private String név;                                 // Paraméter, melyet átadunk a Konstruktornak
    public Kutya(String név) { this.név = név; }        // Konstruktor
    public String GetNév() { return név; }              // Lekérdező Metódus
}

Kutya kutya = new Kutya("Bobbikutya"); // Létrehoz egy új kutya Objektumot Bobbikutya névvel
```

- **1. Kérdés:** Mi a felülete a kutyának?
- **2. Kérdés:** Mi a viselkedése a kutyának?
- **3. Kérdés:** Mi a belső állapota a kutyának?

OBJEKTUMOK

```
public class Kutya {
    private String név;
}
```

1. Kérdés: Mi a felülete a kutyának?

Habár nem klasszikus OOP értelemben vett Interface
DE definiálja, hogy a kutyának van egy név értéke.
Ezt implementáláskor alkalmazzuk, tehát elfogadjuk a „szerződést” ”implicit interface”-ként.

```
public String GetNév() { return név; }
```

2. Kérdés: Mi a viselkedése a kutyának?

Nem írja le, hogy például a kutya „ugat” és ilyenkor ki kell írni azt, hogy „Vau vau”
DE leírja, hogy milyen viselkedéssel lehet lekérdezni a Kutya objektum egységbezárt nevét.

```
Kutya kutya = new Kutya("Bobbikutya");
```

3. Kérdés: Mi a belső állapota a kutyának?

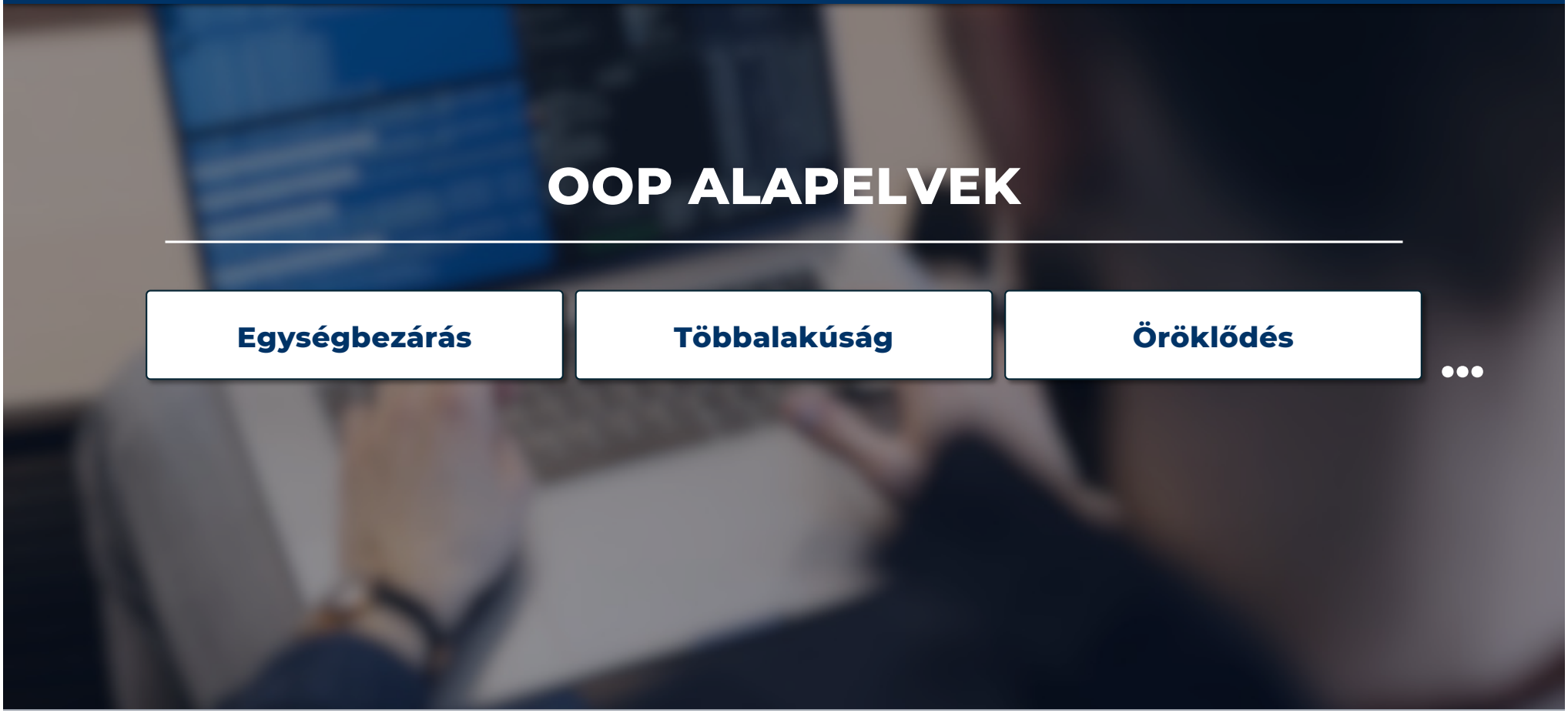
Az új kutya belső állapota az lesz, hogy a neve Bobbikutya.

AZ OBJEKTUM

Kutya példa

- **Az objektum felülete** megegyezik az Osztály felületével
  - A kutya objektum Kutya **Típusú**
  - Egy objektumnak több típusa is lehet
- **Az objektum viselkedését** metódusainak implementációja adja
  - Megegyezik az osztály viselkedésével, példánya az Objektum
  - A viselkedés a program dinamikájában változhat! **Ismétlés:** Statikus és Futó forráskód
- **A belső állapot** a pillanatnyi értéke
  - Az osztály metódusai megváltoztatják a mezők értékeit, mint állapotátmeneti operátorok
  - **Ismétlés:** Kezdő érték és Pillanatnyi érték

- **Interface**
  - Csak felülete van
- **Absztrakt osztály**
  - Felülete és részleges viselkedése van (vagy egyáltalán nincs, ha minden metódusa absztrakt)



OOP ALAPELVEK

- Egységbezárás
- Többalakúság
- Öröklődés
- ...

OOP ALAPELVEK

- **Egységbezárás, mint OOP alapelv**
  - Az objektum belső állapota legyen megváltoztathatatlan
  - Lehetőleg NE használjunk publikus mezőket
  - Lehetőleg ne adjunk vissza olyan referenciát, mely egy ilyen mezőre mutat

  - Védjük az objektum belső állapotát. Ezt hívjuk információ rejtésnek.

- **Egységbezárás, mint klasszikus fogalom**
  - Az adattagokat és rajtuk végrehajtó metódusokat egységbe zárjuk

- **Encapsulation**

Kérdés: Miért fontos az OOP szerinti egységbezárás?

OOP ALAPELVEK

Egységbezárás – Példa

```
// Szolgáltatás példa
class Kutya {
    private ArrayList<String> nevek = new ArrayList<>(); // Továbbra is private!
    public ArrayList<String> getNevek() { return nevek; } // Visszaraktuk a hibát a példába!
    public void addNev(String név) {
        if (!név.equals("Adolf")) nevek.add(név);
        // Vizsgálat: Nem lehet a kutya neve Adolf.
    }
}

// Szolgáltatás példa program
main() {
    Kutya k1 = new Kutya();
    k1.getNevek().add("Adolf"); // Megszegi az egységbezárást, így kikerülheti a vizsgálatot.
    k1.addNev("Adolf"); // Nem szegi meg, ezért működési logika szerint el lesz utasítva.
}
```

- **Hiába private** a 'nevek' ArrayList, a 'getNevek' továbbra is vissza ad referencát rá!
- Az egységbezárás betartása nélkül „lyukas" a vizsgálatunk, könnyen megkerülhető.

OOP ALAPELVEK

Öröklődés

- A kód újrahasznosítás egy formája
- A gyermek osztály az ős minden nem privát mezőjét és metódusát örökölni fogja. Avagy a felületét és megvalósítását.
- Az absztrakt metódusokat felülírhatjuk
- Kényelmes, de veszélyes...

Inheritance

Kérdés: Milyen veszélynek a forrása?
Kérdés: Mit javasolt alkalmazni helyette a GOF2 szerint?

OOP ALAPELVEK

Öröklődés

```java
// Java-ban a Példa 3 – Öröklődés
class Allat extends Object{} // Nem kell kiírni, mert automatikus!
class Gerinces extends Allat{}
class Macska extends Gerinces{}
class HaziMacska extends Macska{}

HaziMacska h1 = new HaziMacska();
```

A h1-típusai:

- Object
- Allat
- Gerinces
- Macska
- HaziMacska

OOP ALAPELVEK

- **Többalakúság**
  - Az öröklődés következménye
  - Egy objektumnak több típusa is lehet
  - Egy objektum több típusként, azaz alakban is felhasználható
- **Polymorphism**

Kérdés: Milyen ajánlás van a többalakúság alkalmazására?

OOP ALAPELVEK

Többalakúság

```
// C# Példa 3 - Többalakúság
class Allat{}
class Gerinces:Allat{}
class Macska:Gerinces{}
class HaziMacska:Macska{}

HaziMacska h1 = new HaziMacska();
Macska m1 = new HaziMacska();
```

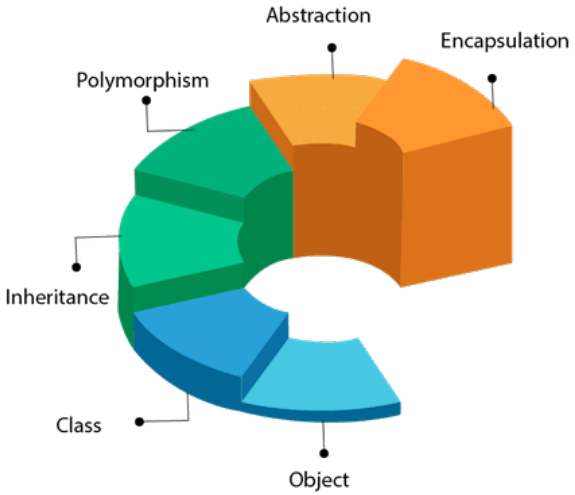
Létrehozhatok Házimacskát, mint Macska, vagy akár Gerinces

ALKALMAZOTT OOP

ALKALMAZOTT OOP

Az OOP nagyon hasznos eszközöket ad, de ismerjük is mire lehet használni őket!

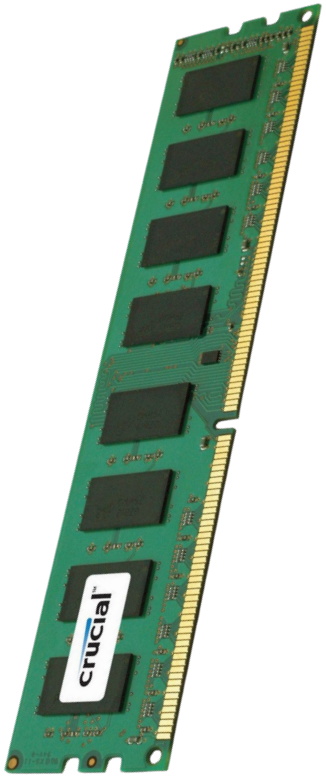
- Automatikus szemétgyűjtés
- Lokális-globális változó mező
- Osztály behelyettesítés többalakúsággal
- Csatoltság csökkentése objektum-összetétellel



AUTOMATIKUS SZEMÉTGYŰJTÉS – GARBAGE COLLECTION

- Nem csak OOP nyelvekre jellemző
- A Memória problémája
- Minden új utasítás memóriát foglal, fel kell szabadítani azt, ami már nem használt
- Az OOP alkalmazása leveszi ennek a terhét a programozóról
- Futás közben történik

1. **Identification**
   A szemétgyűjtő feltérképezi azokat az objektumokat, melyekre már egy változó vagy adat struktúra sem hivatkozik – Ezt támogatja az egységbezárás
   - *Unreferenced Object*
2. **Reclamation**
   A szemétgyűjtő ezeket az objektumokat üríti a memóriából, melyet az általuk használt memória „újrahasználható” megjelölésével tesz – C# és Java automatikusan támogatja
   - *Deallocate*
3. **(Nem mindig) Compaction**
   Bizonyos szemétgyűjtő algoritmusok használják, a memória átrendezésével csökkentik a töredezettséget és az optimalizálva a Memóriahasználatot



MEZŐ, MINT LOKÁLIS-GLOBÁLIS VÁLTOZÓ

- **Globális változó**
  - Gyorsabb és kisebb kód fejlesztés
  - Mellékhatás – Amikor egy alprogram megváltoztatja a környezetét
  - A mellékhatás nehezen visszakövethető hibákat eredményez
- **Mező**
  - Osztályon belül globális
  - Kívülről elérhetetlen
  - Tudunk vele mellékhatást előidézni, de az az osztályon belül lokális
  - Mellékhatásokból eredő hibák könnyen visszakövethetők
  - Az egységbezárás miatt sem javasolt teljesen globális változók használata

```
public class Pelda {
    private int globalMezo; // Globális minden osztályban metódus számára (PRIVÁT!)
    public Pelda() {
        globalMezo = 10; // Konstruktor, ami megadja a kezdő értékét (PUBLIKUS)
    }
    public void lokalisMezoMetodus() {
        int lokalMezo = 20; // Lokális mező, mely csak erre a metódusra érvényes
        System.out.println("Local field value: " + lokalMezo);
        System.out.println("Global field value: " + globalMezo); // Elérjük a Globális mezőt is
    }
}
```

CSATOLTSÁG CSÖKKENTÉSE OBJEKTUM-ÖSSZETÉTELLEL

- **Csatoltság (Coupling)**
  - Egy osztály, vagy más modul
  - Milyen mértékben alapszik a többi osztályon
  - Csatoltság mértéke fordítottan arányos a Kohézióval

- **Larry Constantine**
  - **OOP-ben a csatoltság annak mértéke, hogy milyen erős kapcsolatban áll egy osztály a többi osztállyal. A csatolás mértéke két osztály, mondjuk A és B között növekszik, ha:**
    - A-nak van B típusú mezője.
    - A meghívja B valamelyik metódusát.
    - A-nak van olyan metódusa, amelynek visszatérési típusa B.
    - A B-nek leszármazottja, vagy implementálja B-t.
  - **Csatoltság szintjei:**
    - Erősen csatolt
    - Gyengén csatolt
    - Réteg
  - **Az Erős csatoltság erős függőséget is jelent!**

ISMÉTLÉS GOF 2

- **GOF2:** Favour object composition over inheritance.
- Használjunk objektum összetételt öröklődés helyett, ahol csak lehet.

```
// Öröklődéssel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya extends Geringes {
     public String gyorsanFut() {
          return "gyorsan" + fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
// s = "gyorsanfut 1"
```

- Egyszerűen használjuk a 'fut' metódust, mert megörökölte
- Hierarchikus, nem rugalmas!
- Fordítási időben történik

```
// Objektum-összetétellel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya {
     Geringes g = new Geringes();
     public String gyorsanFut() {
          return "gyorsan" + g.fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
```

- Referencián keresztül használjuk a 'fut' metódust, mert meghívtuk
- Rugalmas, használja, de nincs hierarchikus kapcsolat!
- Futási időben történik

OO TERVEZÉSI ALAPELVEK

GOF1, GOF2	IS-A, HAS-A	OCP
LSP	Design by Contract	ISP
DIP	HP	Law of Demeter

GOF 1

- **GOF1:** Program to an Interface, not an Implementation

Megvalósításra programozni:

- Ha egy osztály kódjában felhasználjuk egy másik osztály implementációját
- **Haszna:** Gyors és rövid kódot eredményez általában
- **Veszélye:** Ha megváltozik az egyik osztály, akkor a másik osztályt is meg kell változtatni
  - **Azaz:** Implementációs függőséget okoz!

GOF 2

- **GOF2:** Favour object composition over inheritance.
- Használjunk objektum összetételt öröklődés helyett, ahol csak lehet. **(Ahol kell többalakúság, ott nem lehet!)**

```
// Öröklődéssel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya extends Geringes {
     public String gyorsanFut() {
          return "gyorsan" + fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
// s = "gyorsanfut 1"
```

- Egyszerűen használjuk a 'fut' metódust, mert megörökölte
- Hierarchikus, nem rugalmas!
- Fordítási időben történik

```
// Objektum-összetétellel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya {
     Geringes g = new Geringes();
     public String gyorsanFut() {
          return "gyorsan" + g.fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
```

- Referencián keresztül használjuk a 'fut' metódust, mert meghívtuk
- Rugalmas, használja, de nincs hierarchikus kapcsolat!
- Futási időben történik

IS-A és HAS-A

IS-A:

- Öröklődés
- HaziMacska **IS-A** Macska

HAS-A:

- Objektum-összetétel
- Gitáros **HAS-A** Gitár – Aggregáció
- Kutya **HAS-A** Kutyafül – Kompozíció

A kutyának van gerince, mert **kutya IS-A gerinces.**

```
// IS-A
class Kutya : Gerinces {
     public void Fut() {
          Console.Write("Gyorsan ");
          LábVezérlés();
     }
}
```

Kutya HAS-A gerinc.

```
// HAS-A
class Kutya2 {
     Gerinces gerinc;
     public Kutya2(Gerinces gerinc) {
          this.gerinc = gerinc;
     }
     public void Fut() {
          Console.Write("Gyorsan ");
          gerinc.LábVezérlés();
     }
}
```

Csatoltság erőssége:

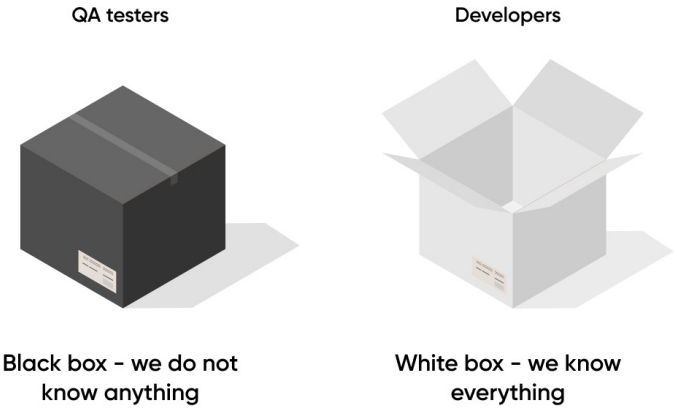
- Öröklődés **>** Kompozíció **>** Aggregáció
- Öröklődés **>** Objektum-összetétel

ÁTLÁTSZÓ ÉS NEMÁTLÁTSZÓ ÚJRAHASZNOSÍTÁS

White-box reuse

- Öröklődés
- Örökölt metódusokat használunk és azokat ismerjük is.

- **Black-box reuse**
- Objektum-összetétel
- Az összetételt megvalósító mezőn keresztül hívunk metódusokat, de azok forrásáról nincs információnk



ÁTLÁTSZÓ ÉS NEMÁTLÁTSZÓ BECSOMAGOLÁS

- Ha **birtoklunk** egy objektumot, hogy saját szolgáltatásaink felelősségét részben- vagy egészben átadjuk (delegáljuk) neki, akkor **becsomagolásról** is beszélünk.

- **Birtoklás:** Egy osztály tartalmaz- vagy használ egy másik osztályt, birtokolja azt.
- **Becsomagolás:** Csak feladatokat- vagy kéréseket továbbít a másik osztály felé, becsomagolja azt.

ÁTLÁTSZÓ BECSOMAGOLÁS

- A becsomagolt példány **ugyanolyan felületű**, mint a becsomagoló.
- A **becsomagolt objektum szolgáltatásai elérhetők** a becsomagolón keresztül.
- Megvalósításhoz kell egy IS-A és egy HAS-A kapcsolat.

- **IS-A** – A karácsonyfa továbbra is karácsonyfa marad, akárhogy is díszítjük.
- **HAS-A** – Dísszel becsomagoljuk a karácsonyfát.

```
class GömbDísz : KarácsonyFa // IS-A kapcsolat a KarácsonyFa osztállyal
{
    KarácsonyFa kf; // HAS-A kapcsolat a KarácsonyFa osztállyal
    public GömbDísz(KarácsonyFa kf) { this.kf = kf; }
    public override string GetTípus()
    {
        return kf.GetTípus(); // Felelősség átadás
    }
    public void A() { kf.A(); } // Felelősség átadás általánosítva
    public override string ToString()
    {
        return "Díszes " + kf.ToString();// Részleges felelősség átadás
    }
}
```

NEMÁTLÁTSZÓ BECSOMAGOLÁS

- A becsomagolt példány **nem ugyanolyan felületű**, mint a becsomagoló.
- A **becsomagolt objektum szolgáltatásai rejtve maradnak, nem elérhetők kívülről.**
- Az elérhető szolgáltatások elvégzéséhez **használhatók a becsomagolt objektum szolgáltatásai.**
- Csak egy HAS-A kapcsolat kell.

- **Csak HAS-A** – A karácsonyfa már Díszes karácsonyfa lesz a végén.

```
class GömbDísz // Nincs IS-A kapcsolat a KarácsonyFa osztállyal
{
    KarácsonyFa kf; // Csak HAS-A kapcsolat van
    public GömbDísz(KarácsonyFa kf) { this.kf = kf; }
    public string GetTípus() { return kf.GetTípus(); } // Felelősség átadás
    public void A() { kf.B(); } // Felelősség átadás általánosítva
    public override string ToString()
    {
        return "Díszes " + kf.ToString(); // Részleges felelősség átadás
    }
}
```

SINGLE RESPONSIBILITY PRINCIPLE

Egy felelősség - egy osztály alapelve

- Minden osztálynak egyetlen felelősséget kell lefednie, de azt teljes egészében.
- *A class should have only one reason to change.*

GOF1

- Ha egy osztály nem fedi le teljesen a saját felelősségi körét, akkor kényszerülünk implementációra programozni.

AOP

- Ha egy osztály több felelősségi kört is ellát, akkor sokkal jobban ki van téve a változtatásoknak.

Példa: HaziAllat tud: Enni, Aludni, Ugatni, Nyávogni
Ha változik, hogy nem csak a Postást de a Futárt is megugatja, vagy változik a macskák viselkedése (esetleg bővül), akkor változnia kell a HaziAllat-nak is.

- Szép elképzelés, hogy minden osztály csak egyetlen felelősséget lát el és azt teljesen lefedi, de gyakorlatban a naplózás, jogosultság ellenőrzés és hasonlók meggátolják ezt.
- **Erre ad megoldást az AOP** – Aspektusokba emeli ki ezeket a felelősségeket, melyeket az osztályhoz kapcsolhatunk.

OPEN-CLOSED PRINCIPLE

Nyitva-zárt alapelv

- A program forráskódja legyen nyitott a bővítésre, de zárt a módosításra.
- *Classes should be open for extension, but closed for modification*

Osztályhierarchia

- Új alosztályt vagy metódust tudjak felvenni
- Meglévőt ne írhassunk felül

Hibalehetőségek

- A változás miatt az eddig működő ágak hibásak lesznek,
- A változás miatt a vele implementációs függőségben lévő kódrészeket is változtatni kell,
- A változás általában azt jelenti, hogy olyan esetet kezelek le, amit eddig nem, azaz bejön egy új if vagy else, esetleg egy switch, ami csökkenti a kód átláthatóságát, és egy idő után már senki se mer hozzányúlni.

OPEN-CLOSED PRINCIPLE

C# szabály

- Ne használjunk override kulcsszót, kivéve ha abstract vagy hook metódust írunk

Absztrakt metódus

- Muszáj felülírni, mert nincs törzse
- **OCP összefüggés:** Csak a törzzsel bővítjük a kódot, nem módosítunk

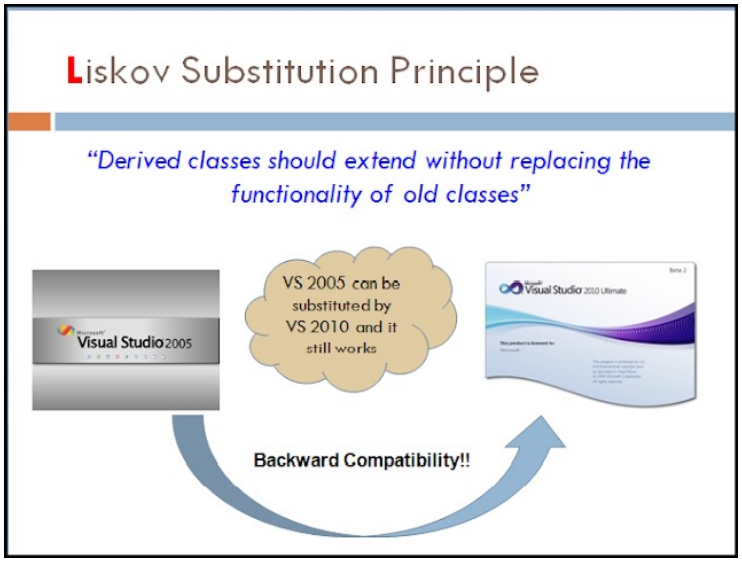
Hook

- A metódusnak van törzse, de az teljesen üres
- Felülírásuk nem kötelező
- **OCP összefüggés:** Felülírás esetén szintén csak bővítjük a kódot, nem változtatunk a meglévő részeken

LISKOV SUBSTITUTIONAL PRINCIPLE

Liskov-féle behelyettesítési alapelv

- A program viselkedése nem változhat meg attól, hogy az ős osztály egy példánya helyett később valamely gyermek osztály példányát használom.
- *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T*



Példa:

- A programom kutyák lábainak számát adja vissza
- Ha korábban kutyát használtam, de már Vizslát, akkor is 4

OCP és LSP általában egymást erősítik!

DESIGN BY CONTRACT

Szerződésalapú programozás

- **Típusai:**
  - **Előfeltétel**, amely a bemenetre és mezőkre ad megkötést
  - **Utófeltétel**, amely a kimenetre és a mezőkre ad megkötést
  - **Invariáns**, amely csak a mezőkre ad megkötést

Előfeltétel:

- A metódus előfeltétele írja le, hogy **milyen bementre működik helyesen a metódus.** Az előfeltétel általában a metódus paraméterei és az osztály mezői segítségével írja le ezt a feltételt.
- Például az Osztás(int osztandó, int osztó) metódus előfeltétele, hogy az osztó ne legyen nulla.

Utófeltétel:

- A metódus utófeltétele írja le, hogy **milyen feltételnek felel meg a visszaadott érték**, illetve **milyen állapotátmenet történt,** azaz az osztály mezői hogyan változnak a metódushívás hatására.
- Például a Maximum(int X, int Y) utófeltétele, hogy a visszatérési érték X, ha X>Y, egyébként Y.

Invariáns:

- Az osztályinvariáns az osztály lehetséges állapotait írja le, azaz az osztály mezőire ad feltételt. Az invariánsnak minden metódushívás előtt és után igaznak kell lennie.

DESIGN BY CONTRACT

Szerződésalapú programozás

- **Altípusok definíciója**
  - Az ős mezői felett az **altípus invariánsa nem gyengébb**, mint az ősé,
  - Az altípusban az **előfeltételek nem erősebbek**, mint az ősben,
  - Az altípusban az **utófeltételek nem gyengébbek**, mint az ősben,
  - Az altípus **betartja ősének történeti megszorítást** (history constraint).

- Az ős mezői felett a belső állapotok halmaza kisebb vagy egyenlő az altípusban, mint az ősben,
- Minden metódus értelmezési tartománya nagyobb vagy egyenlő az altípusban, mint az ősben,
- Minden metódusra a metódus hívása előtti lehetséges belső állapotok halmaza nagyobb vagy egyenlő az altípusban, mint az ősben,
- Minden metódus értékkészlete kisebb vagy egyenlő az altípusban, mint az ősben,
- Minden metódusra a metódus hívása utáni lehetséges belső állapotok halmaza kisebb vagy egyenlő az altípusban, mint az ősben,
- Az ős mezői felett a lehetséges állapotátmenetek halmaza kisebb vagy egyenlő az altípusban, mint az ősben.

INTERFACE SEGREGATION PRINCIPLE

Interfésszegregációs-alapelv

- Egy sok szolgáltatást nyújtó osztály fölé el kell helyezni interfészeket, hogy minden kliens, amely használja az osztály szolgáltatásait, csak azokat a metódusokat lássa, melyeket ténylegesen használ.
- *No client should be forced to depend on methods it does not use*

- Segít a függőség visszaszorításában
- Segít a kövér osztályok kiszorításában (Az SRP ki is zárja!)

DEPENDENCY INVERSION PRINCIPLE

Függőség megfordításának alapelve

- A magas szintű komponensek ne függjenek alacsony szintű implementációs részeket kidolgozó osztályoktól, hanem épp fordítva.
- Magas absztrakciós szinteken álló komponensektől függjenek az alacsony absztrakciós szinten állók.
- *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

- **Library** – Alacsony szintű, sokszor újrafelhasznált komponensek
- Rendszer logikáját leíró magas szintű komponensek újrahasznosítását segíti a DIP.

Példa:

```
public void Copy() { while( (char c = Console.ReadKey()) != EOF) Printer.printChar(c); }
```

DEPENDENCY INVERSION PRINCIPLE

Függőség megfordításának alapelve

```
public void Copy() { while( (char c = Console.ReadKey()) != EOF) Printer.printChar(c); }
```

- A példánkban egy nyomtatószervernél a Copy metódus függ a Console.ReadKey és a Printer.printChar metódustól.
- A Copy magát a főlogikát írja le, forrásból a célba másolunk a file vége jelig.
- Ezt sok helyen újra felhasználhatjuk, hiszen a forrás és a cél is változtatható.

A fenti kód újra hasznosításához jelenleg 2 opcióm van:

- If-else szerkezettel megállapítom, hogy melyik forrás és cél lesz az éppen szükséges páros. Ez nem túl szép és nem is túl effektív.
- A forrás és a cél referenciáját egy hívó felelősség injektálásával kívülről emelem be. Így gyorsan és rugalmasan tudom újra felhasználathóvá tenni a Copy funkciót. **Ezt hívják dependency injection-nek.**

DEPENDENCY INVERSION PRINCIPLE

Dependency Injection

- **Konstruktorral:** Az osztály a konstruktorán keresztül kapja meg azokat a referenciákat, amiken keresztül a neki hasznos szolgáltatásokat meg tudja hívni. Ezt más néven **objektum-összetételnek** is nevezzük.
- **Szetter metódusokkal:** Az osztály szetter metódusokon keresztül kapja meg a szükséges referenciákat. Általában akkor alkalmazzuk, ha **opcionális működés megvalósításához** kell objektum-összetételt alkalmaznunk.
- **Interfész megvalósításával:** Ha a példányt a magas szintű komponens is elkészítheti, akkor elegendő megadni a példány interfészét, amit általában maga a magas szintű komponens valósít meg, de paraméterosztály paramétereként is jöhet.
- **Elnevezési konvenció, konfigurációs állomány, vagy annotáció alapján:** Általában keretrendszerekre jellemző, csak tapasztalt programozóknak ajánlott, mert nyomkövetéssel nem lehet megtalálni, hogy honnan jön a példány.

HOLLYWOOD PRINCIPLE

Ne hívj, majd mi hívunk!

- Az alapelv lényege, hogy csökkentsük a feldolgozási időt.
- Ezt úgy érjük el, hogy megszabjuk: Ne kérdezgessen az, aki az eseményre vár, majd az esemény értesíti a várakozókat.
- Ilyen rendszerek például a „Watchdog” rendszerek, melyek időszakosan pingeléssel figyelnek egy távoli objektumot, hogy az él-e még.
- High Availability esetében ha egy távoli rendszer megáll (pl. áramszünet/egyéb hiba esetén), akkor azt a Watchdog eseményként észleli és átirányítja a forgalmat más rendszerekre, valamint értesítést küld az illetékeseknek.
- Alternatívája a Broadcasting, amikor a forrás sugározza az információt több fogadó felé, viszont a forrás nem feltétlenül ismeri a célt. Ez esetben előfordulhat, hogy az is megkapja az üzenetet, akinek nincs rá szüksége.
- Mindkettőt akkor érdemes alkalmazni, ha objektumain több kapcsolatban vannak és több oldal is dinamikusan változik, fel- és le is lehet iratkozni.

LAW OF DEMETER

A legkisebb tudás elve

- Egy osztály csak a közvetlen ismerőseit hívhatja.
- Avagy, csak annak a példánynak a metódusait hívhatjuk, akikre van közvetlen referenciánk.
- Alkalmazásának köszönhetően a változások csak lokális hatásúak lesznek.

Példák:

- A.getB() és A.getC() helyes az elv szerint, közvetlen referencián keresztül
- A.getB().D() már nem közvetlen referencia, hanem a B() számára az, így A-nak közvetettként nem szabad hívnia.
- B.getD() már helyes.

MILTON FRIEDMAN EGYETEM

KÖSZÖNÖM A FIGYELMET!

5. óra – TERVEZÉSI ALAPELVEK

2024.04.08

Szalai Patrik

szalai.patrik@uni-milton.hu