

Komponens alapú szoftverfejlesztés

8. előadás

Szoftver architektúrák alapvetései

Giachetta Roberto

groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

„Architecture is about the important stuff.
Whatever that is.”

(Martin Fowler)

Szoftver architektúrák alapvetései

A szoftver architektúra

- *Szoftver architektúra*nak nevezzük a szoftver fejlesztése során meghozott *elsődleges tervezési döntések* halmazát
 - meghatározzák a rendszer magas szintű felépítését és működését, az egyes alkotóelemek csatlakozási pontjait
 - megváltoztatásuk később jelentős újratervezést igényelné a szoftvernek
- A szoftver architektúráját különböző szempontok szerint közelíthetjük meg, pl.:
 - a szoftver által nyújtott szolgáltatások (funkciók) szerint
 - a felhasználó és a futtató platform közötti tevékenységi szint szerint
 - az adatátadás, kommunikáció módja szerint

Szoftver architektúrák alapvetései

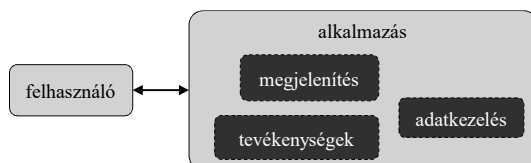
A szoftver architektúra

- Minden szoftver rendelkezik architektúrával
- Az architektúra létrehozása során mintákra hagyatkozunk
 - a szoftver (vagy egy alrendszerének) teljes architektúráját definiáló mintákat nevezzük *architekturális mintáknak* (*architectural pattern*)
 - az architektúra alkalmazásának módját, az egyes komponensek összekapcsolását segítik elő a *tervminták* (*design pattern*)
 - mindkettő tervezési döntések gyűjteménye, amelyek többször felmerülő tervezési kérdésre ad megfelelő választ
 - a minta rendelkezik névvel, céllal (scope), környezettel (context), erősségekkel, és gyengeségekkel

Szoftver architektúrák alapvetései

A monolitikus architektúra

- A legegyszerűbb felépítést a monolitikus *architektúra* (*monolithic architecture*) adja
 - nincsenek programegységekbe szétválasztva a funkciók
 - a felület megjelenítő kód vegyül az adatkezeléssel, a tevékenységek végrehajtásával, stb.



Szoftver architektúrák alapvetései

A monolitikus architektúra

- Összetettebb alkalmazásoknál a monolitikus felépítés korlátozza a program
 - áttekinthetőségét, tesztelhetőségét (pl. nehezen látható át, hol tároljuk a számításokhoz szükséges adatokat)
 - módosíthatóságát, bővíthetőségét (pl. nehezen lehet a felület kinézetét módosítani)
 - újrafelhasználhatóságát (pl. komponens kiemelése és áthelyezése másik alkalmazásba)
- A monolitikus felépítés nem követi az objektumorientált tervezés alapelveit (SOLID)

Szoftver architektúrák alapvetései

Példa

Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- a két játékos felváltva lép, a kereszt és kör szimbólumokat használva egy 9 mezőből álló táblán
- a játék akkor ér véget, ha valamely játékos elfoglal három mezőt (sorban, oszlopban, átlósan), vagy minden mező betelik
- a program automatikusan jelez, ha vége a játéknak (előugró üzenetben), majd új játékot kezd
- a felhasználó ezen felül bármikor menthet és betölthet játékot

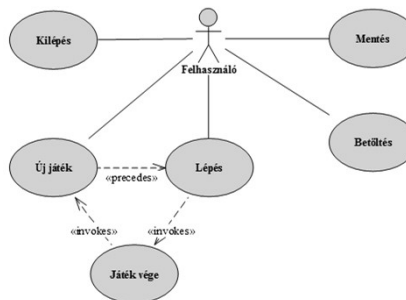
ELTE IK, Komponens alapú szoftverfejlesztés

8:7

Szoftver architektúrák alapvetései

Példa

Tervezés (felhasználói esetek):



ELTE IK, Komponens alapú szoftverfejlesztés

8:8

Szoftver architektúrák alapvetései

Példa

Tervezés (architektúra):

- az alkalmazást egy osztályban (**TicTacToeForm**) valósítjuk meg, amely tartalmazza a grafikus felületet és a játék viselkedését
- a betöltés (**Load**) hatására gombok segítségével legeneráljuk a játéktáblát (**GenerateTable**), majd új játékot kezdünk (**NewGame**)
- a felületen elhelyezzük a játéktábla gombjait (**buttonBoard**), továbbá egy karakterrel eltároljuk az aktuális játékos jelét (**currentPlayer**), valamint a lépésszámot (**stepNumber**)

ELTE IK, Komponens alapú szoftverfejlesztés

8:9

Szoftver architektúrák alapvetései

Példa

Tervezés (architektúra):

TicTacToeForm		Form
-	buttonBoard	Button [1,1]
-	stepNumber	int32
-	currentPlayer	Char
+	TicTacToeForm()	
-	GenerateTable()	.void
-	NewGame()	.void
-	ClickGame()	.void
-	LoadGame(String)	.void
-	SaveGame(String)	.void
-	TicTacToeForm_Load(object, EventArgs)	.void
-	TicTacToeForm_SizeChanged(object, EventArgs)	.void
-	ButtonGrid_Click(object, EventArgs)	.void
-	MenuGameNew_Click(object, EventArgs)	.void
-	MenuGameLoad_Click(object, EventArgs)	.void
-	MenuGameSave_Click(object, EventArgs)	.void
-	MenuGameExit_Click(object, EventArgs)	.void

ELTE IK, Komponens alapú szoftverfejlesztés

8:10

Szoftver architektúrák alapvetései

A tervezés alapelvei

- Az objektumorientált tervezés során öt alapelvet célszerű követnünk (**SOLID**):
 - *Single responsibility principle* (SRP): egy programegység csak egyvalamiért felelhet
 - *Open/closed principle* (OCP): a programegységek nyitottak a kiterjesztésre, de zártak a módosításra
 - *Liskov substitution principle* (LSP): az objektumok helyettesíthetők altípusaik példányával
 - *Interface segregation principle* (ISP): egy általános interfész helyett több kliens specifikus interfészt alkalmazunk
 - *Dependency inversion principle* (DIP): az absztrakciótól függünk, nem a konkretizációtól

ELTE IK, Komponens alapú szoftverfejlesztés

8:11

Szoftver architektúrák alapvetései

Single responsibility principle

- A *Single responsibility principle* (SRP) kimondja, hogy egy programegység (komponens, osztály, metódus) csak egy felelősséggel rendelkezhet
 - a felelősség az a tárgykör, ami változtatás alapegységeként szolgálhat („*reason for a change*”)
 - műveletek és adatok egy halmaza, amelyet ugyanannak az üzleti szerepkörnek (*business role*) a része, és egy követelmény teljesítését biztosítják
 - azonosítanunk kell a szereplőket, majd a követelményeiket, végül a tárgyköröket összefüggéseik szerint (milyen tényezők változtathatóak egymástól függetlenül), ennek megfelelően alakítjuk ki a felelősségeket

ELTE IK, Komponens alapú szoftverfejlesztés

8:12

Szoftver architektúrák alapvetései

Single responsibility principle

- számos, jól megkülönböztethető felelősségi kör található az alkalmazásokban:
 - perzisztencia, üzleti logika, vizualizáció, felhasználói interakció
 - adatformázás, konverzió, validáció
 - hibakezelés, eseménynaplózás
 - típuskiválasztás, példányosítás
- elősegíti a programegységek közötti laza csatolást, viszont túlzott használata feltöredezheti a programszerkezetet
 - célszerű csak azokat a változásokat figyelembe venni, amik várhatóak bekövetkezhetnek

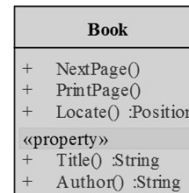
ELTE IK, Komponens alapú szoftverfejlesztés

8:13

Szoftver architektúrák alapvetései

Single responsibility principle

Példa: Modellezzünk egy könyvtári könyv (**Book**), amelynek van szerzője (**Author**), címe (**Title**), lehet lapozni (**NextPage**), kiírni (**PrintPage**), és megkeresni a helyét a könyvtárban (**Locate**).



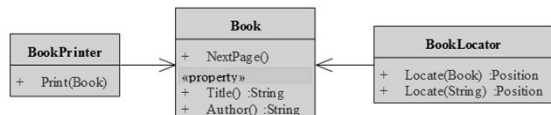
ELTE IK, Komponens alapú szoftverfejlesztés

8:14

Szoftver architektúrák alapvetései

Single responsibility principle

- a könyv elérhető két szerepkörben, az olvasó és a könyvtáros számára is
- a kiírás csak a olvasóra tartozik, és számos módja lehet, ezért célszerű a leválasztása
- a keresés csak a könyvtárosra tartozik, a kölcsönzőre nem, ugyanakkor szintén több módon is elvégezhető, ezért célszerű a leválasztása



ELTE IK, Komponens alapú szoftverfejlesztés

8:15

Szoftver architektúrák alapvetései

Háromrétegű architektúra

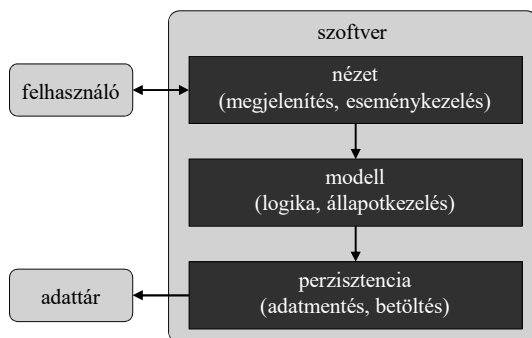
- A háromrétegű (3-tier) architektúra egy olyan általános szoftverfelépítés, amelyben elkülönül:
 - a nézet (*presentation, view, display*): tartalmazza a felhasználói felé történő interakciót (megjelenés, eseménykezelés)
 - a modell (*logic, business logic, application, domain*): tartalmazza a program szolgáltatásait, a funkcionális követelmények megvalósítását
 - a perzisztencia, vagy adatelérés (*data, data access, persistence*): biztosítja a hosszú távú adattárolást, az adattovábbítást más szoftverek felé

ELTE IK, Komponens alapú szoftverfejlesztés

8:16

Szoftver architektúrák alapvetései

Háromrétegű architektúra



ELTE IK, Komponens alapú szoftverfejlesztés

8:17

Szoftver architektúrák alapvetései

Példa

Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- háromrétegű architektúrát építünk fel, amelyben külön programegységbe helyezük a nézetet (**TicTacToeForm**), a modellt (**TicTacToeModel**) és a perzisztenciát (**TextFilePersistence**)
- a modell valósítja meg a játék logikáját, ezért egy logikai reprezentációját tartalmazza a játéknak (a **Player** felsorolási típus segítségével)
- a mentés és betöltés folyamata így a modellen keresztül (amely átalakítja a játékállapotot perzisztálhatóvá) a perzisztenciába fut be, így kialakítva egy *felelősségi láncot* (*chain of responsibility*)

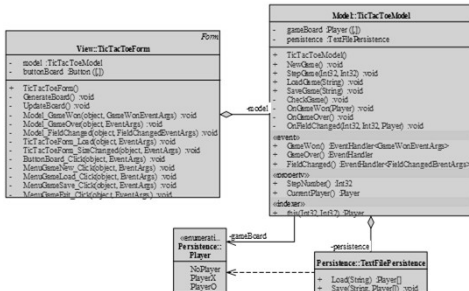
ELTE IK, Komponens alapú szoftverfejlesztés

8:18

Szoftver architektúrák alapvetései

Példa

Tervezés (szerkezet):



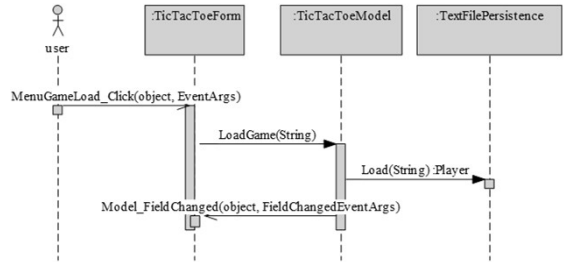
ELTE IK, Komponens alapú szoftverfejlesztés

8:19

Szoftver architektúrák alapvetései

Példa

Tervezés (végrehajtás):



ELTE IK, Komponens alapú szoftverfejlesztés

8:20

Szoftver architektúrák alapvetései

Open/closed principle

- *Open/closed principle* (OCP) kimondja, hogy a programegységek nyitottak a kiterjesztésre, de zártak a módosításra
 - a programszerkezetet úgy kell megvalósítanunk, hogy új funkcionalitás bevezetése ne a jelenlegi programegységek átirását igényelje, sokkal inkább újabb programegységek hozzáadását
 - a kiterjesztést általában öröklődés segítségével valósítjuk meg (pl. absztrakt osztály, vagy interfész bevezetésével)
- A SRP és az OCP kiegészítik egymást, mivel az új funkcionalitás egy új felelősség bevezetését jelenti, így általában egyszerre szegjük meg a két elvet

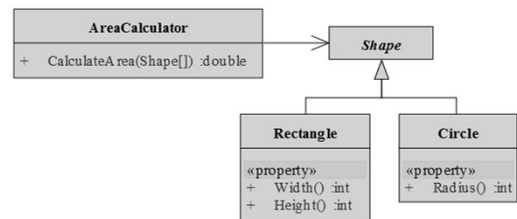
ELTE IK, Komponens alapú szoftverfejlesztés

8:21

Szoftver architektúrák alapvetései

Open/closed principle

Példa: Modellizzunk geometriai alakzatokat (**Shape**), speciálisan téglalapot (**Rectangle**) és kört (**Circle**). Alakzatok gyűjteményének ki tudjuk számítani a területét egy területszámító segítségével (**AreaCalculator**).



ELTE IK, Komponens alapú szoftverfejlesztés

8:22

Szoftver architektúrák alapvetései

Open/closed principle

```
double CalculateArea(Shape[] shapes) {
    double area = 0;
    foreach (Shape shape in shapes) {
        if (shape is Rectangle) {
            Rectangle r = shape as Rectangle;
            area += r.Width * r.Height;
        }
        if (shape is Circle) {
            Circle c = shape as Circle;
            area += Math.Pow(c.Radius, 2) * Math.PI;
        }
    }
    return area;
}
```

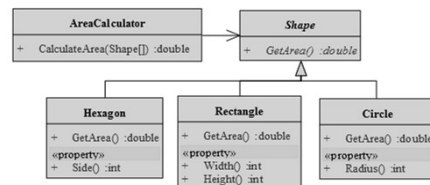
ELTE IK, Komponens alapú szoftverfejlesztés

8:23

Szoftver architektúrák alapvetései

Open/closed principle

- egy új alakzat (pl. hatszög) hozzáadása esetén módosítanunk kell a metódust, új ágat bevezetve
- ha minden alakzat ki tudja számítani a saját területét, akkor az új alakzattal egy új terület képletet lehetne megadni, és nem kellene módosítani a területszámítást



ELTE IK, Komponens alapú szoftverfejlesztés

8:24

Szoftver architektúrák alapvetései

Open/closed principle

```
double CalculateArea(Shape[] shapes) {
    double area = 0;
    foreach (Shape shape in shapes) {
        area += shape.GetArea();
    }
    return area;
}
...
class Hexagon : Shape {
    ...
    public override double GetArea() {
        return ...;
    }
}
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:25

Szoftver architektúrák alapvetései

Példa

Feladat: Készítsünk egy táblajáték programot, amelyben két játékos küzdhet egymás ellen.

- tegyük könnyen módosíthatóvá a Tic-Tac-Toe játékot, engedélyezzük nagyobb játéktábla használatát, és a játékszabályok módosítását származtatás segítségével
- a szükséges mezőket elérhetővé tesszük leszármazottak számára (**protected**), de nem engedélyezzük teljes felüldefiniálásukat (**readonly**)
- kiemeljük a módosítható működést segédműveletté (**StepGameInternal**), és felüldefiniálhatóvá tesszük (**virtual**)
- lehetőségünk van egy új játék, a potyogós amőba bevezetésére öröklődéssel (**ConnectFourModel**)

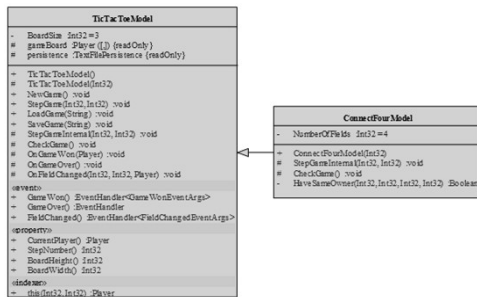
ELTE IK, Komponens alapú szoftverfejlesztés

8:26

Szoftver architektúrák alapvetései

Példa

Tervezés (modell szerkezet):



ELTE IK, Komponens alapú szoftverfejlesztés

8:27

Szoftver architektúrák alapvetései

Liskov substitution principle

- A *Liskov substitution principle* (LSP) értelmében bármely típus (osztály) példánya helyettesíthető altípusának egy példányával úgy, hogy közben a program tulajdonságai (funkcionális és nem funkcionális követelményei) nem változnak
- megszorításokat tesz a szignatúrára
 - elvárja a paraméterek kontravarianciáját, a visszatérési értékek kovarianciáját
 - tiltja a kivételek típusának bővítését
- megszorításokat tesz a viselkedésre
 - elvárja az invariánsok megtartását
 - tiltja az előfeltételek erősítését, az utófeltételek gyengítését

ELTE IK, Komponens alapú szoftverfejlesztés

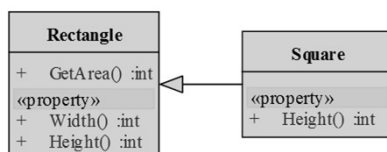
8:28

Szoftver architektúrák alapvetései

Liskov substitution principle

Példa: Modellezzünk téglalap (**Rectangle**) és négyzet (**Square**) alakzatokat, ahol a négyzet egy speciális esete a téglalaprak.

- mindkettőnek megadhatjuk a szélességét/magasságát, és lekérdezhetjük a területét
- a négyzet esetén a magasságot egyenlővé tesszük a szélességgel



ELTE IK, Komponens alapú szoftverfejlesztés

8:29

Szoftver architektúrák alapvetései

Liskov substitution principle

```
class Rectangle { // téglalap
    public virtual int Height { get; set; }
    // felüldefiniálható tulajdonság
    public int Width { get; set; }

    public int GetArea() { return Width * Height; }
}
...
class Square : Rectangle { // négyzet
    public override int Height {
        get { return Width; }
        set { Width = value; }
    } // mindig a magasságot vesszük alapul
}
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:30

Szoftver architektúrák alapvetései

Liskov substitution principle

```

Rectangle rec = new Rectangle {
    Width = 4;
    Height = 6;
}
Assert.AreEqual(24, rec.GetArea());
// elvárt viselkedés

rec = new Square { Width = 4; Height = 6; }
// kicseréljük a példányt a leszármazott
// típusra

Assert.AreEqual(24, rec.GetArea());
// ennek hatására nem az elvártak megfelelően
// viselkedik
    
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:31

Szoftver architektúrák alapvetései

Példa

Feladat: Készítsünk egy táblajáték programot, amelyben két játékos küzdhet egymás ellen.

- mivel a potyogós amőba nem teljesíti a Tic-Tac-Toe-val szembeni elvárásokat, alakítsuk át a modell szerkezetét
- vezessük be az absztrakt táblajátékot (**BoardGameModel**), amelynek leszármazottja lesz a Tic-Tac-Toe játék (**TicTacToeGame**)
- a két játék egyedül a lépés végrehajtását (**StepGameInternal**), illetve az ellenőrzést (**CheckGame**) definiálja, minden mászt az őosztály biztosít
- a programot felbonthatjuk komponensekre a tevékenységi körök és az absztrakció mentén,

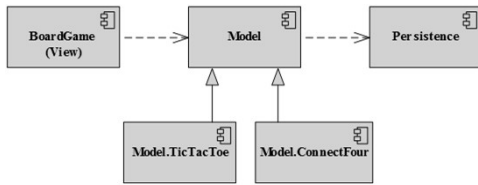
ELTE IK, Komponens alapú szoftverfejlesztés

8:32

Szoftver architektúrák alapvetései

Példa

Tervezés (komponensek):



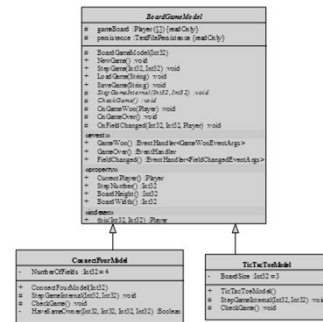
ELTE IK, Komponens alapú szoftverfejlesztés

8:33

Szoftver architektúrák alapvetései

Példa

Tervezés (modell szerkezet):



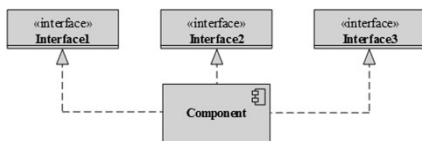
ELTE IK, Komponens alapú szoftverfejlesztés

8:34

Szoftver architektúrák alapvetései

Interface segregation principle

- Az *Interface segregation principle* (ISP) szerint egy kliensnek sem szabad olyan interfészről függenie, amelynek műveleteit nem használja
- ezért az összetett interfészeket célszerű több, kliens specifikus interfészre felbontani, így azok csak a szükséges műveleteiket érthetik el
- a műveletek megvalósításait továbbra is összeköthetjük



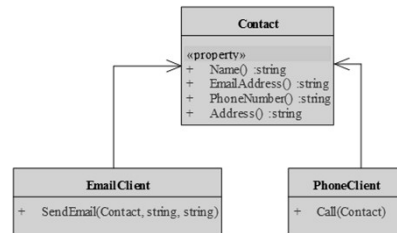
ELTE IK, Komponens alapú szoftverfejlesztés

8:35

Szoftver architektúrák alapvetései

Interface segregation principle

Példa: Modellezzünk egy névjegyet (**Contact**), ahol kliensek küldhetnek e-mailt (**EmailClient**), illetve hívhatják (**PhoneClient**) a címzett az adatok alapján.



ELTE IK, Komponens alapú szoftverfejlesztés

8:36

Szoftver architektúrák alapvetései

Interface segregation principle

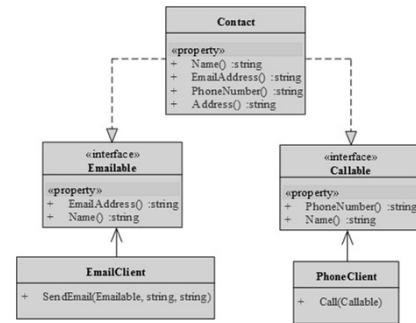
- mindkét kliens hozzáfér olyan információkhoz, amelyre nincs szüksége
- ha az email, vagy a telefonszám formátuma módosul, az kihatással lesz mindkét kliensre
- ha bármely kliens hatáskörét kiterjesztenék további elemekre, akkor annak meg kell valósítania a névjegy minden tulajdonságát
- ezért célszerű kiemelni a két funkcionalitást külön interfészekbe (**Emailable**, **Callable**), így az egyes kliensek csak a számukra szükséges funkcionalitást látják, a névjegy pedig megvalósítja az interfészeket

ELTE IK, Komponens alapú szoftverfejlesztés

8:37

Szoftver architektúrák alapvetései

Interface segregation principle



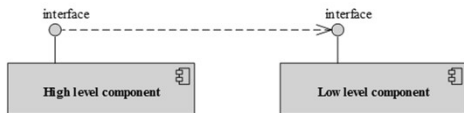
ELTE IK, Komponens alapú szoftverfejlesztés

8:38

Szoftver architektúrák alapvetései

Dependency inversion principle

- A *Dependency inversion principle* (DIP) a függőségek kezelésével szemben fogalmaz meg elvárás, amelynek értelmében:
- magasabb szintű programegységek nem függhetnek alacsonyabb szintű programegységtől, mindkettőnek az absztrakciótól kell függenie
- az absztrakció nem függhet a részletektől, a részletek függhenek az absztrakciótól



ELTE IK, Komponens alapú szoftverfejlesztés

8:39

Szoftver architektúrák alapvetései

Dependency inversion principle

- A DIP nagy fokú modularitást ad a programnak, mivel az absztrakció mentén bármely komponens megvalósítása könnyen cserélhetővé válik, így a függőségek könnyen módosíthatóak
- különösen tesztelés során hasznos, mivel így a függőség könnyen kiváltható (*mocking*)
- A megvalósítás tekintetében:
- az osztály mezői a konkrét osztályok helyett az absztrakció példányát tartalmazzák
- a konkrét osztályok az absztrakció segítségével lépnek kapcsolatba egymással, a konkrét osztályok szükség esetén átalakíthatóak
- szigorú esetben konkrét osztályokból nem örökölhetünk, és már megvalósított metódust nem definiálunk felül

ELTE IK, Komponens alapú szoftverfejlesztés

8:40

Szoftver architektúrák alapvetései

Dependency injection

- A végrehajtás során a maga szintű komponens (*kliens*) által használt alacsony szintű komponensnek (*szolgáltatás*) az adott helyzetnek megfelelő megvalósítását használjuk fel
- erről általában nem dönthet sem a kliens, sem a szolgáltatás, mivel ők nem ismerik a körülményeket, ezért egy külső komponensnek kell megadnia a megvalósítást
- így mind a magas szintű, mind az alacsony szintű programegységek példányosítása a külső programegység feladata, ezt nevezzük a *kontroll megfordításának* (*inversion of control*)
- általános technikái: *gyártó* (*factory*), *szolgáltatáskereső* (*service locator*), *függőség befecskendezés* (*dependency injection*)

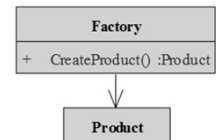
ELTE IK, Komponens alapú szoftverfejlesztés

8:41

Szoftver architektúrák alapvetései

Factory

- A *gyártó* (*factory*) egy olyan objektum, amelynek feladata más objektumok előállítására műveletek segítségével
- lehetőséget ad objektumok példányosításánál:
 - a konkrét példányosítandó típus kiválasztására
 - a példányosítással járó kódismétlés kiküszöbölésére
 - fel nem fedhető környezeti információk használatára
- lehetővé teszi a konkrét legyártott típus cseréjét a *gyártó művelet* (*factory method*) és az *absztrakt gyártó* (*abstract factory*) segítségével



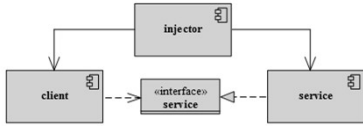
ELTE IK, Komponens alapú szoftverfejlesztés

8:42

Szoftver architektúrák alapvetései

Dependency injection

- *Függőség befecskendezés (dependency injection, DI)* esetén a kliens biztosít egy átadási pontot a szolgáltatás interfésze számára, ahol a végrehajtás során a szolgáltatás megvalósítását adhatjuk át
- az átadási pont lehet: *konstruktor, metódus* (beállító művelet), *interfész* (a kliens megvalósítja a beállító műveletet)
- a befecskendést végző komponens (*injector*) lehet egy felsőbb réteg, vagy a rétegződéstől független környezeti elem



ELTE IK, Komponens alapú szoftverfejlesztés

8:43

Szoftver architektúrák alapvetései

Dependency injection

```

• Pl.:
interface IService // szolgáltatás interfésze
{
    Double Compute(Double value);
}
...
class LogService : IService
// a szolgáltatás egy megvalósítása
{
    public Double Compute(Double value) {
        return Math.Log(value);
    }
}
    
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:44

Szoftver architektúrák alapvetései

Dependency injection

```

• Pl.:
class Client { // kliens
    private IService service; // függőség

    public Client(IService s) {
        this.service = s;
    } // konstruktor befecskendezés

    public void Execute() {
        ...
        v = this.service.Compute(v); // felhasználás
        ...
    }
}
    
```

ELTE IK, Komponens alapú szoftverfejlesztés

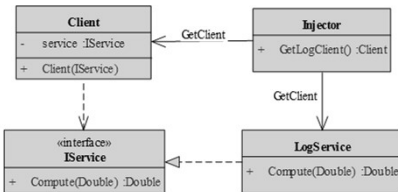
8:45

Szoftver architektúrák alapvetései

Dependency injection

```

• Pl.:
class Injector { // befecskendező
    public Client GetLogClient() {
        return new Client(new LogService());
    }
}
    
```



ELTE IK, Komponens alapú szoftverfejlesztés

8:46

Szoftver architektúrák alapvetései

Példa

Feladat: Készítsünk egy táblajáték programot, amelyben két játékos küzdhet egymás ellen.

- emeljük ki a nézet és a perzisztencia absztrakcióját, és adjunk alternatív megvalósításokat
 - vezérlő alapú felület (**ControlsForm**), rajzolt felület (**DrawingForm**)
 - szöveges fájl perzisztencia (**TextFilePersistence**), bináris fájl perzisztencia (**BinaryFilePersistence**)
- az absztrakció mentén újabb komponenseket emelhetünk ki, így minden réteg tekintetében moduláris lesz az alkalmazás
 - a fő komponens (**BoardGame**) csak a belépési pontot definiálja, és elvégzi a függőségek befecskendezését

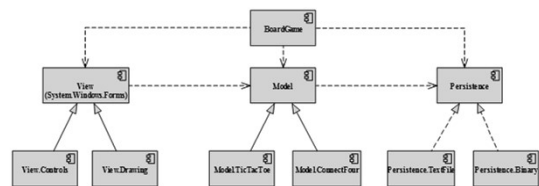
ELTE IK, Komponens alapú szoftverfejlesztés

8:47

Szoftver architektúrák alapvetései

Példa

Tervezés (komponensek):



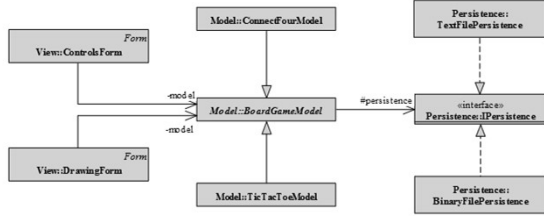
ELTE IK, Komponens alapú szoftverfejlesztés

8:48

Szoftver architektúrák alapvetései

Példa

Tervezés (szerkezet):



ELTE IK, Komponens alapú szoftverfejlesztés

8:49

Szoftver architektúrák alapvetései

Inversion of Control container

- Az IoC tároló (Inversion of Control container) egy függőségekkel rendelkező szolgáltatások tárolására alkalmas komponens
- a típusokat (elsősorban) interfész alapján azonosítja, és az interfészhez csatolja a megvalósító osztályt
- a tárolóba történő regisztrációkor (Register) megadjuk a szolgáltatás interfészét és megvalósításának típusát (vagy példányát)
- a szolgáltatást interfész alapján kérjük le (Resolve), ekkor példányosul a szolgáltatás
 - amennyiben a szolgáltatásnak függősége van, a tároló azt is példányosítja

ELTE IK, Komponens alapú szoftverfejlesztés

8:50

Szoftver architektúrák alapvetései

Inversion of Control container

```

Pl.:
interface IVisualization {
    void PrintComputation();
}

class ConsoleVisualization : IVisualization {
    private IService service; // függőség

    public ConsoleVisualization(IService s) {
        this.service = s;
    } // konstruktor befejezés

    public void PrintComputation() { ... }
}
    
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:51

Szoftver architektúrák alapvetései

Inversion of Control container

```

Pl.:
Container container = new Container();
// tároló példányosítása

container.Register<IService, LogService>();
container.Register<IVisualization,
    ConsoleVisualization>();
// szolgáltatások regisztrációja

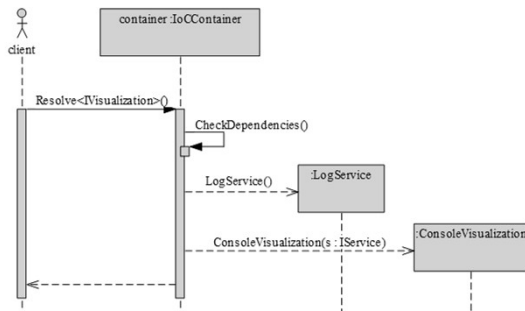
IVisualization visualization =
    container.Resolve<IVisualization>();
// szolgáltatás lekérése (példányosítással)
// egy ConsoleVisualization példányt, és benne
// egy LogService példányt kapunk vissza
    
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:52

Szoftver architektúrák alapvetései

Inversion of Control container



ELTE IK, Komponens alapú szoftverfejlesztés

8:53

Szoftver architektúrák alapvetései

Inversion of Control container

- A Unity programcsomag egy általánosan használható IoC tárolót biztosít, amely lehetővé teszi a regisztráció konfigurációs fájlban történő elvégzését, pl.:


```

<configuration>
    ...
    <unity ...>
        <container>
            <register type="IService"
                mapTo="LogService" />
            ...
        </container>
    </unity>
</configuration>
            
```

ELTE IK, Komponens alapú szoftverfejlesztés

8:54

Szoftver architektúrák alapvetései

Inversion of Control container

- Előnyei:
 - lehetőséget ad szolgáltatások dinamikus időben történő specifikálására, így a futtatás optimalizálására és a megvalósítások cseréjére
- Hátrányai:
 - a szolgáltatások dinamikus betöltése, példányosítása ronthatja a teljesítményt, és futási idejű hibákhoz vezethet, mivel nem ismert előre minden függőség (*Service Locator antipattern*)
 - biztonsági kockázatot jelenthet, mivel megengedi a kód befecskendezést
 - újabb függőséget jelent a kliensek számára