

Objektumorientált tervezési elvek

Szoftvertechnológia

Dr. Simon Balázs

BME, IIT

Tartalom

- OO fogalmak
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Don't Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Tell, Don't Ask (TDA)
- Law of Demeter (LoD)

00 fogalmak

OO fogalmak

- **Osztály: típus (type)**

- definiálja egy objektum metódusait/függvényeit (**viselkedését**, szolgáltatásait)
- a mezők/attribútumok/változók a viselkedést támogatják, és az értékeik határozzák meg egy objektum **állapotát**

- **Objektum: példány (instance)**

- egy osztály egy példánya

- **Statikus tagok (static members): osztály szintű tagok**

- ezek a metódusok és mezők az osztályhoz tartoznak
- csak egyetlen példány van belőlük, amely minden objektumra közös
- statikus függvényeknek nincs this pointere, nem tudják közvetlenül elérni a példány szintű tagokat

- **Példány tagok (instance members): objektum szintű tagok**

- ezek a metódusok és mezők az objektumokhoz tartoznak
- objektumonként külön példány van belőlük
- példány metódusoknak közös az implementációja, de van egy implicit nulladik paraméterük: a **this** pointer (az aktuális objektum)

OO fogalmak

- **Absztrakció (abstraction):**

- az adott kontextusban felesleges részletek elhanyagolása
- a világ objektumai leképezhetők objektumokra a programban

- **Osztályozás (classification):**

- közös tulajdonságokkal és közös viselkedéssel bíró dolgok csoportosítása
- a közös tulajdonságokat és a közös viselkedést az osztály írja le

- **Egységbezárás (encapsulation):**

- egy osztálynak nem szabad engednie, hogy kívülről közvetlenül hozzáférjenek a mezőikhez
- csak metódusokon keresztül szabad
- a mezőknek privátnak kell lenniük

- **Öröklődés (inheritance):**

- a leszármazott osztály újrahasznosítja az ős viselkedését
- az öröklődés “az-egy” kapcsolat a leszármazott és az ős között
- fontos:
 - az öröklődést csak a viselkedés újrahasznosítására használjuk!
 - soha ne használjunk öröklődést az adatok újrahasznosítására! (helyette: delegáció)

- **Polimorfizmus (polymorphism):**

- a hívónak ne kelljen törődnie azzal, hogy egy objektum típusa az ős vagy annak valamelyik leszármazottja
- megvalósítása: virtuális függvények és azok felüldefiniálása

- Láthatóság:
 - **private** (-): csak az adott osztályon belül elérhető
 - **protected** (#): csak az adott osztály és leszármazottai számára elérhető
 - **public** (+): bárki számára elérhető, aki az osztályt ismeri
 - **package** (~): az osztály csomagján belül elérhető
- **Virtuális (virtual) metódus:**
 - virtuális függvények felüldefiniálhatók (override) a leszármazott osztályokban
 - így lehet kiterjeszteni (extend) az őt viselkedését
- **Absztrakt (abstract) metódus:**
 - implementáció nélküli virtuális függvény
- **Absztrakt (abstract) osztály:**
 - absztrakt osztály **nem példányosítható**
 - általában van legalább egy absztrakt függvénye, de ez nem szükséges feltétel
- **Interfész (interface):**
 - függvények halmaza
 - definiálja az interfészt implementáló osztályoktól elvárt viselkedést (szerződést)
- **Osztály interfésze:**
 - az osztály publikus függvényeinek halmaza

■ Csatolás (coupling):

- az egyes modulok/komponensek/osztályok/függvények közötti függőség mértéke
- a közöttük lévő kapcsolat erőssége
- a *laza csatolás (low coupling)* előnyös a karbantarthatóság szempontjából: egy változás csak kis mértékben hat ki a rendszer más részeire

■ Kohézió (cohesion):

- annak a mértéke, hogy a modulok/komponensek/osztályok elemei mennyire tartoznak össze
- a bennük lévő elemek közötti kapcsolat erőssége
- az *erős kohézió (high cohesion)* előnyös a karbantarthatóság szempontjából: az összetartozó funkciók egy helyen vannak lokalizálva

OO tervezési elvek

- Egy szoftver folyamatosan változik
- A jól megtervezett szoftvert könnyű változtatni
- Akkor van gond, ha a követelmények úgy változnak, hogy azokat nehéz beépíteni a szoftverbe
- Ha a szoftver nem tudja követni a változó követelményeket, akkor az a szoftver tervének hibája
- Későbbi fejlesztések megsérthetik az eredeti tervezési filozófiát, és azután csak eszkalálódnak a problémák
 - ezért fontos a dokumentáció

Rossz terv

- A szoftver rosszul van megtervezve, ha:
 - nehéz rajta változtatni, mert a változtatást a rendszer sok különböző részén el kell végezni (a terv merev)
 - a változások a szoftver olyan részeit is elrontják, amelyekre nem számítottunk (a terv törékeny)
 - másik szoftverben nehéz újrahasznosítani az adott szoftver egyes részeit, mert nem lehet leválasztani őket a többi komponenstől (a terv nem mobilis)
- A rossz tervezés oka: túl sok függőség a szoftver egyes részei között
- Megoldás:
 - csökkentsük a részek közötti függőségeket
 - változtassuk meg a függőségek irányát úgy, hogy ne a problémás és gyakran változó részek felé mutassanak

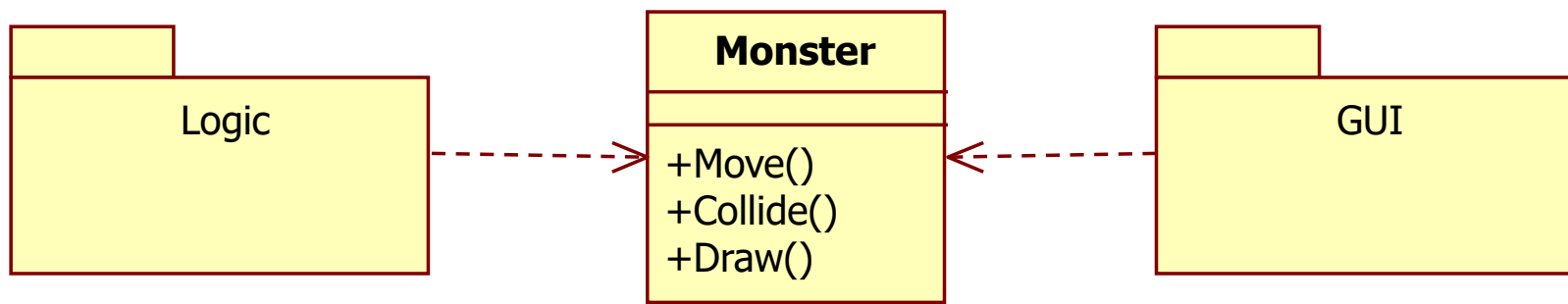
- A jó OO tervezés öt alapelve:
 - Single responsibility (egyetlen felelősség elve)
 - Open-closed (nyílt-zárt elv)
 - Liskov substitution (Liskov-féle helyettesíthetőség elve)
 - Interface segregation (interfészek szétválasztásának elve)
 - Dependency inversion (függőségek megfordításának elve)
- Robert C. Martin vezette be ezeket
- Ezek az elvek elősegítik a későbbi karbantarthatóságot és bővíthetőséget azáltal, hogy csökkentik a szoftver egyes részei közötti függőségeket

Single Responsibility Principle (SRP)

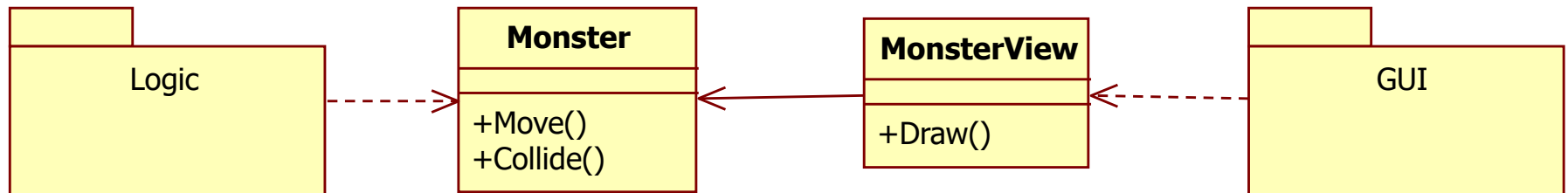
- Egy osztálynak csak egy oka legyen a változásra
 - (Robert C. Martin)
- Felelősség = ok a változásra (\neq a biztosított szolgáltatások)
- Vagyis: ha egy osztály egynél több felelősséggel rendelkezik, akkor az osztályt több osztályra kéne szétbontani

- Ha egy osztálynak több felelőssége van, válasszuk szét a felelősségeket:
 - implementációs szinten (ha szét lehet őket választani)
 - interfész szinten (ha nem lehet őket szétválasztani)
- Implementációs szinten:
 - külön osztályok
 - körkörös függőségek nélkül
- Interfész szinten:
 - felelősségekként külön interfészek
 - az eredeti osztály implementálja az interfészeket
- Előny: a függőségek a problémás felelősségektől elfelé mutatnak

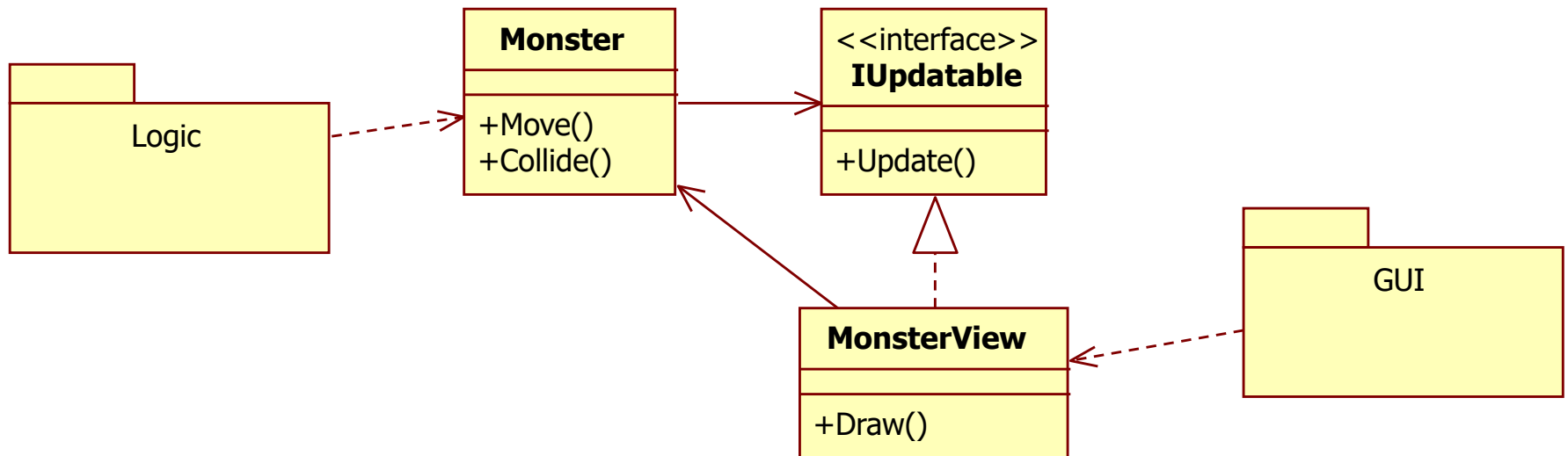
Példa az SRP megsértésére



SRP megoldás I.



SRP megoldás II.



Változás valószínűsége

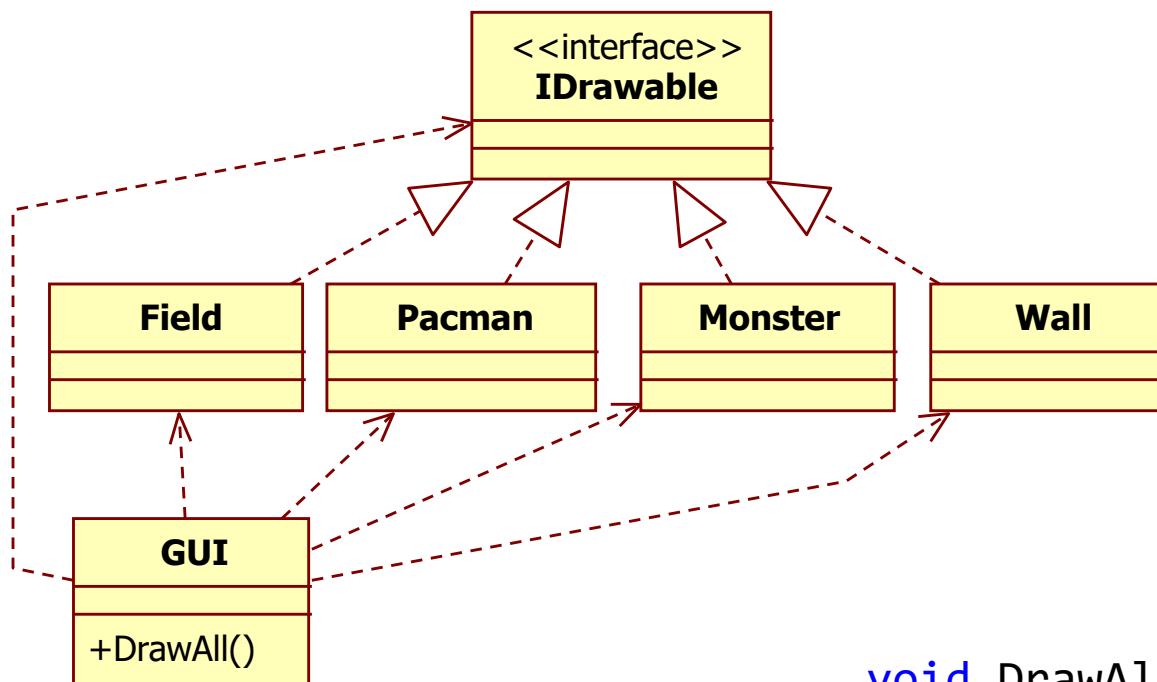
- Nem mindig egyértelmű, hogy több ok is lehet a változásra
- Lehet, hogy a változási igény sosem következik be
- A tervezéskor meg kell becsülni a változás bekövetkezésének valószínűségét
 - múltbeli tapasztalataink és a szakterülettel kapcsolatos ismereteink alapján
- Feleslegesen ne tervezzünk olyan változásra, amelynek bekövetkezése nagyon alacsony valószínűségű
 - YAGNI = You Ain't Gonna Need It

Open/Closed Principle (OCP)

- A szoftver részeinek (osztályok, modulok, függvények, stb.) nyitottnak kell lennie a kiterjesztésre, de zártnak a módosításra
 - (Bertrand Meyer)
- Nyitott a bővítésre: a modul viselkedése kiterjeszthető, hogy megfeleljünk a változó követelményeknek
- Zárt a módosításra: a modul kiterjesztése nem igényelheti a már meglévő kódok átírását

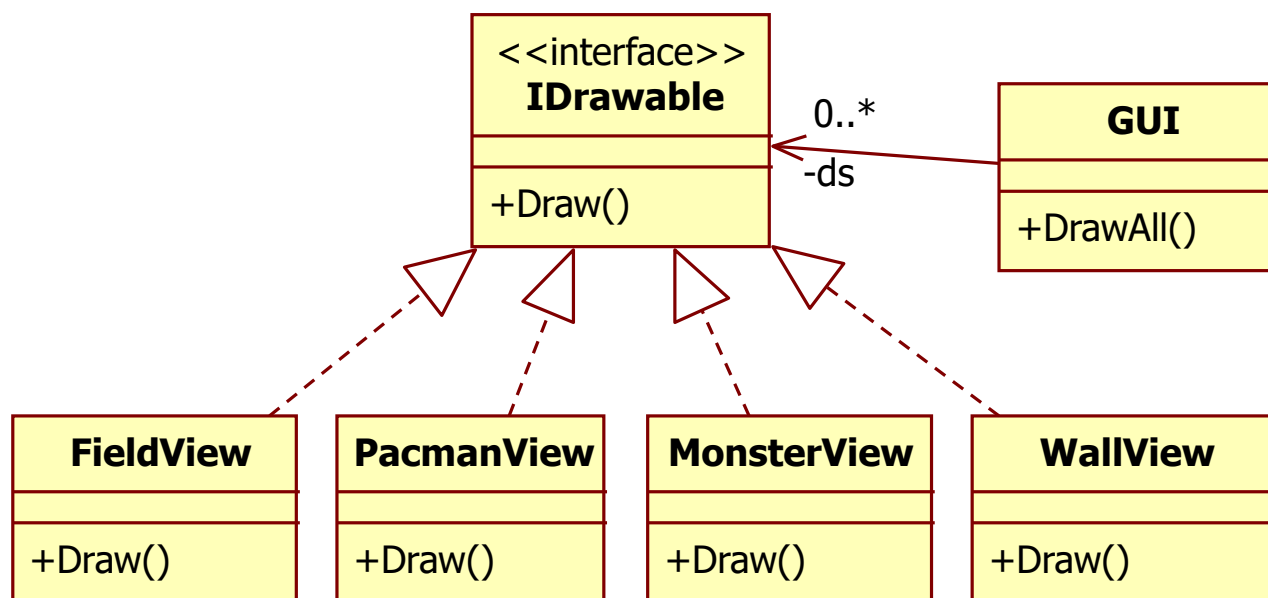
- Készüljünk fel a változásokra
- Nyitott a bővítésre:
 - új leszármazott osztályok
 - metódusok felüldefiniálása
 - polimorfizmus
 - delegáció
- Zárt a módosításra:
 - a már meglévő kód nem változik
 - csak hibajavítások vannak
- A viselkedés kiterjesztése új kód írásával történik, nem a meglévő kód átírásával

Példa az OCP megsértésére



```
void DrawAll(List<IDrawable> ds) {  
    foreach (var d in ds) {  
        if (d is Field) {  
            // ...draw field...  
        } else if (d is Pacman) {  
            // ...draw pacman...  
        } else if ...  
    }  
}
```

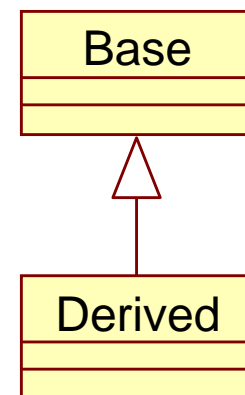
OCP megoldás



```
void DrawAll(List<IDrawable> ds) {
    foreach (var d in ds) {
        d.Draw();
    }
}
```

Liskov Substitution Principle (LSP)

- A leszármazottaknak behelyettesíthetőnek kell lennie az ős típusba
 - (Barbara Liskov)
- Bármely leszármazott (Derived) osztály egy példánya behelyettesíthető egy olyan helyre, ahol az ős (Base) egy példányát használjuk, anélkül, hogy az ős használója észrevenné a különbséget
- Öröklődés:
 - Derived egyfajta Base
 - minden Derived típusú objektum egyben Base típusú is
 - ami igaz a Base-re, igaz a Derived-ra is
 - a Base általánosabb dolgot reprezentál, mint a Derived
 - a Derived egy speciálisabb dolgot reprezentál, mint a Base
 - bárhol ahol a Base használható, egy Derived is használható



- A Liskov-elv fontossága akkor válik szembetűnővé, ha a megsértésének következményeivel találkozunk
- A Liskov-elv megsértése:
 - a leszármazott nem úgy viselkedik, ahogy azt az őstől elvárnánk
 - tipikusan:
 - öröklés az adatok újrahasznosítása céljából (pl. négyzet-téglalap probléma)
 - paraméterek/visszatérési érték értékkészletének megsértése
- Ennek következményei:
 - a speciális leszármazott felismeréséhez explicit típuslekérdezés szükséges
 - `is`, `instanceof`, `dynamic_cast`, stb.
 - vagyis: az OCP elvet is megsértjük

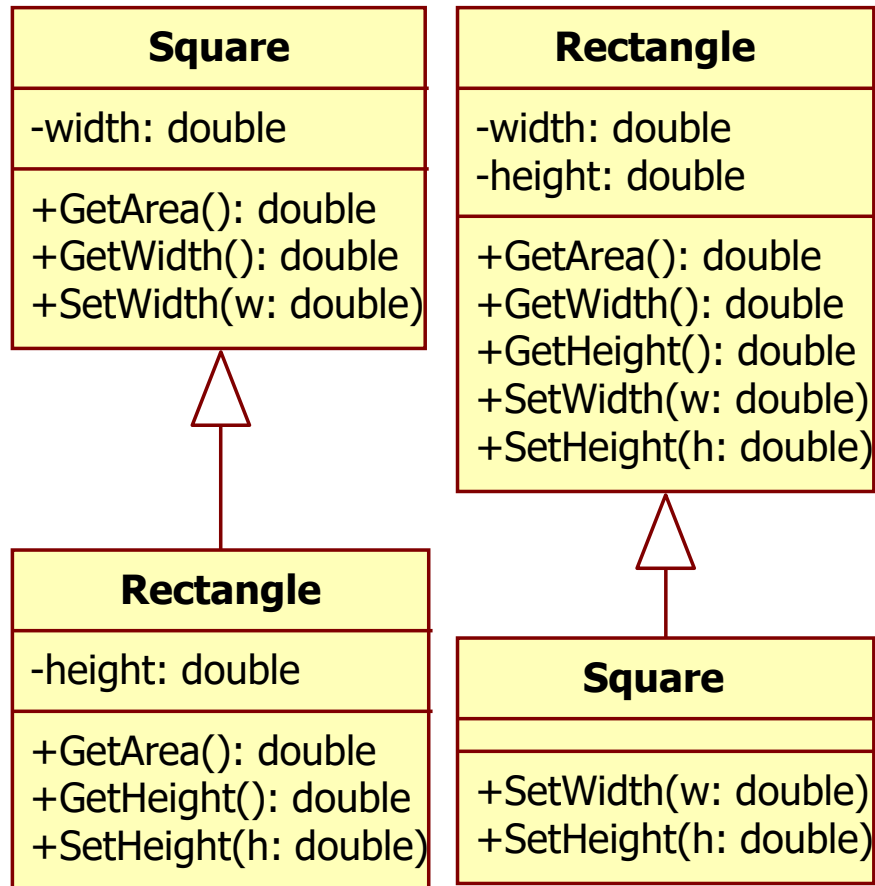
LSP megsértésének következménye: típusellenőrzés

- Ha egy leszármazott megsérti a Liskov-elvet, egy tapasztalatlan fejlesztő sietségében lehet, hogy explicit típusellenőrzéssel oldja meg a problémát
- Például:

```
void Draw(Shape s) {  
    if (s is Rectangle) DrawRectangle((Rectangle)s);  
    else if (s is Ellipse) DrawEllipse((Ellipse)s);  
}
```

- Itt a Draw megsérti az OCP elvet is, mert a Shape minden leszármazottját ismernie kell
- A Liskov-elv megsértése általában az OCP megsértését is magával vonja

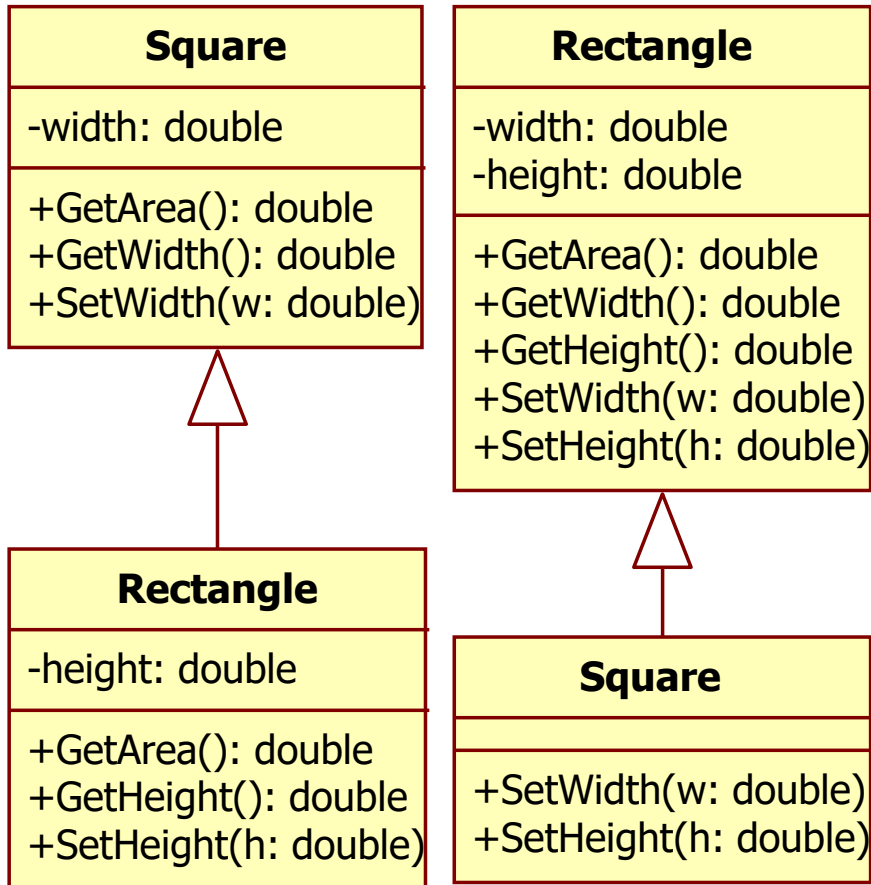
Melyik tervezés jobb?



Egyik sem: mindkettő sérti a Liskov-elvet

```
class Square : Rectangle {  
    void SetWidth(double w) {  
        base.SetWidth(w);  
        base.SetHeight(w);  
    }  
    void SetHeight(double h) {  
        base.SetWidth(h);  
        base.SetHeight(h);  
    }  
    // ...  
}
```

Mindkettő sérti a Liskov-elvet



```
void TestSquare(Square s) {
    s.SetWidth(5);
    Debug.Assert(s.GetArea() == 25);
}
```

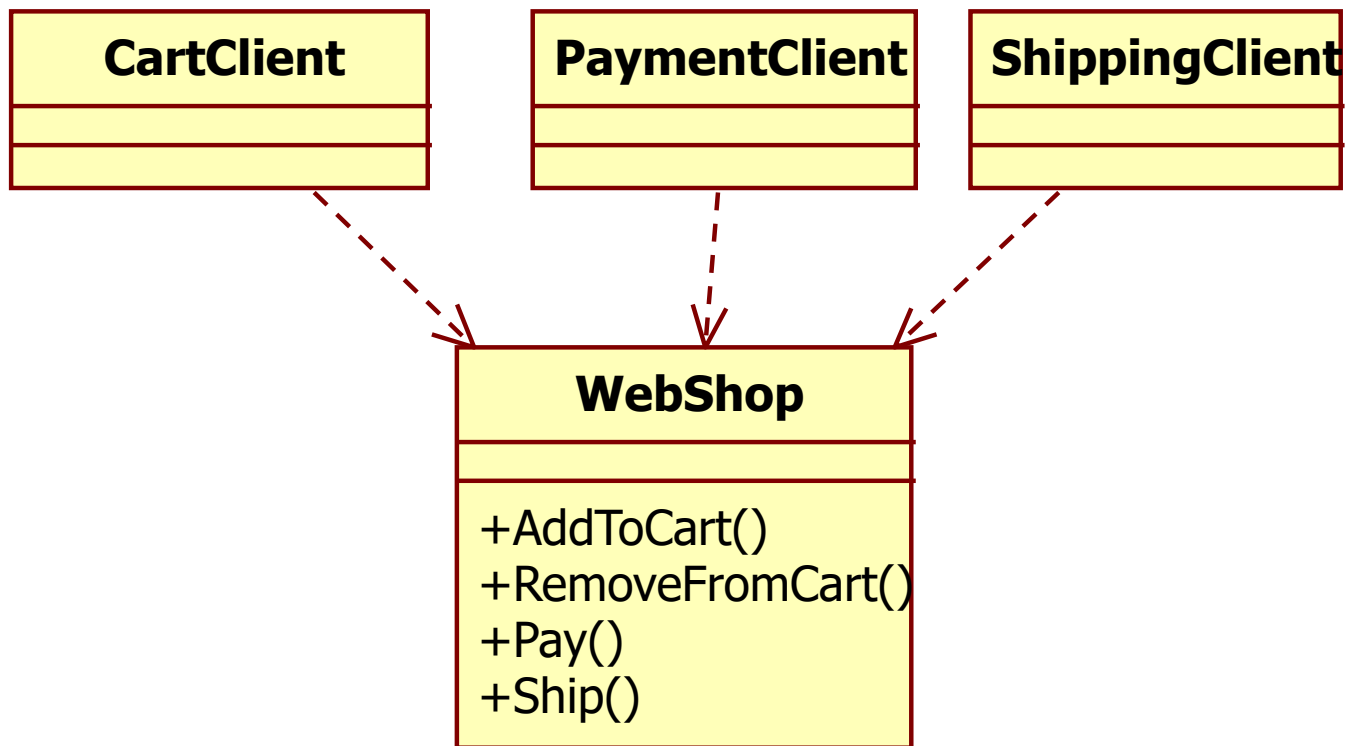
```
void TestRectangle(Rectangle r) {
    r.SetWidth(5);
    r.SetHeight(4);
    Debug.Assert(r.GetArea() == 20);
}
```

- A négyzet-téglalap probléma gyökere:
 - a négyzet matematikai (adat) szempontból egyfajta téglalap
 - de a négyzet viselkedése más, mint a téglalapé (a SetWidth-nek nem kéne a magasságot is állítania)
 - és az Objektumorientáltságban a viselkedés számít
- **Fontos:**
 - Az öröklődés nem az adatok újrahasznosítására való!
 - Az öröklődés célja a viselkedés újrahasznosítása!

Interface Segregation Principle (ISP)

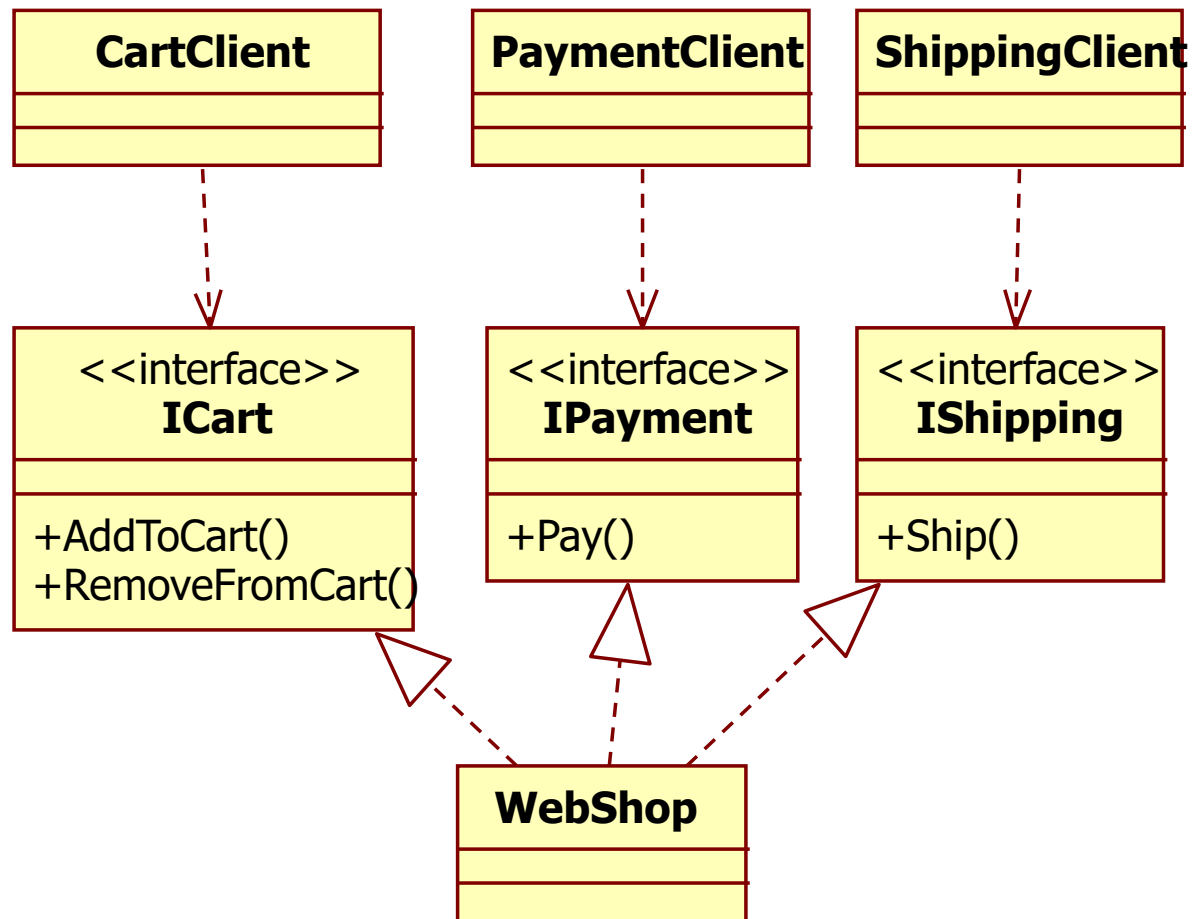
- A klienseket nem kötelezhetjük arra, hogy olyan metódusoktól függjenek, amelyeket nem használnak
 - (Robert C. Martin)
- Az ISP elfogadja, hogy egyes osztályoknak nagy és nem kohézív interfésze van szüksége
- De a klienseknek nem szükséges a teljes osztályt ismerni
- Helyette: elég, ha a kliensek csak kohézív interfésszel rendelkező absztrakciókat ismernek

Példa az ISP megsértésére



- A kliens csak azoktól a metódusoktól függjön, amelyeket ténylegesen meghív
- Daraboljuk fel a nagy kövér osztály interfészét több kliens-specifikus interfészre
- A nagy kövér osztály implementálja ezeket az interfészeket
- A kliensek csak azoktól az interfészekről függjenek, amelyekre szükségük van
- Így a kliensek nem függnek többé a nem használt metódusoktól
- És így a kliensek függetlenek lehetnek egymástól

ISP megoldás

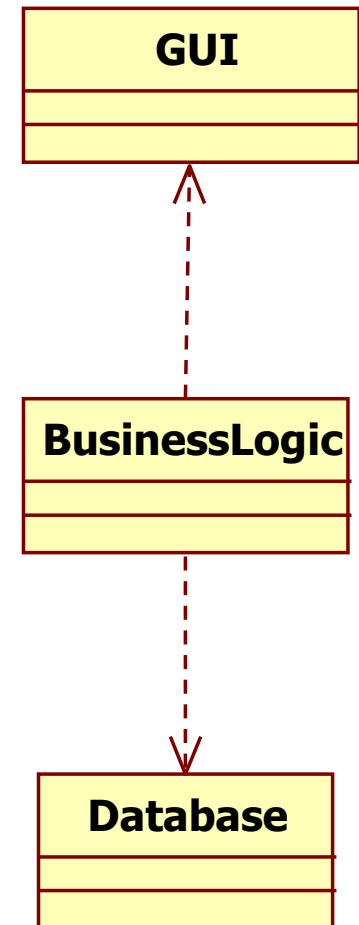


Dependency Inversion Principle (DIP)

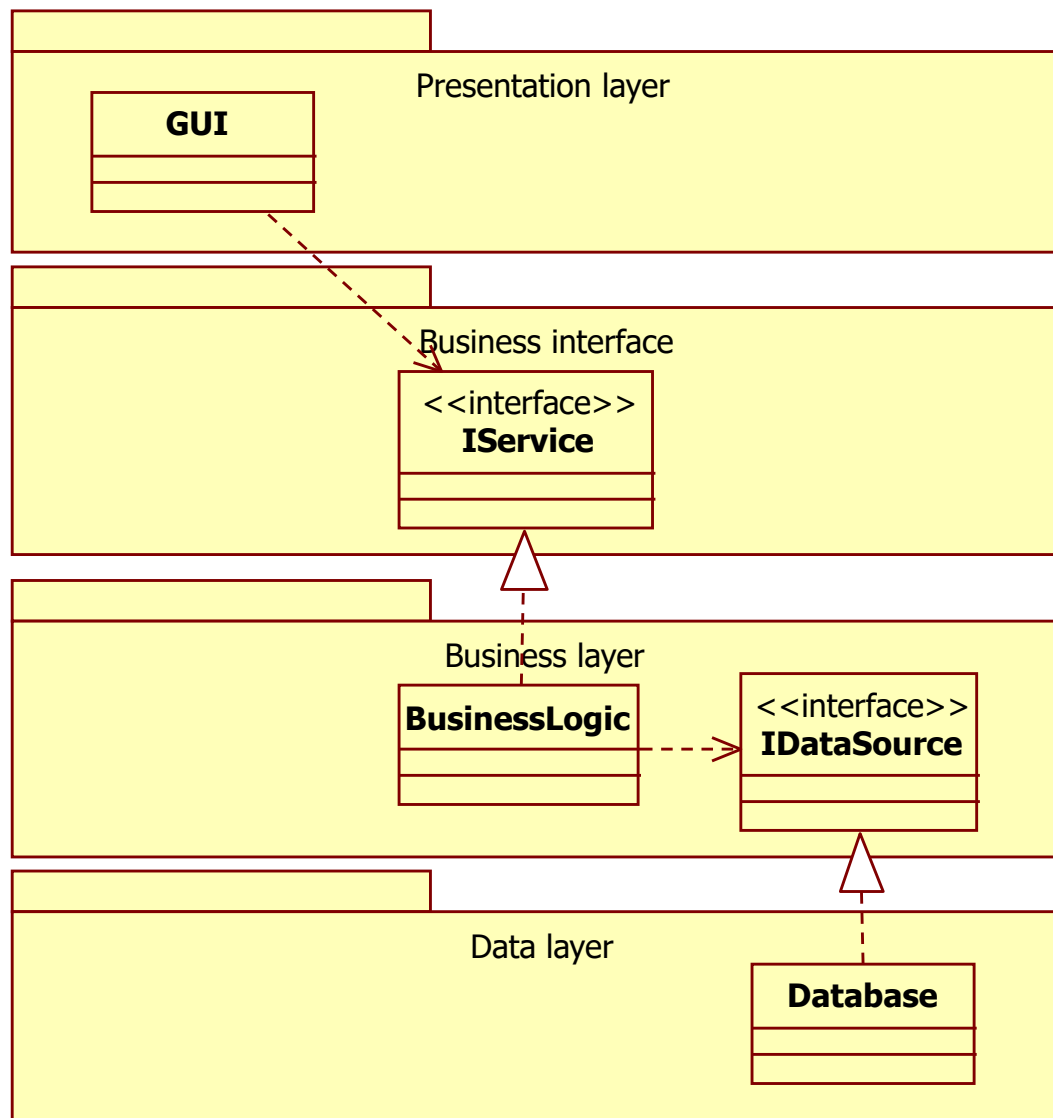
- Magas szintű modulok ne függjenek alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.
- Absztrakciók ne függjenek a részletektől. A részletek függjenek az absztrakcióktól.
 - (Robert C. Martin)
- Egy alkalmazás modulokból/komponensekből áll
- Egy természetes módja a fejlesztésnek az, hogy megírjuk az alacsony szintű modulokat (input-output, hálózat, adatbázis, stb.) majd beépítjük őket a magasabb szintű modulokba
- Azonban ez rossz: egy alacsony szintű modul változása előidézheti egy magasabb szintű modul megváltozását

Példa a DIP megsértésére

- Példa a rossz tervezésre:
 - Üzleti logika ---> Adatbázis
 - Üzleti logika ---> GUI
 - a konkrét adatbázis technológia vagy konkrét GUI technológia változhat
 - a változás az üzleti logikára is kihat
- Dependency Inversion Principle:
 - fordítsuk meg a függőség irányát: az alacsony szintű modulok a magasabb szintű modulok által definiált absztrakcióktól függenek
- A DIP egy másik értelmezése:
 - absztrakcióktól függünk



DIP megoldás



- Vannak olyan helyzetek, amikor egy konkrét osztály interfésze változik
- És ennek a változásnak meg kell jelennie az osztályt reprezentáló interfészben
- Egy ilyen változás felszivárog, és megtöri az absztrakt interfész által biztosított izolációt
- DE:
 - a kliens osztály definiálja az interfészt és annak szolgáltatásait, mert ő tudja, mire van szüksége
 - az interfész csak akkor változhat, ha a kliens kezdeményezi a változást!

Don't Repeat Yourself (DRY)

- Minden tudásnak egyetlen és egyértelmű helyen kell megjelennie a rendszerben
- Csökkentsük az ismétlődést
- A duplikáció rossz:
 - ha az ismétlődő részben valamit meg kell változtatni, akkor azt az összes előfordulási helyen meg kell változtatni
 - nagy a valószínűsége annak, hogy néhány helyen kimarad a változás
- Ha Ctrl+C-t nyomunk, gondoljunk arra, hogy inkább egy függvényt kéne ebből a kódrészletből készíteni

Példa a DRY megsértésére

```
class Producer {
    private Queue queue;
    public void Produce() {
        lock (queue) {
            string item = "hello";
            queue.Enqueue(item);
        }
    }
}

class Consumer {
    private Queue queue;
    public void Consume() {
        lock (queue) {
            string item = queue.Dequeue();
        }
    }
}
```

DRY megoldás: a Queue szálbiztossá tétele

```
class Queue {  
    private object mutex = new object();  
    private List<string> items = new List<string>();  
  
    public void Enqueue(string item) {  
        lock (mutex) {  
            items.Add(item);  
        }  
    }  
  
    public string Dequeue() {  
        lock (mutex) {  
            string result = items[0];  
            items.RemoveAt(0);  
            return result;  
        }  
    }  
}
```

DRY megoldás: a Queue szálbiztossá tétele

```
class Producer {  
    private Queue queue;  
    public void Produce() {  
        string item = "hello";  
        queue.Enqueue(item);  
    }  
}
```

```
class Consumer {  
    private Queue queue;  
    public void Consume() {  
        string item = queue.Dequeue();  
    }  
}
```

Single Choice Principle (SCP)

- Bármikor arra van szükség, hogy a rendszer alternatívákat támogasson, akkor ideális esetben egyetlen helyen koncentrálódjon az alternatívák kezelése
- Ez a DRY és az OCP következménye
- Aka.: Single Point Control (SPC)
 - az alternatívák teljes listája egyetlen helyen koncentrálódjon

Tell, don't ask (TDA)

- A metódusokat úgy hívjuk meg, hogy előtte nem vizsgáljuk a hívott objektum állapotát vagy típusát
- Az állapotot ellenőrző kódnak a célobjektumhoz kéne tartoznia, ez az ő felelőssége lenne

Példa a TDA megsértésére

```
class Pacman {  
    void Step() {  
        Field next = field.GetNext();  
        if (next.IsFree) {  
            next.Accept(this);  
        } else {  
            Thing other = next.GetThing();  
            other.Collide(this);  
        }  
    }  
}
```

TDA megoldás

```
class Pacman {  
    void Step() {  
        Field next = field.GetNext();  
        field.Remove(this);  
        next.Accept(this);  
    }  
}
```

```
class Field {  
    void Accept(Thing t) {  
        if (this.thing != null) {  
            this.thing.Collide(t);  
        } else {  
            this.thing = t;  
        }  
    }  
}
```

- A TDA megsértése a DRY megsértését is jelenti
- Sok problémát tud okozni:
 - a feltétel ellenőrzése lemaradhat néhány helyen
 - konkurencia problémák
- Hagyjuk a feltételek ellenőrzését a hívott objektumra
- A TDA megsértése azt jelenti, hogy a felelősségek rossz helyen vannak

Demeter-törvény (Law of Demeter: LoD)

- „Ne állj szóba idegenekkel!”
- Egy objektum függvénye csak az alábbi objektumok függvényeit hívhatja:
 - saját objektum
 - bejövő paraméterek
 - saját objektum adattagjai
 - az általa létrehozott objektumok
- Egy metódus ne hívja egyéb más objektumok tagjait

Példa a LoD megsértésére:

Vagy:

```
this.field.GetNext().GetThing().Collide(this);
```

Vagy:

```
Field next = this.field.GetNext();
```

```
Thing thing = next.GetThing();
```

```
thing.Collide(this);
```

LoD lehetséges megoldások

//Elfogadható:

```
Field next = this.field.GetNext();  
next.Accept(this);
```

//Inkább:

```
this.field.MoveThingToNextField();
```

- Metódushívások láncolása esetén a lánc összes elemétől függünk
- Megoldás: delegálás
 - a lánc minden objektumába tegyünk egy függvényt, ami a lánc maradékát végrehajtja
- De vigyázzunk, hogy ez ne járjon a metódusok kombinatorikus robbanásával
 - ha mégis, akkor tervezzük át a dolgokat
- Csak az ismerős objektumokhoz delegáljuk a hívásokat!
- Így ha egy elem változik a láncban, az valószínűleg nem lesz hatással a lánc elejére

Összefoglalás

- OO fogalmak
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Don't Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Tell, Don't Ask (TDA)
- Law of Demeter (LoD)