Objektumorientált szoftvertervezés - 1. előadás
Objektumorientált tervezési elvek

OO alapfogalmak

- Osztály: **típus**
  - metódus/tagfüggvény → viselkedés
  - mező/attribútum/tagváltozó → állapot
- Objektum: **példány** (*instance*)
- Statikus tag (*static member*): **osztály szintű**
  - aláhúzással jelöljük, akkor is meghívhatók ha nem létezik még példány az osztályból
  - statikus metódus → nem is tudja elérni közvetlenül a példány szintű tagokat
- Példány tag (*instance member*): **objekt szintű**
  - **this** vagy **self** pointer: aktuális objektum

Objektumok egymásra hivatkozások

- **Asszociáció** (*association*) → a szemközti osztály egy példányát vagy attribútumait egy példányban tároljuk - jelölés: sima vonal
  - **tartalmazás** (*composition*) → erősebb kapcsolat, a tartalmazott objektumok is megszűnnek, ha a tartalmazó is megszűnik - jelölés: teli trapéz végű vonal, a tartalmazó felé
- **függőség** (*dependency*) - jelölés: szaggatott nyíl
  - kliens (nyil nélküli) függ a szervertől (nyíl hegyénél) (azaz a kliens használja a szervert, de csak ideiglenesen)
    - pl.: kliens paraméterként kapja csak meg a szerver objektumot, függvény-ből kilépés után megszűnik a kapcsolat!
- erősebb kapcsolat mindig implikálja a gyengébbet

Interfészek

- **Interfész** (*interface*) → névvel ellátott függvényhalmaz, elvárt viselkedés tartozik hozzá
  - Osztályok implementálják
  - Egy interfész nem példányosítható → önmagában nem rendelkezik semmi funkcionalitással
  - jelölés - üres háromszög végű szaggatott nyíl
- Osztály interfésze → rajta meghívható publikus függvények halmaza

Öröklődés

- **Öröklődés** (*inheritance*) - jelölés: üres háromszög végű folytonos nyíl
  - ősben definiált **virtuális függvények** felülírhatók (*override*) a leszármazottban
  - a felülírt függvényt a leszármazottban is fel kell tüntetni (UML)
- **Polimorfizmus** (*polymorphism*): ha egy kliens az ősön keresztül hív meg egy virtuális függvényt, akkor valójában a leszármazottban felüldefiniált függvény fog lefutni

- **Absztrakt** (*abstract*) **metódus**: olyan függvény, aminek nincs megvalósítása (majd a leszármazott valósítja meg) - jelölés: dőlt betűvel írt
- **Absztrakt** (*abstract*) **osztály**: ha egy osztály, legalább egy absztrakt függvényt tartalmaz, akkor absztrakt osztály lesz (de absztrakt függvény nélkül is lehet **absztrakt osztály**) - jelölése: dőlt betűvel írt név
    - nem példányosíthatóak→ mivel nem biztos, hogy minden metódusuk meg van valósítva
- **Konkrét** (*concrete*) **osztályok**: amik nem absztrakt osztályok
    - mindig példányosíthatóak, kötelezően implementálnak minden olyan függvényt, amit valamely ősük nem implementált

Láthatóságok

- **private** (-): csak az őt tartalmazó osztály fér hozzá az adott mezőhöz
- **protected** (#): csak az adott osztály és az ő leszármazottai férhetnek hozzá
- **public** (+): mindenki hozzáfér, akinek az osztályhoz hozzáférése van
- **package** (~): csak az adott csomagon belül látható

OO alapelvek

- **Absztrakció** (*abstraction*)
    - a világból csak a releváns dolgokat modellezzük
- **Osztályozás** (*classification*)
    - egymáshoz hasonló objektumokat egységesen, ugyanolyan osztállyal modellezzük, ez az osztály írja le a közös viselkedést és a közös állapotot
- **Egységbezárás** (*encapsulation*)
    - külvilág csak azt láthassa az objektum állapotából, amit szükséges
    - alapból minden privát legyen, metódusokon keresztül lehessen csak elérhető (metódus biztosítja a adatok konzisztenciáját)
- **Öröklődés** (*inheritance*)
    - hasonló viselkedésű osztályok közös tudása egy ősosztályba legyen
    - öröklés = viselkedés újrahasznosítása (NEM az adatoké)
- **Polimorfizmus** (*polymorphism*)
    - kliens számára mindegy legyen, hogy az ős vagy annak egy példánya akivel épp kommunikál
- **Kohézió** (*cohesion*)
    - egy osztály/modul tagjai mennyire tartoznak össze, egyes tagok közt mennyire erős a kapcsolat
    - ha túl kicsi a kohézió → külön osztályba kell őket szedni
- **Csatolás** (*coupling*)
    - egyes osztályok/modulok/függvények mennyire függenek egymástól
    - ha nagy a csatolás → egy rész változtatása a többit is magával vonzzaná

OO tervezési elvek

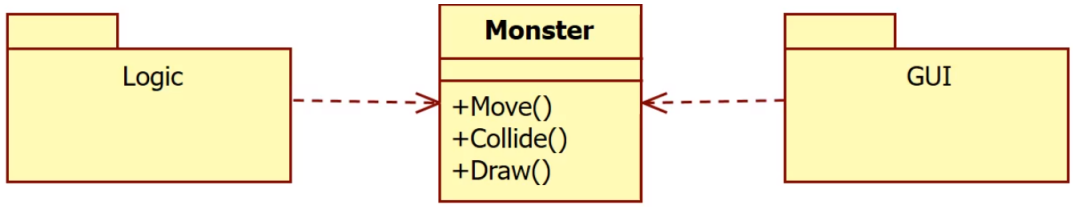
Követelmények változása

- rosszul tervezett szoftver
  - **merev** (*rigid*) → nehéz módosítani a szoftvert (sok helyen kell változtatni)
  - **törékeny** (*fragile*) → egy változtatás elront más részeket
  - **nem újrahasznosítható** (*immobile*) → nem átemelhetőek a részek
  - OK → túl sok a függőség a rendszer egyes részei között
  - CÉL → ezen függőségek csökkentése, függőségek a kevésbé problémás, ritkán változó részeknél legyenek
- dokumentáció → fontos az ellentmondásos tervezési döntések elkerülése végett
- változás valószínűsége
  - YAGNI = You Ain't Gonna Need It → olyanra felesleges felkészülni, amire nagyon kicsi a valószínűség
- jól tervezett szoftver → kevés

SOLID

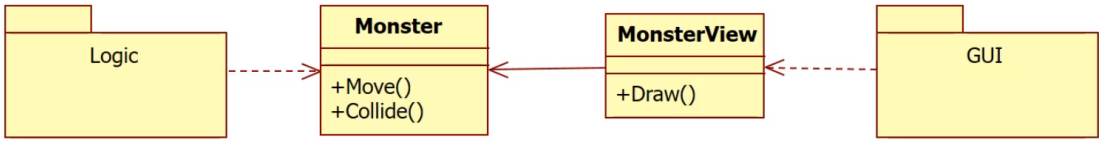
- **Cél**: karbantarthatóság, bővíthetőség, függőségek csökkentése
- A jó OO tervezés öt alapelve:

Single Responsibility Principle - SRP (*egyetlen felelősség elve*)

- 

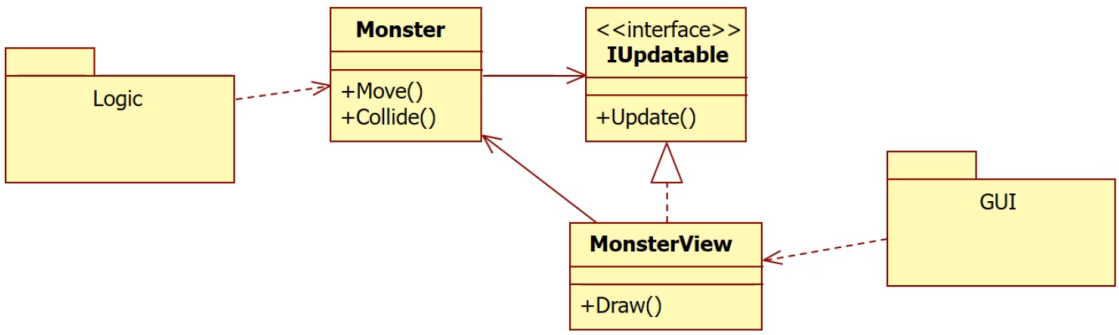
- ebben az esetben a Logika és a GUI is függ a Monster-től → a GUI változása a Logika változását is magával vonja (pl a Draw függvény átírása)
- Def: **Egy osztálynak cask egyetlen oka legyen a változásra, azaz egy felelőssége legyen!**
- javítás: több osztály létrehozása, több interfészre bontás

Pull jellegű megoldás → Grafika a rajzoláshoz mindig lekéri a modell állapotát

- 

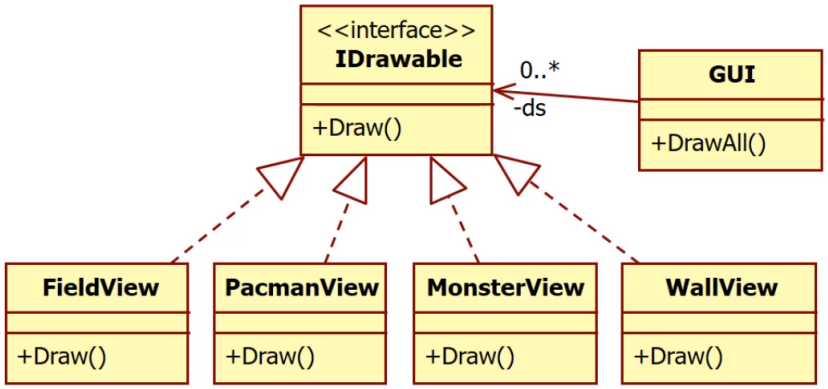
- logikai komponenstől függ a grafikai, de a grafikaitól nem függ a logikai (mivel a grafika többször változik mind a logika)
- hátrány: akkor is lekéri a gui a modell állapotát, ha nem is történt változás benne

Push jellegű megoldás → Modell jelzi a Grafikának, hogy változás történt

- 

- grafika lekérdezi a modell új állapotát
- előnye: csak változáskor történik újrarajzolás
- IUpdatable → a modell csak ezen az interfészen keresztül látja a grafikus komponenseket

Open-Closed Principly - OCP (*nyitottság-zártság elve*)

- Def: **A szoftver részeinek nyitottnak kell lennie a kiterjesztésre, de zártnak a módosításra**
- azaz úgy tudjunk bővíteni, hogy a meglévő részekhez nem nyúlunk hozzá
- 

-

Liskov Substitution Principle - LSP (*Liskov-féle helyettesítés elve*)

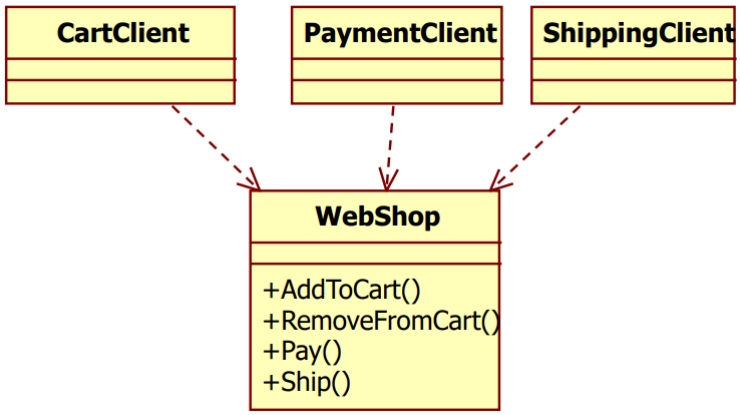
- Def: **A leszármazottaknak behelyettesíthetőnek kell lenniük az ősbe: nem sérthetik meg az őstől elvárt viselkedést**
- ok: kliens egy elvárt viselkedésre számít az őstől és leszármazottjaitól
  - nem lenne jó, ha rákéne kérdeznie a kliensnek a szerver típusára

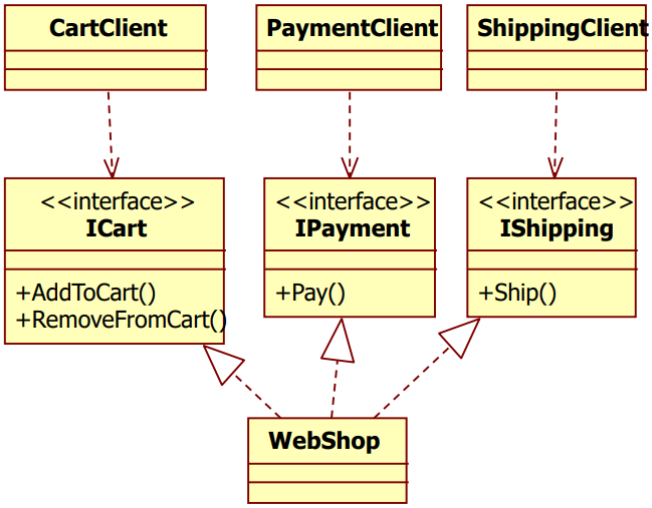
Design-by-Contract (DbC)

- Szerződés (contract):
  - **előfeltételek** (*pre-conditions*): mit várunk el a hívótól
  - **utófeltételek** (*post-conditions*): mit garantálunk a hívónak
  - **invariánsok** (*invariants*): mik állandóak
- DbC biztosítja, hogy a leszármazott ne sértse meg az ős szerződését
- **DbC szabályok**:
  - előfeltétel ugyanolyan vagy gyengíthető (max kevesebb dolgot várhat egy leszármazott)
  - utófeltétel ugyanolyan vagy erősíthető (max több mindent garantálhat egy leszármazott)
  - invariáns ugyanolyan vagy erősíthető (max több állandósága lehet egy leszármazottnak)

Interface Segregation Principle - ISP (*interfészek szétválasztásának elve*)

- Def: **A klienseket nem kötelezhetjük arra, hogy olyan metódusoktól függjenek, amelyeket nem használnak**
-

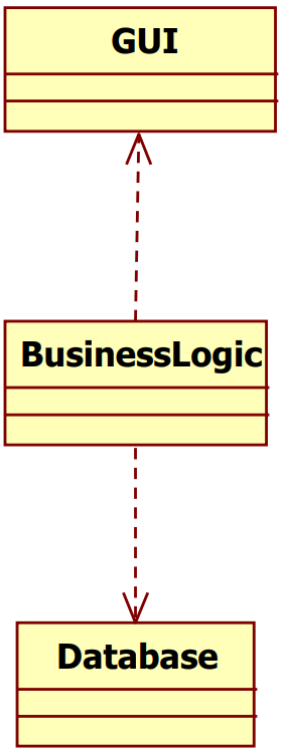


    - ez egy rossz megoldás, mert a kliensek olyan metódusoktól is függnek amik őket nem érintik
- 

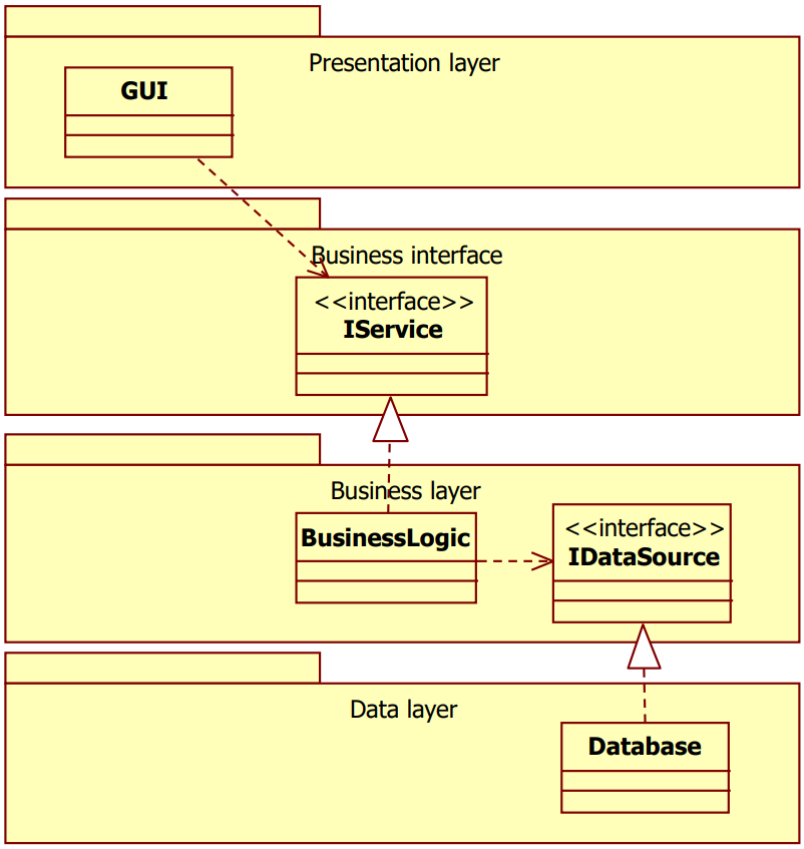
    - ez egy jó megoldás, így a WebShop megvalósítja az összes interfészt, és a kliensek csak a saját interfészüket ismerik, csak attól függenek

Dependency Inversion Principle (*függőségek megfordításának elve*)

- Def: **Magas szintű modulok ne függjenek alacsony szintű moduloktól: mindketten absztrakcióktól függjenek**
    - Magasabb szintű modulok definiáljanak egy interfészt, hogy mit várnak el az alacsonyabb szintű moduloktól
    - Az alacsonyabb szintű modulok ezt az interfészt implementálják → mindketten ettől az absztrakciótól fognak függeni
-



- ez egy hibás megoldás, mert az üzleti logika függ a GUI-tól, ami gyakran változhat, és az adatbázistól is függ, így az nem cserélhető



- ez egy jó megoldás, mert az üzleti logika és az adatbázis is csak az IDataSource-tól függ
- továbbá érdemes rétegekre szétválasztani a dolgokat

További OO tervezési elvek

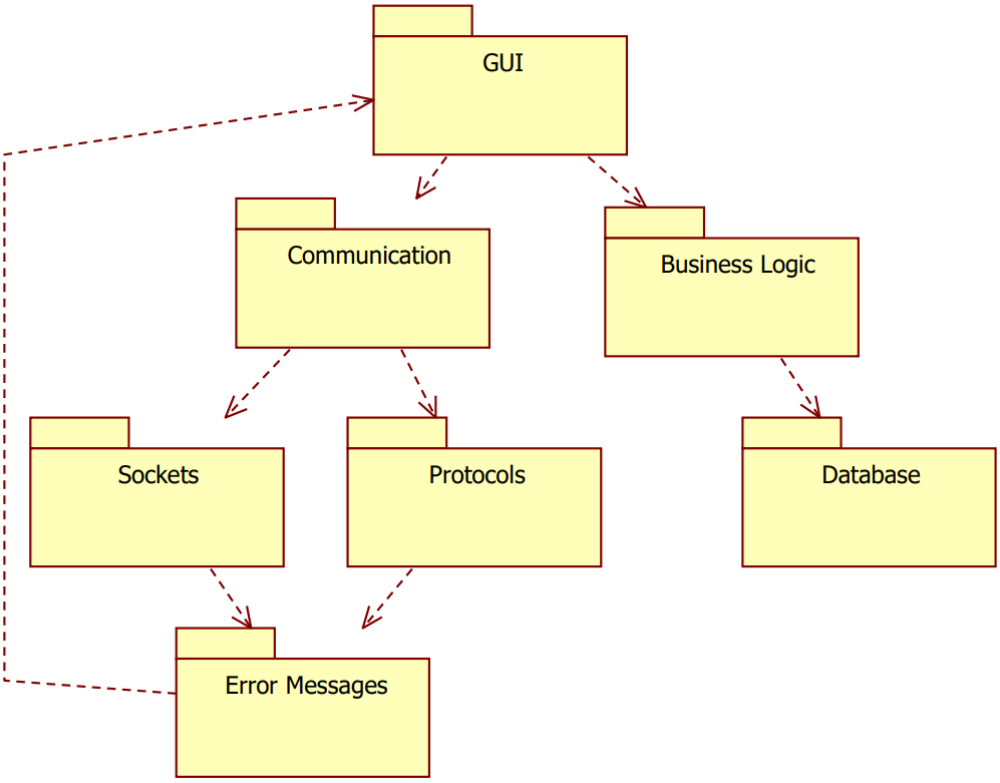
Stable Dependencies Principle - SDP (*stabil függőségek elve*)

- Def: **Mindig a stabilitás fele mutasson a függőség**
- stabilitás = mekkora munka az adott komponens változtatása (kikre van hatással a változása)
- amelyik komponenstől sok minden más komponens függ, az stabil

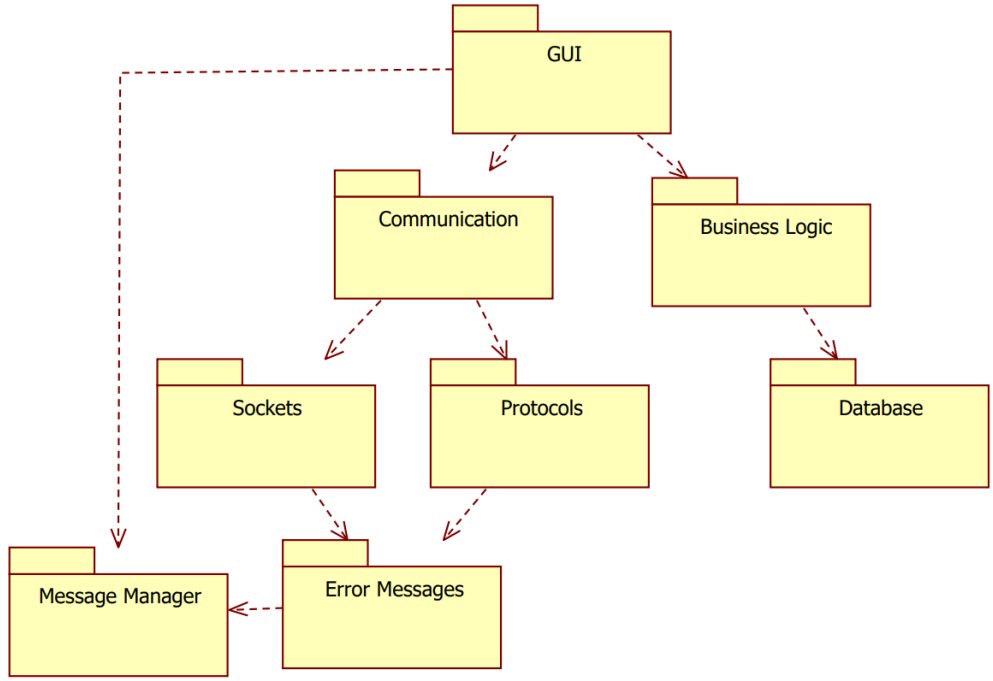
Stable Abstraction Principle - SAP (*stabil absztrakció elve*)

- Def: **A stabil csomagok absztrakt csomagok legyenek**
    - azaz a stabil csomagokat legyen könnyű kiterjeszteni

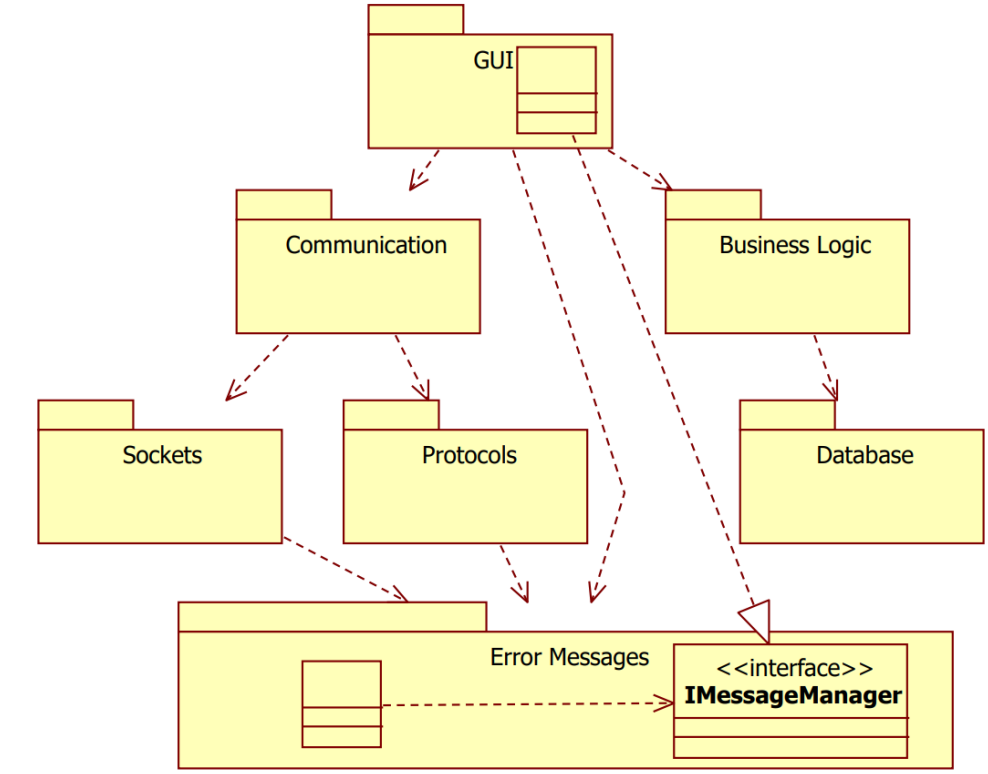
Acyclic Dependencies Principle - ADP (*aciklikus függőség elve*)

- Def: **A modulok, csomagok illetve függőségek között ne legyen körkörös függőség**
- 

    - ez egy rossz megoldás, mert körkörös függőséget tartalmaz, emiatt a körben levő bármely komponens megváltozása az egész kód újrafordítását vonja maga után
- körkörös függőség feloldása
    - új csomag bevezetésével



- DIP és ISP segítségével a függőség irányának megfordítása



Don't Repeat Yorself - DRY (*ne ismételjük magunkat*)

- Def: Ne ismételjük a kódot: minden tudást egyetlen egyértelmű helyen jelenjen meg a kódban
  - pl ismétlődő kód hibás → mindenhol javítani kell
- pl a lock ne a hívók felelőssége legyen, hanem pl a sor-é, akire vonatkozik a lock

Single Choice Principle - SCP / Single Point Control - SPC (*egyetlen helyen történő esetszétválasztás elve*)

- Def: **Ha sok esetett kell szétválasztania a rendszernek, az esetszétválasztás egyetlen helyen szerepeljen a kódban**
- DRY és OCP következménye
- pl: többnyelvűség biztosítása erőforrásfájlokkal

Tell, don't ask - TDA (*mondj, ne kérdezz*)

- Def: **Ne ellenőrizzük a hívott objektum típusát vagy belső állapotát, mielőtt meghívjuk annak egy metódusát**
- pl: NE a Pacman vizsgáljon mindent (szabad e a következő mező, van e ott szörny, ha igen megeteti magát vele)

Law of Demeter - LoD (*demeter törvény*)

- Def: **Ne beszélgess idegenekkel**
- **ismerős**: saját maga, paraméterként kapott objektumok, attribútumban tárolt objektumok, általa létrehozott objektumok
- **idegen**: mindenki más
- megoldás → mindig delegáljuk a hívási lánc maradék részét a következő objektumhoz (csak következő taggal lépnek kölcsönhatásra)

Common Closure Principle - CCP (*együtt változó osztályok elve*)

- Def: **együtt változó osztályok ugyanabba a csomagba kerüljenek, így a változás lokalizált**
- erős kohéziót segíti

Common Reuse Principle - CRP (közös újrahasznosítás elve)

- Def: **nem együtt használt osztályok külön csomagba kerüljenek**
- ISP alkalmazása csomagokra
- erős kohéziót segíti

Release Reuse Equivalency Principle - REP (*kiadott komponensek újrahasznosításának elve*)

- Def: egy általunk kiadott szoftverkomponenshez megfelelő verziózás és karbantartás szükséges