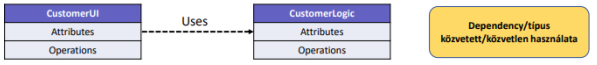
Gang-of-Four tervezési minták 1

# OO relációk

## Dependency - Függőség

* Két elem között akkor áll fenn, ha az egyik (a független) elem változása hatással van a másik (a függő) elemre.
* Kölcsönös függőség akkor van, ha mindegyik elem hat a másikra.



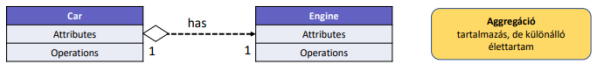
## Asszociáció

* Osztályok közötti tetszőleges viszony.
* Asszociációs kapcsolat áll fenn két osztály között, ha az egyiknek a saját helyes működéséhez ismernie kell a másikat.
  + Az egyik használja a másikat.
  + Az egyik tartalmazza a másikat.



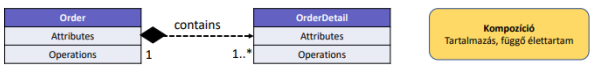
## Aggregáció

* Az asszociáció speciális esete, tartalmazási kapcsolat.
* A tartalmazó osztály példányai magukba foglalják a tartalmazott osztály egy vagy több példányát, ez a rész-egész kapcsolat.
* A tartalmazó és a tartalmazott osztály egymástól függetlenül létezhetnek.
* A tartalmazás lehet, erős illetve gyenge.
* **Erős aggregáció:** A részek élettartalma szigorúan megegyezik az egészével, ez a kompocízió.
* **Gyenge aggregáció:** Egyszerű/általános aggregáció.



## Kompozíció

* Másnéven **erős aggregáció**, tehát szigorú tartalmazási kapcsolat.
* Egy rész objektum csak egy egészhez tartozhat.
* **A tartalmazó és a tartalmazott életciklusa közös**: Például van egy User objektum, aminek van egy Address tulajdonsága. Ha a User objektum megszűnik, akkor megszűnik az Address is, de nem létezhet User objektum Address nélkül. Ezért közös az életciklusuk.



# Újrahasznosítható kód fogalma

* Időt spórolhatunk vele, mert kiküszöböli a már tökéletesen működő és használható kódrészletek újraírásának szükségességét.
* A hasonló funkciók kódját gyakran több projektben is fel lehet használni, hogy felgyorsítsuk a fejlesztés menetét.
* Kockázatok csökkentése azáltal, hogy egy már kipróbált, tesztelt kódot használtunk fel.
  + Ez garantálhatja a jó felhasználói élményt, így zökkenőmentesen is működhet.
* Megakadályozza a kód rohamos növekedését, ami lassú, erőforrás igényes is lehet.
  + Törölni kell a már nem használt kódrészleteket.
* **Nehézségek:**
  + Kommunikáció a projekt növekedése miatt.
  + Dokumentáció készítése, hogy például az adott kódrészletet hol lehet felhasználni újra.

A képen diagram látható

Automatikusan generált leírás

# Kompozíció és öröklés összehasonlítása

|  |  |  |
| --- | --- | --- |
|  | **Öröklés** | **Kompozíció** |
| **Kapcsolat** | Az autó egy jármű. (x egy y) | Az autónak van kormánykereke. (x-nek van y-ja) |
| **Cél** | Az öröklésnek a célja, hogy megtervezzük, hogy az adott osztály mi az | A kompozíciónak a célja, hogy az adott osztály mit csináljon. |
| **Egyéb** | Sokkal szorosabb a kapcsolat az objektumok között | Sokkal lazább a kapcsolata |
| Az ősosztály módosítása kihatással lehet a leszármazottakra. | Rugalmasabb, mert futásidőben tudunk módosítani. |

# SOLID elvek

### Single Responsibility

* Minden osztály egy dologért legyen felelős és azt jól lássa el.
* **Ha nem követjük, akkor:**
  + Spagetti kód, átláthatatlanság
  + Nagy méretű objektumok
  + Mindenért felelős alkalmazások és szolgáltatások

### Open/Closed elv

* Egy osztály legyen nyitott a bővítésre és a zárt módosításra, vagyis nem írhatunk bele, de származtathatunk tőle.
* **Ha nem követjük, akkor:**
  + Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetőek.
  + Leszármazott megírásakor módosítani kell az ősosztályt, ami tilos.
  + Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában.

### Liskov substitutable

* Ősosztály helyett utódpéldány legyen mindig használható.
* Compiler supported, hiszen OOP elv (polimorfizmus)
* Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.

### Interface seggregation

* Sok kis interfészt használjunk egy hatalmas mindent előíró interfész helyett.
* **Ha nem követjük, akkor:**
  + Egy osztályt létrehozunk valamilyen célból, megvalósítjuk az interfészt és rengeteg üres, fölösleges metódusunk lesz.
  + Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással.

### Dependency Inversion

* A függőségeket ne az őket felhasználó osztály hozza létre.
* Várjuk kívülről a példányokat interfészeken keresztül.
* **Példány megadására több módszer is lehetséges:**
  + Dependency Injection
  + Inversion of Control (IoC) container
  + Factory tervezési minta
* **Ha nem követjük, akkor**
  + Egymástól szorosan függő osztályok végtelen láncolata.
  + Nem lehet modularizálni és rétegezni.
  + Kód újrahasznosítás lehetetlen

### Egyéb elvek

* DRY, Don’t Repeat Yourself
* DDD = Domain Driven Design