MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR

Szoftvertechnológia gyakorlatok

GEIAL316-B2

Programtervezési minták

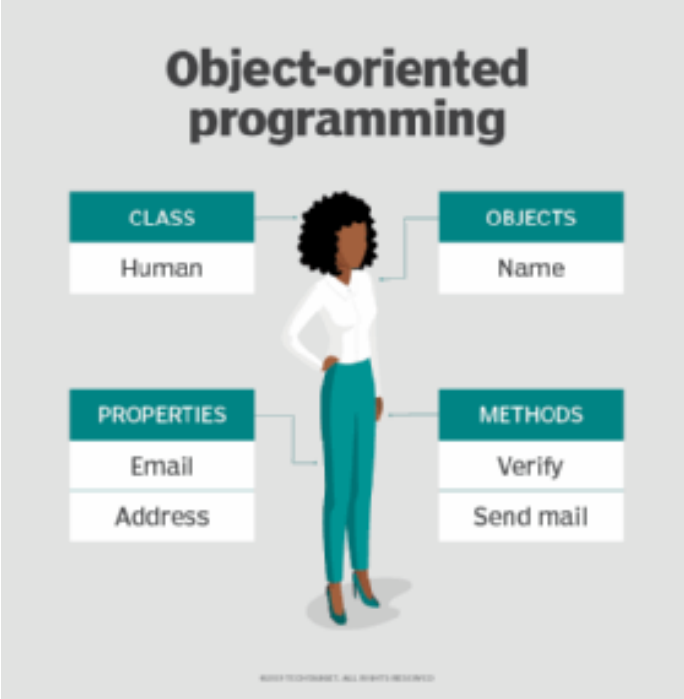
Dr. Tompa Tamás
egyetemi adjunktus
Általános Informatikai Intézeti Tanszék

Miskolc, 2025

OOP

- Egy **OO (objektum orientált) paradigmára** épülő szoftver **egymással kommunikáló objektumok összeségéből** épül fel

- **Objektum:**
  - **önálló entitás**, amely tulajdonságokkal, attribútumokkal rendelkezik
  - valós világot (annak egy részét) **modell**ezi
  - **van állapota**: attribútumok aktuális értékei
  - **van viselkedése**: rajta végrehajtható műveletek, ami módosítja az állapotát
  - **kapcsolata van** más objektumokkal



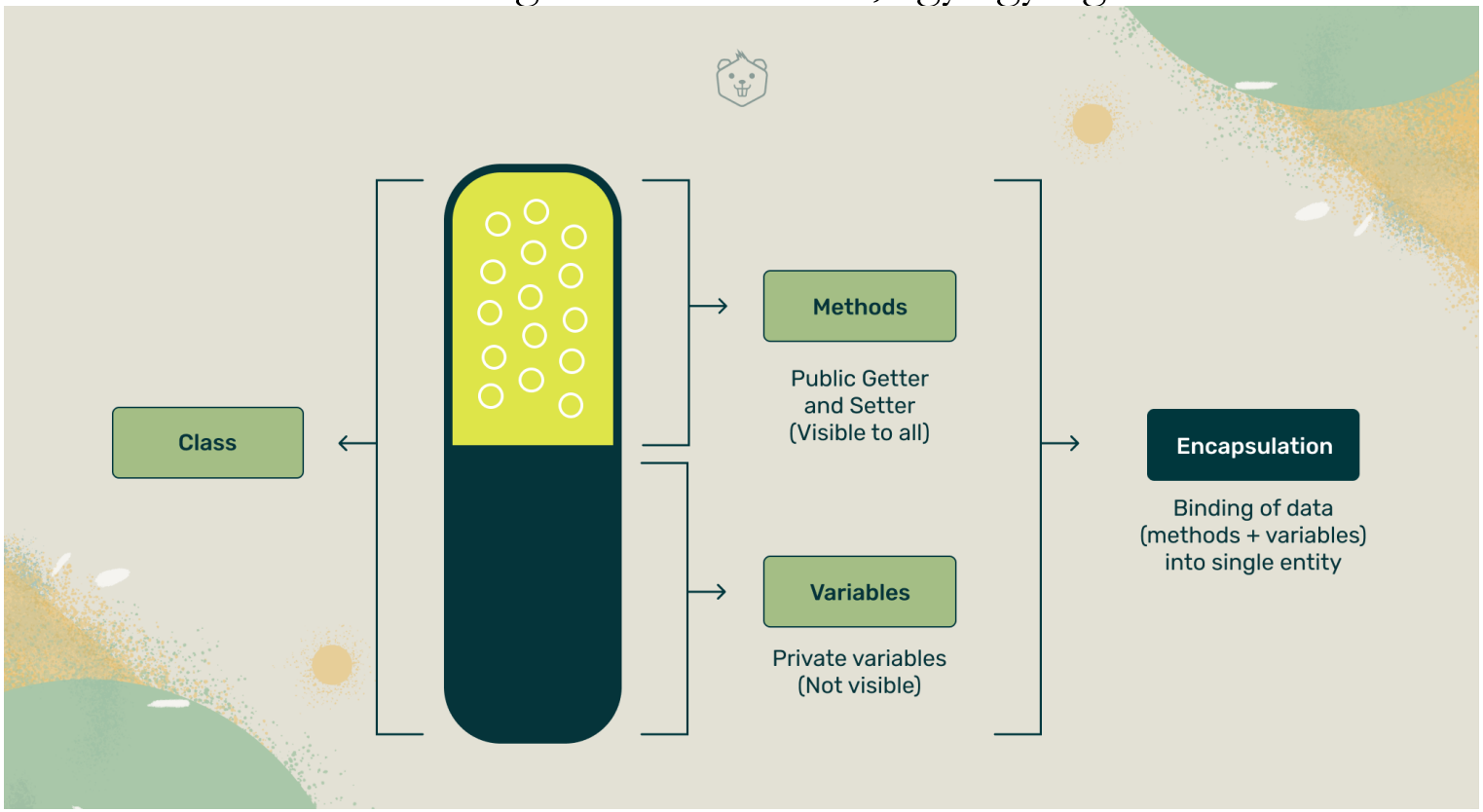
Forrás: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

OOP alapelvek

- **Egységbezárás (encapsulation)**
  - az osztály egységbe zárja az adott típusú objektumok adatait és az azokkal dolgozó műveleteket, egy egységként kezeli
- **Adatrejtés (information hiding)**
  - az osztály műveleteinek implementációja az osztályon kívül nem látható
- **Öröklődés (inheritance)**
  - egy osztály létrehozható úgy, hogy egy másik osztály leszármazottja, amely így örökli az ősosztály tulajdonságait
- **Polimorfizmus (polymorphism)**
  - többalakúság, bizonyos viselkedések működése függ a környezettől, metódus működése az adott osztálytól függ

OOP alapelvek

- **Egységbezárás (encapsulation)**
  - **az osztály egységbe zárja az adott típusú objektumok adatait** és az azokkal dolgozó műveleteket, egy egységként kezeli



Forrás: https://www.crio.do/blog/encapsulation-in-java/

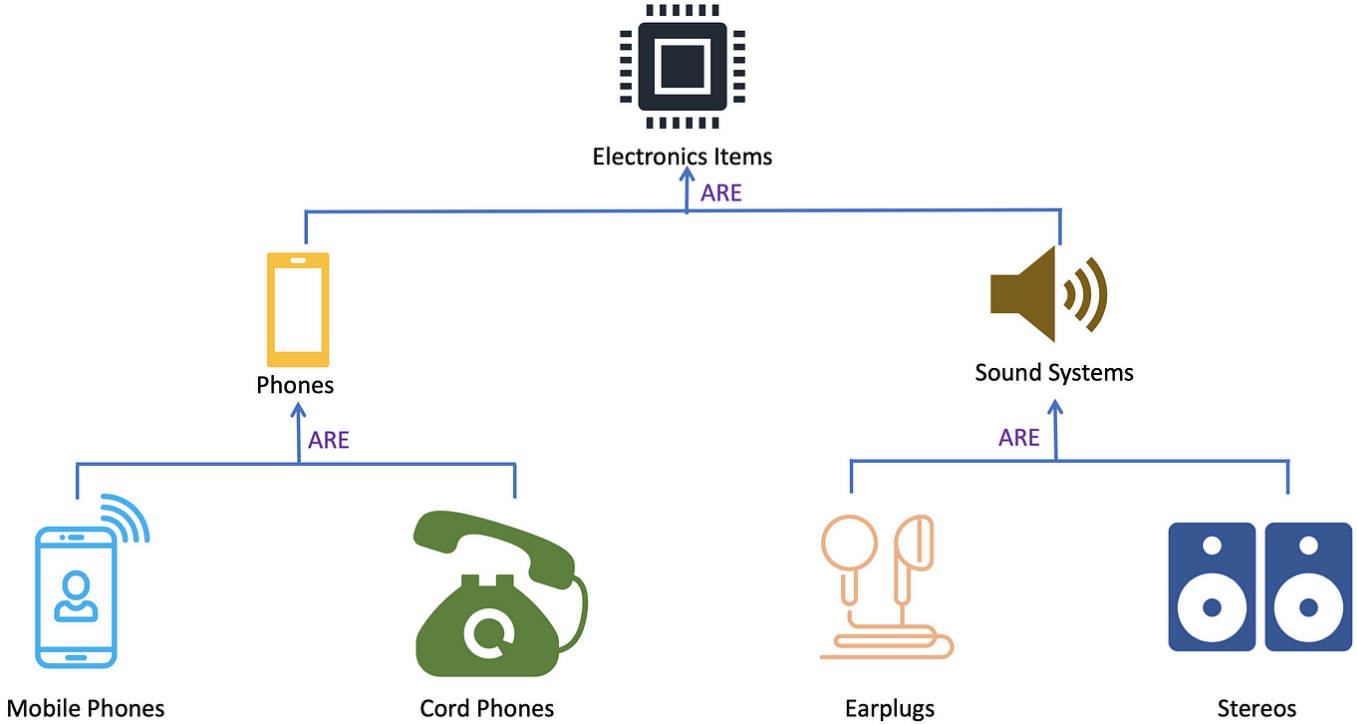
OOP alapelvek

- **Adatrejtés (information hiding)**
  - **az osztály műveleteinek implementációja az osztályon kívül nem látható**

Forrás: https://www.infobrother.com/Tutorial/C++/C++_Information_Hiding

OOP alapelvek

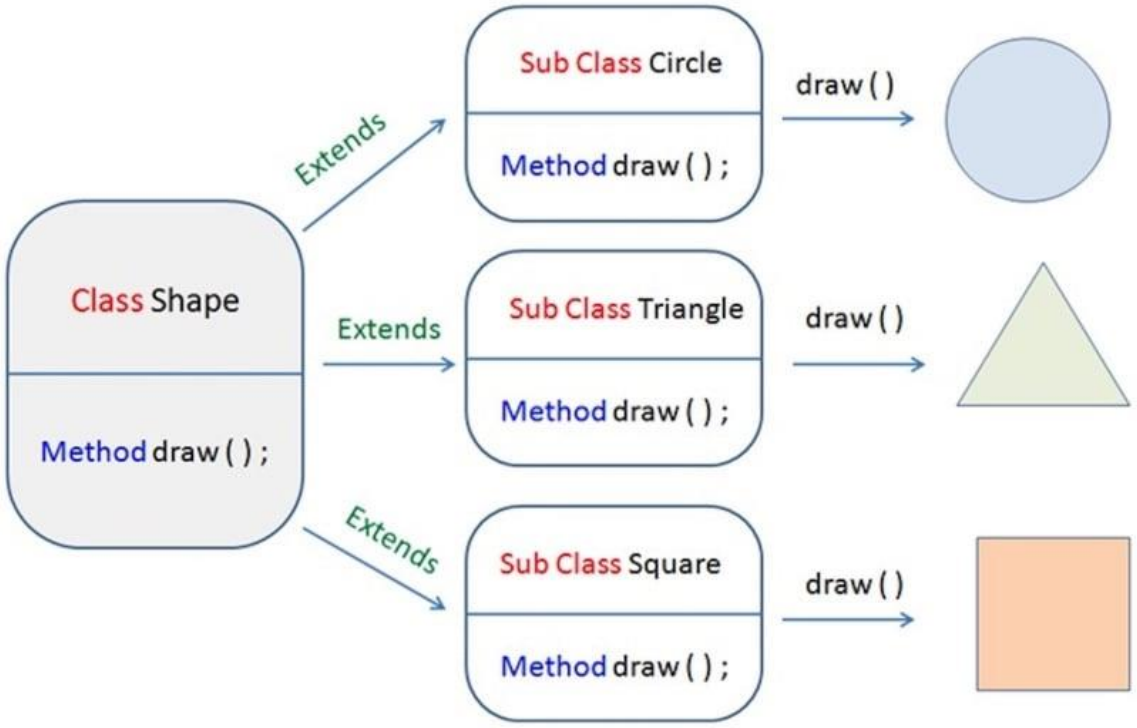
- **Öröklődés (inheritance)**
  - **egy osztály létrehozható úgy, hogy egy másik osztály leszármazottja**, amely így örökli az ősosztály tulajdonságait



Forrás: https://towardsdatascience.com/how-to-code-inheritance-in-java-beginners-tutorial-in-oop-d0fc0a71be98

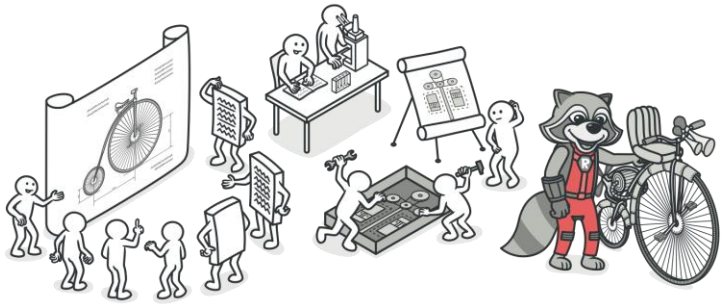
OOP alapelvek

- **Polimorfizmus (polymorphism)**
  - többalakúság, **bizonyos viselkedések működése függ a környezettől**, metódus működése az adott osztálytól függ



Forrás: https://dev.to/divide_r_conquer/three-characteristics-of-oop-polymorphism-1lna

Tervezési minták



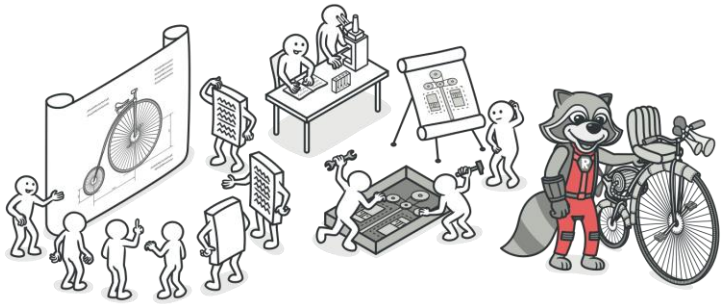
„Minden minta olyan problémát ír le, ami újra és újra felbukkan a környezetünkben, s aztán leírja hozzá a megoldás magját, oly módon, hogy a megoldás milliószor felhasználható legyen, anélkül, hogy valaha is kétszer ugyanazt csinálnák”

Christopher Alexander

- Christopher Alexander **javaslata épületek és városok építésénél, hogy minták alapján kellene megvalósítani**
  - de ugyanez **igaz objektumközpontú szoftverekre is**
  - falak és ajtók helyett objektumok és felületek
- Bonyolult, összetett rendszerek esetében igen fontos
- **Felépítésre definiált egy sablont**
  - sok ilyen létezik (Abstract factory, Adapter, Composite, Decorator, Factory method etc.)

Forrás: https://refactoring.guru/design-patterns

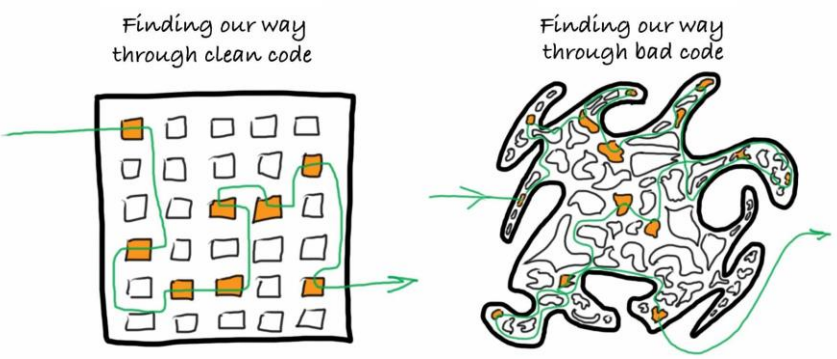
Tervezési minták



- **Egy tervezési minta elemei**
  - **Név: lehessen rá hivatkozni**, utal a problémára, megoldása és a következményre
  - **Probléma: mikor alkalmazzuk** az adott mintát, leírja a problémát és összefüggéseit (feltételek összesége mikor alkalmazható az adott minta)
  - **Megoldás:** azon elemek leírása hatásköreikkel, viszonyaikkal, együttműködési lehetőségeikkel, **melyek felépítik a tervet**. Egy sablont ad meg, a tervezési problémára elvont leírást ad
  - **Következmény:** az adott **minta alkalmazásának előnyei, hátrányai** (tárhely- időfelhasználás, újrahasznosíthatóság)

Kialakulásuk okai

- **Rossz kód**
  - láttunk már ilyet? ☺
  - több idő megérteni mint újraírni…
  - nehéz megmondai mit csinál
  - felesleges iterációk
  - változó- függvénynevek értelmetlenek
  - kicsit módosítom → rögtön elromlik minden



- **Rövid határidők**
  - → kókányoljunk
    - „jó lesz az úgy”
    - „működik, nem?”
    - „majd később átírom”
    - „átmenetileg (örökre…) jó az úgy”

Forrás: https://lvivity.com/how-to-write-good-code

Kialakulásuk okai

- **Kód rombolás (code rot)**
  - **a forráskód fokozatosan elavul**, karbantarthatatlanná válik, és egyre nehezebb vele dolgozni
    - elavulnak a használt technológiák
    - nő a komplexitás, csökken az átláthatóság
    - töredezetté válik a kód (spaghetti code, dead code)
    - nincsenek tesztek, vagy nem frissítik őket
    - új funkciókat adnak hozzá átgondolás nélkül
    - „most jól van ez így, később átírom”
    - egy kicsi hekk itt-egy kicsi hekk ott, sok if sok for egymásbaágyazva
      - a gyors megoldások miatt később újra kell írni az egész kódot, mert túl sok kompromisszum született
    - sok idő alatt, apránként történik
  - **ezek következtében egy idő után kód elkezd „rothadni”**
    - → eredménye: átláthatatlan, olvashatatlan kód keletkezik

Kialakulásuk okai

- **Kód rombolás (code rot)**

  - függőségek szintjén új dolgok jelennek meg
    - nem tervezett
    - hekk következménye
  - régi könyvtárak, technológiák használata, amelyek már nem kompatibilisek az új verziókkal
  - a dokumentáció elavul, mert a kód változik, de a leírások nem frissülnek
  - megelőzés
    - refaktorálás
    - automatizált tesztelés
    - függőségek frissítése
    - tiszta kód írása
    - megfelelő dokumentáció

Kialakulásuk okai

- **Kód rombolás (code rot) példa**

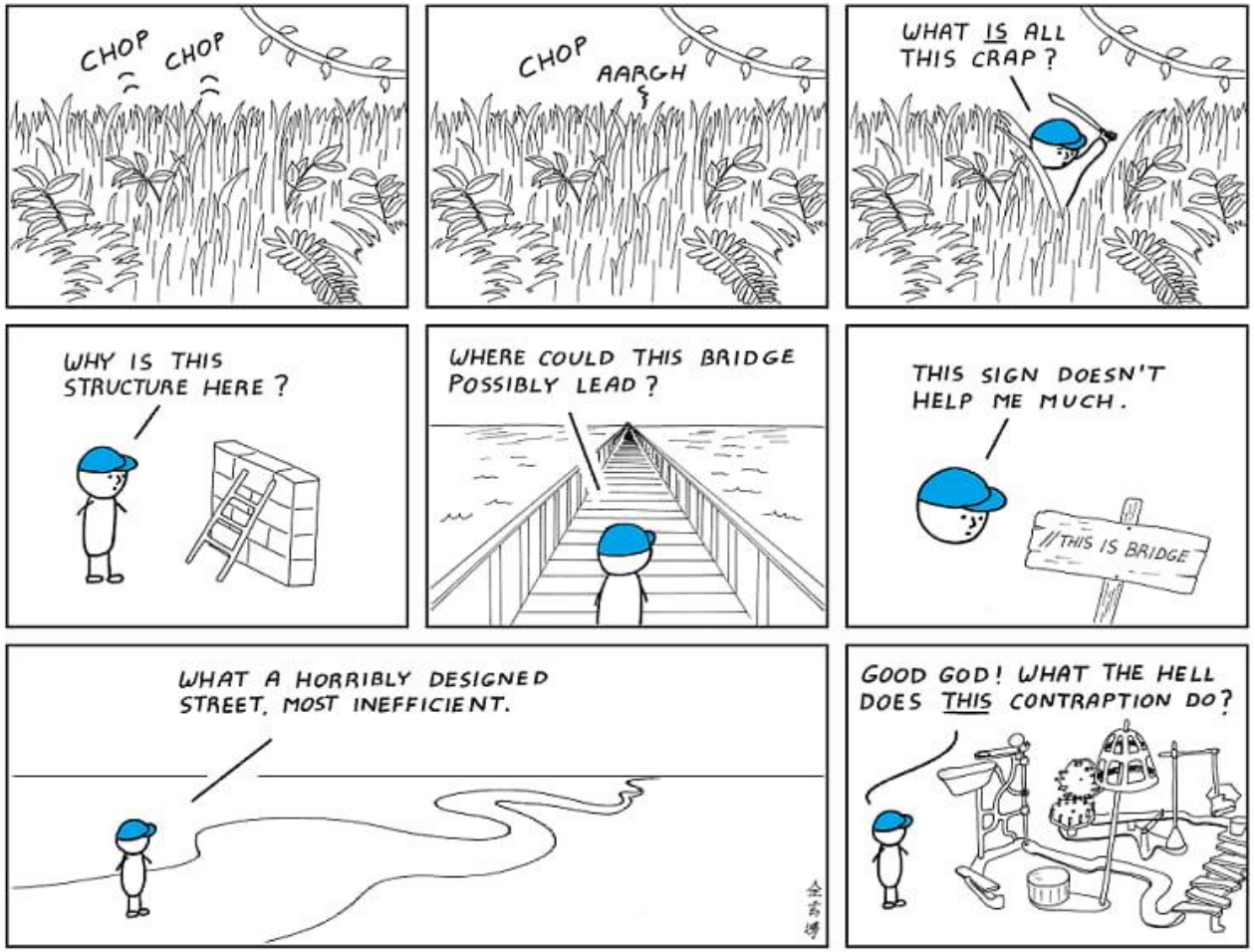
```
def calculate_price(price, tax):
    return price + (price * tax)
```

```
def calc_price(price, tax, discount=0, currency="$"):
    final_price = price + (price * tax)
    if discount > 0:
        final_price -= discount

    print(f"{currency}{final_price}")
    return final_price
```

```
def calculate_price(price, tax, discount=0):
    return price + (price * tax) - discount

def format_price(amount, currency="$"):
    return f"{currency}{amount:.2f}"
```

Kialakulásuk okai

- **Miért van szükség a jó kódra?**

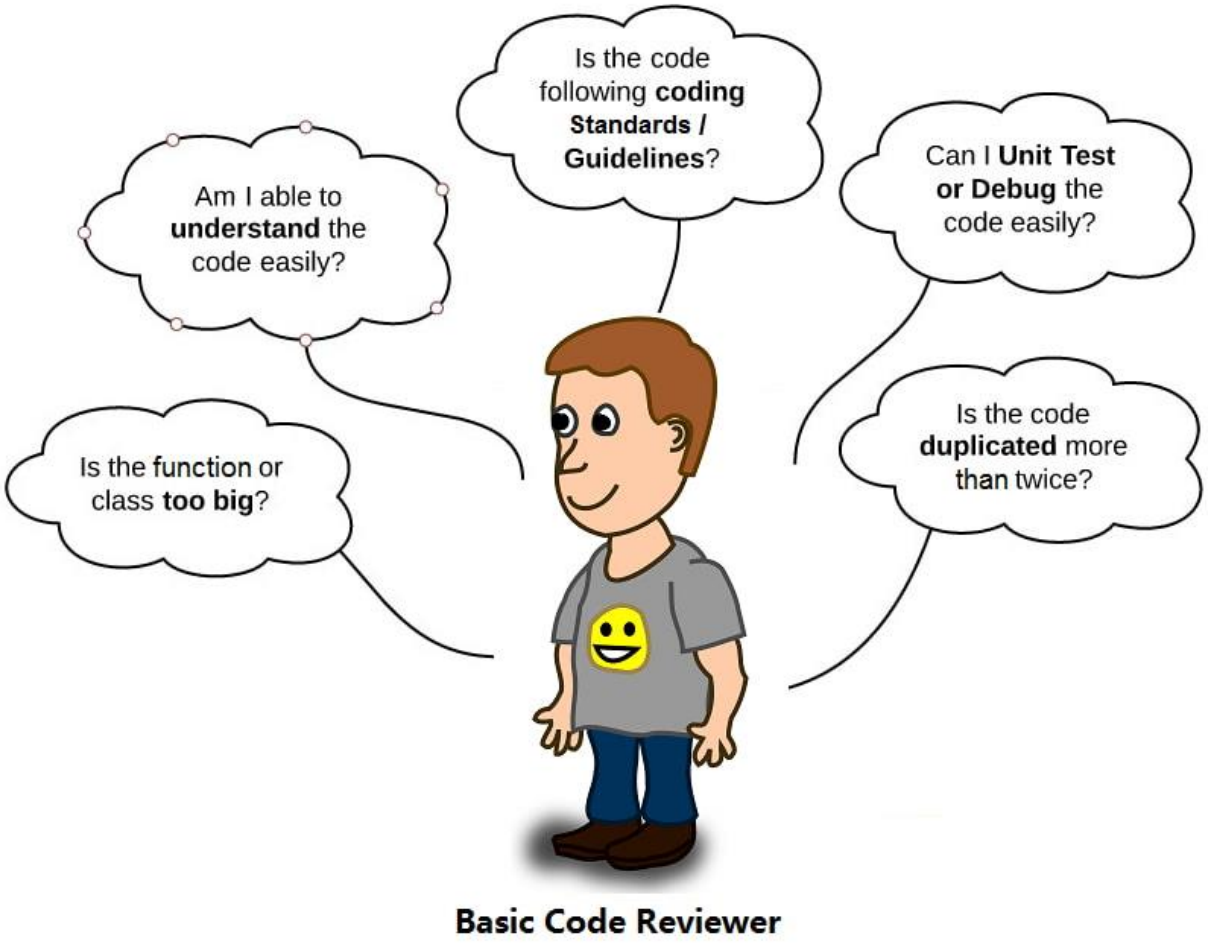


I hate reading other people's code.

Forrás: https://lvivity.com/how-to-write-good-code

Kialakulásuk okai

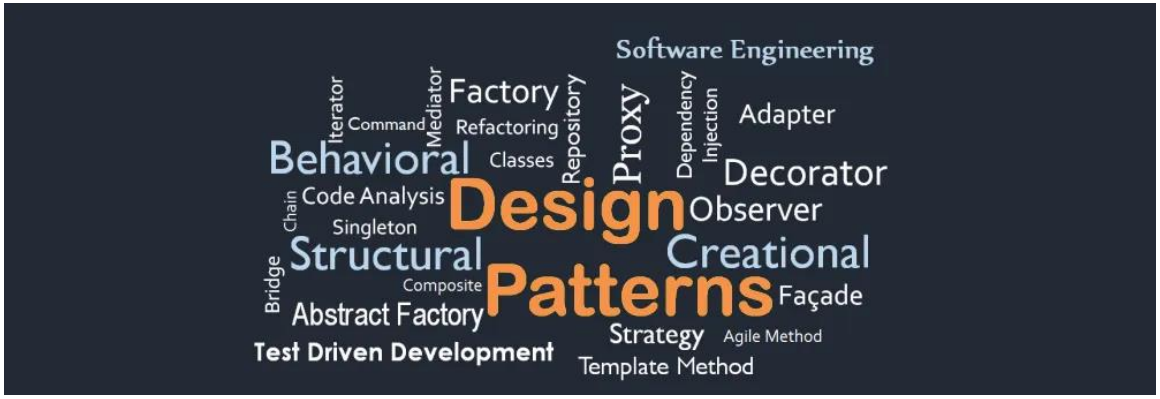
- **Jó a kód?**



Forrás: https://lvivity.com/how-to-write-good-code

Tervezéi minták, szabályok

- **Nehány tervezési minta, szabály**
  - **KISS (Keep it simple, stupid)**
  - **Demeter törvénye**
  - **Separation of Concerns (SoC)**
  - **DRY (Don’t Repeat Yourself)**
  - **SOLID elvek**

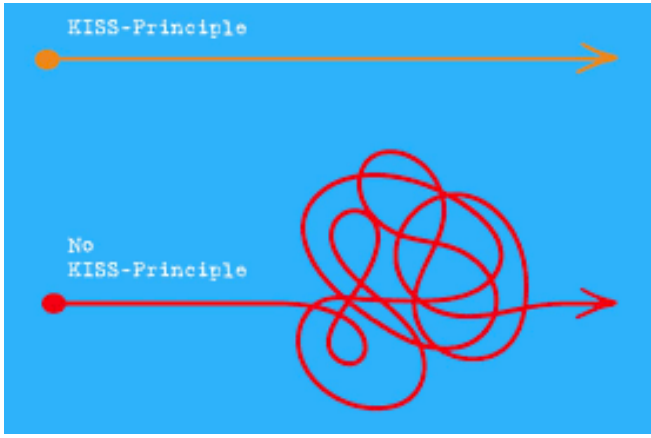


Forrás: https://www.nilebits.com/blog/2022/04/software-design-patterns/

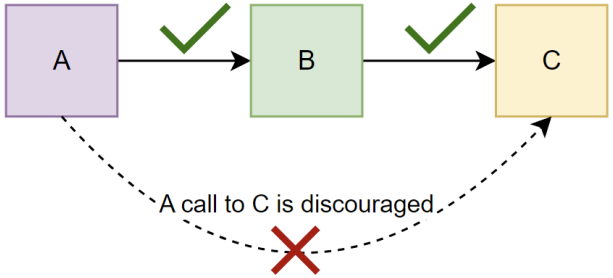
KISS

K.I.S.S
KEEP.IT.SIMPLE, STUPID!

- **Keep it simple, stupid**
  - 1960-as évek, haditengerészet
  - Kelly Johnson (1910–1990) repülőmérnöktől származik
  - általában a legegyszerűbb megoldások a legjobbak
  - mindent olyan egyszerűen kell csinálni, amennyire csak lehetséges
  - felesleges bonyolultság kerülése
  - a változónevek pontosan leírják, hogy milyen értéket tárolnak
  - a metódusnevek tükrözik a metódus célját
  - komment csak ott ahol szükséges
  - globális változók kerülése
  - stb…



Forrás: https://medium.com/illumination/the-secret-to-success-keeping-it-simple-with-the-kiss-principle-5442b0b99b3f
https://dev.to/kwereutosu/the-k-i-s-s-principle-in-programming-1jfg

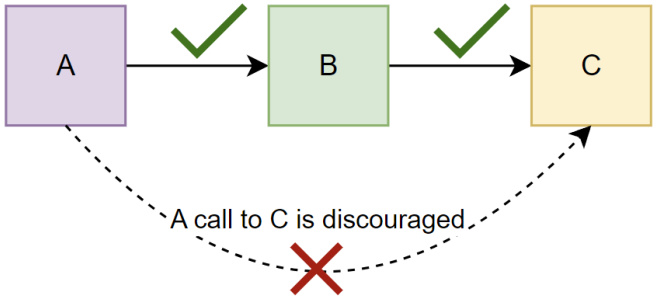


Demeter törvénye

- **Ne beszélgess idegenekkel**
  - **az objektumok kerüljék a hozzáférést más objektumok belső adataihoz és metódusaihoz, egy objektum csak a közvetlen függőségeivel léphet kapcsolatba**
  - korlátozza az objektumok üzenetküldését
  - szűkíti a meghívható metódusok körét
  - minimálisra csökkenti az osztályok közötti kommunikáció
  - a kód kevésbé lesz függő a belső implementáció részleteitől
  - 5 szabály
  - Pl.: `Gamer` , `Weapon` osztályok
    - `Gamer`: `Weapon` adattag, `attack()` metódus, `Weapon`: `use()` metódus
    - Gamer osztály az attack() metódusában közvetlenül hívja a weapon.use() metódust
    - Gamer osztály csak a közvetlenül kapcsolódó Weapon objektummal kommunikál

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter törvénye



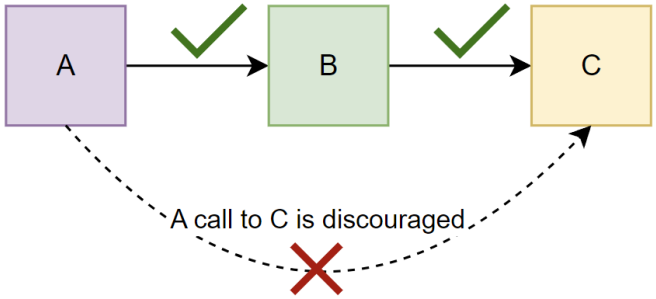
- **1. szabály**
  - **egy C osztály X metódusa csak a C metódusait hívhatja meg**

```
class Greetings {
    String generalGreeting() {
        return "Welcome" + world();
    }

String world() {
    return "Hello World";
    }
}
```

  - `generalGreeting()` metódus meghívja a `world()` metódust ugyanabban az osztályban, ez helyes mert egy osztályba tartoznak

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter törvénye



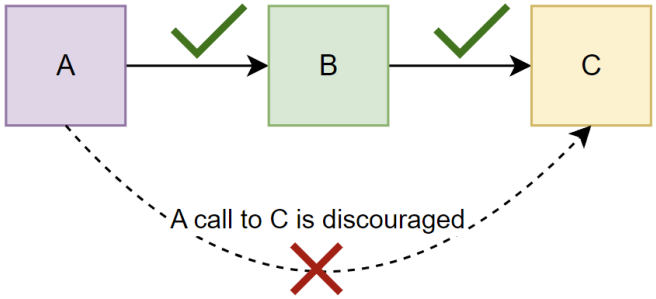
- **2. szabály**
  - **egy C osztály X metódusa csak az X által létrehozott objektum metódusait hívja meg**

```
String getHelloBrazil() {
    HelloCountries helloCountries = new HelloCountries();

    return helloCountries.helloBrazil();
}
```

  - a `getHelloBrazil()` metódus hozta létre az objektumot, így hívhatja annak metódusait

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter törvénye



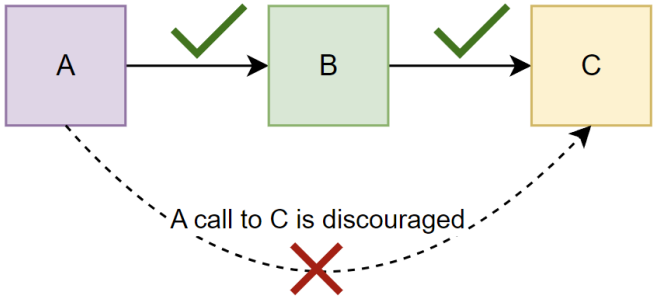
- **3. szabály**
  - **X metódus csak az X-nek argumentumként átadott objektumot hívhat meg**

```
String getHelloIndia(HelloCountries helloCountries) {
    return helloCountries.helloIndia();
}
```

  - a `getHelloIndia()` argumentuma a `HelloCountries` referencia
  - a szabályának megsértése nélkül hívhatja a `helloCountries` metódusait

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter törvénye



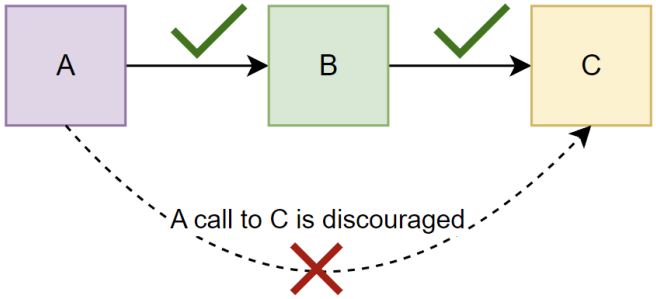
- **4. szabály**
  - **egy C osztály X metódusa csak a C példányváltozójában lévő objektum metódusát hívja meg**

```
HelloCountries helloCountries = new HelloCountries();

String getHelloJapan() {
    return helloCountries.helloJapan();
}
```

  - `helloCountries` példányváltozó az osztályban. `helloJapan()` metódus hívása a `getHelloJapan()` metóduson belüli példányváltozón

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter törvénye



- **5. szabály**
  - **egy C osztály X metódusa csak a C-ben létrehozott statikus mező metódusát hívhatja**

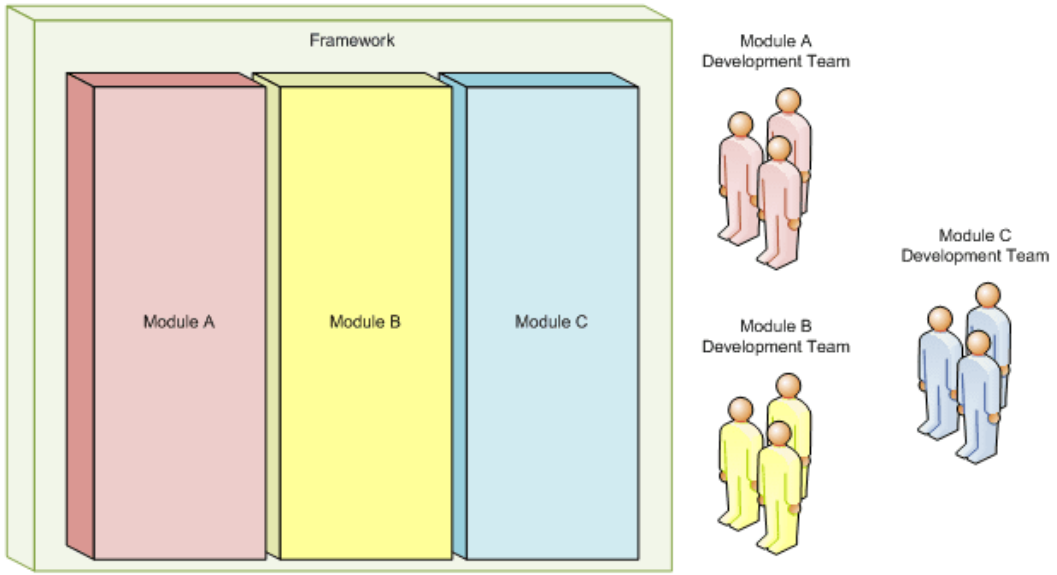
```
static HelloCountries helloCountriesStatic = new HelloCountries();

String getHellStaticWorld() {
    return helloCountriesStatic.helloStaticWorld();
}
```

- `getHellStaticWorld()` meghívja a `helloStaticWorld()` metódust az osztályban létrehozott statikus objektumon

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Separation of Concerns (SoC)

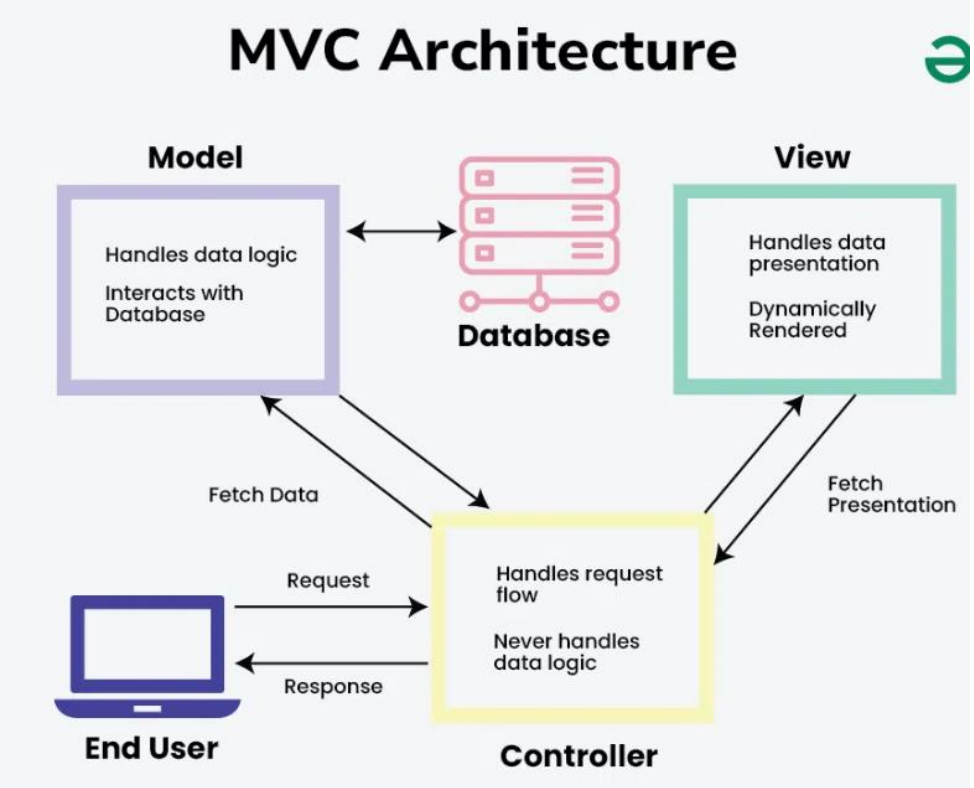
- **Vonatkozások szétválasztása**
  - **minden egyes szoftver komponensnek jól meghatározott szerep és ezek nem fedik egymást**
  - különböző funkciók és felelősségi területek elkülönítésével a rendszer komplexitása csökken
  - a **részeket úgy kell tervezni, hogy azok csak egy konkrét dolgot tegyenek**, és ne vegyék figyelembe a többi rész részleteit

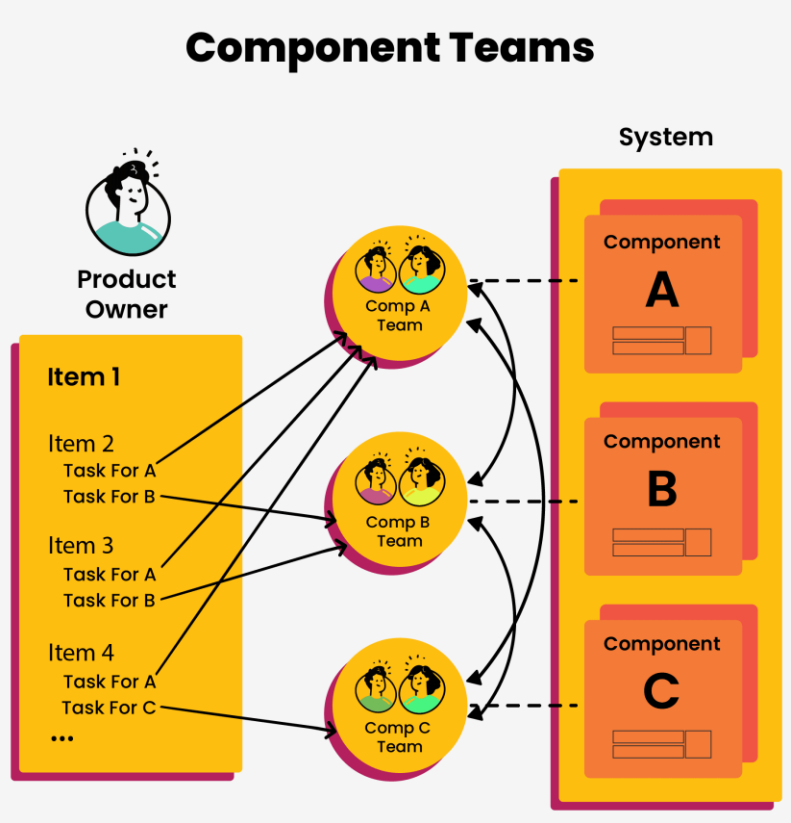


Forrás: https://dev.to/suspir0n/soc-separation-of-concerns-5ak7

Separation of Concerns (SoC)

- **Vonatkozások szétválasztása**
  - Pl.: MVC, komponens alapú fejlesztés





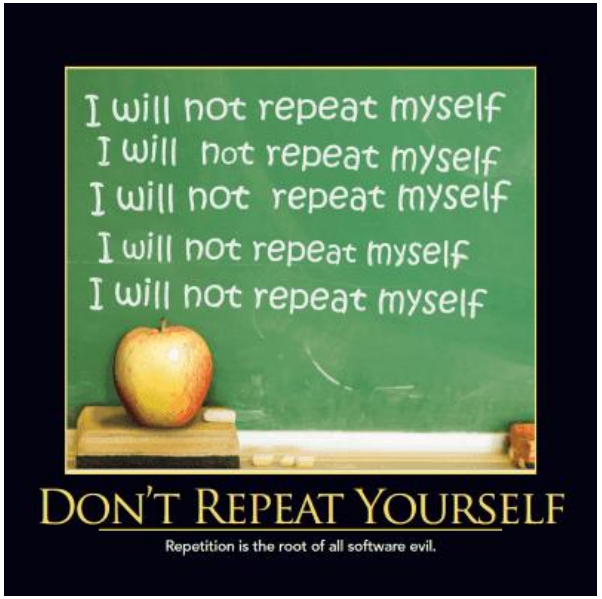
Forrás: https://www.geeksforgeeks.org/mvc-architecture-system-design/
https://marutitech.com/guide-to-component-based-architecture/

DRY (Don’t Repeat Yourself)

- **Kódismétlés kerülése**
  - **semmi se ismétlődjön a kódban**
    - → ha valahol módosítani kell akkor minden előfordulási helyen módosítani kell
  - minden üzleti logika vagy funkció csak egyszer szerepeljen
  - függetlenül attól, hogy az ismétlődés a „Copy Paste” programozásból vagy az absztrakció alkalmazásának gyenge megértéséből ered, rontja a kód minőségét

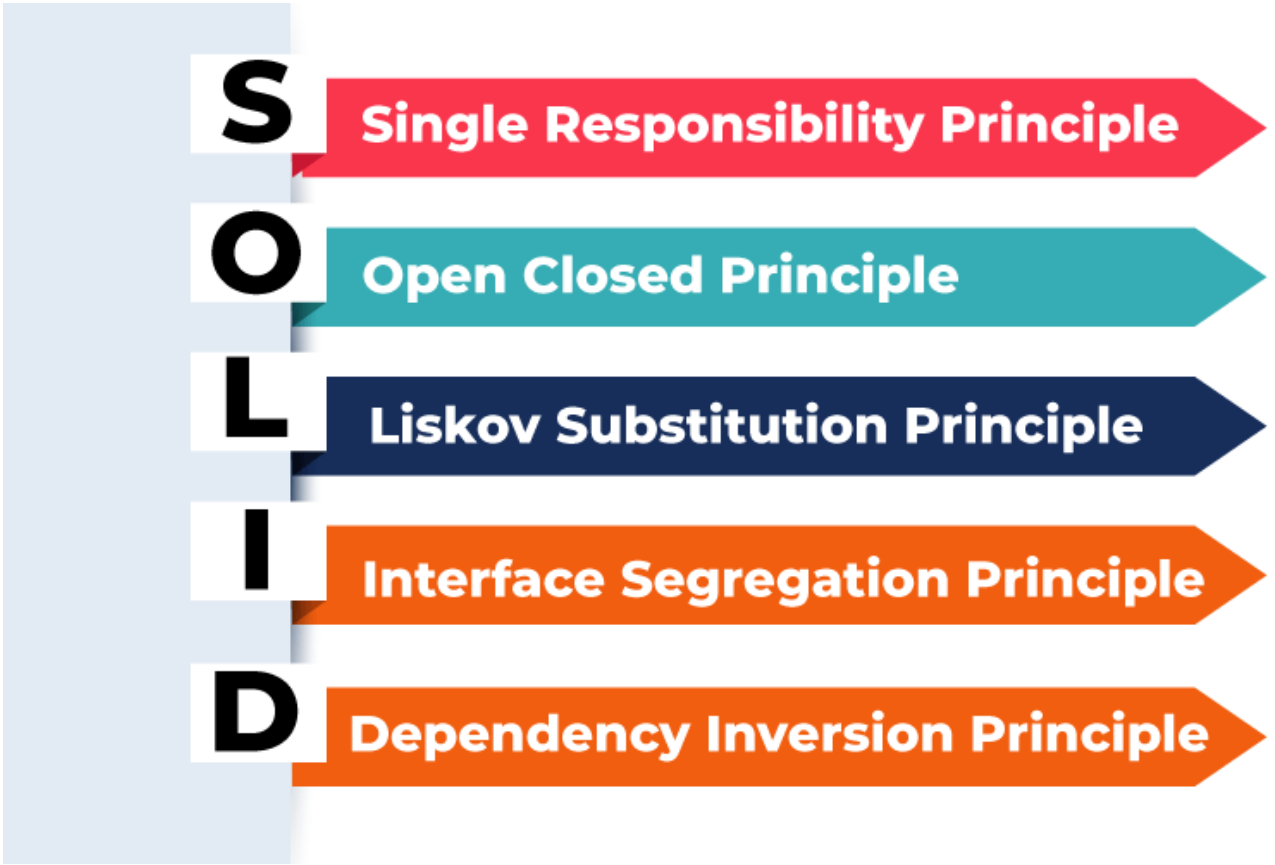
```
def calculate_discounted_price(price):
    return price * 0.9

def calculate_vip_discounted_price(price):
    return price * 0.8
```

```
def apply_discount(price, discount_rate):
    return price * (1 - discount_rate)

print(apply_discount(100, 0.1))
print(apply_discount(100, 0.2))
```



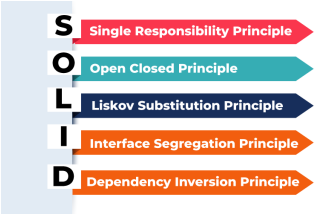
Forrás: https://deviq.com/principles/dont-repeat-yourself

SOLID elvek



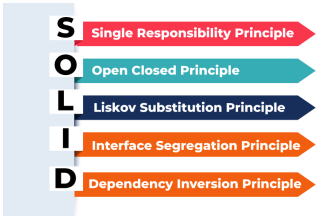
Forrás: https://www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/

Single Responsibility Principle



- **Egy osztály/objektum csak egy dologért feleljen**
- **Egy oka legyen a változásra**
- Különböző feladatoknak külön ösztályok
- Előnyei
  - tesztelés: kevesebb teszt-eset osztályonként
  - összefonódás: kevesebb funkció, kevesebb függőséget jelent
  - szervezés: kisebb, jól rendezett osztályok jobb kereshetőséget biztosítanak

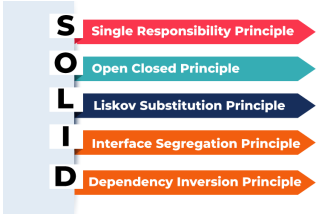
Single Responsibility Principle



- Példa
  - van egy `Book` osztály, amiben egy könyv néhány attribútumát modellezük
  - nem írunk bele olyan metódust, ami kiíratja a konzolra azokat
    - Miért? Nem az a dolga annak az osztálynak
    - Megoldás: Egy `BookPrinter` vagy általánosabban egy `ObjectPrinter` osztály, mely tartalmazhatja a kiíratót metódusokat

- Másik példa
  - `Student` osztály: `addGrade`, `setName` metódusok
    - jegybeírást oktató által, név módosítás titkár által
    - a 2 felelősség összemosódik
    - Megoldás: `Student`-ben adattároló, különböző felelősségeknek más osztályok

Single Responsibility Principle

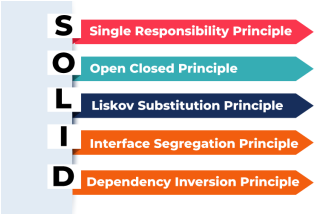



```java
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String replacementWord){
        return text.replaceAll(word, replacementWord);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

```java
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

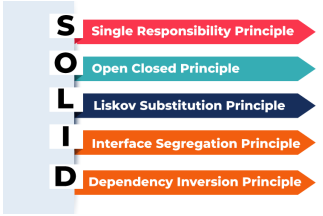
Forrás: https://www.baeldung.com/solid-principles

Open/Closed Principle

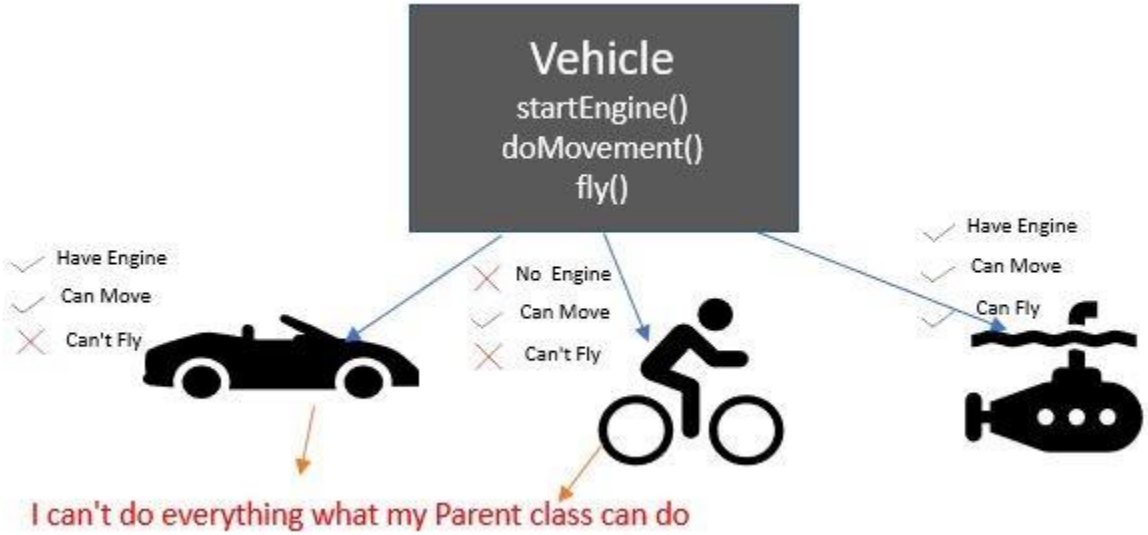


- **Egy osztály vagy modul legyen nyitott a bővítésre, de zárt a módosításra**
- Amit már véglegesítettünk (metódust, objektumot, stb.), azt már nem módosítjuk, legfeljebb kibővítjük
- Előnye: bugok elkerülése, megelőzése
- Példa
  - van egy metódus, ami kiírja egy lista elemeit soronként
  - de mostantól azt szeretnénk, hogy ezt egyetlen sorba tegye
  - nem az előbbi metódust módosítjuk, hanem létrehozunk a feladatra egy új metódust

Liskov Substitution Principle

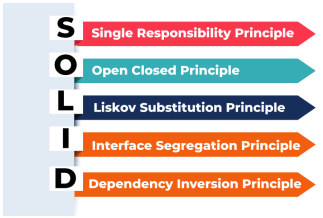


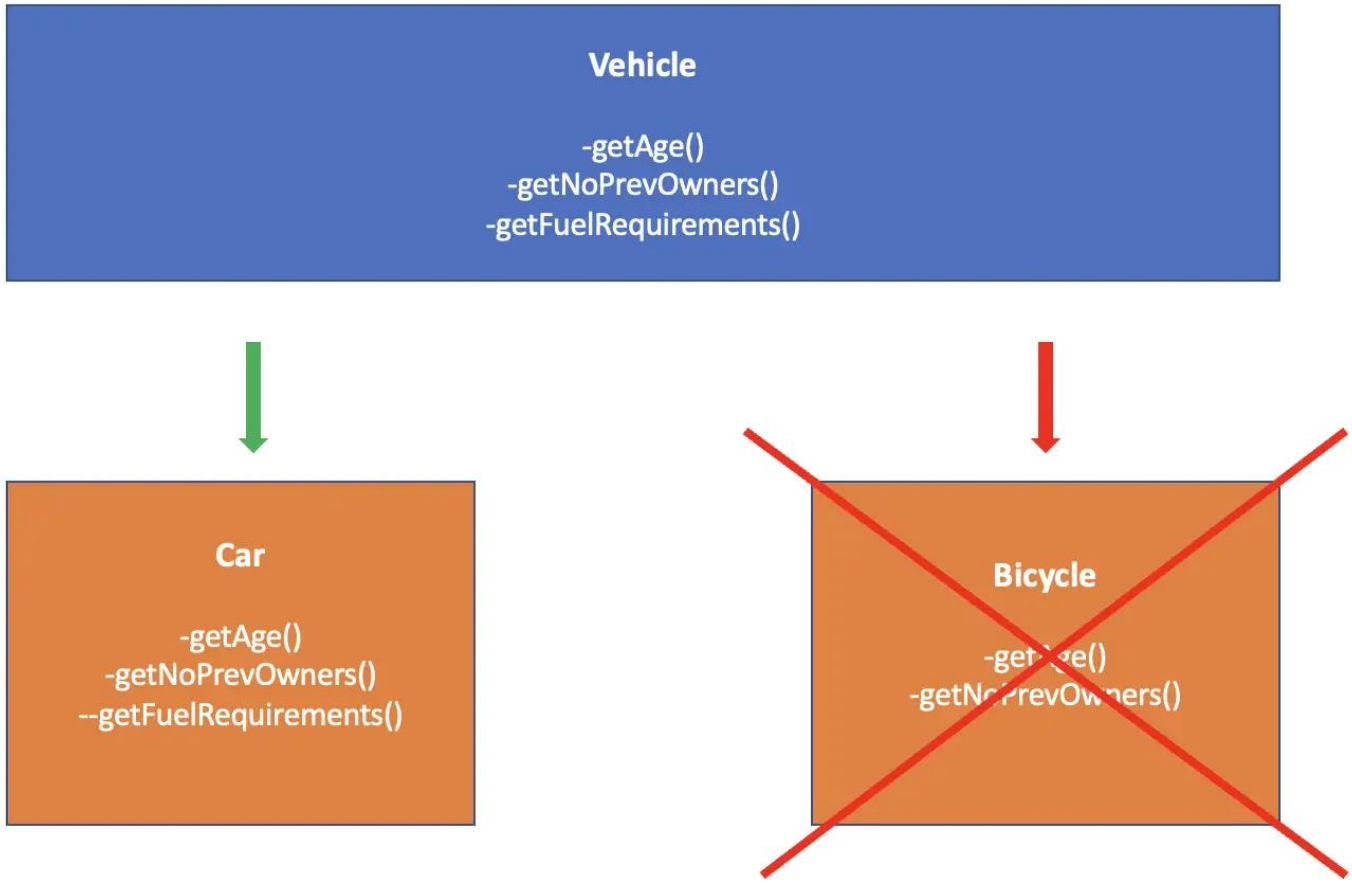
- **Minden szülő osztály helyettesíthető a leszármazottal de anélkül, hogy ez gondot okozna**
- Ha az A osztály a B osztály altípusa, akkor képesnek kell lennünk arra, hogy B-t A-val helyettesítsük probléma nélkül
- Minden leszármazott tudja ugyanazt amit az ős



Forrás: https://www.linkedin.com/pulse/liskov-substitution-principle-mamata-raote-she-her-

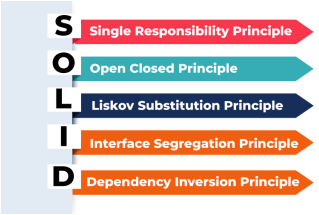
Liskov Substitution Principle



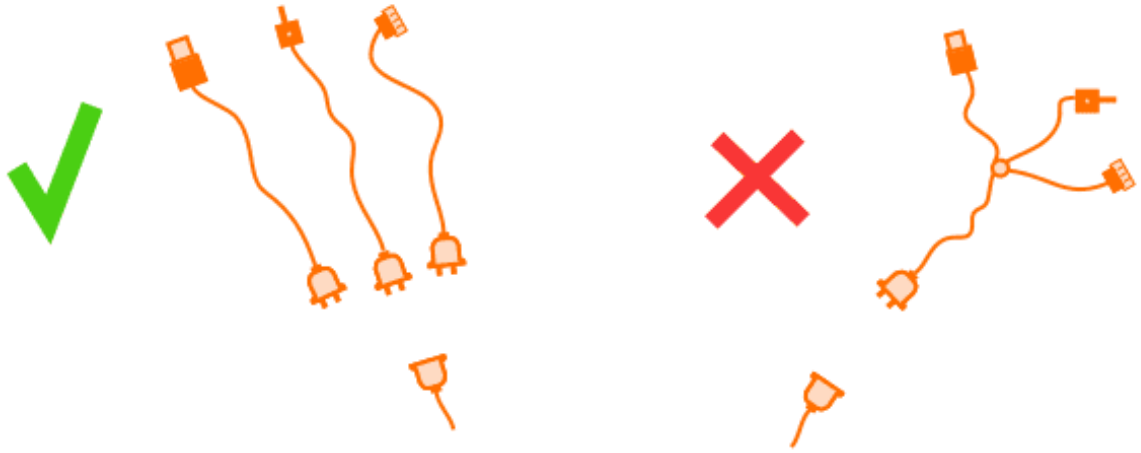


Forrás: https://www.baeldung.com/solid-principles

Interface Segregation Principle

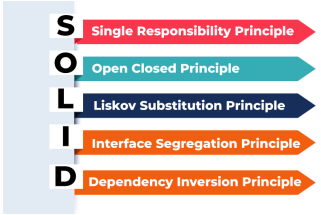


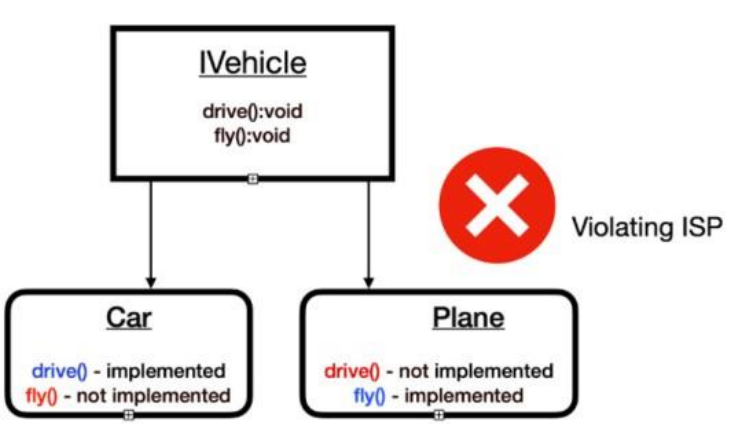
- **Több de jólspecifikált interfész az jobb, mint egy általános**
- Ne csináljunk túl nagy interface-eket, ha sok mindent akarunk leírni bennük, akkor inkább válasszuk szét több kicsire
- Előny: kevesebb függőséget okozunk

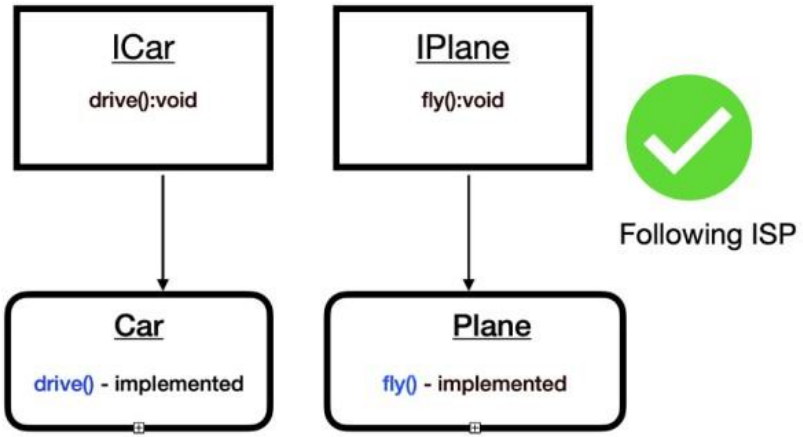


Forrás: https://h-benkachoud.medium.com/interface-segregation-principle-ec256f3828fe

Interface Segregation Principle

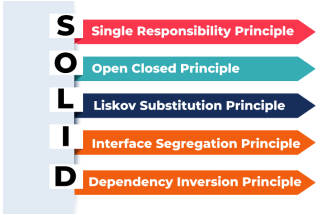




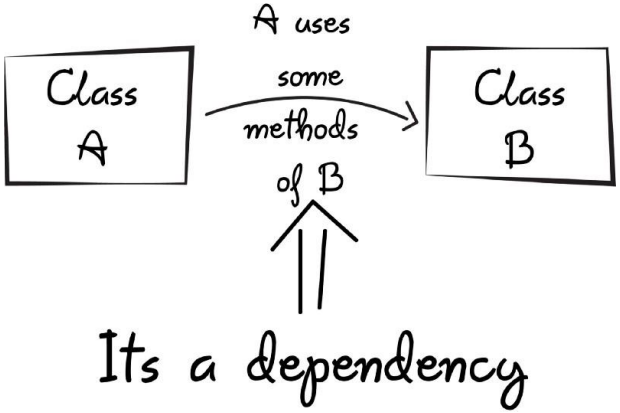


Forrás: https://www.baeldung.com/solid-principles

Dependency Inversion Principle

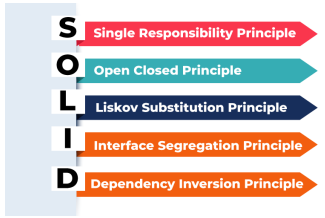


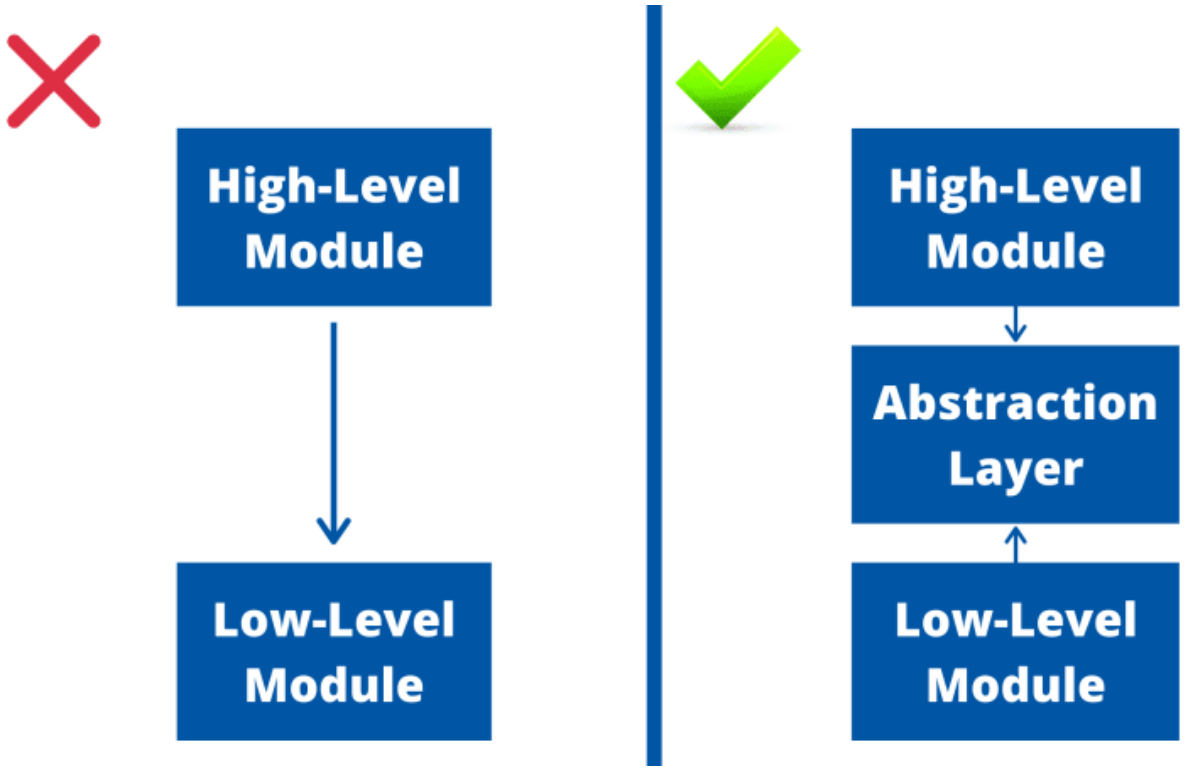
- **Ne konkrét megvalósítástól hanem absztrakciótól való függés legyen**
- Dependency
  - rétegek, modulok közötti kapcsolat, függőség
  - egyik függ a másiktól: egyinek szüksége van másikra
- Loosely coupled
  - lazán kapcsolódó objektumok
- Dependency injection
  - függőség injektálás:
    - másikat az egyikbe injektálni
    - megvalósítás: adattag, konstruktor, set-er
  - a környezet dönti el, hogy mely megvalósítás kerül alkalmazásra (IoC - Inversion of Control)



Forrás: https://www.baeldung.com/solid-principles
https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

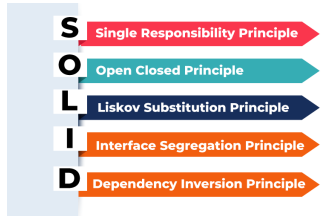
Dependency Inversion Principle

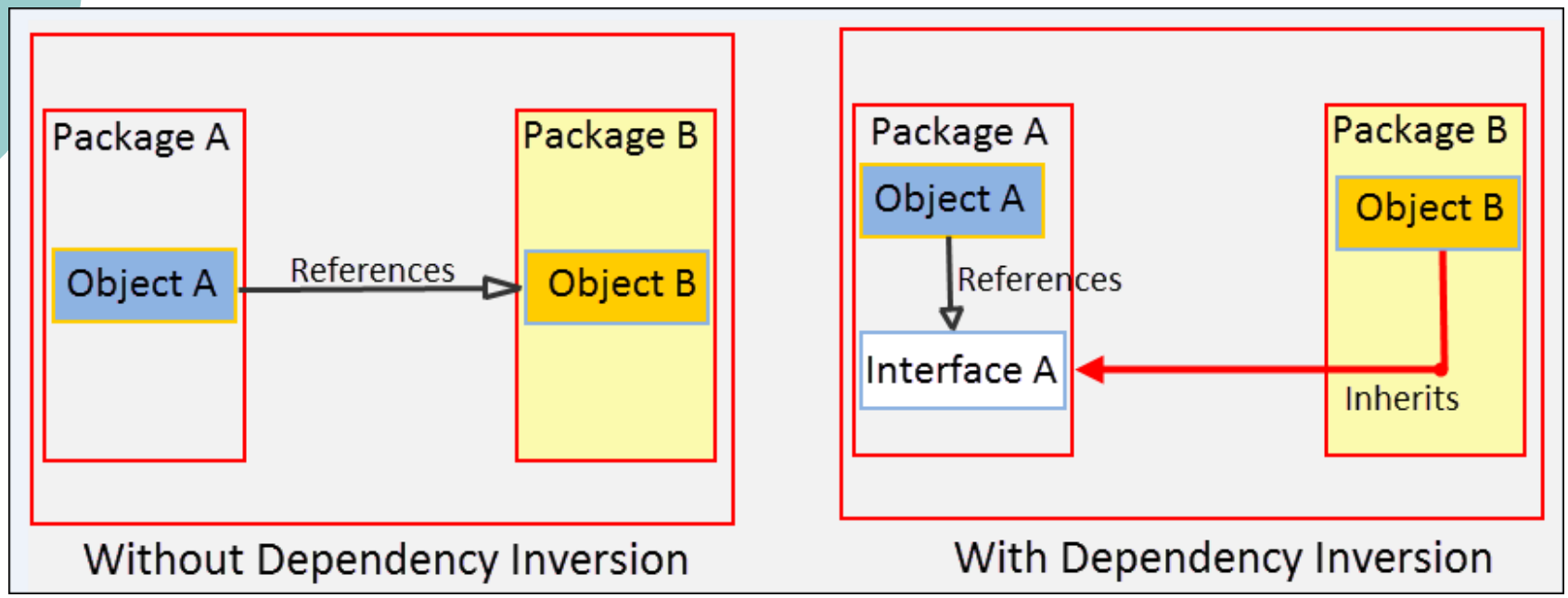




Forrás: https://www.linkedin.com/pulse/dependency-inversion-principledip-mamata-raote-she-her-

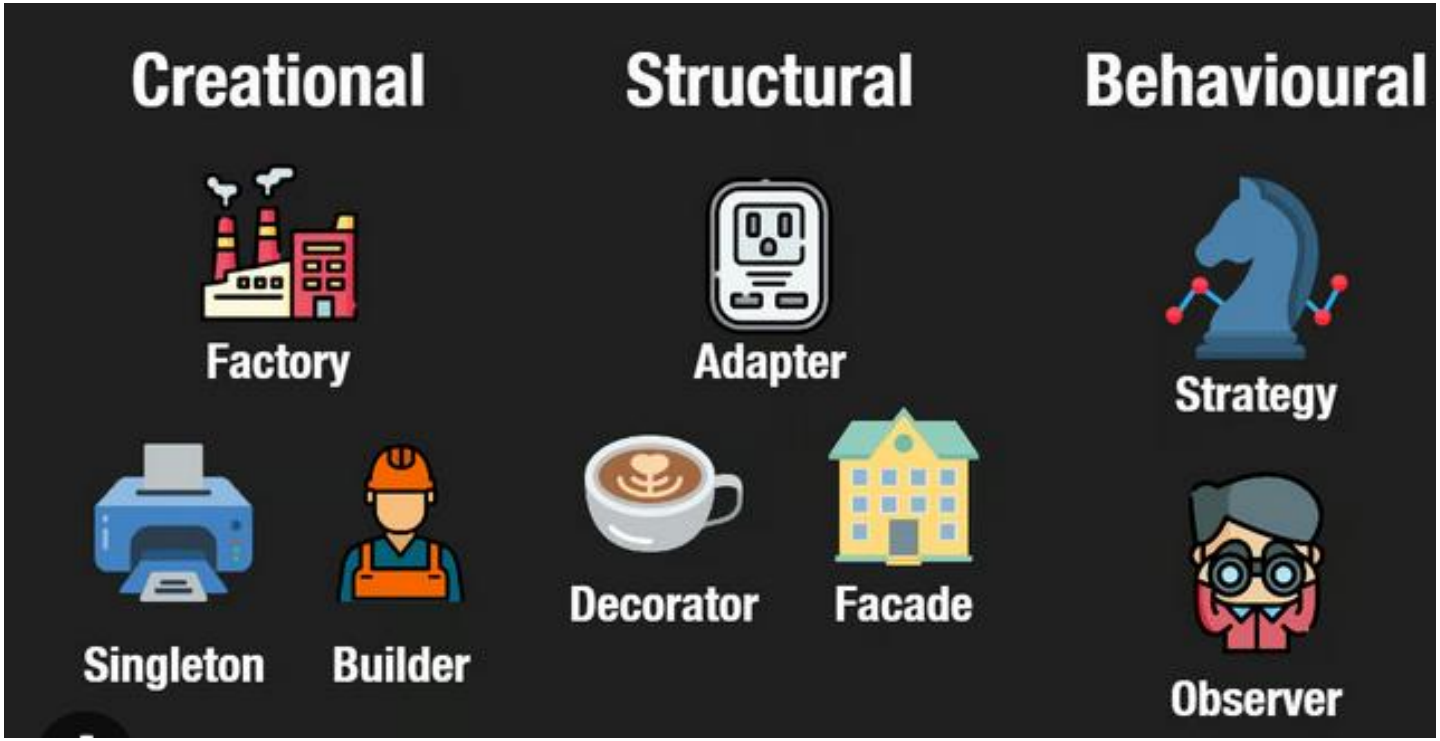
Dependency Inversion Principle





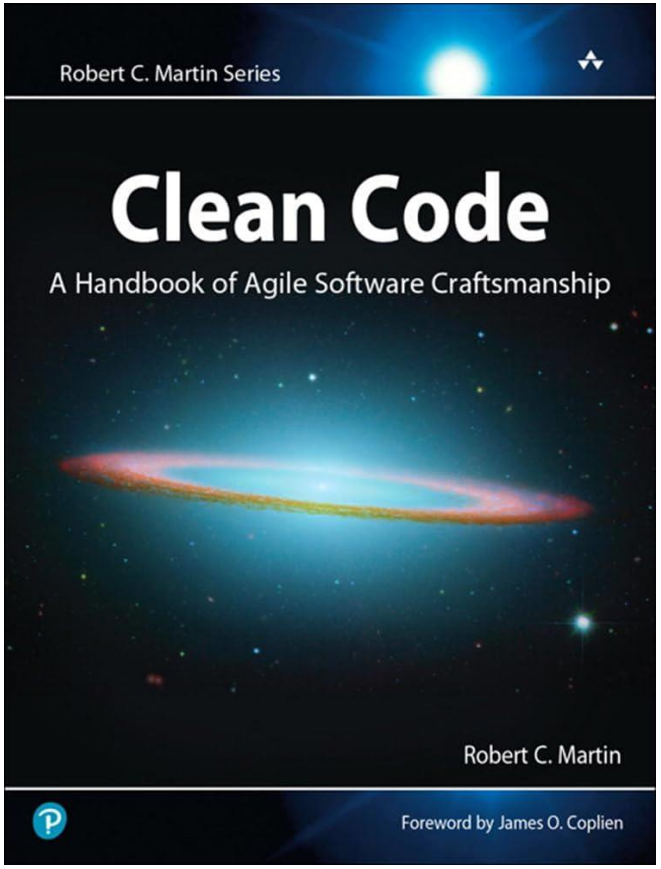
Forrás: https://www.baeldung.com/solid-principles

További programtervezési minták



Forrás: https://neetcode.io/courses/design-patterns/0

Ajánlott könyvek



Köszönöm a figyelmet!