# Szoftvertechnológia és -technikák

## 1. Előadás

*Bevezetés, OOP ismétlés, SOLID elvek*

Automatizálási és
Alkalmazott
Informatikai Tanszék

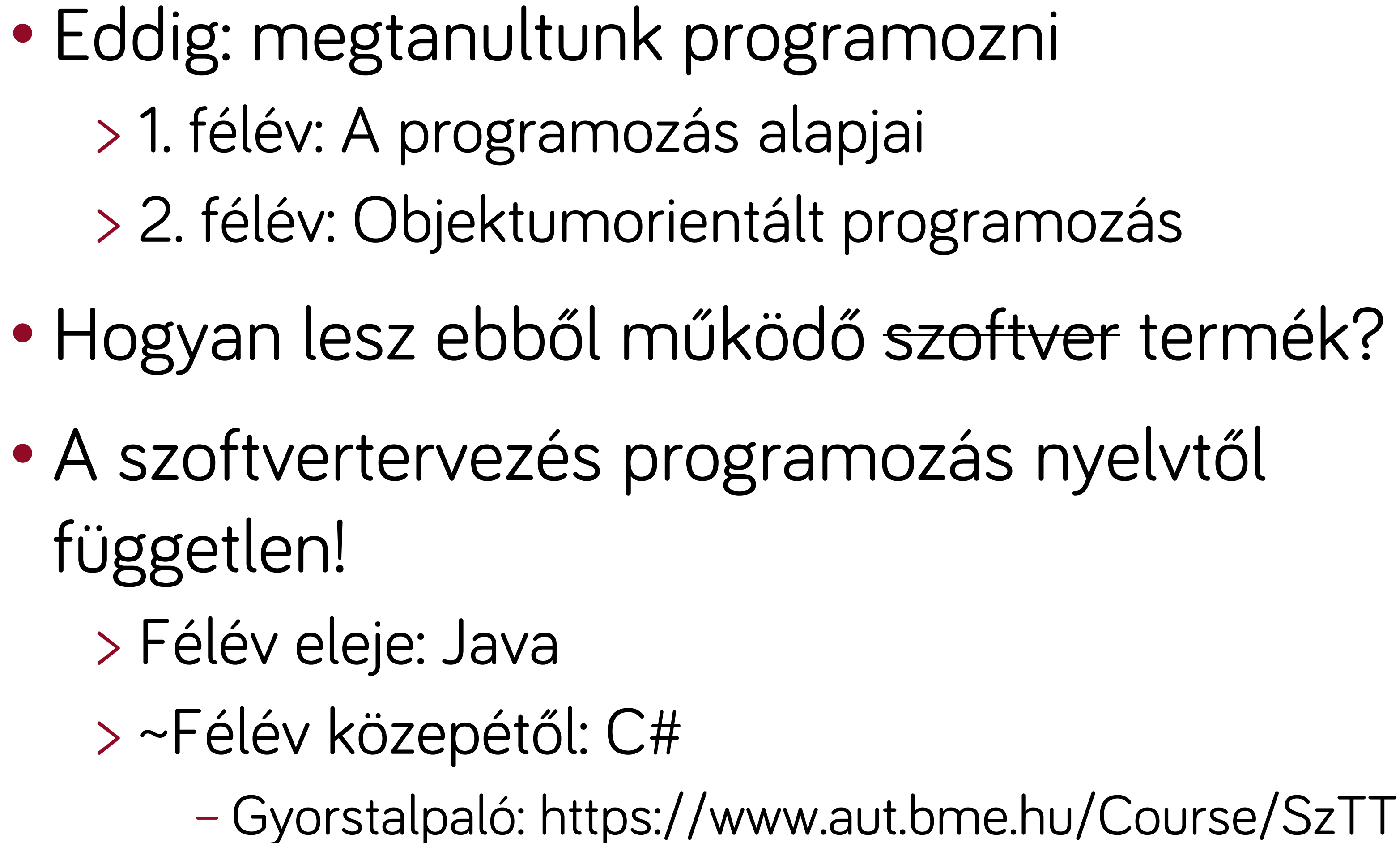
# Szoftvertechnológia és technikák

- Előadás
  - Charaf Hassan,
    Mezei Gergely, Benedek Zoltán, Albert István
  - Kérdezni ér!
  - Email: “[SzTT]”
- Gyakorlat
  - **Erősen** ajánlott
  - 70% részvétel kötelező (4 hiányzás megengedett)
  - Laboronként +1 pont szerezhető a vizsgákhoz
  - Aláírással rendelkezők is jöhetnek (emailben egyeztessünk)

# Számonkérések

- Házi feladat
  - Első házi: tervezési (modellezési) feladat
    - kiadás: ~ 6. hét ; beadás: 10.hét
  - Második házi: tervezési mintás, programozós feladat
    - kiadás: ~ 11. hét ; beadás 13-14. hét/póthét
  - Kötelező!
- ZH
  - Gyakorlati jellegű
  - Minta ZH lesz idén is
- Vizsga
  - Beugró
  - Minta vizsga lesz idén is
- Végső jegy: Vizsga 90p + Házi 2*10p + ZH 40p = 150p
  - + Gyakorlatokért max 12p bónusz!
- **Feladatgyűjtemény** – Gyakorlófeladatok, Teamsen keresztül küldjük

# Miről szól a tárgy?

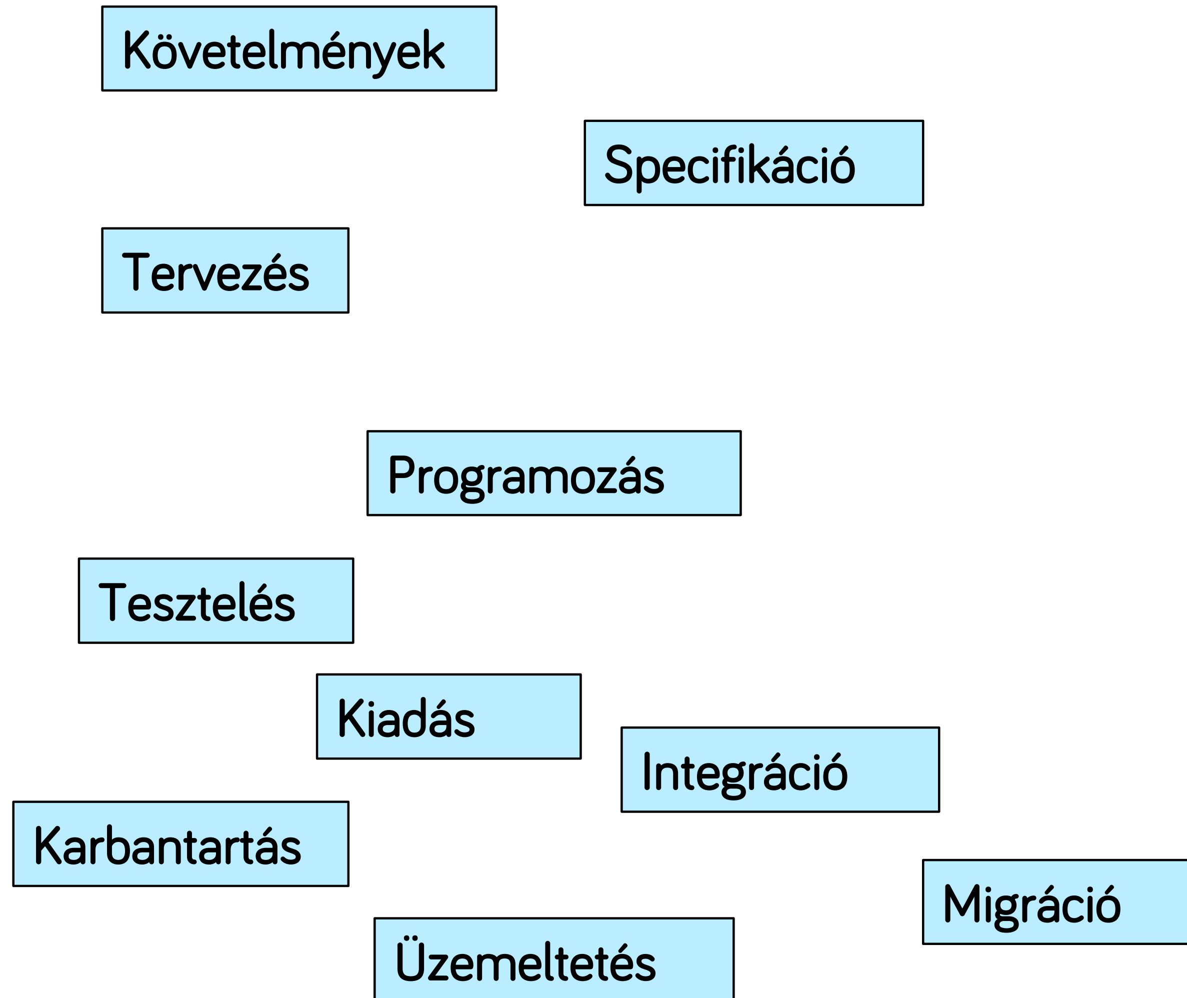


- Eddig: megtanultunk programozni
  - 1. félév: A programozás alapjai
  - 2. félév: Objektumorientált programozás
- Hogyan lesz ebből működő ~~szoftver~~ termék?
- A szoftvertervezés programozás nyelvtől független!
  - Félév eleje: Java
  - ~Félév közepétől: C#
    - Gyorstalpaló: https://www.aut.bme.hu/Course/SzTT

# A szoftverfejlesztés

Programozás

# A szoftverfejlesztés

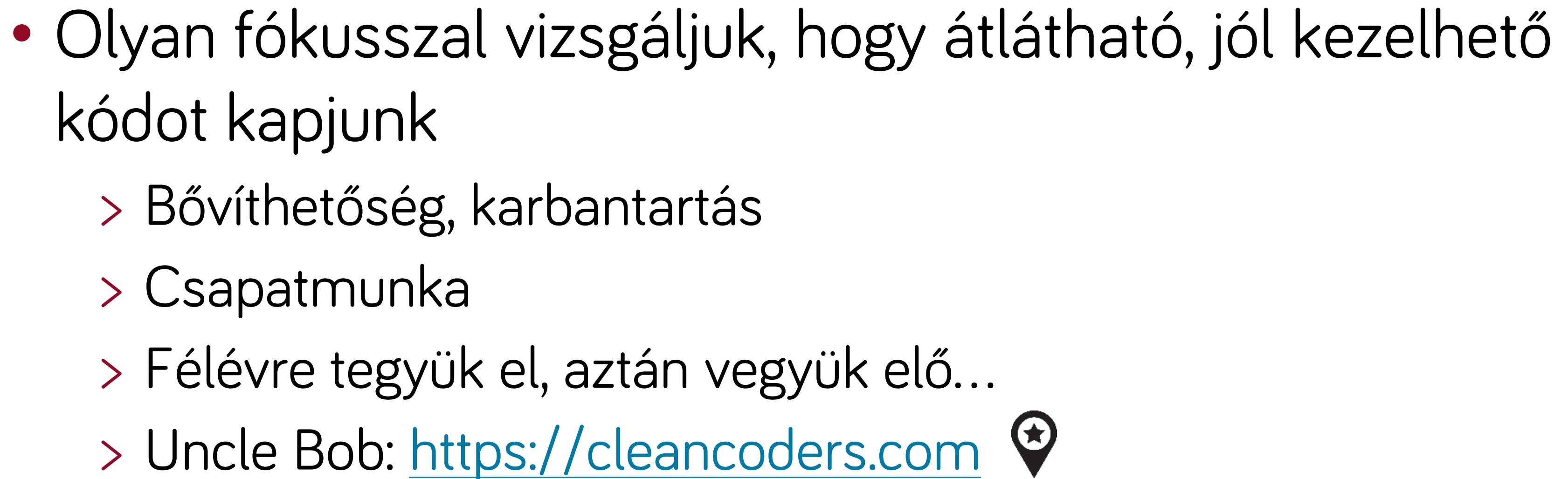


# A tematika

- Bevezés és alapelvek
- Tervezés, UML
- Tervezési minták
- Tervezési módszertanok
- Tesztelés, üzemeltetés

# OOP ismétlés

# Clean code



- Olyan fókusszal vizsgáljuk, hogy átlátható, jól kezelhető kódot kapjunk
  - Bővíthetőség, karbantartás
  - Csapatmunka
  - Félévre tegyük el, aztán vegyük elő…
  - Uncle Bob: https://cleancoders.com

# Refactoring

- Olyan átalakítás, amelyekkel a programkód működése nem változik
- Ha nem tiszta a kód, akkor alkalmazzuk, IDE segít benne, hogy ne rontsuk el
- Bármi, ami a fentinek megfelel, de vannak tipikusak:
  - Átnevezés
  - Kód kiszervezése új metódusba ←→ inline
  - Osztályhierarchián felfelé/lefelé mozgatás
  - ...

# Miért jó OO módon fejleszteni?

1. Közelebb áll az emberi gondolkodásmódhoz
   - A szoftver való életbeli problémát old meg
   - Domén: a szakterület, amelyhez a szoftver kapcsolódik ➔ Domain class, entity class
   - Dolgok, fogalmak kerülnek elő, ezeknek tulajdonságaik vannak, és műveletet végeznek, vagy mi végzünk rajtuk valamilyen műveletet ➔ objektum, tagváltozó, tagfüggvény
   - Főneveket és igék a követelményleírásban!

# Domain class példa #1

```
public class Person {
    private String name;

    private LocalDate birthDate;

    public Person() {}

    public Person(String name, LocalDate birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }

    // …

}
```

# Domain class példa #2

```
public LocalDate getBirthDate() {
    return birthDate;
}

public void setBirthDate(LocalDate birthDate) {
    this.birthDate = birthDate;
}

public long getAge() {
    return Duration.between(birthDate,
            LocalDate.now()).get(ChronoUnit.YEARS);
}
```

# Miért jó OO módon fejleszteni?

2. Jobban strukturálhatjuk vele a programunkat
   - Nem tanultatok nem-OO nyelveket, ezért erre „igazi” példát nem tudunk mutatni
   - De OO nélkül tapasztalhatjuk, hogy bizonyos műveletek megadott adatokhoz tartoznak
   - Mindig paraméterként át kell adni az adatokat, és ez kényelmetlen, nem alkotnak egységet

# Egységbe zárás példa

```
def fill_deck(deck):
    for color in ["♣", "♦", "♥", "♠"]:
        for shape in [2, 3, 4, 5, 6, 7, 8, 9,
                      "T", "J","Q", "K", "A"]:
            deck.append((color, shape))
    random.shuffle(deck)


def draw_from_deck(deck):
    return deck.pop()


my_deck = []
fill_deck(my_deck)
card = draw_from_deck(my_deck)
```

# Egységbe zárás példa

```
class Deck:
    def __init__(self):
        self.deck = []
        for color in ["♣", "♦", "♥", "♠"]:
            for shape in [2, 3, 4, 5, 6, 7,
                          8, 9, "T", "J",
                          "Q", "K", "A"]:
                self.deck.append((color,shape))
        random.shuffle(self.deck)

    def draw(self):
        return self.deck.pop()


deck = Deck()
card = deck.draw()
```

# OO elvek

- Egységbe zárás: ld. előző példa, adatok és műveletek egy egységet alkotnak
- Adatrejtés: adatokat csak metódusokkal érjük el
- Öröklés
- Polimorfizmus

# Adatrejtés példa

```
public class Product1 {
    public int netPrice;
    public int vatPercent;
    public int grossPrice;

  public static void main(String[] args) {
        Product1 product = new Product1();
        product.netPrice = 1000;
        product.vatPercent = 25;
        product.grossPrice = 1500;// 1250?
        product.netPrice = -1000; // ??

  }
}
```

# Adatrejtés példa

```
private int netPrice;
private int vatPercent;
private int grossPrice;

public int getNetPrice() {
    return netPrice;
}

public void setNetPrice(int netPrice) {
    if (netPrice <= 0)
        throw new IllegalArgumentException(
                "Net price must be positive");
    this.netPrice = netPrice;
    this.grossPrice = (int) (this.netPrice *
            ((100 + this.vatPercent) / 100.0));
}
```

# Adatrejtés példa

```
public int getGrossPrice() {
    return (int) (this.netPrice *
             ((100 + this.vatPercent) / 100.0));
}
```

# Öröklés

- Szintén jól illeszkedik a hétköznapi gondolkodáshoz
- Modellezhetünk általános kategóriaként több dolgot, és az általános működés öröklődik
- Az öröklött működés kiegészíthető, felülírható

# Öröklés példa

```
public class Employee extends Person {

    private String officeNumber;

    public Employee() {
    }

    public Employee(String name, LocalDate birthDate,
                    String officeNumber) {
        super(name, birthDate);
        this.officeNumber = officeNumber;
    }

    public String getOfficeNumber() {
        return officeNumber;
    }

    public void setOfficeNumber(String officeNumber) {
        this.officeNumber = officeNumber;
    }
}
```

# Polimorfizmus

- Az öröklés IS-A („AZ EGY”) kapcsolatot jelöl
  - Pl. a bogár az egy rovar
  - Tehát minden jellemzővel bír, amivel a rovarok, de esetleg másképp viselkedik
  - Ahol egy rovar típusú objektumra van szükségünk, ott bogárt is adhatunk
  - Deklarált (statikus) és futásidejű (dinamikus) típus
- Dinamikus kötés: nem a deklarált, hanem a futásidejű típus alapján dől el, hogyan viselkedik az objektum

# ArrayList vs. LinkedList

- A polimorfizmus segítségével deklarálhatunk List őstípust, és a felhasználás jellegétől függően más leszármazottat példányosítunk
- Mindig a választott futásidejű típusnak megfelelő algoritmus fut
- ArrayList: tömböt használ, indexelés, végére fűzés hatékonyabb, de beszúrás, törlés költségesebb
- LinkedList: láncolt listát használ, beszúrás, törlés könnyű, indexelés nehéz

# ArrayList vs. LinkedList

```
start = Instant.now();
for (int i = 0; i < 10_000; i++) {
    nums.add(0, rnd.nextInt()); // LinkedList jobb
    // nums.add(rnd.nextInt()); // ArrayList jobb
}
end = Instant.now();
System.out.println("Filled list in "
        + Duration.between(start, end));
```

# Az öröklés további lehetőségei

- Lehet több ősosztály? ➔ Javában nem!
  - Túl bonyolult, problémákat vet fel
  - ld. Diamond problem
- Viszont néha praktikus lenne, ha egy osztály több különböző típusnak is meg tudna felelni.
- Ezért vezették be az interfészeket
  - Csak előír metódust, nem implementál

# Interfész példa

```
public class Employee extends Person
        implements Comparable<Employee> {

    // ...

    @Override
    public int compareTo(Employee o) {
        return this.getName().compareTo(o.getName());
    }
}
```

# Az absztrakt osztályok

- Az absztrakt osztályok átmenetet jelentenek interfész és osztály közt
- Némely metódust csak előírnak, némelyhez adhatnak implementációt
- Közvetlen nem példányosíthatóak, csak olyan leszármazottjuk, amely minden előírt metódust megvalósít
- Sokszor adott a logika egy része, más pedig majd a leszármazottaktól függ

# Absztrakt osztály példa

```
public class PersonTableModel extends AbstractTableModel {

    private List<Person> personList;

    public PersonTableModel(List<Person> personList) {
        this.personList = personList;
    }

    @Override
    public int getRowCount() {
        return personList.size();
    }

    @Override
    public int getColumnCount() {
        return 2;
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Person person = personList.get(rowIndex);
        return columnIndex ==
                0 ? person.getName() : person.getBirthDate();
    }
}
```

# Hierarchia tervezése

- Legyen interfész ➔ minden testre szabható
- Legyen absztrakt osztály ➔ amihez lehet, adhatunk alapértelmezett implementációt
- El lehet dönteni, melyikből öröklünk
- Pl.
  - List interfész: teljesen általános saját lista
  - AbstractList: pl. addAll() meg van írva az add()-re visszavezetve
- Pl.
  - WindowListener interfész sok eseményhez
  - WindowAdapter üres implementációkkal

# Ha néhány dolog...

- ...csak paraméterekben tér el ➔ egyazon osztály példányai
- ...metódusokban is eltér, de sok a közös ➔ saját leszármazott osztályok

# Az öröklés, mint kétélű fegyver

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0;

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

# Az öröklés, mint kétélű fegyver

```
public class Prism extends Rectangle {

    private int height;

    public Prism() {}

    public Prism(int a, int b, int height) {
        super(a, b);
        this.height = height;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }
}
```

# Öröklés vs. delegálás

- A fenti esetekben a delegációt kellene inkább használni
- Ez azt jelenti, hogy tagváltozóként tárolunk egy másik objektumot, és neki delegálunk hívásokat
- Szokás HAS-A relációként is hívni.

# Delegálás

```java
public class Prism {
    private Rectangle base;
    private int height;

    public Prism(int a, int b, int height) {
        this.base = new Rectangle(a, b);
        this.height = height;
    }

    public int getA() {
        return base.getA();
    }

    public void setA(int a) {
        base.setA(a);
    }

    // …
}
```

# A SOLID-elvek

# Minták és „best practice”-ek

- Jót lopni itt nem bűn
- Később részletesebben lesznek a tárgyban

Architectural patterns

Design patterns

Idioms

# A SOLID-elvek

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

# Single Responsibility Principle

- Egy osztálynak egy felelőssége legyen (egy ok miatt változhasson)
- Példa: riport összeállítása és nyomtatása
  - Rokon dolgok, de mégis különböző
  - Ok a változásra 1.: más formátum szükséges
  - Ok a változásra 2.: másképp kell nyomtatni
- Előnyök:
  - Robosztusság: egyik változtatással nem rontjuk el a másik komponenst
  - Átláthatóság
  - Jobb tesztelhetőség

# Kohézió/csatolás

- Cohesion (kohézió): egy komponensbe tartozó logikák mennyire szorosan függenek össze ➔ erős/gyenge kohézió
- Coupling (csatolás): két komponensnek mennyire kell ismernie egymás belső működését ➔ szoros/laza csatolás
- Erős kohézió, laza csatolás az előnyös
  - Egymást elősegítik
  - A Single Responsibility is ezt segíti elő

# SRP példa

```
public class PaymentService {

    public void transferPayment(Employee initiator,
                                Employee recipient,
                                int amount) {
        if (isAuthorizedToPay(initiator)) {
            // ... transfer payment
        } else {
            // ... log unauthorized access
        }
    }

    public boolean isAuthorizedToPay(Employee user)
{
        // ... check access

        return false;
    }
}
```

# SRP példa javítva

```java
public class PaymentService {

    private AuthorizationService authorizationService;

    public PaymentService(AuthorizationService
                              authorizationService){
        this.authorizationService = authorizationService;
    }

    public void transferPayment(Employee initiator,
                                Employee recipient,
                                int amount) {
       if (authorizationService.isAuthorizedToPay(initiator)) {
            // ... transfer payment
        } else {
            // ... log unauthorized access
        }
    }
}
```

# Open/Closed Principle

- A komponensek legyenek nyitottak a bővítésre, de zártak a módosításra
- Másképpen: anélkül tudjuk őket bővíteni, hogy a meglévő működéshez hozzá kellene nyúlni

# OCP példa

```java
public class AreaCalculatorBad {

    public double calculateArea(List<Shape> shapes) {
        double area = 0;
        for (Shape s : shapes) {
            if (s instanceof Rectangle) {
                Rectangle r = (Rectangle) s;
                area += r.getA() * r.getB();
            } else if (s instanceof Circle) {
                Circle c = (Circle) s;
                area += c.getRadius() * c.getRadius() * Math.PI;
            } else
                throw new UnsupportedOperationException(
                        "Unsupported shape");
        }
        return area;
    }
}
```

# OCP példa

```java
public interface Shape {
    double area();
}

public class AreaCalculator {

    public double calculateArea(List<Shape> shapes) {
        double area = 0;
        for (Shape s : shapes) {
            area += s.area();
        }
        return area;
    }
}
```

# Liskov Substitution Principle

- Objektumokat lehessen leszármazott osztályával kicserélni, a helyes működés megsértése nélkül
- Ezt mondja a polimorfizmus is, de az csak a technikai megvalósíthatóság
- A lényeg: úgy is írjuk a kódot, hogy a szabálynak eleget tegyen
- Pl. lista listaként is viselkedjen, tárolja el, és adja is vissza, amit beletettünk
  - https://docs.oracle.com/javase/10/docs/api/java/util/List.html

# Interface Segregation Principle

- Az osztály ne függjön olyan interfészektől, amelyeknek a metódusait nem használja ki
- Gyakorlatban: használjunk specifikus, kicsi interfészeket a nagy interfészek helyett
- (Single Responsibility-vel és erős kohézióval analóg)

# ISP példa

```java
public interface Car {
    void setSeatHeating(boolean value);
    void setTurnSignalLeft(boolean value);
    void setTurnSignalRight(boolean value);
}
```

# ISP példa

```
public class BmwX5 implements Car {
    @Override
    public void setSeatHeating(boolean value) {
        // TODO
    }

    @Override
    public void setTurnSignalLeft(boolean value) {
        throw new UnsupportedOperationException("BMWs do"+
                " not have turn signal");
    }

    @Override
    public void setTurnSignalRight(boolean value) {
        throw new UnsupportedOperationException("BMWs do"+
                " not have turn signal");
    }
}
```

# ISP példa javítva

```java
public interface SeatHeating {
    void setSeatHeating(boolean value);
}



public class BmwX5 implements SeatHeating {
    @Override
    public void setSeatHeating(boolean value) {
        // TODO
    }
}


//Not only for cars …

public class WinterVehicleTesting {

    public bool TestHeating(SeatHeating subjectVehicle) {
        …
        subjectVehicle.setSeatHeating(true);

        …
    }
}
```

# Dependency Inversion Principle

- Absztrakcióktől függjünk, ne konkrét dolgoktól
- Nagy vonalakban:
  - Használjunk interfészeket, absztrakt osztályokat a függőségeknek
  - A függőségeket ne a függő hozza létre, hanem kívülről kapja
- Ld. későbbi előadások

# A tervezés szintjei

- Főleg osztályokról beszéltünk, de sok dolog általánosítható kisebb/nagyobb egységekre is
- Az elvek megértése után adja magát
- Pl. metódus is csak egy dolgot csináljon
- Pl. package-ek is minél kevesebbet tudjanak a másik működéséről

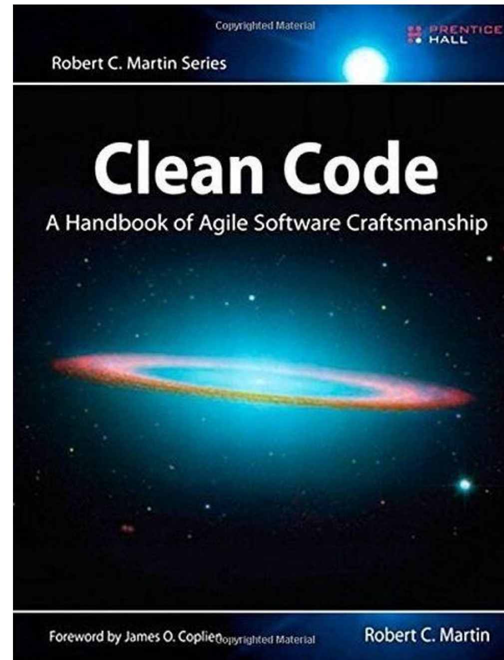
# További tervezési elvek

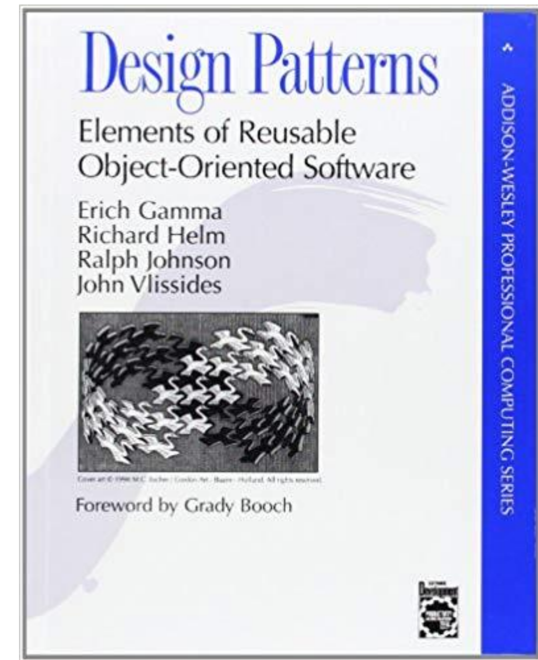
- Keep it simple stupid (KISS):
  - Feleslegesen ne bonyolítsuk a kódot
- You aren't gonna need it (YAGNI)
  - Amíg nem bizonyos, hogy valami kell, addig ne fejlesszük ki
  - Az idő pénz, a fejlesztésnek költsége van
  - Az extra funkciók ráadásul a komplexitást is növelik

# További tervezési elvek

- Do not repeat yourself (DRY)
  - A kódduplikáció problémát okoz(hat)
  - Minden feladatot egyszer oldjunk meg egy helyen
  - Ha ismétlődés van, szervezzük ki egy helyre
  - Az ismétlődés feleslegesen növeli a kód méretét, rontja az áttekinthetőséget
  - És ha módosítani kell, inkonzisztenciát okozhat, ha nem írjuk át egységesen
  - Drasztikusan nehezedik a kód karbantartása
  - **Kivétel: biztonsági aspektusok!**
    - Mindenhol ellenőrizzünk
    - Ha valahol hiányos, a másik rész még védhet
    - Ld. BKK e-jegy

# Bibliográfia



# Köszönöm a figyelmet!