Princípy OOP

Asociácia – používam hlavne v triedach používateľov, menovite v **skladnik.java**, kde veľa funkcií vyžaduje objekt **sklad** a **kniha** na správne fungovanie. Príklad funkcie z triedy **skladnik.java**

```
public void odoberKnihyZRegalu(Sklad s, Kniha k, int pocet, int[] pozicia){
    if(kniha != null) return;
     if(pozicia == null){
         NoveKnihy r = s.getNovyTovar();
         if(r != null && r.existujeKniha(k)) {
             int p = r.odoberKnihy(k, pocet);
             pridajKnihy(k, p);
         }
     }else{
         Regal r = s.getSekcie(pozicia[0]).getRegal(pozicia[1]);
         if(r.existujeKniha(k)){
             int p = r.odoberKnihy(k,pocet);
             pridajKnihy(k,p);
         }
     }
}
```

Agregácia – tento princíp sme použili pri spôsobe ukladania kníh v **regáli** (**Regal.java**) a **palete (NoveKnihy.java). Knihy** nie sú súčasťou jedného regálu ale môžu sa premiestňovať po všetkých regáloch a navyše **sklad.java** pracuje s katalógom **kníh**, kde sú uložene referencie na knihy.

```
public class Regal {
    protected ArrayList<Kniha> zoznamKnih;
    protected HashMap<String, Integer> pocetKnih;
protected void pridajKnihyP(Kniha k, int p){
     if(!existujeKniha(k)){
         zoznamKnih.add(k);
         pocetKnih.put(k.getISBN(),p);
     }else{
         pocetKnih.replace(k.getISBN(),pocetKnih.get(k.getISBN()) + p);
     }
}
```

Kompozícia – je súčasťou kníhkupectva ako celku, pretože **kníhkupectvo (knihkupectvo.java)** by nemalo existovať bez toho aby v sebe nemalo **sklad (sklad.java)** a **predajňu (predajna.java)**. A zase **sklad** ani **predajna** by nemala existovať bez knihkupectva. Tento vztah je aj medzi **skladom** a **sekciami**; a ešte medzi **sekciami** a **regalmi.**

```
public Knihkupectvo(){
    sklad = new Sklad();
    predajna = new Predajna();
}
```

Dedenie – umožňuje, aby funkcionalita z jednej triedy bola súčasťou druhej. Tento vzťah sme uplatnili medzi **Miestnost.java** a **Sklad.java/Predajna.java**. A medzi **Regal.java** a **NoveKnihy.java,** ktoré obidve slúžia na ukladanie kníh určitou formou. **Viacnásobne dedenie** sme využili pri triedach používateľov, kde napr. **skladník** dedí atribúty a funkcie **zamestnanca** a ten zase dedí **používateľa**, ktorý je ešte dedení zákazníkom.

```
abstract class Zamestnanec extends Pouzivatel{

public class Skladnik extends Zamestnanec{
```

Pretože niektoré triedy nepotrebujú aby sa dali vytvoriť ich inštancie, napr. **Pouzivatel.java, Zamestnanec.java** a **Miestnost.java**, tak na tieto triedy sme použili **abstrakciu**.

Odvíja sa nám z toho **Polymorfizmus** typu **Upcasting,** kde mám premennú typu **Pouzivatel** používam v **Knihkupectvo.java** vo funkcií **main.** Táto premenná ukazuje na prihláseného používateľa a spúšťa funkciu na **spracovanie vstupu**.

```
public abstract class Pouzivatel {    ...    }

Pouzivatel p = new Zakaznik();

p.spracuj(command, knihkupectvo.sklad, knihkupectvo.predajna);
```

Abstraktná metóda – využitá v abstraktnej triede **Miestnost.java,** kde každá trieda ktorá dedí by si mala, ak chce metódu používať **init(int)**, by si ju mala sama implementovať.

```
protected abstract void init(int velkost);
```

Trieda i Miestnost.java

Enkapsulácia – všetky nestatické atribúty sú **private**, prípadne ak sa v triede používa atribút svojej rodičovskej triedy, tak v rodičovskej triede je tento atribút **protected.** Skoro všetky tieto atribúty majú nastavení **getter** ale iba niektoré majú nastavene **setter**.

```
public class Kniha {
    private String nazov, autor, isbn, jazyk, vydavatelstvo;
    private Kategoria kategoria;
    private Vazba vazba;
    private int pocetStran, rok, predaneKusy;
    private float cena;


public Kategoria getKategoria() { return kategoria; }
public String getISBN() { return isbn; }
public String[] getBasicInfo() { return new String[] {nazov,autor,isbn,vydavatelstvo};}
public String getVydavatel() {return vydavatelstvo;}
public boolean isBestseller() { return kategoria == Kategoria.BESTSELLER; }

public void setBestseller() { kategoria = Kategoria.BESTSELLER; }
```

Trieda ii Kniha.java

```
public void setOtvorene(boolean b) { otvorene = b; }

public boolean isOtvorene() { return otvorene; }
```

Trieda iii Predajna.java

Overriding – tento princíp nám dovoľuje z dedených tried zobrať funkcie a upraviť ich na nami žiadanú funkcionalitu. Využil som ho hlavne v dedeniach triedy **používateľ**, kde sa nachádza funkcia **spracuj**, ktorá spracováva vstupné príkazy. Ďalej som ho použil v triede **NoveKnihy.java**, ktorá dedí triedu **Regal.java** , tak že funkcia **odoberKnihy** namiesto uvoľňovania miesta kontroluje či ešte zostali knihy v **NoveKnihy.**

```
public int pridajKnihy(Kniha k, int p){
    if(p == 0) return 0;
    if(volneMiesto < p){
        p = volneMiesto;
    }
    pridajKnihyP(k,p);
    volneMiesto -= p;

    return p;
}

public int odoberKnihy(Kniha k, int p){
    int v = odoberKnihyP(k,p);
    volneMiesto += v;
    return v;

}
```

Trieda iv Regal.java

```
@Override
public int odoberKnihy(Kniha k, int p){
    int v = odoberKnihyP(k,p);
    skontrolujMinutie();
    return v;

}

@Override
public int pridajKnihy(Kniha k, int p){
    System.out.println("Na paletu sa nedaju dat knihy");
    return -1;
}
```

Trieda v NoveKnihy.java

Overloading – príklad overloading-u mám v triede **skladnik.java**, kde skladník môže nájsť referencie na **knihu** pomocou reťazca znakov alebo podľa katalógového čísla. Využil som aj preťažovanie konštruktorov; hlavne v **Sekcia.java**

```
public Kniha najdReferenciuNaKnihu(Sklad s, int i) { return s.getKatalog().get(i); }

public Kniha najdReferenciuNaKnihu(Sklad s, String id){
    for(Kniha kp : s.getKatalog()){
        if(kp.getISBN().toLowerCase().equals(id) || kp.getBasicInfo()[0].toLowerCase().equals(id)){
            return kp;
            }
    }
    return null;
}
```

```
public Sekcia(){init(defaultSize);}
public Sekcia(int pocetRegalov) { init(pocetRegalov); }
```

Interface – slúži ako predpis čo by mali triedy ktorého implementujú vyzerať. Tento princíp máme v používateľoch, kde najvyššia trieda **Pouzivatel.java** implementuje rozhranie **InputProcess.java** diktujúce, že triedy musia implementovať funkcie **spracuj(…)** a **help()**.

```
public interface InputProcess {
    void spracuj(String[] s, Knihkupectvo kh);
    void help();
}
```

```
public void help(){
    System.out.println("na retazenie prikazpv pouzi: <prikaz> | <prikaz> | ...");
    System.out.println("---Vseobecne prikazy---");
    System.out.println("info-me - informacie o mne");
    System.out.println("katalog - vypise katalog predajne");
    System.out.println("logout - odhlasit sa");
    System.out.println("help - vypis pomocky");
    System.out.println("exit - vypni system");
}

@Override
public void spracuj(String[] s, Knihkupectvo kh){
    int index = 0;
        while(index < s.length) {
            if (inlineAkcie.containsKey(s[index])) {
                inlineAkcie.get(s[index]).process(s, kh);
            }
            int k;
            for (k = index + 1; k < s.length; k++) {
                if (s[k].equals("|")) break;
            }
            index = k + 1;
        }
}
```

Trieda vi Pouzivatel.java

Statické atribúty – tento druh atribút som použil v práci s predefinovanou veľkosťou **predajne, skladu, sekcii** a aj **regálov**, ale aj na zistenie či sa užívateľ od posledného vykonania zmenil a nachádza sa v triede **Knihkupectvo.java.**

```
public abstract class Miestnost implements java.io.Serializable{
    public static final int defaultSize = 5;
```

Trieda vii priklad statickej atributy z Miestnost.java

```
public static boolean changeUser = false;
```

Trieda viii staticky atribut na zmenu pouzivatela z Knihkupectvo.java

Statické metódy – statické metódy používame v **Knihkupectvo.java** na serializáciu a deserializáciu dát zo súboru *knihkupectvo.ser*

```java
public static void serialize(String path){
    try{
        FileOutputStream fileOut = new FileOutputStream(path);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(instancia);
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void deserialize(String path){
    try {
        FileInputStream fileIn = new FileInputStream( name: "./knihkupectvo.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        instancia = (Knihkupectvo) in.readObject();
        in.close();
        fileIn.close();
        return;
    } catch (IOException e) {
        System.out.println("Nenasiel sa subor 'knihkupectvo.ser' -- vytvara sa nove knihkupectvo'");
    } catch (ClassNotFoundException e) {
    }
    getInstance();
}
```

Finálne atribúty – sú v programe použité na vzťah kompozície v **Knihkupectvo.java**, kde inštancia tejto triedy nikdy nemení **sklad** ani **predajňu,** preto tieto dve atribúty môžu byť označené ako **final**. Ale ako bolo ukázané vyššie tak používame aj statické premenné s označením **final**

```java
private final Sklad sklad;
private final Predajna predajna;
```

Trieda ix Knihkupectvo.java

```java
public final static int miesto = 300;
```

Trieda x Regal.java

Finálna trieda – implementovaná ako trieda **Kniha.java**, pretože každá kniha je predsa len kniha, či už je hrubá alebo tenká, takže každá kniha spadá pod túto triedu.

```java
public final class Kniha implements java.io.Serializable{
```

Finálne metódy – príkladom finálnych metód sú funkcie na ukladanie kníh v regáli v **Regal.java**, tieto funkcie sú podstate backend-om pre prekonateľné funkcie **pridajKnihy(…)** a **odoberKnihy(...)**. Jedná sa o **pridajKnihyP(...)** a **odoberKnihyP(...)**, ktoré vykonávajú logiku ukladania kníh a tá je iba jedna, a teda by žiadnym dedičom nemali byť prekonané. Ďalším príkladom je **printKatalog()** v **Miestnost.java**, kde je táto funkcia označená za final a to kvôli tomu že by nemal existovať žiadny iný spôsob výpisu atribútu **katalog**.

```
protected final void pridajKnihyP(Kniha k, int p){
    if(!existujeKniha(k)){
        zoznamKnih.add(k);
        pocetKnih.put(k.getISBN(),p);
    }else{
        pocetKnih.replace(k.getISBN(),pocetKnih.get(k.getISBN()) + p);
    }
}

protected final int odoberKnihyP(Kniha k, int p){
    if(existujeKniha(k)){
        int pocet = getPocetKnih(k.getISBN());
        int vymazanych = p;
        if(pocet <= p){
            vymazanych = pocet;
            zoznamKnih.remove(k);
            pocetKnih.remove(k.getISBN());
        }else{
            pocetKnih.replace(k.getISBN(),pocet-p);
        }

        return vymazanych;
    }
    return -1;
}
```

Trieda xi Regal.java

```
public final void printKatalog() {
    int i = 0;
    for(Kniha k : katalog){
        String[] s = k.getBasicInfo();
        System.out.println(i++ + ": " + s[0] + " - " + s[1] + " - { " +s[2] + " - " +s[3] + " }");

    }
}
```

Trieda xii Miestnost.java

Singleton a privátny konštruktor – tento návrhový model sme použili v triede **Kníhkupectvo.java**, teda môže naraz existovať iba jediná inštancia tejto triedy, teda v našom programe existuje nanajvýš jedno knihkupectvo

```
public class Knihkupectvo implements java.io.Serializable{
    private static Knihkupectvo instancia = null;
    private final Sklad sklad;
    private final Predajna predajna;
    private LoggedIn prihlaseny = LoggedIn.NONE;
    public static boolean zmenaUzi = false;

    private Knihkupectvo(){
        sklad = new Sklad();
        predajna = new Predajna();
    }

    public static Knihkupectvo getInstance()
    {
        if (instancia == null)
            instancia = new Knihkupectvo();
        return instancia;
    }
```