

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológií  
Ilkovičova 2, 842 16 Bratislava 4

# Lagatoria

Objektovo-orientované programovanie

Róbert Junas

FIIT STU

Cvičenie: streda 14:00

Cvičiaci: Ing. Anna Považanová

16.5.2021

Id: 102970

## Obsah

Obsah.....	2
1. Rámcové zadanie.....	3
2. Štruktúra .....	4
3. Kritéria .....	5
3.1. Hlavné kritéria .....	5
3.1.1 Náplň zadania .....	5
3.1.2 Prvá hierarchia.....	5
3.1.3 Druhá hierarchia .....	10
3.1.4 Oddelenie aplikačnej logiky od používateľského rozhrania .....	14
3.1.5 Organizácia do balíkov .....	14
3.2. Ďalšie kritéria .....	15
3.2.1 Návrhové vzory .....	15
3.2.2 Výnimky .....	22
3.2.3 GUI.....	26
3.2.4 Multithreading .....	27
3.2.5 RTTI.....	27
3.2.6 Vhniezdené triedy .....	28
3.2.7 Lambda výrazy .....	28
3.2.8 Implicitná implementácia v rozhraní.....	29
3.2.9 Použitie serializácie .....	30
4. Verzie .....	31

## 1. Rámcové zadanie

Knihy sú dôležitou súčasťou každého z nás. Preto ich plánovanie a vydávanie je veľmi dôležité. Môj projekt sa zaoberá vydávaním kníh do stánkov a vydateľskej predajne. Teda výsledkom plánovania bude kniha predávajúca sa v kníhkupectve.

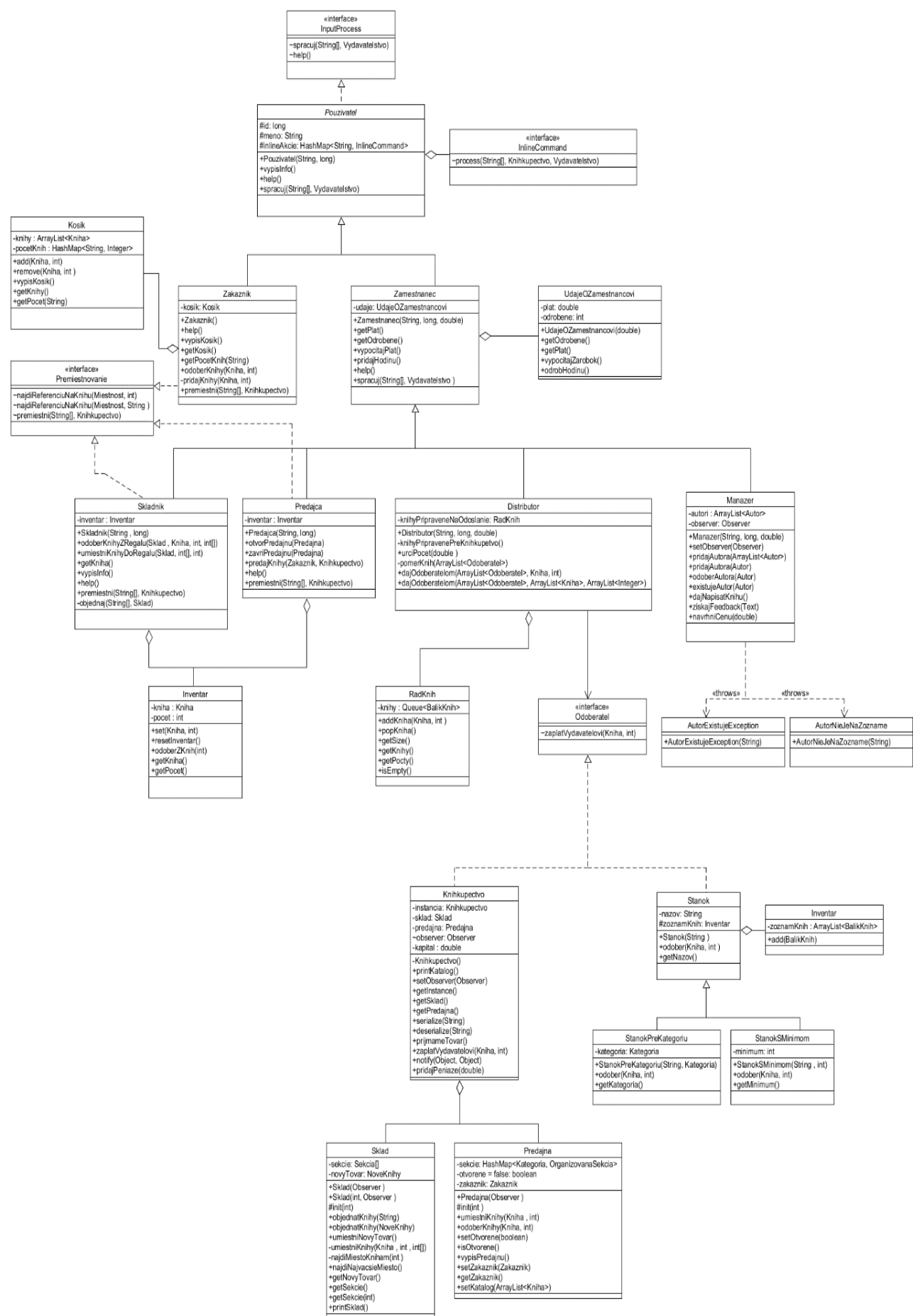
Autor musí najprv knihu napísať – dostane nápad, napíše ju, vymyslí názov. Potom dá svoju knihu vydateľstvu.

Vydavateľstvo sa skladá z ľudí – manažéra, ktorý je v kontakte s autorom, dizajnér, ktorý navrhne obálku a vyberie väzbu knihy; a korektor opravujúci chyby v knihe. Manažér navrhne cenu knihy, zistí aký dopyt je po knihe a nakoniec distribútor povie koľko kníh sa má vytlačiť. V tomto procese sa knihy dajú na tlač, kde sa text pripevní k väzbe. Potom distribútor rozdelí knihy a pošle ich odoberateľom.

Kníhkupectvo sa skladá z predajne a skladu. Predajňa môže byť organizovaná do viacerých kategórií, ako je žáner/druh. V sklade sa ale ukladajú knihy podľa toho, na ktoré miesto, prípadne viac príľahlých miest, sa zmestia. V kníhkupectve pracuje predajca a skladník. Predajca obsluhuje zákazníka a dopĺňa knihy do predajne. Skladník pridáva knihy do skladu a premiestňuje ich do iných sektorov skladu, aby mohol urobiť miesto pre nové knihy. Zákazník môže dať knihy do košíka, ktoré mu následne predajca predá.

2. Štruktúra

Obsahuje dve hlavné hierarchie dedenia Používateľov a odoberateľov.



### **3. Kritéria**

#### **3.1. Hlavné kritéria**

##### **3.1.1 Náplň zadania**

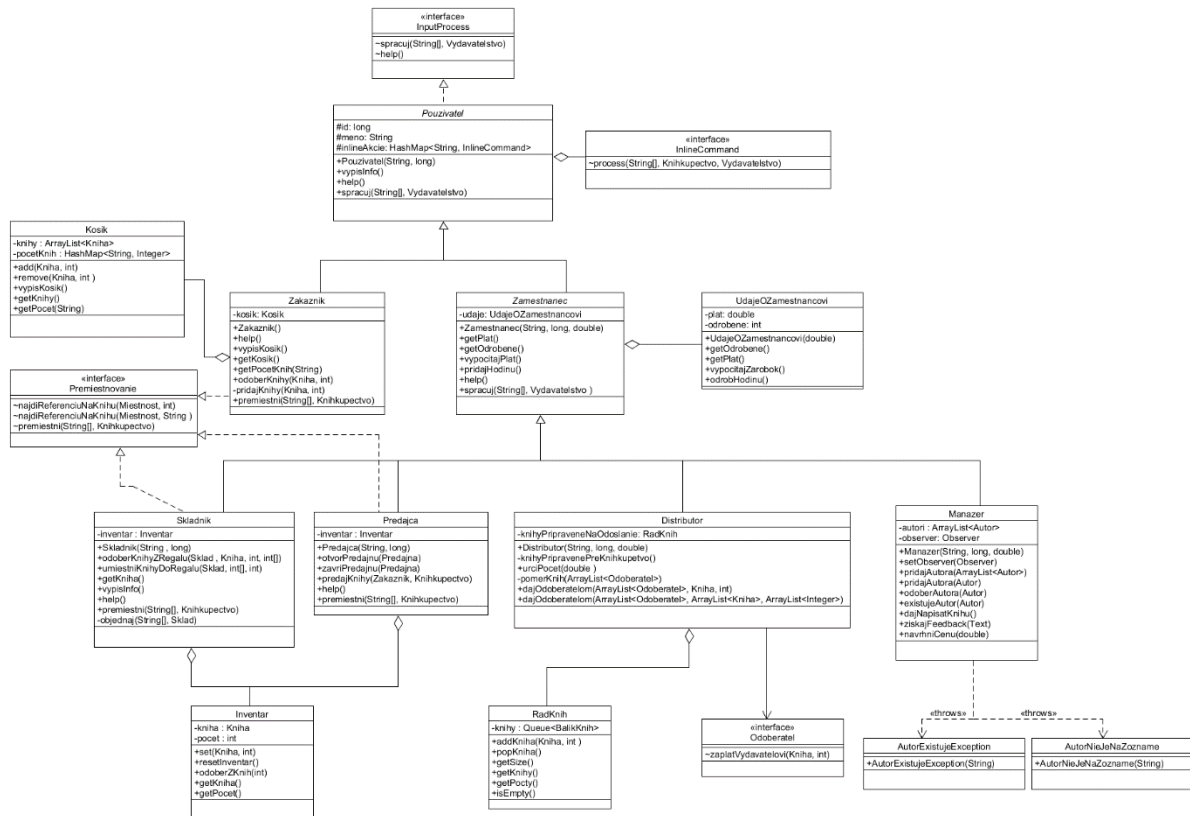
Podľa nášho názoru sa nám podarilo splniť cieľ zadania, čo bolo vytvorenie procesu vydávania kníh ako aj plánovania ich distribúcie ceny a počtu vytlačených kníh. Knihy prijíma Kníhkupectvo iba ak sú splnené podmienky, že sa pošlú 4 rôzne knihy v hocijakom počte kusov a kníhkupectvo je schopné prijať nový tovar do svojho skladu. Ale napríklad na stánky sa nedávajú takéto limity, tie sú schopné prijať hocikedy, ale sú aj stánky, ktoré neprijmú knihy, ktoré nespĺňajú ich požiadavky, napr. prijímajú iba určitú kategóriu.

Proces vytvorenia knihy začína u manažéra, ktorý pošle požiadavku autorom, aby napísali knihu. Tí o svojom postupe informujú vydavateľstvo a následne pošlú texty vydavateľstvu, kde sa pridajú do radu čakania na vydanie. Manažér si môže vybrať akým spôsobom bude vydávať knihy, napr. vydanie iba jednej knihy alebo vydanie všetkých kníh na zozname. Keď sa vyberie kniha, tak najprv dizajnér navrhne obálku, následne si korektor prečíta text a opraví v ňom chyby a skráti text. Následne manažér, zistí aký je po knihe dopyt, podľa čoho sa určí počet výtlačkov a cena. Ešte pred samotným tlačením knihy, tlačiareň vytvorí ISBN identifikátor pre text a nakoniec tlačiareň spojí obálku a text do jednej knihy. Potom sa buď najprv kniha vloží do zoznamu kníh pripravených na distribúciu, s ktorým potom distribútor pracuje (ak vydávame všetky texty) alebo sa kniha pošle hneď distribútorovi na odoslanie odoberateľom. Distribútor určí pomer v akom sa budú knihy posielat' odoberateľom a následne ich im pošle. Nepredané knihy sa vyhadzujú.

##### **3.1.2 Prvá hierarchia**

Prvá hierarchia dedenia je hierarchia zamestnancov/používateľov.

Na vrchu je rozhranie InputProcess, ktorý je implementovaný abstraktnou triedou Používateľ, následne táto trieda je dedená ďalšou abstraktnou triedou Zamestnanec a triedou zákazníka. Zamestnanca následne dedia Skladník, Predajca, Distribútor a Manažér, (aj vnorené triedy Korektor a Dizajnér).



## Rozhranie

Používatelia implementujú rozhranie inputProcess, kde je zadenovaná funkcia *String spracuj(String[] args, Vydavatelstvo vydavatelstvo)* na spracovanie príkazov.

```

public interface InputProcess {
    /**
     * funkcia na spracovanie vstupov do aplikacnej logiky
     * @param args argumenty, ktoré sa dajú vykonať
     * @param vydavatelstvo referencia na vydavatelstvo nad ktorým sa robia metódy
     * @return retazec výstupu
     */
    String spracuj(String[] args, Vydavatelstvo vydavatelstvo);

    /**
     * @deprecated iba konzolová verzia;
     * vypis funkcie, ktoré môže užívateľ urobiť
     * @return retazec informácií o funkciách
     */
    String help();
}

```

```

public abstract class Pouzivatel implements InputProcess {

```

## Polymorfizmus

Polymorfizmus sa v tejto triede vyskytuje tak, že Používateľ implementuje spôsob akým sa spúšťajú funkcie pomocou funkcie pouzivatel a cez funkciu spracuj():

```
@Override
public String spracuj(String[] args, Vydavatelstvo vydavatelstvo){
    String res = "";
    int index = 0;
    //prejde všetky slova zadane na vstupe
    while(index < args.length) {
        //ak sa slovo nachádza v zavolateľných funkciách
        if (inlineAkcie.containsKey(args[index])) {
            //zavola sa funkcia
            res += inlineAkcie.get(args[index]).process(args, Knihkupectvo.getInstance(), vydavatelstvo) + "\n";
        }
        int k;
        //najde ďalší príkaz rozdelený /
        for (k = index + 1; k < args.length; k++) {
            if (args[k].equals("/")) break;
        }
        index = k + 1;
    }
    return res;
}
```

Následne táto metóda sa prekonáva v triede Zamestnanec, kde spracováva funkcie rovnako ako používateľ, ale pri každom zavolaní zvýši zamestnancovi počet odrobených hodín:

```
@Override
public String spracuj(String[] args, Vydavatelstvo vydavatelstvo){
    String res = super.spracuj(args, vydavatelstvo);
    pridajHodinu();
    return res;
}
```

Funkcia je vyvolávaná z triedy model.java, kde sa pri zmenách používateľa nastaví do premennej typu Pouzivatel práve prihlásený používateľ. Z tejto premennej sa volá funkcia spracuj:

```
private Pouzivatel pouzivatel;

public String spracuj(String command) { return pouzivatel.spracuj(command.split(" "), vydavatelstvo); }
```

Funkcie sú zadávané do hašovacej tabuľky, kde kľúč tvoria názvy funkcií a ukladajú sa lambda výrazy, ktoré spúšťajú funkcie. Napr. funkcie predajcu:

```
//pridavanie akcií ktoré môže spraviť trieda
inlineAkcie.put("otvor", (args, kh, vy) -> otvorPredajnu(kh.getPredajna()));
inlineAkcie.put("zavri", (args, kh, vy) -> zavriPredajnu(kh.getPredajna()));
inlineAkcie.put("predajna", (args, kh, vy) -> kh.getPredajna().vypisPredajnu());
inlineAkcie.put("sklad", (args, kh, vy) -> kh.getSklad().printSklad());
inlineAkcie.put("predaj", (args, kh, vy) -> predajKnihy(kh.getPredajna().getZakaznik(), kh));
inlineAkcie.put("prines", ((args, kh, vy) -> premiestni(args, kh)));
```

## Zapuzdrenie

Každý atribút v hierarchii je privátny, prípadne pri niektorých sme použili protected. Ku mnohým atribútom ani nie je možné prístupit' cez gettersy a settersy, keďže zostávajú iba vo funkciách. Prípadne, ak sa v atribútoch menia hodnoty, tak sa volajú funkcie, ktoré ich prepíšu:

```
public abstract class Zamestnanec extends Pouzivatel{

    /**
     * udaje zamestnanca ako su plat a odrobeny cas
     */
    protected UdajeOZamestnancovi udaje;

    /**
     * Prida zakladne funkcie zamestnanca ku zakladnym funkciam pouzivatelya
     * @param meno meno zamestnanca
     * @param id identifikacne cislo
     * @param plat plat zamestnanca
     */
    public Zamestnanec(String meno, long id, double plat) {...}

    /**
     * @return vrati plat zamestnanca
     */
    public double getPlat() { return udaje.getPlat(); }

    /**
     * @return vrati pocet odrobenych hodin
     */
    public double getOdrobene() { return udaje.getOdrobene(); }

    /**
     * @return vrati zarobok aky zamestnanec za beh aplikacie zarobil
     */
    public double vypocitajPlat() { return udaje.vypocitajZarobok(); }

    /**
     * prida hodinu zamestnancovi
     */
    public void pridajHodinu(){ udaje.odrobHodinu(); }
```



Príklad getteru v hierarchii:

```
public class Zakaznik extends Pouzivatel implements Premiestnovanie {  
    private Kosik kosik;  
  
    /**  
     * Rozsiruje funkcie pouzivatelya o funkcie zakaznika.  
     * Meno zakaznika je Guest a id je 0  
     */  
    public Zakaznik(){...}  
  
    @Override  
    public String help() {...}  
  
    /**  
     * @return vrati kosik zakaznika  
     */  
    public Kosik getKosik(){  
        return kosik;  
    }  
}
```

Setter v hierarchii:

```
public class Manazer extends Zamestnanec {  
  
    private Observer observer;  
  
    public void setObserver(Observer observer) { this.observer = observer; }
```

### Agregácia

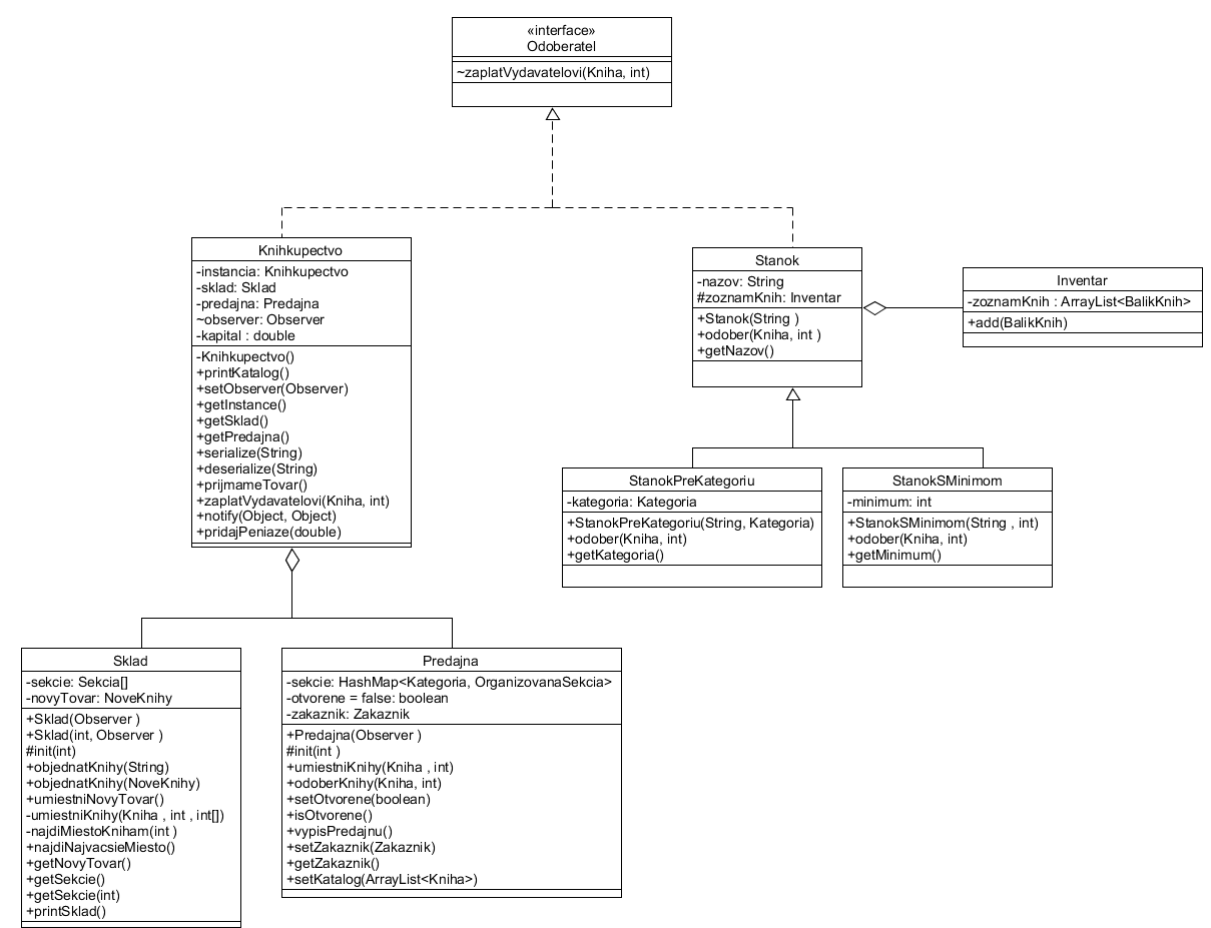
Agregácia bola použitá napr. v abstraktnej triede Pouzivatel, kde sa agregujú funkcie používateľov. Ďalej v Zamestnancovi sa agreguje trieda UdajeOZamestnancovi; v Zákaznikovy je trieda Kosik; v Predajcovi a Skladníkovi sa agreguje inventár, v Distribútorovi sa agreguje trieda RadKnih a v Manažérovi sa agregujú autori, ktorých upozorňuje.

```
public abstract class Pouzivatel implements InputProcess {  
    protected long id;  
    protected String meno;  
    protected HashMap<String, InlineCommand> inlineAkcie;  
  
    public abstract class Zamestnanec extends Pouzivatel{  
        protected UdajeOZamestnancovi udaje;  
  
        public class Zakaznik extends Pouzivatel implements Premiestnovanie {  
            private Kosik kosik;
```

```
public class Predajca extends Zamestnanec implements Premiestnovanie {  
    private Inventar inventar = new Inventar();  
}  
  
public class Distributor extends Zamestnanec {  
  
    /**  
     * Knihy ktore sa rozposielaju knihkupectvu  
     */  
    private RadKnih knihyPripraveneNaOdoslanie;  
}  
  
public class Manazer extends Zamestnanec {  
    /**  
     * autori cakajuci na pisanie  
     */  
    private ArrayList<Autor> autori = new ArrayList<>();  
}
```

3.1.3 Druhá hierarchia

Ako druhú hierarchiu máme odoberateľov, kde máme zadefinované rozhranie Odoberateľ, ktoré implementujú triedy Knihkupectvo a Stanok. Ďalej je stánok dedený triedami StanokSMinimumom a StanokPreKategoriu



Rozhranie

Rozhranie v tejto triede definuje implicitne funkciu na zaplatenie vydavateľovi, ktorá vracia 77% percent ceny knihy.

```
public interface Odoberateľ {  
    /**  
     * @param kniha kniha ktorú sme prijali  
     * @param pocet prijatých kníh  
     * @return celková cena za zaplatenie kníh - 77% z plnej ceny knihy  
     */  
    default double zaplatVydavateľovi(Kniha kniha, int pocet){  
        return kniha.getCena()*0.77*pocet;  
    };  
}
```

### Polymorfizmus

Metódu definovanú v rozhraní prekonáva kníhkupectvo, kde si platí iba 50% knihy a odpočítava si zo svojho kapitálu.

```
/**  
 * vypocet kolko musi Knihupectvo zaplatit za knihy vydavateľovi  
 * @param kniha kniha za ktorú sa platí  
 * @param pocet pocet prebratej knihy  
 * @return celková cena ktorú kníhkupectvo zaplatilo  
 */  
@Override  
public double zaplatVydavateľovi(Kniha kniha, int pocet) {  
    kapital -= kniha.getCena()*0.50*pocet;  
    return kniha.getCena()*0.50*pocet;  
}
```

Ďalej v triede Stanok je metóda odober(...), ktorá pridá knihy na zoznam kníh bezpodmienečne, ale deriváty stánku napr. StanokSMinimom práva knihy len v prípade, že počet posielaných kníh presiahol minimum.

Stanok.java:

```
/**  
 * Prijme knihu a pocet kusov jednej knihy a vlozi ich do inventara ako balik  
 * @param kniha ktorú sme prebrali  
 * @param pocet kusov ktoré sme prebrali  
 * @return vždy vracia 1  
 */  
public int odober(Kniha kniha, int pocet) {  
    zoznamKnih.add(new BalikKnih(kniha, pocet));  
    return 1;  
}
```

StanokPreKategoriu.java:

```
/**
 * @param kniha kniha, ktoru prijame
 * @param pocet pocet knih na prijatie
 * @return ak kategoria knihy je rôzna od kategorie, tak knihy neprijme (0) inak vracia 1
 */
@Override
public int odober(Kniha kniha, int pocet) {
    if(((Text)kniha.getSucast(Text.class)).getKategoria() == kategoria) {
        zoznamKnih.add(new BalikKnih(kniha,pocet));
        return 1;
    }
    return 0;
}
```

StanokSMinimom.java:

```
/**
 * @param kniha prijmana kniha
 * @param pocet pocet prijmanych knih
 * @return ak počet bol väčší alebo rovný minimu tak sa vrati 1 inak sa vracia 0
 */
@Override
public int odober(Kniha kniha, int pocet){
    if(pocet >= minimum){
        zoznamKnih.add(new BalikKnih(kniha, pocet));
        return 1;
    }
    return 0;
}
```

Využitie v Distributor.java vo funkcii dajOdoberatelom(...):

```
if(stanok.odober(knihy.get(i), (int)(pocet.get(i)*pomer.get(o))) == 1) {
```

### Zapuzdrenie

Každý atribút v kníhkupectve je private a v Stanku je názov private a inventar je protected, pretože, každý ďalší stánok potrebuje pracovať s inventárom, ale už si nebudú meniť názov.

Príklad z Knihkupectvo.java:

```
public class Knihkupectvo implements java.io.Serializable, Odoberatel, Observer {
    private static Knihkupectvo instancia = null;
    private Sklad sklad;
    private Predajna predajna;
    transient private Observer observer;
    private double kapital = 100000; //kolko penazi ma knihkupectvo k dispozicii
```

```
/** nuti sklad o odoslanie upozornenia o zmene katalogu */
public void printKatalog() { sklad.printKatalog(); }

/** Nastavenie sledovateľa knihkupectva a nastavenie knihkupectva ako sledovateľa skladu a predajne ...*/
public void setObserver(Observer observer) {
    this.observer = observer;
    sklad.setObserver(this);
    predajna.setObserver(this);
}

/** @return vracia instanciu na Singleton Knihkupectvo */
public static Knihkupectvo getInstance()
{
    if (instanciia == null)
        instanciia = new Knihkupectvo();
    return instanciia;
}

/** @return vrati sklad */
public Sklad getSklad() { return sklad; }

/** @return vrati predajnu */
public Predajna getPredajna() {return predajna;}
```

Stanok.java:

```
public class Stanok implements Odoberatel {
    private String nazov;
    protected Inventar zoznamKnih;

    /** @return nazov stanku */
    public String getNazov() { return nazov; }
```

StanokPreKategoriu.java:

```
public class StanokPreKategoriu extends Stanok {
    private Kategoria kategoria;

    public Kategoria getKategoria() { return kategoria; }
```

StanokSMinimumom.java:

```
public class StanokSMinimumom extends Stanok {
    private int minimum;

    public int getMinimum() { return minimum; }
```

### Agregácia

V knihkupectve sa agreguje sklad, predajňa a objekty implementujúce observer. Ďalej v Stánku sa agreguje trieda inventár.

Knihkupectvo.java

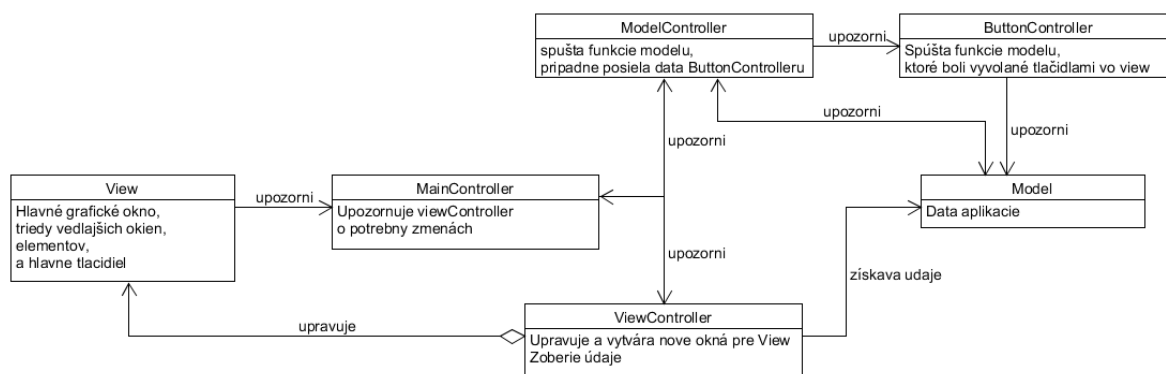
```
public class Knihkupectvo implements java.io.Serializable, Odoberatel, Observer {
    private static Knihkupectvo instancia = null;
    private Sklad sklad;
    private Predajna predajna;
    transient private Observer observer;
    private double kapital = 100000; //kolko penazi ma knihkupectvo k dispozicii
```

Stanok.java

```
public class Stanok implements Odoberatel {
    private String nazov;
    protected Inventar zoznamKnih;
```

### 3.1.4 Oddelenie aplikačnej logiky od používateľského rozhrania

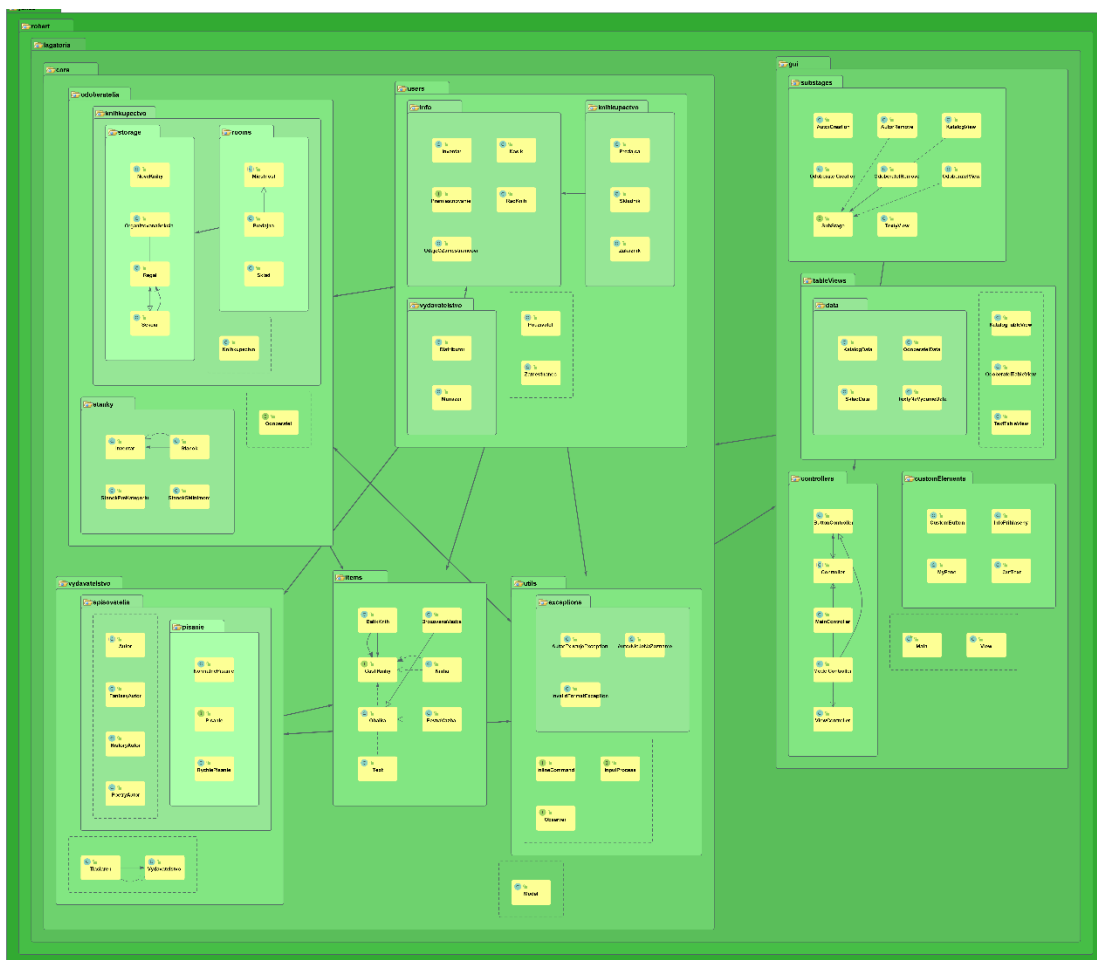
Oddelenie aplikačnej časti od rozhrania sme spravili pomocou MVC modelu, kde aplikačná logika a rozhranie je rozdelené tak, že model nijako nepracuje s rozhraním a ani naopak. Namiesto toho komunikujú cez controller pomocou modelu observer.



### 3.1.5 Organizácia do balíkov

- Junas.robort.lagatoria
  - core – triedy aplikačnej logiky.
    - Items – vytvárané produkty v našom prípade knihy.
    - Odoberatelia – triedy odoberateľov.
      - Knihkupectvo – triedy potrebné na správne fungovanie knihkupectva.
        - Rooms – miestnosti knihkupectva.
        - Storage – ukladacie priestory, napr. regal.
      - Stanky – triedy stánkov a inventár stánku.
    - Users – triedy používateľov a triedy pre ich správne fungovanie
      - Info – triedy obsahujúce triedy ako inventár, UdajeOZamestnancovi atď.
      - Kníhkupectvo – používatelia knihkupectva.
      - Vydavateľstvo – používatelia vydavateľstva.
    - Utils – pomocné triedy.
      - Enums – pomocné enumerácie.

- Exceptions – triedy našich výnimiek.
- Vydavateľstvo – obsahuje triedy na správne fungovanie vydavateľstva.
  - Spisovatelia – triedy autorov, ktorý môžu písať
    - písanie – obsahuje visitor triedy spôsobu ako autori píšu.
- gui – triedy pre správne fungovanie grafického rozhrania.
  - controllers – ovládače na prepojenie modelu so zobrazením.
  - customElements – triedy vychádzajúce z javaFx elementov, napr. tlačidla, text, atď.
  - substages – sekundárne okná napr. tabuľkové zobrazenie, pridávanie autorov atď.
  - tableViews – definovanie tabuliek aké vieme zobrazovať
    - data – triedy na parsovanie dát z modelu do tabuliek.

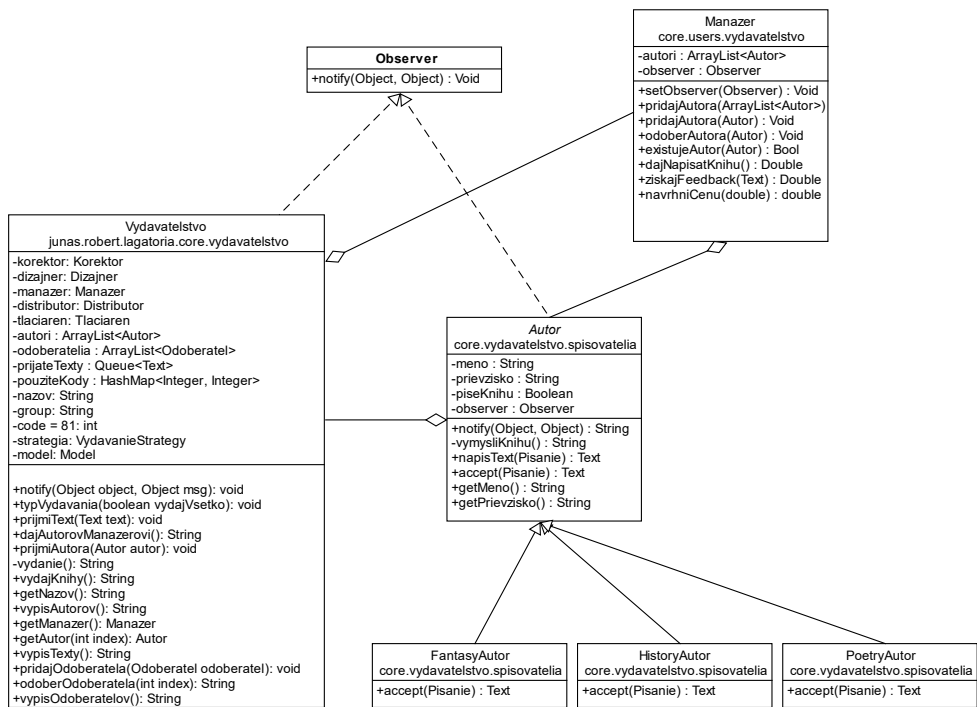


### 3.2. Ďalšie kritéria

### 3.2.1 Návrhové vzory

## Observer

Model Observer sme využili naprieč celým projektom, či už na komunikáciu ovládačov s View alebo Modelom, ale aj medzi triedami Manažér a Autor, kde Manazer upovedomí svojich autorov, že chce aby napísali knihu.



```
public void dajNapisatKnihu(){
    observer.notify( caller: this, msg: "Manazer rozposiela ziadosti o knihu\n");
    for (Autor autor: autori) {
        autor.notify( caller: this, msg: "");
    }
}
```

Observer bol využitý medzi autormi a vydavateľstvom, kde autor upovedomuje vydavateľstvo o svojom postupe v písaní textu.

```
public String vymysliKnihu() {
    observer.notify( caller: this, msg: "Autor [" + getMeno() + " " + getPrievzisko() + "] vymyslel knihu");
    int targetStringLength = 10;
    Random random = new Random();

    return random.ints( randomNumberOrigin: 97, randomNumberBound: 123)
        .limit(targetStringLength)
        .collect(StringBuilder::new, StringBuilder::appendCodePoint, StringBuilder::append)
        .toString();
}

public synchronized void odosliVydavatelovi(Text text){
    observer.notify( caller: this, text);
    piseKnihu = false;
}
```

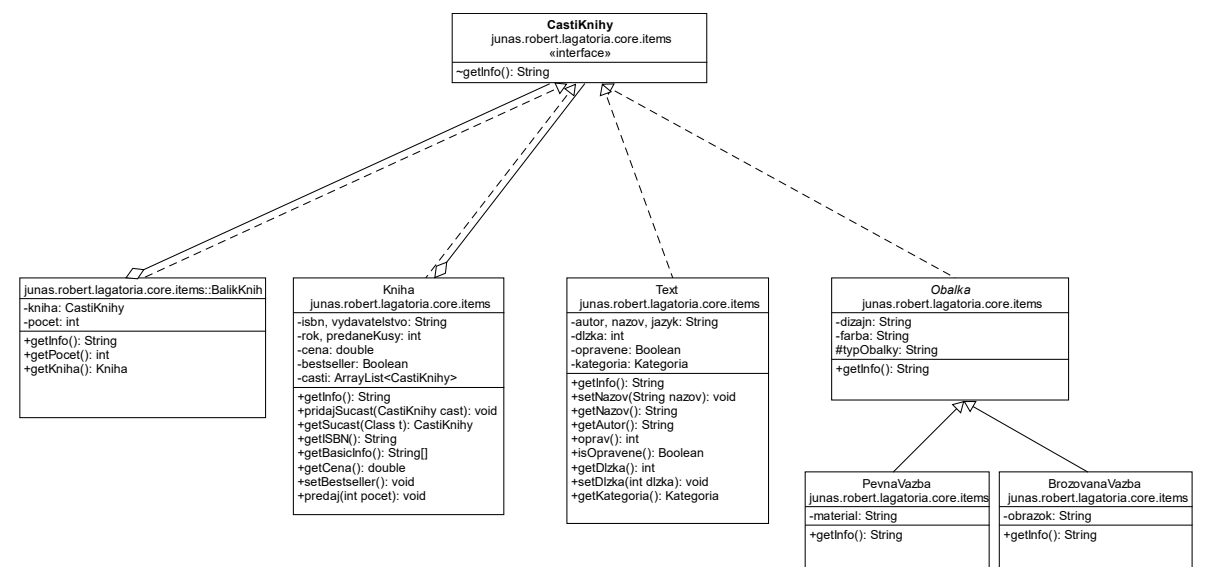
Máme definované aj rozhranie Observer, kde je definovaný prototyp funkcie notify(Object, Object).



```
/**
 * implementuju triedy ktore maju sledovat urcite instance
 */
public interface Observer {
    /**
     * @param caller instancia, ktora vyvolala upovedomovanie
     * @param msg sprava ktoru rozposiela
     */
    void notify(Object caller, Object msg);
}
```

Composite

Využívaný v spojení s knihami, kde kniha sa skladá z textov a obálky a nakoniec aj knihy sa balia do balíkov, kde je ich väčší počet.



```
/**
 * zakladny inteface pre navrhovy vzor Composite pre Knihy
 */
public interface CastiKnihy extends Serializable {
    /**
     * vypisuje informacie o knihe
     */
    String getInfo();
}
```

```
public class Kniha implements CastiKnihy {
    private String isbn, vydavatelstvo;
    private int rok, predaneKusy;
    private double cena;
    private Boolean bestseller;
    private ArrayList<CastiKnihy> casti;

    public Kniha(String isbn, String vydavatelstvo, int rok, double cena){...}

    @Override
    public String getInfo() {
        String res = "";
        for(CastiKnihy c : casti){
            c.getInfo();
        }
        res += "\tISBN: " + isbn + '\n';
        res += "\tcena: " + String.format("%.2f",cena) + "€ - vydavatel: " + vydavatelstvo + " " + rok +"\n";

        return res;
    }

    public void pridaJSucast(CastiKnihy cast){
        if(cast instanceof Obalka){
            for(CastiKnihy c : casti){
                if(c instanceof Obalka){
                    return;
                }
            }
        }

        casti.add(cast);
    }

    public CastiKnihy getSucast(Class t){
        for(CastiKnihy c : casti){
            if(t == c.getClass()){
                return c;
            }
        }
        return null;
    }
}

/**
 * Obalka a cast knihy
 */
public abstract class Obalka implements CastiKnihy {
    private String dizajn;
    private String farba;
    protected String typObalky;

    public Obalka(String dizajn, String farba){
        this.dizajn = dizajn;
        this.farba = farba;
    }

    @Override
    public String getInfo() {
        String res = "";
        res+="\tObalka:\n";
        res+="\t\tdizajn: " + dizajn + ", farba: " + farba + "\n";

        return res;
    }
}
```

```
public class Text implements CastiKnihy {
    private String autor, nazov, jazyk;
    private int dlzka;
    private Boolean opravene;
    private Kategoria kategoria;

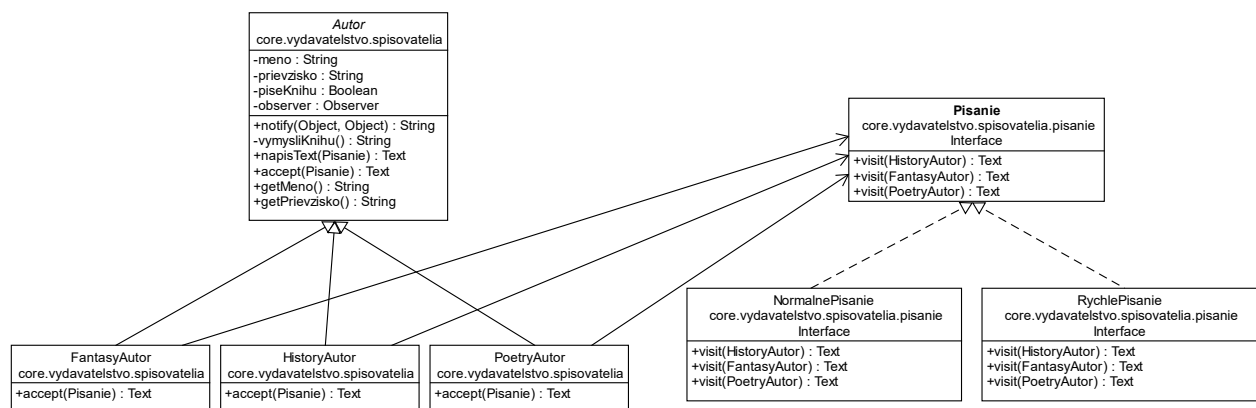
    public Text(String nazov,String autor, String jazyk, int dlzka, Kategoria kategoria, Boolean opravene){
        this.opravene = opravene;
        this.autor = autor;
        this.nazov = nazov;
        this.jazyk = jazyk;
        this.dlzka = dlzka;
        this.kategoria = kategoria;
    }

    public Text(String nazov,String autor, String jazyk, int dlzka, Kategoria kategoria){
        this.opravene = false;
        this.autor = autor;
        this.nazov = nazov;
        this.jazyk = jazyk;
        this.dlzka = dlzka;
        this.kategoria = kategoria;
    }

    @Override
    public String getInfo(){ return "\t["+kategoria.toString()+"] "+autor + ": " + nazov + " [" + jazyk+ "]\n"; }
```

Visitor

V projekte sa vyskytuje aj model Visitor a to tak, že každý druh autora píše iným spôsobom, teda trvá mu to kratšie alebo dlhšie, vzhľadom koľko práce musí dať do samotného písania. Máme definované dva spôsoby písania: Normálne a rýchle písanie. Počet strán sa odráža od rýchlosti akou autor písal.



```
public interface Pisanie {

    Text visit(HistoryAutor autor) throws InterruptedException;
    Text visit(FantasyAutor autor) throws InterruptedException;
    Text visit(PoetryAutor autor) throws InterruptedException;

}
```

```
public class NormalnePisanie implements Pisanie{

    @Override
    public Text visit(HistoryAutor autor) throws InterruptedException {
        Thread.sleep( millis: 8000);
        int pocetStran = (int)(Math.random()*(1200-300+1)+300);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov Historickkej knihy: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.HISTORIA);
    }

    @Override
    public Text visit(FantasyAutor autor) throws InterruptedException {
        Thread.sleep( millis: 5000);
        int pocetStran = (int)(Math.random()*(700-300+1)+300);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov romanu: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.FANTASY);
    }

    @Override
    public Text visit(PoetryAutor autor) throws InterruptedException {
        Thread.sleep( millis: 7000);
        int pocetStran = (int)(Math.random()*(200-50+1)+50);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov basniciek: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.POEZIA);
    }
}

public class RychlePisanie implements Pisanie {
    @Override
    public Text visit(HistoryAutor autor) throws InterruptedException {
        Thread.sleep( millis: 5000);
        int pocetStran = (int)(Math.random()*(900-300+1)+300);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov Historickkej knihy: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.HISTORIA);
    }

    @Override
    public Text visit(FantasyAutor autor) throws InterruptedException {
        Thread.sleep( millis: 2500);
        int pocetStran = (int)(Math.random()*(500-300+1)+300);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov romanu: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.FANTASY);
    }

    @Override
    public Text visit(PoetryAutor autor) throws InterruptedException {
        Thread.sleep( millis: 3000);
        int pocetStran = (int)(Math.random()*(100-50+1)+50);
        String meno = autor.getMeno() + " " + autor.getPriezvisko();
        return new Text( nazov: "Nazov basniciek: ",meno, jazyk: "Slovensky",pocetStran, Kategoria.POEZIA);
    }
}

@Override
public Text accept(Pisanie pisanie) throws InterruptedException {
    return pisanie.visit( autor: this);
}
```

Použitie v triede Autor.java

```
public Text accept(Pisanie pisanie) throws InterruptedException {
    return null;
};
```

```
final Pisanie pisanie;
if((int)(Math.random()*5) == 0){
    pisanie = new RychlePisanie();
}else{
    pisanie = new NormalnePisanie();
}
Thread thread = new Thread(new Runnable() {
    String nazov = "";
    @Override
    public void run() {
        Text text = null;

        Runnable myslienky = new Runnable() {...};

        Runnable hotovo = new Runnable() {...};

        try {...} catch (InterruptedException e) {...}
        Platform.runLater(myslienky);

        try {
            text = autor.napisText(pisanie);
            text.setNazov(nazov);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        autor.odosliVydavatelovi(text);
        Platform.runLater(hotovo);
    }
});
```

## Strategy

V projekte je využitý návrhový vzor Strategy, ktorý určuje stratégiu akou sa budú vydávať knihy. Je definovaný ako vnorené rozhranie triedy Vydavatelstvo.java, kde sa určuje akou formou sa bude vydávať.

```
interface VydavanieStrategy{
    String vydajKnihy();
}
```

Zmena stratégie vydávania, buď vydanie všetkých kníh alebo na vydanie jednej knihy, sa mení pomocou referencie na funkciu a lambda výrazov.

```
/**
 * Funkcia urci strategiu akou sa budu knihy vydavat (Bud vsetky naraz alebo iba jedna)
 * Strategia ma dopad na pocet vytlakov aj cenu knihy
 * pri tlaceni vsetkych sa zhorsuje kvalita teda aj pocet vytlakov sa zhorsuje
 * @param vydajVsetko ak je parameter true tak sa vydaju vsetky knihy v rade, inak sa vydava po jednej
 */
public void typVydavania(boolean vydajVsetko){
    if(vydajVsetko){
        strategia = () -> {
            String res = "";
            ArrayList<Kniha> vytlaceneKnihy = new ArrayList<>();
            ArrayList<Integer> pocetVytlakov = new ArrayList<>();
            if(prijateTexty.isEmpty()){...}
            while(!prijateTexty.isEmpty()) {...}
            res += distributor.DajOdoberatlom(odoberatelja, vytlaceneKnihy, pocetVytlakov);
            return res;
        };
    }else{
        strategia = this::vydanie;
    }
}
```

Zavolanie funkcie uloženej v premennej typu VydavanieStrategy vo Vydavatelstvo.java:

```
/**
 * Vydá text ktorý sa nachadza na vrchu radu, do ktoreho sa text dostane cez funkciu prijmiText(Text text)
 * @return vysledok vydavania, resp. kolko aky stanok prijal a aká kniha bola vydaná a v akom mnozstve + zarobok
 */
public String vydajKnihy(){
    String result = strategia.vydajKnihy();
    model.notify( objekt: this, msg: "001::"+vypisTexty());
    return result;
}
```

### 3.2.2 Výnimky

V aplikácii máme implementovaných niekoľko výnimiek: AutorExistujeException, AutorNieJeNaZozname, InvalidFormatException.

InvalidFormatException sa vyhadzuje práve vtedy, keď v súbore z ktorého načítavame knihy, nie sú knihy zadane v správnom formáte. Správny formát je:

Nazov///Meno Prievzisko///ISBN///Kusy///Strany///Cena///Kategoria///Jazyk  
Nazov///Meno Prievzisko///ISBN///Kusy///Strany///Cena///Kategoria///Jazyk

InvalidFormatException:

```
/**
 * Vyhadzuje sa prave vtedy ked citani subor obsahuje knihu v zle zadanom formate
 * format: nazov///Autor///ISBN///pocet Kusov///pocet Stran///cena///Kategoria///jazyk///vazba///vydavatel///rok
 * kontroluje sa ci pocet Kusov, Stran, rok ci su zadane ako Int a cena ci je typu float
 */
public class InvalidFormatException extends Exception{

    private int loadedRows;

    /**
     * @param message chybove hlasiene
     * @param loadedRows cislo riadku s chybou
     */
    public InvalidFormatException(String message, int loadedRows){
        super(message);
        this.loadedRows = loadedRows;
    }

    /**
     * @return vrati pocet nacitanych riadkov vratane chyboveho
     */
    public int getLoadedRows() { return loadedRows; }
}
```

Funkcia vyhadzujúca výnimku:

```
public boolean nacistajKnihy(String path) throws InvalidFormatException, FileNotFoundException {...}
```

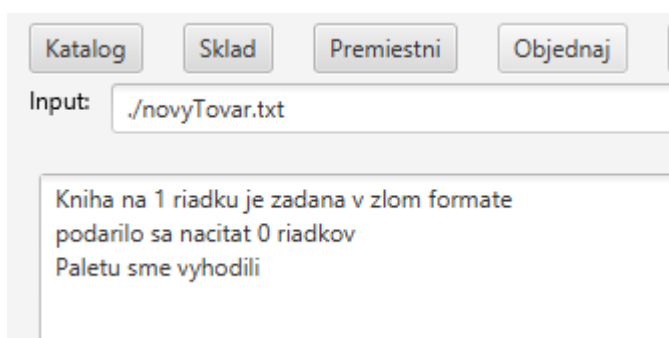
Konštruktor chytá výnimku:

```
public NoveKnihy(String path) throws FileNotFoundException, InvalidFormatException{
    super();
    try {
        minute = !nacistajKnihy(path);
    } catch (InvalidFormatException e) {
        if(e.getLoadedRows() > 0){
            minute = false;
        }
        else{
            minute = true;
        }
        throw e;
    } catch (FileNotFoundException e){
        minute = true;
        throw e;
    }
}
```

Preukázanie vyhadzovania:

```
Hlava 22///Joseph HellerISBN-978-80-556-4294-9200///406///19.99///SVETOVA_LIT///Slovensky jazyk///PEVNA///slovart///2019
Dnešok nie je naposledy///Elan Mastai///ISBN-978-80-551-5360-5///200///421///19.99///SVETOVA_LIT///Slovensky jazyk///PEVNA///1
Mengeleho Dievča///Viola Stern Fischerová; Veronika Homolová Tothová///ISBN-978-80-551-5188-5///200///367///19.99///HISTORIA///
```

*prvý riadok je v zlom formáte*



Výnimka **AutorExistujeException** sa vyhadzuje práve vtedy, keď dáme prídanie autorov na manažérov zoznam autorov čakajúcich na písanie.

**AutorExistujeException:**

```
/**
 * Vyhadzuje sa iba vtedy ak manazer uz ma autora vo svojom zozname
 */
public class AutorExistujeException extends Exception{

    public AutorExistujeException(String msg) { super("autor (" +msg+"), uz je na manazerovom zozname"); }
}
```

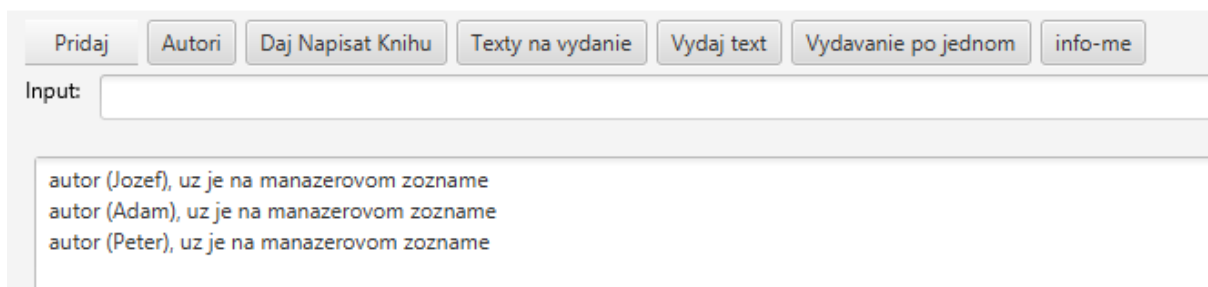
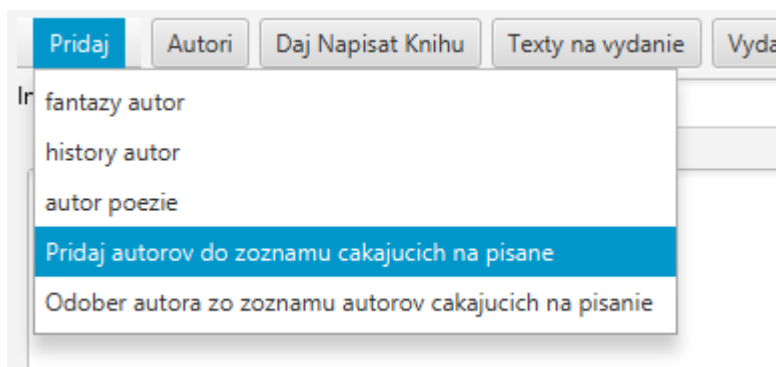
Funkcia vyhadzujúca výnimku:

```
public void pridajAutora(Autor autor) throws AutorExistujeException {  
    if(autor1.contains(author))  
        throw new AutorExistujeException(author.getMeno());  
    else  
        this.autori.add(author);  
}
```

Metóda chytajúca výnimku:

```
/**  
 * Snazi sa pridať vsetkych autorov vo vydavatelstve manazerovi  
 * Osetruje sa výnimka keď autor už daného autora ma  
 */  
public String dajAutorovManazerovi(){  
    String res = "";  
    for (Autor autor: autori) {  
        try {  
            manazer.pridajAutora(author);  
        } catch (AutorExistujeException e) {  
            res += e.getMessage() + "\n";  
        }  
    }  
    return res;  
}
```

Preukázanie vyhadzovania:



Výnimka **AutorNieJeNaZozname** sa vyhadzuje práve vtedy, keď sa snažíme odobrať autora zo zoznamu čakajúcich na písanie.

**AutorNieJeNaZozname:**



```
/**
 * Vyhadzuje sa vtedy keď chceme odobrať autora,
 * ktorý sa nenachádza na zozname a teda sa nedá odobrať
 */
public class AutorNieJeNaZozname extends Exception {
    public AutorNieJeNaZozname(String msg) { super("autor (" + msg + "), nie je na manazerovom zozname"); }
}
```

Funkcia vyhadzujúca výnimku:

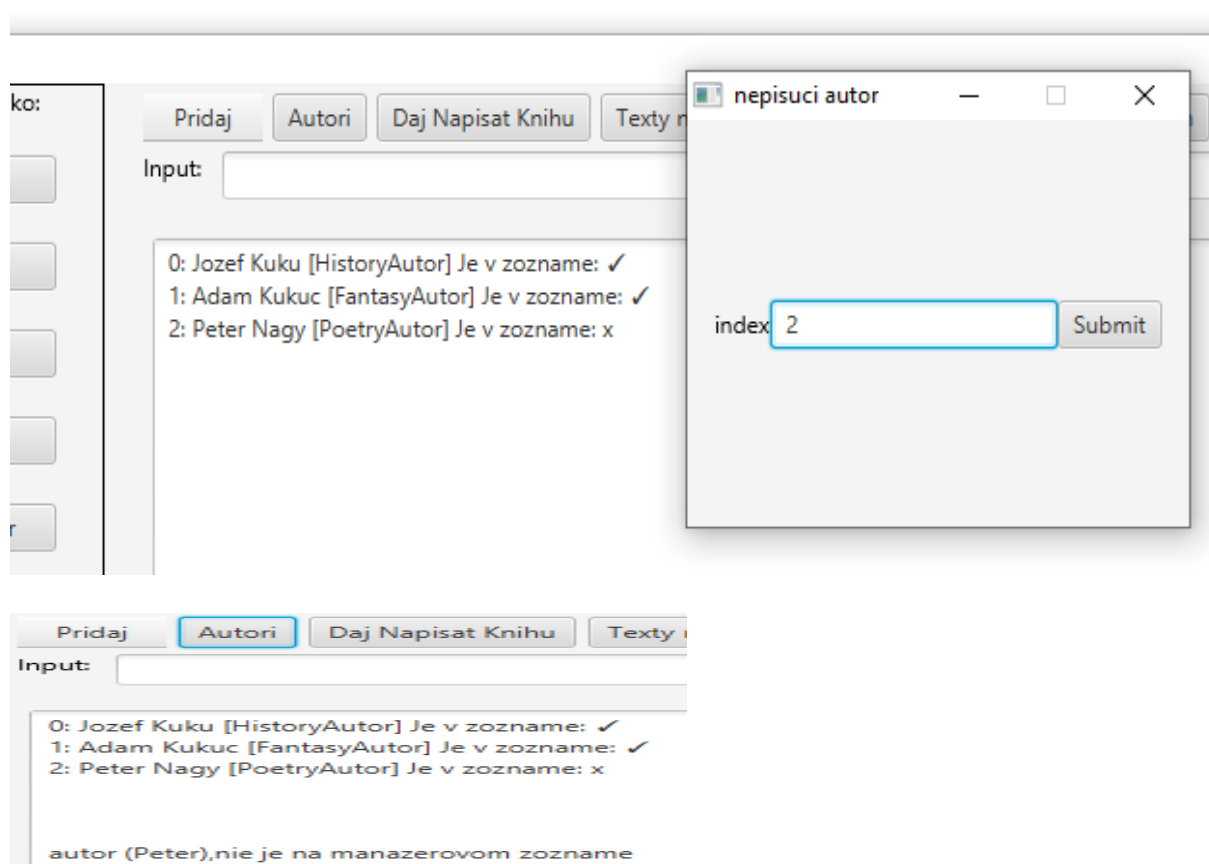
```
public void odoberAтора(Autor autor) throws AutorNieJeNaZozname {
    if (!autori.contains(author)) throw new AutorNieJeNaZozname(author.getMeno());

    autori.remove(author);
}
```

Metóda chytajúca výnimku:

```
try {
    odoberAтора(vy.getAutor(Integer.valueOf(args[1])));
} catch (AutorNieJeNaZozname autorNieJeNaZozname) {
    autorNieJeNaZozname.printStackTrace();
    return autorNieJeNaZozname.getMessage();
}
return "Odobranie autora prebehlo uspesne";
```

Preukázanie vyhadzovania:



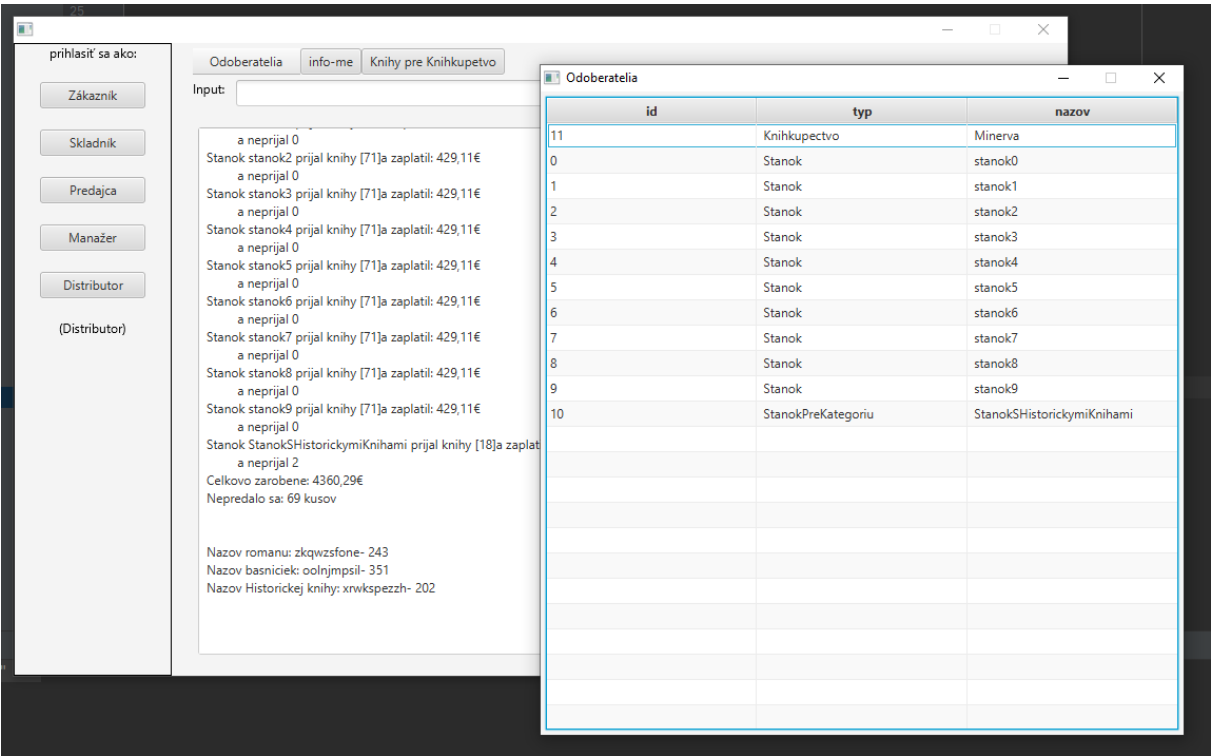
### 3.2.3 GUI

Aplikovali sme model MVC, kde hlavná scéna je vytvorená vo View.java a máme definované aj ďalšie okná ako sú okna na pridanie odobranie autorov/odoberateľov a okná s tabuľkovým zobrazením katalógu kníh, odoberateľov a textov pripravených na vydanie.

Tieto zobrazenia spracováva ViewController, ktorý je upovedomovaný MainController-om, ktorý spracováva vstupy z hlavného okna. Z vedľajších to robí samotný ViewController, ktorý upovedomí Controller, ktorý je nad ním. Prácu s modelom spúšťa ModelController s podkontrolerom ButtonController (spracováva vstupy vyvolané tlačidlami). Zavolajú funkcie v modeli a ak je potrebné tak oznámia ViewControlleru, že treba upraviť View.

Model samotný sa kumuluje v Model.java, kde sa vykonáva celý model a sám o sebe nijako nezasahuje do View, prípadne upovedomí Controller o zmene, ktorú treba vykonať.

Celé grafické rozhranie bolo vytvorené ručne, bez žiadnych nástrojov na vytváranie prostredí.



### 3.2.4 Multithreading

Viacnitosť bola využitá v Autoroch, ktorý sú schopný naraz písať texty, ktoré následne pošlú vydavateľovi.

```
public void notify(Object caller, Object msg){
    if(piseKnihu) {
        observer.notify( caller: this, msg: "Autor neprijal poziadavku");
        return;
    }
    observer.notify( caller: this, msg: "\t" + meno + " " + prievzisko + " prijal poziadavku\n");

    piseKnihu = true;
    Autor autor = this;
    final Pisanie pisanie;
    if((int)(Math.random()*5) == 0){...}else{...}
    Thread thread = new Thread(new Runnable() {
        String nazov = "";
        @Override
        public void run() {
            Text text = null;

            Runnable myslienky = new Runnable() {...};

            Runnable hotovo = new Runnable() {...};

            try {...} catch (InterruptedException e) {...}
            Platform.runLater(myslienky);

            try {
                text = autor.napisText(pisanie);
                text.setNazov(nazov);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            autor.odosliVydavatelovi(text);
            Platform.runLater(hotovo);
        }
    });

    thread.setDaemon(true);
    thread.start();
}
```

### 3.2.5 RTTI

Použitie RTTI je v triede Distribútor, vo funkciách *dajOdobertalom* a *pomerKnih*, kde sa výrazne líši správanie Stánkov a Kníhkupectva. Kníhkupectvo nie vždy musí hneď prijať knihy

od Vydavateľstva a neprijaté knihy sa pridávajú na list kníh čakajúcich na prijatie. Na druhú stranu Stanky, prijímajú knihy neustále, bez zbytočného vyčkávania.

```
public String dajOdoberatelom(ArrayList<Odoberatel> odoberatelia, Kniha kniha, int pocet) {
    HashMap<Odoberatel, Double> pomer = pomerKnih(odoberatelia);
    double kapital = 0;
    int nepredane = pocet;
    String res = "";
    for (Odoberatel o : odoberatelia){
        if(o instanceof Knihupectvo){...}
        else{
            Stanok stanok = (Stanok) o;
            if(stanok.odober(kniha, (int)(pocet*pomer.get(o))) == 1) {
                nepredane -= (int) (pocet * pomer.get(o));
                double zaplatene = stanok.zaplatVydavatelovi(kniha, (int) (pocet * pomer.get(o)));
                res += "Stanok " + stanok.getNazov() + " prijal knihy [" + (int) (pocet * pomer.get(o)) + "] a zaplatil: "
                    + String.format("%.2f", zaplatene) + "\n";
                kapital += zaplatene;
            }else{
                res+= "Stanok " +stanok.getNazov()+ " neprijal knihy\n";
            }
        }
    }
    res += "\nCelkovo zarobene: " + String.format("%.2f", kapital) + "€\n";
    res += "Nepredalo sa: " + nepredane + " kusov\n";

    return res;
}
```

### 3.2.6 Vhniezdené triedy

Vhniezdené triedy a rozhrania sa nachádzajú v triede Vydavatelstvo.java. Implementujeme tu stratégiu vydávania ako aj triedy Korektor a Dizajner, ktoré by mali byť vždy iba súčasťou Vydavateľstva.

```
public class Vydavatelstvo implements Observer {

    interface VydavanieStrategy{
        String vydajKnihy();
    }

    /**
     * Inner class Korektor slúži na opravenie chýb v texte
     */
    class Korektor extends Zamestnanec {...}

    /**
     * inner class dizajner, ma na starosti vymyslenie obalky
     */
    class Dizajner extends Zamestnanec {...}
}
```

### 3.2.7 Lambda výrazy

Jedno využitie lambda výrazov/referencií na metódy sme implementovali v triede Vydavatelstvo.java, kde si ukladáme spôsob vydávania, teda vydá sa práve jedna kniha alebo všetky knihy čakajúce na zozname.

```
/**
 * Funkcia urci strategiu akou sa budu knihy vydavat (Bud vsetky naraz alebo iba jedna)
 * Strategia ma dopad na pocet vytlackov aj cenu knihy
 * pri tlaceni vsetkych sa zhorsuje kvalita teda aj pocet vytlackov sa zhorsuje
 * @param vydajVsetko ak je parameter true tak sa vydaju vsetky knihy v rade, inak sa vydava po jednej
 */
public void typVydavania(boolean vydajVsetko){
    if(vydajVsetko){
        strategia = () -> {
            String res = "";
            ArrayList<Kniha> vytlaceneKnihy = new ArrayList<>();
            ArrayList<Integer> pocetVytlackov = new ArrayList<>();
            if(prijateTexty.isEmpty()){...}
            while(!prijateTexty.isEmpty()) {...}
            res += distributor.DajOdoberatlom(odoberatel, vytlaceneKnihy, pocetVytlackov);
            return res;
        };
    }else{
        strategia = this::vydanie;
    }
}

/**
 * Vydá text ktorý sa nachádza na vrchu radu, do ktoreho sa text dostane cez funkciu prijmiText(Text text)
 * @return vysledok vydavania, resp. kolko aký stanok prijal a aká kniha bola vydaná a v akom množstve + zarobok
 */
public String vydajKnihy(){
    String result = strategia.vydajKnihy();
    model.notify( objekt: this, msg: "001::"+vypisTexty());
    return result;
}
```

Ďalej sme lambda výrazy použili pri volaní funkcií používateľov

```
inlineAkcie.put("otvor", (args, kh, vy) -> otvorPredajnu(kh.getPredajna()));
inlineAkcie.put("zavri", (args, kh, vy)-> zavriPredajnu(kh.getPredajna()));
inlineAkcie.put("predajna", (args, kh, vy) -> kh.getPredajna().vypisPredajnu());
inlineAkcie.put("sklad", (args, kh, vy) -> kh.getSklad().printSklad());
inlineAkcie.put("predaj", (args, kh, vy)-> predajKnihy(kh.getPredajna().getZakaznik()));
inlineAkcie.put("prines", ((args, kh, vy) -> premiestni(args,kh)));
```

### 3.2.8 Implicitná implementácia v rozhraní

Implicitné implementácie metód v rozhraniach sme využili napríklad v rozhraní Odoberateľ:

```
public interface Odoberatel {
    /**
     * @param kniha kniha ktoru sme prijali
     * @param pocet prijatych knih
     * @return celkova cena za zaplatenie knih - 77% z plnej ceny knihy
     */
    default double zaplatVydavateľovi(Kniha kniha, int pocet) { return kniha.getCena()*0.77*pocet; };
}
```

Ale aj v rozhraní Premiestňovanie, ktoré implementujú triedy užívateľov kníhkupectva:

```
public interface Premiestnovanie {  
  
    default Kniha najdiReferenciuNaKnihu(Miestnost s, int i){  
        return (s.getKatalog().isEmpty() || i >= s.getKatalog().size() ) ? null : s.getKatalog().get(i);  
    }  
  
    default Kniha najdiReferenciuNaKnihu(Miestnost s, String id){  
        for(Kniha kp : s.getKatalog()){  
            if(kp.getISBN().toLowerCase().equals(id.toLowerCase())  
                || kp.getBasicInfo()[0].toLowerCase().equals(id.toLowerCase())){  
                return kp;  
            }  
        }  
        return null;  
    }  
  
    String premiestni(String[] args, Knihkupectvo kh);  
  
}
```

### 3.2.9 Použitie serializácie

Serializáciu používame na uloženie stavu kníhkupectva, teda na uloženie stavu predajne, skladu, sekcií, regálov, poličiek, ale aj kníh v uložených v regáloch. Kníhkupectvo sa deserializuje pri spustení programu a serializuje sa, keď sa stlačí tlačidlo na vypnutie okna/programu. Výsledok serializácie sa ukladá do súboru knihkupectvo\_oop.ser v priečinku /res/. Funkcie na serializáciu sú definované v Knihkupectvo.java:

```
/**  
 * Ukladá sa instanciu knihkupectva, knihy v nom a ich ulozenie  
 * @param path cesta k suboru, kde sa uložia informacie o knihkupectve  
 * @return  
 */  
public static String serialize(String path){  
    try{  
        FileOutputStream fileOut = new FileOutputStream(path);  
        ObjectOutputStream out = new ObjectOutputStream(fileOut);  
        out.writeObject(instancia);  
        out.close();  
    } catch (IOException e) {  
        e.printStackTrace();  
        return "nepodarilo sa najst subor";  
    }  
    return "Knihkupectvo sa ulozilo do /res/knihkupectvo_oop.ser";  
}  
  
/**  
 * metoda nacita informacie o knihkupectve do programu  
 * @param path cesta k suboru odkial sa maju data o knihkupectve nacitat  
 */  
public static String deserialize(String path){  
    try {  
        FileInputStream fileIn = new FileInputStream(path);  
        ObjectInputStream in = new ObjectInputStream(fileIn);  
        instancia = (Knihkupectvo) in.readObject();  
        in.close();  
        fileIn.close();  
        return "podarilo sa nacitat knihkupectvo";  
    } catch (IOException e) {  
        return "Nenasiel sa subor '"+path+"' -- vytvara sa nove knihkupectvo";  
    } catch (ClassNotFoundException e) {  
    }  
    getInstance();  
    return "nepodarilo sa nacitat knihkupectvo treba vytvara sa nove";  
}
```

Serializujú sa triedy:

```
public class Knihkupectvo implements java.io.Serializable, Odoberatel, Observer {  
  
public abstract class Miestnost implements java.io.Serializable{  
  
public class Regal implements java.io.Serializable{  
  
public class Sekcia implements java.io.Serializable{  
  
public interface InfoKniha extends Serializable {
```

Volanie funkcie na de/serializáciu:

Keď sa inicializuje objekt Model tak sa deserializuje kníhkupectvo zo súboru. Serializuje sa vtedy, keď hlavné okno dostane požiadavku na vypnutie okna.

```
Model() { deserialize(); }  
  
public String serialize(){ return Knihkupectvo.serialize( path: "./res/knihkupectvo_oop.ser"); }  
  
public void deserialize() {  
    Knihkupectvo.deserialize( path: "./res/knihkupectvo_oop.ser");  
    Knihkupectvo.getInstance().setObserver(this);  
}  
  
public void start(Stage stage) {  
    Model model = new Model();  
    MainController controller = new MainController(model);  
    View view = new View(controller);  
    controller.setView(view);  
  
    enabled = true;  
  
    stage.setOnCloseRequest(e -> {  
        controller.notify( caller: null, out: "serialize");  
    });  
  
    stage.setScene(view.getMainScene());  
    stage.sizeToScene();  
    stage.setResizable(false);  
    stage.show();  
}
```

## 4. Verzie

**Commit - 6a16c955b1a998761afc408d17bf35fd213d173b** – „rozdelenie controllera do viacerých častí“

- Rozdelenie controllerov do viacerých tried
  - MainController – spracovanie vstupu od používateľa.

- ViewController – spracovanie požiadaviek na úpravu view (výpis/otvorenie ďalšieho okna...)
  - ModelController – posielanie požiadaviek modelu
  - ButtonController – súčasť ModelControllera – spracovanie požiadaviek z tlačidiel.
- Prepojenie controllerov pomocou modelu observer.

**Commit - 8272aa2ea6d19d04c5caf858b657488b79711e8c** – „Vytvorenie tabuliek v GUI, prepojenie niektorých tried pomocou observer“

- Vytvorenie GUI tabuliek pre katalóg kníh v kníhkupectve, pre odberateľov, pre texty prijaté na vydanie
- Vytvorenie tried na premieňanie String outputov na dáta vyložiteľné do tabuľky
- Vytvorenie interface Observer (implementujú Autor, Vydavateľstvo, Kníhkupectvo a triedy dedené rozhraním Miestnosť)
- Prerobenie ako Controller posiela dáta View (pomocou Observer)
- Vytvorenie vlastných grafických elementov implementujúcich Observer.

**Commit - ab54cc26c8f8a19a047d17de55b63fc65e33b975** – “podokna presunuté z View do vlastných tried; vytvorenie tried BalikKnih, tried v users.info a vytvorenie viacerých stánkov”.

- V tomto commit-e sme presunuli podokná z View do vlastných tried a ich vytváranie spracováva Controller.
- Ďalej sme pridali triedu BalikKnih a RadKnih, ktoré nahrádzajú potrebu mať v triede, kde sa kumulujú knihy mať 2 polia na uloženie Knihy a jej počtu (v Distributor.java).
- Ďalej sme vytvorili viaceré druhy stánkov a aj možnosť ich vytvárať pomocou GUI. S novými stánkami upravené funkcie DajOdberateľom().
- Vytvorenie triedy UdajeOZamestnancovi pre uchovanie údajov o zamestnancoch (plat a odrobený čas)
- Vytvorenie triedy users.info.Inventar, ktorá slúži na uchovanie knihy u zamestnanca kníhkupectva
- Vytvorenie triedy stanky.Inventar na uloženie všetkých kníh, ktoré má stánok.

**Commit - 30d0cce22bc7d08e58d3fbb43f7fe17381f749d3** – „kníhkupectvo gui - v1.0“

- Dokončenie prvej verzie GUI – podporované iba kníhkupectvo

**Commit - 2376a809d83267c261df1c929bf364c0e49f2687** – „vytvorenie vydávania“

- Nastavenie getterov a setterov v triede Text.java



- Úprava Kníhkupectva aby dokázal prijať knihy od Vydavateľstva
- Implementované metódy zamestnancov Vydavateľstva a implementácia Vydavateľstva a Tlačiarne
- Použitie návrhového vzoru visitor pre Autor.
- Vytvorenie nite pre autorovo písanie v Autor.java

**Commit - 92d987fc3f86b5866c0ee725318bf8edebf522a7 – „gui try”**

- Úprava triedy Kniha.java na návrhový model Composite. Rozdelenie do tried Kniha, Text a Obálka.
- Úprava funkcií na načítanie kníh zo súboru, aby používali novú úpravu.
- Pridanie triedy Organizovaná sekcia (použitá v predajni)
- Vytvorenie súborov tried pre vydavateľstvo
- Pridanie atribútov triede kniha

**Commit - 1cfd4a4f8210c2c375dfc52e72b6c91117de898b – „predajňa dokončená”**

- Dokončenie funkcií predajcu a zákazníka
- Vstup zákazníka do predajne
- Funkcie na nájdenie referencie na knihy v kníhkupectve.

**Commit - b9d82d1537f8754e32b7346ffd769d743c3e5b22 – „ui overhaul”**

- Spustenie funkcií používateľov pomocou lambda funkcií.
- Implementácia niektorých funkcií predajcu

**Commit - b9d82d1537f8754e32b7346ffd769d743c3e5b22 – „serialization”**

- Implementácia serializácie, ukladanie dát kníhkupectva.

**Commit - 28362d3be8e9a6641e4257419f574e2214ef5277 – „UI for skladník”**

- Vytvorenie druhej verzie prijímania vstupu od používateľa
- Vstup mohli byť funkcie skladníka, zamestnanca alebo používateľa

**Commit - 8f93454d7817df98b00747359f5421e019e9f382 – „viacnásobne dedenie”**

- Vytvorenie abstraktnej triedy Zamestnanec pre lepšie rozoznanie Zamestnancov od Zákazníkov
- Umiestňovanie kníh do regálov v sklade.

**Commit - 917c70f42e6126a012a89fd2a8040503cddb6d8 – „ja nechápem”**

- Prvá verzia prijímania vstupu z konzole

- Implementácia niektorých funkcií skladníka
- Jednoduché implementácie funkcií Predajcu a abstraktnej triedy Pouzivatel
- Implementácia ďalších funkcií skladu.

**Commit - d44447e21aa4736dcf34876c5cb5dc85ec50ebec** – „*zoop projekt*”

- Prvá verzia aplikácie
- Vytvorenie kníhkupectva a užívateľov aplikácie: (Skladník, Predajca, Zákazník)
- Vytvorenie regálov, sekcií a skladu
- Predajňa nie ešte dokončená
- Objednávanie tovaru iba zo súboru