Zadanie 3: Použité OOP Princípy

Ondrej Bodnár, B-INFO4

1. Dedenie

V projekte som využil princípy dedenia napríklad pri rozšírení triedy Marketplace triedou HappyHourMarket. Vďaka tomuto rozšíreniu hráč získa náhodný bonus pri predávaní svojich resources. V tomto prípade som využil aj override pôvodnej metódy v parent triede Marketplace.

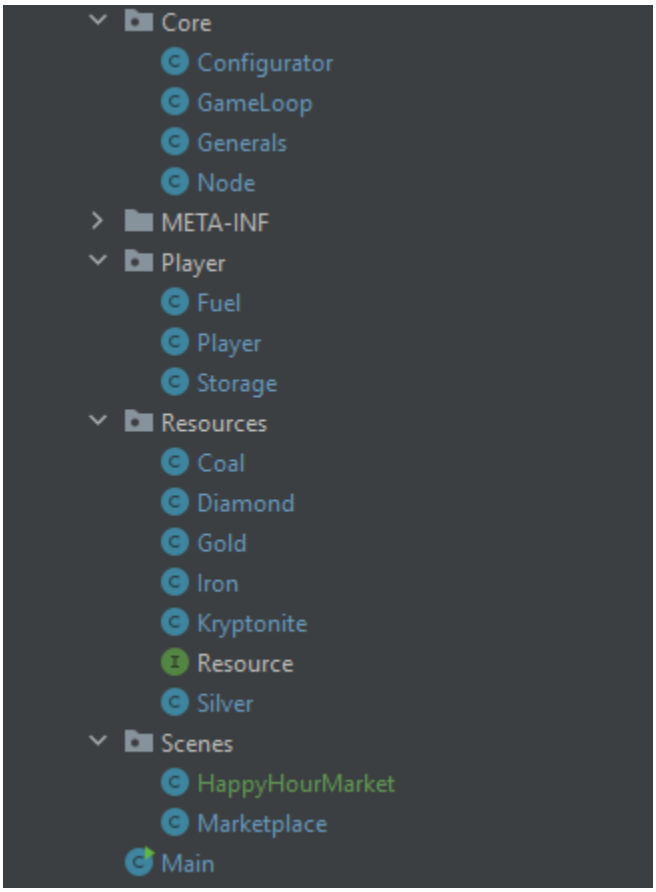
```
3 usages
public class HappyHourMarket extends Marketplace{

    2 usages
    @Override
    public int sellResources(Storage s, Player p) {
        int profits = 0;
        for (Resource r : s.getContent()) {
            profits += r.getProfitPerOne();
        }
        s.getContent().clear();
        p.increaseMoney(calculateBonus(profits));

        return calculateBonus(profits);
    }

    2 usages
    private int calculateBonus(int profits) { return (int) (profits + Math.round((profits/100.0) * 50.0)); }
}
```

2. Zapuzdrenie

Princíp zapuzdrenia som využil v celom kóde programu. Napríklad pri triede Player, kde atribút money predstavuje senzitívnu informáciu, ktorú som chcel zabezpečiť princípom zapuzdrenia. Využitím getter a setter metód som zabezpečil, aby nastavovanie peňazí daného hráča bolo možné iba v danej triede (s pomocou private modifikátora metódy setMoney()). Následne môzme využívať getter metódu kdekoľvek v programe, ktorá nám vráti množstvo peňazí, ktorými daný hráč disponuje.

3. Organizácia kódu

Program je organizovaný do viacerých balíkov následovne:



4. Overloading (preťaženie) metód

Preťaženie metód som využil napríklad pri odoberaní paliva hráčovi. Postupom hry má hráč 5% šancu na získanie “zľavy” paliva. To znamená, že následujúci layer(podlažie) mu nebude odrátaná spotreba paliva (10litrov).

5. Overriding (prekonanie) metód

Prekonávanie metód som využíval vo viacerých prípadoch. Každá trieda má prekonané metódy toString(), equals() a hashCode(). Prekonávanie som využil tiež pri nastavovaní behu programu (TimerTask), kde bolo nutné implementovať vlastný TimerTask s prekonaním default metódy run(). Okrem toho som využil prekonávanie pri vytváraní tried s implementáciou rozhrania Resource.

```
2 usages  FreeDayZ
@Override
public int getDropChance() { return dropChance; }

1 usage  FreeDayZ
public void setDropChance(int dropChance) { this.dropChance = dropChance; }

 FreeDayZ
@Override
public int getProfitPerOne() {
    return (int) Math.round((ThreadLocalRandom.current().nextDouble(
            getMinPrice(),
             bound: getMaxPrice() + 1.0)));
}
```

```
@Override
public String toString() {
    return "Player{" +
            "money=" + money +
            ", moneyLimit=" + moneyLimit +
            ", layerScore=" + layerScore +
            '}';
}

new *
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Player player = (Player) o;
    return money == player.money && layerScore == player.layerScore;
}

new *
@Override
public int hashCode() {
    return Objects.hash(money, moneyLimit, layerScore);
}
```

6. Agregácia

Agregáciu som využil v hlavnom cykle programu. Zabezpečíme vzťah medzi triedou GameLoop a triedami Timer,Fuel,Player,Storage pomocou atribútov a enkapsulácie. Inštancie daných tried sú vytvorené v triede GameLoop iba raz.

```
5 usages   FreeDayZ *
public class GameLoop {
    2 usages
    public Timer t;
    2 usages
    public Fuel f;
    2 usages
    public Player p;

    2 usages
    public Storage s;
```

7. Kompozícia

Princíp kompozície bol využitý najmä v hlavnom cykle programu. Pomocou getter a setter metód som si vytvoril inštancie, ktoré sú vytvárané jedine v tomto konštruktore, nikde inde v programe. V programe k ním pristupujem pomocou getter metódy.

```
3 usages   new *
public GameLoop(int fuelAmount, int moneyAmount, int storageAmount) {
    setF(new Fuel(fuelAmount));
    setP(new Player(moneyAmount));
    setS(new Storage(storageAmount));
}
```

10. Asociácia

Vzťah pomocou asociácie je využitý napríklad v triede Storage, kde som použil ArrayList Resource rozhraní, vďaka čomu dokážem do atribútu content vložiť rôzne objekty implementujúce Resource rozhranie a to nám slúži ako “inventár” vyťažených resources.

```
13 usages  FreeDayZ *
public class Storage {

    6 usages
    private int storage;

    8 usages
    public ArrayList<Resource> content;

    1 usage  FreeDayZ
    public Storage(int storage) {
        setStorage(storage);
        setContent(new ArrayList<>());
    }
```

11. Finálny atribút, finálna metóda

Pomocou finálneho atribútu som zabezpečil to, aby nebolo možné zmeniť obsah tohto atribútu, pomocou ktorého je kód bezpečnejší. Taktiež som využil finálne metódy, ktoré po nastavení final nemôžu byť neskôr prekonané.

Príklad finálneho atribútu moneyLimit

```
11 usages  FreeDayZ *
public class Player {
    6 usages
    protected int money;
    3 usages
    private final int moneyLimit = -300;
```

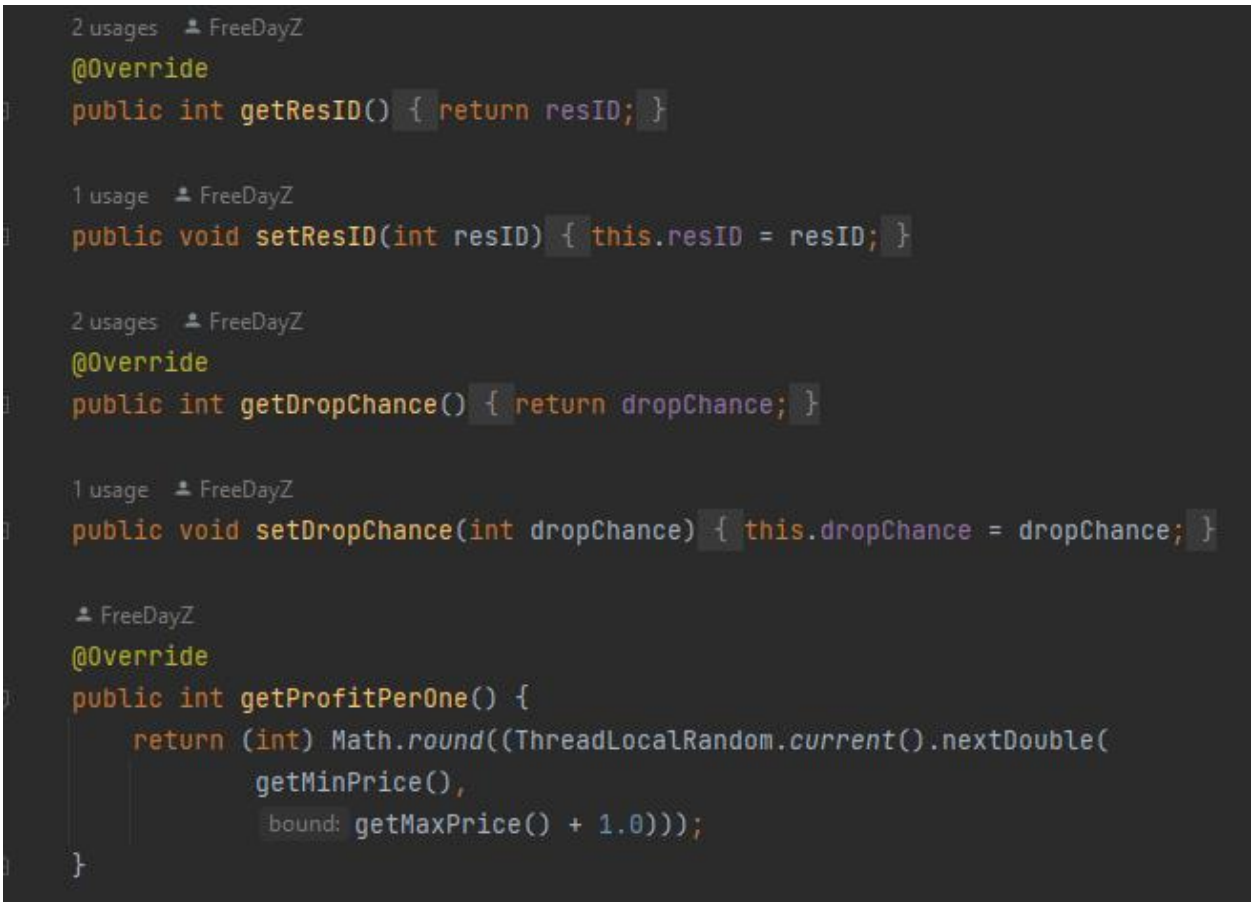
Príklad finálnych metód

```
1 usage  FreeDayZ *
public final void decreaseMoney(int price) { setMoney(getMoney() - price); }

2 usages  FreeDayZ *
public final void increaseMoney(int price) { setMoney(getMoney() + price); }
```

12. Rozhranie, abstraktná trieda

V programe som využil rozhranie Resource. Toto rozhranie uľahčuje vytváranie nových resources, ktoré môže hráč počas hry objaviť. Triedy, ktoré implementujú dané rozhranie majú prekonané metódy v danom rozhraní.

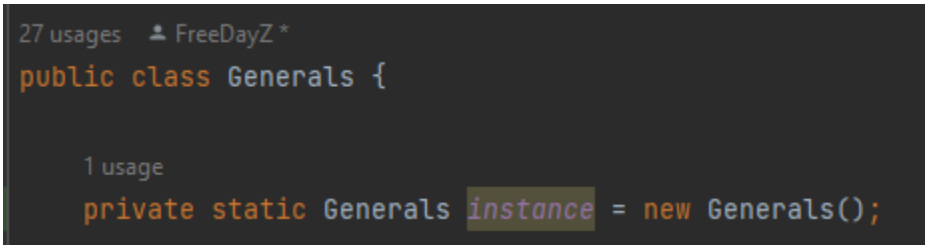
```
FreeDayZ
public interface Resource {

    2 usages  6 implementations  FreeDayZ
    int getDropChance();

    2 usages  6 implementations  FreeDayZ
    int getResID();

    6 implementations  FreeDayZ
    String getName();

    6 implementations  FreeDayZ
    int getProfitPerOne();

    6 usages  6 implementations  FreeDayZ
    double getMinPrice();

    6 usages  6 implementations  FreeDayZ
    double getMaxPrice();
}
```



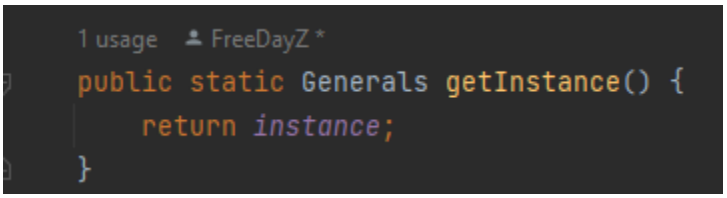
Ukážka triedy Coal, ktorá implementuje rozhranie Resource

13. Statická metóda, statický atribút, Singleton

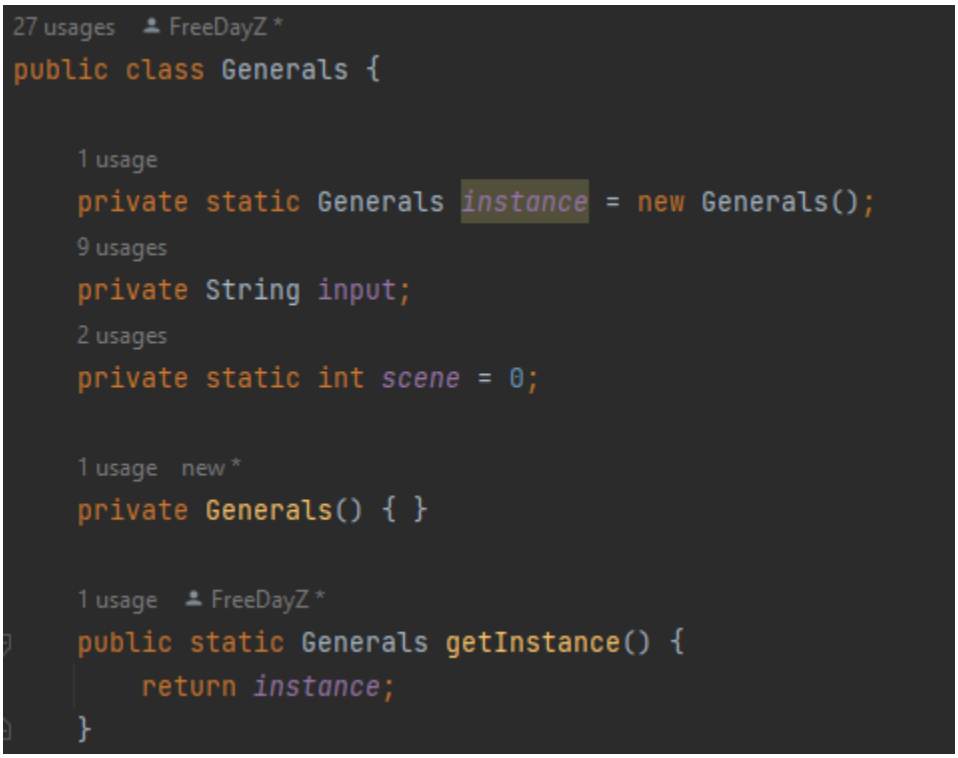
V programe som využil Singleton návrhový vzor pri triede Generals. Dosiahol som ho pomocou statického atribútu inštancie triedy Generals a statickej metódy getInstance(). Okrem toho je využitý privátny konštruktor. Pri vytváraní vo funkcii Main dokážeme získať inštanciu danej triedy bez jej vytvárania. Inštancia je vytvorená už v triede a my k nej pristupujeme pomocou statickej getInstance() metódy.



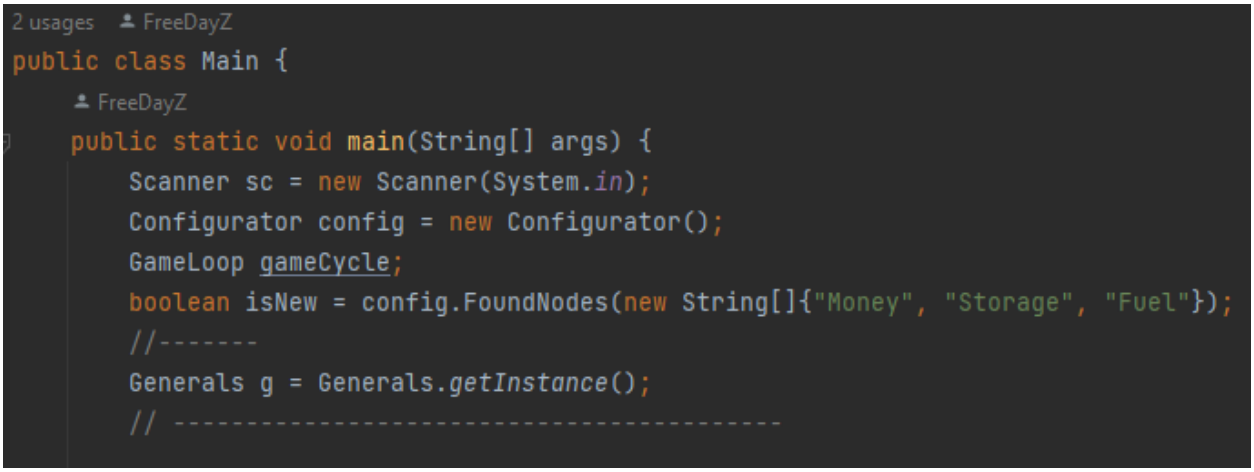
Statický atribút instance



Statická metóda getInstance()

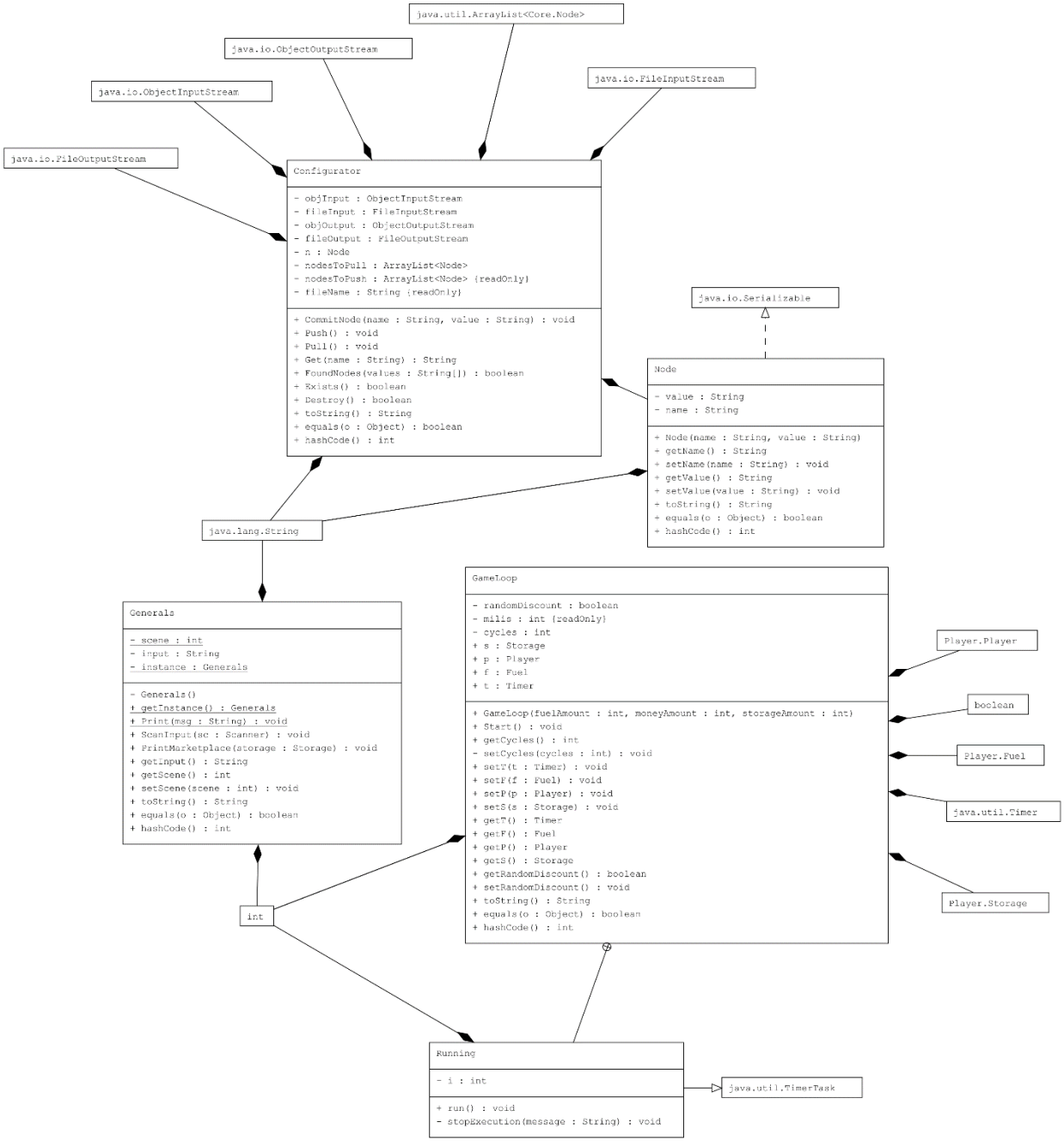


Využitie návrhového vzoru Singleton v triede Generals

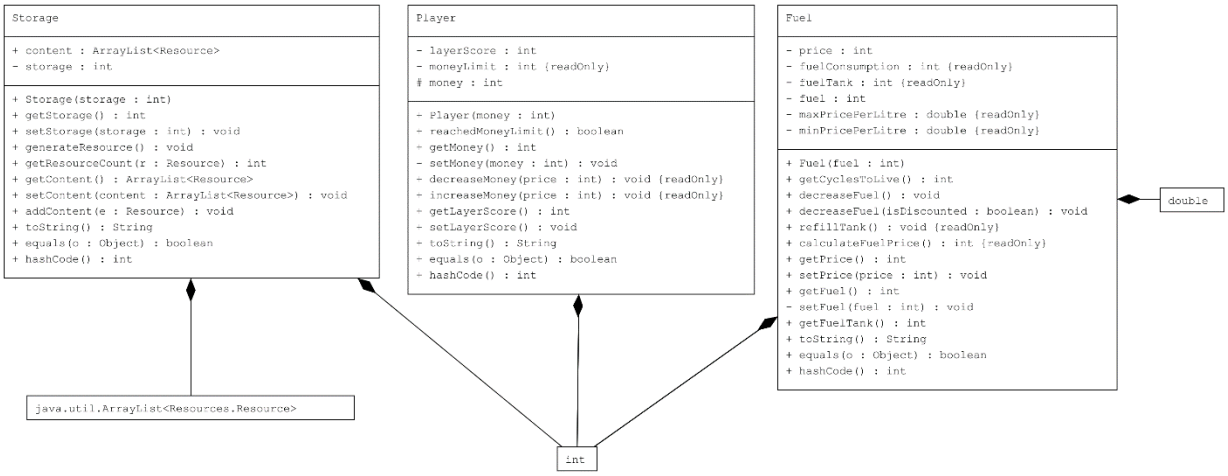


Ukážka získania inštancie pomocou návrhového vzoru Singleton v Main.java

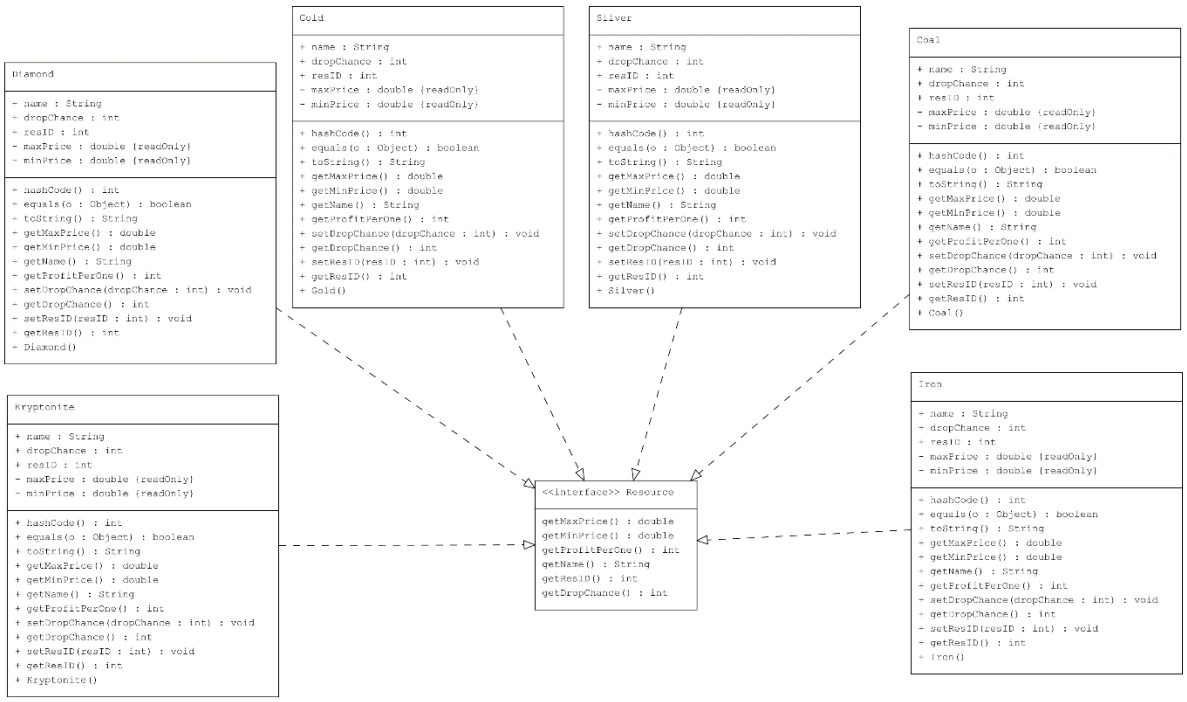
UML Diagramy tried programu



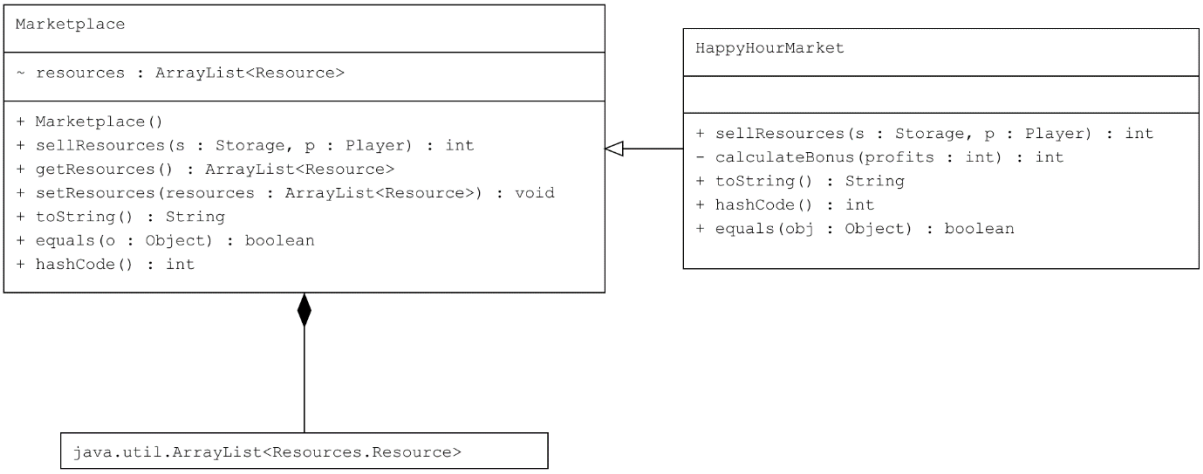
Package Core



Package Player



Package Resources



Package Scenes