B-OOP: Úloha č. 7

Pozorne si najprv prečítajte celé zadanie!

Budeme pokračovať v práci na informačnom systéme pre dopravný inšpektorát z minulého zadania. Na inšpektoráte sú síce spokojní s Vašou prácou, no požadujú určité vylepšenia vášho produktu.

Niektorí zamestnanci inšpektorátu nevytvárajú objekty korektne. Napríklad, stalo sa, že jeden zamestnanec vytvoril objekt typu **Car** a premennú **owner** nastavil na hodnotu **null**. Inštancie objektov nesmie byť možné vytvoriť, ak ich zodpovedajúce konštruktory dostanú nezmyselné vstupy!

Vo všetkých triedach projektu prepíšte konštruktory tak, aby v prípade nezmyselného vstupu vyhodili vhodnú výnimku. Príkladmi nezmyselných vstupov sú:

- vlastník vozidla je **null**;
- kapacita vozidla je záporná alebo nulová;
- atď.

Vo funkcii **main**, kde vytvárate inštancie vozidiel a databázu, zabaľte všetky metódy (a konštruktory), ktoré môžu spôsobiť výnimku, do try-catch bloku.

V triede **Database** pridajte getter metódu **getVehicle** s parametrom **int i** a návratovou hodnotou typu **Vehicle**, ktorá vráti i-te registrované vozidlo, ak také existuje. Vykonajte vhodnú validáciu vstupu **i**, napr. **i** nemôže byť negatívne. Vyhoďte vhodnú výnimku/výnimky.

V metóde **main** pridajte do databázy aspoň 3 vozidlá. Potom pomocou **getVehicle** získajte referenciu na tretie vozidlo. Na tejto referencii nastavte premennú **owner** na hodnotu **null** zodpovedajúcou setter metódou. Potom na databáze zavolajte metódu **toString** a vypíšte jej výstup. Čo sa stalo?

Druhý problém Vášho aktuálneho riešenia spočíva v tom, že z databázy je možné vypísať informácie len pre všetky vozidlá. Zamestnanci inšpektorátu by navyše chceli mať možnosť vypísať iba tie vozidlá, ktoré sú typu auto.

Po chvíľke konverzácie s ChatGPT zistíte, že v jazyku Java existuje kľúčové slovo **instanceof**. Potom si však spomeniete, že o tomto kľúčovom slove ste už počuli na hodinách OOP. Tam Vám povedali, že síce ide o slovo mocné, no zároveň aj veľmi nebezpečné. Ide o slovo s takými negatívnymi vlastnosťami, že by ste ho v tomto prípade zrejme nemali použiť. Skúsite teda tento problém vyriešiť inak.

V triede **Database** vytvorte:

- privátnu metódu **toStringVehicle** s parametrom **Car car**. Zavolá metódu **toString** na parametri **car** a vráti výsledný **String**;
- privátnu metódu **toStringVehicle** s parametrom **Vehicle vehicle**. Metóda vráti prázdny **String**;
- verejnú metódu **toStringOnlyCars** bez parametrov s návratovou hodnotou typu **String**. Táto metóda pomocou cyklu preiteruje pole **registeredVehicles** a zavolá metódu **toStringVehicle** s jednotlivými vozidlami. Metóda vráti zreťazenie výstupov všetkých volaní metódy **toStringVehicle**.

V metóde **main** vypíšte výstup metódy **toStringOnlyCars**. Dostali ste skutočne výpis, ktorý obsahuje informácie o všetkých registrovaných autách? Zamyslite sa, čo sa udialo.

Evidentne je potrebné iné riešenie tohto problému. Do triedy **Vehicle** pridajte verejnú metódu **accept** s návratovou hodnotou **String** a parametrom **Database db**. Metóda nech zavolá metódu **toStringVehicle** so vstupom **this** a vráti jej výstup. V triede **Car** prepíšte (**@Override**) metódu **accept**. Jej telo môže byť identické, ako telo tejto metódy v triede **Vehicle**.

Upravte metódu **toStringOnlyCars** tak, aby namiesto metódy **toStringVehicle** volala na každej inštancii vozidla v databáze metódu **accept** s parametrom **this**. Opäť nech vráti zreťazenie výstupov volaní. Bude výpis vo funkcii **main** teraz fungovať podľa očakávania? Skúste sa zamyslieť nad tým, prečo tomu tak je. Čo by stalo, keby ste odstránili prepísanú metódu **accept** z triedy **Car** a ponechali iba metódu **accept** v triede **Vehicle**?

Komunikácia medzi triedami, ktorú ste práve implementovali, je veľmi podobná návrhovému vzoru **Visitor** (no líši sa v niekoľkých bodoch). Zatiaľ nie je potrebné, aby ste tento vzor detailne poznali.

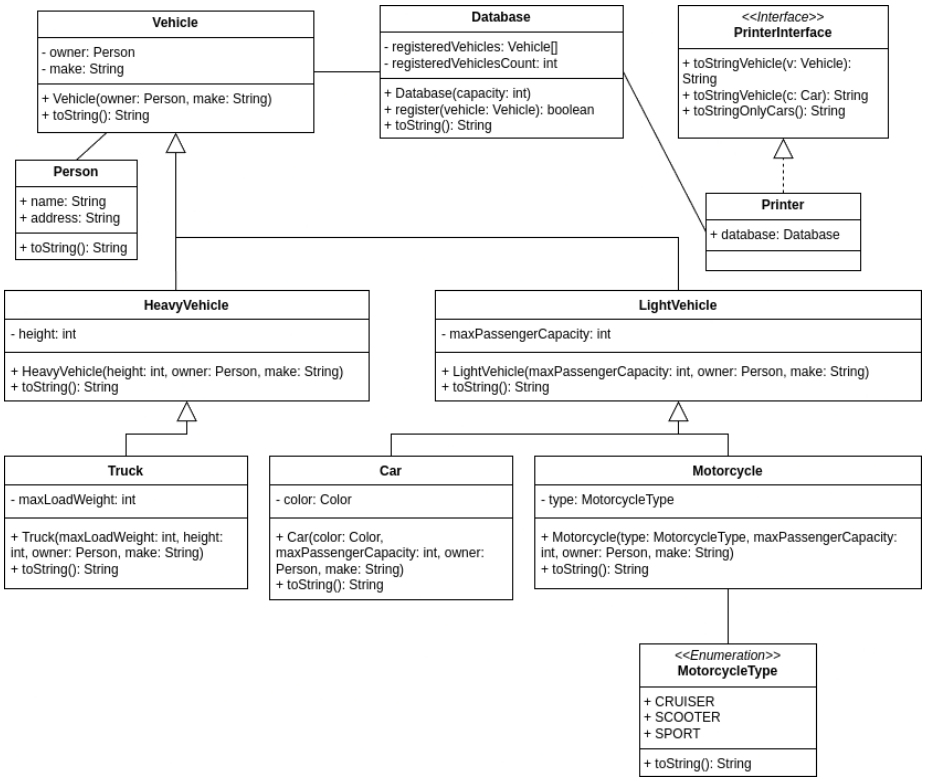
V triede **Database** sa teraz nachádzajú pomocné metódy, ktoré nevyhnutne nesúvisia s jej interným fungovaním. Čo očakávame od databázy? Intuitívne od databázy očakávame iba to, že sa do nej ukladajú dáta. Neočakávame však, že databáza samotná dokáže vytlačiť entity, ktoré sú v nej uložené. To je mimo jej rozsah pôsobnosti. Preto upravíme projekt tak, aby mal o niečo logickejšiu štruktúru.

Vytvorte rozhranie (interface) **PrinterInterface**.[1] V tomto rozhraní vytvorte metódy **toStringVehicle** a **toStringOnlyCars** tak, ako sú aktuálne deklarované v triede **Database**.

[1] Vstavané rozhrania v jazyku Java pomenúvajú rozhrania tak, aby popisovali atribút, ktorý pridávajú triede, ktorá ich implementuje. Z toho dôvodu ich názvy končia na príponu -able, napr. **Serializable** pridáva triede možnosť serializácie. Ide o konvenciu, ktorá má svoje rácio, no nie vždy je ľahké ju dodržať. Iné jazyky pomenúvajú rozhrania odlišne, napr. jazyk C# by hypotetické rozhranie zodpovedné za serializáciu pomenoval **ISerialize**, častá je tiež konvencia v tvare **SerializeInterface**, ktorú používame v tomto zadaní. V zadaniach chceme, aby ste dodržali predpísanú konvenciu pre dané zadanie.

Ďalej vytvorte triedu **Printer**, ktorá implementuje toto rozhranie rovnakým spôsobom, ako je to v triede **Database**. Z nej metódy odstráňte. Trieda **Printer** bude nevyhnutne potrebovať nejakým spôsobom ukladať referenciu na inštanciu **Database**. Odporúčame vytvoriť parametrický konštruktor triedy **Printer** a nastaviť v nej privátnu premennú. Nie je to však jediné správne riešenie. Navyše bude potrebné modifikovať metódy **accept** v triedach vozidiel.

V metóde **main** vytvorte inštanciu **Printer** a vypíšte autá. Po týchto úpravách by Váš projekt mal zodpovedať UML diagramu tried, ktorý vidíte na obrázku 1. Povšimnite si, že v tomto diagrame sa nachádza nový typ šípky medzi triedou **Printer** a rozhraním **PrinterInterface**. Ide o šípku reprezentujúcu fakt, že **Printer** implementuje **PrinterInterface**. Táto šípka je prerušovaná, v praxi sa však častokrát používa rovnaká šípka ako pri dedení. Ide najmä o preferenciu tvorcu diagramu.



Obr. 1: UML diagram popisujúci štruktúru a vzťahy tried zadania

Správna databáza by mala poskytovať určitú mieru perzistencie. Na tento účel využijeme štandardný mechanizmus jazyka Java na reprezentáciu tried vo formáte vhodnom na zápis do súboru. Obe o mechanizmus serializácie. Serializácia funguje tak, že sekvenčne zapíše do jednej reprezentácie všetky premenné

a konštanty danej triedy a navyše aj nevyhnutné metadáta.

V našom prípade chceme serializovať inštanciu databázy, ktorá potom serializuje uložené vozidlá. Za týmto účelom je potrebné implementovať rozhranie **Serializable**. Toto rozhranie musí implementovať aj trieda **Database**, ale aj triedy vozidiel.

Keď ich implementujú, zápis do súboru je jednoducho možné vykonať nasledujúcim spôsobom:

```
// mechanizmus zapisu do suboru
FileOutputStream fileOutputStream
  = new FileOutputStream("subor.txt");
ObjectOutputStream objectOutputStream
  = new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(db); // db je instancia databazy
objectOutputStream.flush();
objectOutputStream.close();
```

Načítanie zo súboru je tiež jednoduché:

```
// mechanizmus citania zo suboru
FileInputStream fileInputStream
  = new FileInputStream("subor.txt");
ObjectInputStream objectInputStream
  = new ObjectInputStream(fileInputStream);
Database db = (Database) objectInputStream.readObject();
objectInputStream.close();
```

Overte, že po uložení a nasledovnom načítaní zo súboru sa databáza vytvorí korektne. Vhodný dodatočný zdroj ku mechanizmu serializácie nájdete na tomto linku.

Do AIS odovzdajte zdrojové súbory (s príponou .java): ZIP súbor priečinku src.