



UML diagram kelas

Sažetak

Ovo poglavlje opisuje elemente dijagrama klasa iz UML 2.0. Objasñjeni su svi glavni elementi od kojih se dijagram klasa sastoji te nabrojani oni koji postoje ali se smatraju izvan opsega studentskih projekata kojima je namijenjen ovaj priručnik. Ovo se poglavlje nužno naslanja na poznavanje principa objektno orijentiranog razvoja. Ako čitač nije upoznat s OO principima, preporuča se da prije čitanja ovog teksta, prouči navedeno poglavlje.

Ključne riječi: UML dijagram klasa, vidljivost, nasljeđivanje, enkapsulacija, polimorfizam, apstraktna klasa, programsko sučelje

1 UML dijagram klasa – koncepti

1.1 Uvod

UML se pojavio sredinom 90-ih godina prošlog stoljeća kao jedan od uspješnijih pokušaja uvođenja standarda i umanjivanja kaotičnosti u kojoj se nalazilo programsko inženjerstvo kao mlado područje. Mogli bismo reći da UML-om opisujemo programski proizvod slično kao što CAD-om opisujemo projekt kuće ili zgrade. UML općenito dijeli vrste nacрта u dva dijela: dijagrami ponašanja i strukturni dijagrami.

Dijagrami ponašanja pokazuju dinamiku različitih dijelova u sustavu odnosno tok rada sustava. Popularni dijagrami ponašanja su dijagram aktivnosti i dijagram slijeda. Dijagram aktivnosti je veću primjenu našao u modeliranju poslovnih procesa, a dijagram slijeda kao jedan od najkorištenijih UML-ovih dijagrama zorno prikazuje interakciju korisnika i različitih dijelova (komponenti) sustava i pri tome uzimajući u obzir vremensku komponentu.

Dijagrami strukture su po prirodi namijenjeni tome da prikažu sliku sustava ne uzimajući u obzir kretanje poruka niti vremensku komponentu u izvršavanju sustava. Bit ove vrste dijagrama je pokazati dijelove sustava (njihovu internu strukturu) i veze koje između njih postoje. Zato se može reći da su po prirodi PI analogija tlocrtima, planovima elektroinstalacija, vodoinstalacija za neku zgradu. U ovom poglavlju govorit će se o jednom i to o najkorištenijem UML dijagramu općenito – dijagramu klasa.

1.2 Opis klase

Prije nego krenemo u detaljno objašnjavanje klase, vrlo je važno naglasiti da UML dijagram klasa ne mora nužno pokazivati samo obične klase. Na prvi mah se ne čini logično, ali postoji objašnjenje. Ono što ovaj dijagram prikazuje se zapravo zovu *klasifikatori* (eng. *Classifiers*) jer se može raditi o:

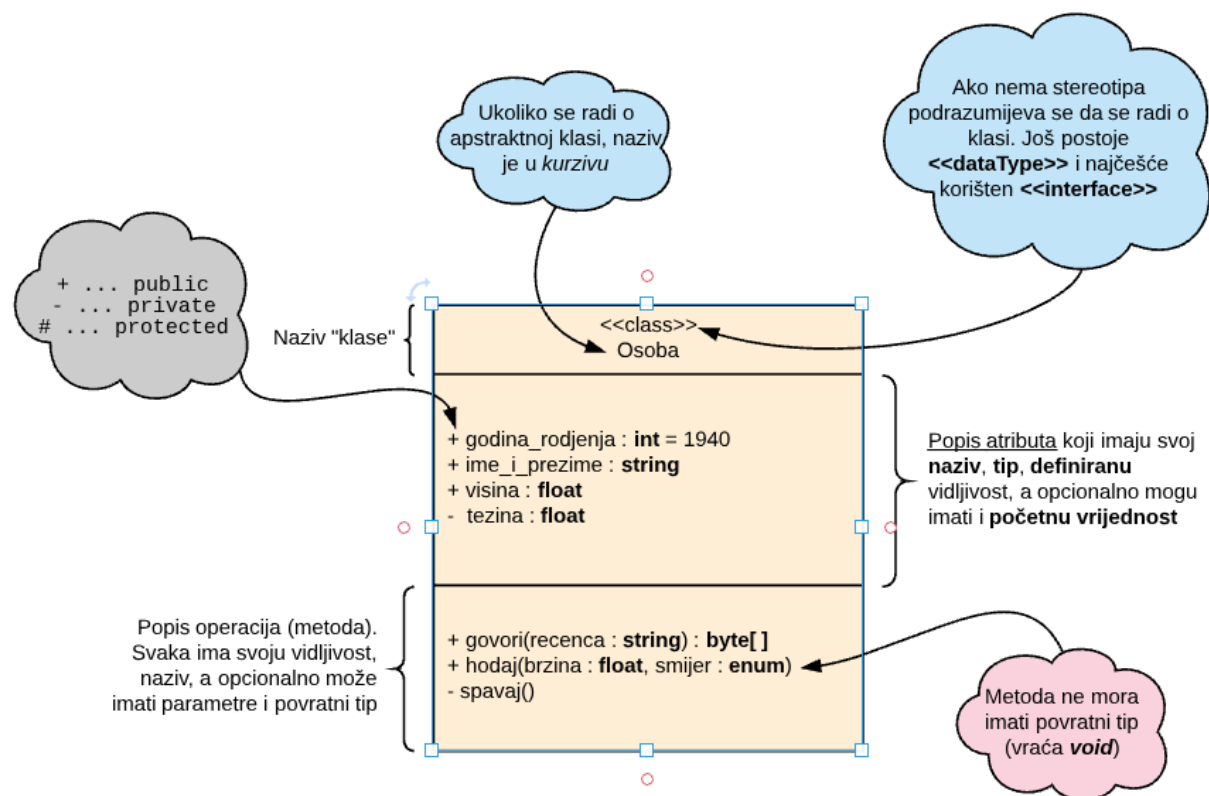
- Sučelju (eng. *Interface*)
- Tipu podatka (eng. *Data type*)
- Klasi (eng. *Class*)
- Paketu (eng. *Package*)
- Objektu (eng. *Object*)
- Enumeraciji (eng. *Enumeration*)
- Artefaktu (eng. *Artifact*)

Najčešće se dakako prikazuju klase (apstraktne i konkretne), a česta su i sučelja nakon kojih slijede manje česti tipovi podataka. Ostale komponente se rjeđe koriste no dobro je znati da

na dijagramu klasa možemo naći i konkretan objekt kao instancu određene klase, enumeraciju i ostale artefakte kao što je dokument, baza podataka, izvršne datoteke, komponente i sl. Sučelja i klase su poznate iz prijašnjih poglavlja o objektno-orijentiranim principima, a tipovi podataka su zapravo obične klase koje predstavljaju one entitete koje bismo obično spremili u SQL tablicu. Pravilo po kojem znamo da govorimo o tipu podatka a ne o klasi je to da ukoliko dva tipa podataka imaju iste vrijednosti na svim atributima smatramo da se radi o jednom te istom podatku.

Slika 1 prikazuje sve elemente koje klasa u sebi ima. Klasa se u UML-u opisuje kao pravokutnik koji je po okomici podijeljen u tri odjeljka.

U gornjem odjeljku se nalazi naziv klase tj. klasifikatora. U primjeru ispod namjerno je postavljen stereotip `<<class>>` iako se taj stereotip podrazumijeva kada ne piše ništa iznad naziva. Kada se želi prikazati sučelje potrebno je to naznačiti stereotipom `<<interface>>` ili `<<dataType>>` ukoliko se radi o tipu podatka. Apstraktna klasa u pravilu ne treba imati naznačen stereotip, ali pošto se radi o posebnoj vrsti klase, njen je naziv u kurzivu kako bi čitatelj modela znao da se radi o apstraktnoj klasi.



Slika 1: Opis klase u UML 2.0

Srednji odjeljak služi za prikaz atributa klase. Način pisanja atributa je standardiziran i slijedi sljedeće pravilo:

(+, -, #) name: type [= default_value]

Prvi dio opisuje vidljivost atributa: "+" vidljivo svima (eng. *public*), "-" vidljivo samo unutar klase (eng. *private*) i "#" zaštićen atribut vidljiv samo pripadajućoj klasi i njenim potklasama (eng. *protected*). Nakon vidljivosti slijedi naziv atributa i njegov tip podataka, a opcionalno (dio iznad unutar uglatih zagrada) može biti dodijeljena predodređena vrijednost.

Zadnji odjeljak sadrži popis operacija koje klasa ima, a prati format zapisa atributa. Svaka operacija ima svoju vidljivost, može imati parametre određenih tipova i na koncu povratnu vrijednost.

Osim grafičkog prikaza koncepata pomoću standardne UML sintakse, nas posebno zanima i način na koji se svi gore prikazani koncepti preslikavaju u programski kod. Stoga u primjeru programskog koda ispod vidimo doslovnu presliku klase sa slike (Slika 1) u kod (Kôd 1).



Kôd 1: Primjer implementacije klase po specifikaciji iz dijagrama

```
//Vidljivost i naziv klase
public class Osoba
{
    //atributi i svojstva klase
    public int godina_rodjenja = 1940;
    public string ime_i_prezime;
    public float visina;
    private float tezina;

    //privatne i javne metode klase
    public byte[] govori(string recenica)
    {
        return null;
    }

    public void hodaj(float brzina, Smjer smjer) { }

    private void spavaj() { }
}

public enum Smjer
{
    lijevo, desno, naprijed, nazad
}
```





Iz prikazanog koda možete primijetiti da je preslikavanje koncepata sa dijagrama klasa u programski kod vrlo striktno i precizno, te da bi smo mogli reći da se preslika vrši u omjeru 1:1. Dakle, svaki koncept na dijagramu klasa ima točnu presliku (implementaciju) u programskom kodu.

Također, možemo primijetiti da, u ovom primjeru nisu poštivana pravila imenovanja koja se primjenjuju na .Net programske jezike, te je u tom slučaju dopušteno napraviti iznimku u preslici dijagrama klasa u kod. Međutim, ako znamo u kojoj tehnologiji će se implementirati programski proizvod, te ako poznamo pravila imenovanja onda ih možemo primijeniti i već kod izrade samog dijagrama.

Za .Net svijet, poseban problem predstavlja **prikaz svojstava na dijagramu klasa**. UML specifikacija ne nudi direktan odgovor na to, a različiti timovi ovom problemu pristupaju na različite načine. Rješenje koje mi preporučamo je uz pomoć spomenutih stereotipa. Dakle, uz ispravno imenovanje (veliko početno slovo), za svojstva se može dodati i stereotip <<get>>, <<set>> ili <<get, set>>. Ako je slučajno neki od pristupnika privatni, onda se i to može upisati u stereotip <<get, - set>>.

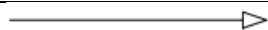
1.3 Asocijacije među klasama

Kako je prije spomenuto, veliki dio onoga što čini dijagram klasa kao jednog od dijagrama strukture je prikazivanje odnosa među klasama. Prema UML superspecifikaciji¹ dijagram klasa može imati sljedeće asocijacije:

	Asocijacija – najopćenitija veza koja označava interakciju između dvije klase. Može biti neusmjerena (linija iznad) ili usmjerena (linija ispod). U prvom slučaju obje klase su osvještene jedna o drugoj, a u drugom samo jedna klasa zna da postoji druga.
	Agregacija – tip asocijacije koji daje dodatnu informaciju o odnosu klase i njihovim životnim ciklusima. Ako postoji agregacija od klase A prema klasi B, objekt klase A nije uvjetovan životnim ciklusom klase B. Objekt klase B samo sadrži (agregira) objekte klase A.
	Kompozicija – tip asocijacije, sličan kao i agregacija. U slučaju kompozicije, objekt klase A sastavni je dio klase B. Prestankom životnog ciklusa objekta B, prestaje i životni ciklus klase A. Slično tome, nastankom objekta klase B, nastaje i objekt klase A.
	Zavisnost – je tip asocijacije koja označava da određen skup elemenata modela za svoju implementaciju ili specifikaciju treba neki drugi skup elemenata modela. Zavisnost definira vezu

¹ OMG - Unified Modeling Language: <https://www.omg.org/spec/UML/2.4.1/About-UML/>, pristupano ožujak 2020.

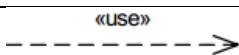
poslužitelj/klijent, pri kojoj promjene u poslužitelju mogu utjecati na klijenta.



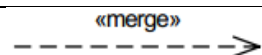
Generalizacija – je poseban tip asocijacije koji prikazuje podrijetlo klasa. Veza generalizacije od klase A prema klasi B, ukazuje da je B generalizacija (općenitija klasa), a A je specijalizacija klase B.



Realizacija – generalizirani slučaj realizacije programskog sučelja. Generički ukazuje na odnos između poslužiteljske klase i klijentske klase kojim poslužiteljska klasa sadrži specifikacije koje klijentska klasa implementira (primijetite da je simbol jednak, no semantika je nešto općenitija).



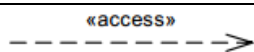
Korištenje – je vrsta veze kod koje jedan element treba drugi element modela kako bi njegova implementacija bila potpuna ili upotreba omogućena.



Spajanje paketa – prikazuje na koji način se sadržaj jednog paketa proširuje sadržajem drugog paketa. Ovaj mehanizam podrazumijeva skup transformacija kojima se sadržaj jednog paketa spaja sa sadržajem drugog paketa. U slučajevima gdje elementi dva paketa prikazuju isti entitet, njihovi sadržaji se konceptualno spajaju u jedan element sukladan formalnim pravilima spajanja.



Javni uvoz paketa – je vrsta odnosa između paketa koja omogućuje korištenje elemenata koji su definirani u uvezenom paketu. Uvezeni elementi će biti vidljivi i drugim paketima.



Privatni uvoz paketa – jednak odnos kao i ranije, samo uvezeni elementi neće biti vidljivi drugim paketima.

UML dijagram klasa dopušta određivanje brojnosti veza između klasa (engl. cardinality).

Tablica 1: Brojnosti

0..1	Instanca ne mora postojati ili mora postojati točno jedna.
1	Mora postojati jedna instanca.
0..*	Instance ne mora postojati ili ih može biti više.
1..*	Mora postojati barem jedna ili više instanci.
m..n	Mora postojati toliko broj instanci koji se nalazi u intervalu od m do n.

1.3.1 "Obična" asocijacija

Standardna ili "obična" asocijacija je najopćenitija veza između dvije klase koja daje do znanja da dvije klase imaju neku interakciju. Postoje dvije vrste: usmjerena i neusmjerena. Primjer ispod (Slika 2) stoga čitamo na način da klasa "OverdrawnAccountsReport" zna za klasu "BankAccount", dok obrnuto ne možemo zaključiti. Ovo u kontekstu primjera može značiti da zapravo klasa izvještaja o prekoračenim računima sadrži popis bankovnih računa koji su u prekoračenju, a istovremeno bankovni račun ni ne zna da se nalazi na takvom izvještaju.



Slika 2: Usmjerena asocijacija

Druga vrsta je neusmjerena asocijacija gdje se podrazumijeva da obje klase "znaju" jedna za drugu.

U terminima programskog koda to se može očitovati na način da se u prvoj klasi spominje druga i obrnuto. Međutim, kako se ovdje radi samo o logičkom modelu koji još nije do kraja definiran, onda treba promisliti kako točno u kodu implementirati ovu asocijaciju. Promotrite primjer programskog koda ispod (Kôd 2).



Kôd 2: Primjer implementacije asocijacije

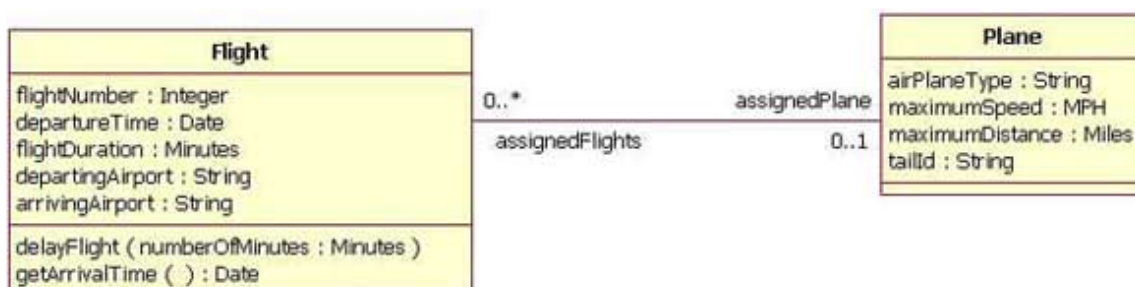
```
class OverdrawnAccountsReport
{
    DateTime generatedOn;
    void refresh()
    {
        List<BankAccount> overdrawnAccounts;
        overdrawnAccounts = Bank.GetOverDrawnAccounts();

        //some code of generating report
        //goes here
        //...
    }
}
```

Dakle, kako smo i zaključili, iz dijagrama smo znali da postoji usmjerena asocijacija, ali način na koji će ona biti implementirana **nije bilo jasno navedeno**. Jedna od bezbroj mogućih implementacija je upravo ona prikazana u odsječku programskog koda prikazanom iznad. U tom odsječku koda se vidi da klasa *OverdrawnAccountsReport* poznaje klasu *BankAccount*, te da, ovisno o situaciji može dobiti pristup do 0, 1 ili više instanci klase *BankAccount*, upravo kako je prikazano na dijagramu (Slika 2). Kako na dijagramu nije bilo navedeno, način na koji će klasa uključiti instance također nije poznat (nije navedeno radi li se o jakoj ili slaboj

agregaciji), ali za pretpostaviti je, sudeći po kontekstu primjera koji promatramo da se radi o slaboj agregaciji, to jest da klasa *OverdrawnAccountsReport* nema instance klase *BankAccount* u vlasništvu i ne kreira ih, već ih samo koristi. O tome će biti detaljnije govora u sljedećem potpoglavlju.

U kontekstu primjera sa slike ispod (Slika 3), klasa "Flight" poznaje klasu "Plane" jer svaki let treba imati dodijeljenog mu najviše jednog aviona. U drugom smjeru, klasa "Plane" poznaje klasu "Flight" i to u množini. To je, naravno, logično jer avion može pamtit i letove na kojima je bio.



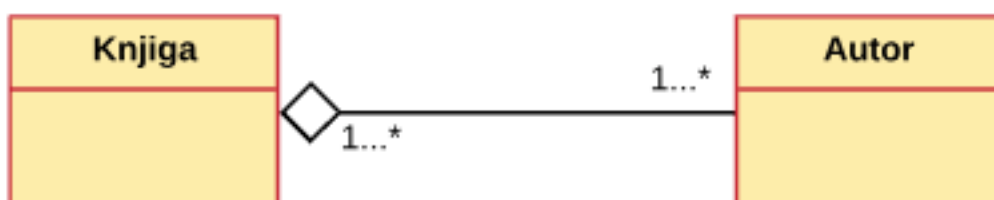
Slika 3: Neusmjerena asocijacija

U primjerima iznad možemo samo pretpostaviti fizičke veze između ovih klasa koje se mogu preslikati u programski kod.

1.3.2 Agregacija i kompozicija

Vezu asocijacije kojoj je jasno definiran tip nazivamo agregacijom ili kompozicijom. Tip asocijacije je tip veze kojim u UML dijagramu opisujemo odnos u kojem su dvije promatrane klase. Postoje dva tipa agregacije: *osnovna agregacija* i *kompozicija*.

Osnovna agregacija (Slika 4) označava se punom linijom i neispunjenim romбом kod klase koja označava cjelinu. Zovemo je još i "slaba" agregacija. Tu vezu tumačimo na način da ako cjelinu obrišemo, dijelovi cjeline ne prestaju postojati.



Slika 4: Primjer slabe agregacije

Kao i u prethodnom primjeru i u ovom primjeru vidimo da puno informacija o klasama *Knjiga* i *Autor* te načinu njihovog međudjelovanja nije navedeno. Ono što sigurno znamo jeste da klasa *Knjiga* **mora** imati minimalno jednog autora, te da također samo dohvaća postojeće instance objekata (agregira ih), ali ih ne kreira. Stoga, jedan mogući način implementacije bi mogao biti kao u primjeru koda ispod (Kôd 3).

**Kôd 3: Primjer implementacije slabe agregacije**

```
class Knjiga
{
    List<Autor> autoriKnjige;

    void DohvatiAutore()
    {
        autoriKnjige = Katalog.DohvatiAutore(this);
    }

    //Ostatak programskog koda
    //ide ovdje
    //...
}
```

Također, iz ovog dijagrama nije jasan niti način na koji se **osigurava obveza postojanja** minimalno jedne instance klase *Autor* za svaku instancu klase *Knjiga*. To bi vjerojatno bili osigurano u nekoj trećoj klasi koja upravlja knjigama i autorima, kao npr. u klasi *Katalog*, ali to možemo samo pretpostaviti.

Kompozicija ili "jaka" agregacija se označava slično, razlika je jedino u tome što je romb na strani klase koja označava cjelinu ispunjen. Značenje veze možemo opisati na način da ako cjelina prestaje postojati, s postojanjem prestaju i dijelovi te cjeline. Primjer ispod je odnos tvrtke (lijevo) i odjela (desno). Ukoliko neka tvrtka koja ima odjele prestane postojati, nema smisla da odjeli nastave živjeti pa se i oni gase.

Još važnije od prestanka postojanja, kompoziciju možemo objasniti u kontekstu nastajanja instanci. Dakle, instanca klase *Department* uvijek nastaje unutar već postojeće instance klase *Company*. Naravno, iz dijagrama prikazanog na slici ispod način nastajanja spomenute instance nije jasno prikazan, ali jedna od implementacija bi mogla biti kao u kodu ispod.



Slika 5: Primjer kompozicije (jake agregacije)

**Kôd 4: Primjer implementacije kompozicije**

```
class Company
{
    public string Name { get; set; }
    public List<Department> Departments { get; set; }

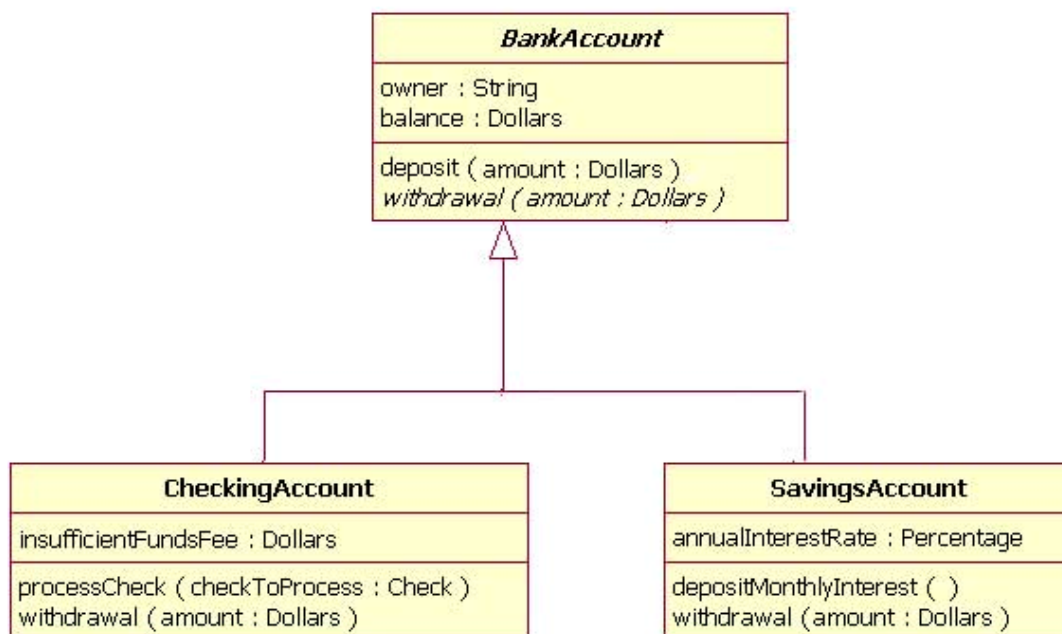
    public Company()
    {
        Departments = new List<Department>();
        Departments.Add(new Department() { Name = "Production" });
        Departments.Add(new Department() { Name = "Sales" });
        Departments.Add(new Department() { Name = "Accounting" });
    }
}
```

U primjeru koda iznad prikazano je kako se pri kreiranju instance klase *Company*, automatski, kroz konstruktor, inicijalizira lista odjela te tvrtke (klasa *Department*). Naravno, u stvarnim primjerima, podaci o odjelima se čitaju iz baze podataka ili ih se unosi usporedo s unosom nove tvrtke u aplikaciju.

1.3.3 Nasljeđivanje (generalizacija)

Generalizacija je jedan od temeljnih principa OO programiranja i stoga ova veza između klasa zauzima vrh ljestvice važnosti, uz vezu realizacije o kojoj će više biti u sljedećem potpoglavlju. Na primjeru ispod (

Slika 6) vidi se apstraktna klasa koja predstavlja općeniti bankovni račun. Tu apstraktnu klasu nasljeđuju dvije potklase, tj. dvije konkretne vrste bankovnog računa: tekući račun ("**CheckingAccount**") i štedni račun ("**SavingsAccount**").



Slika 6: Primjer generalizacije (apstrakcije)

Sve ono što općenito vrijedi za sve vrste bankovnih računa sadržano je u nadklasi "BankAccount". Možemo reći da je ta klasa poopćenje (generalizacija) onoga što se u stvarnosti pojavljuje u više oblika, tj. izvlači (apstrahira -> apstrakcija) zajedničke značajke svih konkretnih pojavljivanja bankovnog računa. Elementi koji su specifični za konkretnu vrstu bankovnog računa (posebne metode i atributi) se samim time nalaze u pojedinoj konkretnoj klasi. Na tekućem računu ne možemo napraviti oročenu štednju s mjesečnom uplatom i kamatom, dok na štednom računu možemo (operacija "**depositMonthlyInterest**").



Kôd 5: Implementacija apstraktne klase

```
abstract class BankAccount
{
    public string owner;
    public float balance;

    public void deposit(Dollars amount)
    {
        //programski kod ide ovdje
    }

    public abstract void withdrawal(Dollars amount);
}
```

Bitno je pojasniti još jedan važan koncept opisan u primjeru iznad. Primijetimo da je naziv natklase pisan u kurzivu, što govori da se radi o **apstraktnoj klasi**. Postavlja se pitanje: zašto nije "obična" klasa umjesto apstraktne? Autor dijagrama je htio reći da ne može postojati instanca općenitog "bankovnog računa" pa je to postigao tako da je klasu "BankAccount" proglasio apstraktnom kako se ne bi mogla instancirati.

Oštrovidan čitatelj će primijetiti da je i naziv jedne metode također u kurzivu. Time je dano do znanja da se radi o apstraktnoj operaciji koju **mora** implementirati svaka potklasa, što i jest slučaj kada gledamo primjer iznad. U programskom kodu primijetite da apstraktna operacija nema tijelo, već samo zaglavlje.



Kôd 6: Implementacija nasljeđivanja i premošćivanja

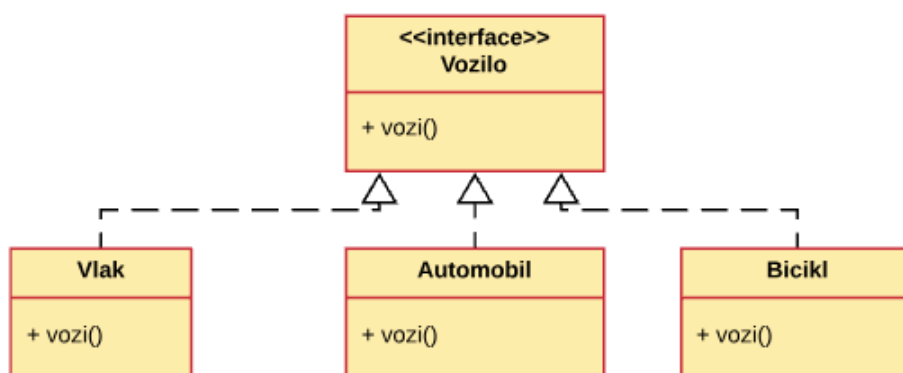
```
class CheckingAccount : BankAccount
{
    public Dollars insufficientFundsFee;

    public void processCheck(Check checkToProcess)
    {
        //ovdje ide programski kod
    }

    public override void withdrawal (Dollars amount)
    {
        //ovdje ide programski kod
    }
}
```

1.3.4 Realizacija

Realizacija je, kako je već spomenuto, jedan od temelja OO programiranja. Ne smije se zamijeniti generalizacijom iz više razloga, a jedan očiti razlog je da se umjesto natklase radi o posebnoj vrsti koja se zove sučelje (engl. *Interface*). Sučelje je posebna vrsta klase koja nema nikakvu implementaciju i ne može se instancirati. Služi isključivo da propiše metode koje moraju biti implementirane u klasama koje žele ispuniti (realizirati) taj propis.



Slika 7: Primjer realizacije (implementacija)

Ovime se omogućava da konkretne klase koje implementiraju sučelje možemo promatrati samo kroz sučelje ne vežući se za konkretnu klasu. Recimo (Slika 7) da prikazujemo listu vozila na Windows formi. Želimo u jednom trenutku pokrenuti sva vozila. Naša forma mora biti vezana samo za sučelje "Vozilo" i jednostavno za svaki element liste vozila pozvati metodu *vozi*. Naravno, s obzirom da svaki objekt iz te liste ima svoj konkretan tip (Vlak, Automobil ili Student), sam će objekt znati na koji način će voziti, tj. kakva će biti implementacija te metode. U ovom primjeru forma nije trebala znati da postoje različite implementacije, već samo da postoji standard operacije bez parametara koja ne vraća vrijednost i da se zove *vozi*.

Ono što razlikuje generalizaciju i realizaciju, između ostalog, je to što u slučaju generalizacije možemo naknadno dodati atribut ili operaciju koju će potom sve potklase naslijediti i pritom postati proširene. Dodavanje operacije ili atributa obavljamo samo na jednom mjestu - u natklasi.

Kod sučelja se zapravo olakšava dodavanje nove klase koja će pratiti isti standard. Kada bismo u program opisan iznad htjeli dodati novu klasu - Brod, ne bismo trebali uopće mijenjati programski kod forme koja pokreće sva vozila jer će i Brod pratiti standard propisan sučeljem.

**Kôd 7: Primjer definiranja sučelja**

```
interface Vozilo
{
    string Naziv { get; set; }
    void Vozi();
}
```

Iz priloženog koda (Kôd 7) se vidi da je implementacija sučelja (ugovora kojeg klasa koja ga realizira vrlo jednostavna. U sučelju se mogu definirati svojstva i metode koje klase moraju realizirati, a atributi se ne mogu definirati kroz sučelja. U svrhu pokaza načina definiranja svojstva, programski kod malo nadopunjuje dijagram dodajući svojstvo Naziv, te primjenjujući imenovanje sukladno .Net jezicima.

**Kôd 8: Primjer realizacije sučelja**

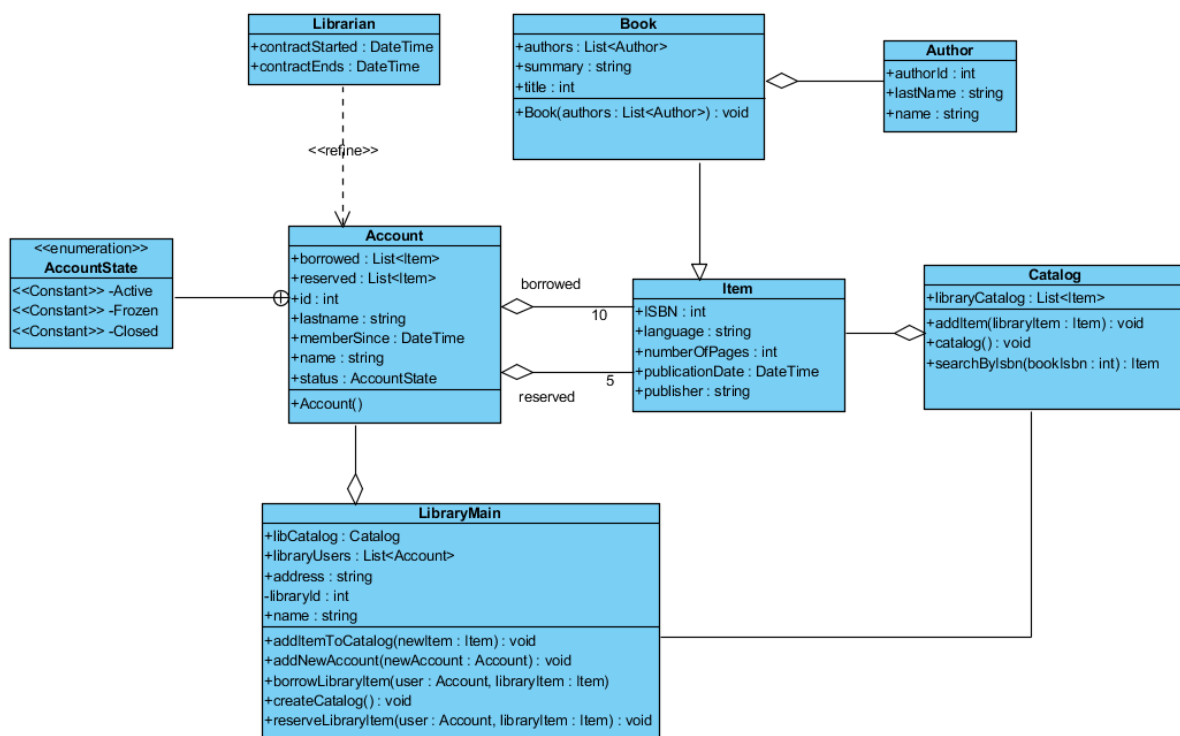
```
class Vlak : Vozilo
{
    public string Naziv {
        get => throw new NotImplementedException();
        set => throw new NotImplementedException();
    }

    public void Vozi()
    {
        //programski kod ide ovdje
    }
}
```

5.1. Pitanja i zadaci

Zadaci

1. Pomoću alata za izradu UML dijagrama, preporučljivo (Visual Paradigm Community Edition²), nacrtajte ovaj dijagram klasa (Slika 8).

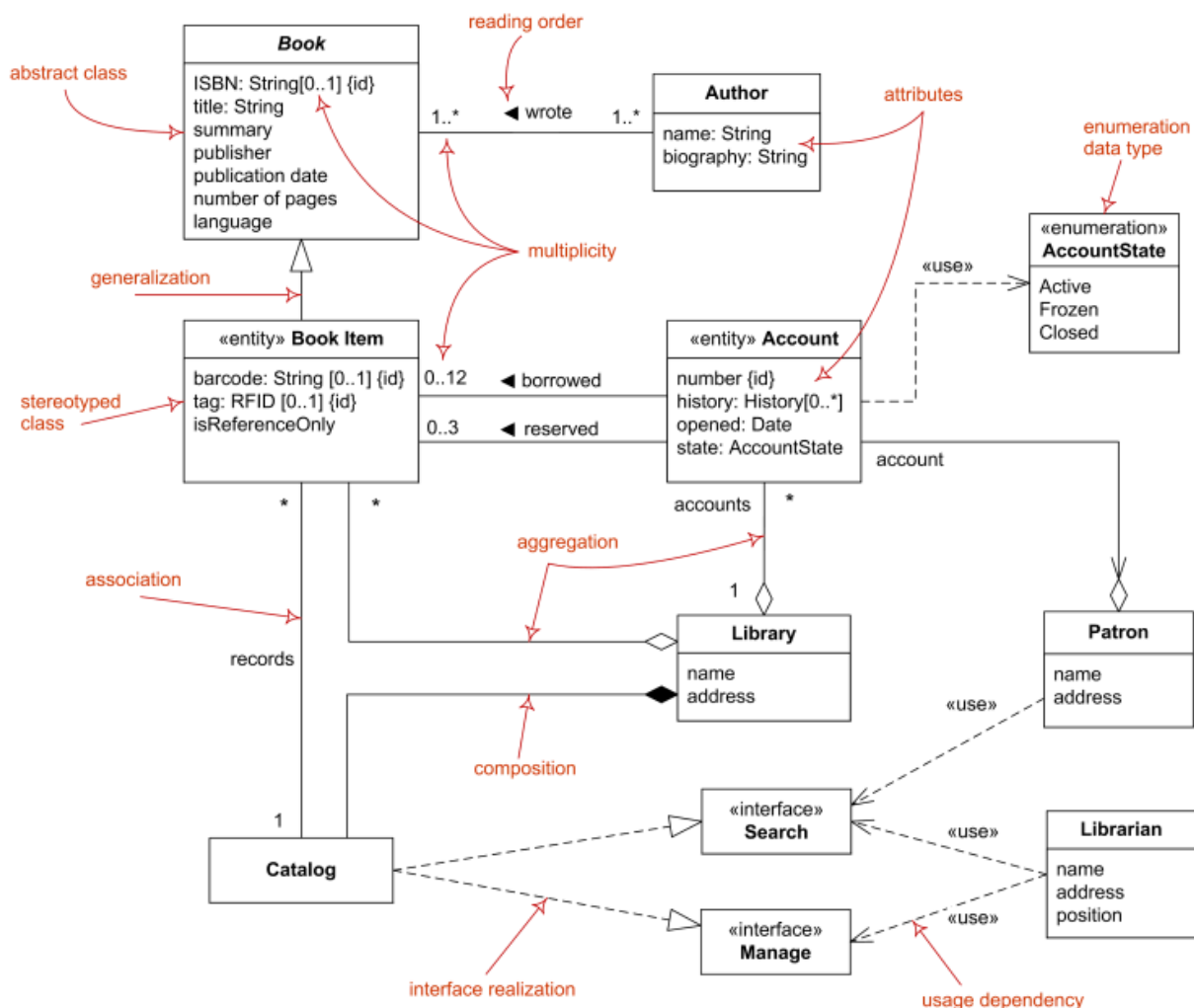


Slika 8: Zadatak za vježbu³

2. Postoje li mjesta gdje vidite da je specifikacija prikazana na slici (Slika 8) nedovršena? Dovršite.
3. Implementirajte dijagram klasa (Slika 8), koristeći asocijacije u Visual Studiju.
4. Napravite dijagram klasa za svoj projekt i promislite na je li na implementacijskoj razini primjenjiv.
5. Napravite aplikaciju na temelju primjera iz poglavlja 1.3.4 (Slika 7)
6. Analizirajte dijagram klasa sa slike ispod (Slika 9). Pokušajte implementirati programski proizvod temeljen na prikazanim klasama te implementirajte korisničko sučelje koje vam omogućuje upravljanje podacima (dodavanje, promjenu, brisanje zapisa koji se odnose na entitete sa slike).

² Besplatno preuzimanje sa: <https://www.visual-paradigm.com/download/community.jsp>

³ Preuzeto i prilagođeno iz <http://www.uml-diagrams.org/class-diagrams-examples.html>



Slika 9: Zadatak za vježbu⁴

- Osmislite vlastiti primjer zadatka po želji, skicirajte dijagram klasa i pokušajte implementirati rješenje striktno po dijagramu koji ste kreirali. Jesu li u implementaciji potrebna odstupanja od dijagrama? Ako jesu, možete li napraviti novi primjer u kojem odstupanja neće biti?
- U Visual Studiju postoji alat za dizajn klasa iz kojih se automatski može generirati programski kod. Instalirajte ga i proučite alat i njegove mogućnosti. Pogledajte temu Working with Class Designer: <https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types> i temu za instalaciju alata dostupnu ovdje: <https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects>.
- Istražite razliku između UML dijagrama klasa i .Net dijagrama klasa (engl. .NET class diagram).

⁴ Preuzeto sa: <https://www.uml-diagrams.org/class-diagrams-overview.html>

Pitanja za razmišljanje i odgovor:

1. Što je klasa?
2. Što je objekt?
3. Što je programsko sučelje?
4. Može li apstraktna klasa implementirati metode?
5. Što je svojstvo klase?
6. Koja je implementacijska razlika kompozicije i agregacije s aspekta stvaranja objekta?
7. Objasnite razliku između protected i private vidljivosti.
8. Koja je razlika kompozicije i agregacije s aspekta brisanja objekata?
9. Može li bilo koja klasa naslijediti bilo koju drugu klasu?

5.2. Više informacija o temi

1. IBM UML Basics - the class diagram (<https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html>), dostupno: ožujak, 2020.
2. UML class and object diagrams overview (<https://www.uml-diagrams.org/class-diagrams-overview.html>) dostupno: ožujak, 2020.
3. UML class diagram tutorial (<https://www.lucidchart.com/pages/uml-class-diagram>) dostupno: ožujak, 2020.
4. Microsoft – Visual Studio Docs – UML Class Diagrams: Reference (<http://msdn.microsoft.com/en-us/library/vstudio/dd409437.aspx>), dostupno: ožujak, 2020.
5. Microsoft – Visual Studio Docs – UML Class Diagrams: Guidelines (<http://msdn.microsoft.com/en-us/library/vstudio/dd409416.aspx>), dostupno: ožujak, 2020.
6. Materijali s predavanja PI 2020: prof. Vjeran Strahonja, doc. Zlatko Stapić.
7. Predavanja prof. Brassa, Svučilište Halle (http://users.informatik.uni-halle.de/~brass/dd04/c6_umlcl.pdf), dostupno: ožujak, 2020.