SI Vježbe 1

Šta je OOA/D?

1. Uvod

Na prethodnim godinama studija, u okviru predmeta Programiranje 2 obrađivani su osnovni objektno orijentisani koncepti u programskom jeziku C++. Nakon toga, u okviru ovog predmeta kratko je prikazan jezik JAVA (jer mu je sintaksa dosta slična C++). Međutim, poznavanje objektno orijentisanog programskog jezika nije dovoljno za razvijanje kvalitetnog softvera. Poznavanje jezika jeste jedan neophodan aspekt u razvoju softvera, ali ne i dovoljan. Programski jezik je alat za razvoj, međutim, posjedovanje alata nije dovoljno da vas učini majstorom bilo kog "zanata", pa tako ni razvoja softvera.

Također, često ljudi imaju (pogrešnu) predstavu da je za razvoj softvera uz pomoć OO tehnologija dovoljno poznavanje UML notacije. Imajte na umu da je UML "samo" standardna notacija za crtanje dijagrama, što zaista jeste korisna stvar, međutim nedovoljna za razvoj softvera. Potpuno je beskorisno poznavati UML ili neki CASE alat, ako ne znate kreirati dobar OO dizajn (ili procijeniti i unaprijediti postojeći). Upravo je to vještina koja je neophodna, ali tom vještinom nije lako ovladati.

Koje klase će postojati u sistemu? Koja zaduženja će imati pojedina klasa? Na koji način će objekti komunicirati? Sve su to ključna pitanja na koja treba dati odgovore prilikom objektno orijentisane analize i dizajna. Provjereni principi i tehnike kojim se dolazi do odgovara na ta pitanja također će biti predstavljeni u obliku softverskih uzoraka (patterns).

Upravo zato ćemo, u okviru nastave, pokušati približiti studentima kako primijeniti UML (kao alat) prilikom OO analize i dizajna. "Responsibility-driven" dizajn će biti prikazan na primjeru sistema za studentsku službu.

Najvažnija "vještina" u OO razvoju je efikasno dodjeljivanje zaduženja softverskim objektima. Biće prikazano 9 fundamentalnih principa za objektno orijentisani dizajn. Ti principi su organizovani u GRASP (General Responsibility Assignement Software Patterns) uzorke (patterns) koji će služiti kao pomoć pri učenju.

Objektno orijentisanom dizajnu prethodi analiza zahtijeva koja najčešće uključuje pisanje scenarija korištenja ("use-case") i crtanje "use-case" dijagrama. Zato ćemo u nastavku, na početku izrade sistema za studentske službe, prvo obrađivati teme vezane za izradu scenarija korištenja ili slučaj upotrebe (use-cases).

2. Analiza i dizajn

Analiza – istraživanje problema i zahtjeva (a ne rješenja). Analiza je širok pojam i obuhvata analizu zahtjeva (istraživanje zahtjeva sistema) i analizu objekata (istraživanje objekata iz domene problema).

Dizajn – Za cilj ima identifikaciju konceptualnog rješenja problema (a ne implementaciju). Na primjer, rezultat je shema baze podataka i softverski objekti (dizajn objekata ili dizajn baze podataka). Naposljetku se dizajn implementira.

Za vrijeme **objektno orijentisane analize** naglasak je na pronalasku i opisivanju objekata (ili koncepata) iz domene problema.

Za vrijeme **objektno orijentisanog dizajna** naglasak je na pronalasku softverskih objekata i načina na koji oni komuniciraju u cilju ispunjavanja zahtjeva.

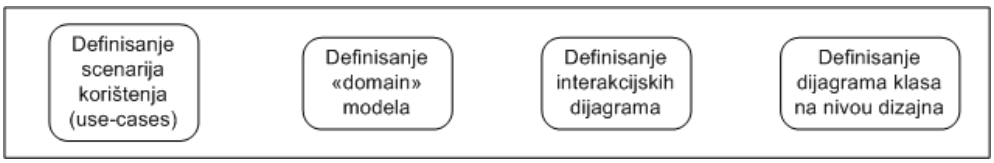
3. Razvoj softvera u iteracijama

Razvoj softvera uključuje mnogo različitih aktivnosti. Postavlja se pitanje kojim redoslijedom krenuti i na koji način. Analiza zahtijeva i OOA/D (Object Oriented Analysis/Design) moraju biti primijenjeni u kontekstu nekog ***razvojnog procesa***. U okviru ovog materijala biće primijenjen ***agilni*** (fleksibilan) pristup UP (Unified Process) razvojnom procesu, kao primjer razvoja softvera u iteracijama.

4. Kratak primjer

Prije detaljne razrade iterativnog procesa za razvoj softvera, biće prikazan sažeti prikaz ključnih koraka i ključnih dijagrama procesa na jednostavnom primjeru jedne igre (bacanje kocke). Igrač baca dvije kocke. Ako je rezultat 7, igrač je pobijedio. U suprotnom, igrač je izgubio.

Cjelokupan proces se može opisati slijedećim dijagramom.



4.1 Definisanje scenarija korištenja

Analiza zahtjeva može sadržati pisanje scenarija (priča) o načinima na koji korisnici mogu koristiti sistem. Zovemo ih ***scenariji korištenja*** (use-cases).

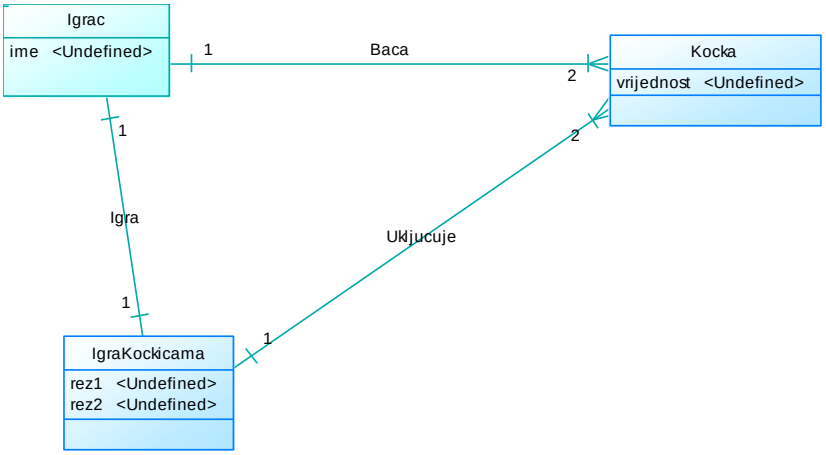
Scenariji korištenja nisu objektno orijentisani koncept. Jednostavno su to napisane priče. Međutim, vrlo su popularne pri analizi zahtijeva. Na primjer "sažeta" verzija scenarija "Igranje igre sa kockicama" mogla bi biti:

- Igranje igre sa kockicama: Igrač izdaje komandu za bacanje kockica. Sistem prikazuje rezultat. Ako je zbir vrijednost na kockicama 7 igrač pobjeđuje, u suprotnom igrač gubi.

4.2 Definisanje "domain" modela

Objektno orijentisana analiza ima za cilj kreiranje opisa problema (domene problema) iz perspektive objekata. To uključuje identifikaciju ključnih koncepata, atributa i veza između objekata. Rezultat se izražava u obliku ***domain modela*** koji prikazuje ***relevantne*** koncepte ili objekte iz domene problema.

Djelomičan "domain" model je prikazan na slijedećoj slici.



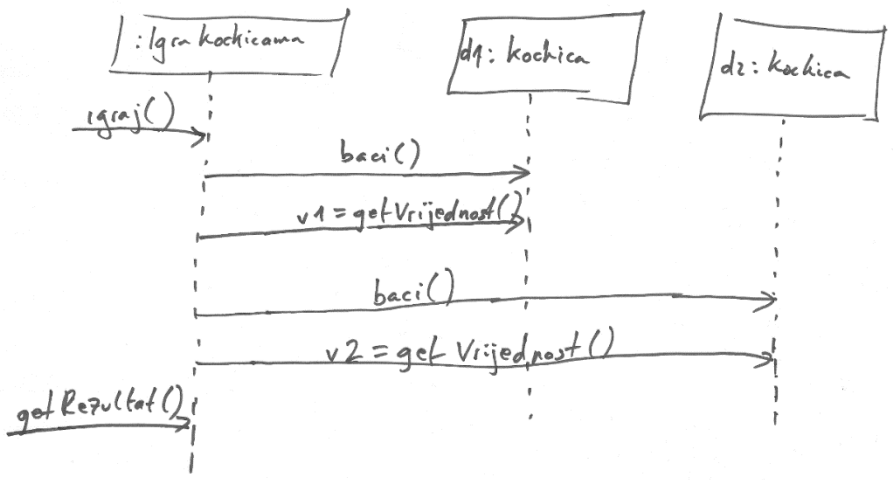
Slika 4.1. Domain model

Dijagram ilustruje relevantne koncepte Igrac, Kockica i IgraKockicama sa njihovim atributima i vezama. Treba napomenuti da domain model ne prikazuje softverske objekte, nego koncepte iz stvarnog svijeta. Zato se često naziva **konceptualni model objekata.**

4.3 Dodjeljivanje odgovornosti i crtanje interakcijskih dijagrama

Objektno orijentisani dizajn se bavi identifikacijom softverskih objekata, njihovih zaduženja i njihove interakcije (saradnje). Uobičajeno se interakcija između objekata predstavlja na sekvencijalnim dijagramima (sequence diagram – jedan tip interakcijskih dijagrama iz UML-a). Dijagram prikazuje tok poruka između objekata (i na osnovu toga pozive metoda).

Na primjer, slijedeći sekvencni dijagram prikazuje OO dizajn softvera. Igra se pokreće slanjem poruke "igraj" objektu klase "IgraKockicama". Prikazani dijagram ilustruje način na koji se obično primjenjuje UML (skiciranjem na tabli).



Slika 4.2. Sistem sekvencijalni dijagram

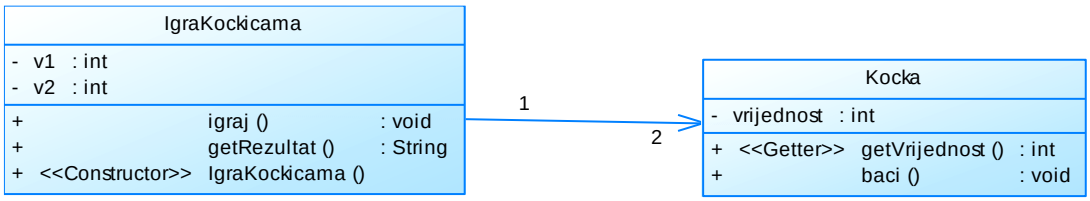
Primijetite da, za razliku od stvarnog svijeta gdje igrač baca kockice, u softverskom dizajnu objekat IgraKockicama vrši bacanje kockica.

Iako se pri dizajnu softvera i softverskih objekata za inspiraciju uzimaju koncepti iz stvarnog svijeta, softverski modeli ne predstavljaju direktne modele ili simulacije stvarnog svijeta.

4.4 Definisanje dijagrama klasa na nivou dizajna

Pored dinamike sistema, koja se prikazuje na interakcijskim dijagramima, korisno je prikazati i statički pogled na sistem i to na dijagramima klasa. Dijagram klasa na nivou dizajna prikazuje softverske klase (za razliku od domain modela), njihove atribute, metode i veze između klasa.

Analizom prethodnog sekvencijalnog dijagrama dolazimo do slijedećeg dijagrama klasa.



Slika 4.3. Dijagram klasa

Pošto se poruka "igraj()" šalje objektu klase "IgraKockicama", onda ta klasa treba imati metodu "igraj()". Također se sa prethodnog dijagrama može zaključiti da klasa "Kockica" treba da ima metode "baci()" i "getVrijednost()".

Primijetite da klase na ovom dijagramu nose ista imena kao da domain modelu. Obično, softverske klase jesu inspirisane konceptima iz stvarnog svijeta (domain modela), ali ne predstavljaju doslovne preslike. Time se pokušava smanjiti razlika između načina na koji ljudi razmišljaju o problemu (objekti) u odnosu na kompjutersku predstavu problema (varijable, funkcije, procedure). Često se za tu razliku u predstavljanju problema koristi termin "***representational gap***" (praznina u predstavljanju problema). Smanjivanjem te "praznine" olakšava se dizajn softvera za rješenje datog problema.

4.5 Implementacija

Na kraju, softverske klase se implementiraju u nekom od objektno orijentisanih programskih jezika. Veze između klasa, zavisno od "navigabilnosti" (o smjeru veza će kasnije biti mnogo više govora), obično se implementiraju kao atributi klase koja ima mogućnost navigacije prema klasi sa kojom je u vezi (strelica na dijagramu). Tako će "kockice" biti atribut klase "IgraKockicama".

4.5.1 Implementacija klase Kocka

Java

```
package ba.fit.kocke.model;

public class Kocka
{
	private int vrijednost;

	public int getVrijednost()
	{
		return vrijednost;
	}

	public void baci()
	{
		double r = Math.random();
		vrijednost = (int) (r * 6) + 1;
	}
}
```

C#

```
using System;

namespace Kocke.Model
{
    public class Kocka
    {
        private int _Vrijednost;

        public int GetVrijednost()
        {
            return _Vrijednost;
        }

        public void baci()
        {
            Random rnd = new Random();
            double r = rnd.NextDouble();
            _Vrijednost = (int)(r * 6) + 1;
        }
    }

}
```

4.5.2 Implementacija klase IgraKockicama

Java

```
package ba.fit.kocke.model;

public class IgraKockicama
{
	private int rez1;
	private int rez2;

	private Kocka[] kocke = new Kocka[2];

	public void igraj()
	{
		kocke[0].baci();
		rez1 = kocke[0].getVrijednost();

		kocke[1].baci();
		rez2 = kocke[1].getVrijednost();
	}

	public String getRezultat()
	{
		int r = rez1 + rez2;
		if (r % 2 == 0)
			return "Zbir vrijednosti je: " + r + " (paran broj)";
		else
			return "Zbir vrijednosti je: " + r + " (neparan broj)";
	}

	public Igra()
	{
		kocke[0] = new Kocka();
		kocke[1] = new Kocka();
	}
}
```

C#

```
using System;
using Kocke.Model;

namespace Kocke.Model
{
    public class IgraKockicama
    {
        private int rez1;
        private int rez2;

        private Kocka[] kocke = new Kocka[2];

        public void igraj()
        {
            kocke[0].baci();
            rez1 = kocke[0].GetVrijednost();

            kocke[1].baci();
            rez2 = kocke[1].GetVrijednost();
        }
```

```
        public String getRezultat()
        {
            int r = rez1 + rez2;
            if (r%2 == 0)
                return "Zbir vrijednosti je: " + r + " (paran broj)";
            else
                return "Zbir vrijednosti je: " + r + " (neparan broj)";
        }

        public Igra()
        {
            kocke[0] = new Kocka();
            kocke[1] = new Kocka();
        }
    }
}
```

4.5.3 Primjer konzolnog UI-a

Da bi napravili funkcionalan program moramo napisati ulaznu tačku ("Main") u program.

Java

```
package ba.fit.kocke.ui;

import ba.fit.kocke.model.Igra;

public class Program
{
       public static void main(String[] args)
       {
              IgraKockicama igra = new IgraKockicama();
              igra.igraj();
              System.out.println(igra.getRezultat());
       }
}
```

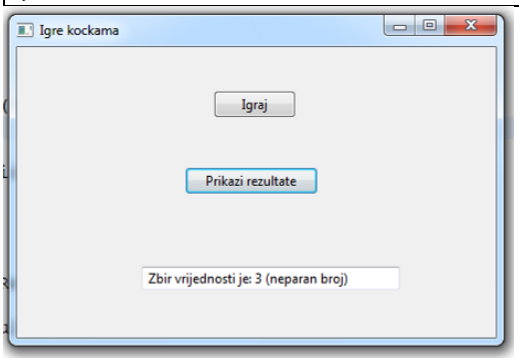
C#

```
namespace Kocke
{
    class Program
    {
        static void Main(string[] args)
        {
             IgraKockicama igra = new IgraKockicama();
            igra.igraj();
            Console.WriteLine(igra.getRezultat());
        }
    }
}
```

4.5.4 Primjer grafičkog UI-a:

JAVA - SWT

```
package ba.fit.kocke.ui;

//...
import ba.fit.kocke.model.IgraKockicama;
import org.eclipse.swt.widgets.Text;



public class Pocetna2
{
	public static void main(String[] args)
	{
		try
		{
			Pocetna2 window = new Pocetna2();
			window.open();
		} catch (Exception e)
		{
			e.printStackTrace();
		}
	}

	public void open()
	{
		//...
	}

	protected void createContents()
	{
		//...
	}

	private IgraKockicama igra = new IgraKockicama ();
	private Text txtRezultat; //textbox

	protected void do_btnIgraj_widgetSelected(SelectionEvent e)
	{
		igra.igraj();
	}

	protected void do_btnPrikaziRezultate_widgetSelected(SelectionEvent e)
	{
		txtRezultat.setText(igra.getRezultat());
	}
}
```



C# Windows Forms

```
public partial class Pocetna2 : Form
    {
        private IgraKockicama igra = new IgraKockicama();

        public Pocetna2()
        {
            InitializeComponent();
        }

        private void btnIgraj_Click(object sender, EventArgs e)
        {
            igra.igraj();
        }

        private void btnPrikaziRezultate_Click(object sender, EventArgs e)
        {
            txtRezultat.Text = igra.getRezultat();
        }
    }
```