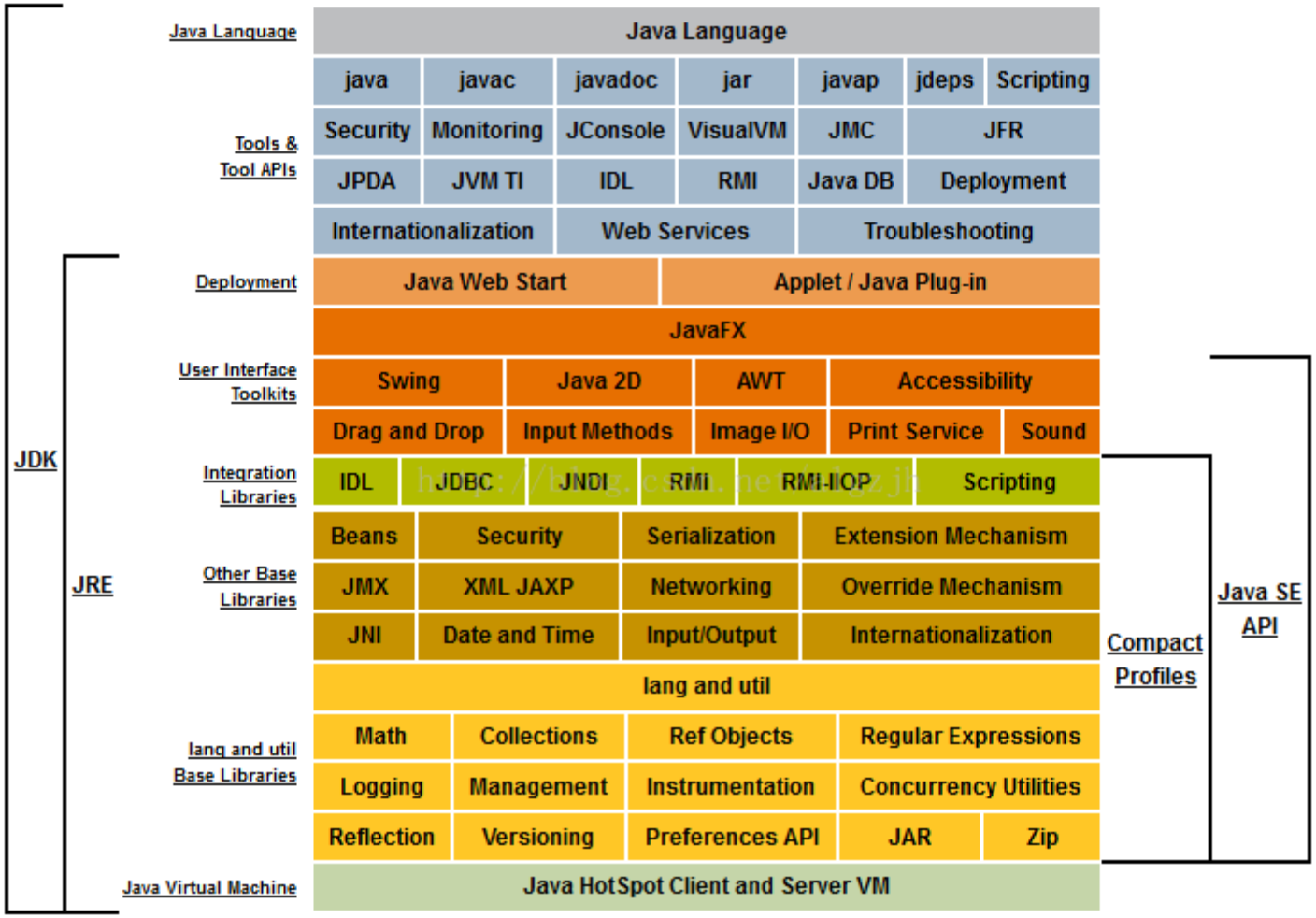
JAVA PLATFORMA

Specifičnosti OOP u Javi

- Sve klase izvedene su iz klase `Object`
- U Javi je sve objekat, osim promenljivih nekog od osam primitivnih tipova
- Nikad ne morate da uništite objekat - automatski sakupljač smeća (engl. garbage collector)
- Zabranjeno višestruko nasleđivanje – koriste se intefejsi
- Tip može biti klasa, interfejs ili neki od osam primitivnih. Ovo su jedine mogućnosti. Samo klase se mogu koristiti za kreiranje novih objekata
- Filozofija Jave – “napiši jednom, pokreni bilo gde” (“write once, run anywhere” - WORA)

Konceptualni dijagram Java komponenti



Java arhive - Java ARchive (JAR)

- JAR je fajl format koji se tipično koristi za agregiranje više Java .class fajlova i pridruženih metapodataka i resursa (tekst, slike, itd.) u jedan fajl radi distribucije
- JAR se zasniva na ZIP formatu, ima ekstenziju .jar
- JAR fajlovi omogućavaju da se efikasno dopremi i pokrene čitava aplikacija, uključujući sve prateće resurse, u jednom zahtevu – primene: web i mobilno programiranje
- Kreiranje .jar iz Eclipse: File ➡ Export ➡ Java ➡ JAR file ➡ ➡ izbor željenih klasa i resursa za uključivanje u JAR
- Sadržaj .jar fajla može biti raspakovan bilo kojim standarnim alatom za dekompresiju ili korišćenjem jar komandog alata

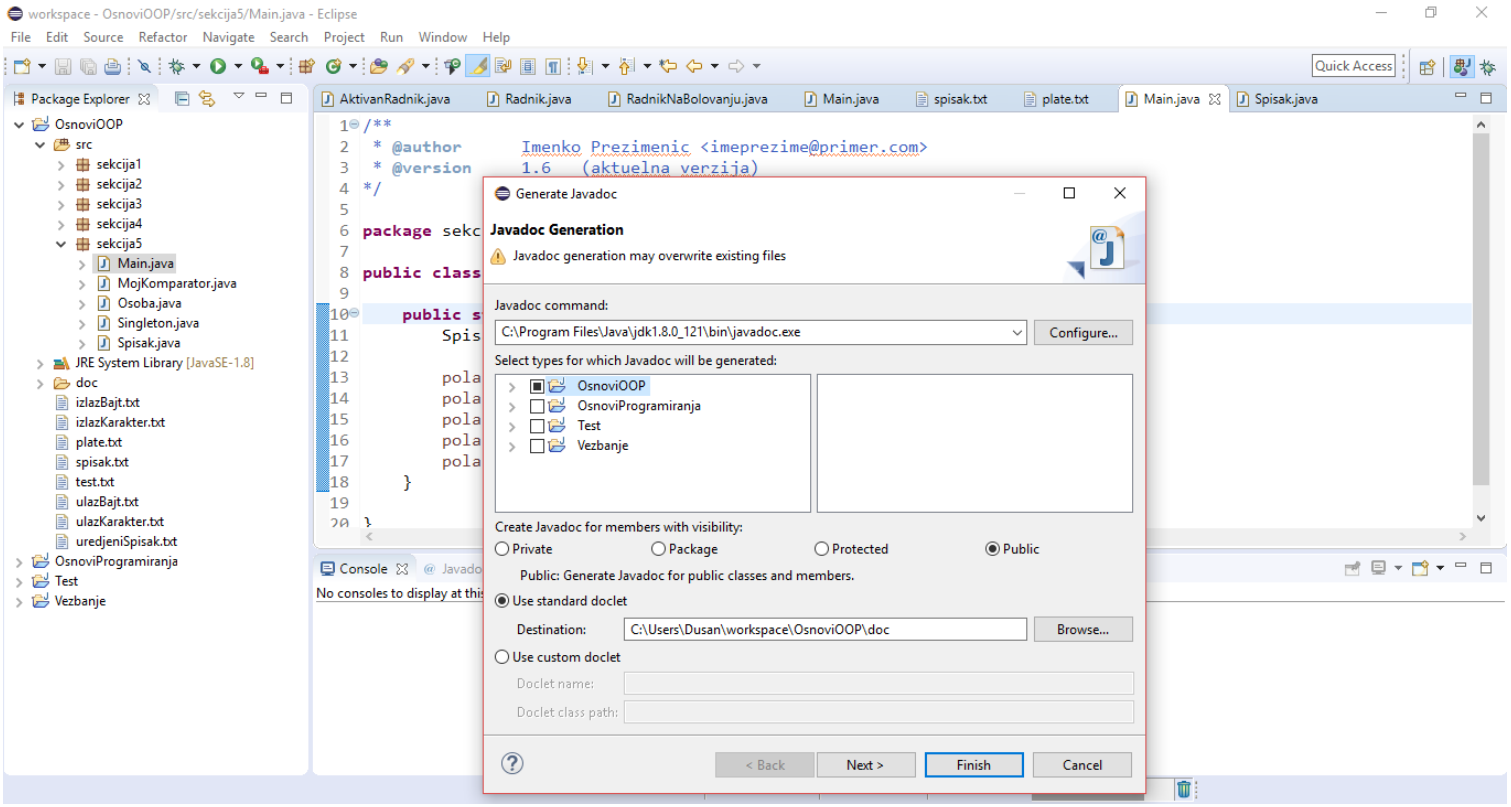
Javadoc

- Javadoc je generator dokumentacije koji služi za generisanje API dokumentacije u HTML obliku direktno iz fajlova sa Java izvornim kodom
- Komentari oblika /** ... */
- Koristi i tagove - `@author`, `@version`, `@param`...
- De facto industrijski standard za dokumentovanje u Javi
- Primer korišćenja:

```
/**
 * @author       Imenko Prezimenic <imeprezime@primer.com>
 * @version      1.6   (aktuelna verzija)
*/
public class Test {
    // telo klase
}
```

Javadoc

- Kod metoda postaviti `@param` i `@return`
- Pokretanje: Project ➡ Generate Javadoc (prethodno podesiti putanju do javadoc.exe)



Java platforma

Java biblioteka klasa može se podeliti u dve osnovne grupe paketa:

1. Prvu grupu čine standardni paketi sa klasama neophodnim za programiranje u Javi

   Primeri:

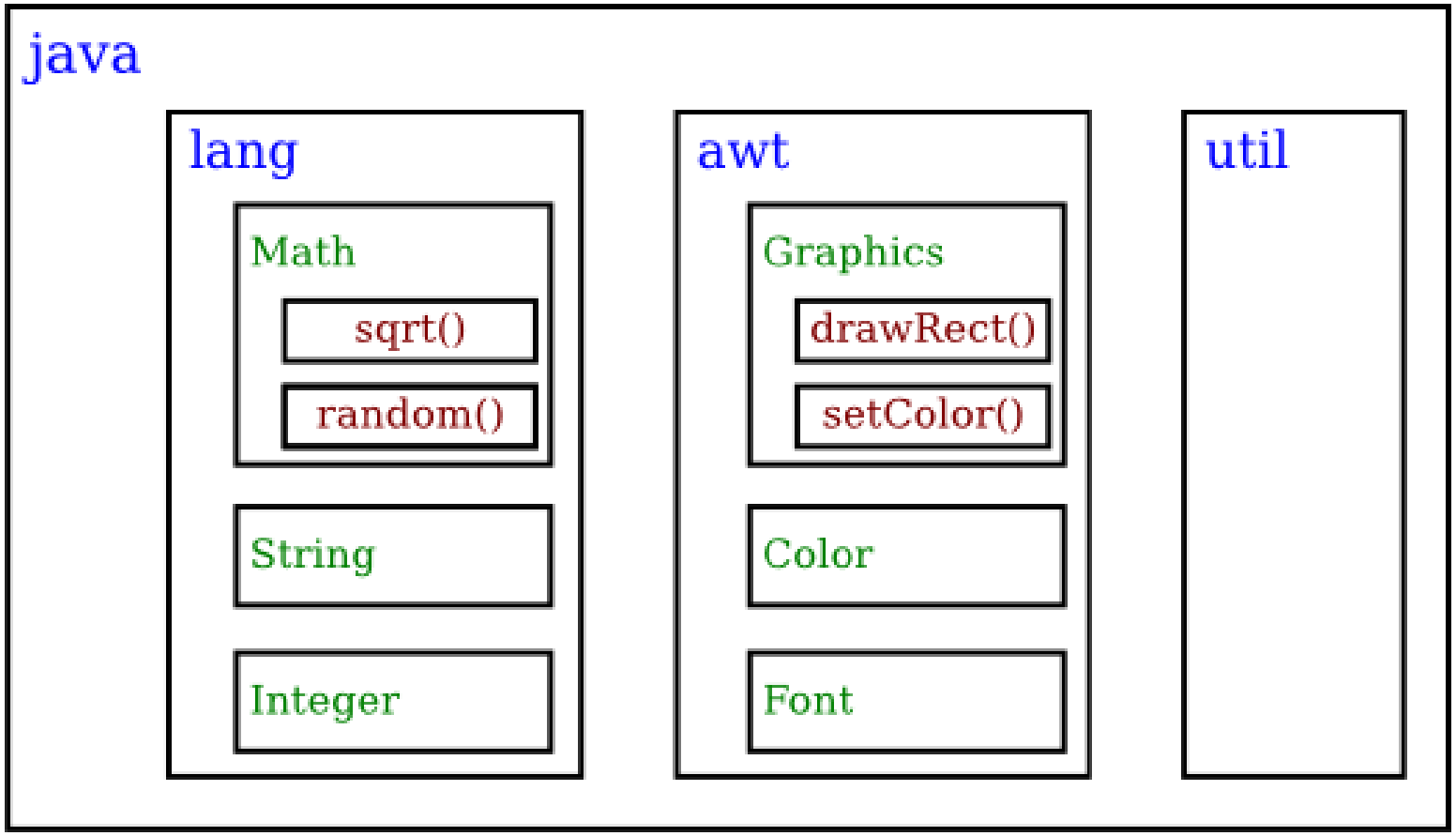
   ```
   java.lang
   java.io
   java.util
   ```

2. Drugu grupu čine dodatni paketi sa klasama za kreiranje apleta, rad sa mrežom itd.

   Primeri:

   ```
   java.applet
   java.net
   ...
   ```

Java platforma – standardni paketi



Metode ugnježdene u klase ugnježdene u dva sloja paketa.
Puno ime metode sqrt() je java.lang.Math.sqrt().

Java platforma – `java.lang`

- Paket `java.lang` sadrži osnovne interfejse i klase koji su neophodni za programiranje u Javi. Ovde spadaju hijerarhija klasa, tipovi koji su deo definicije jezika, osnovni izuzetci, matematičke funkcije itd. Klase iz ovog paketa su **automatski uključene** u svaki Java izvorni fajl.
- Najvažnije klase u `java.lang` su:
  - `Object` - korenska klasa svih klasa
  - `System` – klasa koja pruža sistemske operacije
  - `Math` – klasa sa osnovnim matematičkim funkcijama
  - `Throwable, Exception, Error` – klase za rad sa greškama i izuzecima
  - `String` – klasa za rad sa stringovima
  - `Character, Integer, Float`... – omotač (engl. *wrapper*) klase za primitivne tipove

Java niti – `java.lang`

- Java ima odličnu podršku za multiprocesiranje i rad sa nitima koji su veoma važni na savremenim računarima
- Niti (engl. thread) se predstavljaju objektom koji pripada klasi `java.lang.Thread` (ili nekoj podklasi ove klase) ili objektom klase koja implementira interfejs `java.lang.Runnable`
- Svrha objekta `Thread` je da samo jednom izvrši neki metod. Ovaj metod predstavlja zadatak koji nit treba da izvrši. Više niti može da se izvršava paralelno
- Niti se mogu programirati tako što se kreira klasa izvedena iz klase `Thread` ili klasa koja implementira intefejs `Runnable` i u njoj definiše metod `public void run()`. Implementacija ovog metoda definiše zadatak koji će nit izvršavati

Java platforma – `java.io`

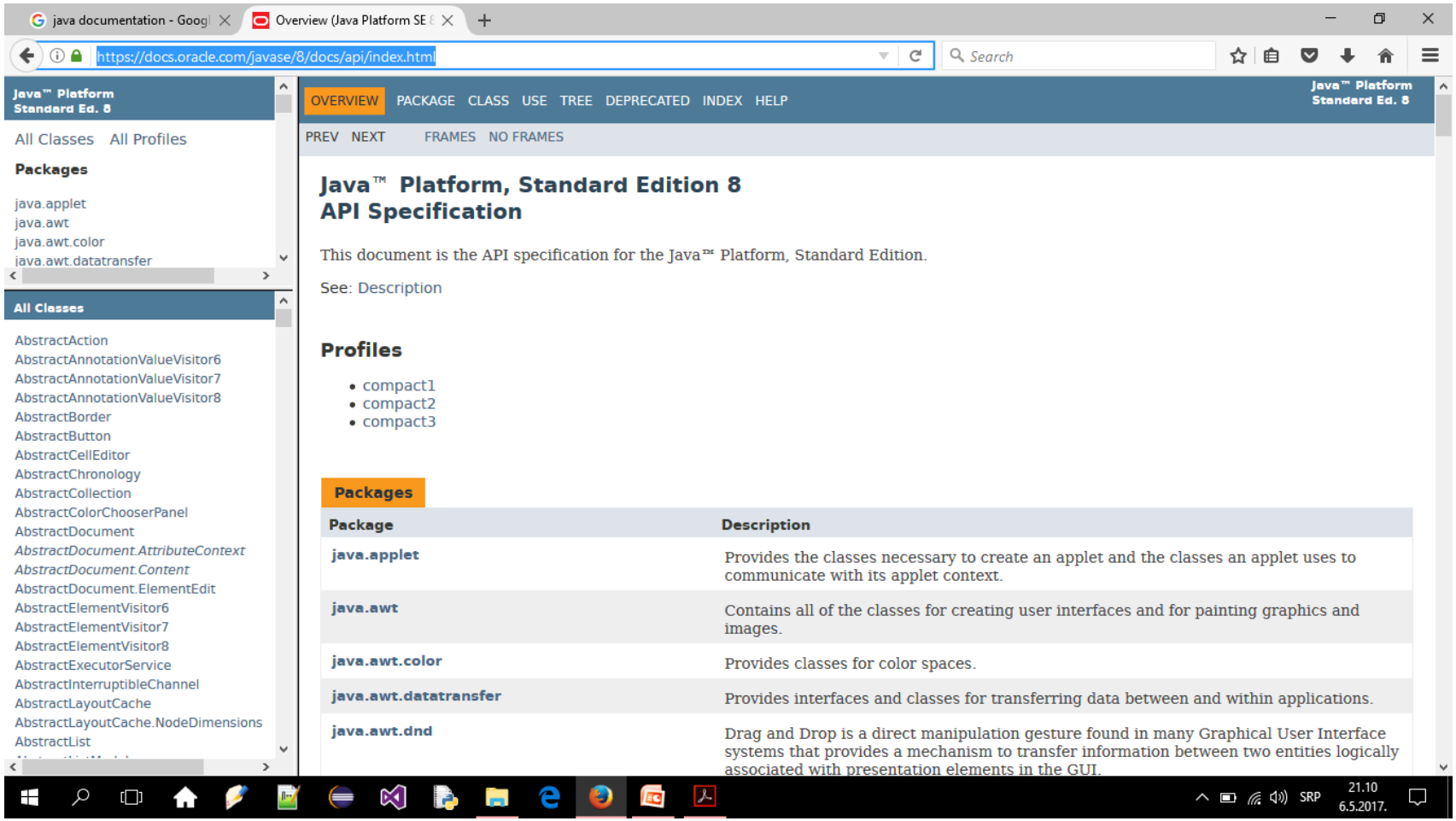
- Paket `java.io` sadrži interfejse i klase za rad sa ulazom i izlazom
- Klase u okviru ovog paketa realizuju rad sa tokovima
- Najvažnije klase su:
  - Za rad sa bajt tokovima – apstraktne klase `InputStream` i `OutputStream`
  - Za rad sa karakter tokovima – apstraktne klase `Reader` i `Writer`
- Metodi klasa ovog paketa generišu izuzetke tipa `IOException` u slučaju da ne mogu biti izvršeni - treba ih pozivati u okviru `try-catch-finally` struktura
- Paket `java.io` sadrži i klase kao što su `RandomAccessFile` (rad sa fajlovima sa slučajnim pristupom) i `File` (predstavlja fajl ili putanju u fajl sistemu)

Java platforma – `java.util`

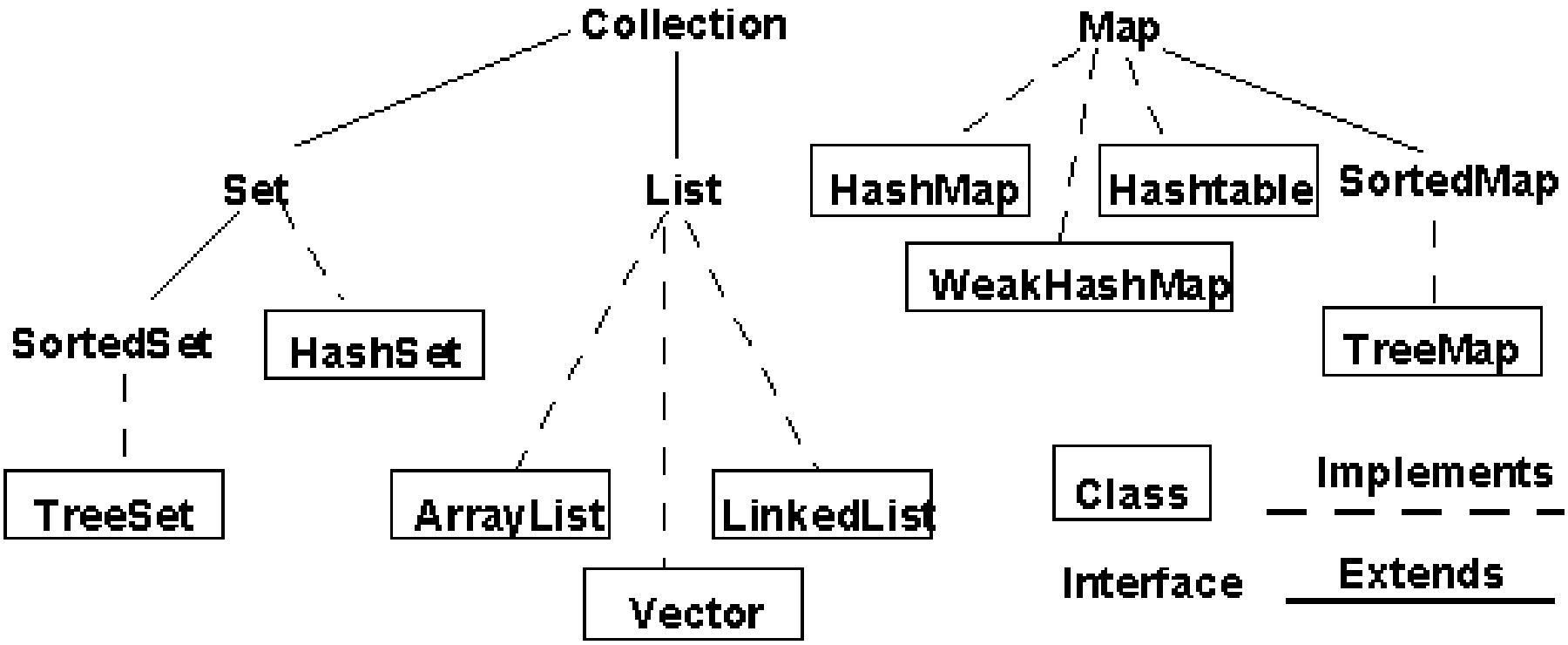
- Paket `java.util` sadrži interfejse i klase sa strukturama podataka, generatom slučajnih brojeva, vremenom i datum i drugim pomoćnim alatima.
- Najvažniji deo ovog paketa je Collections radno okruženje – organizovana hijerarhija struktura podataka koja je projektovana pod jakim uticajem projektnih obrazaca, sadrži npr. `ArrayList, LinkedList, HashTable`, itd.
- U ovom paketu se nalaze važni intefejsi `Iterator, Comparator, Collection` ili `Map`, kao i klase kao što su `Scanner, Vector` ili `Calendar`

Java API dokumentacija

https://docs.oracle.com/javase/8/docs/api/index.html



Java Collections Framework - JCF



Liste i skupovi u JCF

- Dva osnovna tipa kolekcija elemenata u Javi su **lista** i **skup**
- **Lista se sastoji od sekvence elemenata u linearnom uređenju**. Lista ima tačno određeno uređenje, ali to ne znači da su vrednosti elemenata u njoj sortirane
- **Skup je kolekcija u kojoj ne postoje duplirani elementi**. Elementi skupa mogu, ali i ne moraju da imaju neko uređenje
- Treći tip kolekcija koji se nešto ređe koristi nego liste i skupovi su **redovi sa prioritetom** (engl. priority queues)

Liste, skupovi i mape u JCF

- Dva standardne strukture podataka za predstavljanje listi su **dinamički niz** i **lančana lista**
- Skupovi u Javi, za razliku od matematičkog pojma skupa, moraju biti konačni i sadržati samo elemente istog tipa
- **Mape su vid generalizovanih nizova**. Sastoje se od elemenata u vidu parova (ključ, vrednost). Osnova za rad sa mapama u Javi je interfejs `Map<K,V>`
- Savremeni sistemi za rad sa velikim skupovima podataka kao što su Hadoop i Spark, zasnovani su na Javi i radu sa mapama – MapReduce programski model

JCF liste - ArrayList, LinkedList

- Objekat tipa `ArrayList<T>` predstavlja uređenu sekvencu objekata tipa `T`, smeštenih u nizu koji može da raste po potrebi – kad god se doda novi element
- Objekat tipa `LinkedList<T>` takođe predstavlja uređenu sekvencu objekata tipa `T`, ali objekti se čuvaju u čvorovima (engl. nodes) koji su međusobno uvezani pokazivačima
- Klasa `LinkedList` je efikasnija u primenama gde se često dodaju ili uklanjaju elementi na početku ili u sredini liste, dok je klasa `ArrayList` efikasnija kada je potreban čest slučajan pristup elementima liste
- Obe liste implementiraju metode interfejsa `Collection`, pa je moguće njihovo lako sortiranje (`sort`), okretanje (`reverse`), itd.

JCF skupovi - TreeSet, HashSet

- Skupovi implementiraju sve metode interfejsa `Collection`, ali na takav način da obezbede da se nijedan elemenat ne može pojaviti dva puta u skupu
- Skup `TreeSet` ima svojstvo da su njegovi elementi uređeni u rastući redosled
- Skup `HashSet` čuva svoje elemente u posebnoj strukturi podataka poznatoj kao heš tabela (engl. hash table)
- Kod heš tabela su operacije pronalaženja, dodavanja i brisanja elementa vrlo efikasne (dosta brže nego kod `TreeSets`). Elementi `HashSet`-a se ne čuvaju u nikakvom posebnom uređenju

Zadatak 5.1 – Spisak polaznika

- Napraviti program koji čitanjem iz ulaznog tekstualnog fajla *spisak.txt* prihvata podatke o polaznicima (ime, prezime, JMBG) i prikazuje ih na ekranu. Potom treba spisak polaznika sortirati po JMBG-u, ponovo ga prikazati na ekranu i na kraju ga upisati i u izlazni fajl *uredjeniSpisak.txt*

- Klase testirati u glavnom programu kreiranjem objekta sa spiskom polaznika i pozivanjem odgovarajućih metoda

Zadatak 5.1 – klasa Osoba

```
public class Osoba {
	private String ime;
	private String prezime;
	private String jmbg;

	Osoba() {}

	public Osoba(String ime, String prezime, String jmbg){
		this.ime = ime;
		this.prezime = prezime;
		this.jmbg = jmbg;
	}

	public String pribaviIme(){
		return this.ime;
	}
	...
```

Zadatak 5.1 – klasa Osoba

```
...

public String pribaviPrezime(){
        return this.prezime;
}

public String pribaviJMBG(){
        return this.jmbg;
}

public void postaviIme(String ime){
        this.ime = ime;
}

public void postaviPrezime(String prezime){
        this.prezime = prezime;
}
...
```

Zadatak 5.1 – klasa Osoba

```
        ...


        public void postaviJMBG(String jmbg){
                this.jmbg = jmbg;
        }

        @Override public String toString() {
                return ("Ime:" + this.pribaviIme() + " Prezime: "
                        + this.pribaviPrezime() + " JMBG: “
                        + this.pribaviJMBG());
        }

}
```

Zadatak 5.1 – klasa MojKomparator

```
import java.util.*;

class MojKomparator implements Comparator<Osoba> {

	@Override public int compare(Osoba o1, Osoba o2) {
		int i = o1.pribaviJMBG().compareTo(o2.pribaviJMBG())
		if (i > 0) {
			return -1;
		}
		else if (i < 0) {
			return 1;
		}
		return 0;
	}

}
```

Zadatak 5.1 – klasa Spisak

```
import java.io.*;
import java.util.*;

public class Spisak {
        ArrayList<Osoba> listaPolaznika;

        public void ucitajListu(String imeFajla) {
                Scanner s = null;
                ArrayList<Osoba> listaPolaznika = new ArrayList<Osoba>();
                try {
                        s = new Scanner(new File(imeFajla));
                        do {
                                String ime = s.next();
                                String prezime =  s.next();
                                String jmbg = s.next();
                                Osoba noviPolaznik = new Osoba(ime, prezime, jmbg);
                                listaPolaznika.add(noviPolaznik);
                        } while (s.hasNext());
                } catch (IOException e) {
                        System.out.println(e.getMessage());
                } ...
```

Zadatak 5.1 – klasa Spisak

```
        ...
        finally {
                        if (s != null) {
                                s.close();
                        }
        }
        this.listaPolaznika = listaPolaznika;
}

public void sortirajListu() {
        Collections.sort(this.listaPolaznika, new MojKomparator());
}

public void stampajListu() {
        System.out.println(Arrays.toString(this.listaPolaznika.toArray()));
}

...
```

Zadatak 5.1 – klasa Spisak

```
...
public void upisiListu(String imeFajla) {
        PrintWriter pw = null;
        try {
                pw = new PrintWriter(new FileOutputStream(imeFajla));
                for (Osoba polaznik : this.listaPolaznika)
                        pw.println(polaznik.pribaviIme() + " " +
                                polaznik.pribaviPrezime() + " " +
                                polaznik.pribaviJMBG());
        } catch (IOException e) {
                System.out.println(e.getMessage());
        }
        finally {
                if (pw != null) {
                        pw.close();
                }
        }
}

}
```

Zadatak 5.1 – klasa Main

```
public class Main {

	public static void main(String[] args) {

		Spisak polaznici = new Spisak();

		polaznici.ucitajListu("spisak.txt");
		polaznici.stampajListu();
		polaznici.sortirajListu();
		polaznici.stampajListu();
		polaznici.upisiListu("uredjeniSpisak.txt");
	}
}
```

Zadaci za rad na času

- Modifikovati paket `zaposleni` tako da uključuje i klasu `Spisak`. Za čuvanje spiska radnika upotrebiti pogodnu strukturu iz Java Collections Framework-a.
- Modifikovati klase `Institucija`, `Ucionica`, `Zaposleni` (koja nasleđuje klasu `Osoba`) i `Racunar` tako da koriste gotove strukture podataka iz Java Collections Framework. Pod kojim uslovima je za čuvanje sekvence objekata efikasnije koristiti `ArrayList`, a pod kojima `LinkedList`?

UNIFIED MODELING LANGUAGE (UML)

UML

- Objedinjeni jezik za modelovanje - UML (Unified Modeling Language)
- UML predstavlja standardizovani jezik i grafičku notaciju za
  - vizuelizaciju,
  - specifikaciju,
  - modelovanje i
  - dokumentovanje

  delova softverskog sistema koji se projektuje
- UML predstavlja zajednički “rečnik” za sporazumevanje između osoba uključenih u projekovanje i razvoj nekog softverskog sistema

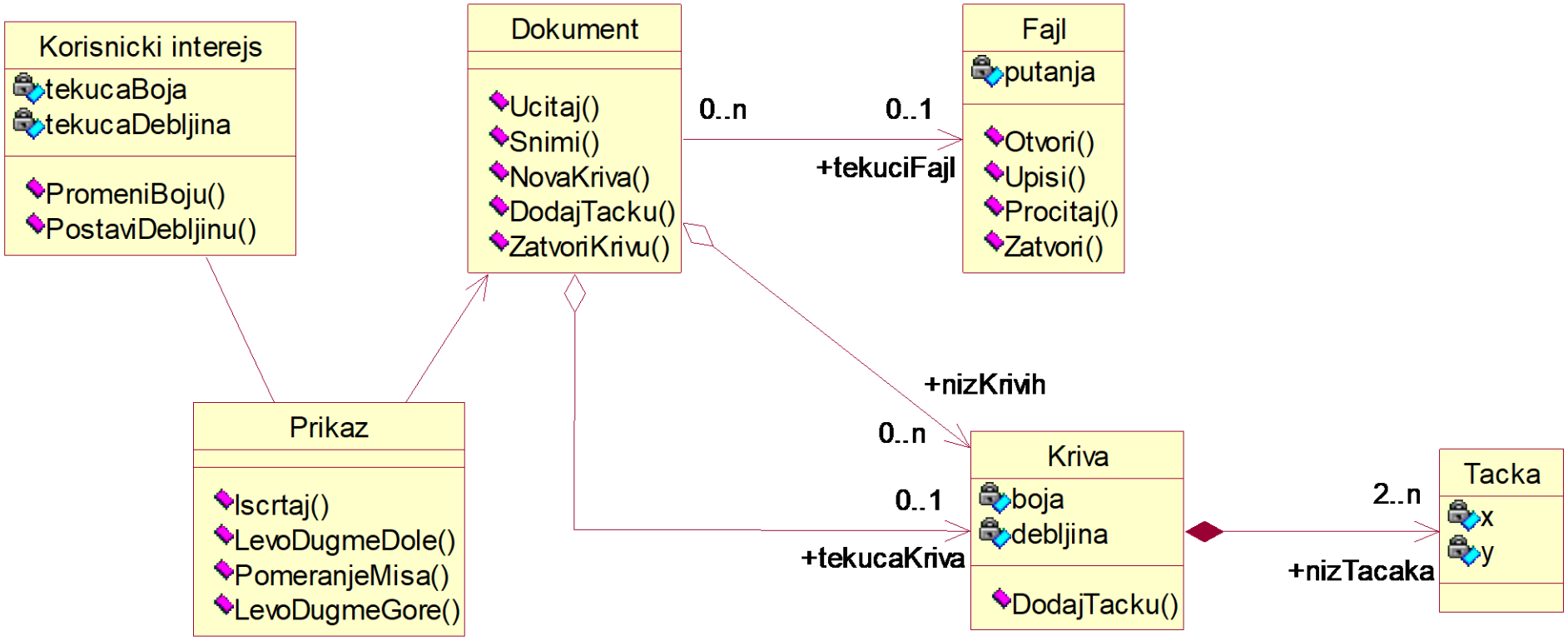
Tipovi UML dijagrama

- Dijagrami klasa (engl. *Class Diagram*)
- Dijagrami slučajeva korišćenja (engl. *Use-Case Diagram*)
- Dijagrami sekvence (engl. *Sequence Diagram*)
- Dijagrami saradnje (engl. *Collaboration Diagram*)
- Dijagrami stanja (engl. *Statechart Diagram*)
- Dijagrami aktivnosti (engl. *Activity Diagram*)
- Dijagrami komponenti (engl. *Component Diagram*)
- Dijagrami razmeštaja (engl. *Deployment Diagram*)

Dijagrami klasa

- Koriste se za predstavljanje klasa i njihove organizacije u pakete
- Dijagrami klasa se koriste za modelovanje
  - domena sistema
  - aplikacije
- Elementi dijagrama su:
  - klase
  - veze između klasa
    - nasleđivanje
    - asocijacija
    - agregacija
    - kompozicija
  - paketi
  - veze zavisnosti između paketa

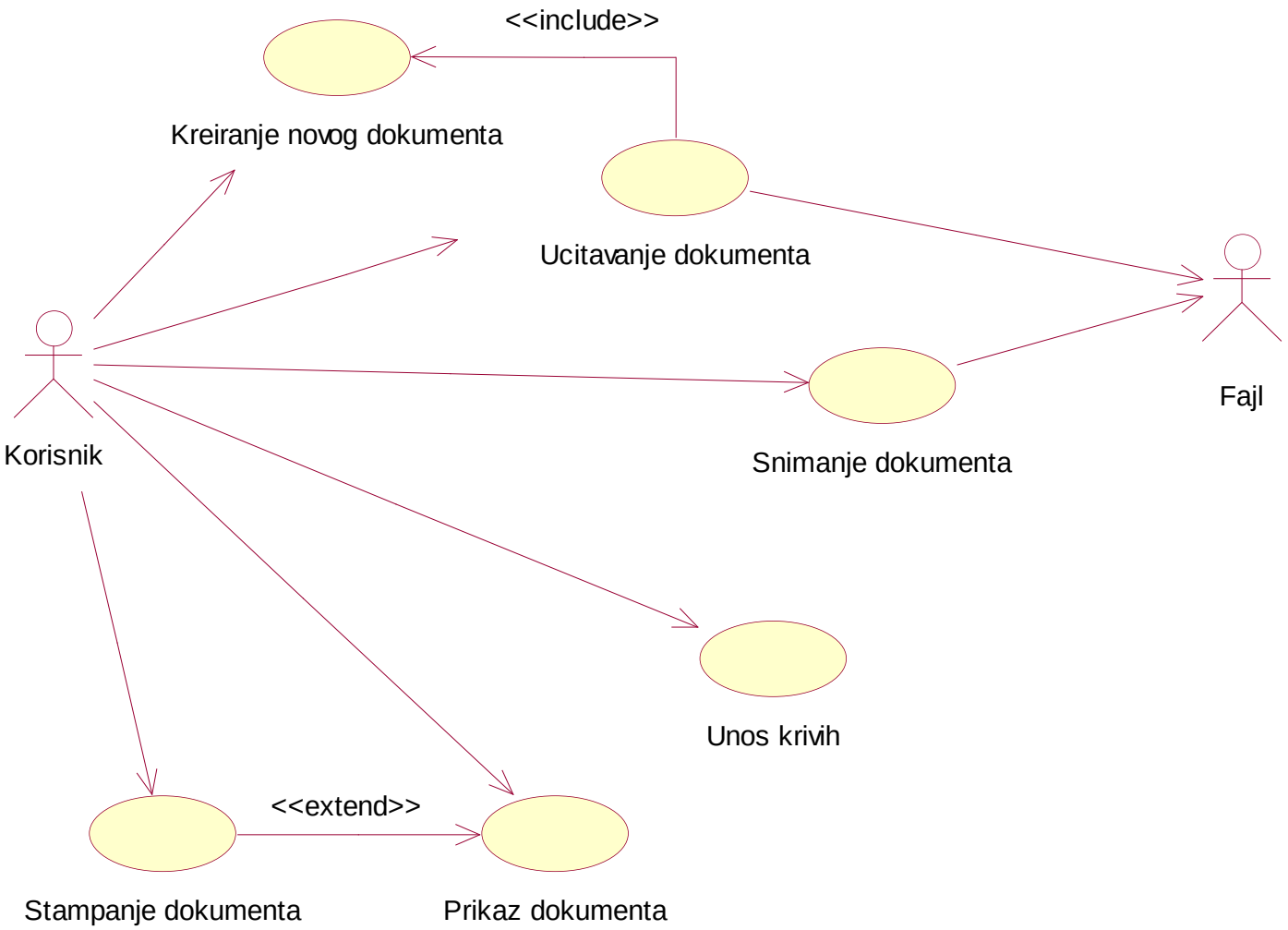
Dijagram klasa - primer



Dijagrami slučajeva korišćenja

- Koriste se u procesu prikupljanja i dokumentovanja korisničkih zahteva
- Elementi dijagrama su:
  - akteri
    - korisnici sistema
    - drugi sistemi iz okruženja
  - slučajevi korišćenja sistema
  - veze između aktera i slučajeva korišćenja
    - asocijacija
    - generalizacija
  - paketi
  - veze zavisnosti između paketa

Dijagram slučajeva korišćenja - primer

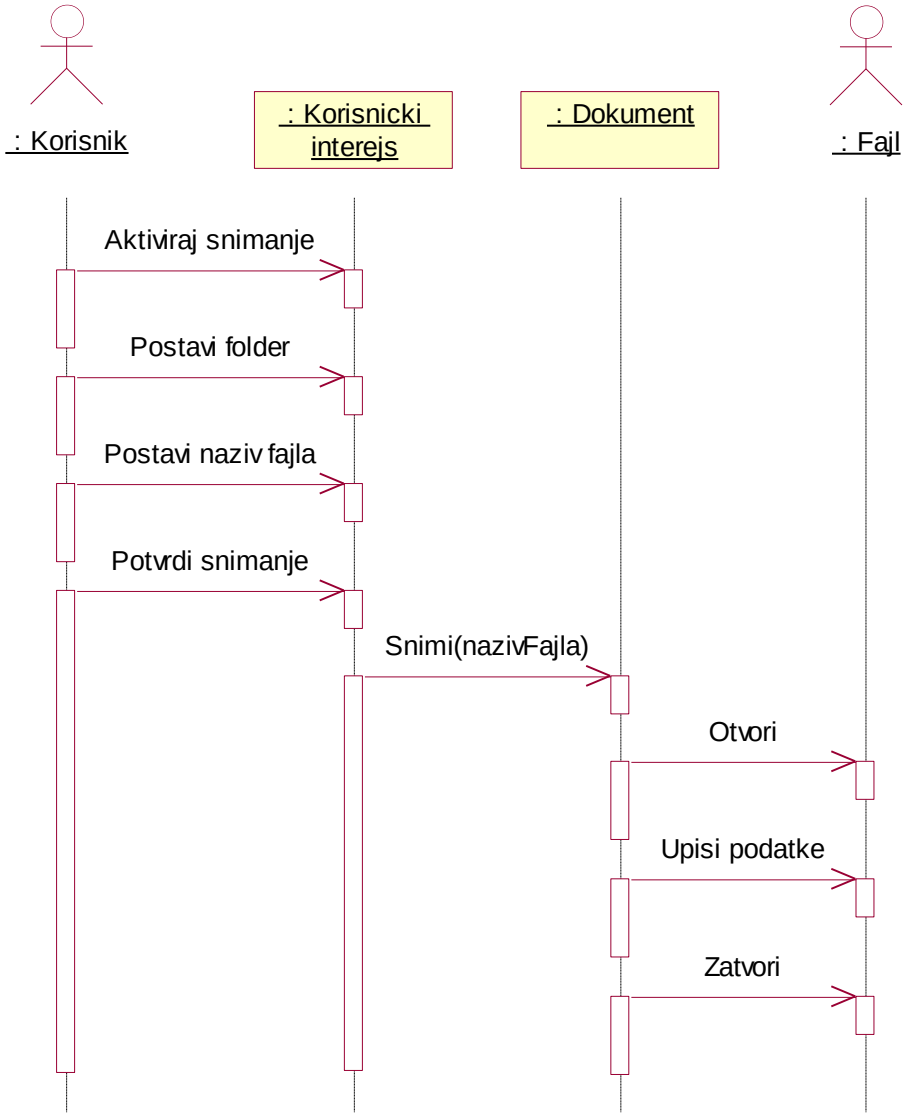


Dijagrami sekvence

- Koriste se za predstavljanje scenarija interakcije između objekata u sistemu
  - Najčešće se ovi scenariji odnose na slučajeve korišćenja sistema

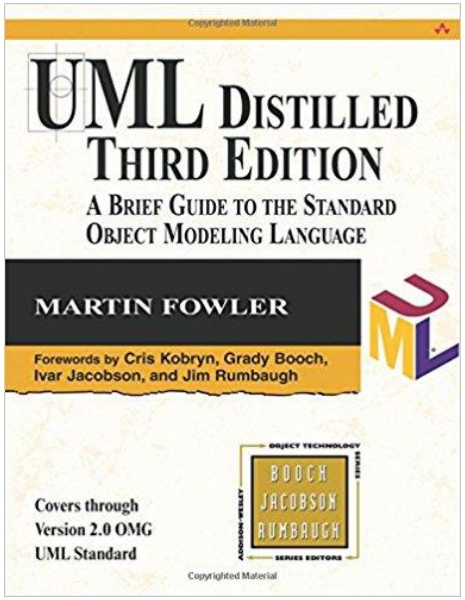
- Elementi dijagrama su:
  - objekti
  - vremenska linija
  - poruke između objekata

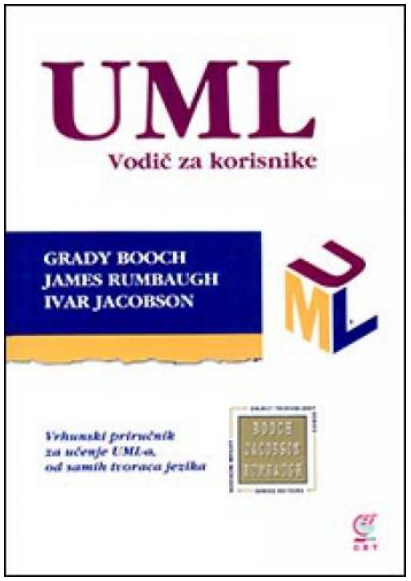
Dijagram sekvence - primer



Literatura za UML

- Martin Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition, Addison Wesley, 2003.
- Grady Booch, James Rumbaugh, Ivar Jacobson, UML - Vodič za korisnike, CET, 2001.





Zadaci za vežbanje

- Nacrtati UML dijagrame klasa i dijagrame slučajeva korišćenja za klase ranije implementirane u okviru paketa `zaposleni`, `matematika` i `institucija`. Identifikovati tipične slučajeve korišćenja za aplikacije koje koriste prethodne pakete.

OBJEKTNO-ORIJENTISANO PROJEKTOVANJE SOFTVERA

Projektovanje OO programa

"*Postoje dva načina za projektovanje i razvoj programa. Jedan je da ih učinite toliko jednostavnim da je očigledno da nemaju nedostatke. Drugi je da ih učinite toliko komplikovanim da nemaju* ***očiglednih*** *nedostataka.*"

C.A.R. Hoare

- Ključ za **uspešno OO projektovanje programa** je **dobro osmišljena apstrakcija problema** kroz objektne koncepte:

  "*Suština apstrakcije je čuvanje informacija koje su relevantne u datom kontekstu i ignorisanje informacija koje su irelevatne za dati konkteks*"

  John Guttag

Ulazni podaci za OO projektovanje

- **Konceptualni model** je rezultat objektno-orijentisane analize i opisuje koncepte u problemskom domenu. Eksplicitno se kreira tako da bude nezavisan od implementacionih detalja, kao što su konkurentnost ili čuvanje podataka
- **Slučajevi korišćenja** (use cases) su opisi sekvence događaja koji zajedno dovode do toga da sistem realizuje neku korisnu aktivnost
- **Dijagrami sekvence** grafički prikazuju, za određeni scenario u okviru nekog slučaja korišćenja, događaje koje generišu eksterni akteri, njihov redosled i moguće događaje unutar sistema

Ulazni podaci za OO projektovanje

- **Dokumentacija za korisnički interfejs** (engl. user interface – UI) prikazuje i opisuje izgled i tok rada sa programom putem korisničkog interfejsa finalnog softverskog proizvoda
- **Relacioni model podataka** – model podataka je apstraktni model koji opisuje kako se podaci predstavljaju i koriste. Ako se ne koristi objektna baza podataka, tada je relacioni model podataka neophodno unapred kreirati, pošto je izabrana strategija za objektno-relaciono mapiranje jedan od izlaza procesa objektno-orijentisanog projektovanja

Projektovanje OO programa

- **Definisati objekte** (kreirati dijagrame klase na osnovu konceptualnih dijagrama). Tom prilikom se entiteti tipično mapiraju na klase
- **Definisati elemente klasa** (atributi i metode)
- **Definisati broj objekata**, **trenutak** njihovog **nastajanja** i **nestajanja**, kao i **način medjusobne interakcije** tokom vremena
- **Definisati odgovornosti** svakog dela sistema

Projektovanje OO programa

- **Upotrebiti projektne obrazce** (ako su primenljivi) - glavna prednost primene projektnih obrazaca je mogućnost njihovog ponovnog korišćenja u više aplikacija. Objektno-orijentisani projektni obrazci obično prikazuju odnose i interakcije između klasa i objekata, bez specifikacije konačnih aplikacionih klasa i objekata
- **Izabrati radno okruženje** (ako je primenljivo) - radna okruženja uključuju veliki broj bibilioteka i klasa koje se mogu iskoristi za implementaciju standarnih struktura u aplikaciji. Na ovaj način se može dosta uštedeti na vremenu razvoja softvera pošto se izbegava ponovno pisanje velikog dela koda prilikom razvoja novih aplikacija
- **Identifikovati perzistentne objekte/podatke** (ako je primenljivo) – potrebno je identifikovati objekte koji treba da postoje duže od trajanja jednog izvršenja aplikacija. Ako aplikacija koristi relacionu bazu podataka, projektovati objektno-relaciono mapiranje

Projektni obrasci

- **Projektni obrasci** su **opšta, ponovo upotrebljiva rešenja za probleme koji se često javljaju** u određenom kontekstu projektovanja softvera
- Oni **nisu kompletan projekat koji se može direktno transformisati u izvorni ili mašinski kod**, već su opis ili šablon za rešavanje problema koji može da se koristi u mnogo različitih situacija
- Projektni obrasci su **formalizovani postupci najbolje prakse** (engl. best practices) koje programeri mogu koristiti kako bi efikasno rešili tipične probleme koji se javljaju prilikom projektovanja aplikacije ili sistema

Projektni obrasci

- Projektni obrasci su originalno (prema GoF) podeljeni na: **stvaralačke** (engl. creational), **strukturalne** (engl. structural) i **bihevijoralne** (engl. behavioral), a danas se koriste i **konkurentni** (npr. blockchain) i **arhitekturalni** (npr. Model-View-Controller - MVC)
- **Stvaralački**: Singleton (osigurava da klasa ima samo jednu instancu za koju postoji globalni pristup), Builder, Factory...
- **Strukturalni**: Adapter, Facade, Decorator...
- **Bihevijoralni**: Iterator, Interpreter, Visitor...

Primer 5.1 – obrazac Singleton u Javi

```
public class Singleton {

	private static Singleton instanca = null;

	protected Singleton() {
		// Postoji samo kako bi sprecili instanciranje
	}

	public static Singleton pribaviInstancu() {
		if(instanca == null) {
			instanca = new Singleton();
		}
		return instanca;
	}
}
```

Tipične primene: čuvanje podešavanja aplikacije (konfiguracioni fajlovi), logovanje podataka, itd.

Primer 5.2 – obrazac Factory u Javi

- Sa Factory obrascem, kreiramo objekat bez potrebe da izložimo logiku kreiranja objekta (engl. *creation logic*) klijentu i potom se obraćamo novostvorenom objektu korišćenjem zajedničkog interfejsa
- Kreiraćemo interfejs `Figura` i konkretne klase koje implementiraju ovaj interfejs. Potom ćemo u narednom koraku definisati klasu „fabriku“ `FabrikaOblika`
- Napravićemo i test klasu `ObrazacFabrikaTest` koja će koristiti klasu `FabrikaOblika` kako bi pribavila odgovarajući objekat nekog oblika. Test klasa će samo prosleđivati informaciju fabrici oblika da li je u pitanju krug, kvadrat ili pravouganik, a klasa `FabrikaOblika` će potom „isporučivati“ traženi oblik test klasi