Објектно орјентисано програмирање

Владимир Филиповић
vladaf@matf.bg.ac.rs
Александар Картељ
kartelj@matf.bg.ac.rs

Апстрактне класе и интерфејси

Владимир Филиповић
vladaf@matf.bg.ac.rs
Александар Картељ
kartelj@matf.bg.ac.rs

Математички факултет

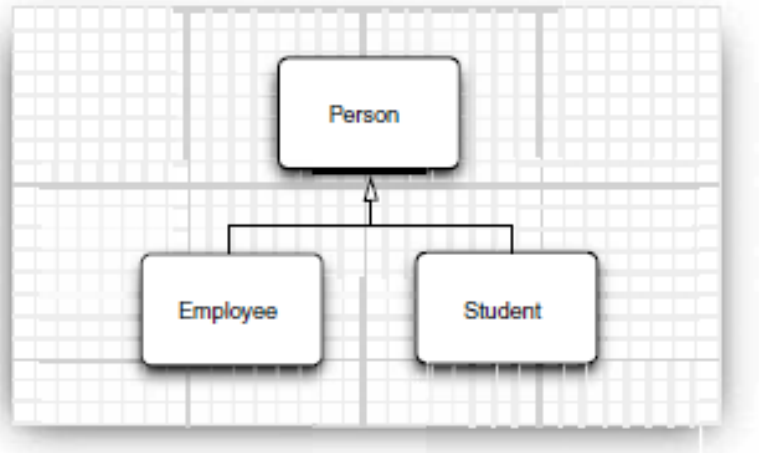
{vladaf,kartelj}@matf.bg.ac.rs

Апстрактне класе

- Како се крећемо уз хијерархију наслеђивања, класе постају све општије и све апстрактније.
- У неком тренутку наткласа постаје у тој мери општа да више представља основу за друге класе него класу чије конкретне примерке желимо да користимо.

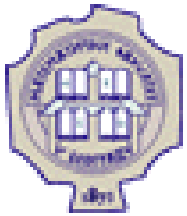
Пример:

- Проширење хијерархије класа за запослене и студенте.
- Иако свака особа има опис, ипак тај опис зависи од тога шта и како дата особа ради.



Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Апстрактне класе (2)

- Апстрактне класе и методе карактерише кључна реч abstract.
- Класа мора бити апстрактна ако садржи бар један апстрактни метод.
- Класа може бити апстрактна чак иако не садржи ни један апстрактни метод.

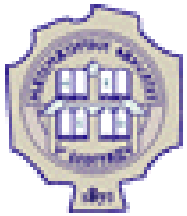
```
public abstract class Person {
        private String jmbg;

        void checkJMBG(){
                System.out.println("JMBG checking");
        }

        abstract void doMedicalTreatment();
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Апстрактне класе (3)

- Примерак конкретне класе може да користи неапстрактне методе апстрактне наткласе.

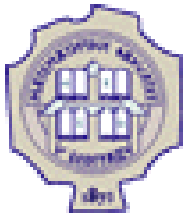
```
public class Student extends Person{
        private String index;

        @Override void doMedicalTreatment() {
                System.out.println("Going to student clinic");
                checkJMBG();
                checkIndex();
        }

        void checkIndex(){
                System.out.println("Checking index");
        }
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Апстрактне класе (4)

- Тај примерак може бити декларисан као инстанца апстрактне класе.
- Он може бити креиран само помоћу конструктора конкретне класе.
- Апстрактна класа може да има конструктор којим дефинише сопствена поља, а тај контруктор се потом позива од стране конструктора конкретне потклассе.

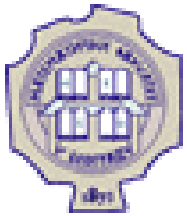
```
public abstract class Person {
        ...
        public Person(String jmbg) { this.jmbg=jmbg; }
        ...
}

public class Student extends Person{
        ...
        public Student(String jmbg, String index) {
                super(jmbg);
                this.index=index;
        }

        public static void main(String[] args) {
                Person s=new Student("xxxxxxxxxxxxx", "yyyyy");
        }
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

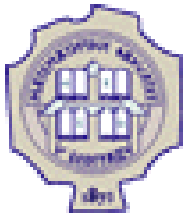


Интерфејси

- У развоју софтвера је често важно да се различите групе програмера договоре око „уговора“ о интеракцији софтвера.
- Свака од тих група треба да буде у могућности да напише свој део кода, а да при томе нема информације како је писан код друге стране.
- У језику Јава, интерфејс је референтни тип, сличан класи, али може садржати само константе и потписе метода.
- Интерфејс не може да садржи тела метода (изузетак су подразумевани методи, почев од Јава 8).
- Није могуће директно правити примерак интерфејса:
  - он само може да буде имплементиран од стране класе
  - или наслеђен од стране другог интерфејса

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

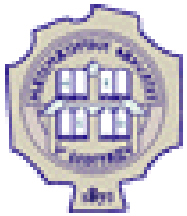


Интерфејси (2)

- Интерфејси (као апстрактне класе и методи) обезбеђују шаблоне за неко понашање, а које ће друге класе користити.
- Преко интерфејса уводи се неки вид ограниченог вишеструког наслеђивања.
- Интерфејс обезбеђује апстрактно понашање које се додаје било којој класи, а које није обезбеђено преко њених надкласа.
- Они представљају неку врсту протокола за комуникацију између класа, тј. дефинишу шаблоне за понашање класа.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



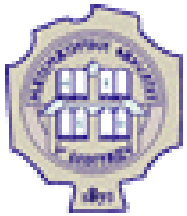
Интерфејси (3)

- Интерфејс се понаша свуда као класа, али не може имати инстанце (не може се на њега применити оператор new).
- Интерфејс садржи апстрактне методе (што се не мора посебно нагласити јер се подразумева) и константе.
- Дакле, интерфејс не може садржавати променљиве.
- Нови интерфејс се креира са:

```
public interface MojInterfejs {
      ……
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси (4)

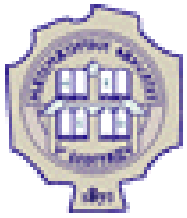
- Код интерфејса нема хијерархијске организације.
- Како би нагласили да један интерфејс наслеђује више других, иза кључне речи extends наводимо све интерфејсе које овај наслеђује.

```
public interface DrugiInterfejs extends Prvi, Primarni {
        …// svi metodi su public i abstract
        …//sve promenljive su: public, static i final
}
```

- Интерфејси се, као и класе, смештају у пакете.
- Ако се за методе и променљиве у интерфејсу не нагласи да су abstract, public и final, подразумеваће се да је тако.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



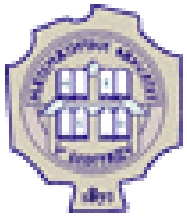
Интерфејси (5)

- Дефинисани интерфејси се имплементирају од стране Јава класа.
- Можемо користити већ раније дефинисане интерфејсе (који већ постоје у Јава-библиотеци) или направити своје.
- Када класа имплементира интерфејс, тада се морају имплементирати сви методи интерфејса (не могу се бирати само неки међу њима да се имплементирају, а неки да се оставе неимплементираним).

```
class MojAplet extends java.applet.Applet implements Runnable {
        ……..
        //implementacije svih metoda iz interfejsa Runnable
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси (6)

- Када се интерфејс имплементира у некој класи, њена потклаcа наслеђује све методе и може их превазићи (предефинисати).
- Ако је у класи имплеметиран интерфејс, није неопходно да се реч implements јави и у дефиницији потклаcе.

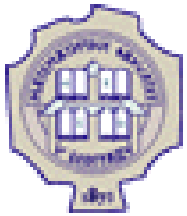
Пример:

```
interface Radoznao{

        void pita();
        void interesuje_se();
        //…
}
class Naucnik implements Radoznao{
        String ime;
        // …
}
class Istrazivac extends Naucnik {
// Ovde se mogu koristiti metodi pita() i Interesuje_se()
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси (7)

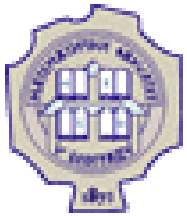
- Једна класа може имплементирати више интерфејса.
- На пример, може се писати:

```
public class Moja implements Prvi, Drugi, Treci {
        // …
}
```

- Овде се могу појавити иста имена метода (са истим потписом!) у различитим интерфејсима.
- Тада се коришћењем кратког имена може имплементирати само један од два таква метода (ако се редефинишу оба метода унутар класе, неопходно је користити пуна имена).

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси (8)

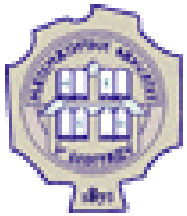
- Могу се декларитати променљиве које ће бити типа интерфејс (јер скоро свуда где користимо класе, можемо користити и интерфејсе!)
- На пример, могуће је креирати објекат на следећи начин:

```
Runnable trceci = new MojObjekat();
```

- Од објекта trceci се очекује да извршава метод run() интерфејса Runnable.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси (9)

- Како се могу користити параметри у методима интерфејса ако ће их имплементирати различите класе?
- Једна од могућности је да се параметри декларишу тако да буду типа интерфејса.

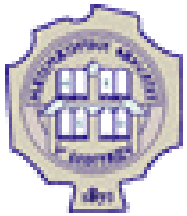
```
public interface Radoznao {
        void pita(Radoznao neko);
        //…
}

public class Naucnik implements Radoznao {

        public pita(Radoznao neko){
                Naucnik pravi = (Naucnik)neko;
                // …
        }
}
```

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси у ЈДК-у

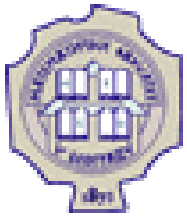
- Обично испоручилац сервиса тврди: “Ако ваша класа испуњава конкретни интерфејс, ја ћу онда пружити услугу.”
- Размотримо конкретан пример. Метод sort у класи Arrays обећава да ће сортирати низ објеката, али под једним условом:
  - објекти у низу морају сами знати како да се упореде тј. морају припадати класи која имплементира интерфејс Comparable.

```
public interface Comparable {
        int compareTo(Object other);
}
```

- Да би класа имплементирала интерфејс Comparable она мора да садржи метод compareTo.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Интерфејси у ЈДК-у (5)

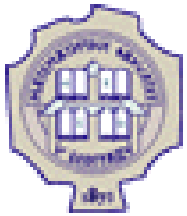
- **java.lang.Comparable**
  - int compareTo(Object other)
- **java.util.Arrays**
  - static void sort(Object[] a)
    - Сортира елементе низа побољшаном верзијом сортирања учешљавањем (енг. Merge sort).
    - Елементи низа морају имплементирати Comparable интерфејс.
  - static void sort(Object[] a, Comparator c)

  - Друга варијанта која користи експлицитни начин поређења дефинисан класом Comparator
- **java.util.Comparator**
  - int compare(Object o1, Object o2)

    - Пореди два објекта и враћа:
      1. негативан број ако први претходи другом,
      2. враћа нулу ако су исти по уређењу
      3. или позитиван број ако други претходи првом.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



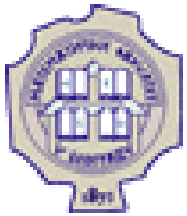
SOLID принципи - S

Принцип једнозначне одговорности (**S**ingle responsibility) – свака класа треба да има тачно једну одговорност

- По овом принципу, може постојати само један разлог због ког класа треба да буде промењена
- Класа треба да реализује само један задатак
    - ако би се у оквиру једне класе налазила функционалност за пословну логику и за чување података примерка у датотеку или у базу, то би нарушавало овај принцип
- Када се поштује овај принцип, тестирање је једноставније
- Мање функционалности у једној класи такође значи да има мање зависности од осталих класа, што доводи до боље организације кода, јер се мање класе са јасном сврхом могу лакше пронаћи

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



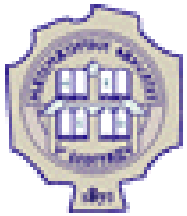
SOLID принципи - O

Принцип отворености и затворености (**O**pen closed) – Софтверске компоненте треба да буду отворене за проширивање, али затворене за модификацију

- По овом принципу, класа треба да буде структуирана тако да реализује своје задатке без претпостављања да ће у будућности бити мењано њено понашање
- Истовремено, треба да буде омогућено проширивање функционалности класе, на следећи начин:
  - Наслеђивањем дате класе
  - Превазилажењем понашања које је захтевано од класе
  - Превазилажењем појединих понашања класе
- Пример: аналогија са прегледачем и проширењима – укључена проширења не ометају функцију прегледача

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



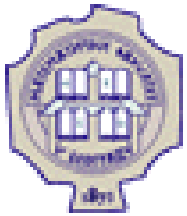
SOLID принципи - L

Принцип замене Лисков (**L**iskov substitution) – Примерци потклаcа морају да буду у могућности да у потпуности (у свим аспектима и контекстима) замењују примерке надкласе.

- Принцип захтева да: ако је S подтип типа T, тада објекти типа T у програму могу бити замењени објектима типа S без промене пожељних особина програма
- Другим речима, класа мора реализовати све уговоре својих надкласа
- Поштовањем овог принципа се спречава злоупотреба наслеђивања. Такође, тиме се постиже сагласност са односом наслеђивања „јесте“

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



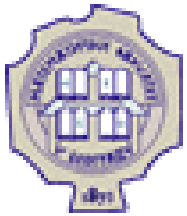
SOLID принципи - I

Принцип раздвајања интерфејса (**I**nterface segregation) – Клијенти не треба да буду приморани да имплементирају непотребне методе, тј. методе које неће користити

- По овом принципу, клијент никад не сме да има обавезу да зависи од било ког метода који не користи.
- Другим речима, фаворизују се мали интерфејси који зависе од клијента, а не монолитни и велики интерфејси

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

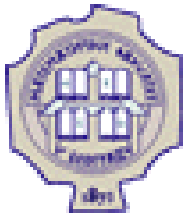


SOLID принципи - D

Принцип инверзије зависности (**D**ependency inversion) – Треба зависити од апстрактција, а не од конкретне реализације

- По овом принципу, модули вишег нивоа никако не треба да зависе од модула нижег нивоа, већ и модули нижег нивоа и модули вишег нивоа треба да зависе од апстракција
- Апстракције не треба да зависе од детаља, већ детаљи треба да зависе од апстракције
    - Пример: само процесирање кредитних картица не зависи од типа кредитне картице
- У реализацији се обично користи контејнер за инверзију зависности (нпр. контејнер у оквиру радног оквира Spring код програмског језика Јава)

Математички факултет
{vladaf,kartelj}@matf.bg.ac.rs



Додатне препоруке за наслеђивање

1. **Заједничке операције и поља сместити у надкласе.**

Тако је оформљена класа Person као надкласа Employee и Student.

2. Избегавати употребу заштићених поља.

- Модификатор protected не пружа много заштите, из два разлога:
    1. Може се увек направити поткласа неке класе и тиме приступити protected променљивој.
    2. У програмском језику Јава све класе у истом пакету имају приступ protected пољима, тако да се класа може сместити у исти пакет и тиме омогућити приступ.
- Међутим, protected методи могу бити корисни за назначавање да дати метод није спреман за општу употребу и да треба да буде редефинисан у потклasama.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

Додатне препоруке за наслеђивање (2)

3. Користити наслеђивање за моделирање односа “јесте”.

SOLID

- Понекад програмери претерују у коришћењу наслеђивања.

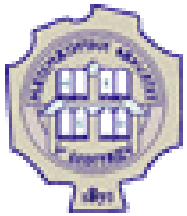
На пример, претпоставимо да нам требају радници по уговору, тј. класа Contractor. Радници под уговором садрже имена и датум запослења, али не садрже плату, већ се плаћају по сату.

Иако постоји изазов да се класа Contractor направи као подкласа класе Employee којој је додато поље hourlyWage, то не би била добра идеја јер би тада примерак класе Contractor садржао и поље за плату и поље за сатницу, а то би водило у проблеме.

Наиме однос између ентитета радник по уговору и запослени не пролази тест “јесте”. Радник по уговору није специјалан случај запосленог.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Додатне препоруке за наслеђивање (3)

4. Не користити наслеђивање сем уколико оно има смисла за све методе класе из које се наслеђује.

SOLID

На пример, претпоставимо да желимо да направимо класу за празнике Holiday.

Будући да је сваки празник дан, а да су дани примерци класе LocalDate, то можемо користити наслеђивање.

```
class Holiday extends LocalDate { . . . }
```

На несрећу, скуп празника није затворен у односу на наслеђене операције. Један од јавних метода класе LocalDate је метод add.

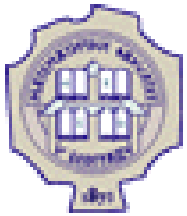
Међутим, овај метод може да претвори празник у нерадни дан:

```
Holiday christmas;
christmas.plusDays(12);
```

Стога, у овом примеру наслеђивање није адекватно.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Додатне препоруке за наслеђивање (4)

5. Приликом превазилажења метода не мењати очекивано понашање тј. поштовати принцип замене.

SOLID

Принцип замене се примењује и на синтаксу и на понашање.

На пример, при превазилажењу метода се не сме неразумно мењати његово понашање. На пример, ако се „поправи“ проблем са методом add у класи Holiday тако да сада add пребацује на следећи празник, тада бива нарушен принцип замене.

Наиме, секвенца нареби:

```
d1 = …;
d2 = d1.plusDays(1);
System.out.println(datum2.until(datum1,ChronoUnit.DAYS));
```

треба да има очекивано понашање, тј. да врати 1, без обзира да ли су променљиве d1 и d2 типа LocalDate или Holiday.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

Додатне препоруке за наслеђивање (5)

6. Користити полиморфизам, а не информације о типу.

SOLID

- Кад год се наиђе на код облика:

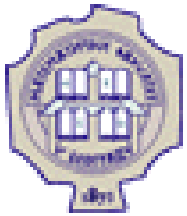
```
if (x is of type 1)
        action1(x);
else if (x is of type 2)
        action2(x);
```

  треба размотрити могућност полиморфизма.

- Да ли action1 и action2 представљају заједничке концепте?
  - Ако је одговор да, тај заједнички концепт треба да буде метод заједничке надкласе или интерфејса.
  - Потом се једноставно треба позвати x.action(); па да механизам динамичког активирања који је инхерентан полиморфизму покреће одговарајућу акцију.

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Догађаји

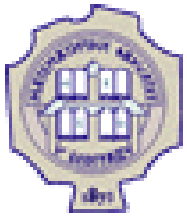
Иако различити оквири за развој у Јави имају различите објекте за рад са догађајима, њихово понашање је фундаментално исто.

Рад са догађајима (испаљивање и руковање) захтева следеће елементе:

- Класу за тип догађаја – садржи информације везане за тај тип догађаја
- Интерфејс који описује како ће о испаљивању догађаја бити информисани заинтересовани објекти, тзв. ослушкивачи
    - Ослушкивачи морају да имплементирају интерфејс
- Структура података која води рачуна о томе који су ослушкивачи претплаћени на догађаје
- Методи којима се омогућује да се додају/уклоне ослушкивачи на догађаје

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

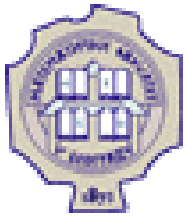


Класа Event

- Први захтев за испаљивање догађаја је постојање класе која представља информацију о догађају
- Класе за догађаје треба да буду имутабилне, тј. после креирања не могу бити мењани
- Испаљивање догађаја прослеђује референцу на један приемрак догађаја према свим регистрованим ослушкивачима
  - Редослед прослеђивања према ослушкивачима није специфициран
  - Ослушкивачи не могу модификовати догађај који им је прослеђен
  - Ослушкивачи не могу променити начин на који ће се наставити процесирање од стране других ослушкивача

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

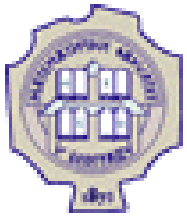


Класа Event (2)

- Класе за догађаје су изведене из класе java.util.EventObject
  - EventObject обезбеђује метод getSource() којим се одређује извор догађаја
  - Извођењем из класе EventObject, креирани објекти могу да буду генерички обрађени

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

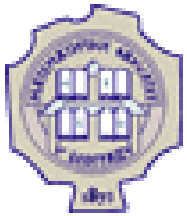


Интерфејс EventListener

- Ослушкивачи догађаја су класе које су исказале интерес за догађаје који се испаљују
- Класе ослушкивачи бивају информисани о испаљеном догађају тако што се позове одговарајући метод ослушкивача
- Како се зна који је метод одговарајући, тј. који ће се метод позвати?
    - Дефинисање догађаја захтева да се дефинише уговор између извора за догађај и ослушкивача тог догађаја
    - Тај уговор одређује који ће се метод ослушкивача звати
    - Сваки ослушкивач мора имплементирати метод
- Дакле, помоћу интерфејса се дефинише тај уговор, па сваки ослушкивач мора имплементирати интерфејс

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs

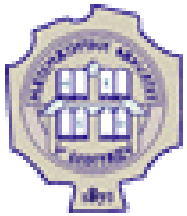


Интерфејс EventListener (2)

- У примерима са догађајима, интерфејси тј. уговори ће проширивати интерфејс java.util.EventListener
- Интерфејс EventListener не захтева да се имплементира неки метод, већ се ради о тзв. маркерском интерфејсу – то је само ознака да се ради о интерфејсу који служи као ослушкивач догађаја
    - Овим се олакшава да се одреди који догађај може бити испаљен

Математички факултет

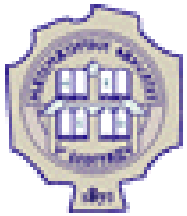
{vladaf,kartelj}@matf.bg.ac.rs



Извор за догађаје

- Објекат који представља извор за догађаје чува информације о ослушкивачима
- Истовремено, објекат ове класе испаљује догађаје
- У различитим оквирима за развој обично већ постоји извор за догађаје, па тада нема потребе да га програмер посебно програмира

Математички факултет
{vladaf,kartelj}@matf.bg.ac.rs

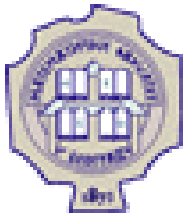


Ослушкивач догађаја

- Објекат који представља ослушкивач догађаја, у методи која имплементира интерфејс изведен из интерфејса EventListener специфицира шта ће се урадити када догађај буде испаљен – наравно под претпоставком да је објекат ослушкивач догађаја већ регистрован за слушање датог догађаја
    - Метод ослушкивача који се реализује приликом испаљивања догађаја обично има један параметар – догађај који је испаљен

Математички факултет

{vladaf,kartelj}@matf.bg.ac.rs



Захвалница

Велики део материјала који је укључен у ову презентацију је преузет из презентације коју је раније (у време када је он држао курс Објектно орјентисано програмирање) направио проф. др Душан Тошић.

Хвала проф. Тошићу што се сагласио са укључивањем тог материјала у садашњу презентацији, као и на помоћи коју ми је пружио током конципцирања и реализације курса.