9.1. Interfejsi

9.2. Unutrašnje klase

9.3. Programi vođeni događajima

9.4. Obrada događaja u Javi

9.5. Zadatak

9.1. Interfejsi

Interfejsi, slično apstraktnim klasama, obezbeđuju šablone ponašanja koje druge klase treba da implementiraju. Java interfejs je kolekcija apstraktnih metoda, koja se može prilagoditi svakoj klasi, bez potrebe da ta klasa nasleđuje neku drugu klasu. Interfejs sadrži isključivo definicije apstraktnih metoda i konstante - ne postoje atributi i implementacije metoda.

Interfejsi veoma podsećaju na klase, deklarišu se na sličan način i mogu se organizovati u hijerarhije. Deklaracija interfejsa je ista kao i deklaracija klase, pri čemu je ključna reč **class** zamenjena sa **interface**. Unutar definicije interfejsa sve metode se definišu kao javne i apstraktne i ukoliko nisu eksplicitno definisane kao takve one će biti pretvorene u javne i apstraktne metode. Unutar interfejsa ne mogu se definisati metode korišćenjem modifikatora pristupa private ili protected. Kao i kod apstraktnih metoda u klasama, metode unutar interfejsa ne sadrže kod, već samo njihovu deklaraciju. Interfejs sa dakle sastoji samo od deklaracije metoda, implementacija metoda nije uključena u interfejs. Pored metoda, interfejsi mogu da sadrže i promenjive koje moraju biti deklarisane korišćenjem ključnih reči **public static final** (čime postaju konstante).

Primer 1. U primeru je prikazan izgled interfejsa MouseListener iz paketa java.awt.event.

```
public interface MouseListener {
        /**
        * Invoked when the mouse has been clicked on a component.
        */
        public void mouseClicked(MouseEvent e);
        /**
        * Invoked when a mouse button has been pressed on a component.
        */
        public void mousePressed(MouseEvent e);
        /**
        * Invoked when a mouse button has been released on a component.
        */
        public void mouseReleased(MouseEvent e);
        /**
        * Invoked when the mouse enters a component.
        */
        public void mouseEntered(MouseEvent e);
        /**
        * Invoked when the mouse exits a component.
        */
        public void mouseExited(MouseEvent e);
}
```

Interfejsi predstavljaju komplement klasa i proširuju njihove mogućnosti, pa se interfejsi i klase mogu tretirati na isti način, ali interfejs ne može da se instancira: upotrebom ključne reči **new** može da se kreira samo instanca klase i to u slučaju da nije reč o apstraktnoj klasi. Ukoliko nije moguće instancirati interfejs sa : **new MouseListener()**, kako se onda interfejsi koriste?

Kao primer korišćenja interfejsa iskoristićemo predhodno prikazani interfejs MouseListener:

Klasa JButton poseduje metodu **addMouseListener(MouseListener m)** koja kao argument m očekuje objekat tipa **MouseListener** koji je kao što smo videli interfejs i nije ga moguće instancirati. Objekat prosleđen u ovu metodu biće odgovoran za ponašanje objekta klase JButton kada se sa njim izvrši neka od akcija miša (klik miša, pritisak na taster miša, otpuštanje tastera miša, ulazak kursora miša u prostor dugmeta ili izlazak kursora miša iz prostora dugmeta), odnosno objekat prosleđen kao argument u ovu metodu mora biti „sposoban“ da odgovori na bilo koji od ovih 5 događaja. To može biti bilo koja klasa koja na tačno određeni način poseduje 5 implementiranih metoda - za svaki od navedenih događaja po jednu, odnosno to može biti bilo koja klasa koja **implementira interfejs** MouseListener.

Da bi se implementirao interfejs moraju se implementirati sve metode koje su navedene u tom interfejsu - iako se možda neće sve koristiti. Implementiranjem interfejsa u nekoj klasi daje se do znanja da ta klasa pruža sve one funkcionalnosti koje su definisane samim interfejsom.

Primer klase MyMouseListener1 koja implementira intefejs MouseListener:

```
public class MyMouseListener1 implements MouseListener{

	public void mouseClicked(MouseEvent arg0) {
		System.out.println("Kliknuo neko na miša. Pozvana metoda u klasi
					MyMouseListener1");
	}

	public void mouseEntered(MouseEvent arg0) {
	}

	public void mouseExited(MouseEvent arg0) {
	}

	public void mousePressed(MouseEvent arg0) {
	}

	public void mouseReleased(MouseEvent arg0) {

	}

}
```

Primer klase MyMouseListener2 koja implementira intefejs MouseListener:

```
public class MyMouseListener2 implements MouseListener{

	public void mouseClicked(MouseEvent arg0) {
		System.out.println("Kliknuo neka na miša. Pozvana metoda u klasi
					MyMouseListener2");
	}

	public void mouseEntered(MouseEvent arg0) {
		System.out.println("Neko je iznad dugmeta! Pozvana metoda u
					klasi MyMouseListener2");
	}

	public void mouseExited(MouseEvent arg0) {

	}

	public void mousePressed(MouseEvent arg0) {

	}

	public void mouseReleased(MouseEvent arg0) {

	}

}
```

Ključna reč **implements** označava da data klasa „implementira“ interfejs čiji naziv sledi iza te reči, odnosno označava se da će data klasa obezbediti svu funkcionalnost koju je interfejs definisao. To se postiže implementiranjem svih metoda iz interfejsa. Unutar klase koja implementira interfejs pored definicije metoda iz interfejsa može da se nalazi i bilo koji drugi, proizvoljan korisnički kod. Neograničen broj klasa može da implementira jedan interfejs, dok s druge strane jedna klasa može da implementira proizvoljan broj interfejsa:

```
public class MyClass implements Interface1, Interface2, Interface3{
….
…
}
```

Klasa je tada naravno u obavezi da implementira sve metode iz svih interfejsa koji su navedeni iza ključne reči **implements**. U slučaju da postoje izvedene klase iz klase koja implementira neki interfejs, tada i klasa naslednik automatski implementira dati interfejs sa mogućnosti da se implementirane metode interfejsa nadjačaju (redefinišu).

U našem primeru i klasa **MyMouseListener1** i klasa **MyMouseListener2** implementiraju interfejs **MouseListener**, što znači da su obe klase sposobne da pruže svu funkcionalnost koju zahteva ovaj interfejs, odnosno imaju pravo da zamene interfejs MouseListener na bilo kom mestu, bilo kao argumenti metoda, bilo kao tip podataka za atribut...

Kako metoda addMouseListener(MouseListener m) klase JButton očekuje kao argument objekat tipa MouseListener u poziv metode možemo proslediti objekat bilo koje klase koja implementira na navedeni način dati interfejs: svaka klasa koja implementira interfejs MouseListener garantovano će imati svu potrebnu navedenu funkcionalnost (jer je u obavezi da definiše sve navedene metode).

```
JButton btn1=new JButton("JButton 1");
btn1.addMouseListener(new MyMouseListener1());

JButton btn2=new JButton("JButton 2");
btn2.addMouseListener(new MyMouseListener2());
```

U slučaju da je prosleđena instanca klase koja ne implementira dati interfejs javiće se greška!!!

Takođe je moguće i promenjivoj koja je deklarisana kao tip nekog interfejsa dodeliti instancu klase koja implementira dati interfejs:

```
MouseListener m;

m=new MyMouseListener1();

JButton btn1=new JButton("JButton 1");
btn1.addMouseListener(m);

m=new MyMouseListener2();
JButton btn2=new JButton("JButton 2");
btn2.addMouseListener(m);
```

Importovati projekat Termin9 i pokrenuti aplikaciju sa argumentom 1. Pogledati i prokomentarisati klase MyMouseListener1, MyMouseListener2 i FrameInterface.

Za razliku od Jave, većina ostalih objektno-orijentisanih programskih jezika uključuju koncept višestukog nasleđivanja, kojim se rešava problem u vezi sa nasleđivanjem više od jedne nadklase, čime izvedena klasa u isto vreme preuzima sve metode i atribute svih nadklasa. Zbog toga što je jedan od osnovnih razloga prilikom kreiranja programskog jezika Java bio da se formira jednostavan jezik, odbačeno je višestruko nasleđivanje, a implementirano samo jednostruko nasleđivanje. Korišćenjem interfejsa moguće je prevazići ovaj problem jer jedna klasa može implementirati proizvoljan broj interfejsa.

9.2. Unutrašnje klase

Sve klase sa kojima smo do sada radili bile su članice paketa a datoteke u kojima su se nalazile imale su isti naziv kao klase. Ove klase se nazivaju klasama najvišeg nivoa. Pored njih u Javi je moguće kreirati i klasu unutar klase. Ove klase se nazivaju unutrašnjim klasama :

```
public class FrameInnerClass extends JFrame {
        public FrameInnerClass() {
                        Toolkit kit = Toolkit.getDefaultToolkit();
            Dimension screenSize = kit.getScreenSize();
            int screenHeight = screenSize.height;
            int screenWidth = screenSize.width;

            setSize(screenWidth / 2, screenHeight / 2);
            setTitle("Interface");
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setLocationRelativeTo(null);

            JPanel pan=new JPanel();

            MouseListener m;
            m=new MyMouseListener3();
            JButton btn1=new JButton("JButton 1");
            btn1.addMouseListener(m);
            pan.add(btn1);

            add(pan,BorderLayout.CENTER);


         }

         class MyMouseListener3 implements MouseListener{

                public void mouseClicked(MouseEvent arg0) {
                        System.out.println("Kliknuo neko na miša. Pozvana metoda u
                                                klasi MyMouseListener3");
                }

                public void mouseEntered(MouseEvent arg0) {
                }

                public void mouseExited(MouseEvent arg0) {
                }

                public void mousePressed(MouseEvent arg0) {
                }

                public void mouseReleased(MouseEvent arg0) {

                }
        }

}
```

Pokrenuti aplikaciju sa argumentom 2. Pogledati i prokomentarisati klasu FrameInnerClass.

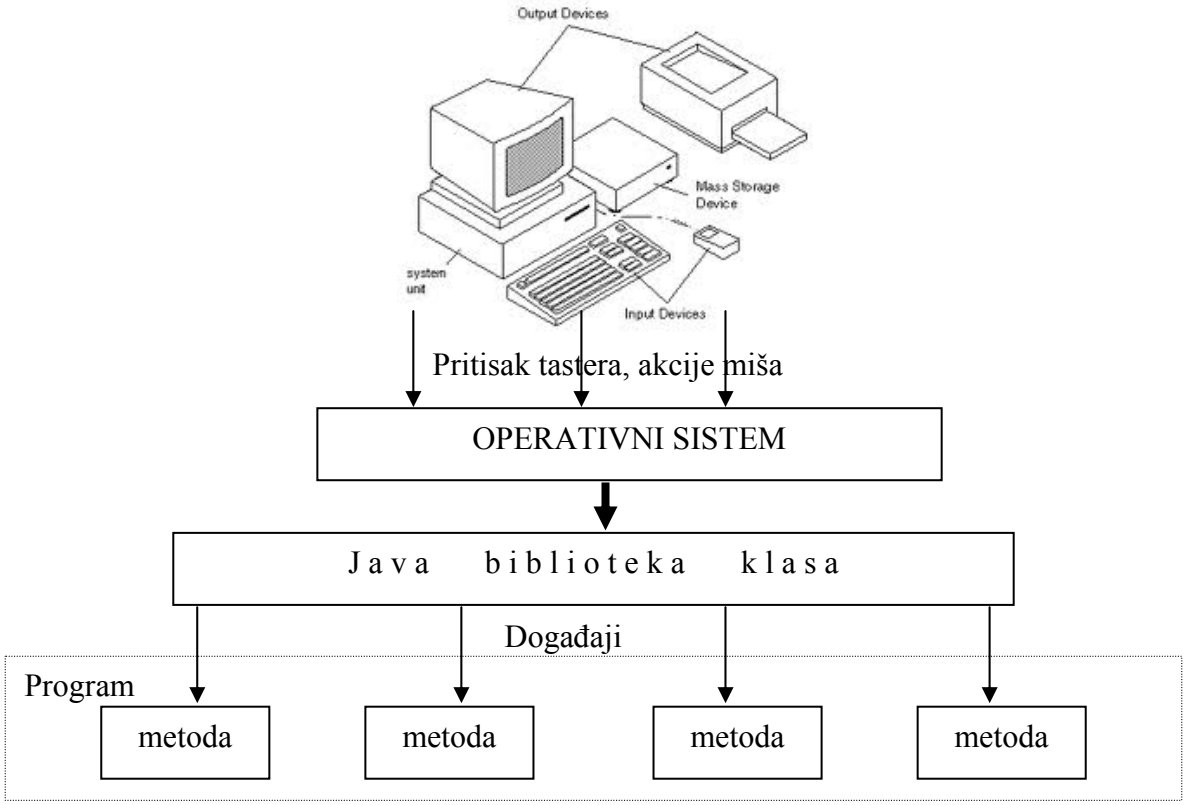
1) Unutrašnje klase su vidljive u klasi najvišeg nivoa.
2) Unutrašnje klase mogu da pristupaju promenjivim i metodama unutar dometa klasa najvišeg nivoa.
3) Pravila u vezi sa vidljivošću unutrašnje klase slična su pravilima u vezi sa vidljivošću atributa. Naziv unutrašnje klase nije vidljiv izvan klase (unutrašnju klasu je moguće instancirati i izvan klase u kojoj se nalazi). U kodu unutrašnje klase može se koristiti standardna konvecija u vezi sa davanjem naziva promenjivim i metodama.

Posebnu vrstu unutrašnje klase predstavlja **anonimna unutrašnja klasa**. U situacijama kada nije potrebno da unutrašnja klasa poseduje ime može se koristiti anonimna unutrašnja klasa. Ova klasa najčešće nasleđuje neku drugu klasu ili implementira neki interfejs. Ona se definiše u izrazu **new** kao deo naredbe. Anonimna klasa ne može se naslediti ni implementirati:

```
public class FrameAnonymousInnerClass extends JFrame {

	public FrameAnonymousInnerClass() {
		Toolkit kit = Toolkit.getDefaultToolkit();
	    Dimension screenSize = kit.getScreenSize();
	    int screenHeight = screenSize.height;
	    int screenWidth = screenSize.width;
	    JPanel pan=new JPanel();
	    JButton btn1=new JButton("JButton 1");
	    btn1.addMouseListener(new MouseListener() {

	                @Override
	                public void mouseReleased(MouseEvent arg0) {
	                }

	                @Override
	                public void mousePressed(MouseEvent arg0) {
	                }

	                @Override
	                public void mouseExited(MouseEvent arg0) {
	                }

	                @Override
	                public void mouseEntered(MouseEvent arg0) {

	                }
	                @Override
	                public void mouseClicked(MouseEvent arg0) {
	                    System.out.println("Kliknuo neko na miša.
	                Pozvana metoda u anonimnoj unutrašnjoj klasi ");

	                }
	            });
	    pan.add(btn1);
	    add(pan,BorderLayout.CENTER);
	}
}
```

9.3. Programi vođeni događajima

Kada se koristi aplikacija zasnovana na prozorima, rad programa se pokreće u interakciji sa korisničkim interfejsom (GUI). Biranje stavke menija ili dugmadi korišćenjem miša ili preko tastature izaziva određene akcije unutar programa. Bilo koja od ovih akcija koje se iniciraju klikom miša ili pristiskom na taster tastature biva prvo prepoznata od strane operativnog sistema računara.

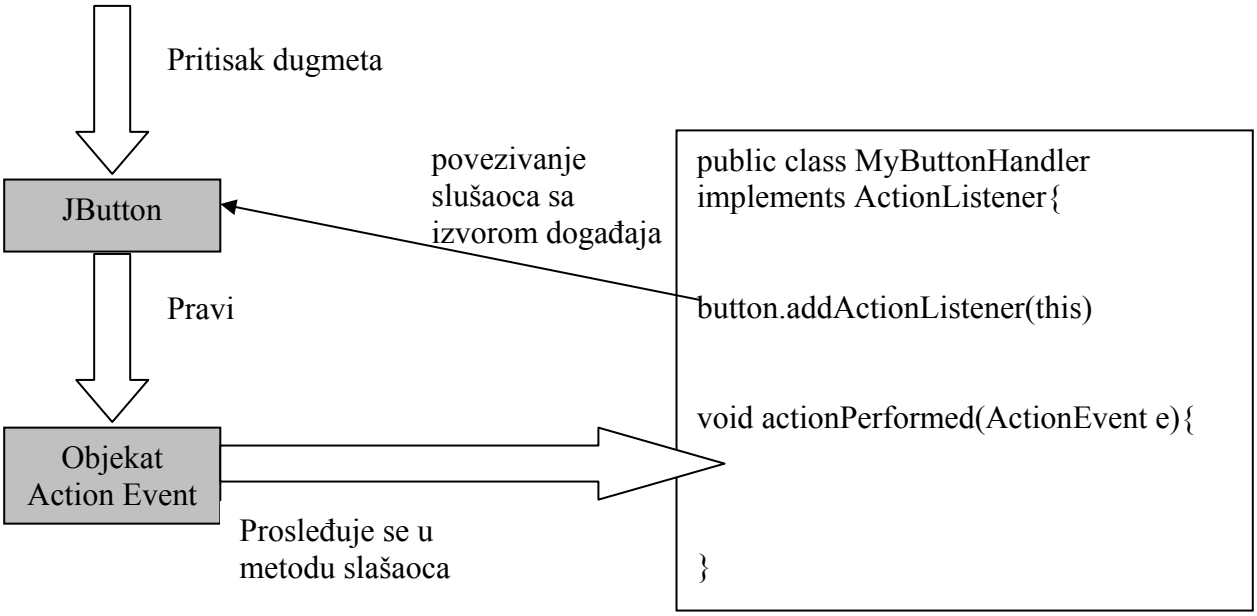


Za svaku akciju, operativni sistem određuje koji od programa koji trenutno rade na računaru treba da znaju da se akcija desila i prosleđuju akciju tom programu. Kada na primer pritisnete taster miša, operativni sistem to registruje i beleži poziciju kursora miša na ekranu. On zatim određuje koja aplikacija kontroliše prozor na kome je kursor bio kada je dugme pritisnuto i tom programu prenosi pritisak dugmeta miša. Signali koje program prihvata od operativnog sistema kao rezultat akcije korisnika nazivaju se **događaji**.

Program ne mora da odgovori ni na jedan određeni događaj. Ako se na primer samo pomeri miš, program ne mora da pozove određenu metodu koja bi reagovala na pomeranje miša. Program zasnovan na GUI-u naziva se program vođen događajima, jer je redosled događaja u programu stvoren interakcijom korisnika sa grafičkim interfejsom.

Da bi se rukovalo događajima u Javi potrebno je razumeti opšti koncept koji je u ovom programskom jeziku primenjen.

Pretpostavimo da na glavnom prozoru aplikacije postoji dugme. Kada korisnik klikne na dugme potrebno je da se izvrši neka **akcija**, odnosno potrebno je da se izvrši neki programski kod. **Događaj** koji se u posmatranom primeru javlja jeste pritisak miša na dugme. Događaj uvek ima objekat **izvora** i u ovom slučaju to je objekat JButton. Kada se pritisne dugme, stvoriće se novi objekat koji predstavlja i identifikuje ovaj događaj - to je uvek instanca neke klase odgovarajuće za vrstu događaja koji se desio. U ovom konkrektnom slučaju to je objekat klase ActionEvent. Ovaj objekat sadrži informaciju o događaju i njegovom izvoru. Ovaj objekat se kao **argument prosleđuje metodi** koja će na željeni način rukovati događajem.



Objekat događaja koji odgovara pritisku dugmeta, biće prosleđen bilo kom objektu slušaocu koji ima prethodnu registrovan interes za tu vrstu događaja. Prosleđivanje događaja slušaocu znači da izvor događaja poziva određeni metod u objektu slušaocu i prosleđuje mu objekat kao argument. Slušalac se definiše tako što se u proizvoljnoj klasi implementira interfejs slušaoca. Postoji veliki broj interfejsa slušalaca za opsluživanje različitih vrsta događaja. U slučaju pritiska na dugme u posmatranom primeru, klasa treba da implementira interfejs **ActionListener** da bi prihvatila događaj od dugmeta.

Sama implementacija interfejsa slašaoca nije dovoljna da bi se objekat slušaoca povezao sa izvorom događaja. Potrebno je i povezati slušaoca sa izvorom događaja sa kojim se želi raditi. Objekat slušaoca se registruje u izvoru pozivanjem određenog metoda u objektu izvora. U našem primeru, da bismo registrovali objekat slušalac da sluša događaje pritiska dugmeta pozivamo metod **addActionListener(**) objekta JButton i prosleđujemo objekat slušaoca kao argument tog metoda.

Pokrenuti aplikaciju sa argumentom 4. Pogledati i prokomentarisati klase JavaEventListener i FrameJavaEvent.

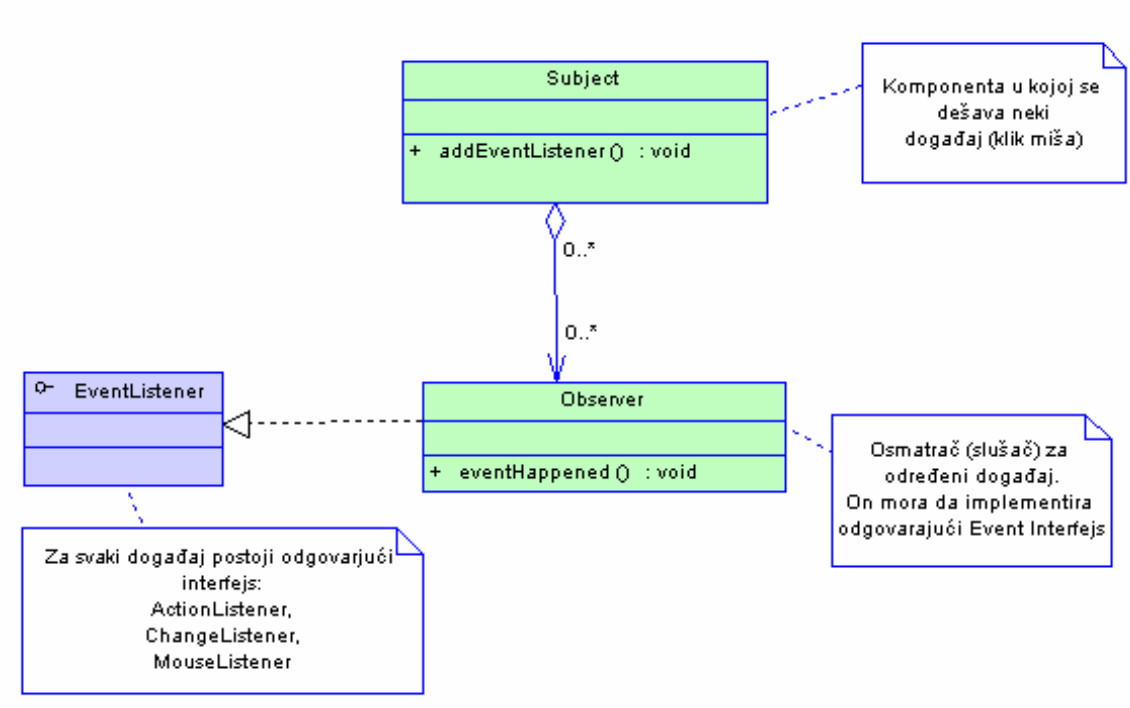
9.4. Obrada događaja u Javi

Događaj (Event) u Javi je obaveštenje da se dogodilo nešto zanimljivo (klik miša, selekcija, promena u komponenti, promena fokusa, pomeranje miša…) Obrada događaja predstavlja reakciju na to obaveštenje. Kada kreiramo korisnički interfejs nisu nam nam svi događaji interesantni, niti želimo da reagujemo na sve moguće događaje. Dakle, pri kreiranju GUI-a, odredićemo koji su nam događaji u radu sa komponentama od značaja i prilikom obaveštenje o tim događajima, izvršićemo obradu tih događaja na način na koji to nama odgovara.

Javin mehanizam obrade događaja je modeliran na osnovu *Observer pattern-a* (šablona osmatranja). Šablon osmatranja (*Observer pattern*) radi na sledeći način: mesto gde se događaj (klik miša, promena fokusa, selekcija...) dešava se naziva Subject (subjekat) i Subject u GUI-u predstavljaju komponente. Subject (komponenta) nudi skup događaja koji mogu biti posmatrani. Mogućnost posmatranja je objavljena obezbeđivanjem metode za registraciju za navedeni događaj. Ove metode se imenuju kao :

addActionListener(), addChangeListener(), addItemListener, addFocusListener()....
dakle uopšteno gledajući kao *addDogađajListener()*.

Ukoliko nešto sebe registruje kao slušača (listener) za *Subject*, to nešto se zove *Observer*(osmatrač). Dakle, kada se desi određeni događaj u Subjectu , obaveštavaju se svi registrovani *Observeri* za taj događaj.



U ovom primeru, Subject predstavlja JButton , dakle mesto gde se dešava određeni događaj. On nudi mogućnost osmatranja određenih događaja preko metoda za registraciju istih. U primeru je iskorišćena metoda za registraciju : addActionListener. Observer(osmatrač, slušač) u primeru je sama klasa Beeper koja implementira ActionListener.

```
public class Beeper extends JFrame implements ActionListener{

        public Beeper() {
                JButton btn=new JButton("Button 1");
                btn.addActionListener(this);
                getContentPane().add(btn,BorderLayout.CENTER);

        }
        public void actionPerformed(ActionEvent arg0) {
                java.awt.Toolkit.getDefaultToolkit().beep();

        }
        public static void main(String[] args) {
                 Beeper b=new  Beeper();

                b.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
               b.setSize(200,200);
                b.setVisible(true);
        }



}
```

Osmatranje događaja komponente nije ograničeno na jednog slušača. U većini slučajeva može se pridružiti neograničen broj slušača za isti događaj. Ako se radi sa više osmatrača za jedan subjekat, važno je znati da redosled po kome su oni dodati subjektu nema značaja za redosled po kome se oni obaveštavaju.

Slušač za akciju može se kreirati i putem anonimne unutrašnje klase. Kod anonimnih unutrašnjih klasa ne moramo da se brinemo o njihovom imenovanju, jer njih nećemo koristiti nigde izvan spoljašnje klase.

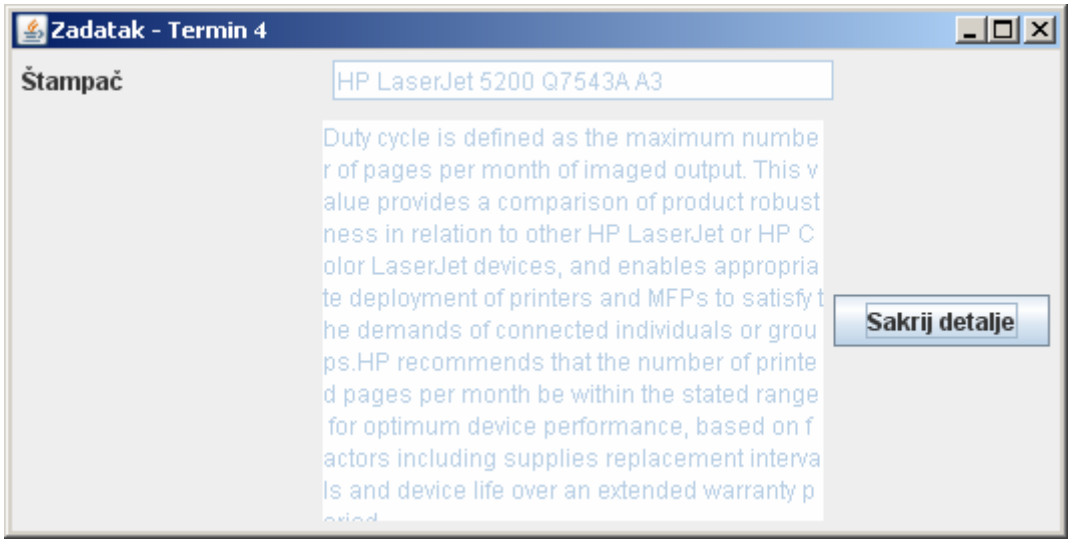
Rezultat ovog primera je isti kao rezultat iz prethodnog primera, samo što se sada slušač događaja registruje kao unutrašnja anonimna klasa a ne kao klasa koja implementira interfejs.

```
public class Beeper extends JFrame {

	public Beeper() {
		JButton btn=new JButton("Button 1");

		getContentPane().add(btn,BorderLayout.CENTER);
		btn.addActionListener(new ActionListener(){

			public void actionPerformed(ActionEvent arg0) {
				Toolkit.getDefaultToolkit().beep();

			}

		});



	}

	public static void main(String[] args) {
			Beeper b=new  Beeper();
	b.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
	b.setSize(200,200);
	b.setVisible(true);
	}



}
```

9.5. Zadatak

Realizovati aplikaciju kao na slici:

koja nakon pritiska na dugme „Prikaži detalje” ima izgled:



Probati rešenje realizovati sa jednom implementacijom ActionListener-a, a ne sa 2 anonimne unutrašnje klase.