

13 Događajima upravljano, OO i GUI programiranje

Kod konvencionalnog programiranja, sekvenca operacija u nekoj aplikaciji određena je centralnim programskim modulom – glavnim programom (Main program). Kod događajima upravljano programiranju (event-driven programming) sekvenca operacija u aplikaciji određena je korisnikovom interakcijom putem grafičkog interfejsa (formi, menija, dugmića, itd).

Program za događajima upravljano aplikacijom ostaje u pozadini sve dok se ne desi neki događaj, kao na primer:


- Kada se promeni vrednost u nekom polju forme aktivira se program za verifikaciju ispravnosti unetog podataka.
- Kada korisnik pritisne neko dugme i time označi da je završen unos podataka koim se menja stanje u skladištu, aktivira se procedura za ažuriranje skladišta.

Događajima upravljano programiranje, grafički korisnički interfejs (GUI) i objektno-orijentisano programiranje su međusobno povezani jer su forme i grafički interfejs objekti (boksovi, dugmići, i sl.) predstavljaju skelet čitave aplikacije.

Primer konvencionalne (konzolne) aplikacije u C#

```
using System;
using System.Collections.Generic;
using System.Text;

namespace prviprojekat
{
    class Program
    {
        static void Main(string[] args)
        {
            string naslov = "POZDRAV SVIMA";
            Console.WriteLine(naslov);
            Console.ReadKey();
        }
    }
}
```



U gornjem primeru vidi se kako glavni program (Main) upravlja odvijanjem aplikacije: najpre se deklarira varijabla `naslov` i dodeljuje joj se vrednost "POZDRAV SVIMA", zatim se poziva procedura `Console.WriteLine` kojom se ispisuje na ekranu vrednost varijable `naslov`, i na kraju se procedurom `Console.ReadKey` čeka da korisnik pritisne neki taster na tastaturi pa da aplikacija završi rad.

Sledeći primer prikazuje jednu događajima vođenu aplikaciju u C#.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

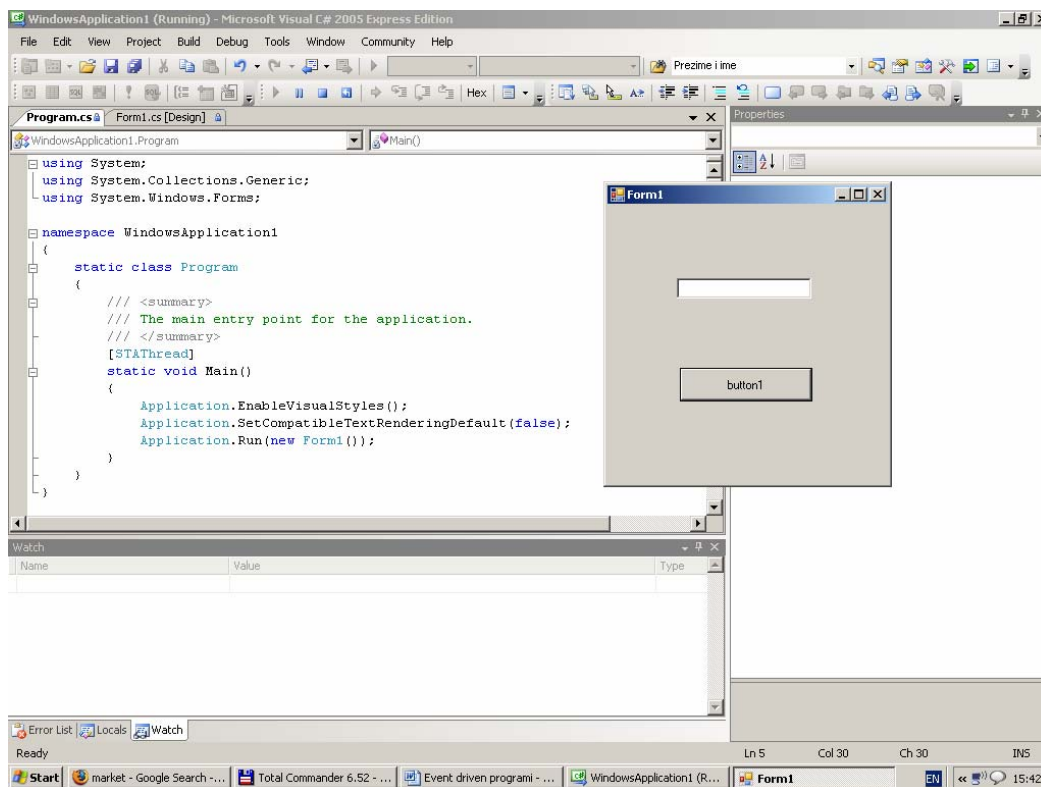
namespace WindowsApplication1
{
    static class Program
    {
```

```

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }
}

```

Ovde se, za razliku od prethodnog primera, u glavnom programu samo aktivira jedan objekat tipa forme, a sva dalje događanja u aplikaciji zavisice od interakcije korisnika i objekata prikazanih u formi. Jedan jednostavan primer forme koja sadrži samo dva objekta (button1 i tekst boks) prikazan je na sledećoj slici:



Programski segment kojim se definiše upravljanje događajima sa formom Form1 iz gornjeg primera može da, na primer, glasi ovako:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form

```

```

{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        Random r = new Random();
        textBox1.BackColor =
        Color.FromArgb(r.Next(0,255), r.Next(0,255), r.Next(0,255));
    }
}

```

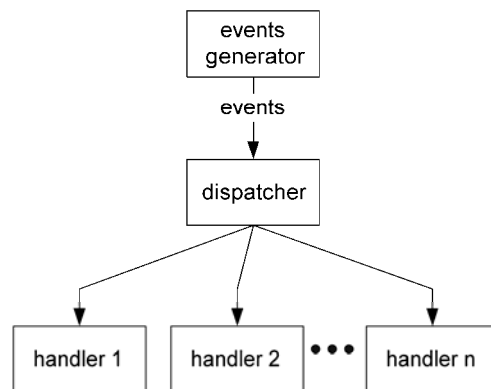
Ovde se može uočiti procedura **button1_Click** kojom se definiše šta će biti urađeno kada korisnik pritisne dugme `button1`. U ovom slučaju svaki pritisak na dugme `button1`, obojiće tekst boks drugom (slučajno odabranom) bokom.

Sada, kada smo se upoznali sa osnovnom razlikom između konvencionalnih (konzolnih) i događajima upravljanim (windows) aplikacijama, možemo se malo detaljnije pozabaviti mehanizmom izvršavanja događajima upravljanim programima (aplikacijama).

Očekivanje (listening) i opsluživanje (handling) događaja (events)

Kada kreira jednu događajima vođenu aplikaciju, programer zapravo kreira niz malih programa koje se nazivaju opslužiocima događajima (event handlers), koji se dodeljuju događajima koje pokreće neki od objekata iz aplikacije. Ovi mali programi, opslužioci, definišu se u modulu kojim se opisuje forme (vidi gornji primer).

Proces čekanja i opsluživanja događaja može biti ilustrovan sledećom slikom:



Gornji proces možemo prikazati i pseudokodom kako sledi:

```

while(true) {
    if (događaj == krajAplikacije) završi;

```

```

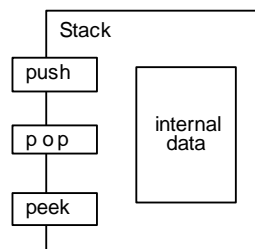
if (dogadaĳ == X) pozvatiOdgovarajućuProceduruX;

if (dogadaĳ == Y) pozvatiOdgovarajućuProceduruY;
...
if (dogadaĳ == ...) pozvatiOdgovarajućuProceduru...;
}

```

Objekti

Tokom 1990-tih godina objektno-orjentisana tehnologija je postepeno prevladala strukturalnu metodologiju 1970-tih i 1980-tih godina. U softversku metodologiju su uvedeni novi dijagrami kojima su opisivani objekti, različiti od dijagrama koje smo razmatrali u poglavlju 10 (blok, N/S i JSD dijagrami). U to vreme, korišćeni su takozvani *objektni dijagrami*, kao ovaj prikazan na sledećoj slici:



Objektni dijagram za STACK

U ovom dijagramu STACK je tip objekta. Umesto reči tip koju smo koristili kod varijabli, kod objekata se koristi reč klasa (class). Push, pop i peek su takozvane metode. Da bi koristili Stack klasu, potrebno je da kreirate stack objekat a onda da koristite metode kojima se obavljaju operacije nad objektima. Na primer:

```

# create a stack object by instantiating the Stack class
myStack = new Stack()

myStack.push("abc")
myStack.push("xyz")

print myStack.pop() # prints "xyz"
print myStack.peek() # prints "abc"

```

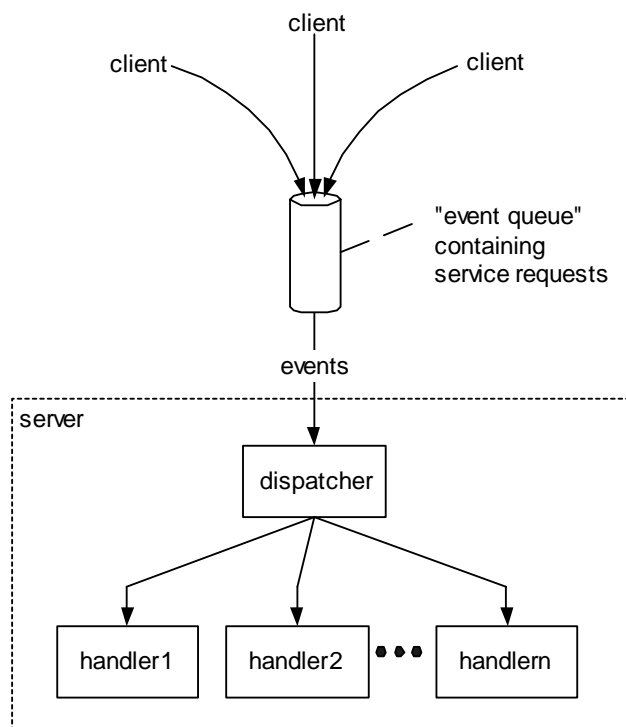
U gornjem primeru se naredbom `myStack = new Stack()` kreira jedan novi objekat sa imenom `myStack`, zatim se naredbama `myStack.push("abc")` i `myStack.push("xyz")` na stek postavljaju dva stringa ("abc" i "xyz"), a zatim se naredbama `print myStack.pop()` i `print myStack.peek()` štampaju isti stringovi obrnutim redom (jer je stek, setite se, LIFO lista).

Možemo da uočimo sličnost između OO programiranja i event-driven programiranja, koja se ogleda u tome da metode iz OO programa veoma podsećaju na opslužioce događaja (event handlers).

Klijent-server arhitektura

Još jedan poznati slučaj gde se pojavljuje opsluživanje događaja jeste takozvana klijent-server arhitektura. **Server** je neki hardverski ili softverski modul koji pruža neku uslugu (service) **klijentima**. Posao servera je da čeka na zahtev za uslugu (*service requests*) od klijenata, da odgovara na te zahteve pružanjem zahtevane usluge, a onda da nastavi čekanje na nove zahteve. Poznati primeri servera su: serveri za štampanje, fajl serveri, database serveri, aplikacioni serveri, web serveri. Na primer, kad god posetite neku novu web adresu pomoću vašeg brauzera, vaš brauzer (koji je u tom slučaju klijent) šalje zahtev web serveru na koji ovaj odgovara slanjem tražene web stranice.

Ovaj proces traženja i pružanja usluge u interakciji klijent-server može biti ilustrovan kao na sledećoj slici.

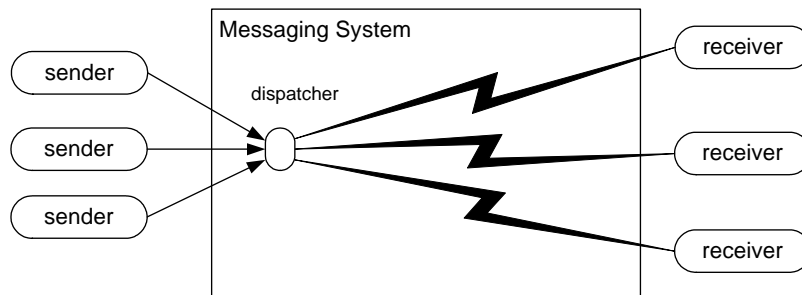


Klijent-server arhitektura

Sistemi za razmenu poruka (Messaging Systems)

Sistemi za razmenu poruka predstavljaju ekstremni slučaj opslužioca događaja (handler-a). Svrha sistema za razmenu poruka je da preuzme događaj (poruku) od generatora događaja (pošiljaoca) i prosledi ga opslužiocu (primaocu), pa je tako uloga dispečera jednostavna. Jedan poznat primer sistema za razmenu poruka su pošte. Pošiljalac predaje poruku (pismo ili paket) na poštu (sistemu za razmenu poruka). Pošta prepoznaje adresu primaoca i transportuje pošiljki na tu adresu.

Sledeća slika ilustruje ovaj proces_



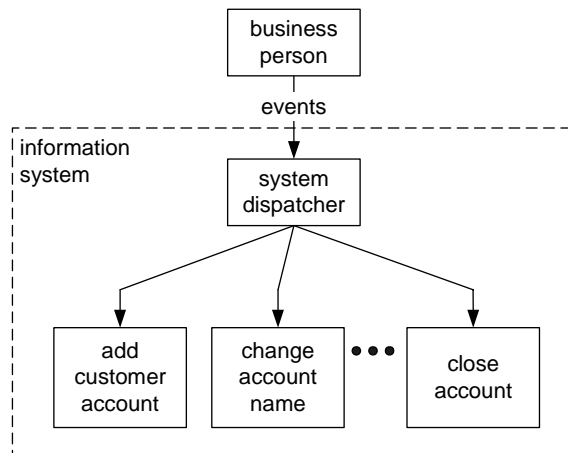
E-mail sistem vrši istu ovakvu funkciju, kao i klasična pošta, jedina razlika je što su poruke u elektronskoj a ne u fizičkoj formi.

Objektno-orjentisano događajima upravljano programiranje

Sada kada smo u stanju da vidimo potpunu sliku kako se koncept opsluživanja manifestuje u kompjuterskim sistemima, možemo da se pozabavimo I pitanjem kako to sve funkcioniše u konkretnim programima.

Posmatrajmo jedan poslovni sistem koji posluje sa mušterijama. Prirodno je da vlasnici tog biznisa žele da imaju na raspolaganju informacioni sistem u kojem mogu da memorišu, pretražuju i ažuriraju informacije o računima svojih mušterija. Oni žele sistem koji može da opsluži različite zahteve: zahtev za otvaranjem računa nove mušterije, promenu naziva nekog postojećeg računa, zatvaranje postojećeg računa itd. Takav sistem, znači mora da ima opslužioce (handler-e) za sve ove događaje.

Sledeća slika ilustruje arhitekturu jednog takvog informacionog sistema.



Pre pojave objektno orjentisanog programiranja, gornji opslužioci bi bili programski implementirani kao niz potprograma (subroutines).

Tako bi dispečerski deo programa mogao da izgleda kao što je prikazano u sležećem pseudokodu:

A

```

get eventType from the input stream

if  eventType == EndOfEventStream :
    quit # break out of event loop

if  eventType == "A" : call addCustomerAccount()
elif eventType == "U" : call setCustomerName()
elif eventType == "C" : call closeCustomerAccount()
... and so on ...
  
```

potprogrami kojima se opslužuju zahtevi mogli bi da izgledaju ovako:

```

subroutine addCustomerAccount():
    get customerName           # from data-entry screen
    get next available accountNumber # generated by the system
    accountStatus = OPEN
    # insert a new row into the account table
    SQL: insert into account
        values (accountNumber, customerName, accountStatus)
subroutine setCustomerName():
    get accountNumber         # from data-entry screen
    get customerName          # from data-entry screen
    # update row in account table
    SQL: update account
        set customer_name = customerName
        where account_num = accountNumber
subroutine closeCustomerAccount():
    get accountNumber         # from data-entry screen
    # update row in account table
    SQL: update account
        set status = CLOSED
        where account_num = accountNumber
  
```

Danas, korišćenjem OO tehnologije, opsluživanje događaja se implementira metodama za objekte. Tako sada, programski kod kojim se postiže isti efekat kao u predhodnom primeru može da izgleda ovako:

```
get eventType from the input stream

if  eventType == "end of event stream":
    quit # break out of event loop

if  eventType == "A" :
    get customerName          # from data-entry screen
    get next available accountNumber # from the system

    # create an account object
    account = new Account(accountNumber, customerName)
    account.persist()

elif eventType == "U" :
    get accountNumber        # from data-entry screen
    get customerName         # from data-entry screen

    # create an account object & load it from the database
    account = new Account(accountNumber)
    account.load()

    # update the customer name and persist the account
    account.setCustomerName(customerName)
    account.persist()

elif eventType == "C" :
    get accountNumber        # from data-entry screen

    # create an account object & load it from the database
    account = new Account(accountNumber)
    account.load()

    # close the account and persist it to the database
    account.close()
    account.persist()

... and so on ...
```

A *Account* klasa, sa metodama koje funkcionišu kao handler-i, može da izgleda ovako:

```
class Account:

    # the class's constructor method
    def initialize(argAccountNumber, argCustomerName):
        self.accountNumber = argAccountNumber
        self.customerName = argCustomerName
        self.status = OPEN

    def setCustomerName(argCustomerName):
        self.customerName = argCustomerName

    def close():
        self.status = CLOSED

    def persist():
```

```
... code to persist an account object

def load():
    ... code to retrieve an account object from persistent storage
```

Grafički interfejsi (GUI programiranje)

Grafički interfejsi su, danas, najčešće korišćeni okviri (frameworks) za razvoj objektno-orijentisanih i događajima upravljanih aplikacija (programa).

GUI programiranje je teško iz više razloga.

Kao prvo, potrebno je uložiti dosta truda da se specificira izgled GUI-a na ekranu kompjutera. Sve te stvarčice koje se pojavljuju na ekranu – svako dugme, labela, meni, tekst boks, list boks, itd. – moraju biti definisane preko oblika, veličine, boje pozadine, fontova itd., mora se definisati njihova početna pozicija unutar forme, kako će se ponašati ako im se promeni veličina, kao i kao se uklapaju sa ostalim objektima definisanim na ekranu. Zato se kao pomoć pri dizajnu koriste IDE alati kojima se olakšava definisanje izgleda.

Drugo, GUI programiranje je teško jer postoji jako mnogo vrsta različitih događaja koje treba opslužiti. Skoro svaki element GUI-a – svako dugme, ček-boks, radio dugme, polje za unos podataka, list boks, tekst boks, meni, itd. – predstavljaju generatore jednog ili više događaja. Osim toga i hardverski uređaji mogu da generišu događaje. Recimo, miš može da generiše pritisak (klik) na levo dugme, desno dugme, dvisotruki klik na levo i desno dugme, itd. Slično svaki pritisak na tastaturi slova, broja ili funkcijske tipke može generisati događaje. Za sve takve događaje moraju biti programirani opslužioc (handleri).

Integrisana razvojna okruženja kao što je Microsoft Visual Studio značajno olakšavaju razvoj GUI aplikacija jer u sebi sadrže takozvane form dizajnere kojima se olakšava definisanje svih karakteristika (properties) objekata koji se koriste u interfejsu. Međutim, hendleri moraju i dalje biti programirani na način sličan konvencionalnim programima.

Pitanja

- 1) Kakva je razlika između konvencionalnog (proceduralnog) i događajima upravljanih programiranja?
- 2) Kakvu ulogu ima glavni program (Main) u konvencionalnom, a kakvu u event-driven programiranju.
- 3) Kako se u Microsoft Visual Studio razvojnom okruženju označavaju konvencionalne, a kako event-driven aplikacije?
- 4) Koje sve komponente kompjuterskog sistema mogu da generišu događaje?
- 5) Navedite primer jednog softverski generisanog događaja.
- 6) Navedite jedan primer hardverski generisanog događaja?

- 7) Šta je to opsluživanje događaja?
- 8) Šta je server?
- 9) Šta je klijent?
- 10) Šta je dispečer događaja?
- 11) Šta su objekti (klase)?
- 12) Od čega se sastoje objekti (klase)?
- 13) Šta je OO programiranje?
- 14) Šta je GUI programiranje?
- 15) Kakva veza postoji između događajima upravljanim, OO i GUI programiranja?