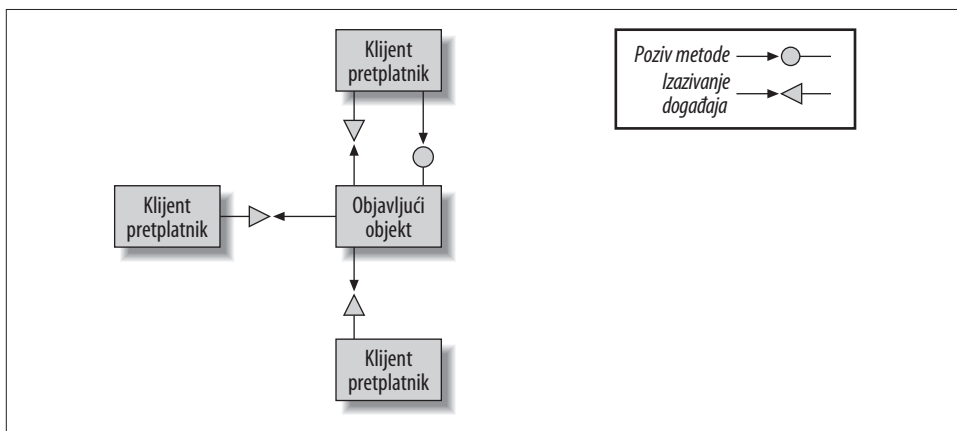


# Događaji

U komponentno orijentiranom programu objekt pruža usluge klijentima tako što im dopušta da pozivaju metode i postavljaju svojstva objekta. Ali što ako klijent (ili više klijenata, kao na slici 6-1) želi biti obaviješten o događaju koji se zbio na strani objekta? Ta situacija je vrlo česta i gotovo svaka aplikacija se oslanja na neki mehanizam pretplate na događaje i objavljivanja događaja.



Slika 6-1. Objavljući objekt može izazivati događaje na više klijenata pretplatnika

Budući da događaji nisu ništa više od poziva metoda, ne postoji ništa posebno u izazivanju događaja, te sam zato odlučio ne smatrati podršku za događaje temeljnim načelom komponentno orijentiranog programiranja. Međutim, to ne znači da korištena komponentna tehnologija ne bi trebala pokušati olakšati zadaću pretplaćivanja na događaje i njihovog objavljivanja. Nije iznenađujuće da .NET podrška za događaje automatizira taj proces koliko god je moguće. Ovo poglavlje počinje objašnjenjem najvažnijih pojmova .NET-ove podrške za događaje te zatim pruža skup praktičkih smjernica za učinkovito upravljanje događajima i proširivanje osnovne podrške za događaje.

# Događaji temeljeni na delegatima

Prije nego opišem .NET podršku za događaje, evo nekoliko termina. Objekt koji objavljuje događaj zove se *izvor* (engl. *source*) ili *objavljiivač* (engl. *publisher*), a bilo koja stranka zainteresirana za događaj zove se *odvod* (engl. *sink*) ili *pretplatnik* (engl. *subscriber*). Obavijesti o događajima su u obliku objavljiivača koji poziva metode na pretplatnicima. Objavljivanje događaja naziva se i *izazivanjem* (engl. *firing*) događaja. .NET nudi izvornu podršku za događaje osiguravanjem specijaliziranih CLR tipova i implementacija osnovne klase. .NET definira standardni mehanizam za uspostavljanje i uklanjanje veze između izvora i odvoda, standardni i sažeti način za izazivanje događaja, te gotovu implementaciju popisa odvoda.

.NET podrška za događaje oslanja se na *delegate* (engl. *delegates*). Konceptualno, delegat nije ništa više od reference metode sigurne za tipove – možete ga smatrati pokazivačem funkcije C sigurnim za tipove ili objektom funkcije u C++. Kao što ime sugerira, delegat dopušta da delegirate čin pozivanja metode nekom drugom. Delegat može pozvati statičke metode i metode instance. Razmotrite, na primjer, delegat `NumberChangedEventHandler` definiran kao:

```
public delegate void NumberChangedEventHandler(int number);
```

On se može koristiti za pozivanje bilo koje metode sa odgovarajućim potpisom (povratni tip `void` i jedan parametar `int`). Ime delegata, imena ciljnih metoda i imena parametara tih metoda nisu važna. Jedini preduvjet je da metode koje se pozivaju imaju točno onaj potpis (tj. iste tipove) koji delegat očekuje. Delegatu ćete obično dati ime koje ima smisla, kako biste prenijeli njegovu svrhu čitaocima svog koda. Na primjer, ime `NumberChangedEventHandler` indicira da se taj delegat koristi za objavljivanje događaja koji obavještava pretplatnike da se promijenila vrijednost određenog broja koji oni nadziru.

U slučaju upravo opisanog događaja, izvor događaja ima javnu varijablu članicu tipa delegata:

```
public delegate void NumberChangedEventHandler(int number);
```

```
public class MyPublisher
{
    public NumberChangedEventHandler NumberChanged;
    /* Druge metode i članovi */
}
```

Pretplatnik na događaj mora implementirati metodu sa traženim potpisom:

```
public class MySubscriber
{
    public void OnNumberChanged(int number)
    {
        string message = "New value is " + number;
        MessageBox.Show(message, "MySubscriber");
    }
}
```

Prevoditelj zamjenjuje definiciju tipa delegata sa naprednom klasom koja osigurava implementaciju popisa odvoda. Stvorena klasa delegata izvodi iz apstraktne klase Delegate, pokazane u primjeru 6-1.

*Primjer 6-1. Djelomična definicija klase Delegate*

```
public abstract class Delegate : //Popis sucelja
{
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public object DynamicInvoke(object[] args);
    public virtual Delegate[] GetInvocationList( );
    //Ostali clanovi
}
```

Za upravljanje popisom ciljnih metoda možete koristiti operatore =, += i -=. Taj popis je zapravo popis objekata delegata i svaki referencira jednu ciljnu metodu. Operator += dodaje novog pretplatnika (zapravo, samo novu ciljnu metodu omotanu u delegat) na kraj popisa. Kako biste dodali novi cilj, morate stvoriti novi objekt delegata omotan oko ciljne metode. Operator -= uklanja ciljnu metodu sa popisa (stvarajući novi objekt delegata omotan oko ciljne metode ili koristeći postojećeg delegata koji je namijenjen za tu metodu). Operator = može inicijalizirati popis, obično delegatom koji pokazuje na jedan cilj. Prevoditelj pretvara upotrebu operatora u odgovarajuće pozive za statičke metode klase Delegate, kao što su Combine( ) ili Remove( ). Kada želite izazvati događaj, jednostavno pozovite delegata prosljeđujući parametre. To uzrokuje iteriranje delegata po internom popisu ciljeva i pozivanje svake ciljne metode sa tim parametrima.

Primjer 6-2 pokazuje kako dodati dva pretplatnika na popis odvoda delegata, kako izazvati događaj, te kako ukloniti pretplatnika sa popisa.

*Primjer 6-2. Upotreba delegata za upravljanje pretplatom na događaje i objavljivanjem događaja*

```
MyPublisher publisher = new MyPublisher( );
MySubscriber subscriber1 = new MySubscriber( );
MySubscriber subscriber2 = new MySubscriber( );

//Dodavanje pretplata:
publisher.NumberChanged += new NumberChangedEventHandler
    (subscriber1.OnNumberChanged);
publisher.NumberChanged += new NumberChangedEventHandler
    (subscriber2.OnNumberChanged);

//Objavljivanje događaja:
publisher.NumberChanged(3);

//Uklanjanje pretplate:
publisher.NumberChanged -= new NumberChangedEventHandler
    (subscriber2.OnNumberChanged);
```

Sve što kod u primjeru 6-2 radi jest da delegira čin pozivanja metoda pretplatnika delegatu `NumberChanged`. Obratite pozornost na činjenicu da ciljnu metodu istog pretplatnika možete dodati više puta, kao u:

```
publisher.NumberChanged += new NumberChangedEventHandler
    (subscriber1.OnNumberChanged);
publisher.NumberChanged += new NumberChangedEventHandler
    (subscriber1.OnNumberChanged);
```

ili:

```
NumberChangedEventHandler del;
del = new NumberChangedEventHandler(subscriber1.OnNumberChanged);
publisher.NumberChanged += del;
publisher.NumberChanged += del;
```

Delegat zatim jednostavno poziva tog pretplatnika odgovarajući broj puta. Kada uklonite ciljnu metodu sa popisa, a ona se nalazi na popisu više puta, prva nađena (tj. ona najbliža početku popisa) bit će uklonjena.



Delegati se u .NET-u često koriste, ne samo kao dosljedan način upravljanja događajima već i za druge zadaće, kao što su sinkrono pozivanje metode (opisano u sedmom poglavlju) i stvaranje novih dretvi (opisano u osmom poglavlju).

## Izvođenje delegata

U C# 2.0 prevoditelj može zaključiti koji tip delegata da instancira kada dodaje ili uklanja ciljnu metodu. Umjesto da eksplicitno instancirate novi objekt delegata, možete napraviti izravno pridruživanje imena metode u varijablu delegata, bez da je prvo omotate objektom delegata. Ja to svojstvo zovem *izvođenje delegata* (engl. *delegate inference*). Primjer 6-3 pokazuje isti kod kao i primjer 6-2, ali ovaj put uz upotrebu izvođenja delegata.

*Primjer 6-3. Upotreba izvođenja delegata*

```
MyPublisher publisher = new MyPublisher( );
MySubscriber subscriber1 = new MySubscriber( );
MySubscriber subscriber2 = new MySubscriber( );

//Dodavanje pretplata:
publisher.NumberChanged += subscriber1.OnNumberChanged;
publisher.NumberChanged += subscriber2.OnNumberChanged;

//Objavljivanje događaja:
publisher.NumberChanged(3);

//Uklanjanje pretplata:
publisher.NumberChanged -= subscriber2.OnNumberChanged;
```

Koristeći izvođenje delegata možete proslijediti ime metode na bilo koju metodu koja očekuje delegata. Na primjer:

```

delegate void SomeDelegate( );

class SomeClass
{
    public void SomeMethod( )

    {...}
    public void InvokeDelegate(SomeDelegate del)

    {

        del( );

    }

}

SomeClass obj = new SomeClass( );
obj.InvokeDelegate(obj.SomeMethod);

```

Kada pridružite ime metode delegatu, prevoditelj prvo zaključuje koji je tip delegata, a zatim verificira da postoji metoda s tim imenom te da njezin potpis odgovara potpisu tipa delegata. Na kraju prevoditelj stvara novi objekt tog tipa delegata koji omotava metodu, te to pridružuje delegatu. Prevoditelj može zaključiti koji je tip delegata samo ako je taj tip posebni tip delegata – to jest, bilo što osim apstraktnog tipa Delegate. Izvođenje delegata čini kod čitkim i sažetim, a to je i stil pisanja koda korišten kroz cijelu ovu knjigu.



Samo C# 2.0 podržava izvođenje delegata. Ako vaš kod treba raditi na ranijim inačicama .NET-a, upotrijebite eksplicitno instanciranje delegata, kao u primjeru 6-2.

## Generički delegati

Uvođenje generika u .NET-u 2.0 otvorilo je cijeli novi skup mogućnosti za definiranje događaja i upravljanje njima. Kao što ćete vidjeti kasnije u ovom poglavlju, sposobnost definiranja generičkih delegata znači da ćete rijetko morati definirati nove delegate za rad s događajima. Delegat definiran u klasi može iskoristiti generički parametar tipa te klase. Na primjer:

```

public class MyClass<T>
{
    public delegate void GenericEventHandler(T t);
    public void SomeMethod(T t)
    {...}
}

```

Kada specificirate parametar tipa za sadržavajuću klasu, to će utjecati i na delegata:

```

MyClass<int>.GenericEventHandler del;
MyClass<int> obj = new MyClass<int>( );
del = obj.SomeMethod;
del(3);

```

Obratite pozornost na činjenicu da prevoditelj može zaključiti točan tipa delegata za instanciranje , uključujući i upotrebu točnog generičkog parametra tipa. Dakle, ovo pridruživanje:

```
del = obj.SomeMethod;
```

zapravo se tijekom prijevoda pretvara u ovo pridruživanje:

```
del = new MyClass<int>.GenericEventHandler(obj.SomeMethod);
```

Kao i klase, sučelja, strukture i metode, i delegati mogu definirati generičke parametre tipa:

```
public class MyClass<T>
{
    public delegate void GenericEventHandler<X>(T t, X x);
}
```

Delegati definirani izvan dosega klase također mogu koristiti generičke parametre tipa. U tom slučaju, morate osigurati specifične tipove za delegata kada ga deklarirate i instancirate:

```
public delegate void GenericEventHandler<T>(T t);
```

```
public class MyClass
{
    public void SomeMethod(int number)
    {...}
}
```

```
GenericEventHandler<int> del;
```

```
MyClass obj = new MyClass( );
del = obj.SomeMethod;
del(3);
```

I, naravno, delegat može definirati ograničenja koja će pratiti generičke parametre tipa:

```
public delegate void GenericEventHandler<T>(T t) where T : IComparable<T>;
```

Ograničenja na razini delegata provode se samo na korisničkoj strani, kada se deklarira varijabla delegata i kada se instancira objekt delegata, slično bilo kojem drugom ograničenju u dosegu tipova i metoda.

## Ključna riječ event

Upotreba sirovih delegata za upravljanje događajima je jednostavna, ali ima manu: klasa objavljiivača trebala bi izložiti delegata kao javnu varijablu članicu tako da bilo koja stranka može dodavati pretplatnike na popis. Međutim, izlaganje delegata kao javnog člana dopušta bilo kome da mu pristupa i objavljuje događaj, čak i ako se nikakav događaj ne zbiva na strani objekta. Kako bi riješio tu manu, C# rafinira tip delegata korištenog za pretplatu na događaj i obavijest o događaju, koristeći rezerviranu riječ event. Kada definirate varijablu članicu delegata kao događaj, čak i ako je ta članica

javna, samo klasa objavljivača može izazvati događaj (iako bilo tko može dodavati ciljne metode popisu delegata). Tada ovisi o diskreciji programera klase izvora događaja hoće li pružiti javnu metodu za izazivanje događaja:

```
public delegate void NumberChangedEventHandler(int number);

public class MyPublisher
{
    public event NumberChangedEventHandler NumberChanged;
    public void FireEvent(int number)
    {
        NumberChanged(number);
    }
    /* Ostale metode i članovi */
}
```

Kod koji povezuje pretplatnike sa izvorom ostaje isti kao u primjeru 6-2 ili primjeru 6-3 (koristi operatore += i -=). Upotreba događaja umjesto sirovih delegata promovira i labavije vezanje između izvora i pretplatnika, jer je poslovna logika na strani izvora koja pokreće izazivanje događaja sakrivena od pretplatnika.



Kada utipkate operator += kako biste dodali ciljnu metodu delegatu u Visual Studiju 2005, IntelliSense vam, nakon što pritisnete tipku Tab, nudi dodavanje novog delegata odgovarajućeg tipa. Ako vam se ne sviđa podrazumijevano ime ciljne metode, jednostavno utipkajte drugo ime. Ako ciljna metoda ne postoji, IntelliSense vam dopušta da stvorite metodu za obradu tog imena tako što ćete još jednom pritisnuti Tab. Ta IntelliSense podrška radi samo sa delegatima koji su definirani kao događaji (ne i sa običnim delegatima). Ne postoji IntelliSense podrška za uklanjanje pretplate.

## Događaji u Visual Basicu 2005

Semantika operacija za rad s događajima u Visual Basicu 2005 je potpuno ista kao ona u C#. Međutim, sintaksa je dovoljno različita da zasluži nekoliko riječi i određenu količinu primjera koda. Visual Basic 2005 ne osigurava preklopljene operatore za dodavanje i uklanjanje metoda za rad s događajima. Umjesto toga, on koristi rezervirane riječi AddHandler i RemoveHandler. Operator AddressOf koristi se za dobivanje adrese metode za rad s događajima. Kako biste izazvali događaj u Visual Basicu 2005 trebate upotrijebiti operator RaiseEvent, umjesto da izravno delegirate, kao u C#.

Primjer 6-4 koristi Visual Basic 2005 kako bi implementirao događaj NumberChanged. Kod je usporediv sa kodom u primjeru 6-2, ali umjesto sirovog delegata koristi događaj.

*Primjer 6-4. .NET događaji u Visual Basicu 2005*

```
Public Delegate Sub NumberChangedEventHandler(ByVal number As Integer)

Public Class MyPublisher
    Public Event NumberChanged As NumberChangedEventHandler
    Public Sub FireEvent(ByVal number As Integer)
```

Primjer 6-4. .NET događaji u Visual Basicu 2005 (nastavak)

```
        RaiseEvent NumberChanged(number)
    End Sub
End Class

Public Class MySubscriber
    Public Sub OnNumberChanged(ByVal number As Integer)
        Dim message As String
        message = "New value is " + number.ToString( )
        MessageBox.Show(message, "MySubscriber")
    End Sub
End Class

Dim publisher As MyPublisher = New MyPublisher( )
Dim subscriber1 As MySubscriber = New MySubscriber( )
Dim subscriber2 As MySubscriber = New MySubscriber( )

'Adding subscriptions:
AddHandler publisher.NumberChanged, AddressOf subscriber1.OnNumberChanged
AddHandler publisher.NumberChanged, AddressOf subscriber2.OnNumberChanged

publisher.FireEvent(3)

'Removing a subscription:
RemoveHandler publisher.NumberChanged, AddressOf subscriber2.OnNumberChanged
```

## Rad sa .NET događajima

Ovaj odjeljak objašnjava .NET smjernice za projektiranje događaja i razvojne prakse koje promoviraju labavo povezivanje između izvora događaja i pretplatnika, poboljšavaju dostupnost, prilagođavaju se postojećim konvencijama i općenito iskorištavaju .NET-ovu bogatu infrastrukturu za podršku za događaje. Još jedna tehnika vezana za događaje (asinkrono objavljivanje događaja) opisana je u sedmom poglavlju.

### Definiranje potpisa delegata

Iako, tehnički, deklaracija delegata može definirati bilo kakav potpis metode, u praksi bi delegat događaja trebalo prilagoditi određenim smjernicama.

Prvo, ciljne metode trebale bi imati povratni tip `void`. Na primjer, za događaj koji se bavi novom vrijednošću za broj, takav potpis bi mogao biti:

```
public delegate void NumberChangedEventHandler(int number);
```

Trebali biste koristiti povratni tip `void` jednostavno zato što nema smisla vratiti vrijednost izvoru događaja. Što bi izvor trebao učiniti sa tim vrijednostima? On nema pojma zašto se pretplatnik uopće i želi pretplatiti na događaj. Osim toga, klasa delegata skriva sam čin objavljivanja od izvora. Delegat iterira kroz interni popis odvoda (pretplatničkih objekata), pozivajući odgovarajuće metode. Povratne vrijednosti se ne

## Labavo povezani događaji u .NET-u

.NET podrška za događaje olakšava zadaću upravljanja događajima, te vas rješava tereta pisanja dosadnog koda za upravljanje popisom pretplatnika. Međutim, .NET događaji temeljeni na delegatima pate od nekoliko mana:

- Pretplatnik (ili klijent koji dodaje pretplatu) mora ponoviti kod za dodavanje pretplate za svaki objekt izvora od kojeg želi primati događaje. Ne postoji način za pretplaćivanje na tip događaja i dostavljanje klijenta pretplatniku, bez obzira na to tko je izvor.
- Pretplatnik nikako ne može filtrirati događaje koji se izazivaju (npr. da kaže: "Obavijesti me o događaju samo ako on ispunjava određeni uvjet").
- Pretplatnik mora imati način da dođe do objekta izvora kako bi se pretplatio na njega, što zauzvrat uvodi povezivanje između klijenata i objekata te povezivanje između individualnih klijenata.
- Izvor i pretplatnici imaju povezano trajanje. I izvor i pretplatnik moraju raditi u isto vrijeme. Ne postoji način da pretplatnik kaže .NET-u: „Ako bilo koji objekt izazove ovaj određeni događaj, molim te stvori instancu tog događaja i daj mi da radim s njom.“
- Nema lakog načina za obavljanje nepovezanog posla, gdje objekt izvora izazove događaj sa stroja izvan mreže i događaj se kasnije isporučuje klijentima pretplatnicima kada se stroj umreži. Obrnuto – da klijent koji radi na stroju izvan mreže kasnije primi događaje koji su izazvani dok je bio izvan mreže – također nije moguće.
- Uspostavljanje veza mora se učiniti programski. Nema administrativnog načina za uspostavljanje veza između izvora i pretplatnika.

Kako bi nadoknadio te nedostatke, .NET podržava zasebnu vrstu događaja, zvanu *labavo povezani događaji* (engl. *loosely coupled events*, LCE). Podrška za LCE nalazi se u imenskom prostoru `System.EnterpriseServices`. Iako LCE nisu temeljeni na delegatima, njihova upotreba je laka i jasna te nudi dodatne pogodnosti, kao što su povezivanje događaja sa transakcijama i sigurnost. .NET Enterprise Services izvan su raspona ove knjige, ali možete čitati o njima u mojoj knjizi *COM and .NET Component Services* (u izdanju O'Reilly Media). Labavo povezani događaji detaljno su opisani u poglavljima 9 i 10 te knjige.

prenose u kod izvora. Logika po kojoj trebate koristiti povratni tip `void` navodi vas i da izbjegavate izlazne parametre koji koriste modifikatore `ref` ili `out`, jer se izlazni parametri raznih pretplatnika ne prenose izvoru događaja.

Drugo, neki pretplatnici će vjerojatno htjeti primiti isti događaj iz više izvora. Budući da nema fleksibilnosti u osiguravanju toliko metoda koliko je i izvora, pretplatnik će htjeti ponuditi istu metodu većem broju izvora. Kako bi dopustio pretplatniku da razlikuje događaje koje su izazvali različiti izvori, potpis bi trebao sadržavati identitet

izvora. Najlakši način da se to učini bez oslanjanja na generike (opisano kasnije), jest da se doda parametar tipa `object`, zvan parametar *pošiljatelja* (engl. *sender*):

```
public delegate void NumberChangedEventHandler(object sender,int number);
```

Izvor događaja tada jednostavno proslijedi samog sebe kao pošiljatelja (koristeći `this` u `C#` ili `Me` u Visual Basicu 2005).

Napokon, definiranje argumenata događaja (kao što je `int number`) povezuje izvore događaja sa pretplatnicima, jer pretplatnik mora očekivati određeni skup argumenata. Ako budete htjeli promijeniti te argumente u budućnosti, takva će promjena utjecati na sve pretplatnike. Kako bi smanjio utjecaj promjene argumenata, .NET pruža kanonski spremnik argumenata događaja, klasu `EventArgs`, koju možete upotrijebiti umjesto određenog popisa argumenata. Definicija klase `EventArgs` je:

```
public class EventArgs
{
    public static readonly EventArgs Empty;
    static EventArgs( )
    {
        Empty = new EventArgs( );
    }
    public EventArgs( )
    {}
}
```

Umjesto određenog skupa argumenata, možete proslijediti objekt `EventArgs`:

```
public delegate void NumberChangedEventHandler(object sender,EventArgs eventArgs);
```

Ako izvor nema potrebe za argumentom, jednostavno prosljedite `EventArgs.Empty`, koristeći statički konstruktor i statičkog člana samo za čitanje `Empty`.

Ako događaj zahtijeva argumente, izvedite klasu iz `EventArgs`, kao što je `NumberChangedEventArgs`. Dodajte varijable članice, metode ili svojstva po potrebi, te prosljedite izvedenu klasu. Pretplatnik bi trebao pretvoriti `EventArgs` u zadanu klasu argumenta povezanu sa tim događajem (`NumberChangedEventArgs` u ovom primjeru) te pristupiti argumentima. Primjer 6-5 pokazuje upotrebu klase izvedene iz `EventArgs`.

*Primjer 6-5. Argumenti događaja koriste klasu izvedenu iz EventArgs*

```
public delegate void NumberChangedEventHandler(object sender,EventArgs eventArgs);
```

```
public class NumberChangedEventArgs : EventArgs
{
    public int Number;//Ovo bi zaista trebalo biti svojstvo
}
public class MyPublisher
{
    public event NumberChangedEventHandler NumberChanged;
    public void FireNewNumberEvent(EventArgs eventArgs)
    {
        //Uvijek provjerava je li delegat null prije pozivanja
        if(NumberChanged != null)
```

Primjer 6-5. Argumenti događaja koriste klasu izvedenu iz EventArgs (nastavak)

```
        NumberChanged(this,eventArgs);
    }
}
public class MySubscriber
{
    public void OnNumberChanged(object sender,EventArgs eventArgs)
    {
        NumberChangedEventArgs numberArg;
        numberArg = eventArgs as NumberChangedEventArgs;
        Debug.Assert(numberArg != null);
        string message = numberArg.Number;
        MessageBox.Show("The new number is "+ message);
    }
}
//Klijentski kod
MyPublisher publisher = new MyPublisher( );
MySubscriber subscriber = new MySubscriber( );
publisher.NumberChanged += subscriber.OnNumberChanged;

NumberChangedEventArgs numberArg = new NumberChangedEventArgs( );
numberArg.Number = 4;
```

//Obratite pozornost da izvor može objaviti događaj bez poznavanja tipa argumenta  
publisher.FireNewNumberEvent(numberArg);

Izvođenje klase iz EventArgs za proslijeđivanje zadanih argumenata dopušta vam da dodate argumente, uklonite nekorištene argumente, izvedete još jednu klasu iz EventArgs i tako dalje, bez nametanja promjena na pretplatnike koje ne zanimaju novi aspekti.

Budući da je rezultirajuća definicija delegata sada tako prilagodljiva, .NET pruža EventHandler delegata:

```
public delegate void EventHandler(object sender,EventArgs eventArgs);
```

.NET kosturi za aplikacije, kao što su Windows Forms i ASP.NET, često koriste EventHandler. Međutim, fleksibilnost amorfnе osnovne klase dolazi na trošak sigurnosti tipova. Kako bi riješio taj problem, .NET osigurava generičku inačicu delegata EventHandler:

```
public delegate void EventHandler<E>(object sender,E e)
                                where E : EventArgs;
```

Ako ste pretplatnik mišljenja da svi vaši objekti uzimaju oblik pošiljatelja object i klase izvedene iz EventArgs, onda će taj delegat zadovoljiti sve vaše potrebe. U svim ostalim slučajevima, morate još definirati delegate za specifične potpise sa kojima morate raditi.



Konvencija za ime metode pretplatnika koja upravlja događajima je On<EventName>, što čini kod standardnim i čitkim.

## Definiranje prilagođenih argumenata događaja

Kao što sam objasnio u prethodnom odjeljku, trebali biste dati argumente funkciji za obradu događaja u klasi izvedenoj iz EventArgs te držati argumente kao članove klase. Klasa delegat jednostavno iterira po popisu pretplatnika, prosljeđujući objekte argumenta od jednog pretplatnika do sljedećeg. Međutim, ništa ne sprječava određenog pretplatnika da modificira vrijednosti argumenta i tako utječe na sve kasnije pretplatnike koji rade sa argumentom. Obično biste trebali spriječiti pretplatnike da modificiraju te članove dok se prosljeđuju od jednog do drugog pretplatnika. Kako biste spriječili mijenjanje argumenta, pružite pristup argumentima kao svojstvima samo za čitanje ili ih izložite kao javne članove i primijenite modifikator pristupa readonly. U oba slučaja trebali biste inicijalizirati vrijednosti argumenata u konstruktoru. Primjer 6-6 pokazuje obje tehnike.

*Primjer 6-6. Sprječavanje pretplatnika da modificiraju parametre u klasi argumenta*

```
public class NumberEventArgs1 : EventArgs
{
    public readonly int Number;
    public NumberEventArgs1(int number)
    {
        Number = number;
    }
}
public class NumberEventArgs2 : EventArgs
{
    int m_Number;

    public NumberEventArgs2(int number)
    {
        m_Number = number;
    }
    public int Number
    {
        get
        {
            return m_Number;
        }
    }
}
```

## Generička funkcija za obradu događaja

Generički delegati osobito su korisni kada se radi sa događajima. Pod pretpostavkom da će svi delegati upotrijebljeni za upravljanje događajem vratiti void te da neće imati izlaznih parametara, jedino što razlikuje jednog takvog delegata od drugog je broj argumenata i njihov tip. Ta razlika lako se može uopćiti s pomoću generika. Razmotrite ovaj skup definicija delegata:

```
public delegate void GenericEventHandler( );
public delegate void GenericEventHandler<T>(T t);
```

```

public delegate void GenericEventHandler<T,U>(T t,U u);
public delegate void GenericEventHandler<T,U,V>(T t,U u,V v);
public delegate void GenericEventHandler<T,U,V,W>(T t,U u,V v,W w);
public delegate void GenericEventHandler<T,U,V,W,X>(T t,U u,V v,W w,X x);
public delegate void GenericEventHandler<T,U,V,W,X,Y>(T t,U u,V v,W w,
                                                    X x,Y y);
public delegate void GenericEventHandler<T,U,V,W,X,Y,Z>(T t,U u,V v,W w,
                                                    X x,Y y,Z z);

```



Tehnika upotrijebljena u definiciji `GenericEventHandler` zove se *preklapanje po broju parametara tipa* (engl. *overloading by type parameter arity*). Prevoditelj zapravo pridruži različita imena preklapljenim delegatima, razlikujući ih na temelju broja argumenata ili broja generičkih parametara tipa. Na primjer, sljedeći kod:

```

Type type = typeof(GenericEventHandler<, >);
Trace.WriteLine(type.ToString( ));

```

prati:

```

GenericEventHandler`2[T,U]

```

jer prevoditelj dodaje `2 imenu delegata (pri čemu je 2 broj upotrijebljenih generičkih parametara tipa).

Različite inačice `GenericEventHandler` mogu se koristiti da se pozove bilo koja metoda koja radi sa događajima i koja prihvaća između nula i sedam argumenata (više od otprilike pet argumenata ionako je loša praksa, te biste za prosljeđivanje više argumenata trebali koristiti strukture, ili klasu izvedenu iz `EventArgs`,). Možete definirati bilo koju kombinaciju tipova koristeći `GenericEventHandler`. Na primjer:

```

GenericEventHandler<int> del1;
GenericEventHandler<int,int> del2;
GenericEventHandler<int,string> del3;
GenericEventHandler<int,string,int> del4;

```

ili, za prosljeđivanje više argumenata:

```

struct MyStruct
{...}
public class MyArgs : EventArgs
{...}
GenericEventHandler<MyStruct> del5;
GenericEventHandler<MyArgs> del6;

```

Primjer 6-7 pokazuje upotrebu `GenericEventHandler` i generičke metode za rad s događajima.

*Primjer 6-7. Generičko upravljanje događajima*

```

public class MyArgs : EventArgs
{...}
public class MyPublisher
{
    public event GenericEventHandler<MyPublisher,MyArgs> MyEvent;
    public void FireEvent( )

```

### Primjer 6-7. Generičko upravljanje događajima (nastavak)

```
{
    MyArgs args = new MyArgs(...);
    MyEvent(this,args);
}
}
public class MySubscriber<A> where A : EventArgs
{
    public void OnEvent(MyPublisher sender,A args)
    {...}
}
MyPublisher publisher = new MyPublisher( );
MySubscriber<MyArgs> subscriber = new MySubscriber<MyArgs>( );
publisher.MyEvent += subscriber.OnEvent;
```

Ovaj primjer koristi `GenericEventHandler` sa dva parametra tipa, tipom pošiljatelja i klasom spremnika argumenata, sličan generičkoj i posebnoj inačici `EventHandler`. Međutim, za razliku od `EventHandler`, `GenericEventHandler` je siguran za tipove, jer prihvaća samo objekte tipa `MyPublisher` (a ne samo `object`) kao pošiljatelje. Jasno, možete koristiti `GenericEventHandler` sa svim potpisima događaja, uključujući i one koji se ne slažu sa pošiljateljem `object` i smjernicom za izvođenje `EventArgs`.

Zbog demonstracije primjer 6-7 također koristi generičkog pretplatnika, koji prihvaća generički parametar tipa za spremnik argumenata događaja. Mogli biste definirati pretplatnika da koristi određeni tip argumenta umjesto toga, bez da imalo utječete na kod izvora događaja:

```
public class MySubscriber
{
    public void OnEvent(MyPublisher sender,MyArgs args)
    {...}
}
```

Ako želite upotrijebiti klasu izvedenu iz `EventArgs` kao argument, možete postaviti ograničenje za `GenericEventHandler`:

```
public delegate void GenericEventHandler<T,U>(T t,U u) where U : EventArgs;
```

Međutim, zbog generičke upotrebe `GenericEventHandler`, bolje je staviti ograničenje na pretplatnički tip (ili pretplatničku metodu), kao što je pokazano u primjeru 6-7.



Ponekad je korisno izraditi alias za određenu kombinaciju specifičnih tipova. Možete to učiniti s pomoću iskaza `using`:

```
using MyHandler = GenericEventHandler
                    <MyPublisher,MyArgs>;

public class MyPublisher
{
    public event MyHandler MyEvent;
    //Ostatak MyPublisher
}
```

Obratite pozornost na činjenicu da je doseg aliasa jednak dosegu datoteke, tako da morate ponoviti izradu aliasa u svim datotekama projekta, na isti način na koji biste to učinili da koristite imenske prostore.

## Obrambeno objavljivanje događaja

U .NET-u, ako delegat nema ciljeve na svojem internom popisu, njegova vrijednost bit će postavljena na null. C# izvor događaja uvijek bi trebao provjeriti je li delegat na vrijednosti null prije nego ga pokuša pozvati. Ako nijedan klijent nije pretplaćen na događaj, popis ciljeva delegata bit će prazan i vrijednost delegata postavljena na null. Kada izvor pokuša pristupiti poništenom delegatu, izbacuje se iznimka. Visual Basic 2005 programeri ne moraju provjeravati vrijednost delegata jer iskaz RaiseEvent može prihvatiti praznog delegata. Interno, RaiseEvent provjerava da delegat nije null prije nego mu pristupi.

Zaseban problem sa događajima temeljenim na delegatima su iznimke. Bilo koja neoobrađena iznimka koju je izbacio pretplatnik bit će prosljeđena izvoru događaja. Neki pretplatnici mogu sresti iznimku pri obradi događaja, ne obraditi je, te uzrokovati blokadu izvora. Iz tih razloga biste uvijek trebala objavljivati unutar bloka try/catch. Primjer 6-8 prikazuje te napomene.

*Primjer 6-8. Obrambeno objavljivanje*

```
public class MyPublisher
{
    public event EventHandler MyEvent;
    public void FireEvent( )
    {
        try
        {
            if(MyEvent != null)
                MyEvent(this, EventArgs.Empty);
        }
        catch
        {
            //Obrada iznimaka
        }
    }
}
```

Međutim, kod u primjeru 6-8 prekida objavljivanje događaja u slučaju da pretplatnik izbaci iznimku. Ponekad želite nastaviti s objavljivanjem čak i ako pretplatnik izbaci iznimku. Kako biste to učinili, morate ručno iterirati po internom popisu delegata i uloviti sve iznimke koje su izbacili neki delegati s popisa. Internom popisu pristupate koristeći posebnu metodu koju podržava svaki delegat (primjer 6-1), zvanu GetInvocationList( ). Ona je definirana kao:

```
public virtual Delegate[] GetInvocationList( );
```

GetInvocationList( ) vraća kolekciju delegata po kojima možete iterirati, kao što je pokazano u primjeru 6-9.

*Primjer 6-9. Nastavak objavljivanja unatoč iznimkama koje su izbacili pretplatnici*

```
public class MyPublisher
{
    public event EventHandler MyEvent;
    public void FireEvent( )
    {
        if(MyEvent == null)
        {
            return;
        }
        Delegate[] delegates = MyEvent.GetInvocationList( );
        foreach(Delegate del in delegates)
        {
            EventHandler sink = (EventHandler)del;
            try
            {
                sink(this,EventArgs.Empty);
            }
            catch{}
        }
    }
}
```

## Klasa EventsHelper

Problem sa kodom koji objavljuje događaj u primjeru 6-9 je to što nije ponovno upotrebljiv – morate ga kopirati svaki put kada želite izdvajanje pogrešaka između izvora i pretplatnika. Međutim, moguće je napisati pomoćnu klasu koja može objavljivati bilo kojem delegatu, prosljeđivati bilo koju kolekciju argumenata i hvatati moguće iznimke. Primjer 6-10 pokazuje statičku klasu EventsHelper, koja pruža statičku metodu Fire( ). Ona obrambeno izaziva bilo koji tip događaja.

*Primjer 6-10. Klasa EventsHelper*

```
public static class EventsHelper
{
    public static void Fire(Delegate del,params object[] args)
    {
        if(del == null)
        {
            return;
        }
        Delegate[] delegates = del.GetInvocationList( );
        foreach(Delegate sink in delegates)
        {
            try
            {
                sink.DynamicInvoke(args);
            }
            catch{}
        }
    }
}
```

Postoje dva ključna elementa u implementiranju `EventsHelper`. Prvi je njezina sposobnost da pozove bilo kojeg delegata. To je omogućeno korištenjem metode `DynamicInvoke( )`, koju osigurava svaki delegat (primjer 6-1). `DynamicInvoke( )` poziva delegata, prosljeđujući mu kolekciju argumenata. Definira se kao:

```
public object DynamicInvoke(object[] args);
```

Drugi ključ u implementiranju `EventsHelper` je prosljeđivanje otvorenog broja objekata kao argumenata za pretplatnike. To se čini s pomoću C# modifikatora parametra `params` (`ParamArray` u Visual Basicu 2005), koji dopušta upisivanje objekata kao parametara. Prevoditelj pretvara izravno upisane argumente u polje objekata i prosljeđuje ga.

Upotreba `EventsHelper` je elegantna i jasna: jednostavno joj prosljedite delegata za pozivanje i parametre. Na primjer, za delegata `MyEventHandler`, definiranog kao:

```
public delegate void MyEventHandler(int number,string str);
```

objavljajući kod može biti sljedeći:

```
public class MyPublisher
{
    public event MyEventHandler MyEvent;
    public void FireEvent(int number, string str)
    {
        EventsHelper.Fire(MyEvent,number,str);
    }
}
```

Kada koristite `EventsHelper` iz Visual Basicu 2005, trebate izravno pristupiti delegatu. Visual Basic 2005 prevoditelj stvara skrivenu varijablu članicu koja odgovara članu događaja. Ime skrivenog člana je ime događaja sa sufiksom `Event`:

```
Public Delegate Sub MyEventHandler (ByVal number As Integer, ByVal str As String)
Public Class MyPublisher
    Public Event MyEvent As MyEventHandler
    Public Sub FireEvent (ByVal number As Integer, ByVal str As String)
        EventsHelper.Fire(MyEventEvent, number, str)
    End Sub
End Class
```

## Kako učiniti `EventsHelper` sigurnom za tipove

Problem sa `EventsHelper` koja je prikazana u primjeru 6-10 je to da nije sigurna za tipove. Metoda `Fire( )` smatra da kolekcija amorfni objekata dopušta bilo koju kombinaciju parametara, uključujući i netočnu kombinaciju. Na primjer, pogledajte sljedeću definiciju delegata:

```
public delegate void MyEventHandler(int number,string str);
```

sljedeći objavljujući kod će se bez problema prevesti ali neće uspjeti objaviti događaj:

```
public class MyPublisher
{
    public event MyEventHandler MyEvent;
    public void FireEvent(int number, string str)
    {
```

```

        EventsHelper.Fire(MyEvent, "Not", "Type", "Safe");
    }
}

```

Bilo koje neslaganje u broju ili tipu argumenata neće biti otkriveno tijekom prijevoda. Povrh toga, `EventsHelper` će ugasiti iznimke i nećete ni biti svjesni problema.

Međutim, ako se možete obvezati da ćete uvijek koristiti `GenericEventHandler` umjesto da definirate vlastite delegate za rad s događajima, postoji način da provedete sigurnost tipova tijekom prijevoda. Umjesto da ovako definirate delegata:

```
public delegate void MyEventHandler(int number, string str);
```

izravno upotrijebite `EventHandler`, ili mu izradite alias:

```
using MyEventHandler = GenericEventHandler<int, string>;
```

Zatim kombinirajte `EventsHelper` sa `GenericEventHandler`, kao što je pokazano u primjeru 6-11.

*Primjer 6-11. EventsHelper sigurna za tipove*

```
public static class EventsHelper
{
    //Isto kao i Fire( ) u primjeru 6-10
    public static void UnsafeFire(Delegate del, params object[] args)
    {...}
    public static void Fire(GenericEventHandler del)
    {
        UnsafeFire(del);
    }
    public static void Fire<T>(GenericEventHandler<T> del, T t)
    {
        UnsafeFire(del, t);
    }
    public static void Fire<T, U>(GenericEventHandler<T, U> del, T t, U u)
    {
        UnsafeFire(del, t, u);
    }
    public static void Fire<T, U, V>(GenericEventHandler<T, U, V> del,
                                    T t, U u, V v)
    {
        UnsafeFire(del, t, u, v);
    }
    public static void Fire<T, U, V, W>(GenericEventHandler<T, U, V, W> del,
                                        T t, U u, V v, W w)
    {
        UnsafeFire(del, t, u, v, w);
    }
    public static void Fire<T, U, V, W, X>(GenericEventHandler<T, U, V, W, X> del,
                                          T t, U u, V v, W w, X x)
    {
        UnsafeFire(del, t, u, v, w, x);
    }
    public static void Fire<T, U, V, W, X, Y>(GenericEventHandler<T, U, V, W, X, Y> del,
                                             T t, U u, V v, W w, X x, Y y)

```

Primjer 6-11. EventsHelper sigurna za tipove (nastavak)

```
{
    UnsafeFire(del,t,u,v,w,x,y);
}
public static void Fire<T,U,V,W,X,Y,Z>(GenericEventHandler<T,U,V,W,X,Y,Z> del,
                                     T t,U u,V v,W w,X x,Y y,Z z)
{
    UnsafeFire(del,t,u,v,w,x,y,z);
}
}
```

Budući da su broj i tip argumenata prosljeđenih `GenericEventHandler` poznati prevoditelju, možete provesti sigurnost tipova tijekom prijevoda. Zahvaljujući preklapanju po broju generičkih parametara tipa, ne morate čak ni specificirati generičke parametre tipa prosljeđene metodama `Fire( )`. Objavljujući kod ostaje isti kao da koristite definiciju iz primjera 6-10:

```
using MyEventHandler = GenericEventHandler<int,string>;

public class MyPublisher
{
    public event MyEventHandler MyEvent;
    public void FireEvent(int number, string str)
    {
        //Ovo je sada sigurno za tip
        EventsHelper.Fire(MyEvent,number,str);
    }
}
```

Prevoditelj zaključuje o korištenom tipu i broju parametara tipa te odabire odgovarajuću preklaplenu inačicu `Fire( )`.

Ako možete jamčiti samo upotrebu `EventHandler` umjesto `GenericEventHandler`, možete dodati ove preklopljene metode klasi `EventsHelper`:

```
public static void Fire(EventHandler del,object sender,EventArgs e)
{
    UnsafeFire(del,sender,e);
}
public static void Fire<E>(EventHandler<E> del,object sender,E e)
                                     where E : EventArgs
{
    UnsafeFire(del,sender,t);
}
```

To će vam omogućiti da obrambeno objavljujete, na način siguran za tip, bilo koji događaj temeljen na `EventHandler`.

`EventsHelper` može još ponuditi i metodu `UnsafeFire( )` koja nije sigurna za tip:

```
public static void UnsafeFire(Delegate del,params object[] args)
{
    if(args.Length > 7)
    {
```

```

        Trace.TraceWarning("Too many parameters. Consider a structure
                           to enable the use of the type-safe versions");
    }
    //Ostalo je isto kao i u primjeru 6-11
}

```

To je potrebno u slučaju da radite sa delegatima koji nisu temeljeni na `GenericEventHandler` ili `EventHandler`, ili kada imate više parametara nego što `GenericEventHandler` može prihvatiti. Očito, trebalo bi izbjegavati upotrebu više od pet parametara za bilo koju metodu, ali sada je bar programer koji koristi `EventsHelper` svjestan zamke sigurnosti tipova. Ako želite uvijek provoditi upotrebu metoda `Fire( )` sigurnih za tipove, jednostavno definirajte `UnsafeFire( )` kao privatnu:

```
static void UnsafeFire(Delegate del,params object[] args);
```

## Pristupnici događaju

Kako biste povezali pretplatnika sa izvorom, izravno pristupate varijabli članici događaja izvora. Javno izlaganje članova klase znači izazivanje nevolje. To narušava temeljno objektno orijentirano projektno načelo učajurivanja i skrivanja informacija, te povezuje sve pretplatnike sa točnom definicijom varijable članice. Kako bi ublažio taj problem, `C#` osigurava mehanizam sličan svojstvima koji se zove *pristupnik događaju* (engl. *event accessor*). Pristupnici pružaju pogodnost sličnu pogodnosti svojstava, skrivajući samog člana klase, no zadržavajući izvornu lakoću upotrebe. `C#` koristi `add` i `remove` – koji izvode funkcije operatora `+=` i `-=` – kako bi učajurio varijablu članicu događaja. Primjer 6-12 prikazuje upotrebu pristupnika događaju i odgovarajući kod klijenta. Obratite pozornost na činjenicu da označavanje učajurenog člana delegata kao događaja ne donosi nikakvu prednost kada koristite pristupnike događaju.

*Primjer 6-12. Upotreba pristupnika događaju*

```

using MyEventHandler = GenericEventHandler<string>;
public class MyPublisher
{
    MyEventHandler m_MyEvent;
    public event MyEventHandler MyEvent
    {
        add
        {
            m_MyEvent += value;
        }
        remove
        {
            m_MyEvent -= value;
        }
    }
    public void FireEvent( )
    {
        EventsHelper.Fire(m_MyEvent,"Hello");
    }
}

```

Primjer 6-12. Upotreba pristupnika događaju (nastavak)

```
public class MySubscriber
{
    public void OnEvent(string message)
    {
        MessageBox.Show(message);
    }
}

//Klijentski kod:
MyPublisher publisher = new MyPublisher( );
MySubscriber subscriber = new MySubscriber( );

//Postavlja vezu:
publisher.MyEvent += subscriber.OnEvent;

publisher.FireEvent( );

//Prekida vezu:
publisher.MyEvent -= subscriber.OnEvent;
```

## Upravljanje velikim brojem događaja

Zamislite klasu koja objavljuje vrlo velik broj događaja. To je uobičajeno pri razvijanju kostura. Na primjer, klasa `Control` u imenskom prostoru `System.Windows.Forms` ima na desetke događaja koji odgovaraju mnogim Windows porukama. Problem vezan za upravljanje velikim brojem događaja jest da je jednostavno nepraktično alocirati člana klase za svaki događaj: definicija klase, dokumentacije, CASE dijagrami pa čak i IntelliSense bili bi neupravljivi. Kako bi riješio taj problem, .NET pruža klasu `EventHandlerList` (koja se nalazi u imenskom prostoru `System.ComponentModel`):

```
public sealed class EventHandlerList : IDisposable
{
    public EventHandlerList( );
    public Delegate this[object key]{get;set;}
    public void AddHandler(object key, Delegate value);
    public void AddHandlers(EventHandlerList listToAddFrom);
    public void RemoveHandler(object key, Delegate value);
    public virtual void Dispose( );
}
```

`EventHandlerList` je linearni popis koji pohranjuje parove ključ/vrijednost. Ključ je `object` koji identifikira događaj, a vrijednost je instanca `System.Delegate`. Budući da je indeks `object`, on može biti cjelobrojna vrijednost, niz, određena instanca gumba i tako dalje. Vi dodajete i uklanjate individualne metode za rad s događajima koristeći metode `AddHandler` i `RemoveHandler`. Možete i dodati sadržaj već postojeće `EventHandlerList` koristeći metodu `AddHandlers`( ). Kako biste izazvali događaj, pristupate popisu događaja koristeći indeks sa ključnim objektom, te dobivate natrag objekt `System.Delegate`. Zatim pretvarate taj delegat u sam delegat događaja i izazivate događaj.

Primjer 6-13 prikazuje upotrebu klase `EventHandlerList` pri implementaciji klase `MyButton`. Gumb podržava mnoge događaje, kao što je pritiskanje tipke miša i pomicanje miša, sve

usmjerene na isti popis događaja. Koristeći pristupnike događaju, to je potpuno učahurenjeno od klijenata.

*Primjer 6-13. Upotreba klase EventHandlerList za upravljanje velikim brojem klijenata*

```
using System.ComponentModel;

using ClickEventHandler = GenericEventHandler<MyButton,EventArgs>;
using MouseEventHandler = GenericEventHandler<MyButton,MouseEventArgs>;

public class MyButton
{
    EventHandlerList m_EventList;
    public MyButton( )
    {
        m_EventList = new EventHandlerList( );
        /* Ostatak inicijalizacije */
    }
    public event ClickEventHandler Click
    {
        add
        {
            m_EventList.AddHandler("Click",value);
        }
        remove
        {
            m_EventList.RemoveHandler("Click",value);
        }
    }
    public event MouseEventHandler MouseMove
    {
        add
        {
            m_EventList.AddHandler("MouseMove",value);
        }
        remove
        {
            m_EventList.RemoveHandler("MouseMove",value);
        }
    }
    void FireClick( )
    {
        ClickEventHandler handler = m_EventList["Click"] as ClickEventHandler;
        EventsHelper.Fire(handler,this,EventArgs.Empty);
    }
    void FireMouseMove(MouseButtons button,int clicks,int x,int y,int delta)
    {
        MouseEventHandler handler = m_EventList["MouseMove"] as MouseEventHandler;
        MouseEventArgs args = new MouseEventArgs(button,clicks,x,y,delta);
        EventsHelper.Fire(handler,this,args);
    }
    /* Ostale metode i definicije dogadjaja */
}
```

Problem sa primjerom 6-13 je u tome što se događaji kao što je pomak miša ili čak pritisak tipke miša često događaju, te stvaranje novog niza kao ključa za svako pozivanje povećava pritisak na upravljivu gomilu. Bolji pristup bio bi koristiti ranije alocirane statičke ključeve koje dijele mnoge instance:

```
public class MyButton
{
    EventHandlerList m_EventList;
    static object m_MouseMoveEventKey = new object( );

    public event MouseEventHandler MouseMove
    {
        add
        {
            m_EventList.AddHandler(m_MouseMoveEventKey,value);
        }
        remove
        {
            m_EventList.RemoveHandler(m_MouseMoveEventKey,value);
        }
    }
    void FireMouseMove(MouseButtons button,int clicks,int x,int y,int delta)
    {
        MouseEventHandler handler;
        handler = m_EventList[m_MouseMoveEventKey] as MouseEventHandler;
        MouseEventArgs args = new MouseEventArgs(button,clicks,x,y,delta);
        EventsHelper.Fire(handler,this,args);
    }
    /* Ostatak implementacije */
}
```

## Pisanje sučelja odvoda

Skrivanjem članova događaja, pristupnici događaju pružaju jedva dovoljno učajurivanja. Međutim, to se može poboljšati. Za primjer, razmotrite slučaj u kojem se pretplatnik želi pretplatiti na skup događaja. Zašto bi morao napraviti više, potencijalno skupih, poziva kako bi uspostavio i prekinuo veze? Zašto pretplatnik uopće mora znati za pristupnike događaju? Što ako pretplatnik želi primati događaje na cijelom sučelju, umjesto od individualnih metoda? Sljedeći korak je pružanje jednostavnog, generičkog načina za upravljanje vezama između izvora događaja i pretplatnika – načina koji će štiti od suvišnih poziva, učajurivati pristupnike događaju i članove događaja, te dopuštati sučelje za odvod. Ovaj odjeljak opisuje tehniku koju sam razvio upravo da čini sve te stvari. Razmotrite sučelje koje definira skup događaja, sučelje `IMySubscriber`:

```
public interface IMySubscriber
{
    void OnEvent1(object sender,EventArgs eventArgs);
    void OnEvent2(object sender,EventArgs eventArgs);
    void OnEvent3(object sender,EventArgs eventArgs);
}
```

Bilo tko može implementirati to sučelje, a sučelje je zapravo sve za što bi izvor trebao znati:

```
public class MySubscriber : IMySubscriber
{
    public void OnEvent1(object sender,EventArgs eventArgs)
    {...}
    public void OnEvent2(object sender,EventArgs eventArgs)
    {...}
    public void OnEvent3(object sender,EventArgs eventArgs)
    {...}
}
```

Zatim definirajte enumeraciju događaja i označite ju atributom `Flags`:

```
[Flags]
public enum EventType
{
    OnEvent1,
    OnEvent2,
    OnEvent3,
    OnAllEvents = OnEvent1|OnEvent2|OnEvent3
}
```

Atribut `Flags` naznačuje da bi se vrijednosti enumeratora mogle upotrijebiti kao maska bitova (pogledajte definiciju tipa `EventType.OnAllEvents`). To dopušta da kombinirate različite vrijednosti enumeratora primjenom operatora `|` (ILI) nad bitovima ili ih maskirate korištenjem operatora `&` (I).

Izvor osigurava dvije metode, `Subscribe( )` i `Unsubscribe( )`, a svaka od njih prihvaća dva parametra: sučelje i oznaku maske bitova za naznačavanje toga ne koje događaje pretplatiti sučelje odvoda. Interno, izvor može imati jednog člana delegata događaja po metodi na sučelju odvoda, ili samo jednog člana za sve metode (to je implementacijska pojedinost, koja je skrivena od pretplatnika). Primjer 6-14 koristi po jednu varijablu članicu događaja za svaku metodu na sučelju odvoda. Primjer pokazuje `Subscribe( )` i `Unsubscribe( )` kao i metodu `FireEvent( )`, a obrada pogrešaka uklonjena je zbog jasnoće. `Subscribe( )` provjerava oznaku i pretplaćuje odgovarajuću metodu sučelja:

```
if((eventType & EventType.OnEvent1) == EventType.OnEvent1)
{
    m_Event1 += subscriber.OnEvent1;
}
Unsubscribe( ) removes the subscription in a similar fashion.
```

`Unsubscribe( )` uklanja pretplatu na sličan način.

*Primjer 6-14. Sučelje za odvod*

```
using MyEventHandler = GenericEventHandler<object,EventArgs>;
public class MyPublisher
{
    MyEventHandler m_Event1;
    MyEventHandler m_Event2;
```

Primjer 6-14. Sučelje za odvod (nastavak)

```
MyEventHandler m_Event3;

public void Subscribe(IMySubscriber subscriber,EventType eventType)
{
    if((eventType & EventType.OnEvent1) == EventType.OnEvent1)
    {
        m_Event1 += subscriber.OnEvent1;
    }
    if((eventType & EventType.OnEvent2) == EventType.OnEvent2)
    {
        m_Event2 += subscriber.OnEvent2;
    }
    if((eventType & EventType.OnEvent3) == EventType.OnEvent3)
    {
        m_Event3 += subscriber.OnEvent3;
    }
}

public void Unsubscribe(IMySubscriber subscriber,EventType eventType)
{
    if((eventType & EventType.OnEvent1) == EventType.OnEvent1)
    {
        m_Event1 -= subscriber.OnEvent1;
    }
    if((eventType & EventType.OnEvent2) == EventType.OnEvent2)
    {
        m_Event2 -= subscriber.OnEvent2;
    }
    if((eventType & EventType.OnEvent3) == EventType.OnEvent3)
    {
        m_Event3 -= subscriber.OnEvent3;
    }
}

public void FireEvent(EventType eventType)
{
    if((eventType & EventType.OnEvent1) == EventType.OnEvent1)
    {
        EventsHelper.Fire(m_Event1,this,EventArgs.Empty);
    }
    if((eventType & EventType.OnEvent2) == EventType.OnEvent2)
    {
        EventsHelper.Fire(m_Event2,this,EventArgs.Empty);
    }
    if((eventType & EventType.OnEvent3) == EventType.OnEvent3)
    {
        EventsHelper.Fire(m_Event3,this,EventArgs.Empty);
    }
}
}
```

Kod potreban za pretplaćivanje ili ukidanje pretplate jednako je jasan:

```
MyPublisher publisher = new MyPublisher( );
IMySubscriber subscriber = new MySubscriber( );
//Pretplata na događaje 1 i 2
publisher.Subscribe(subscriber, EventType.OnEvent1|EventType.OnEvent2);
//Objavljuje samo događaj 1
publisher.FireEvent(EventType.OnEvent1);
```

Ipak, on pokazuje eleganciju ovog pristupa za potapanje cijelog sučelja jednim pozivom, te pokazuje kako su članovi klase događaja zaista potpuno učahureni.