Rudolf Huttary, Arne Schäpers, Pieter-Paul Spiertz

Action painting

Programmeren met C#, deel 5: Event-driven programmeren onder Windows

In een computer is het nooit saai. Als een gebruiker wat met zijn muis rondcirkelt, krijgt Windows een hele stortvloed aan gebeurtenissen (zogenaamde events) te verstouwen. Al die events worden vervolgens weer aan diverse programma's doorgegeven. Het maken van GUI-programma's voelt in C# dan ook een beetje als spelen met een onbeperkt grote Lego-doos.

Wie onze vorige C#-verhalen nog pittig vond, kan weer aanhaken in dit deel van onze kleine programmeercursus. Vanaf deze c't duiken we namelijk in de bonte vensterwereld die Windows-programmeren heet. Toch zullen we daarbij veel van de theorie uit de vorige delen hergebruiken. Wees dus niet bang om terug te bladeren als het gaat over methoden, klassen, eigenschappen en overerving. In dit deel zullen we ingaan op zogenaamde events.

Als je een applicatie met een grafische user interface (GUI) ontwikkelt, heb je allerlei besturingselementen nodig: menu's, knoppen, textboxes, panels, enzovoort. Deze heten ook wel *widgets*, en specifiek in .NET-jargon *controls*. Zulke controls voeg je toe aan een venster, in jargon een *formulier* (afgekort *form*). Als een gebruiker in een form een handeling uitvoert – hij drukt bijvoorbeeld op een knop – treedt er een zogenaamde *event* op. Je kunt voor dat ene event in je programma een bijbehorende *event handler* maken, waarin vervolgens je gewenste code wordt uitgevoerd. Daarover zometeen veel meer.

Controls maak je in C# niet met de stekelige Windows-functies, want speciaal hiervoor bevat de .NET-bibliotheek een eigen 'toolkit', genaamd *Windows.Forms*, waarin allerlei GUI-componenten als klasse gereedstaan. Met Windows.Forms bind je je overigens niet aan Windows: ook Mono 1.16 en het onlangs verschenen SharpDevelop 2.0 ondersteunen het vrij goed.

Rapid Application Development

Het aardige is dat je forms tegenwoordig met de muis kunt ontwerpen. Het Apple-programma Hypercard introduceerde dit al in 1987. Start maar eens een nieuw project in Visual C#. (Als je SharpDevelop gebruikt, heet dit een nieuwe 'solution'.) Kies vervolgens het sjabloon 'Windows-application' en geef het een geschikte projectnaam, bijvoorbeeld 'HelloWindows'. Zodra het project is aangemaakt, vind je in de IDE een kant-en-klaar, zonder aanpassingen uitvoerbaar programma, dat bestaat uit één leeg venster: je form. Visual C# laat de form automatisch in de zogenaamde *designer view* zien.

De bedoeling is dat ons programma na een druk op een knop een venstertje met een welkomsttekst laat zien. Eerst moet er dus een knop komen. Klik in de linker vensterbalk de toolbox open (of kies in SharpDevelop de 'werkbalk' uit het menu), en selecteer in de categorie 'common controls' de `Button`. Nu kun je een instantie hiervan op je form plaatsen, naar keuze via drag&drop of point&click.

In Visual C# hangt de toolbox nu deels over je form. Het is daarom handig om op het pin-icon te klikken, waarmee je de toolbox een eigen oppervlak geeft of voortaan elders vastzet.

Bovendien vind je de toolbox in het menu bij 'View / Toolbox'.

De tekst op de button kun je aanpassen door in het snelmenu van de control 'Properties' te kiezen, dan in het rechtsonder verschijnende eigenschappenvenster de property Text op te zoeken en de waarde ervan van 'button1' in 'Hallo' te veranderen. Die waarde 'button1' is trouwens ook de identifier van je button, die de form designer zelf heeft gekozen. Deze kun je veranderen via de (Name)-property, bijvoorbeeld in 'bHallo'. Je form ziet er nu uit zoals in het middelste plaatje rechts.

Neem even de tijd en bekijk eens de andere 'design-time' properties in het properties-window. Je ziet hier de eigenschappen van het element dat je in de designer view geselecteerd hebt. Vermoedelijk zul je het leeuwendeel daarvan nooit gebruiken en de lijst is niet eens compleet (tijdens runtime zijn er afhankelijk van de control nóg meer properties beschikbaar), maar het is niet verkeerd om een algemeen beeld op te krijgen. Let ook op de nuttige tooltips. In dit properties-window kun je ook de properties van het form bewerken, als je het form selecteert met een rechtsklik in een leeg gedeelte van het form. De Text-eigenschap bepaalt bij forms bijvoorbeeld de venstertitel.

Wat er nu nog ontbreekt, is de code die uitgevoerd moet worden als iemand op de button klikt. Dubbelklik hiervoor op de button in je form, en plots bevind je je midden in de bijbehorende *code view*. De IDE heeft alvast zelf al het beginsel van een methode voorbereid met als naam bHallo_Click() (of als je de button-identifier nog niet had gewijzigd: button1_Click()). Je hoeft alleen nog maar in de body de code in te voeren die na de klik moet worden uitgevoerd. De methode wordt namelijk uitgevoerd als de event Click van de bHallo-control optreedt. Zet bijvoorbeeld de volgende code in de body:

```
MessageBox.Show(
  "Welkom in de wereld\n"
  + "van het Windows-programmeren"
```

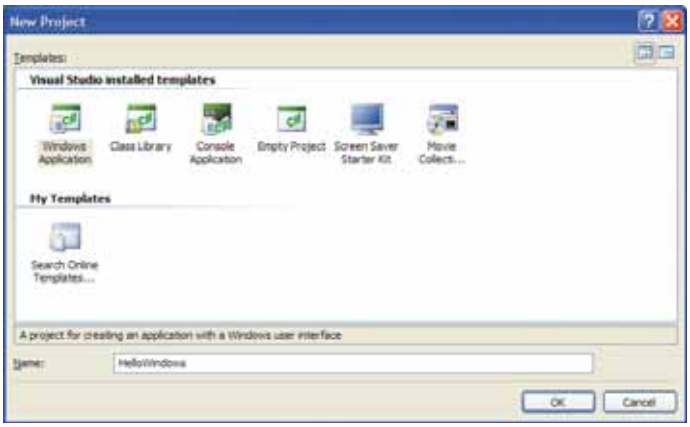
Het onderste screenshot laat het eindresultaat zien: de statische methode Show() van de Windows.Forms-klasse MessageBox toont de tekst in een eigen form, een zogenaamd 'modal dialog window' – een venster dat gebruikersinvoer op andere vensters van het programma blokkeert zolang als het actief is. Je kunt het sluiten met een klik op OK. Alle benodigde code hiervoor zit trouwens al in de implementatie van Show(): je hoeft hiervoor geen eigen form te ontwerpen of een eigen Click-handler te bouwen.

De magie van Form1.cs

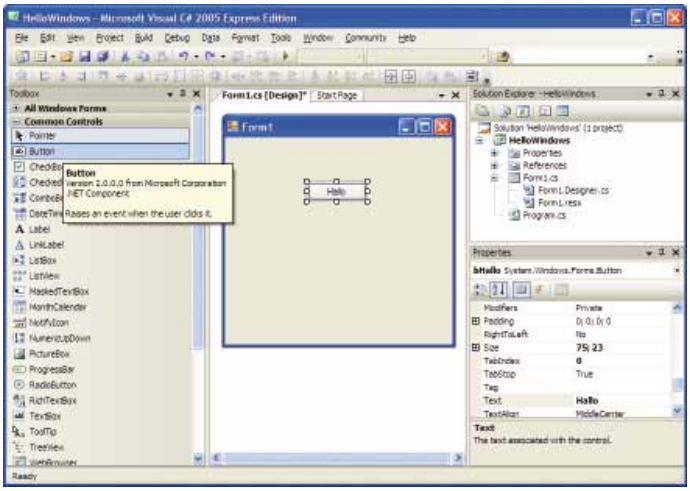
Een klik in de designer, één regel code, en er gebeurt al iets. We verwijzen je voor meer van zulke snelle resultaten naar de leerzame video's op de website van Microsoft [1]. Maar helaas, Visual C# is Powerpoint niet en alleen de interface van Windows-applicaties kun je met 'sleur&pleur' blijven programmeren. Om je goede gevoel te houden, zul je uiteindelijk willen weten hoe de structuur van een Windows-programma er écht uitziet. Eerst laten we je zien uit welke .cs-bestanden je project nu eigenlijk bestaat en wat ze doen, en daarna hoe je je programma kunt laten reageren op de vele mogelijke events.

Kijk als eerste eens naar de klasse Form1, waar de methode bHallo_Click() een deel van is. Ogenschijnlijk is die verantwoordelijk voor het hele form van de applicatie, maar kan dat in zó weinig regels als het screenshot laat zien? Ook zie je in de constructor van Form1 een oproep van de tot nu toe onbekende methode InitializeComponent(). Deze blijkt niet van de royaal uitgeruste basisklasse Form te zijn afgeleid. Waar is de lijm, waar zit de methode Main() en waar zijn de objecten voor het form en de Button-control?

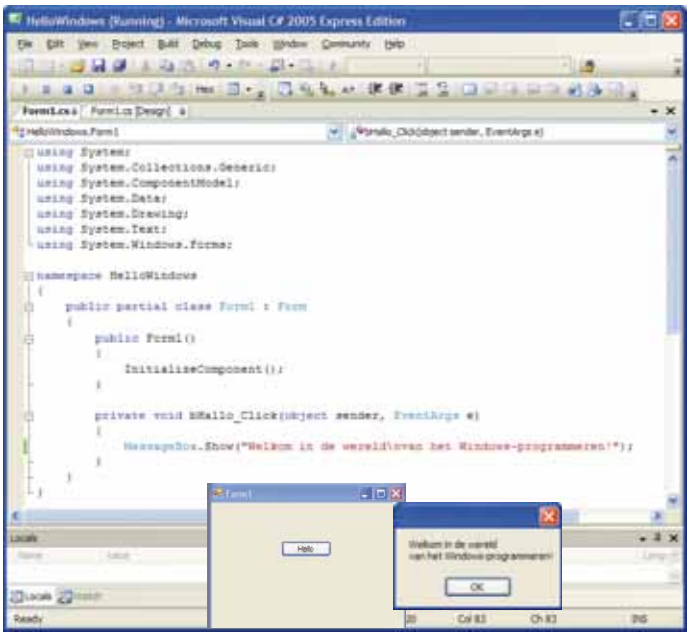
Wie gewend is met .NET 1.0 te werken, fronst zijn wenkbrauwen, want Form1.cs was daar veel langer, inclusief Main(). Het keyword partial, dat nieuw is in C# 2.0, levert de hint: de klassedefinitie van Form1 is onvolledig. De designer heeft nog een tweede bestand gegenereerd, dat de klassedefinitie aanvult: Form1.Designer.cs. In Visual C# is het enigszins verborgen onder de node Form1.cs in de Solution Explorer; in SharpDevelop is het zelfs niet direct zichtbaar. Hier vind je alle code die de designer op eigen houtje genereert en onderhoudt. Feitelijk is dat



Het projectsjabloon 'Windows-application' genereert het geraamte voor een werkend Windows-programma.



In de 'design view' van onze nieuwe Windows-applicatie kunnen we ons form ontwerpen.



Omdat de form designer niet alleen het geraamte van de click-handlermethode aanmaakt in Form1.cs, maar hem ook meteen koppelt aan de eigen GUI-code in Form1.Designer.cs, is de code meteen 'actief'. Een druk op de button tovert een messagebox met een tekst naar keuze tevoorschijn – en daar was slechts één regel code voor nodig.

In het kielzog van de gebeurtenissen

Als je een multitasking-besturingssysteem als Windows bent, moet je ermee leren leven dat je gebruikers hun invoer vaak in nogal onlogische volgorde geven. Ze klikken in een venster, geven drie letters van een zoekterm in, gaan naar een ander venster, scrollen daar wat rond, starten nog een derde programma en switchen nog voordat het helemaal gestart is alweer terug om ongeduldig te zien hoe ver de zoekopdracht al gevorderd is.

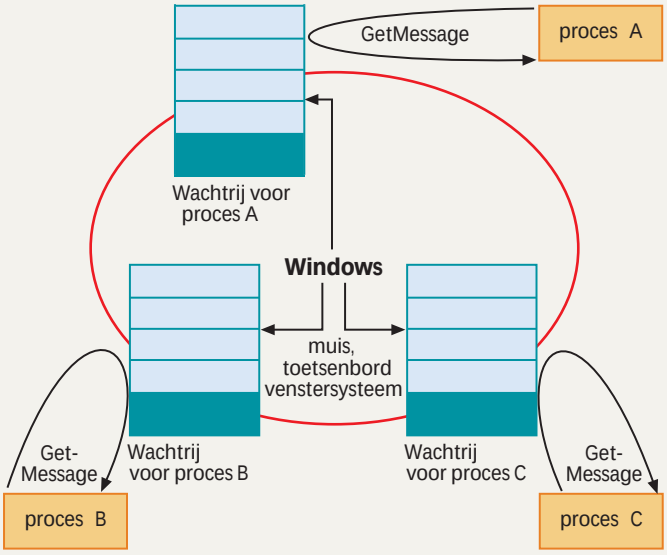
Om de menselijke wanorde in geordende vorm aan applicaties te presenteren, wijzen Windows en vergelijkbare systemen aan elk proces een speciale buffer toe. In deze zogenaamde *event-queues* worden events als muiskliks en toetsaanslagen opgeslagen, maar ook berichten over andere gebeurtenissen. Windows verdeelt (*dispatcht*) events over deze queues volgens een vaste logica: toetsaanslagen gaan bijvoorbeeld principieel naar het proces dat op dat moment het actieve venster heeft. Voor muis-events zoals kliks, beweging of een verandering van de vorm van de muiscursor geldt een soortgelijk verhaal: afhankelijk van de muispositie en het op die plek zichtbare venster komen die terecht in de eventqueue van het bijbehorende proces, zelfs al is dat venster niet actief of misschien elders overdekt door andere vensters. Vandaar dat ook voor een leeg bureaublad een proces nodig is: alle kliks op dit 'venster' worden afgehandeld door explorer.exe.

Al die processen kunnen natuurlijk de zo opgestapelde events weer uit de wachtrijen verwijderen. Puur technisch bekeken is het opvragen van een element uit de eigen wachtrij via GetMessage() niet meer en niet minder dan de oproep van een willekeurige systeemfunctie van Windows – waarbij in de syntaxis meteen diverse addertjes schuilen. Het belangrijkste punt is het pull-principe: applicaties roepen GetMessage() zelf regelmatig op, dus op een moment waar het in hun eigen logica past zodat ook events als muisklik zondermeer plotseling kunnen binnenvallen – ze worden dan gewoon later verwerkt.

De complete Windows-programmering werkt dus event-gestuurd: applicaties halen een event uit hun wachtrij, reageren erop en halen het volgende event op. Wat gebeurt er als bij GetMessage() de queue leeg is? Dan zet Windows de oproepende applicatie net zolang op inactief totdat er weer iets gebeurt. Vanuit de applicatie gezien betekent dit gewoon dat een oproep van GetMessage() wel eens minuten of uren kan duren. Op zijn Nederlands: je programma wordt pas weer actief als er ook iets te doen is.

Dat achter het cyclisch opvragen van de queue een lus zit en dat die het hoofddeel vormt van de methode Run() , is nu niet verrassend meer. Hiermee is echter nog niet alles gezegd: zoals het tweede plaatje laat zien, ziet het verwerken van de events binnen een applicatie er weer iets ingewikkelder uit.

Als je met de muis op het oppervlak van een knop button1 klikt, zet Windows een WM_CLICK in de wachtrij van de applicatie. Die wordt op een gegeven



Windows verdeelt events direct over de *queues* ofwel wachtrijen van de afzonderlijke processen. Het uitlezen van die queues gebeurt in een eventlus van de processen zelf (pull-principe).

een complete beschrijving van het GUI-design, maar dan in de vorm van code. Zo declareert Form1.Designer.cs onder andere je knop bHallo van het type Button als dataveld van Form1. Daarnaast zie je in de code een nieuwsgierig makend dichtgeklapt gebied met de naam 'Windows Form Designer generated code', omsloten door #region/#endregion-directives. Als je deze 'fold' open klikt met het plus-icon op de linker editorrand vind je dan de methode InitializeComponent().

In deze 'verborgen' methode komen eindelijk alle puzzelstukjes samen. Hier wordt bHallo namelijk geïnitialiseerd met een nieuwe Button-instantie en vervolgens komen alle properties aan bod die je in de designer view zelf had ingesteld:

```
this.bHallo.Location = new System.Drawing.Point(87, 90);
this.bHallo.Size = new System.Drawing.Size(50, 29);
this.bHallo.Text = "Hallo";
this.bHallo.Click += new System.EventHandler(this.bHallo_Click);
```

De positie en de grootte van de button volgen uit het (x,y) coördinatensysteem dat voor vensters wordt gebruikt – de pixel linksboven is altijd (0,0). Op de Click-eigenschap komen we zo terug.

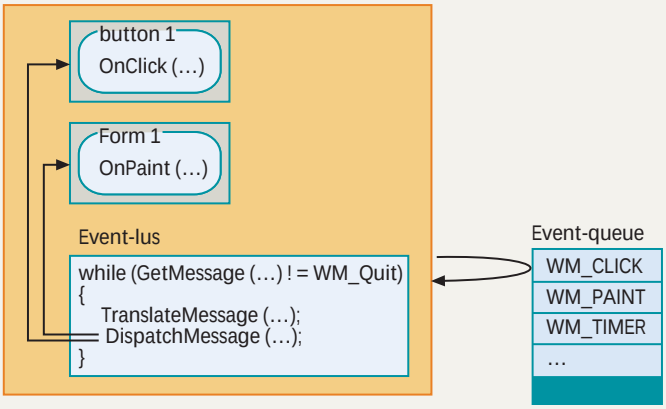
Onder het commentaarkopje 'Form1' worden tenslotte verschillende (blijkbaar overgeërfde) eigenschappen van het form geïnitialiseerd. Belangrijk is hier vooral dat je nieuwe Button-object wordt opgenomen in de Controls-container. Dat is een overerfd en tijdens runtime vooraf geïnstantieerd object van het type ControlCollection, dat een verzameling vormt van alle controls op het form-object. Op die manier kun je al die controls doorlopen met bijvoorbeeld een foreach-lus.

Aan het begin en eind van InitializeComponent() worden voor de veiligheid respectievelijk de methoden SuspendLayout() en ResumeLayout() aangeroepen. Zo kan er tijdens het layouten van het form geen interactie plaatsvinden, totdat alle controls geïnitialiseerd zijn. De components-eigenschap en ook de methode Dispose() vormen de basis voor verdergaande concepten in samenhang met applicatieoverkoepelend bruikbare componenten (bijvoorbeeld COM- of COM+-componenten). Voorlopig mag je deze negeren.

Het bestand Form1.Designer.cs is opzettelijk ietwat verstopt, omdat je als programmeur met dit bestand dikwijls niets hoeft te doen. Ook al staat de 1:1-relatie tussen code en design bij C# niet alleen in de reclame en past de designer view het form meestal automatisch aan als je handmatig wijzigingen in dit bestand aanbrengt, dan nog is het alleen iets om in speciale gevallen te gebruiken. Bijvoorbeeld om foutmeldingen van de compiler op te lossen, in het geval dat je in Form1.cs handmatig een eventhandler hebt verwijderd, maar er wel nog naar verwezen wordt in Form1.Designer.cs. Je eigen initialisaties horen thuis in het hoofdbestand (Form1.cs), typisch in de constructor na de InitializeComponent()-oproep.

De eventlus

Maar je programma bevat nog een derde en laatste .cs-broncodebestand, dat de methode Main() huisvest waar elk C#-programma mee begint. Sinds



Een Windows-applicatie leest zijn event-queue in een lus uit en verdeelt event voor event aan de objecten die verantwoordelijk zijn voor de bijbehorende venstergebieden.

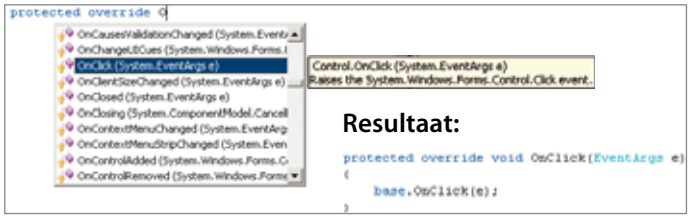
moment gelezen, vertaald, intern toegewezen – en belandt uiteindelijk bij de OnClick()-methode van het button1-object.

Hoe bouw je je programma nu zo, dat als het een forse taak moet uitvoeren de normale eventverwerking toch doorgaat? De gebruiker mag immers niet de indruk krijgen dat het programma niet meer reageert. Voor simpele multitasking is er de methode Application.DoEvents(), die je in noodgevallen ergens in je code kunt inlassen. Deze methode dwingt het lezen en verwerken van alle volgende events af en geeft de controle pas na diens verwerking terug aan de oproeper. Zo kan de eventverwerking doorgaan, ook als de Application.Run()-lus nog steeds zit te wachten tot het eerste event afgewerkt is. Kijk wel uit wat je doet: handmatig DoEvents() aanroepen kan oneindige loops veroorzaken, bijvoorbeeld als het programma precies gedurende een DoEvents()-eventverwerking moet worden afgesloten. In dergelijke gevallen moet je het FormClosing-event dat tijdens het afsluiten optreedt, afvangen met een toestandsvariabele, anders blijft de eventhandler voortdurend indirect zichzelf aanroepen.

.NET 2.0 genereert het sjabloon 'Windows-application' ook Main() in een apart bestand (Program.cs) met de klasse Program. Wel zo netjes, inderdaad. De Main() daarin definieert het centrale commando voor het starten van een Windows-applicatie:

```
Application.Run(new Form1());
```

De methode Run() zet de zogenaamde eventlus van de applicatie op, start een nieuwe instantie van het Form1-formulier en bombardeert deze tot hoofdvenster van de applicatie. Dat betekent vooral dat het afsluiten van het venster ook het programma afsluit.



Visual Studio 2005 biedt erg prettige ondersteuning om virtuele methodes te overschrijven. Het keyword override activeert de IntelliSense-functie, die je alle in de huidige klassencontext overschrijfbare elementen op een bordje aanbiedt en vervolgens complete methodengeraamtes genereert, inclusief de oproep van de base-versie.

Zoals het kader hiernaast uitlegt, zorgt de eventlus van je programma ervoor dat events worden doorgegeven aan de objecten die ze moeten ontvangen – in een gemiddeld C#-programma dus de Form-objecten en de control-objecten daarop. De code daarvoor overerven deze ontvangstobjecten automatisch van de gezamenlijke basisklasse Control. Van deze klasse overerft Form1 een reeks eigenschappen en virtuele methodes die met event-signaling te maken hebben. Een afgeleide klasse kan zelf het event-mechanisme uitbreiden door nieuwe (secundaire) events en bijbehorende handlers te definiëren, en vervolgens zo'n event op te wekken (raisen).

Hoe event-management precies werkt, demonstreren we met een voorbeeld van een enkele muisklik. Andere events worden volgens hetzelfde schema afgehandeld. Om te beginnen gaan we uit van de virtuele methode OnClick() in de klasse Control, met het prototype

```
protected virtual void
  OnClick(EventArgs e)
{ ... }
```

Telkens als een Control-object een zogenaamde WM_CLICK-message ontvangt, roept het deze virtuele methode aan met een zogenaamd EventArgs-object. Interessant is dat zo'n EventArgs-object zelf informatievrij is, want deze klasse definieert geen datavelden. De OnClick-signature volgt hier echter een conventie van het .NET-eventmechanisme: de signatures voor On*Xxx*-routines moeten voor het aanbieden van een eventcontext precies één EventArgs-parameter hebben. Feitelijk ontvangt OnClick() een object van de van EventArgs afgeleide klasse MouseEventArgs. Na een typecast kun je dus in principe ook nauwkeurigere gegevens over het muisevent uitlezen.

Een afleiding van Control, zoals de voorbeeldklasse Form1, zou OnClick() kunnen overschrijven en zo specifiek op muiskliks reageren:

```
protected override void
OnClick(EventArgs e)
{
    base.OnClick(e);
    MouseEventArgs mea =
          (MouseEventArgs)e;
    this.test ="Klik in het form op "+
          mea.Location.ToString();
}
```

Het oproepen van de basisklasse met base.OnClick() is meestal nodig om ook het inwendige object een kans te geven het event af te werken.

Delegates

Dit soort Click-afhandeling werkt, maar is niet praktisch. De routine die een event afhandelt moet los kunnen staan van bijvoorbeeld de Button-klasse waar het optreedt. Bovendien kan het voorkomen dat een event door meerdere controls moet worden afgehandeld. Het wordt een rommeltje om op deze manier meerdere eventhandlers op een event te laten reageren. Gelukkig kan het mooier en bevat het .NET-eventmechanisme de klasse EventHandler. Objecten van deze klassen zijn een voorbeeld van *delegates*.

Je bent er al één tegengekomen, namelijk bij het Click-event van het Button-object bHallo:

```
bHallo.Click += new System.
          EventHandler(bHallo_Click);
```

Je moet je de *delegate* bHallo.Click voorstellen als een object van een speciaal type, dat één of meerdere pointers naar methodes met dezelfde signature kan bevatten. Het toevoegen van een nieuwe eventhandler aan een delegate-object gebeurt met de operator +=, en met de operator -= verwijder je een delegate weer uit de lijst. Sinds .NET 2.0 staat de compiler ook de vereenvoudigde syntax

```
bHallo.Click += bHallo_Click;
```

toe. Om de verzamelde oproepen uit te voeren, roep je gewoon de delegate zelf aan als ware het een functie, met de parameters die hij volgens zijn signature nodig heeft. Als een methode meerdere malen aan de delegate is toegevoegd, wordt hij ook meerdere keren uitgevoerd. Probeer het maar eens.

Het onderwerp delegates verdient een eigen voorbeeld. Laten we eens een eigen delegate-type declareren:

```
delegate void MyLDelegate(ref long val2);
```

Dit houdt in, dat alle methoden die je later aan MyLDelegate toevoegt, een call-by-reference long parameter moeten hebben:

```
MyLDelegate myL;
myL = LLinLuit;
..
long LLinLuit (ref long v)
{
return ++v;
}
```

Nu kunnen we allerlei handlers gaan toevoegen aan de delegate:

```
MyLDelegate myL1 = LLinLuit;  // +1
myL1 += LLinLuit;       // +1
myL1 += myL1;           // +2
myL1 -= LLinLuit;       // -1
myL1 -= LLinLuit;       // -1
```

Voor het oproepen van een delegate gebruik je de methode Invoke(), maar de compiler staat ook toe dat je hem weglaat. Het uitvoeren van een delegate met een lege lijst methoden levert een run-time error op.

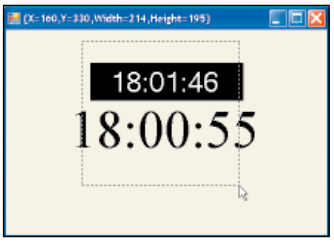
```
long val = 0;
myL1.Invoke(ref val);    // +2
myL1(ref val);  // +2
if (val==4)   // true
```

Overigens maken dit soort terugroepacties en events geen deel uit van de eventlus, maar worden ze direct uitgevoerd. Wat hierboven wellicht ingewikkeld oogt, kost in de praktijk echter nauwelijks moeite, omdat de Form Designer normaal gesproken het meeste werk hiervan automatisch voor je doet. Het Click-event is als standaardevent van de Button-control gedefinieerd, zodat een dubbelklik direct vanuit de ontwerpweergave naar het gepast uitgerust methodengeraamte leidt. Om ook voor andere events handlers te implementeren, bijvoorbeeld Paint, waar het kader hier onder over spreekt, is er in het properties-venster de Events-weergave (via het bliksemsymbool inschakelbaar). Als je op een entry dubbelklikt, krijg je direct het codegeraamte voor een nieuwe, automatisch van een naam voorziene afhandelmethode voor je neus. Om een event te verbinden met een al zelf geschreven afhandelingsmethode met de vereiste signatuur, kies je die uit het ernaast staande combinatieveld.

Codevoorbeeld: rubber band selection

Zoals ook in de vorige delen, presenteren we ter demonstratie een voorbeeldprogramma waarmee je het bovenstaande kunt bekijken en oefenen. Ook dit keer weer is dat een kleine animatie.

Wat je direct opvalt, is de tekstuitvoer: die ziet er bij een Windows-applicatie anders uit dan bij een consoleprogramma. Om snel iets te debuggen of voor een simpele statusmelding wordt daarvoor overigens vaak



Het programma Rubberband tekent een geanimeerde selectierechthoek, puur als reactie op muis-events.

de titelbalk misbruikt, maar vooral controls als Label, TextBox of zelfs RichTextBox zijn prima geschikt voor eenvoudige tekstuitvoer. Textboxes kunnen daarnaast meerdere regels beslaan en zijn ook geschikt voor invoer. Het kost meer werk om expliciet tekst te *tekenen* naar het client-gebied van het form via de Paint-opdracht. Niet zo verbazingwekkend, maar die

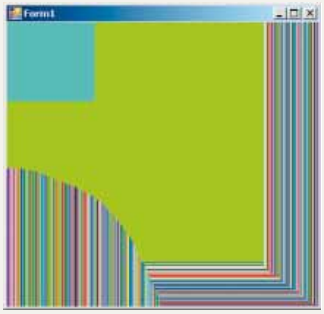
Zichtbaar bijgeschaafd

In de documentatie van Windows-programma's lees je af en toe dat vensters zichzelf opnieuw tekenen. Daarmee wordt formeel bedoeld dat Windows een WM_PAINT-message in de queue van een proces plaatst. Pure Windows-programma's hebben sinds jaar en dag een 'window procedure' waarin deze met veel omhaal wordt opgevangen, maar in onze veilige C#-virtuele machine signaleert het ontvangerobject gewoon een signaal voor een Paint-event en handelt dit af door zichzelf opnieuw te tekenen.

Een Form-klasse implementeert voor dit event meestal een handler, die door de designer automatisch als Form*x*_Paint() wordt aangemaakt. Daarin zijn alle tekenacties samengevoegd, voor zover die nog niet door controls (in hun vensters) worden afgehandeld. Het staat je natuurlijk vrij om buiten de Paint-events om en dus zonder automatische verversing iets in een venster te tekenen, wat ons voorbeeldprogramma Rubberband ook doet, maar dan moet je de daarvoor nodige zaken zoals de 'graphics context' eerst zelf ophalen (welkom in het jargon van GDI+, de graphicsbibliotheek van .NET). De Paint-handler krijgt die graphics context in zijn parameters cadeau, net als de informatie welk deel van de vensterinhoud volgens Windows opnieuw getekend moet worden. Het standaardvoorbeeld daarvan is dat een gebruiker van actief venster switcht, waarbij een eerder afgedekt gebied van venster X zichtbaar wordt. Op dat moment voegt Windows in de queue van proces X een WM_PAINT-message met de bijbehorende coördinaten in.

Een tweede situatie waarbij Paint-events optreden, is als je een venster vergroot of creëert (dat laatste kun je zien als een vergroting van nul naar x pixels). De derde en eigenlijk laatste situatie is dat een programma via de Form-methode Invalidate() een gedeelte van zijn venster op eigen houtje vrijgeeft en zo Windows dwingt om een bijbehorende WM_PAINT-message in de queue te plaatsen.
Windows hertekent vrij slim: het voegt meerdere Paint-berichten bij elkaar en beperkt tekenacties tot het gebied dat daadwerkelijk opnieuw getekend moet worden. Dit is een optimalisatie uit de tijd dat de transfersnelheid naar de grafische kaart nog een bottleneck vormde.

Het voorbeeldprogramma RepaintDemo demonstreert dit gedrag op bonte wijze. Het definieert een handler voor Paint-events die niks anders doet dan het hele venster (ClientRectangle) in te kleuren. Omdat de kleur via een randomgenerator willekeurig wordt gekozen, is die bij elke oproep weer anders. Zo kun je goed zien hoe veel WM_PAINT-berichten Windows verstuurt, al hertekent het alleen de gebieden die door de laatste verandering zichtbaar zijn geworden.
(Wij hadden de Beeldscherm/Effecten-eigenschap 'Inhoud van het venster weergeven tijdens het slepen' van Windows aanstaan.)



Ondanks dat RepaintDemo telkens de hele vensterinhoud opnieuw wil inkleuren, beperkt Windows dat tot het gebied dat echt is vrijgekomen. Het groene gebied ontstond direct bij de start, daarna vergrootten we het window door te trekken aan de hoek rechtsonder.

```
1 public partial class Form1 : Form
2 {
3   Random r = new Random();
4   public Form1()
5   {
6     InitializeComponent();
7   }
8   private void Form1_Paint(object sender, PaintEventArgs e)
9   {
10    Color c = Color.FromArgb(
11      r.Next(0, 255), r.Next(0, 255), r.Next(0, 255));
12    SolidBrush b = new SolidBrush(c);
13    e.Graphics.FillRectangle(b, this.ClientRectangle);
14  }
15 }
```

opdracht biedt je dan ook wel alle mogelijkheden die je van Windows-applicaties kent. We hebben in deze C#-artikelreeks helaas te weinig plek voor een overzicht van de vele GDI+-klassen in het .NET-framework, waaronder penselen, pennen, kleuren, gebieden, enzovoort. De beste manier om hiermee te leren omgaan is aan de hand van de Windows-help of leerboeken.

Onze Paint-routine (vanaf regel 64 in de listing) maakt een Font-object uit de Times-familie met een fontgrootte van 48 pt aan en berekent in de eerste stap welk formaat in beeldpunten voor de outputstring (actuele tijd in hh:mm:ss) zal hebben. Hieruit berekent het dan de punt linksboven voor een gecentreerde uitgave. De tekenoperatie gebeurt via DrawString() – een van de vele methodes van het Graphics-object uit de eventcontext van de handler.

Zonder verdere maatregelen zou de weergave van de tijd alleen dan worden bijgewerkt als het venster (om welke reden dan ook) opnieuw moet worden getekend. Voor een ietwat levendigere tijdweergave hebben we daarom het form voorzien van een Timer-control, dat we met de eigenschappen Enabled en Interval meteen geactiveerd hebben. Dit control genereert meteen vanaf de programmastart om de 300 ms een Tick-event. Onze Tick-handler levert vervolgens de actuele tijd in hh:mm:ss-formaat om de Text-eigenschap van een Label-object te vullen. Dat laatste heeft op zijn beurt een zwarte achtergrond en witte tekst, zodat het net een digitale klok lijkt.

De derde feature die het programma demonstreert is de techniek van de 'rubber band selection'. Denk daarbij aan de manier van selectie in Windows Verkenner of in tekenprogramma's, waarbij je bepaalde inhoud selecteert door met een ingedrukte muistoets een rechthoekig gebied uit te rekken. Deze selectie biedt Windows noch .NET kant-en-klaar aan, dus moeten we de animatie handmatig programmeren, om precies te zijn door het handlen van de drie muis-events MouseDown, MouseMove en MouseUp en incrementele tekenoperaties. De logica is redelijk eenvoudig, maar er moet een hoop worden gedaan:

- Form1_MouseDown() bewaart de huidige muispositie als eerste (en vaste) hoekpunt in een dataveld van het form-object.
- Form1_MouseMove() overschildert indien nodig de keuzerechthoek van de vorige animatiefase, berekent uit de actuele muispositie de nieuwe grootte van de keuze en tekent de muisrechthoek voor de nieuwe animatiefase.
- Form1_MouseUp() wist de laatst getekende keuzerechthoek.

Je vindt gelukkig in het .NET-framework ControlPaint.DrawReversibleFrame(), oftewel een tekenoperatie op XOR-basis met de mooie eigenschap dat elke tweede oproep met dezelfde parameters het resultaat van de eerste oproep weer onzichtbaar maakt. Omdat deze methode absolute beeldschermcoördinaten verwacht, moet de MouseMove-handler de in de eventcontext e gevonden vensterrelatieve coördinaten óf omrekenen (de uitgecommentarieerde coderegels) óf de muispositie van Control.MousePosition() in absolute coördinaten opvragen. Je ziet snel dat de tekenoperatie direct en dus onafhankelijk van het Paint-resultaat naar het outputvenster tekent, wat normaliter ook geen probleem oplevert – maar niet altijd, zo blijkt, want er schuilt een adder onder het gras.

Het huiswerk:

- Verander het programma zo dat ten eerste ook de door de Paint-routine uitgegeven tijd om de seconde wordt bijgewerkt en ten tweede dat na een grootteverandering van het form meteen gecentreerde uitvoer plaatsvindt. Zoek uit waarom de correcte oplossing een nare bijwerking heeft op de rubberbandani-

Het programma Rubberband demonstreert verschillende soorten informatie-output in één form. We zetten de tijd op het scherm met de control 'label1' en door direct een string naar het form te tekenen. Als de gebruiker zijn linker muisknop indrukt, tekent het programma een gestippelde rechthoek met behulp van paint-opdrachten.

```
1 using System;
2 using System.Drawing;
3 using System.Windows.Forms;
4
5 namespace Rubberband
6 {
7    public partial class Form1 : Form
8    {
9      Rectangle rectGum;
10     const string Titel = "Rubberband - selecteer gebied met muis" ;
11     public Form1()
12     {
13        InitializeComponent();
14        Text = Titel;
15     }
16
17     private void Form1_MouseDown(
18        object sender, System.Windows.Forms.MouseEventArgs e)
19     {
20        if ((e.Button & MouseButtons.Left) > 0) // linker muisknop?
21        {
22           // rectGum.Location = PointToScreen(new Point(e.X, e.Y));
23           rectGum.Location = Control.MousePosition;
24        }
25     }
26
27     private void Form1_MouseMove(
28        object sender, System.Windows.Forms.MouseEventArgs e)
29     {
30        if ((e.Button & MouseButtons.Left) > 0) // linker muisknop?
31        {
32           if (this.ClientRectangle.Contains(new Point(e.X, e.Y)))
33           {
34              ControlPaint.DrawReversibleFrame(
35                 rectGum, this.BackColor, FrameStyle.Dashed);
36              rectGum.Size =
37                 (Size)Control.MousePosition - (Size)rectGum.Location;
38              // rectGum.Size = (Size) PointToScreen(
39              // new Point(e.X, e.Y)) - (Size) rectGum.Location;
40              ControlPaint.DrawReversibleFrame(
41                 rectGum, this.BackColor, FrameStyle.Dashed);
42              Text = rectGum.ToString();
43           }
44        }
45     }
46
47     private void Form1_MouseUp(
48        object sender, System.Windows.Forms.MouseEventArgs e)
49     {
50        if ((e.Button & MouseButtons.Left) > 0) // linker muisknop?
51        {
52           ControlPaint.DrawReversibleFrame(
53              rectGum, this.BackColor, FrameStyle.Dashed);
54           rectGum.Size = new Size(0, 0);
55           Text = Titel;
56        }
57     }
58
59     private void timer1_Tick(object sender, EventArgs e)
60     {
61        label1.Text = DateTime.Now.ToLongTimeString();
62     }
63
64     private void Form1_Paint(object sender, PaintEventArgs e)
65     {
66        string tijd = DateTime.Now.ToLongTimeString();
67        Font fnt = new Font("Times", 48);
68        SizeF sz = this.ClientRectangle.Size
69               - e.Graphics.MeasureString(tijd, fnt);
70        PointF p = new PointF(sz.Width / 2, sz.Height / 2);
71        e.Graphics.DrawString(tijd, fnt, Brushes.Black, p);
72     }
73  }
74 }
```

Compileren met Mono

Na onze vorige artikelen kregen we aardig wat lezerreacties over het uitvoeren van de scripts onder Mono. We zijn dan ook wat uitleg verschuldigd aan degenen die onze codevoorbeelden uit de vorige delen willen compileren met deze open-source .NET-implementatie. Dat bleek niet helemaal triviaal, omdat Mono pas sinds kort sommige .NET2-commando's ondersteunt. Je hebt hiervoor namelijk Mono 1.16 of nieuwer nodig. Maak een buildfile met de naam BeeldschermTennis.rsp met de inhoud:

```
/target:exe
/out:BeeldschermTennis.exe
/reference:mscorlib.dll
  Program.cs
```

Start nu een shell (onder Windows de Mono-shell) en gebruik daar de compiler met de .NET2-omgeving (dus niet mcs):

```
gmcs @BeeldschermTennis.rsp
```

Nu heb je een BeeldschermTennis.exe, die je kunt uitvoeren met

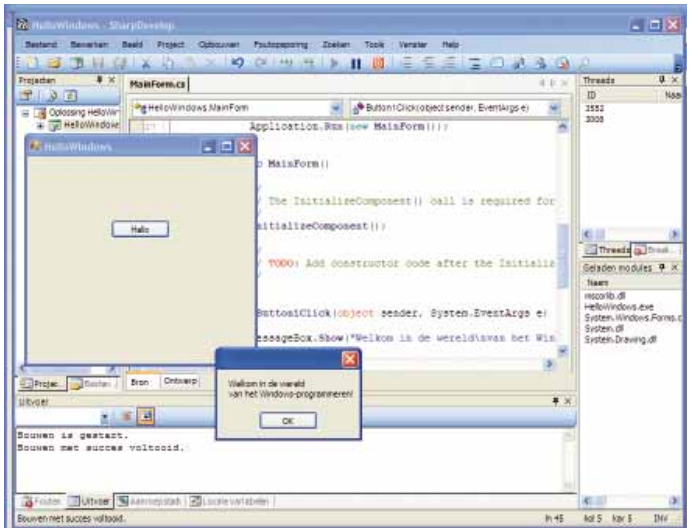
```
mono BeeldschermTennis.exe
```

Voor RepaintDemo gaat het ongeveer hetzelfde, maar dan met de volgende rsp-file:

```
/target:winexe
/out:RepaintDemo.exe
/reference:mscorlib.dll
/reference:System.Windows.Forms.dll
/reference:System.Drawing.dll
/reference:System.Data.dll
  Program.cs
  Form1.cs
  Form1.Designer.cs
```

Hoe je dit vergelijkbaar doet in Mono-IDE's als SharpDevelop en MonoDevelop, lees je o.a. op http://www.devx.com/opensource/Article/32085.

matie (niet gewiste animatiefasen) en implementeer een workaround met behulp van een toestandsvariabele.
- Teken een gevulde kleurige rechthoek met de coördinaten van het laatste met de rubberband gekozen gebied naar het clientgebied van het form (FillRectangle)
- Implementeer de complete rubberbandfunctie door alleen het MouseMove-event te bewerken.



De open-source IDE SharpDevelop 2.0 is geheel Nederlandstalig en bezit veel van de functionaliteit van Visual C# Express.

Refactoring

Na al deze praktijk is het tijd voor bezinning. Want al ben je een snelle programmeur, je bent hooguit een zesde van de tijd bezig met programmeren. En dat geldt voor alle programmeertalen, niet alleen voor C#. De rest van de tijd ben je kwijt aan alle andere fasen van het software engineering-proces. Je analyseert aan welke specificaties je programma moet voldoen (requirement engineering) en maakt misschien een ontwerp. Je bent bezig met testen: unittests om alle details van bijvoorbeeld een klasse te controleren en integratietests voor het geheel. Daarnaast is er nog de kwaliteitscontrole, het versiebeheer, het buildsysteem en de documentatie. Veel programmeertijd gaat bovendien op aan het aanpassen van bestaande code, die je soms niet eens zelf hebt geschreven.

Doorgaans veroorzaakt het repareren van defecten in zulke code in 20 tot 50% van de gevallen nieuwe defecten [4], vaak op een andere plaats. Daarom kun je slecht te begrijpen code het beste aanpakken door in veel kleine stapjes systematisch wijzigingen te maken, en dat heet *refactoring*. In hun boek geven Martin Fowler en Kent Beck talloze voorbeelden van refactoring-technieken, die elk telkens hooguit enkele pagina's beslaan [5].

Eerst schrijf je een klein testprogramma, zodat je snel geautomatiseerd kunt controleren of je nieuwe code straks nog precies doet wat de oude goed deed. Het makkelijkst plaats je deze testcode ergens in de te verbeteren klasse zelf. Voor kleine refactorings (lange methods opsplitsen, variabelen renamen) bevatten veel IDE's zoals Visual Studio en Eclipse nog handige hulpfuncties, maar daarna volgt onvermijdelijk het echte werk: methoden verplaatsen naar andere klassen, tijdelijke variabelen vervangen door methoden (gevolg: een grotere scope), voorwaardelijke logica vervangen door polymorfisme, typen vervangen door (sub)klassen, enzovoort. Na elke kleine verandering compileer je opnieuw en controleer je je wijziging met het testprogramma. Er bestaan allerlei technieken voor het repareren van de tientallen soorten 'code smells' (gedupliceerde code, te grote klassen, te lange argumentlijsten, data clumps…).

Het schrijven van tests voor je klassen is trouwens sowieso een goed idee. Het is één van de onderdelen van de zogenaamde *extreme programming*-techniek, die ook bij grote softwarebedrijven steeds populairder wordt, om zulke controlefuncties al te schrijven vóórdat je begint met het echte programmeerwerk. De tijd die je daarmee verliest win je later weer terug, en het levert kwalitatief betere en leesbaardere code op.

Toekomst

Je hebt nu je eerste stappen in GUI-design gezet. Hoe je een user interface het meest ergonomisch vormgeeft, is een heel eigen vakgebied, dat op zijn Nederlands mens-machine-interactie heet. Een vuistregel daarbij luidt dat de minst opvallende GUI's vaak de beste zijn. Het ontwerpen van user interfaces is overigens erg hot: er staat een nieuwe Microsoft-ontwikkeling op stapel waarbij je in de Visual Studio-designer je GUI's niet genereert in C#, maar in de op XML gebaseerde opmaaktaal XAML. Op die manier zijn ze ook bruikbaar voor websites. Microsoft zal hiervoor begin 2007 drie nieuwe tools introduceren onder de naam 'Expression', die verweven zijn met Visual Studio.

In deel 6 laten we je meer zien over Windows-programmeren. We kijken verder naar controls, leggen *attributes* (de C#-term voor annotations) uit en we gaan onze eigen componenten schrijven, die ook hun eigen events genereren.

Literatuur

[1] Video's: http://msdn.microsoft.com/vstudio/express/visualcsharp/learning/default.aspx#forms

[2] Microsoft Expression-lijn: www.microsoft.com/products/expression/en/default.mspx

[3] Charles Petzold, Programming Microsoft Windows with C#, Microsoft Press, 2001.

[4] Frederick P. Brooks jr; The Mythical Man-Month, essays on software engineering, Addison-Wesley, 1975.

[5] Martin Fowler; Refactoring, Improving the Design of Existing Code, Addison-Wesley, 1999.

[6] Philip Chu Seven habits of highly effective programmers, http://www.technicat.com/writing/programming.html

[7] Hoe het níet moet: The Daily WTF, Curious Perversions in Information Technology, www.thedailywtf.com

ct