Ir. Jacques de Hooge

IronPython: Programmeren met plezier

DYNAMIC TYPING LEIDT TOT FLEXIBEL ONTWERP

Hebt u ook wel eens het gevoel dat een programmeertaal u voor de voeten loopt? Ondanks vele talen met talloze uitbreidingen lijkt het leven van de ontwikkelaar er soms niet makkelijker op te worden. Schoolvoorbeeld is natuurlijk C++. Toegegeven, een powertool voor real-time en rekenapplicaties. Maar wat begon als een 'kleine' taal is uitgegroeid tot een ondoordringbaar woud van taalfaciliteiten, die soms haaks op elkaar staan. C# en Java dragen minder historie met zich mee en zijn daardoor minder topzwaar. In dit artikel bespreekt de auteur de programmeertaal IronPython en gaat hij in op twee praktijksituaties.

Spreekt 'lean and mean' u aan en houdt u van leren door doen? Maak dan eens kennis met IronPython, een lichtgewicht objectgeoriënteerde programmeertaal, die zich in een toenemende populariteit mag verheugen. Wat u kunt verwachten:

- Heldere, leesbare code
- Volledige toegang tot .NET Framework
- Ongeëvenaarde productiviteit

Python

Python is een geïnterpreteerde programmeertaal. Omdat Python een krachtige ingebouwde set datastructuren en bijbehorende geoptimaliseerde bewerkingen heeft, is de performance goed. Het geïnterpreteerde karakter van de taal zorgt ervoor dat de effecten van wijzigingen binnen enkele seconden zichtbaar zijn. Dit maakt de taal geschikt voor rapid prototyping in samenspraak met gebruikers.

Python is objectgeoriënteerd en ondersteunt modules. Dit werkt ordelijke, leesbare programma's in de hand. De taal biedt de volgende voorzieningen voor het werken met objecten:

- **Vrijwillige inkapseling.** Members waarvan de naam begint met __ zijn slechts op een speciale manier van buiten de klasse toegankelijk.
- **Meervoudige overerving.** Het gaat hierbij om implementatie-overerving, niet slechts om interface-overerving. Complexiteit zoals virtuele overerving is vermeden.
- **Dynamische veelvormigheid.** Alle datastructuren mogen references naar objecten van welke klasse dan ook bevatten. Ze hoeven geen gemeenschappelijke voorouders te hebben. De interface van een willekeurig object is runtime op te vragen.

```
class A: #Basisklasse
   def __init__(self,x): self.__x=x #Constructor
   def show(self): print self.__x,

class B(A): #B erft van A
   def __init__(self,x,y): A.__init__(self,x); self.__y=y
   def show(self): A.show(self); print self.__y,

class X: #Geen familie van A en B
   def show(self): print'Hi, I am an X',

objects=A(1),B(2,3),X() #Tuple assignment
for anObject in objects: anObject.show (); print
```

Codevoorbeeld 1. Objecten

Codevoorbeeld 1 'Objecten ' laat het gebruik van inkapseling, overerving en veelvormigheid in Python zien.

Python is net als C# en Java reference-based. Variabelen bevatten geen objecten, maar references naar objecten. Indien er geen enkele reference meer naar een object bestaat, wordt het door de garbage collector opgeruimd. Bij IronPython is dat de .NET-garbage collector. Dit houdt in dat ook cyclic references geen probleem zijn.

Python behandelt alles als objecten, ook integers, floats en zelfs functies. Dit houdt in dat het bijvoorbeeld mogelijk is runtime uit te vinden of een variabele een integer, string, float of iets anders bevat. Codevoorbeeld 2 'Typeinfo' laat zien hoe.

Dat een functie ook een object is, zorgt ervoor dat een functie bij haar creatie informatie uit de omgeving kan meenemen. Daarnaast kent Python de lambda-notatie om functies te definiëren op de plaats waar ze worden aangeroepen. Zoals blijkt uit codevoorbeeld 3 'Functies' is dit handig bij callback-functies met een vast aantal parameters.

De grote kracht van IronPython is echter dynamic typing. Typeinformatie hoort niet bij een variabele, maar bij het object waar de variabele naar verwijst. Een variabele die naar een ander soort object gaat wijzen, verandert dus van type. Wie is opgevoed met strictly typed gecompileerde talen als C# en Pascal moet misschien even slikken. Geen vaste datatypen? Maar dan is elke variabele in principe een void pointer… en de weg naar de hel is geplaveid

```
class Goblywook: pass
things=4711,'Eau de Cologne',3.14,Goblywook()

for thing in things:
   print repr(thing),
   if thing.__class__==int: print'is an integer'
   elif thing.__class__==str: print'is a string'
   elif thing.__class__==float: print'is a float'
   else: print '\nis a '+repr(thing.__class__)
```

Codevoorbeeld 2. Typeinfo

met void pointers, toch? Er is echter een cruciaal verschil: aan een Python-variabele kun je vragen naar wat voor soort object hij verwijst, aan een void pointer niet. Het checken van het gerefereerde type gebeurt meestal onzichtbaar door de interpreter. Met andere woorden: daar waar void pointers bijna garant staan voor geheugenlekken en crashes, geeft dynamic typing flexibel maar gecontroleerd gedrag. Codevoorbeeld 4 'Dynatype' laat zien hoe je daar plezier van kunt hebben.
Python heeft een eenvoudige 'gezond verstand' syntax. Een treffend voorbeeld daarvan is het gebruik van inspringen in plaats van accolades om blokken (groepen statements) aan te geven. Geen nieuw idee, dat geef ik toe, maar het voorkomt fouten door vergeten accolades.

Python + .NET = IronPython

IronPython is Python, draaiend boven op .NET. Uiteindelijk wordt de code uitgevoerd door precies dezelfde virtuele machine die ook uw managed C#-code uitvoert. Anders dan bij C# is er bij IronPython echter geen sprake van rechtstreekse compilatie naar MSIL, maar van een efficiënte vorm van interpretatie.
De virtuele machine van .NET is een beproefd werkpaard. Dat geïnterpreteerde IronPython-code uiteindelijk op deze virtuele machine wordt uitgevoerd, leidt tot een stabiele, veilige werking. Daarnaast is er een sprake van een goede integratie met andere managed code. Dit houdt in dat vanuit IronPython gebruikgemaakt kan worden van bijvoorbeeld code die geschreven is in C#. IronPython is daarmee bijvoorbeeld geschikt voor scripting van uw C#-applicaties. IronPython is echter veel meer dan een scripttaal. Heel de .NET CLR is vanuit IronPython te gebruiken. Dankzij de snelheid en de goede faciliteiten voor objectoriëntatie blijkt het een aantrekkelijke taal voor het ontwikkelen van applicaties als geheel. Codevoorbeeld 5. 'Timer' laat zien hoe een instance van System.Windows.Forms.Timer kan worden gebruikt om in IronPython meerdere callback-functies met verschillende intervallen te activeren.
Gebruik van .NET CLR-objecten vanuit Python lijkt zo sterk op gebruik vanuit C#, dat de C#-documentatie goed bruikbaar is voor ontwikkeling met IronPython. Klassen zoals ListView en TreeView gebruiken vanuit IronPython is één ding, maar de functionaliteit van die klassen uitbreiden in IronPython gaat een stap verder. Het bleek in IronPython zonder meer mogelijk ListView en TreeView uit te breiden met respectievelijk in-place editing en drag-and-drop reordering.

Ervaringen met IronPython bij Fugro-Jason

Fugro-Jason is gespecialiseerd in toepassingen voor het opsporen van aardolie en gas. Kleine explosies zenden geluidsimpulsen de bodem in. Deze worden weerkaatst door gesteenten met verschillende akoustische eigenschappen. Het weerkaatste signaal wordt digitaal opgeslagen. Het verwerken van deze 'ruwe seismiek' vergt snelle computers en slimme rekenprogrammatuur.

```
def next(callback): return callback(1) #callback moet 1 param hebben
def prev(callback): return callback(-1)
def adder(base,step): return base+step #heeft 2 params, 1 te veel
def doubler(base,step): return base * 2**step

def context1():
   base=100
   print next(lambda step: adder(base,step)), #1 param over: step
   print next(lambda step: doubler(base,step))

def context2 ():
   base = 1000
   print prev(lambda step: adder(base,step)),
   print prev(lambda step: doubler(base,step))

context1(); context2()
```

Codevoorbeeld 3. Functies

```
def asList(itemOrList): #Parameter itemOrList heeft geen vast type
   if itemOrList.__class__==list: return itemOrList #Test objecttype
   else: return [itemOrList]

for thing in 1,'one',False,1.1,[2,'two',True,2.2]:
   print'\nthing: ',
   for item in asList (thing): print repr (item),
```

Codevoorbeeld 4. Dynatype

Doel is uit de uitgesmeerde, verruiste reflecties een nauwkeurig model te maken van de positie en grootte van olie- en gashoudende structuren. Vanwege de gewenste hoge verwerkingssnelheid is de gebruikte programmeertaal voor het rekenwerk C++. Op basis van het verkregen model kan de stroming van olie en gas worden gesimuleerd. Zo kan een schatting worden gemaakt van de winbare reserves. Tevens worden de resultaten gebruikt voor het bepalen van een optimale productiestrategie. Twee ontwikkelingen werkten het gebruik van IronPython daarbij in de hand:

1. Python werd al gebruikt voor scripting van rekenapplicaties. De flexibiliteit van IronPython en de geschiktheid voor rapid prototyping maakten deze taal een voor de hand liggende keuze voor vervaardiging van de GUI's van deze applicaties.
2. Met de toename van de stabiliteit, adresruimte en netwerkcapaciteiten is er een verschuiving richting Windows merkbaar. Veel gebruikers hebben tegenwoordig een Windows-machine in plaats van een Unix-workstation op hun bureau staan.

De wens wat betreft softwareontwikkeling is enerzijds de Unix-gebruikers onverminderd te blijven bedienen, anderzijds te anticiperen op het toenemende Windows-gebruik. Vanwege de vereiste stabiliteit heeft op Windows een native oplossing daarbij de voorkeur boven een toepassing waarbij lagen afkomstig van twee of meer vendors met elkaar moeten samenwerken. In dit laatste geval is de ervaring dat kleine versieverschillen voor hardnekkige problemen kunnen zorgen.
IronPython bovenop .NET is zo'n native oplossing. Als GUI-library werd WinForms gebruikt. Dit gebeurt via een encapsulatielaag, zodat op Unix via hetzelfde API een andere GUI-library gebruikt kon worden. Bij twee projecten werd van IronPython gebruikgemaakt. Het ene project betrof het reconstrueren van onderaardse gesteentestructuren uit de seismiek. Het tweede project betrof het simuleren van de stroming

```
import clr
clr.AddReference('System');clr.AddReference('System.Windows.Forms')
from System.Windows import Forms

class IntervalTimer: #Multi-interval timer based upon Forms.Timer
   def __init__(self):
      self.tasks=[]; self.secs=[]; self.secsLeft=[]
      t=Forms.Timer(); self.timer=t #Use Forms.Timer as workhorse
      t.Interval=100; t.Tick+=self.tick; t.Start()

   def addTask(self,intervalSecs,task):
      self.secs.append(int(10 *intervalSecs))
      self.secsLeft.append(0); self.tasks.append(task)

   def tick(self,sender,event):
      for i,task in enumerate(self.tasks):
         if not self.secsLeft[i]: task(); self.secsLeft[i]=self.secs[i]
         self.secsLeft[i]-=1

form=Forms.Form ()
def yeller(yell): form.Text=(yell+form.Text)[:80]
it=IntervalTimer ()
it.addTask(1,lambda: yeller('hey ')); it.addTask(5,lambda: yeller('HO '))
form.ShowDialog()
```

Codevoorbeeld 5. Timer

van olie en gas in die structuren. Het gebruik van IronPython ging in beide gevallen om de vervaardiging van een GUI om interactief Python berekeningsscripts samen te stellen. Op de auteur van dit artikel na, had geen van de betrokkenen voorafgaand aan deze projecten enige Python-ervaring. In een introductiecursus van drie dagen leerden de betrokkenen en een aantal andere belangstellenden binnen Fugro-Jason de taal. Alle betrokkenen waren ervaren C++-programmeurs. Concepten die ook in C++ voorkomen, zoals inkapseling, overerving en veelvormigheid, bleek men in IronPython snel te kunnen toepassen. Daarnaast was het een grote winst om ook te kunnen profiteren van aspecten van IronPython die niet in C++ terug te vinden zijn, zoals dynamic typing, lokale functies en de ingebouwde datastructuren. Dit zijn aspecten die invloed hebben op de ontwerpstijl, en de beste manier bleek te zijn ontwerpen samen met de betrokkenen te reviewen en suggesties te doen over alternatieven. Met name de overstap van static op dynamic typing was hierbij een eye-opener. Omdat het om twee nieuwe applicaties ging, veranderden de requirements tijdens het ontwikkelen op basis van opgedane ervaringen. Dit leidde tot een voortdurende stroom van wijzigingen in de applicatielogica. Applicatielogica is een onderbelicht aspect bij GUI-builders. Hier geldt: What you see, is all you get. Als je een GUI bij elkaar sleept en klikt met een GUI-builder, dan heb je daarmee nog geen subject-observer-gedrag, enabling/disabling van controls, situatie-afhankelijk doorlopen van wizards, messages en foutmeldingen. Bij de betreffende applicaties moet de GUI de gebruiker door de stappen van een complexe workflow heenleiden. Een prototype bestaande uit een aantal statische windows uit een GUI- designer is onvoldoende om toekomstige gebruikers een beeld te geven van de uiteindelijke feel van een applicatie. Met IronPython bleek het mogelijk de hele applicatie, inclusief logica, op te bouwen in kleine stappen, soms van dagen, soms slechts van minuten. Deze benadering moet niet worden overdreven. Er is niets mee mis als een ontwikkelaar zich na een aantal prototyping-stappen enkele weken terugtrekt om het ontwerp te consolideren. Als dit achterwege blijft, ontstaat wegwerpcode. Perioden van consolidatie werden afgewisseld met perioden van voortdurende feedback.
Door het gebruik van dynamic typing en doordat in IronPython relatief weinig code nodig is om een bepaald doel te bereiken, bleek de code over het algemeen eenvoudig te reorganiseren.

Praktische patterns

Er wordt wel beweerd dat de keuze van een programmeertaal uiteindelijk een sluitpost is. Een goed ontwerp valt in elke programmeertaal uit te drukken, zo luidt het dogma. Theoretisch is dat waar, hoeveel lagen er ook tussen uw sourcecode en de processor zitten, uiteindelijk had u uw ei ook in assembler kunnen leggen. De praktijk blijkt anders. Met name het ver doorgevoerde 'everything is an object' van IronPython blijkt te leiden tot een consequente ontwerpstijl, resulterend in compacte, heldere code. Een goed voorbeeld hiervan is de manier waarop een aantal gebruikelijke problemen rond de toepassing van het observer pattern in IronPython kon worden opgelost. Het observer pattern is misschien wel het bekendste design pattern. Als de waarde van objecten X, Y en Z afhankelijk zijn van de waarde van object A, B en C, kun je objecten A, B en C bij waardeverandering X, Y en Z rechtstreeks laten aanpassen. Een betere manier is vaak, X, Y en Z een notification te sturen, waarna ze zelf hun nieuwe waarde berekenen. Op die manier is elk object slechts verantwoordelijk voor berekening van zijn eigen waarde: loose coupling op z'n best. Helaas wordt toepassing van het observer pattern bijna altijd geplaagd door een drietal problemen:

1. In complexe netwerken van objecten hebben de notifications de neiging te gaan rondzingen. Omdat het versturen van een notification message naar aanleiding van een andere notification message neerkomt op een recursieve call, resulteert dit uiteindelijk in een stack overflow.
2. Een ander probleem van het gebruik van het observer pattern zijn multipath updates. In een netwerk van objecten die elkaar notificeren, kunnen notificaties langs meerdere wegen van ongelijke lengte hetzelfde doelobject bereiken. Het gevolg is dat dit doelobject meerdere malen een update pleegt. Indien het doelobject gekoppeld is aan een GUI-element, leidt dit tot hinderlijk knipperen van het scherm. Meestal wordt dit met allerlei lapmiddelen, zoals het suspenden van schermupdates, opgelost.
3. Een derde probleem is rollback. Indien ergens in de keten van updates een object een ongeldige waarde aanneemt en een exception triggered, dienen alle objecten hun oorspronkelijke waarde te herkrijgen. Dit is niet triviaal. Het houdt onder andere in dat die oorspronkelijke waarde bij onomkeerbare operaties gecached moet worden.

In IronPython bleek het mogelijk alle drie deze bezwaren eenvoudig te ondervangen. De eigenschap van IronPython dat ook standaard datatypen zoals integers, floats, strings en booleans objecten zijn, is daarbij cruciaal. Elke waarde van een variabele bij IronPython wordt op dezelfde manier voorgesteld: als een reference naar een object. Dit geldt onverschillig of die waarde nu een datastructuur van vele megabytes of bijvoorbeeld simpelweg een integer is. Caching komt daarom altijd neer op het onthouden van de vorige reference. Oude objectwaarden blijven in stand, omdat de garbage collector ze pas weggooit als er geen verwijzingen meer naar zijn. Rollback komt altijd neer op het vervangen van de nieuwe reference door de oude reference. Dit alles zonder de in andere talen benodigde uitzonderingen voor standaard datatypen, gebruik van templates of boxing/unboxing. Omdat de vorige waarde van elk object sowieso wordt gecached, is het mogelijk de waarden van X, Y en Z uit te drukken in de vorige waarden van A, B en C in plaats van in de huidige waarden. Zo worden rondzingen en stack overflows voorkomen. Multipath updates, tenslotte, worden voorkomen door eerst alle notifications uit te voeren en daarna, in een aparte fase, eenmalig alle updates. Deze principe-oplossingen zijn al zeker twintig jaar oud, maar vergden tot nu toe altijd vele bladzijden kwetsbare code doorspekt van, in historische volgorde: void pointers, templates of generic typing. In IronPython is het, voorzien van veel commentaar, één bladzijde.

Ir. Jacques de Hooge is als zelfstandig softwareontwikkelaar, projectbegeleider en kwaliteitsmanager werkzaam via zijn bedrijf Geatec Engineering sinds 1987. Daarnaast geeft hij in-house trainingen: onder andere IronPython, C++ , C#, requirements-analyse, objectgeoriënteerd ontwerp, gestructureerd testen, real time systemen en kwaliteitsmanagement. Website: www.geatec.com, Email: info@geatec.com

Referenties

IronPython: www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython
IronPython freeware library gebruikt bij projecten uit artikel: www.qquick.org
Fugro-Jason: www.fugro-jason.com
Python algemeen: www.python.org
IronPython blog: http://blogs.msdn.com/ironpython/