Centrum Wiskunde & Informatica

DOMEIN-SPECIFIEKE TALEN: MEER MOGELIJK MAKEN MET MINDER CODE

Tijs van der Storm, senior researcher SWAT group

Publication date: 09-09-2019

Programmeertalen zijn er in vele soorten en maten. Sommige staan dicht hoe bij de computer werkt (bv. C en C++), andere staan verder van de hardware af en hebben een hoger abstractieniveau. Een hoger niveau van abstractie stelt de programmeur in staat zich minder druk te maken over incidentele details, en zich meer te concentreren op het probleem dat opgelost moet worden.

DSLs zijn een verregaande vorm van dit idee: computertalen toegespitst op een specifiek applicatiedomein. Hiermee kun je dus niet álle problemen oplossen, maar sommige problemen juist heel efficiënt en effectief. Programmeurs, en soms zelfs eindgebruikers, specificeren software-oplossingen met een nadruk op *wat* er nodig is, in plaats van de computer te instrueren *hoe* het resultaat bereikt moet worden.

DSLs zijn geen nieuw idee. De eerste DSL was waarschijnlijk APT, een taal om numerieke controlsystemen mee te besturen, ontworpen in 1956[1]. Sindsdien zijn DSLs veelvuldig succesvol toegepast. Enkele bekende voorbeelden zijn SQL voor het bevragen van databases, HTML voor webpagina opmaak, DOT voor graafvisualisatie, of "make" voor het configureren en executeren van softwarebouwprocessen. Tegenwoordig zijn DSLs opnieuw populair door de opkomst van "low code" platforms (bv. Mendix, Outsystems), en geavanceerde language workbenches die de constructie van DSLs eenvoudiger en doeltreffender maken.
De sidebar laat een illustratief voorbeeldprogramma (geïnspireerd op de belastingaangifte) zien in de "Questionnaire Language" (QL), een simpele, educatieve DSL toegespitst op het domein van interactieve vragenformulieren en enquêtes[2]. Naast de code staat een afbeelding van het resultaat: een user interface waarin de gebruiker de vragenlijst kan beantwoorden.

QL bestaat uit een verzameling vragen met een label (bv. "Kocht u een huis in 2018?") en een naam (bv. huisGekocht). Elke vraag heeft een type (bv. boolean, money) dat bepaalt hoe het formulier getoond wordt aan de gebruiker; een vraag van het type boolean, bij voorbeeld, wordt hier getoond als een ja-nee selectie-element. Vragen kunnen conditioneel getoond worden en sommige "vragen" worden berekend door middel van een expressie in termen van gegevens die de gebruiker invult (bv. restWaarde).

Figuur 1: QL: een DSL voor vraagformulieren en enquêtes

```
form aangifte {
  "Kocht u een huis in 2018?"
    huisGekocht: boolean

  "Heeft u een hypotheek?"
    hypotheek: boolean

  "Verkocht u een huis in 2018?"
    huisVerkocht: boolean

  if (huisVerkocht) {
    "Wat was de verkoopprijs?"
      verkoopPrijs: money
    "Wat was u privéschuld voor het huis?"
      schuld: money
    "Uw restwaarde is:"
      restWaarde: money = verkoopPrijs - schuld
  }
}
```

Voordelen van DSLs

Ondanks de eenvoud van QL, illustreert de taal al veel van de voordelen van DSLs. De meeste van deze voordelen zijn een gevolg van de scheiding tussen *wat* en *hoe* : de gebruiker specificeert *wat* , de taal-engineer realiseert *hoe* .

DSL code abstraheert van low-level implementatiedetails en vermijdt "boilerplate" code. Een gevolg hiervan is dat DSL code vaak ordegroottes kleiner is dan traditionele programmacode. In QL bijvoorbeeld, is geen spoor te bekennen van hoe GUI events afgehandeld worden, en hoe en wanneer rekenvragen uitgerekend worden; die logica zit onder de motorkap van de QL compiler. Een ontwerper van een vragenlijst hoeft dus geen ervaren GUI programmeur te zijn om QL te gebruiken.

Een meer specifieke en minder expressieve notatie betekent ook dat er minder vrijheidsgraden voor de programmeur zijn, met als gevolg dat de kans op fouten lager wordt. Programmeerfouten als null-pointer dereferencing en oneindige loops

zijn bij voorbeeld niet mogelijk in QL omdat de taal eenvoudigweg niet toestaat om code met dat soort fouten op te schrijven. Daarbij komt dat domeinspecifieke notatie het mogelijk maakt betere foutdetectie en optimalisatie te doen. In het geval van QL zou de compiler cyclische afhankelijkheden kunnen detecteren (bv. "restwaarde: money = schuld - restwaarde") en hierover zinvolle foutmeldingen rapporteren. Dit is vrijwel ommogelijk als de vragenlijst direct in Java geprogrammeerd zou zijn; de Java compiler heeft immers geen kennis van het domein van vragenlijsten. Eenzelfde redenering geldt voor domein specifieke optimalisaties, die buiten het bereik van gewone programmeertalen liggen.

Tot slot is DSL code minder afhankelijk van een specifiek executieplatform. Als gevolg hiervan kan dezelfde applicatie op basis van verschillende platforms gerealiseerd worden en naar nieuwe technologieën geporteerd worden, zonder dat de DSL code zelf te hoeven aanpassen. QL als taal is bij voorbeeld onafhankelijk van of executie plaatsvindt via traditionele desktop GUI frameworks, webgebaseerde UIs, of mobiele platforms.

Language Engineering met Language Workbenches

De voordelen die DSLs software-ontwikkeling kunnen bieden, zijn niet gratis: er moet immers een taal ontwikkeld worden, met parsers, compilers, checkers, interpreters, IDEs, documentatie, etc. DSL implementaties zijn bovendien zelf software die getest, onderhouden, en gedocumenteerd moet worden. Hoe kan DSL ontwikkeling effectiever en laagdrempeliger gemaakt worden?

Dit is precies wat language workbenches proberen te bereiken. Martin Fowler noemde het een potentiële "killer-app for domain-specific languages"[3]. Language workbenches kunnen gezien worden als compiler-compilers "on steroids": geïntegreerde omgevingen om *alle* aspecten van taalontwikkeling te ondersteunen, inclusief Integrated Development Environment (IDE) faciliteiten als syntax highlighting, outline views, foutmarkering, jump-to-definition, incrementeel bouwen etc. Een overzicht en vergelijking van de meest bekende language workbenches, zowel industrieel als academisch, is te vinden in referentie 2.

Language workbenches verhogen de productiviteit van DSL ontwikkeling door veel aspecten van taalimplementatie te automatiseren. Hierdoor komt meer tijd vrij voor het *ontwerp* van de taal, waar de ware uitdaging ligt: geen DSL is vaak beter dan een slecht ontworpen DSL. Een goed begrip van het domein is essentieel om precies *die* aspecten in de taal te modelleren die er toe doen, en om te voorkomen dat onnodige complexiteit wordt toegevoegd. Language workbenches ondersteunen het ontwerpproces met krachtige gereedschappen, waardoor een eerste prototype vaak in een week klaar kan zijn, in plaats van maanden.

In de Software Analysis & Transformation (SWAT) onderzoeksgroep aan het CWI ontwikkelen we de Rascal metaprogrammeertaal en language workbench[4]. Rascal is een geïntegreerde programmeertaal voor de ontwikkeling van source code analyse- en transformatietools, waaronder DSL compilers en IDEs. Enkele voorbeelden van DSLs die met Rascal zijn ontwikkeld zijn Machino (voor routerconfiguratie), MicroMachinations (voor spel-economieën), Rebel (voor financiële producten), en Derric (voor digital forensics). Verder wordt Rascal gebruikt om DSL engineering aan de Rijksuniversiteit van Groningen te onderwijzen, aan de

hand van QL, en aan de UvA, TU/e en OU in vakken over software analyse en evolutie.

De ervaring leert dat tools als Rascal de productiviteit van DSL ontwikkeling enorm kan verhogen. Je zou Rascal kunnen zien als een DSL voor het ontwikkelen van DSLs: meer mogelijk maken met minder code, in het domein van DSLs zelf. Om deze ervaring breder in te zetten is in 2017 de spin-off SWAT.engineering[5] opgericht, die nu voor diverse klanten DSLs ontwerpt.

Zoals zo veel stijlen en technieken voor software-ontwikkeling zijn DSLs geen panacee. Maar een goed ontworpen DSL voor een goed gedefinieerd domein, gemaakt met de juiste gereedschappen, kan productiviteit en kwaliteit naar een nieuw plan tillen.

[1] APT programming language

[2] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, *et al.* *Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future* Computer Languages, Systems & Structures, Volume 44, Part A, December 2015, pp 24-47

[3] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*

[4] Rascal: the one-stop shop for metaprogramming.

[5] SWAT.engineering

Dit artikel verscheen eerder in de *AG Connect van 31 juli 2019*

Tijs van der Storm *is senior onderzoeker bij de SWAT groep (Software Analysis & Transformation) aan het Centrum Wiskunde & Informatica (CWI) te Amsterdam. Daarnaast is hij parttime hoogleraar aan de Rijksuniversiteit Groningen (RUG).*

ADDRESS

CWI Location:	Science Park 123 1098 XG Amsterdam NETHERLANDS
Postal address:	P.O. Box 94079 1090 GB Amsterdam NETHERLANDS
Phone:	+31 20 592 9333
E-mail:	INFO@CWI.NL