TOOLS

**Xtext is onderdeel van openArchitectureWare, een toolset bovenop Eclipse die de implementatie en het gebruik van domein-specifieke talen (DSLs) vereenvoudigt. In dit artikel laten we zien hoe Xtext gebruikt kan worden om tekstuele domein-specifieke talen te maken en te integreren in het ontwikkelproces.**

# Domein-specifieke Talen met Xtext

**Jeroen Benckhuijsen en Dick Eimers**
zijn werkzaam bij Atos Origin BAS Emerging Technologies, waar ze zich bezighouden met Java en open-sourcetechnologie.

In dit artikel willen wij het fenomeen Domain Specific Languages (DSLs) en meer specifiek het realiseren ervan met behulp van het openArchitectureWare Xtext Framework[1] beschrijven. We laten zien hoe Xtext gebruikt kan worden om DSLs te maken en te integreren in het ontwikkelproces.

Hoewel de term DSL wellicht niet iedereen bekend zal voorkomen, is het iets waar we in de dagelijkse ontwikkelpraktijk veel mee werken. Bekende DSL's zijn SQL, HTML, maar ook de MS Excel macrolanguage. Aan de andere kant van het spectrum vinden we General Purpose Languages (GPL), zoals Java en C#. Het onderscheid is niet altijd even goed te maken en is afhankelijk van de definitie. Zo kan COBOL aangemerkt worden als een GPL, maar ook als een DSL voor bedrijfsapplicaties. Wat we van belang vinden, is dat een DSL het voordeel heeft dat de concepten van een bepaald probleemdomein beter uitgedrukt kunnen worden dan in een GPL. Een database query is bijvoorbeeld beter uit te drukken in SQL dan in Java.

Het gebruik van DSLs zien we steeds meer terug in Model-gedreven softwareontwikkeling (MD*). Dit is een manier van softwareontwikkeling, waarbij modellen een intrinsiek onderdeel zijn van de oplossing in plaats van slechts aanwezig ter illustratie en documentatie. Modellen kunnen grafisch zijn – denk aan UML en BPEL – maar ook tekstueel, zoals SQL en verschillende XML dialecten. De meest bekende vorm van MD* is MDA (Model Driven Architecture) van de Object Management Group. Momenteel krijgen vooral de meer pragmatische varianten van model-gedreven softwareontwikkeling veel aandacht. Een andere vorm is Model Driven Software Development, waarvan openArchitectureWare de meest prominente implementatie vormt. Xtext is onderdeel van openArchitectureWare dat weer een onderdeel is van het Eclipse Generative Modeling Technologies (GMT) project. OpenArchitectureWare is wat Martin Fowler zou noemen een language workbench. Een language workbench faciliteert zowel de definitie als het gebruik van DSLs. Andere voorbeelden van dergelijke workbenches die momenteel redelijk veel aandacht genieten in de industrie zijn Microsofts Software Factories en de nog ietwat in geheimzinnigheid gehulde Intentional Domain Workbench van Intentional Software Corporation, medeopgericht door ex-Microsoft medewerker, ruimtetoerist en miljardair Charles Simonyi.

## Xtext tooloverzicht

Xtext integreert een aantal tools in Eclipse om meerdere facetten van modelgedreven software ontwikkeling te ondersteunen;

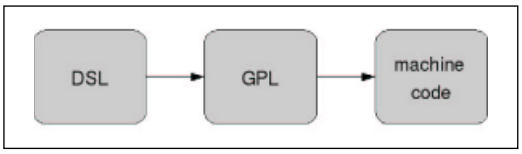
Gegeven een grammatica genereert Xtext een metamodel, een parser en een volledig geïntegreerde editor met moderne features als syntax highlighting en code completion. Met behulp van de gegenereerde editor kan een gebruiker van de DSL zeker weten dat zijn programma syntactisch correct is. Daarnaast is een tool, genaamd Check, beschikbaar om semantische regels te definiëren en modelvalidatie uit te voeren.

Vervolgens kunnen ander e-tools binnen oAW gebruikt worden, bijvoorbeeld om model-naar-model

transformaties te definiëren met xTend en eventueel model-naar-code transformaties met xPand.

## Van DSL tot uitvoerbaar programma

Een DSL heeft meerdere toepassingen, maar in alle gevallen zal een DSL één of meer transformaties ondergaan om te komen tot een uitvoerbaar programma. In Figuur 1 zijn de verschillende fasen weergegeven waarin een programma geschreven in een DSL zich binnen een framework als oAW typisch kan bevinden. De eerste transformatie van de DSL zal resulteren in een programma in een taal zonder specifiek domein (GPL). In de laatste stap zal bestaande tooling rond de GPL gebruikt worden om deze te compileren naar machine code die uitgevoerd kan worden in de executieomgeving. In het geval van oAW is de taal uiteraard Java en dan ligt het voor de hand dat deze wordt gecompileerd tot Java bytecode, daarna zal executie in een Java Virtual Machine het gevolg zijn.




Figuur 1: Typische transformaties van DSL tot uitvoerbaar programma

De verschillende toepassingen van het gebruik van een DSL komen voort uit het tijdstip waarop deze transformaties gedaan worden: tijdens ontwikkeling of tijdens het uitvoeren van het programma. In het eerste geval spreken we van generatie – de DSL wordt met templates gegenereerd naar Java code – of in het tweede geval van configuratie – de DSL wordt ingeladen tijdens uitvoering en hiermee worden verschillende Java classes geconfigureerd conform het GoF Builder pattern[4]. In dit artikel behandelen we een voorbeeld van een DSL voor run-time configuratie.

Figuur 2 : De Agreement DSL

```
plan LowPay

    rate BASE_RATE
        1999 - 10 - 01 : 10.0 [USD/KwH]
        2008 - 04 - 16 : 15.37 [USD/KwH]

    rate REDUCED_RATE
        1999 - 10 - 01 : 5.0 [USD/KwH]

    rate CAP
        1999 - 10 - 01 : 50.0 [USD/KwH]

    event USAGE
        1999 - 10 - 01 : amount : IF usage > CAP THEN BASE_RATE * usage ELSE REDUCED_RATE * usage
                         account: base-usage

    event SERVICE_CALL
        1999 - 10 - 01 : amount : 10.0 [USD]
                         account : service
        1999 - 12 - 02 : amount : fee * 0.5 + 15.0 [USD]
                         account : base-usage

    event TAX
        2000- 02- 29 : amount : fee * 0.055
                       account : tax
```

## Configuratie: XML vs. DSL

Vaak zien we dat XML gebruikt wordt in plaats van DSL voor configuratie of generatie. Dit heeft een aantal voor- en nadelen ten opzichte van de definitie van een eigen DSL, waarvan we er enkele noemen:

Voordelen:

- XML is bekend bij veel ontwikkelaars
- Tools voor de definitie zijn veel voorhanden en bekend
- Frameworks voor het parsen behoren tot de standaard gereedschapskist

Nadelen:

- XML documenten kunnen vrij complex en slecht leesbaar worden. Binnen jBPM wordt bijvoorbeeld een grafische editor geleverd om dit probleem op te lossen. XML is dan slechts een opslagformaat.
- XML biedt niet de volledige uitdrukkingskracht en syntaxmogelijkheden van een DSL.
- Hoewel volwaardige editors met code-completion standaard aanwezig zijn, bevatten deze meestal geen semantische controles.

## Agreement DSL

Om de werking van de tools uit het framework te demonstreren ontwikkelen we een echte DSL. Het voorbeeld dat we gebruiken wordt door Martin Fowler uitgewerkt in z'n nog niet gepubliceerde – maar online te bezichtigen boek[2] – over domeinspecifieke talen. Deze DSL, genaamd de Agreement DSL, is speciaal ontworpen om een programma omtrent overeenkomsten voor diensten tussen partijen en de financiële consequenties van gebruik hiervan uit te drukken.

Zie het voorbeeld van een programma geschreven in de Agreement DSL in Figuur 2 voor een fictief energiebedrijf. Er zijn een aantal events gedefinieerd en deze events hebben financiële consequenties. De details van de consequenties van een bepaalde dienst kunnen veranderen, bijvoorbeeld de prijsstijging van energie per kilowattuur (kWh) per 1 december. Vandaar dat de DSL notie heeft van tijd. Afhankelijk van de periode heeft een 'event' uitwerking op een bepaalde rekening ('account') tegen een bepaald bedrag ('amount'). Ter vergelijking: als we het voorbeeld van Figuur 2 direct uitdrukken in Java, dan zou het er ongeveer uit kunnen zien als in Figuur 3.

Het lijkt ons duidelijk dat het bovenstaande minder goed leesbaar is. Of beter, het is moeilijker om de business logica te lezen door de Java-syntax heen. In dit voorbeeld is de implementatie van de engine, de verschillende classes die de implementatie van Fowler's Agreement Dispatcher pattern[3] vormen, overigens achterwege gelaten.

Martin Fowler is trouwens van mening dat de

DSL oplossing in Figuur 2 geschikt zou kunnen zijn om te gebruiken als communicatiemiddel. Dit in tegenstelling tot de GPL oplossing in Figuur 3. Een programma uitgedrukt in een DSL zou immers (met enige hulp) beter door niet-programmeurs te begrijpen moeten zijn. Waardoor een domeinexpert mogelijk in staat is om in de DSL geschreven code te reviewen en onregelmatigheden te herkennen. Nog een stap ambitieuzer is de DSL die door de domeinexpert wordt gebruikt om zelf de business logica uit te drukken. (Het betrekken van 'de business' bij het definiëren van software in welke vorm dan ook is in het verleden geen groot succes gebleken, maar we laten ons graag positief verassen!)
We zullen nu het stappenplan bespreken voor de definitie van de nieuwe DSL:

## Stap 1 : Eclipse project definities

De definitie van een nieuwe DSL gebeurt op basis van een nieuw Xtext project, dat vanuit Eclipse te creëren valt als oAW geïnstalleerd is. Er worden standaard drie projecten aangemaakt:

1. Het project voor de definitie van de DSL. Hierin worden ook de gegenereerde parser en het metamodel geplaatst. In de klassieke compilerconstructie noemen we dit typisch het 'front end'.
2. Het tweede project is voor de Eclipse-editor, die gebruikt gaat worden tijdens het ontwikkelen middels de DSL.
3. Het derde en laatste project dat in Eclipse zal verschijnen is voor de code generatie (ook wel bekend als het 'back end'). Binnen het huidige voorbeeld gebruiken we dit project niet, zoals later duidelijk zal worden.

## MD* vs. Compiler Constructie

Voor wie bekend is met theorie van klassieke compiler constructie doet de terminologie in de MD* gemeenschap mogelijk als eigenzinnig aan. Een paar korte vertalingen:
- DSL: concrete grammatica
- Meta Model: abstracte grammatica
- Tekstuele representatie van het model: concrete syntax tree
- Model: abstract syntax tree
- Model Validation: Semantische analyse

## Stap 2 : DSL definitie

De definitie van een DSL met XText start met het definiëren van de grammatica van de taal. Dit gebeurt in de .xtxt file die automatisch gegenereerd wordt. In Figuur 4 wordt een deel van de grammatica getoond. De grammatica bestaat uit een aantal productieregels. Een productieregel bestaat uit gereserveerde symbolen en keywords die tussen aanhalingstekens staan, zoals plan , andere productieregels, en built-in tokens waardoor een waarde wordt toegekend aan een attribuut. Voorbeelden van deze built-in tokens zijn ID voor strings en INT voor getallen. Deze zijn impliciet aanwezig om onnodige herdefinitie van dergelijke productieregels te voorkomen.

```
public Agreement setupLowpay() {
    StaticAgreement agreement = new StaticAgreement();
    agreement.registerValue("BASE_RATE");
    agreement.setValue("BASE_RATE",
            new GregorianCalendar(1999, 10, 1, 0, 0, 0).getTime(), 10.0);
    agreement.setValue("BASE_RATE",
            new GregorianCalendar(2008, 04, 16, 0, 0, 0).getTime(), 10.0);
    agreement.registerValue("REDUCED_RATE");
    agreement.setValue("REDUCED_RATE",
            new GregorianCalendar(1999, 10, 01, 0, 0, 0).getTime(), 5.0);
    agreement.registerValue("CAP");
    agreement.setValue("CAP",
            new GregorianCalendar(1999, 10, 01, 0, 0, 0).getTime(), 50.0);

    agreement.registerPostingRule("USAGE");
    agreement.addPostingRule("USAGE",
            new GregorianCalendar(1999, 10, 01, 0, 0, 0).getTime(),
            new IfPostingRule("base-usage",
                    evaluateIf().eventValue("usage").LargerThan().value("CAP"),
                    new CalculationPostingRule("base-usage",
                            compute().variable("BASE_RATE").multipliedBy().eventValue("usage")),
                    new CalculationPostingRule("base-usage",
                            compute().variable("REDUCED_RATE").multipliedBy().eventValue("usage"))));
    agreement.registerPostingRule("SERVICE_CALL");
    agreement.addPostingRule("SERVICE_CALL",
            new GregorianCalendar(1999, 10, 01, 0, 0, 0).getTime(),
            new StaticValuePostingRule("service", new BigDecimal(10.0)));
    agreement.addPostingRule("SERVICE_CALL",
            new GregorianCalendar(1999,12, 02, 0, 0, 0).getTime(),
            new CalculationPostingRule("base-usage",
                    compute().eventValue("fee").multipliedBy(new BigDecimal(0.5)).add(new BigDecimal(15.0))));
    agreement.registerPostingRule("TAX");
    agreement.addPostingRule("TAX",
            new GregorianCalendar(2000, 02, 29, 0, 0, 0).getTime(),
            new CalculationPostingRule("tax",
                    compute().eventValue("fee").multipliedBy(new BigDecimal(0.055))));

    return agreement;
}
```

Figuur 3: De agreement DSL in Java code

De eerste regel beschrijft de definitie van een plan, de start van onze DSL. We definiëren dat een plan moet starten met het keyword 'plan', gevolgd door de naam van het plan. Hierna mogen een aantal Rates en een aantal EventRules gedefinieerd worden. Het symbool '*' geeft hierbij aan dat in de DSL meerdere Rates mogen voorkomen per periode, zoals te zien is in Figuur 2.
De in Figuur 4 getoonde regels zijn vrij eenvoudig. Naast de mogelijkheid om verschillende regels te definiëren en hierna te verwijzen bevat de Xtext taal nog meer mogelijkheden, zoals:
- Optionele elementen met het '?' symbool;
- Abstracte regels. Op de plaats waar naar een abstracte regel verwezen wordt mogen de verschillende 'afgeleide' regels gebruikt worden. In ons plan wordt bijvoorbeeld het amount op verschillende manieren berekend: vaste waardes, formules of if-statements. Deze worden door aparte regels gedefinieerd die elk op de plaats van de abstracte regel gebruikt kunnen worden.

```
// definition of the basic structure:

Plan:
    "plan" name=ID                      // main container
    (rates += RateDecl)*                // temporal values, act as local constants
    (eventRules += EventRulesDecl)*;    // declaration of temporal rules

RateDecl:
    "rate" name=ID
    (values += ValueDecl)*;     // temporal property

ValueDecl:
    date=Date ":" amount=Amount;

EventRulesDecl:
    "event" type=EventType
    (postingRules += TemporalPostingRule)*; // an "event" block has a number of temporal posting rules
```

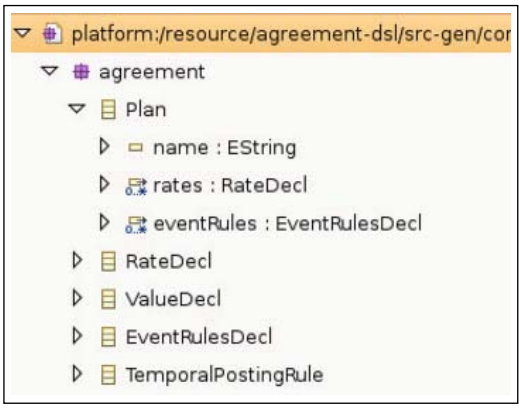
Figuur 4 : De Agreement DSL grammatica

- Enumeraties. Hiermee zijn de toegestane symbolen welke als waarde gebruikt worden te beperken. Deze waardes zijn dus geen keywords, maar een domein van toegestane waardes in de uiteindelijke DSL.

Vanwege plaatsgebrek zullen we niet de volledige DSL definitie hier behandelen. Ter referentie is deze wel te downloaden[5].

## Stap 3 : Generatie van de parser en de editor

Als de grammatica van de DSL gedefiniëerd is kan deze gebruikt worden om het model, de parser en de editor te genereren. De meegeleverde "generate.oaw" file kan uitgevoerd worden om dit te bereiken. In Figuur 2 werd de editor zelf al getoond met een voorbeeld DSL. Daarnaast toont Figuur 5 het EMF model, Xtext gebruikt het Eclipse Modeling Framework[7] voor de implementatie van het metamodel , dat ook gegenereerd wordt uit deze definitie.




Figuur 5 : Het EMF Model

In deze figuur zien we direct dat de naam van een productieregel in Xtext resulteert in een metatype en dat de definitie van de productieregel resulteert in nul of meerdere attributen van dit type. Hier gaan we in de volgende stap gebruik van maken.

## Stap 4 : Uitbreiden van de editor met semantische controles

De gegenereerde editor zal controleren of de DSL-eindgebruiker code schrijft die voldoet aan de syntax van onze Agreement DSL. Om daarnaast semantische regels uit te drukken wordt gebruik gemaakt van Check. Er wordt onderscheid gemaakt in fouten en waarschuwingen. Een voorbeeld van een fout is de definitie van een ongeldige datum of een 'plan' zonder 'event'. Hoewel de syntax van onze DSL dit toestaat, kunnen met Check deze controles ingebouwd worden om zo de gebruiker verder te ondersteunen. Gevonden fouten worden binnen Eclipse meteen getoond in het bekende foutvenster. De definitie en het resultaat hiervan worden getoond in Figuur 6 en Figuur 7. Hierbij zien we dat we gebruik maken van het metatype en de attributen die resulteren uit de definitie van een productieregel.

Figuur 6: Definities check

```
context Plan ERROR "plan " + this.name + " must have events" : this.eventRules.size > 0;

context RateDecl ERROR "rate " + this.name + " must have values" : this.values.size > 0;

context Date ERROR "invalid month in date: " + this.month : validMonthNr();

context Date if validMonthNr() ERROR "invalid day in date: " + this.day :
    ( 1 <= this.day ) && ( this.day <= monthNr2nDays(this.month, this.year) );

context Date ERROR "invalid year in date: " + this.year : this.year > 0;
```

## Stap 5 : Het gebruiken van de DSL

Nu we een DSL, een editor, een meta-model en een parser hebben, kunnen we onze DSL ook daadwerkelijk gaan gebruiken. Zoals gezegd kunnen we de transformatie van DSL naar code doen tijdens het ontwikkelen – generatie – of tijdens uitvoeren van het programma – configuratie. In dit artikel geven we een voorbeeld van een configuratie. De aanpak is in feite gelijk aan verschillende frameworks die middels tekstuele files te configureren zijn. Je kan hierbij denken aan Spring, Mule en jBPM. Deze frameworks bevatten allen een engine, welke geconfigureerd wordt. De configuratie bevat dus het gedrag van de applicatie, de engine geeft de mogelijkheid om dit gedrag uit te voeren.

In ons voorbeeld zullen we dan ook een engine moeten maken die het gedrag van onze DSL kan modelleren. Hoewel we de implementatie van deze engine – simpelweg een set van POJO's – niet zullen bespreken is deze wel te vinden bij de code horende bij het artikel dat is te downloaden vanaf de eerdergenoemde lokatie[5].

Nadat de engine gemaakt en getest is, kunnen we de code gaan schrijven om deze te configureren. In feite is dit de implementatie van een (set van) Builder-patterns. De parser implementatie wordt door Xtext gegenereerd uit de grammatica en kost ons dan ook geen enkele inspanning. Daarnaast hebben we gebruik gemaakt van de standaardmogelijkheden van EMF om het model dat gegenereerd wordt uit de grammatica om te zetten naar Java classes.

In de uiteindelijke implementatie kunnen we de engine configureren op basis van verschillende plans en hebben we de mogelijkheid ingebouwd om per klant een plan te kiezen. Tenslotte kan het gedrag van het systeem gesimuleerd worden, waardoor we verschillende events afgehandeld zien worden.

## Conclusie en discussie

Programmatuur geschreven in een goed ontworpen DSL is beter leesbaar dan de equivalente oplossing geschreven in een taal zonder speci-

fiek domein eventueel aangevuld met een voor het probleemdomein ontwikkelde bibliotheek en API. Deze verbeterde leesbaarheid heeft te maken met de grotere uitdrukkingskracht die de DSL heeft voor het beperkte toepassingsgebied. Deze verbeterde leesbaarheid kan leiden tot een toename van de kwaliteit van de software en productiviteit van de ontwikkelaar.

Daarnaast biedt de inzet van DSLs een uitgelezen mogelijkheid om 'separation of concerns' toe te passen. Technische en infrastructurele (non-functionele) wijzigingen aan de software, zoals het veranderen van de ORM bibliotheek of zelfs de volledige executie omgeving, is gemakkelijker te realiseren bij programmatuur geschreven in een DSL. Een dergelijke wijziging is een wijziging in de transformatie van de DSL naar de GPL. De transformatie zal worden aangepast om aan de nieuwe non-functionele eisen te voldoen, maar de functionele eisen (business logica) geschreven in de DSL hoeven in het ideale geval niet te worden veranderd.

Wie pakt de rol van DSL ontwerper?

DSL ontwerp is niet triviaal. Er is zowel kennis van het domein als kennis van taalontwerp nodig. Met de komst van language workbenches, zoals oAW, neemt het gewicht op het laatste iets af.

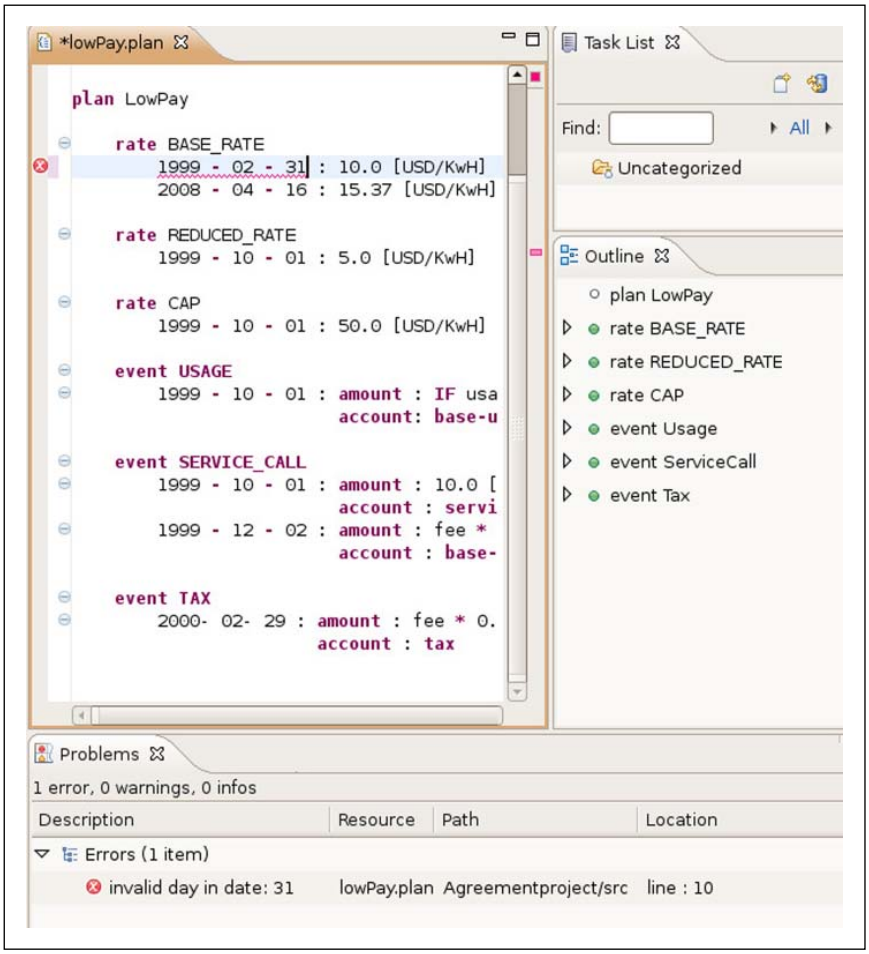
De architect is mogelijk de juiste persoon voor het ontwerpen van de DSL. Ook is de architect de aangewezen persoon voor de definitie van de transformaties van DSL naar GPL, omdat hier de architectuur beslissingen in gevat kunnen worden. Hiermee zijn technisch complexe en foutgevoelige zaken te verbergen in de transformaties en dus voor gebruikers van de DSL.

Xtext brengt een bruikbare verzameling van tools voor de ontwikkeling van DSLs welke direct integreren in het dagelijkse ontwikkelproces of ontwikkelstraat. Voor de geïnformeerde compilerbouwer is er weinig tot geen nieuws onder de zon, maar wat de language workbenches in het algemeen en oAW in het bijzonder voor Java wel toevoegen is het beschikbaar maken van deze mogelijkheden binnen een bekende ontwikkelomgeving waardoor de drempel tot daadwerkelijk gebruik wordt verlaagd.

Er zijn niet alleen maar voordelen te noemen. Een risico is dat het vooralsnog moeilijk te bepalen is of de investering, het ontwerpen en implementeren van de DSL en generatoren, voldoende rendement geeft. Dit zal afhangen van een verscheidenheid aan factoren, waarbij valt te denken aan de omvang van het project of projecten waarbij eenzelfde DSL van dienst kan zijn (mogelijkheid tot hergebruik) en de samenstelling van het team. Een volledige beschrijving valt buiten de scope van dit artikel, zie hiervoor bijvoorbeeld [6], maar het lijkt moeilijk vast te stellen.

De mogelijke wildgroei aan talen die kan ontstaan wordt ook vaak als een risico genoemd. Dit is niet geheel te ontkennen, maar een goed ontworpen DSL is simpel en intuïtief in gebruik. Daarnaast nemen wij aan dat een complexe DSL een zeker zo complexe GPL tegenoplossing heeft.

DSLs in combinatie met language workbenches zijn ons inziens een van de meest ambitieuze en potentiële ontwikkelingen van de laatste tijd. Misschien horen we er over een jaar niets meer over, misschien wordt het een ware revolutie in de software ontwikkeling. We gaan het zien! «




Figuur 7: Eclipse foutenvenster met onjuiste datum

**Referenties:**


1. openArchitectureWare: http://www.eclipse.org/gmt/oaw/
2. Fowlers ongepubliceerde boek: http://martinfowler.com/dslwip/
3. Fowlers Agreement Dispatcher pattern: http://www.martinfowler.com/eaaDev/AgreementDispatcher.html
4. Gang Of Four – Builder pattern
5. Agreement DSL implementation: http://code.google.com/p/agreementdsl/.
6. When and How to Develop Domain-Specific Languages: http://ftp.cwi.nl/CWIreports/SEN/SEN-E0517.pdf
7. Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/


Met dank aan Meinte Boersma die een bijdrage heeft geleverd aan de grammatica van de Agreement DSL in Xtext.