

Vrije Universiteit Brussel
Programming Technology Lab
Faculteit Wetenschappen - Departement Informatica
Academiejaar 2001 - 2002



Automatische Codegeneratie voor
Samenwerkende Componenten aan de
hand van een Declaratieve Beschrijving

Wim R. Crols

*Proefschrift ingediend met het oog
op het behalen van de graad van
Licentiaat in de Informatica*

Promotor: Prof. Dr. Theo D'Hondt
Begeleider: Tom Tourwé

Inhoudsopgave

1	Inleiding	2
2	Probleemstelling	4
2.1	Het VRT MPEG-project	4
2.2	Het experiment	5
2.3	Problemen	5
2.4	Conclusie	7
3	XML	8
3.1	XML	8
3.1.1	Elementen	9
3.1.2	Attributen	9
3.1.3	Entiteiten	11
3.1.4	Verwerkingsinstructies	11
3.2	DTD	11
3.2.1	Elementdeclaratie	12
3.2.2	Attribuutdeclaratie	12
3.3	XML Schema	12
3.3.1	Elementen	13
3.3.2	Types	13
3.4	XSL	13
3.4.1	XSLT	14
3.5	SAX	16
3.5.1	DOM vs. SAX	16
3.5.2	SAX-methoden	18
3.6	Conclusie	20
4	SOUL	21
4.1	Afkomst	21
4.2	Syntax	22
4.2.1	Feiten	22
4.2.2	Regels	22
4.2.3	Query	23

4.2.4	Smalltalk-termen	24
4.2.5	Smalltalk-clauses	24
4.2.6	Quoted code-termen	24
4.3	SOUL vs. XSLT	25
4.4	Conclusie	29
5	Architectuur van het systeem	30
5.1	Voorbeeld: iTunes	30
5.1.1	XMMS-XML	31
5.1.2	iTunes API	31
5.2	Modellering	32
5.2.1	Pluginnaam	32
5.2.2	Actionmappings	33
5.3	Generatie en gebruik	34
5.4	Architectuur	35
5.4.1	De Declaratieve laag	35
5.4.2	De Implementatielaag	37
5.4.3	De API-laag	37
5.4.4	Taalonafhankelijkheid	37
5.5	Conclusie	38
6	Implementatie van het experiment	39
6.1	Generatie van Smalltalk-klassen	39
6.1.1	Interfacing naar Smalltalk	40
6.2	XMLAction-klassen	42
6.2.1	Werking in Smalltalk	42
6.2.2	Generatie in SOUL	44
6.3	Plugin	47
6.3.1	Werking in Smalltalk	47
6.3.2	Generatie in SOUL	50
6.4	Conclusie	54
7	Conclusie	55
7.1	Samenvatting	55
7.2	Bijdrage	56
7.3	Verder werk	56
7.3.1	Uitvoer	57
7.3.2	Flexibelere XML	57
7.3.3	Flexibeler API	58
7.3.4	Hybride specificatie	61
7.3.5	Taalonafhankelijkheid	61

Lijst van figuren

2.1	Omschakeling naar gemeenschappelijk XML-formaat	5
2.2	Generatie van plugin a.d.h.v. beschrijving API	6
3.1	Voorbeeld van een elementsstructuur	9
3.2	Voorbeeld van een XML-document	10
3.3	Voorbeeld van een DTD	12
3.4	Het XSLT transformatieproces	14
3.5	De SAX-events	18
5.1	Voorbeeld van XMMS-XML	32
5.2	De architectuur van de 3 talen	35
5.3	Mapping van XML op API door XMLActions	36
6.1	SOUL-code voor generatie XMLAction	44
6.2	SOUL-code voor generatie CompositeXMLAction	45

Abstract

Deze thesis behandelt de generatie van vertalings-plugins die de communicatie tussen verschillende componenten mogelijk maakt. Deze automatische generatie wordt gespecificeerd door modellen in een declaratieve beschrijving.

Dankwoord

Ik wil mijn begeleider Tom Tourwé bedanken om in de eerste plaats een onderwerp te bedenken nadat mijn eerste poging tot een stage mislukte. Ook heeft hij ontzettend veel geholpen bij de implementatie van mijn experiment, en de kwaliteit (en het volume, zonder twijfel) van deze thesis fors verhoogt door ze na te lezen en commentaar te geven.

Ik wil ook Theo d'Hondt bedanken om deze thesis te promoten, de mensen op het PROG-lab voor hun advies tijdens de tussentijdse presentaties, en iedereen die me dit jaar gesteund heeft.

Hoofdstuk 1

Inleiding

Stel dat je verschillende applicaties hebt, elk met een specifieke taak. Nu wil je die componenten met elkaar laten communiceren. Die communicatie gebeurt door hun in- en uitvoer om te zetten in XML en aan elkaar door te geven. Echter, deze XML is niet altijd noodzakelijk in hetzelfde formaat. Als de componenten elkaar moeten kunnen verstaan, is er iets nodig dat deze XML die hen wordt gegeven, vertaalt naar een voor hen begrijpbaar formaat, een formaat dat gelezen kan worden door de API van de component. In eerste instantie kunnen we dit probleem oplossen door code te schrijven die de communicatie vertaalt naar deze API en die gebruikt kan worden door de component als een plugin. Echter, als je nieuwe applicaties toe zou voegen, zou je al dat werk opnieuw moeten doen, voor elke component en elk XML-formaat, terwijl het grotendeels hetzelfde is. Daarom onderzoeken we in deze thesis de automatische generatie van zulke code, voor eender welke gebruikte component en taal.

Hoe kunnen we zulke automatische generatie mogelijk maken? De data en de manier waarop die verwerkt moet worden moet door de vertalings-plugin uit de XML gehaald worden en op de correcte manier doorgegeven worden aan de API van de component waarvoor de plugin werkt. Dus de plugin moet de XML kunnen lezen en begrijpen, en ook weten hoe de functies van API werken. We moeten een correspondentie vinden tussen de XML en de API-functies, zodat de juiste data aan de juiste functies wordt doorgegeven. We moeten dus een soort model opstellen, dat deze dingen beschrijft, dat specifiek is voor een koppel van XML en een components-API. Daarvan kan de plugin gegenereerd worden.

We kijken hoe we deze zaken kunnen modelleren. We willen een model schrijven zonder dat we er extra parse-programma's voor hoeven te bouwen. Dus bijvoorbeeld geen tekstformaat dat we moeten uitvinden, maar een formaat dat als het genoteerd is direct betekenis heeft voor de generatie. In deze thesis stellen we daarom hiervoor een meta-declaratieve taal voor. Dit is een declaratieve omgeving die over een tweede, lagere taal redeneert,

en toelaat rechtstreeks de source code van deze onderliggende taal te manipuleren. In deze taal kunnen bijvoorbeeld patronen worden opgeschreven die de structuur beschrijven van de plugins die we moeten genereren. Op die manier kunnen we op een leesbare, ondubbelzinnige manier modellen opschrijven, die tezelfdertijd al een deel van de generatie specificeren. Wij gebruiken hiervoor SOUL, een logische taal die is ontwikkeld om over de object-georiënteerde taal Smalltalk te redeneren.

De structuur van de thesis is als volgt: in het eerstvolgende hoofdstuk leggen we het kader uit van deze thesis en de problemen die we moeten zien op te lossen. Daarna komen twee hoofdstukken aan bod die de gebruikte materie uitleggen. Hoofdstuk 3 bekijkt XML, waarin de communicatie tussen de componenten gebeurt. Hoofdstuk 4 legt de declaratieve taal SOUL uit, die we gebruiken om de specificaties van generatie in te modelleren. Daarna zijn we toe aan het eigenlijke thesisexperiment in hoofdstuk 5. Aan de hand van de case die we gebruiken wordt een handleiding van het systeem gegeven. We bekijken ook de achterliggende architectuur en taalonafhankelijkheid. Hoofdstuk 6 dat daar op volgt behandelt de volledige implementatie van ons systeem. We sluiten af met de algemene conclusie die alles recapituleert, en ook kijkt naar mogelijke verbeteringen.

Hoofdstuk 2

Probleemstelling

Hier leggen we het kader uit waarin deze thesis is uitgevoerd: het MPEG-project van de VRT, waaraan ook *PROG (Programming Technology Lab)* deelneemt, en de problemen die we voor de thesis moeten trachten op te lossen.

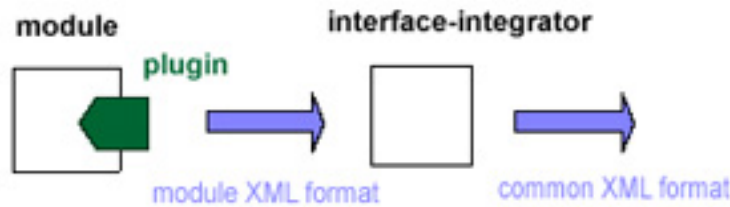
2.1 Het VRT MPEG-project

Het *MPEG-onderzoeksproject* [18] maakt deel uit van de uitbouw van een *Content Management System* op de VRT, dat al het nieuwe en bestaande inhoudelijke materiaal (beeld, geluid, tekst, grafiek, interactieve scenario's...) moet beheren.

De uitbouw van dit beheersysteem impliceert ook de digitalisering van het bestaande archiefmateriaal. Hiervoor wordt gebruik gemaakt van bestaande en nieuwe MPEG-normen, die beeld en geluid kunnen omvormen voor verdeling via kanalen met beperkte capaciteit zoals het Internet. Ook laten ze de omroep toe via genormaliseerde beschrijvingen van het programmamateriaal zijn aanbod af stellen op de individuele wensen van de kijker, eveneens geformuleerd in deze MPEG-beschrijvingen.

Het onderzoeksprogramma zal nagaan hoe deze digitalisering op een uitwisselbare en duurzame manier mogelijk is, zodat dit de ontsluiting faciliteert over bestaande en toekomstige kanalen (kabel, telefoonlijn, ether, ...) en platformen (televisie, digitaal thuisplatform, pc, spelconsoles, gsm, ...).

De hoofddoelstelling van het project is een competentiecentrum rond MPEG-technieken te scheppen, uitgaande van de verwachting dat zij de dominante norm aangaande mediadata wordt. Uit dit centrum en de verschillende daarin deelnemende partijen zullen nieuwe projecten groeien, die op hun beurt bijdragen tot kennisopbouw in de VRT.



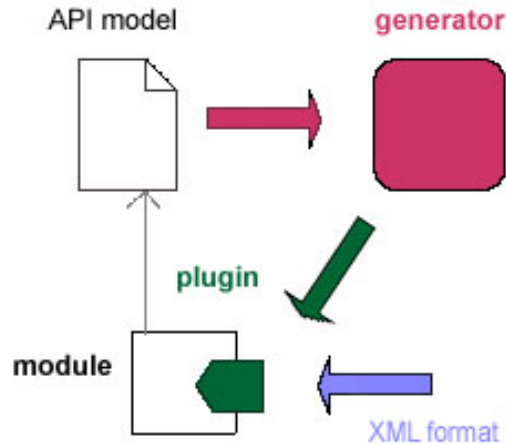
Figuur 2.1: Omschakeling naar gemeenschappelijk XML-formaat

2.2 Het experiment

De VRT gebruikt in zijn bestaande informatica-infrastructuur in verscheidene diensten allerhande modules. Modules zijn applicatiecomponenten, elk met een specifieke taak. In het project wordt onderzoek verricht naar modelleertechnieken voor componenten die het mogelijk maken om al die onderdelen uniform te behandelen en te laten samenwerken, vooral met het oog op metadata. De in- en uitvoer van (meta)data bij deze componenten gebeurt meestal in *XML* (zie hoofdstuk 3). Echter, die XML-formaten zijn niet altijd gelijk, of bevatten soms fouten, zeker bij modules die niet ontwikkeld zijn door de VRT zelf, maar bijvoorbeeld aangekocht zijn. Om dit op te lossen, bestaat langs de kant van de component de API voor de vertaling van componentdata naar XML, geïmplementeerd door een *plugin*. Langs de andere kant beschikt de VRT over een *interface-integrator* die bestaande XML omzet naar syntactisch andere XML (over XML transformaties zie ook sectie 3.4.1) om zo een gemeenschappelijk formaat te bereiken voor alle modules (figuur 2.1). Het schrijven van deze plugins voor elke module die gaat gebruikt worden is repetitief werk. Het idee is daarom om in een volgende fase te proberen om deze plugins automatisch te genereren.

2.3 Problemen

In deze thesis onderzoeken we hoe we de automatische generatie van de invoerplugin mogelijk kunnen maken. Het experiment dat aan de basis ligt van deze thesis bekijkt het geval van een *invoer*-plugin. Bij invoer moet XML-data die binnenkomt uiteindelijk gevoerd worden aan de module via zijn API. De gegenereerde plugin die dit uiteindelijk moet doen, moet dus de XML kunnen lezen en voldoende begrijpen om de instructies erin te herkennen. Hij moet ook de functies van de API kennen, zodat hij de juiste data aan de juiste functie kan voeren. Deze dingen zijn specifiek voor een gegeven configuratie van een XML-formaat en een component-API. Bij generatie van een plugin is dit hetgene dat verschilt van configuratie tot



Figuur 2.2: Generatie van plugin a.d.h.v. beschrijving API

configuratie, de rest is gelijk voor de verschillende plugins, dus we moeten dit zien te beschrijven voor elke generatie.

We gaan dus op een bepaalde manier zowel de XML als de API moeten modelleren. Dit model kan op verscheidene manieren opgeschreven worden, in een tekstbeschrijving, in XML, in een programmeertaal, . . . Wat wij echter willen is hiervan code genereren. We willen vermijden om bijvoorbeeld eerst een programma hoeven te schrijven en een model te verzinnen dat dan moet ingelezen en geparsed worden om ermee verder te werken. We willen een model dat een betekenis heeft waarvan we de code die de plugin implementeert kunnen genereren. Daarom gebruiken we een declaratieve meta-taal. Dit is een declaratieve taal wiens regels kunnen redeneren over een tweede programmeertaal. In deze declaratieve taal kunnen patronen worden opgeschreven die de structuur specificeren voor code in de onderliggende taal. Op deze manier wordt het mogelijk om een model te maken, dat onmiddellijk als deel van een programma dient om de plugin automatisch te laten genereren, zoals voorgesteld in figuur 2.2.

Wij gaan gebruik maken van de hogere orde declaratieve taal *SOUL* bij onze modellering, beschreven in hoofdstuk 4. Het framework van *SOUL* is zo gebouwd dat het over Smalltalk kan redeneren en de source code ervan kan manipuleren.

Een ander probleem is dat de gegeven de XML, de juiste API-calls gegenereerd moeten worden. We moeten dus niet enkel de API en XML modelleren, we moeten deze twee modellen op elkaar mappen. Er moet een manier gevonden worden om aan elk XML-fragment dat een instructie beschrijft de correcte API-functie te koppelen, waarbij ook gezorgd moet worden dat alle

argumenten die hierbij horen, vertaalt worden van de XML naar de API.

In de volgende hoofdstukken zullen de oplossingen die wij voor deze problemen vonden uitgelegd worden.

2.4 Conclusie

De VRT gebruikt verschillende applicatiecomponenten met van elkaar afwijkende in- en uitvoer in XML. Plugins, die de data omzetten voor de API van de component, worden hiervoor geschreven. In de toekomst willen we deze automatisch laten genereren. Het doel van deze thesis is experimenteren hoe we die generatie mogelijk kunnen maken, wat we ervoor moeten modelleren, en hoe dit modelleren moet gebeuren. Als case passen we deze generatie toe op een XML-invoerplugin.

Hoofdstuk 3

XML

XML, het communicatiemedium gehanteerd in mijn thesis, is een metataal gebruikt om markuptalen voor datastructuratie te creëren. Maar XML behelst ook een uitgebreide familie standaarden aanbevolen door het *World Wide Web Consortium (W3C)* [3]. In dit hoofdstuk zal ik, na XML zelf, DTDs die de talen declareren aankaarten, alsook zijn jongere vervanger XML Schema, verder de transformaties van XSL en als laatste de SAX-interface, die we gebruiken als basis om XML te lezen.

Verdere uitleg over al deze standaarden kan je altijd vinden in [1, 2, 3, 7, 11, 20].

3.1 XML

De *Extensible Markup Language* is een manier om gestructureerd data op te schrijven voor uitwisseling tussen mensen en applicaties. Gestructureerde data bevat inhoud (tekst, verwijzingen naar tekeningen, ...) en informatie over welke rol die data speelt (bv. een bepaald stuk tekst kan de titel zijn). In XML wordt die data genoteerd met *elementen* en *attributen*, gelijkaardig aan HTML. Ook XML-toepassingen zijn gewoon tekstformaat. Dit bevordert leesbaarheid door mensen, mocht het nodig zijn.

In tegenstelling tot HTML kan je je eigen elementen definiëren zodat je volledig vrij bent in de opbouw van je documenten. Dit betekent impliciet dat XML geen semantiek vastlegt. Bijvoorbeeld in één XML-taal of -toepassing kan het element '<p>' "prijs" betekenen en in de andere "paragraaf".

Een voorbeeld van een XML-document is te zien op het einde van deze sectie in figuur 3.2. Deze XML beschrijft de instructie om de data over een MP3 aan te passen.

```

<vaderelement>
  <kindelement>Tekstinhoud</kindelement>
  <kindelement></kindelement>
  Inhoud van vaderelement
</vaderelement>

```

Figuur 3.1: Voorbeeld van een elementsstructuur

3.1.1 Elementen

Elementen zijn de bouwstenen van XML. Ze worden geopend met een tag als '`<elementnaam>`' en gesloten met '`</elementnaam>`'. Ze kunnen op zich weer elementen of tekst of beide als inhoud hebben, zodat het geheel een soort boomstructuur wordt. In tegenstelling tot de lakse syntax van HTML, moeten om *welgevormde* XML te zijn, elementen altijd een sluitingstag hebben en correct genest zijn, dat wil zeggen, geen element mag sluiten voor al zijn kinderen gesloten zijn. Een voorbeeld van een goede structuur zie je in figuur 3.1. Lege elementen, zonder tekst of kinderelementen, kunnen ook weergegeven worden als '`<\elementnaam>`'.

3.1.2 Attributen

Attributen worden genoteerd in de openingstag van het element als '`<elementnaam attribuutnaam="attribuutwaarde">`' (of met enkele aanhalingstekens '`<elementnaam attribuutnaam='attribuutwaarde'>`').

Attributen zijn naam/waarde-koppels die bij een element horen. Hun data gaat eerder over de aard van het element, of om het anders te zeggen: attributen zijn meestal belangrijker voor de parser dan voor de lezer. De beslissing of bepaalde informatie als inhoud of als attribuut wordt genoteerd, ligt echter volledig in de handen van de ontwikkelaar van de XML-toepassing.

```

<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<?xml-stylesheet type="text/xml" href="XMMS.xsl"?>

<!-- Test for changing track-->
<XMMS>
  <action type="editMP3">
    <Sender>http://localhost:8080</Sender>
    <PlaylistName type="currentPlaylist">Super Playlist 5
    </PlaylistName>
    <MP3 type="currentMP3">
      <Key>12345</Key>
      <Title>The Riever</Title>
      <Artist>Bruus Springsteen</Artist>
      <Album>The best off</Album>
      <Year></Year>
      <Genre></Genre>
      <Comment>a &lt;modification&gt;</Comment>
    </MP3>
    <MP3 type="newMP3">
      <Key>54321</Key>
      <Title>The River</Title>
      <Artist>Bruce Springsteen</Artist>
      <Album>The best of</Album>
      </Year>
      </Genre>
      <Comment>This is the original version.</Comment>
    </MP3>
  </action>
</XMMS>

```

Figuur 3.2: Voorbeeld van een XML-document

3.1.3 Entiteiten

Daarnaast zijn er *entiteiten*. Entiteiten worden gebruikt als je tekens wil in XML-tekst die normaal als opmaak dienen, zoals '<'. In plaats daarvan zouden we schrijven '<'. Dit noemt men een ingebouwde entiteit, omdat je ze zelf niet eerst moet definiëren. Ze maken sowieso deel uit van XML, samen met de andere vier ingebouwde entiteiten: `>` (>), `"` ("), `&apost;` (') en `&` (&).

Algemener gebruik je, naast deze ingebouwde symbolen, entiteiten als afkortingen van stukken tekst. Deze entiteiten dienen eerst gedefinieerd te worden.

Daarnaast zijn er ook *karakterentiteiten*. Deze laten je toe eender welk *Unicode* symbool te gebruiken, zelfs degenen die niet op je toetsenbord staan, door ze decimaal te noteren zoals `℞` of hexadecimaal als `∞`.

3.1.4 Verwerkingsinstructies

XML-documenten kunnen aparte instructies bevatten voor de programma's die ermee werken. Deze heten *verwerkingsinstructies* (processing instructions) en worden genoteerd als '`<?doel naam="waarde" ?>`'.

Doelen die beginnen met 'xml', in elke mogelijke combinatie van hoofd- en kleine letters, zijn gereserveerd door het W3C. Een voorbeeld ervan is de XML-declaratie die als eerste lijn in elk document moet staan:

```
<?xml version=1.0" ?>.
```

3.2 DTD

De *Document Type Definition* bepaalt de geldigheid van een XML-toepassing. Hierin worden alle elementen en hun attributen gedefinieerd, hun inhoud, hun volgorde, of met andere woorden: de XML-taal. Bij de meeste toepassingen die XML lezen is de DTD niet noodzakelijk.

Een DTD gaat de eigenlijke XML-data vooraf in de documenten en is van de vorm '`<!DOCTYPE rootelement externid [declaraties]>`'. Het `rootelement` is de naam van het unieke element dat de voorouder is van alle elementen in de XML. De `externid`, enkel aanwezig indien de DTD niet in het document zelf is neergeschreven, is een string verwijzend naar het DTD-bestand. Bij persoonlijke bestanden is dat '`SYSTEM bestand.dtd`', waar `bestand.dtd` de *Universal Resource Identifier (URI)* is (meestal een pad of een URL) die de locatie van het bestand aangeeft. Bij een gestandaardiseerd, openbaar beschikbaar DTD is dat '`PUBLIC naam bestand.dtd`', waarbij de `naam` de officiële naam is en `bestand.dtd` opnieuw een URI. De hoofdzakelijk gebruikte `declaraties` worden hieronder summier beschreven. Tenslotte geeft figuur 3.3 een voorbeeld van een DTD. Voor de volledige specificatie zie de officiële W3C aanbevelingen [3].

```

<!ELEMENT playlist (song*)>
<!ELEMENT song (title, artist+, duration, file)>
  <!ATTLIST song action NMTOKEN "i">
<!ELEMENT title (#PCDATA)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT duration (#PCDATA)>
  <!ATTLIST duration unit NMTOKEN "s">
<!ELEMENT file (drive, path, name, size)>
<!ELEMENT drive (#PCDATA)>
<!ELEMENT path (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
  <!ATTLIST size unit NMTOKEN "B">

```

Figuur 3.3: Voorbeeld van een DTD

3.2.1 Elementdeclaratie

Een element “naam” wordt gedeclareerd ‘<!ELEMENT naam inhoudspec>’. Hierbij geeft *inhoudspec* weer wat er allemaal in het element mag. Dit kan zijn ‘ANY’ voor geen inhoudsrestricties, ‘EMPTY’ voor lege elementen, of een opeenvolging tussen haakjes van kinderelementnamen en/of ‘#PCDATA’ voor tekst (parsed character data) zoals in

```
<!ELEMENT vaderelement (kinderelement*, #PCDATA)>
```

Zoals hierboven kan willekeurige herhaling worden weergegeven met ‘*’ achteraan aan de elementnaam. Daarnaast is minstens eenmaal herhalen ‘+’ en optioneel voorkomen ‘?’. Keuze tussen reeksen elementen kan aangeduid worden met ‘|’.

In figuur 3.3 zie je de declaraties van elementen (en hun bijhorende attributen) voor een XML-taal over MP3s te modelleren.

3.2.2 Attribuutdeclaratie

Om een attribuut “naam” bij element “element” te definiëren noteer je ‘<!ATTLIST element naam type defaultspec>’. Hier duidt *type* het type inhoud aan (meestal ‘#CDATA’) en *default* of het attribuut verplicht is of niet en wat zijn standaardwaarde is met ‘#IMPLIED’, ‘#REQUIRED’, ‘#FIXED "standaardwaarde"’ of enkel “standaardwaarde”.

3.3 XML Schema

DTDs hebben een aantal nadelen. Hun syntax is zelf geen XML zodat ze niet geanalyseerd kunnen worden door een XML-parser. Alle declaraties in

een DTD zijn globaal, zodat geen twee verschillende elementen met dezelfde naam gedefinieerd kunnen worden, zelfs al komen ze voor in verschillende contexten. En tenslotte, DTDs bieden geen mogelijkheid om te bepalen wat voor soort informatie een element of attribuut kan bevatten. Daarom heeft het W3C de taal *XML Schema* ontwikkeld, die de functie van DTDs overneemt, maar meer controle geeft over je XML-toepassing.

3.3.1 Elementen

In XML Schema neemt een elementdeclaratie de vorm aan van een leeg XML-element met alle data in de attributen:

'<xsd:element name="naam" type="type"/>'. Type is hier één van XML Schema's ingebouwde types zoals `xsd:string`, `xsd:integer` of `xsd:date` om er enkelen te noemen.

3.3.2 Types

De echte kracht van types ligt in het feit dat je er ook zelf kan definiëren. In XML Schema heb je elementen van eenvoudige types, die enkel tekst bevatten, en complexe types, met eventuele attributen, kinderelementen en/of tekst.

Om een nieuw eenvoudig type af te leiden vertrek je van een bestaand type en ga je daarop een aantal restricties doorvoeren. Dit wordt genoteerd

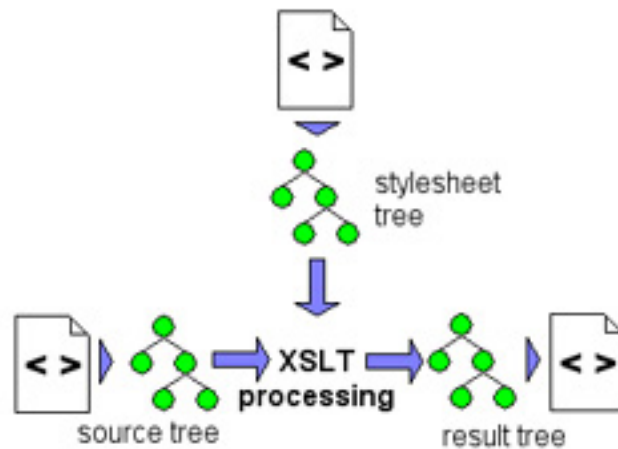
```
<xsd:simpleType name="naam">
  <xsd:restriction base="basis">
    restricties
  </xsd:restriction>
</xsd:simpleType>
```

Er zijn een hoop restricties beschikbaar zoals bereiken vastleggen, enumeraties, reguliere expressies in patronen, en anderen.

Bij complexe elementen, zijn er extra specificaties die de volgorde van kinderelementen bepalen, de keuze tussen elementen, toegelaten aantal, herhaling, enzoverder. Omdat dit buiten het bereik van de thesis gaat, gaan we er hier echter niet verder op in.

3.4 XSL

Een ander deel van de XML-specificaties is de *Extensible Stylesheet Language (XSL)*. XSL zorgt voor het transformeren en opmaken van XML, wat nodig is omdat XML niet vastlegt hoe zijn elementen getoond moeten worden. Je kan zelf je elementen creëren dus kan er geen vaste presentatie aan een element verbonden zijn. Wat je wel kan doen is specificeren hoe je elk element wil opmaken of je XML omzetten naar een formaat dat wel een vaste opmaak



Figuur 3.4: Het XSLT transformatieproces

heeft, zoals bijvoorbeeld HTML. Voor het eerste zou je de ene helft van XSL gebruiken: *XSL-FO (Formatting Objects)*. Wij werken enkel in het hanteren van data dus dat bespreken we hier niet. We gebruiken wel het deel om te transformeren: de *Extensible Stylesheet Language Transformations (XSLT)*.

3.4.1 XSLT

De transformatieregels, zelf ook een XML-taal, worden neergeschreven in een apart *stijlblad*. Dit wordt samen met het te transformeren XML-document doorgegeven aan een XSLT-processor, die afhankelijk van de instructies in het stijlblad een nieuw XML- of HTML-fragment genereert, zoals te zien in figuur 3.4. Hoewel XSLT gebruikt kan worden om willekeurige tekst uit te voeren, is het voornamelijk bedoeld voor XML naar XML transformaties.

XSLT bekijkt elk document als een boom, waarin elk element een knooppunt is. In het XSLT-stijlblad worden *templates* gedefinieerd:

```
<xsl:template match="knooppunten">
  uitvoer
</xsl:template>
```

die op de overeenstemmende knooppunten van de boom worden toegepast. De attribuutwaarde *knooppunten* is een patroon uitgedrukt in de taal *XPath* [3]. XPath laat je toe om delen van de XML-boomstructuur te refereren, je kan er bijvoorbeeld mee vragen ‘Geef alle knooppunten “*artist*” die het kind zijn van “*song*”’. Het template vervangt alle geldige knooppunten door de tekst *uitvoer*. Dit is XML-tekst maar waar ook XSLT-instructies in kunnen zitten. De belangrijkste instructies daarbij zijn:

- `<xsl:apply-templates/>`, wat verder alle mogelijke templates gaat toepassen op de onderliggende knooppunten in de boom (dat kunnen er meerdere zijn per knooppunt). Een restrictie kan opgelegd worden door het attribuut ‘`select="expressie"`’ mee te geven met een XPath-expressie. Deze kiest de knooppunten die verwerkt moeten worden uit.

Deze `<xsl:apply-templates/>`-regels bepalen dus de volgorde en de manier waarop de knooppunten verwerkt worden; de volgorde in het stijlblad heeft geen belang.

- `<xsl:value-of select="expressie"/>` die waarden uit de elementen haalt (zoals de waarde van een attribuut of van tekst in het element) en in de uitvoer plaatst, waarbij ook hier `expressie` een expressie in XPath is.

Laten we even hiervan een klein voorbeeld bekijken. We nemen het XML-fragment

```
<song action="i">
  <title>Time Is Now</title>
  <artist>Moloko</artist>
  <file>
    <path>Mijn documenten/Mijn muziek/Regula style</path>
    <size unit="KB">7.450</size>
  </file>
</song>
```

en we gaan dit omzetten naar een HTML-representatie met XSLT. Hiervoor maken we een XSLT-stijlblad met de volgende twee templates:

```
<xsl:template match="/">
  <html><body>
    <xsl:apply-templates/>
  </body></html>
</xsl:template>
<xsl:template match="song">
  Title: <xsl:value-of select="title"/><br>
  Artist: <xsl:value-of select="artist"/><br>
</xsl:template>
```

Dan zou de XML door dit stijlblad omgezet worden naar de volgende HTML:

```
<html><body>
  Title: Time Is Now<br>
  Artist: Moloko<br>
</body></html>
```

Het root-template (met 'match="/"') wordt eerst opgeroepen, dit schrijft de <html>- en <body>-tags. Daarna wordt de XSLT-processor gevraagd templates te gaan toepassen op de rest van de boom, waarvan het eerste element <song> is. Het tweede template kan hierop toegepast worden, omdat zijn `match`-attribuut hiermee overeenstemt. In dat template wordt dan met <xsl:value-of/> de inhoud van de kinderelementen <title> en <artist> van het <song>-element in de uitvoer gezet.

3.5 SAX

Om applicaties XML in te laten lezen en te laten parsen bestaan er verschillende aanbevelingen en tools. Wij gebruiken voor onze doeleinden SAX. In de eerste sectie leggen we uit waarom we SAX verkiezen boven andere technieken. Daarna hebben we het over zijn werking.

3.5.1 DOM vs. SAX

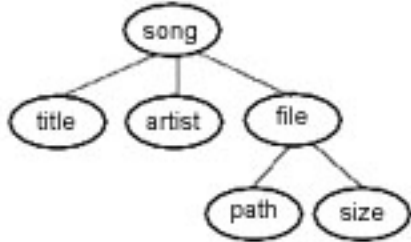
In onze invoerplugin moeten we de inkomende XML kunnen parsen om hem te kunnen ontleden en bewerken. De twee belangrijkste programmeerinterfaces die hiervoor bestaan zijn *Document Object Protocol* [3], aanbevolen door het W3C, en *Simple API for XML* [10], oorspronkelijk ontwikkeld voor Java.

DOM

DOM representeert een XML-document als een boom van knopen. De top van de boom is het XML-rootelement en zijn kinderelementen de vertakkingen. Bijvoorbeeld het XML-fragment

```
<song action="i">
  <title>Time Is Now</title>
  <artist>Moloko</artist>
  <file>
    <path>Mijn documenten/Mijn muziek/Regula style</path>
    <size unit="KB">7.450</size>
  </file>
</song>
```

wordt vertegenwoordigd door de volgende boom:



Verschillende XML-entiteiten (elementen, attributen, tekst, commentaar, ...) worden voorgesteld door verschillende types knopen. Voor manipulatie wordt een vaste interface verstrekt, die dikwijls wordt geïmplementeerd met objecten: het document of de knoop wordt als een object bekeken waarnaar boodschappen gestuurd worden. Om een idee te geven, de root wordt verkregen door de boodschap `documentElement` te sturen naar de variabele waarin het document is ingelezen. Vandaar vertrekkend gebruik je bijvoorbeeld `childNodes` om zijn kinderelementen te verkrijgen. Zo is er een keyword voor elk mogelijk onderdeel van een knoop te lezen. Er zijn natuurlijk ook opdrachten om knopen te schrijven.

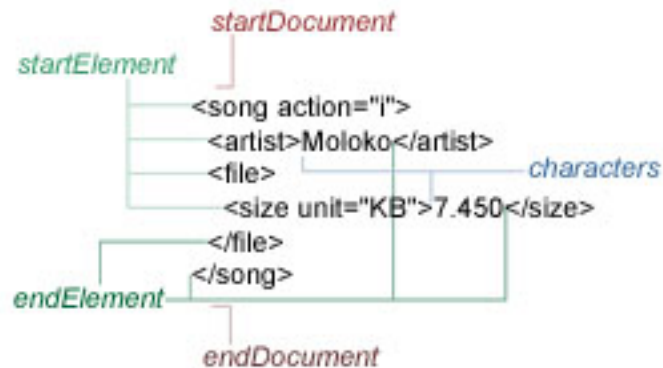
Wat zijn de voor- en nadelen van DOM?

- DOM is intuïtief in concept
- DOM is een uitgebreide en wijd geaccepteerde standaard, wat ervoor zorgt dat er veel ondersteuning voor te vinden is.
- Een document wordt bij DOM altijd sowieso geparsed en omgezet naar een boomstructuur, iets wat niet altijd nodig is, als er bijvoorbeeld maar een klein stuk van het XML-document behandeld moet worden, maar wat wel geheugen inneemt.
- Om een document volledig te doorlopen van begin tot einde, moeten bij gebruik van DOM nog bijkomende verwerkingsmethoden (zoals een Visitor design pattern [6]) geprogrammeerd worden, aangezien DOM zelf enkel een structuur vastlegt, maar geen algoritmen.

DOM leent zich meer tot traditionele verwerkings- en manipulatietoepassingen. Nog een praktisch nadeel bij deze thesis is dat er voor Squeak Smalltalk geen goede implementaties waren voor DOM die de specificatie correct volgden.

SAX

Daarentegen SAX is event-gebaseerd. Dat wil zeggen dat de parser die SAX ondersteunt het XML-document gaat doorlezen; telkens hij iets belangrijk tegenkomt, zoals een document of element dat opent, gaat hij een methode aanroepen. Zie figuur 3.5 ter verduidelijking van de SAX-events.



Figuur 3.5: De SAX-events

Een SAX-implementatie geeft je een klasse die alles bestuurt, de zogeheten *SAXHandler*, die voor elke mogelijke event een abstracte callback-methode voorziet. De bedoeling is dat jij voor jouw applicatie een nieuwe klasse definieert die overerft van deze event handler-klasse, en alle event-methodes die je wil gebruiken overschrijft. In deze methodes laat je dan de verwerking gebeuren van de XML-gegevens die je erin binnenkrijgt bij hun desbetreffende event.

- SAX is geen standaard van het W3C, het is oorspronkelijk ontwikkeld voor Java door David Megginson, en dus minder bekend en ondersteund.
- SAX geeft je meer vrijheid dan DOM.
- SAX vraagt wel meer werk als je je eigen structuur wil maken om je ingelezen XML in te modelleren en bewerken. DOM geeft je hiervoor direct zijn boomstructuur. Dit is natuurlijk ook een nadeel, aangezien je bij DOM geen structuur kan kiezen. Je kan in principe je SAX-methoden zo schrijven dat een DOM-boomstructuur wordt gegenereerd door de events (zo zijn trouwens wel meer DOM-implementaties gebeurd). Voor onze toepassing moest het document ook eerder doorlopen worden, en dat is veel vanzelfsprekender onder SAX.

Om deze redenen, en het feit dat onder Squeak de kwaliteit van de SAX-implementaties beter waren dan die van DOM, kozen we voor SAX, in de vorm van *YaX (Yet Another XML framework)* [8] voor SmallTalk Squeak.

3.5.2 SAX-methoden

Zoals eerder vermeld wordt het verwerken van XML onder SAX gedaan door een handler-klasse, die beantwoordt aan de SAX-interface van callbacks die

worden afgevuurd bij de verschillende events. Er zijn verschillende interfaces waaraan een SAX-handler kan voldoen, waaronder bijvoorbeeld een DTD-handler die dan event callbacks heeft om DTDs te lezen (zie de Java-specificaties [10]). Hier limiteren we ons tot de basis (en geven dan ook geen verdere uitleg over de technische details in bepaalde methoden, gezien deze buiten de scope van de thesis gaan). Die voorziet de volgende methoden:

- **startDocument**
Afgevuurd bij het begin van een document als het lezen van de XML begint. Deze methode wordt eenmaal opgeroepen, voor alle andere methoden.
- **endDocument**
Afgevuurd op het einde van het document, nadat al het parsen is afgehandeld (of nadat een parser het heeft opgegeven door fouten). Deze methode wordt opgeroepen na alle andere methoden.
- **startPrefixMapping: prefix uri: uri**
Afgevuurd bij het begin van een URI-prefix Namespace-mapping. Aangezien we Namespaces niet gebruiken, verwijzen we hiervoor door naar [10].
- **endPrefixMapping: prefix**
Afgevuurd bij het einde van een URI-prefix Namespace-mapping. Zie voor uitleg ook hier [10].
- **startElement: elementName
namespaceURI: namespaceURI
qualifiedName: qualifiedName
attributeList: attributeList**
Bij de openingstag van een element afgevuurd. De attributen worden meegegeven als een associatielijst. De verschillende parameters geven de lokale naam, de Namespace URI mee en de gekwalificeerde naam. Aangezien wij niet met Namespaces werken, kunnen we de laatste twee negeren en gebruiken we de handige alias die YaX voorziet:
startElement: elementName attributeList: attributeList
- **endElement: elementName
namespaceURI: namespaceURI
qualifiedName: qualifiedName**
Bij de sluitingstag van een element afgevuurd. Dezelfde opmerking als bij **startElement** geldt. Wij concentreren ons op de lokale naam en gebruiken daarom de verkorte vorm:
endElement: elementName
- **characters: aString**
Alle tekstuele inhoud van elementen (parsed character data) wordt,

als één string, in deze methode behandeld. Sommige parsers kunnen tijdens deze methode ook `ignorableWhitespace: aString` oproepen.

- **`ignorableWhitespace: aString`**
Deze methode wordt afgevuurd als er in de tekst whitespace zit die genegeerd mag worden, zoals meer dan één spatie tussen twee opeenvolgende woorden (zie de officiële specificatie [3] voor details). Deze methode is bijna enkel nodig in XML-validerende applicaties.
- **`processingInstruction: piName data: dataString`**
Afgevuurd bij een verwerkingsinstructie. De eerste parameter is het doel, de tweede geeft de rest van de instructie als string. Deze methode zal nooit afgevuurd worden bij de XML-declaratie (`<?xml version="1.0" ?>`).
- **`skippedEntity: aString`**
Afgevuurd als een entiteit overgeslagen wordt. Entiteiten kunnen onder andere worden overgeslagen als ze niet gedeclareerd zijn.

Op deze manier worden, door de verschillende event-methoden, de diverse delen van het XML-document behandeld.

3.6 Conclusie

XML bestaat uit een grote familie standaarden waarvan we de meeste zelfs nog niet vermeld hebben. De XML-talen zelf worden vastgelegd met DTD of XML Schema. XSLT wordt gebruikt om XML te transformeren. DOM en SAX geven applicaties een interface om XML te parsen. Wij verkiezen in ons experiment SAX, omdat SAX beter aansluit bij de werkwijze van ons systeem, en de kwaliteit van de implementaties voor Smalltalk beter zijn vergeleken met DOM.

Hoofdstuk 4

SOUL

In dit hoofdstuk bekijken we de declaratieve taal SOUL, die gebruikt wordt in het thesisexperiment om hogere orde modellen in te noteren, die de co-degeneratie specificeren. Eerst bekijken we de afkomst van SOUL, en daarna haar syntax. Als laatste bespreken we waarom we SOUL gebruiken in plaats van XSLT om code te genereren.

4.1 Afkomst

SOUL [15, 19] staat voor *Smalltalk Open Unification Language*. Het is een reflectieve, logische programmeertaal origineel ontwikkeld voor VisualWorks Smalltalk. Het achterliggende idee was om PROLOG [5, 13] te gebruiken om te redeneren over de structuur van object-georiënteerde talen. Concreet laat SOUL toe regels en feiten zoals in PROLOG te gebruiken, maar bezit het ook nog een verzameling predikaten die verwijzen naar de onderliggende Smalltalk. Met andere woorden, SOUL kan refereren naar de broncode van Smalltalk als een logische database.

Aan de andere kant is er *TyRuBa* (*Type Rule Base*) [17]. TyRuBa is een logische programmeertaal, voornamelijk bedoeld om metaprogramma's te schrijven, die is uitgebreid met een *quasi-quoting* systeem (zoals in Scheme). Quasi-quoting is een procedure die, net zoals gewone 'quoting', een string teruggeeft. Het verschil is dat er in die string variabelen kunnen zitten die eerst geëvalueerd en tussengevoegd worden in de string vooraleer hij wordt teruggegeven (zie sectie 4.2.6 voor een voorbeeld). Dank zij zijn quasi-quoting kan TyRuBa ook strings met logische variabelen interpreteren.

De ideeën van SOUL en TyRuBa werden samengevoegd tot *Quasi-quoted SOUL* of *QSOUL* [4, 9]. Bij de laatste versie van SOUL is besloten om van zijn spin-off QSOUL zijn officiële opvolger te maken. Deze versie heet nu opnieuw SOUL, maar is dus zowel een opvolger voor de originele SOUL als voor QSOUL. De versie die gebruikt wordt in deze thesis is die voor Smalltalk *Squeak* [12, 14], omdat dat de taal is waarin ik mijn experiment uitvoer.

4.2 Syntax

De syntax van SOUL is, met opzet, gelijkaardig aan PROLOG, maar verschilt op een aantal punten. (Voor uitleg over logisch programmeren zelf verwijzen we naar [13].)

4.2.1 Feiten

De basisbouwsteen van een logische taal, het *feit*, dat een relatie weergeeft tussen gegevens, wordt opgeschreven

```
predikaat(term1, term2)
```

De termen zijn *constanten* zoals ‘one’ in

```
number(one)
```

of samengestelde termen, opnieuw gevormd uit een predikaat.

```
number(plus(one, two))
```

Hier is ‘plus(one, two)’ een samengestelde term, waarbij ‘plus’ een functor is (niet te verwarren met een functie zoals in imperatieve programmeertalen; functors worden nooit geëvalueerd maar dienen als een soort recordstructuur om gegevens samen te houden), en daarin ‘one’ en ‘two’ weer constanten.

4.2.2 Regels

Regels leiden uit gekende feiten nieuwe feiten af. Een regel bestaat uit een hoofd met hun naam en argumenten, het keyword ‘if’, en daarna een lijst clauses zoals

```
regel(term1, term2)
  if predikaat1(term3), predikaat2(term4, term5, term6)
```

Bij een SOUL programma worden de verschillende regels en feiten van elkaar gescheiden door punten ‘.’ ertussen (niet na de laatste dus).

4.2.3 Query

Hoe kunnen we informatie afleiden uit een verzameling feiten? Dat gebeurt door middel van een *query*. Een query wordt geformuleerd als een regel zonder hoofd:

```
if predikaat1(term3), predikaat2(term4, term5, term6)
```

waarbij logische variabelen aangeduid worden doordat ze met een vraagteken '?' beginnen. Logische variabelen krijgen de waarden van alle termen in de feiten waarmee het patroon van het predikaat waarin ze staan overeenkomt. Bijvoorbeeld met deze feiten in de database

```
number(one).  
number(two)
```

zal in de query

```
if number(?x)
```

de variabele '?x' de waarden 'one' en 'two' toegekend krijgen. Dit proces noemt men *unificatie*.

Opmerkelijk is dat in SOUL ook de predikaten variabelen mogen zijn bv. '?pred(foo, ?bar)'.

Ter illustratie geven we een typisch logisch programma, over voorouders en nakomelingen. In de database zetten we een aantal feiten die zeggen dat de eerste term de vader is van de tweede.

```
father(John, Martin).  
father(John, Mary).  
father(Steve, John).  
father(Luke, Steve).  
grandfather(?x,?y) if father(?x,?z), father(?z,?y)
```

De regel *grandfather* zegt dat iemand andermans grootvader is als hij de vader is van diens vader. De volgende query uitvoeren

```
if grandfather(?x,?t)
```

geeft dan de koppels van ?x en ?t terug waarvoor ?x de grootvader is van ?t, volgens de feiten in de database. Dit zijn

- (?x -> Luke, ?y -> John)
- (?x -> Steve, ?y -> Martin)
- (?x -> Steve, ?y -> Mary)

4.2.4 Smalltalk-termen

Het unieke aan SOUL is natuurlijk de mogelijkheid om te redeneren over de onderliggende Smalltalk. In SOUL kan in de plaats van de eerder beschreven logische termen ook *Smalltalk-termen* gebruikt worden, die effectief handelen over de Smalltalk-broncode.

Alle uitdrukkingen geschreven tussen vierkante haken ‘[...]’ refereren naar entiteiten uit Smalltalk, en worden ook eerst geëvalueerd onder Smalltalk bij unificatie. Bijvoorbeeld de query

```
if class([Foo])
```

bepaalt of Foo een klasse is in Smalltalk.

In deze termen kunnen even goed logische variabelen gebruikt worden, waarvan de waarde onder de SOUL-database eerst doorgegeven wordt aan Smalltalk voor de evaluatie.

Als tweede voorbeeld: volgende query toegepast op een variabele

```
if class(?class)
```

gaat ook zoeken in de Smalltalk-broncode voor de unificatie met ?class en geeft op die wijze alle klassen in Smalltalk weer op dat moment.

Op deze manieren kan je van in SOUL actief naar Smalltalk-entiteiten refereren.

4.2.5 Smalltalk-clauses

Het idee van een *Smalltalk-clause* is technisch en syntactisch hetzelfde als dat van een Smalltalk-term, namelijk een Smalltalk-uitdrukking, genoteerd tussen vierkante haken, die uitgevoerd wordt door de Smalltalk-interpret. Enkel, een Smalltalk-clause wordt geacht een predikaat te vervangen in plaats van een term, en dient dus een boolean terug te geven.

Bijvoorbeeld met de regel

```
write(?String) if  
  [Transcript show: ?String. true]
```

laat de volgende query

```
if write(['Hello from SOUL'])
```

Smalltalk de string ‘Hello from SOUL’ schrijven in een Transcript.

4.2.6 Quoted code-termen

Tenslotte zijn er de *quoted code*-termen in SOUL, genoteerd tussen accolades ‘{...}’. Deze zijn vergelijkbaar met Smalltalk-strings, enkel kunnen er ook logische variabelen in voorkomen. Deze worden dan in de string vervangen

door de waarde waarmee ze unificeren, zodat er uiteindelijk een volledige Smalltalk-string uitkomt.

De naam 'quoted code' komt van de praktijk dat deze termen vooral worden gebruikt voor codegeneratie. Laten we bijvoorbeeld kijken hoe we accessors zouden genereren voor een klasse onder Smalltalk. Accessors zijn methoden die een instantievariabele van de klasse opvragen of overschrijven.

```
generateAccessorFor(?var, {?var ^var}).  
generateAccessorFor(?var, {?var: anObject ?var := anObject})
```

Deze feiten hebben als eerste term een variabele, waarvan het de bedoeling is dat deze geunificeerd zal worden met een variabelenaam. Die wordt dan ingevuld in de quoted code-termen om zo de Smalltalk-methoden voor de accessors van die variabele te verkrijgen met volgende query

```
if generateAccessorFor(foo, ?code)
```

De gegenereerde Smalltalk-methoden zijn dan

```
?code -> foo  
        ^foo  
?code -> foo: anObject  
        foo := anObject
```

4.3 SOUL vs. XSLT

Wij gebruiken SOUL in het experiment als metataal om code te genereren. We willen hier even onze redenen daarvoor benadrukken, zeker gegeven het feit dat XSLT niet alleen ook kan zorgen voor elke willekeurige tekstuele output, het is ook Turing-compleet en dus in principe een programmeertaal.

Laten we het volgende experiment bekijken over Java-codegeneratie uit een XML-beschrijving. De bedoeling is dat gegeven een XML-document dat een klasse beschrijft, zoals

```
<class>  
<name>Foo</name>  
<super>Object</super>  
<instancevariable>int a</instancevariable>  
<instancevariable>String b</instancevariable>  
<instancevariable>Collection c</instancevariable>  
</class>
```

Java-code wordt gegenereerd dat de klasse definieert als

```

public class Foo extends Super {
    protected int a;
    protected String b,
    protected Collection c;

    public Foo(int a) {}
    public Foo(String b) {}
    public Foo(Collection c) {}
    public Foo(int a, String b) {}
    public Foo(int a, String b, Collection c) {}
}

```

Elke variabele wordt gedefinieerd als instantievariabele, samen met alle juiste constructors voor die variabelen. Om dit te realiseren moet je gebruik maken van algoritmen. Onder andere om de verschillende constructors te kunnen genereren voor meerdere variabelen tegelijk. Met andere woorden, we hebben voor dit soort codegeneratie een volwaardige programmeertaal nodig, waarin we deze algoritmen kunnen neerschrijven.

Het equivalent model in SOUL zou zijn

```

instanceVariable({Foo}, {int}, {a}).
instanceVariable({Foo}, {String}, {b}).
instanceVariable({Foo}, {Collection}, {c})

```

Dit lijkt op het XML-model, dus het enige waar SOUL en XSLT grondig in kunnen verschillen is in de codegeneratie. Waarom dan niet in de XML-familie blijven, en XSLT hiervoor gebruiken, in plaats van eerst een logische taal aan te moeten leren? Dat is een kwestie van expressiviteit van de talen. Bovenstaande generatie uitvoeren neemt in SOUL zo'n 20 lijntjes in. In XSLT echter, zijn er verschillende stijlbladen nodig die samen wel meer dan 500 lijnen bevatten.

Omdat dit voorbeeld te groot is om hier bij te voegen, geven we een ander: we vergelijken een som en produkt-functie, geprogrammeerd in SOUL en XSLT.

De SOUL-code hiervoor is:

```

sum(?list, ?result) if fold(add, 0, ?list, ?result).
product(?list, ?result) if fold(product, 1, ?list, ?result).

fold(?function, ?element, <>, ?element).
fold(?function, ?element, <?first|?rest>, ?result) if
    fold(?function, ?element, ?rest, ?restResult),
    call(?function(?first, ?restResult, ?result))

```

Het call-predikaat dient als een soort 'apply' in SOUL, het kan een ander predikaat oproepen gegeven zijn naam en zijn argumenten. Op die manier

kunnen de ingebouwde binaire predikaten voor optellen en vermenigvuldigen in SOUL opgeroepen worden op de lijst argumenten.

Dezelfde functies gerealiseerd in XSLT:

foldl:

```
<xsl:template name = "foldl" >
  <xsl:param name = "pFunc" select = "/.." />
  <xsl:param name = "pA0" />
  <xsl:param name = "pList" select = "/.." />
  <xsl:choose>
    <xsl:when test = "not($pList)" >
      <xsl:copy-of select = "$pA0" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name = "vFuncResult" >
        <xsl:apply-templates select = "$pFunc[1]" >
          <xsl:with-param name = "arg0"
            select = "$pFunc[position() > 1]" />
          <xsl:with-param name = "arg1"
            select = "$pA0" />
          <xsl:with-param name = "arg2"
            select = "$pList[1]" />
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:call-template name = "foldl" >
        <xsl:with-param name = "pFunc"
          select = "$pFunc" />
        <xsl:with-param name = "pList"
          select = "$pList[position() > 1]" />
        <xsl:with-param name = "pA0"
          select = "$vFuncResult" />
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

sum:

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  xmlns:sum-fold-func = "sum-fold-func"
  exclude-result-prefixes = "xsl sum-fold-func" >
  <xsl:import href = "foldl.xsl" />
  <sum-fold-func:sum-fold-func/>
  <xsl:template name = "sum" >
```

```

<xsl:param name = "pList" select = "/.." />
<xsl:variable name = "sum-fold-func:vFoldFun"
  select = "document('')/*/sum-fold-func:*[1]" />
<xsl:call-template name = "foldl" >
  <xsl:with-param name = "pFunc"
    select = "$sum-fold-func:vFoldFun" />
  <xsl:with-param name = "pList" select = "$pList" />
  <xsl:with-param name = "pA0" select = "0" />
</xsl:call-template>
</xsl:template>
<xsl:template name = "add"
  match = "*[namespace-uri() = 'sum-fold-func']" >
  <xsl:param name = "arg1" select = "0" />
  <xsl:param name = "arg2" select = "0" />
  <xsl:value-of select = "$arg1 + $arg2" />
</xsl:template>
</xsl:stylesheet>

```

product:

```

<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  xmlns:prod-fold-func = "prod-fold-func"
  exclude-result-prefixes = "xsl prod-fold-func" >
<xsl:import href = "foldl.xsl" />
<prod-fold-func:prod-fold-func/>
<xsl:template name = "product" >
  <xsl:param name = "pList" select = "/.." />
  <xsl:variable name = "prod-fold-func:vFoldFun"
    select = "document('')/*/prod-fold-func:*[1]" />
  <xsl:call-template name = "foldl" >
    <xsl:with-param name = "pFunc"
      select = "$prod-fold-func:vFoldFun" />
    <xsl:with-param name = "pList" select = "$pList" />
    <xsl:with-param name = "pA0" select = "1" />
  </xsl:call-template>
</xsl:template>
<xsl:template name = "multiply"
  match = "*[namespace-uri() = 'prod-fold-func']" >
  <xsl:param name = "arg1" select = "0" />
  <xsl:param name = "arg2" select = "0" />
  <xsl:value-of select = "$arg1 * $arg2" />
</xsl:template>
</xsl:stylesheet>

```

In XSLT zijn er geen echte variabelen of functies. De `call-template`-regel wordt misbruikt om functioneel programmeren na te bootsen, maar XSLT is hier niet voor bedoeld.

Hiermee is dus duidelijk aangetoond dat XSLT veel minder expressief is dan SOUL. Expressiviteit is belangrijk omdat expressieve code even krachtig is, maar eenvoudiger, overzichtelijker en makkelijker aanpasbaar. Dit is zeker belangrijk voor ons, omdat wij meta-declaratief programmeren, wij schrijven code die handelt over verschillende talen tegelijk. Dit vraagt een duidelijk, structureel inzicht in de code, en daarom willen we met een zo expressief mogelijke taal werken. Het is duidelijk dat het in XSLT bijna niet haalbaar is om de complexe code te schrijven die wij vereisen. Hiervoor is het expressieve SOUL een veel betere keuze.

4.4 Conclusie

De logische programmeertaal SOUL ligt als een meta-taal bovenop Smalltalk. Het biedt dezelfde mogelijkheden als andere logische talen plus de mogelijkheid om Smalltalk-entiteiten te manipuleren.

We toonden aan dat we voor codegeneratie een volwaardige programmeertaal nodig hebben, maar dat ze daarenboven ook expressief moet zijn. Hierdoor komt XSLT niet in aanmerking, omdat het veel minder expressief is dan onder andere SOUL.

Hoofdstuk 5

Architectuur van het systeem

Dit is het eerste hoofdstuk waarin we het eigenlijke experiment bekijken. Hier leggen we uit hoe we het systeem moeten gebruiken om een invoerplugin, afgesteld op een bepaalde XML-taal, te genereren voor een bepaalde component.

We beginnen met te beschrijven wat hiervoor in SOUL gemodelleerd moet worden en daarna concreet hoe je van deze modellen de invoerplugin laat genereren en werken, aan de hand van de case die we gebruikten.

In de tweede helft van dit hoofdstuk worden de algemene architectuur en de principes van het systeem onder de loep genomen. We bekijken de verschillende lagen, met elk hun eigen taal en nut. Daarna spreken we ook over het gebruik van andere talen, en hoe dat mogelijk is.

5.1 Voorbeeld: iTunes

In het experiment maken gebruik van een ander experiment dat in ook in het kader van het VRT MPEG-project werd uitgevoerd [16], en dat de ontwikkeling bekeek van een architectuur voor de uitwisseling van metadata tussen verschillende applicaties. Het experiment gebruikte audio-applicaties op verschillende platformen, waaronder de MP3-spelers *XMMS* (Linux) en *iTunes* (Mac). De opdrachten en veranderingen door de gebruiker uitgevoerd aan één speler moesten doorgegeven worden via metadata aan de andere spelers. Die verwerkten de data zodat ze uiteindelijk aan hun API gevoerd kon worden om dezelfde opdrachten uit te voeren, zodat alle spelers zich konden synchroniseren. De metadata werd opgeschreven in XML, en door in- en uitvoer plugins van de applicaties respectievelijk gelezen en geschreven.

Het doel van onze case is een plugin te maken die de XML met instructies van XMMS omzet naar API-calls die iTunes aanspreken en de analoge instructies uitvoeren. We leggen eerst de XML en de API uit, en kijken dan hoe we het systeem moeten toepassen.

5.1.1 XMMS-XML

De XML die de action-instructies bevat beschrijft hoofdzakelijk twee elementen: playlists en MP3s. Playlists worden voorgesteld door het XML-element `PlaylistName`, wiens attribuut `type` ofwel "`currentPlayList`" kan zijn voor verandering op de huidige playlist, of "`newPlaylist`", als er een nieuwe playlist moet gemaakt worden. De character data-inhoud geeft de naam van de playlist.

Daarnaast is er het element `MP3`, wat een gelijkaardig attribuut `type` heeft. Hier handelt "`currentMP3`" over de huidige MP3 en "`newMP3`" over een nieuw te creëren MP3. `MP3` heeft als kinderelementen `Key`, een unieke identificatie, en de voor zichzelf sprekende `Title`, `Artist`, `Album`, `Genre`, `Year` en `Comment`.

De verschillende types instructies in de XML zijn

- `newPlaylist`
Creatie nieuwe playlist.
- `editMP3`
Verandert de gegevens van een MP3.
- `addMP3`
Voegt een MP3 toe.
- `deleteMP3`
Verwijdert een MP3.

Zie figuur 5.1 voor een voorbeeld van zo'n XML-document, met de opdracht een MP3 toe te voegen aan de huidige playlist.

5.1.2 iTunes API

AppleScript, de scripting taal van de Mac, wordt gebruikt om te interfacen met iTunes. De gebruikte API-functies van iTunes zijn

- `addPlaylist(aString)`
Voegt een playlist toe met naam `aString`.
- `editTrack(aKey, aTitle, anArtist, anAlbum, aYear, aGenre)`
Verandert de gegevens van de track geïdentificeerd door `aKey`.
- `addTrack(aKey, aPlaylist, aTitle, anArtist, anAlbum, aYear, aGenre)`
Voegt de track geïdentificeerd door `aKey` toe aan de playlist `aPlaylist`.
- `deleteTrack(aKey, aPlaylist)`
Verwijdert de track geïdentificeerd door `aKey` uit de playlist `aPlaylist`.

```

<XMMS>
  <action type="addMP3">
    <Sender>http://localhost:8080</Sender>
    <PlaylistName type="currentPlaylist">
      Super Playlist 5
    </PlaylistName>
    <MP3 type="newMP3">
      <Key>12345</Key>
      <Title>The River</Title>
      <Artist>Bruce Springsteen</Artist>
      <Album>The best of</Album>
      <Year></Year>
      <Genre></Genre>
      <Comment>This is the original version.</Comment>
    </MP3>
  </action>
</XMMS>

```

Figuur 5.1: Voorbeeld van XMMS-XML

Alle argumenten zijn hier strings, ook de keys.

We hebben nu de case die we gebruiken beschreven, maar vooraleer we verder ingaan op hoe we hiervoor een invoerplugin genereren, leggen we eerst het algemeen gebruik uit van ons systeem.

5.2 Modelling

Het programmeren van invoerplugins gebeurt enkel in SOUL. Natuurlijk kan je altijd zelf zaken toevoegen aan de gebruikte regels en structuren, mocht je dat nodig vinden.

5.2.1 Pluginnaam

We geven een naam voor de plugin. Deze wordt de naam van de Smalltalk-klasse waarmee we de plugin gaan besturen. Dit duiden we aan met het predikaat

```
plugin(pluginNaam)
```

Hier is `pluginNaam` een gewone SOUL-term.

We passen dit op onze case toe, en kiezen als naam voor de invoerplugin ‘iTunesPlugin’.

```
plugin(iTunesPlugin)
```

5.2.2 Actionmappings

Actionmappings leggen vast hoe de XML die de plugin binnenkrijgt doorgegeven wordt aan het API van zijn component. Elk XML-document wordt verondersteld instructies te bevatten voor de component. Die instructies worden geëncapsuleerd in een XML-element ‘action’, en onderling worden de types van instructies onderscheiden door het attribuut ‘type’. Een API-functie wordt aan een type instructie gekoppeld door een actionmapping.

```
actionMapping(xmlInstruction, apiInstruction, xmlArguments)
```

`xmlInstruction` is de waarde van het `type`-attribuut in de XML-tag `action` waarmee de instructie geïdentificeerd wordt; `apiInstruction` is de naam van de API-functie die ermee overeenstemt. `xmlArguments` is een SOUL-lijst van de XML-elementen die de waarden bevatten voor de argumenten van de API-functie, in de volgorde waarin ze daarin moeten voorkomen. De volgorde of diepte van genestheid in de XML heeft geen belang, zolang ze maar staan in hetzelfde `action`-element.

Als voorbeeldje voor het schrijven van een actionmapping nemen we de `deleteTrack`-functie uit het API van iTunes, en de XML die ermee gekoppeld is.

```
deleteTrack(aKey, aPlayList)
```

Deze functie neemt een MP3, bepaald door zijn key, en verwijdert deze uit een gegeven playlist. De XML die bijvoorbeeld opdraagt de MP3 met key ‘12345’ te verwijderen uit playlist ‘Super Playlist 5’ zou er zo uit zien:

```
<XMMS>
  <action type="removeMP3">
    <Sender>http://localhost:8080</Sender>
    <PlaylistName type="currentPlaylist">
      Super Playlist 5
    </PlaylistName>
    <MP3 type="currentMP3">
      <Key>12345</Key>
      <Title>The River</Title>
      <Artist>Bruce Springsteen</Artist>
      <Album>The best of</Album>
      <Year></Year>
      <Genre></Genre>
    </MP3>
  </action>
</XMMS>
```

```

        <Comment>This is the original version.</Comment>
    </MP3>
</action>
</XMMS>

```

In SOUL zouden we dan volgende actionMapping schrijven:

```

    actionMapping(removeMP3, deleteTrack, <Key, PlayListName>)

```

Het laatste symbool is een SOUL-lijst¹.

De overige iTunes API-functies zouden op gelijkaardige manier gemodelleerd worden.

```

    actionMapping(addPlaylist, newPlaylist, <PlaylistName>).
    actionMapping(editTrack, editMP3,
        <Key, Title, Album, Year, Genre>).
    actionMapping(addTrack, addMP3,
        <Key, PlaylistName, Title, Album, Year, Genre>)

```

Merk op dat de volgorde en diepte van de elementen in de XML volledig vrij is. De restricties die op de XML liggen zijn het gebruik van het `action`-element, en de groepering van alle elementen die argumentwaarden bevatten voor een instructie onder hetzelfde `action`-element, want meerdere instructies per XML-document zijn toegelaten.

5.3 Generatie en gebruik

Nadat de pluginnaam en alle actionmappings zijn gemodelleerd, laat de volgende query alle nodige Smalltalk klassen genereren²:

```

    if generateClass(?)

```

Nadien zal er een nieuwe Smalltalk-klassecategorie zijn, getiteld ‘Generated’, met daarin alle gegenereerde klassen, waaronder ook de invoerplugin. Om deze XML te laten verwerken gebruik je de Smalltalk klassemethode

```

    pluginNaam >> parseFile: fileName

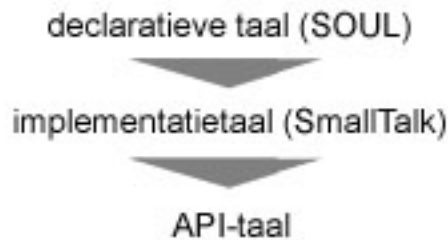
```

waar `fileName` het in te lezen XML-document is. De methode geeft dan de vertaling van de instructies in de XML naar API-calls terug als string.

Als we onze iTunesPlugin hebben beschreven als vermeld, dan maakt de `generateClass`-query de Smalltalk-klasse `iTunesPlugin` aan. Laten we deze bijvoorbeeld de XML lezen, te zien in figuur 5.1, om een MP3 toe te voegen,

¹Lijsten worden in SOUL genoteerd als ‘<a, b, c>’. Dit is eigenlijk een verkorte vorm voor ‘<a | <b | <c | <>>>>’ (analoog aan lijsten in Scheme). De notatie ‘<head | tail>’ laat toe de lijst op te splitsen in een hoofd en staart.

²In SOUL is ‘?’ de logische variabele die met alles unificeert, zoals ‘_’ in PROLOG.



Figuur 5.2: De architectuur van de 3 talen

```
iTunesPlugin parseFile: 'XMMS_addMP3.xml'
```

dan geeft de invoerplugin de string³ met de correcte API-call.

```
'addTrack(''http://localhost:8080'',
           ''Super Playlist 5'',
           ''12345'',
           ''The River'',
           ''Bruce Springsteen'',
           ''The best of'',
           ''',
           ''',
           ''This is the original version.'')'
```

Dit is exact de manier waarop de `addTrack`-functie moeten worden aangesproken, zoals beschreven in de XML die we wouden vertalen. Merk op dat de lege XML-elementen vertaald worden als lege strings.

5.4 Architectuur

Vooraleer we aan de implementatie beginnen, bekijken we eerst de algemene architectuur van ons systeem.

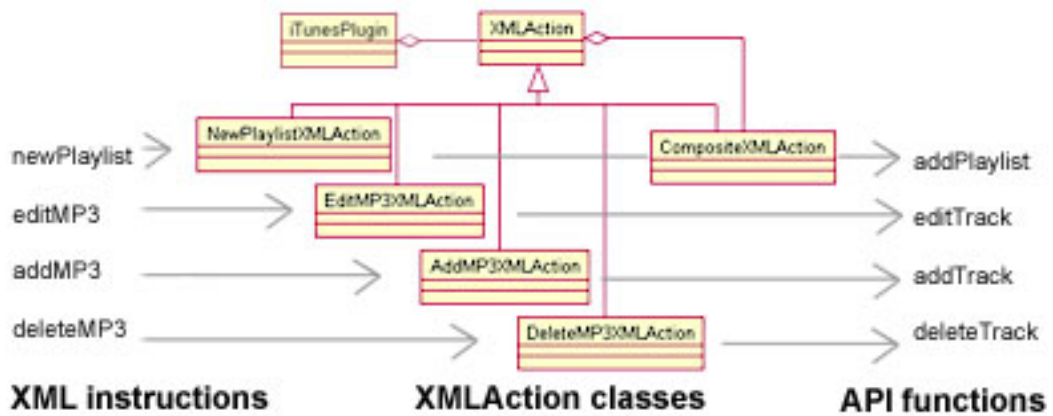
We hanteren in ons systeem (generator en plugins) drie verschillende talen. Elke taal genereert of manipuleert code in de onderliggende laag (zie figuur 5.2).

5.4.1 De Declaratieve laag

De declaratieve laag stuurt het systeem en modelleert de plugin. Zoals eerder vermeld, gebruiken wij als declaratieve taal SOUL, uitgelegd in hoofdstuk 4.

Eerst en vooral moeten we hier regels voorzien die klassen, hun variabelen en hun methoden in de onderliggende Smalltalk kunnen aanmaken. Die

³In Smalltalk is '' de escape voor een single quote.



Figuur 5.3: Mapping van XML op API door XMLActions

worden gehanteerd om de gebruikte klassen mee te maken, die de invoerplugin implementeren in de onderliggende laag.

De modellen voor de API worden opgeschreven in deze taal met behulp van actionmappings, zoals eerder uitgelegd. Daarna gaan we kijken naar het probleem dat we eerder aanbrachten: als we de code voor de API automatisch willen genereren vanuit de XML, moeten we de instructie-XML vertalen naar de API van de component. Hoe doen we dit? Om een hoop conditionele code te vermijden die zou moeten bepalen wat moet uitgevoerd worden voor welke instructie, maken we gebruik van het Strategy design pattern[6]. Om kort uit te leggen: een Strategy definieert een familie van algoritmen, encapsuleert deze, en maakt ze onderling uitwisselbaar. Wij doen dit door voor elke mogelijke instructie, dus elke actionmapping, een bijhorende *XMLAction*-klasse te genereren (zie ook figuur 5.3). Deze klassen abstraheren een instructie, ze houden een lijst van argumenten bij nodig voor die instructie. Ze worden door de plugin gebruikt als hij de XML leest. XMLActions hebben methoden voor de manipulatie van hun argumentenlijst en een methode die hen uitvoert, dit wil zeggen die hen de API-code laat schrijven. In het volgende hoofdstuk worden XMLActions en hun methoden in detail uitgelegd.

Bij elke nieuwe instructie die begint wordt een instantie gemaakt van de corresponderende XMLAction-klasse (genomen uit een lijst van alle mogelijke XMLActions, bepaald door de actionmappings die gedefinieerd waren) en bijgehouden in de invoerplugin. Die XMLAction verzamelt dan alle argumenten tot de instructie in de XML eindigt.

Dit gaat zo verder tot alle instructies in de XML bewerkt zijn. De verschillende XMLActions worden bijgehouden in een instantie van de klasse *CompositeXMLAction*, door middel van een Composite pattern[6].

5.4.2 De Implementatielaag

De *implementatie-laag* is de laag waarover de declaratieve taal redeneert en de taal waarin de plugin en de ondersteunende klassen, zoals `XMLAction`, zijn geïmplementeerd. SOUL is een meta-taal voor Smalltalk, dus de implementatie gebeurt daarin. Het dialect dat wij gebruiken is Squeak Smalltalk. De keuze van de taal hier bepaalt ook de bruikbare technologie in de plugins. Bijvoorbeeld wij moesten een parser vinden om XML in te lezen die werkt onder Squeak Smalltalk.

De implementatie-taal zou degene zijn waar al het (repetetieve) programmeerwerk in gebeurt als er geen enkele automatische generatie of bovenliggende declaratieve redenering zou zijn. Het enige schrijfwerk dat nu hierin gebeurt in ons experiment, voornamelijk om de skeletstructuren van de ondersteunende klassen te specificeren, wordt eenmalig in de regels van de declaratieve taal gedaan.

5.4.3 De API-laag

Als onderste laag is er de *API-laag*, die de component gebruikt als interface. Deze component-specifieke API-calls zijn de uitvoertaal van de plugin. Deze API-taal wisselt constant afhankelijk van de component waarvoor een plugin wordt geschreven.

Als de `XMLAction`-klassen alle argumenten hebben verzameld uit de XML-invoer voor hun functies, schrijven zij de code voor de API-call, met de argumenten in de juiste plaats ingevuld. Deze API-code wordt dan uitgevoerd door middel van ofwel rechtstreekse aanroepen in de implementatietaal, ofwel door een tekstueel scriptbestand te maken dat later aan de component gevoerd kan worden. Op het moment gebruiken wij de laatste mogelijkheid, namelijk de invoerplugin geeft een Smalltalk-string terug, maar dit hangt af van de configuratie implementatietaal/API-taal.

Enkel de structuur van de API-functies moet gemodelleerd worden in de actionmappings. De implementatielaag staat in voor het eigenlijke schrijven van de code.

5.4.4 Taalafhankelijkheid

We proberen om zoveel mogelijk taalafhankelijkheid te voorzien, met andere woorden het systeem moet werken voor eender welke API-taal, en geïmplementeerd kunnen worden door eender welke programmeertaal, zonder dat alles in het declaratieve niveau moet veranderd moet worden. Dit kan in de praktijk gebracht worden op twee verschillende manieren.

Een eerste vorm van onafhankelijkheid is dat de eerste twee lagen hetzelfde blijven, maar dat een andere API-taal wordt gebruikt door de implementatietaal, bijvoorbeeld bij een port van het onderliggende componenten-

systeem naar een andere besturingssysteem. Dit is ook weer aanpasbaar in de declaratieve taal.

De tweede vorm baseert zich op onafhankelijkheid op niveau van de implementatietaal. Aangezien het enige onderhoudende programmeerwerk voor generatie gebeurt in de declaratieve taal, kunnen we de daaronder liggende taal makkelijk veranderen, zolang de declaratieve taal deze maar kan manipuleren. Hetzelfde principe geldt voor de implementatie- en de API-taal: aanpassingen in de API-uitvoer worden enkel veroorzaakt door de implementatietaal. Het kan belangrijk zijn dat de implementatietaal veranderd kan worden, als er geen op zichzelf staande API-taal is, omdat het API misschien enkel via de implementatietaal kan aangesproken worden, bijvoorbeeld bij een database.

Het is voor de taalafhankelijkheid belangrijk dat er duidelijk onderscheid wordt gemaakt tussen de drie talen waarmee het systeem werkt, en dat er in de declaratieve modellering geen afhankelijkheden tussen bestaan.

5.5 Conclusie

Het systeem vertaalt de invoer-XML naar de API-calls, door actionmappings opgeschreven in SOUL. Die koppelen elke mogelijke instructie in de XML aan een functie in het API van de component, en bepalen hoe de argumenten daarvoor uit de XML moet worden gehaald. De case die wij gebruiken in het experiment is een vertaling van instructies voor MP3-spelers in XML, naar het API van iTunes, een MP3-speler voor de Mac.

In feite is er weinig dat gespecificeerd moet worden, tenzij je andere technologie gaat gebruiken, zoals een andere SAX-parser of implementatietaal.

Qua architectuur bestaat het systeem uit drie lagen. SOUL is de declaratieve taal in de bovenste laag. Deze genereert Smalltalk, de implementatietaal in de daaronder liggende laag. Smalltalk wordt gebruikt om verwerking van XML tot de laatste laag, API-calls, te implementeren. Taalafhankelijkheid voor verschillende APIs is aanwezig in twee mogelijke vormen. Ofwel wordt de API-taal vervangen, maar blijft de implementatietaal dezelfde, ofwel wordt de implementatietaal vervangen en verandert dus ook de manier waarop die interfacet naar de API.

Hoofdstuk 6

Implementatie van het experiment

Tenslotte bekijkt dit hoofdstuk hoe we alles geïmplementeerd hebben. De eerste sectie beschrijft de predikaten die we schreven om vanuit SOUL in Smalltalk klassen te creëren. De volgende sectie beschrijft XMLAction-klassen, de implementatie van actionmappings. De laatste sectie behandelt de werking en generatie van de controlerende klasse van de invoerplugin.

6.1 Generatie van Smalltalk-klassen

Eerst maken we een aantal predikaten die toelaten vanuit SOUL klassen te definiëren in Smalltalk Squeak.

```
generateClass(?) if
    gClass(?className, ?superclassName, ?category),
    className(?superClass, ?superclassName),
    makeClass(?className, ?superClass),
    generateMethods(?className),
    generateClassMethods(?className),
    generateVariables(?className),
    generateClassVariables(?className)
```

Dit predikaat gaat alle klassen genereren in Smalltalk die beschreven staan met het predikaat `gClass`. Daarin wordt gemodelleerd de naam van de klasse (`?className`), de naam van de klasse waarvan de nieuwe klasse moet overerven (`?superclassName`), en de categorie (`?category`). De eerste twee parameters zijn SOUL-termen, de categorie een Smalltalk-symbool.

Vb.

```
gClass(CompositeXMLAction, XMLAction, [#Generated])
```

Het in SOUL ingebouwde predikaat `className` verbindt een klasserepresentatie (als Smalltalk-term) als eerste term met zijn klassenaam als tweede. Vb.

```
className([XMLAction], XMLAction)
```

De eigenlijke creatie van de klasse gebeurt in `makeClass`, maar daarover vertellen we straks meer.

De volgende predikaten definiëren methoden.

```
generateMethods(?className) if
  className(?class, ?className),
  forall(
    gMethod(?className, ?protocol, ?code),
    makeMethod(?class, ?code, ?protocol))
```

Zo gaat `generateMethods`, voor de klasse die gespecificeerd wordt, voor alle methoden aangeduid met `gMethod`, `makeMethod` uitvoeren. Hierbij is `?protocol` de methode-categorie waarin de methode geplaatst gaat worden en `?code` de eigenlijke code van de methode als quoted-code string. Vb.

```
gMethod(TESTCompositeXMLAction, [#accessing],
  {actions ^actions})
```

Op analoge wijze laat je klassemethoden, aangeduid met `gClassMethod`, aanmaken door het predikaat `generateClassMethods`.

Dan zijn er de predikaten voor de aanmaak van de variabelen van een klasse.

```
generateVariables(?className) if
  className(?class, ?className),
  forall(
    gInstVar(?className, ?varName),
    makeInstanceVariable(?class, ?varName))
```

waarbij de instantievariabelen worden aangeduid met `gInstVar`, samen met de klassen waarvoor ze gegenereerd moeten worden (`?className`) en hun naam (`?varName`). Idem voor klassevariabelen aangeduid met het predikaat `gClassVar`.

6.1.1 Interfacing naar Smalltalk

In al deze regels zorgen de verschillende predikaten als `makeClassVariable`, `makeClass`, enzoverder voor de eigenlijke aanmaak van de entiteiten in Smalltalk. De meest voor de hand liggende manier om dit te doen is met Smalltalk-clauses, zoals bijvoorbeeld voor aanmaak van een klasse

```

makeClass(?class, ?superClass) if
  not(class(?class)),
  [?superClass
    subclass: ?className
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Generated']

```

Dit is in Smalltalk de standaard manier om een nieuwe klasse te creëren: door een message te sturen naar zijn superklasse.

Omdat SOUL gebruikt wordt in meerdere Smalltalk-dialecten is er een abstractie gemaakt van code in Smalltalk-clauses. De *Meta Level Interface* encapsuleert dialect-specifieke code voor grotere draagbaarheid. Instructies worden naar de MLI gestuurd, en die roept de correcte Smalltalk op. Op die manier wordt de `makeClass`-regel

```

makeClass(?class, ?superClass) if
  not(class([?class])),
  [QSOULExplicitMLI current
    addClass: ?class asSubClassOf: ?superClass. true]

```

met in de MLI de (Smalltalk-)implementatie

```

QSOULExplicitMLI >>
  addClass: className asSubClassOf: superClass
  superClass
    subclass: className
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Generated'

```

Deze extra abstractielaag heeft geen invloed op onze thesis, daarom gaan we voor de eenvoudigheid aannemen dat we rechtstreeks Smalltalk-clauses schrijven. Onder die aanname is de implementatie van `makeMethod`

```

makeMethod(?class, ?code, ?protocol) if
  [?class
    compile: (Text fromString: ?code asString)
    classified: ?protocol]

```

En voor de aanmaak van instantievariabelen

```

makeInstanceVariable(?class, ?name) if
  [?class addInstVarName: ?name asString]

```

Klassemethoden en -variabelen worden op eenzelfde manier aangemaakt.

6.2 XMLAction-klassen

XMLAction-klassen, zijn Smalltalk-klassen die een mogelijke instructie vertegenwoordigen in de XML-invoer, en dus een corresponderende API-functie voor de component. Een XMLAction wordt door de plugin gebruikt als implementatie van een actionmapping. Zij bewaren de nodige argumenten voor hun instructie die uit de XML gehaald worden, en schrijven daarna de code voor de API-call.

Eerst leggen we in verder detail uit hoe XMLActions werken, daarna zien we hoe ze gegenereerd worden vanuit SOUL. Denk eraan dat we wel uitleggen hoe de klassen werken in Smalltalk, maar dat we ze niet direct schrijven in Smalltalk.

6.2.1 Werking in Smalltalk

Voor elke actionmapping wordt een aparte XMLAction-klasse aangemaakt. Deze erven allen over van eenzelfde basisklasse: `XMLAction`. Die definieert een associatielijst ‘arguments’ als instantievariabele. Daar kan een argumentwaarde aan toegevoegd worden met

```
XMLAction >> addArgument: anArgumentName value: aString
               self arguments at: anArgumentName put: aString
```

Als de XML wordt ingelezen, worden de waarden van de argumenten voor de betreffende instructie eruit gehaald en in deze associatielijst gestoken, met als sleutel hun XML-elementnaam als Smalltalk-symbool. Bijvoorbeeld het element dat de titel beschrijft in de XML als

```
<Title>The River</Title>
```

wordt opgeslagen als argument in een XMLAction met

```
anXMLAction addArgument: #Title value: "The River"
```

Het opvragen van een argument wordt gedaan met

```
XMLAction >> argumentAt: aString
              ^ self arguments
                at: aString
                ifAbsent: [']
```

Het schrijven van de API-code wordt ingevuld door de specifieke kinderklassen.

```
XMLAction >> execute
               self subclassResponsibility
```

Omdat een XML-document meerdere instructies kan bevatten, definiëren we de klasse `CompositeXMLAction` die `XMLActions` bij kan houden. Alle gebruikte `XMLActions` gaan in een lijst `'actions'`. Bij de uitvoering laat `CompositeXMLAction` deze na elkaar uitvoeren, verzamelt hun API-code op een stream en geeft deze terug.

```
CompositeXMLAction >> execute
  | stream |
  stream := WriteStream on: ''.
  self actions
    do: [:action | stream nextPutAll: action execute].
  ^ stream contents
```

De eigenlijke `XMLActions` worden specifiek voor een actionmapping gegenereerd. Wat er gebeurt als ze uitgevoerd worden, is: hun overeenstemmende API-functie wordt geschreven, gevolgd door alle argumenten die in de correcte volgorde uit de argumentenlijst worden gehaald.

Laten we als voorbeeld de *addMP3*-instructie gebruiken (XML-instructie in figuur 5.1) uit onze case. Deze komt overeen met de iTunes API-functie

```
addTrack(aKey, aPlaylist, aTitle,
         anArtist, anAlbum, aYear, aGenre)
```

Voor deze instructie zou de klasse `AddMP3XMLAction` gecreëerd worden wiens `execute`-methode er zo zou uit zien:

```
AddMP3XMLAction >> execute
  | stream |
  stream := WriteStream with: #addTrack asString , '(''.
  stream nextPutAll: ''''
    , (self argumentAt: #Sender) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #PlaylistName) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Key) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Title) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Artist) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Album) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Year) , ''', '.
  stream nextPutAll: ''''
    , (self argumentAt: #Genre) , ''', '.
```

```

gClass(XMLAction, Object, [#Generated]).

gInstVar(XMLAction, arguments).

gMethod(XMLAction, [#accessing],
  { addArgument: anArgumentName value: aString
    self arguments at: anArgumentName put: aString }).
gMethod(XMLAction, [#accessing],
  { argumentAt: aString
    ^ self arguments at: aString ifAbsent: ['']}).
gMethod(XMLAction, [#accessing],
  { arguments
    ^ arguments }).
gMethod(XMLAction, [#'initialize-release'],
  { initialize
    arguments := Dictionary new }).
gMethod(XMLAction, [#executing],
  { execute
    self subclassResponsibility }).

gClassMethod(XMLAction, [#'instance creation'],
  { new
    ^ super new initialize })

```

Figuur 6.1: SOUL-code voor generatie XMLAction

```

stream nextPutAll: '''
    , (self argumentAt: #Comment) , ''''.
stream nextPut: $).
^ stream contents

```

De argumenten worden één voor één uit de lijst gehaald en als string, dus tussen single quotes, op een stream geschreven. De inhoud van deze stream wordt daarna teruggeven.

Voor de rest zijn de methodes van de specifieke XMLAction-klassen hetzelfde als die van de basis XMLAction.

6.2.2 Generatie in SOUL

De klasse XMLAction en CompositeXMLAction worden gegenereerd door hun onderdelen eenvoudigweg op te schrijven in de SOUL predikaten die we hebben gebouwd om Smalltalk-klassen mee aan te maken. Voor de volledige code zie respectievelijk figuur 6.1 en 6.2.

```

gClass(CompositeXMLAction, XMLAction, [#Generated]).

gInstVar(CompositeXMLAction, actions).

gMethod(CompositeXMLAction, [#accessing],
  { actions
    ^ actions}).
gMethod(CompositeXMLAction, [#accessing],
  { addAction: anAction
    self actions add: anAction}).
gMethod(CompositeXMLAction, [#'initialize-release'],
  { initialize
    actions := OrderedCollection new}).
gMethod(CompositeXMLAction, [#executing],
  { execute
    | stream |
    stream := WriteStream on: ''.
    self actions do:
      [:action | stream nextPutAll: action execute].
    ^ stream contents})

```

Figuur 6.2: SOUL-code voor generatie CompositeXMLAction

Specifieke XMLActions zijn iets ingewikkelder omdat ze geen unieke vastgelegde vorm hebben, maar afhangen van een actionmapping. Zoals al eerder gezien, worden actionmappings geschreven als

```
actionMapping(xmlInstruction, apiInstruction, xmlArguments)
```

We schrijven een regel die voor elk van die mappings een XMLAction-klasse maakt:

```
gClass(?className, XMLAction, [#Generated]) if
  actionMapping(?action, ?, ?),
  actionClassName(?action, ?className).
```

De regel kijkt of er een actionmapping bestaat, neemt de actie (de overige termen zijn op dit moment onbelangrijk, daarom de variabele ‘?’ die met alles unificeert) en gaat daarvoor een klasse definiëren. De naam van die klasse wordt gemaakt met de regel

```
actionClassName(?action,?className) if
  atom(?action),
  equals(?className,
    [(?action capitalized , 'XMLAction') asSymbol ])
```

Deze regel neemt de actie binnen, zet het SOUL-symbool om in een Smalltalk-symbool met ‘XMLAction’ eraan geconcateneerd. Bijvoorbeeld ‘addMP3’ zou ‘AddMP3XMLAction’ opleveren.

Tot zover de klassen. De enige methode die nog gedefinieerd moet worden is de execute-methode die degene van XMLAction overschrijft, de rest wordt onveranderd overgeërfd.

```
gMethod(?className, [#executing], ?code) if
  actionMapping(?action, ?function, ?args),
  actionClassName(?action, ?className),
  generateMethodCode(?function, ?args, ?code).
```

Deze regel laat voor elke actionmapping, in zijn overeenkomstige XMLAction-klasse, zoals bepaald in de regels die we net zagen voor de generatie daarvan een methode maken. De eigenlijke code (?code) wordt gegenereerd door de API-functie-aanroep ?function en argumentenlijst ?args mee te geven aan

```
generateMethodCode(?function, ?args,
  { execute
    | stream |
    stream := WriteStream with: (?function asString) , '( '.
    ?argumentCode
    stream nextPut: $).
    ^ stream contents }) if
  generateArgumentCode(?args, ?argumentCode).
```

Dit geeft de Smalltalk-code voor de `execute`-method, die je zal herkennen van de vorige sectie over de werking. De API-functie-aanroep wordt hier ingevuld, de code voor het refereren van de argumenten (`?argumentCode`) wordt gedaan door een paar laatste regels:

```
generateArgumentCode(<?el>,
  { stream nextPutAll: ''', (self argumentAt: ?el), '''. }).

generateArgumentCode(<?first|?rest>,
  { stream nextPutAll: ''',
    (self argumentAt: ?first), ''', '. ?restCode }) if
  generateArgumentCode(?rest,?restCode)
```

Dit is een recursieve verwerking van de SOUL-lijst van argumenten van de actionmapping. Als het eerste argument opgesplitst kan worden in een lijst, unificeert het met de tweede regel. Die verwerkt het hoofd van de lijst tot Smalltalk-code en plakt `?restCode` eraan, de recursieve verwerking van de rest. Als je aan het einde bent en er is nog maar één element, dan maakt de eerste regel het af, en stopt de verwerking.

Op die manier wordt voor elke argument in de lijst een stukje code gemaakt dat de XMLAction-klasse bij uitvoering in zijn argumentenlijst de waarde laat lezen met als key dit argument, en daarmee is de methode voltooid.

6.3 Plugin

Om de SAX-parserklasse die de plugin bestuurt uit te leggen, volgen we hetzelfde stramien als voor de XMLAction-klassen: eerst leggen we uit hoe de klasse zou werken in Smalltalk na generatie, daarna bekijken we die generatie zelf vanuit SOUL.

6.3.1 Werking in Smalltalk

Zoals eerder vermeld is de invoerpluginklasse een SAX-parser, dus om te beginnen moet ze overerven van de SAX-handler die we gebruiken. Bij de YaX-implementatie is dat de klasse `SAXHandler`. We werken nog altijd met iTunes als voorbeeld, en daarbij is de definitie van de pluginklasse in Smalltalk

```

SAXHandler subclass: #iTunesPlugin
    instanceVariableNames: 'resultAction
                           currentAction
                           currentElement '
    classVariableNames: 'APIResult
                        ActionClasses '
    poolDictionaries: ''
    category: 'Generated'

```

De pluginklasse heeft een associatielijst van alle mogelijke XMLAction-klassen die gebruikt kunnen worden tijdens het lezen van de XML, geïndexeerd op de overeenstemmende types van XML-instructies als Smalltalk-symbolen. Deze lijst wordt aangemaakt tijdens de initialisatie van een instantie van de klasse, en bijgehouden in de klassevariabele `ActionClasses`.

```

iTunesPlugin >> initialize
    super initialize.
    ActionClasses := Dictionary new.
    ActionClasses at: #newPlaylist put: NewPlaylistXMLAction.
    ActionClasses at: #editMP3 put: EditMP3XMLAction.
    ActionClasses at: #addMP3 put: AddMP3XMLAction.
    ActionClasses at: #deleteMP3 put: DeleteMP3XMLAction.
    resultAction := CompositeXMLAction new

```

Een tweede klassevariabele `APIResult` dient om de resulterende API-calls in bij te houden na het vertalen van de instructie-XML (zie ook de methoden `parseFile` en `endDocument`).

De instantievariabelen van de plugin zijn

`resultAction` Een instantie van `CompositeXMLAction`. Telkens een instructie in de XML verwerkt is door een `XMLAction` en deze alle argumenten heeft geëxtraheerd, wordt de `XMLAction` hierin bewaard.

`currentAction` Een instantie van één van de `XMLAction`-klassen uit de lijst in `ActionClasses`. Deze houdt de `XMLAction` bij die bezig is een instructie te verwerken.

`currentElement` Een string die de naam is van het huidige XML-element dat gelezen wordt.

De klassemethode `parseFile` wordt opgeroepen om de pluginklasse een XML-document in te laten lezen.

```

iTunesPlugin class >> parseFile: fileName
  | stream |
  stream := FileDirectory default readOnlyFileName: fileName.
  [self parseDocumentFrom: stream]
    ensure: [stream close].
  ^ APIResult

```

In deze methode wordt het eigenlijke parsen van het XML-bestand doorgegeven aan de SAX-handler met de methode `parseDocumentFrom`. Als laatste expressie wordt de waarde van de klassevariabele `APIResult` teruggegeven, die de uitgeschreven API-calls bevat, ingevuld op het einde van het parsen.

SAX-methoden

De eigenlijke verwerking gebeurt in de SAX-methoden van de invoerplugin. Bij het begin van een document hoeft er niets bijzonders te gebeuren. De SAX-methode `startDocument` hoeft dus niet overschreven te worden. De eerste verwerking gebeurt bij het lezen van een element.

```

iTunesPlugin >> startElement: elementName attributeList: aList
  elementName = 'action'
    ifTrue: [currentAction := (ActionClasses
      at: (self getAction: aList)) new]
    ifFalse: [currentElement := elementName]

```

In de SAX-methode `startElement` wordt gekeken naar de naam van het element dat opent, en dus doorgegeven wordt in deze methode. Is dit `action`, dan betekent dit dat dit element een nieuwe instructie bevat en zal er een nieuwe `XMLAction` aangemaakt worden afhankelijk van het type instructie, wat bepaald is in het gelijknamige XML-attribuut `type`. Dit type wordt uit de attributenlijst gehaald met de hulpmethode `getAction`.

```

iTunesPlugin >> getAction: aList
  ^ (aList at: 'type') asSymbol

```

Als we dit type hebben gebruiken we het als sleutel om de corresponderende `XMLAction`-klasse uit de lijst in `ActionClasses` te halen en te instantiëren. Deze instantie wordt de `currentAction`.

In het tweede geval, het element is geen `action`-element, wordt het gewoon bijgehouden in de variabele `currentElement`.

Als we de karakterdata in een element tegenkomen wordt de SAX-methode `characters` afgevuurd.

```

iTunesPlugin >> characters: aString
  ^ currentAction addArgument: currentElement asSymbol
    value: aString

```

Deze data is de waarde van een argument benodigd voor een API-functie, waarbij de naam van het element de aard van het argument aanduidt. Deze methode neemt de waarde, en voegt ze toe aan de huidige XMLAction (zie sectie 6.2.1) met als sleutel de naam van het huidige element. Hiermee bezit de XMLAction de argumentwaarde voor dat element.

De SAX-methode `endElement` wordt afgevuurd als een XML-element sluit.

```
iTunesPlugin >> endElement: elementName
    elementName = 'action'
    ifTrue: [resultAction addAction: currentAction]
```

Deze methode moet enkel iets uitvoeren als het sluitende element een `action`-element is. In dit geval is de instructie beëindigd en wordt de XMLAction, wiens werk voltooid is, toegevoegd aan de `CompositeXMLAction` in `resultAction`.

Nadat heel de verwerking van het proces is afgelopen wordt tenslotte `endDocument` uitgevoerd.

```
iTunesPlugin >> endDocument
    APIResult := resultAction execute
```

Deze laat de XMLActions uitvoeren (zie sectie 6.2.1), welke de API-calls aan de hand van hun verzamelde argumenten schrijven, en plaatst het resultaat in de klassevariabele `APIResult`.

6.3.2 Generatie in SOUL

Om te beginnen modelleren we de event handler-klasse die we gebruiken (waarvan de plugin moet overerven aangezien hij een SAX-parser is) met het SOUL-predikaat `saxHandler`.

```
saxHandler(SAXHandler).
```

De verschillende SAX-methoden, zoals hierboven beschreven, leggen we vast met `saxMethod` en `saxClassMethod`, waarbij we ook hier, zoals in `gMethod` en `gClassMethod`, de klasse specificeren waarvoor de methode gemaakt wordt, de methode-categorie en de eigenlijke code van de methode.

```

saxClassMethod(?plugin, [#parsing],
  { parseFile: fileName
    | stream |
    stream := FileDirectory default
      readOnlyFileName: fileName.
    [self parseDocumentFrom: stream]
      ensure: [stream close].
    ^APIResult }) if
  plugin(?plugin).

saxMethod(?plugin, [#content],
  { characters: aString
    ^ currentAction addArgument: currentElement asSymbol
      value: aString}) if
  plugin(?plugin).

saxMethod(?plugin, [#content],
  { endDocument
    APIResult := resultAction execute}) if
  plugin(?plugin).

saxMethod(?plugin, [#content],
  { endElement: elementName
    elementName = 'action'
    ifTrue: [resultAction addAction: currentAction]}) if
  plugin(?plugin).

saxMethod(?plugin, [#content], { startDocument
    ^ self}) if
  plugin(?plugin).

saxMethod(?plugin, [#content],
  { startElement: elementName attributeList: aList
    elementName = 'action'
    ifTrue:
      [currentAction := (ActionClasses at:
        (self getAction: aList)) new]
    ifFalse:
      [currentElement := elementName]}) if
  plugin(?plugin)

```

Merk op dat we deze methoden altijd laten genereren voor klassen die een plugin zijn, door de `plugin`-clause op het einde van de regels wiens variabele unificeert met de klasse in het hoofd van de regel.

De pluginnaam wordt gegeven in ons SOUL-model door het predikaat `plugin`, en daarmee hebben we nu voldoende informatie om deze klasse te laten genereren.

```
gClass(?plugin, ?saxHandler, [#Generated]) if
    saxHandler(?saxHandler),
    plugin(?plugin).
```

We geven aan dat SAX-methoden gegenereerd moeten worden op dezelfde manier als andere methoden met volgende regels

```
gMethod(?saxClass, ?protocol, ?code) if
    saxMethod(?saxClass, ?protocol, ?code).
```

```
gClassMethod(?saxClass, ?protocol, ?code) if
    saxClassMethod(?saxClass, ?protocol, ?code).
```

De reden dat we aparte notaties gebruiken voor de SAX-methoden en dergelijke, is omdat dit ons een extra abstractie geeft op het model van de SAX-parser, wat het makkelijker maakt deze te vervangen.

We modelleren ook de in de vorige sectie beschreven variabelen die elke invoerplugin nodig heeft.

```
gInstVar(?plugin, resultAction) if
    plugin(?plugin).
```

```
gInstVar(?plugin, currentAction) if
    plugin(?plugin).
```

```
gInstVar(?plugin, currentElement) if
    plugin(?plugin).
```

```
gClassVar(?plugin, APIResult) if
    plugin(?plugin).
```

```
gClassVar(?plugin, ActionClasses) if
    plugin(?plugin).
```

Als laatste onderdeel van de plugin-klasse is er de initialisatie-methode. Deze is iets ingewikkelder, omdat die afhangt van de gedefinieerde action-mappings.

```

gMethod(?plugin, [#'initialize-release'],
  { initialize
    super initialize.
    ActionClasses := Dictionary new.
    ?addActionCode
    resultAction := TESTCompositeXMLAction new }) if
plugin(?plugin),
generateAddActionCode(?addActionCode).

```

Hierin wordt de eigenlijke code waarin de XMLActions worden toegevoegd aan de ActionClasses-lijst in de invoerplugin (?addActionCode) gegenereerd door de regel

```

generateAddActionCode(?code) if
  findall(
    <?action,?className>,
    and(   actionMapping(?action,?,?),
          actionClassName(?action,?className)),
    ?list),
  generateActionCode(?list, ?code).

```

Deze regel gaat een lijst maken van alle koppels van gedefinieerde instructie-types (het eerste lid van een actionmapping) en hun bijhorende XMLAction-klasse-namen, door middel van het ingebouwde SOUL-predikaat `findall` dat een lijst maakt van alle variabelen die voldoen aan de query die als tweede parameter wordt meegegeven. Deze lijst wordt doorgegeven aan `generateActionCode` die hem verwerkt en voor elk XMLAction-klasse erin de juiste code accumuleert.

```

generateActionCode(<<?action, ?className> >,
  { ActionClasses at: ?action put: ?class. }) if
  className(?class, ?className).

generateActionCode(<<?action, ?className> | ?rest>,
  { ActionClasses at: ?action put: ?class. ?restCode }) if
  className(?class, ?className),
  generateActionCode(?rest, ?restCode)

```

Deze regels werken gelijkaardig aan de `generateArgumentCode`-regels in sectie 6.2.2, alleen wordt hier gewerkt met lijsten van koppels, in plaats van lijsten met elementen. Voor elk van die koppels wordt een regel Smalltalk-code geschreven die opdraagt de XMLAction-klasse in het koppel in de ActionClasses associatielijst te zetten met als key het instructietype waarmee het gekoppeld is.

6.4 Conclusie

Allereerst hebben we een verzameling predikaten geschreven in SOUL voor de aanmaak van klassen, methoden en variabelen in Smalltalk. Alle gebruikte klassen worden hiermee aangemaakt.

De invoerplugin is een SAX-parser die werkt met XMLAction-klassen. Een XMLAction-klasse wordt gegenereerd voor elke actionmapping. Tijdens het lezen van de XML wordt voor elke instructie de correcte XMLAction geselecteerd, corresponderend met zijn instructietype. Deze verzamelt de argumenten in een lijst, doorgegeven in de SAX-methoden. Nadien schrijft deze de API-calls.

Hoofdstuk 7

Conclusie

Tenslotte zetten we alles nog eens op een rijtje. We kijken naar wat goed was en wat beter kan.

7.1 Samenvatting

Deze thesis is uitgevoerd in het kader van het VRT MPEG-project, dat handelt over standaardisatie van metadata over mediamateriaal, gebruik makend van recente en toekomstige MPEG-normen. De VRT bezit verschillende componenten die met elkaar communiceren en metadata uitwisselen door middel van XML. Die XML is echter niet altijd van hetzelfde formaat. We schrijven daarom plugins voor de componenten die de XML die binnenkomt vertaalt naar de API van die component. Omdat we dit niet telkens opnieuw willen doen, willen we dit proces automatisch laten verlopen en de plugin laten genereren. We kijken naar wat we voor dit generatieproces voor invoerplugins moeten modelleren en hoe we dat met elkaar laten werken.

Hiervoor bestuderen we eerst XML, een metataal voor markuptalen die gebruikt kunnen worden om data te structureren. DTDs of het nieuwere XML Schema leggen een XML-taal vast. XSLT kan een XML-document transformeren naar een andere vorm. Om XML te laten lezen door applicaties worden er interfaces aangeboden. De eerste is DOM, die een boomstructuur opbouwt van het XML-document. Wij gebruiken echter SAX, wat een event-gebaseerde verwerking aanbiedt, omwille van zijn grotere vrijheid.

We kiezen om de modellering waar de generatie van afhangt te doen in een declaratieve taal. Wij gebruiken de logische taal SOUL, die redeneert over Smalltalk en deze effectief kan manipuleren. Wij gebruiken SOUL voor code-generatie en blijven niet in de XML-familie met XSLT omdat deze laatste verre van expressief is.

In ons systeem gaan we als case een plugin genereren die XML met instructies vertaalt naar de API van de MP3-speler iTunes. Om de generatie mogelijk te maken mappen we de XML-instructies op API-functies van iTu-

nes. Dit wordt vastgelegd in zogenaamde actionmappings in SOUL. Aan de hand daarvan genereert ons systeem de nodige Smalltalk-classes.

Ons systeem werkt in drie lagen: de bovenste is degene waar het programmeerwerk in gebeurt en wordt uitgedrukt in SOUL. Daaronder is er de Smalltalk-laag die de plugin implementeert. Tenslotte is er de API-laag die de component aanspreekt, en die afhangt van welke component gebruikt wordt. Ook de implementatie-taal ligt niet vast, en zou vervangen kunnen worden zonder dat de bovenste laag van het systeem aangepast moet worden.

De implementatie van deze generatie gebeurt door XMLAction-classes. Deze worden aangemaakt voor elke actionmapping en opgeroepen door de plugin als hun corresponderende instructie in de XML wordt gelezen. De XMLAction verzamelt alle nodige argumenten en kan nadien de API-call uitschrijven.

7.2 Bijdrage

We hebben onderzocht hoe we generatie van een invoerplugin, die XML met instructies vertaalt naar de API van een component, mogelijk maken. Hiervoor vonden we dat het nodig is een model te maken van de zaken die de plugin specificeren.

Het specifieke aan onze invoerplugins is de XML die ingevoerd wordt en de API waarnaar vertaald wordt. We moeten dus de XML op de API mappen. Hiervoor hebben we de *actionmappings* uitgevonden (sectie 5.2.2) die dat vastleggen. Deze lieten ons toe uiteindelijk de plugin met een minimum van specificaties succesvol te laten genereren en werken.

We restricteren wel de XML zodat elke instructie met zijn argumenten voorkomt in een bepaald element, namelijk ‘**action**’, met het onderscheid tussen de verschillende types van instructies gegeven in het attribuut ‘**type**’. Dit is hier van minder belang omdat de XML toch eerst een transformatieproces dient te ondergaan in de interface-integrator van de VRT.

We hebben ook aangetoond dat voor zulke codegeneratie als wij doen een programmeertaal nodig is, omdat er algoritmen geschreven moeten worden voor alles behalve de meest simpele generatie. We bekeken ook de transformatietaal XSLT hiervoor (sectie 4.3), aangezien deze Turing-compleet is, maar deze bleek veel te weinig expressief.

7.3 Verder werk

Hoewel de hoofdzaak van het experiment gebeurd is, zijn er nog vele details die bijgeschaafd kunnen worden.

7.3.1 Uitvoer

Zowat de belangrijkste aanpassing voor echt gebruik van het systeem is de uitvoer die uiteindelijk uit de gegenereerde invoerplugin komt. Nu zijn dat uitgeschreven API-calls. Deze worden echter nog niet zo uitgevoerd, omdat de machine waarop het experiment werd ontwikkeld geen Mac was, en dus geen AppleScript kon verwerken. Het uitgevoerde scriptbestand zou dus nog manueel uitgevoerd moeten worden, zodat de instructies uiteindelijk aan het API van iTunes doorgegeven worden.

Wat ook een mogelijkheid is, is het aanspreken van de API te laten gebeuren in de Smalltalk code zelf, als deze een interface naar de gebruikte component ondersteunt. Hierbij laten we de `execute`-methode van de `XMLActions` geen API-code afdrukken, maar de ingebouwde interface aanspreken met de corresponderende functies, om op die manier de instructies door te geven. Dit zou uitgetest kunnen worden door ons generatiesysteem toe te passen op applicaties die onder Smalltalk Squeak draaien, zoals de webbrowser Scamper.

7.3.2 Flexibelere XML

Zoals net vermeld is er een restrictie op de invoer-XML: elke instructie wordt geëncapsuleerd in een `action`-element zodat de parser die als zodanig kan herkennen. Dit kan makkelijk aanpasbaar gemaakt worden, zodat je zelf kan kiezen welk element de instructies bevat. Hiervoor hoeft slechts één extra regel in SOUL bijgevoegd te worden.

```
actionElement(action)
```

Dit was echter voor ons niet belangrijk, omdat de XML sowieso een transformatie ondergaat in de interface-integrator en tijdens dat proces kunnen de instructies omgevormd worden naar het juiste element.

Wat op het moment nog niet mogelijk is, is dat de XML elementen die argumentwaarden bevatten dezelfde naam hebben maar andere attribuutwaarden. Dit gaf problemen bij bijvoorbeeld de instructie om een playlist van naam te veranderen:

```

<XMMS>
  <action type="renamePlaylist">
    <Sender>http://localhost:8080</Sender>
    <PlaylistName type="newPlaylist">
      Super Playlist 5
    </PlaylistName>
    <PlaylistName type="currentPlaylist">
      Superplaylist 5
    </PlaylistName>
  </action>
</XMMS>

```

De huidige playlist en de nieuwe zijn beide in het element ‘PlaylistName’ beschreven. Het onderlinge onderscheid zit in hun attribuut ‘type’ dat respectievelijk de waarden ‘newPlaylist’ en ‘currentPlaylist’ heeft. Nu echter wordt in de actionmapping enkel de elementnaam gegeven als specificatie voor de argumenten. Het is dus niet mogelijk dit op te schrijven voor een API-functie die een playlist hernoemt.

```
renamePlaylist(oldPlaylist, newPlaylist)
```

Want dan was de corresponderende actionmapping:

```
actionMapping(renamePlaylist, renamePlaylist,
              <PlaylistName, PlaylistName>)
```

Dit kan natuurlijk nooit werken, omdat de twee argumentnamen niet te onderscheiden zijn. Dit kan opgelost worden door de actionmappings uit te breiden, en niet enkel de elementnamen in argumentenlijst te schrijven, maar ook de lijst van nodige koppels van attribuutnamen en -waarden die elk element identificeren. Bijvoorbeeld

```
actionMapping(renamePlaylist, renamePlaylist,
              <<PlaylistName | <<type | new>>>>,
              <PlaylistName | <<type | new>>>>)
```

7.3.3 Flexibeler API

De API-syntax zou nog wat uitgebreid kunnen worden. Op het moment kunnen er enkel API-calls van de vorm

```
functie(arg1, arg2)
```

gegenereerd worden. De zaken zoals een komma tussen de argumenten, en een rond haakje ervoor en erna zijn vastgelegd in de regels die de `execute`-methode, waarin het uitschrijven van de API-calls gebeurt, voor de XMLAction-classes genereren.

```

generateMethodCode(?function,?args,
  { execute
    | stream |
    stream := WriteStream with: (?function asString) , '( '.
    ?argumentCode
    stream nextPut: $).
    ^ stream contents }) if
generateArgumentCode(?args,?argumentCode).

```

```

generateArgumentCode(<?el>,
  { stream nextPutAll: ''',
    (self argumentAt: ?el), '''. }).
generateArgumentCode(<?first|?rest>,
  { stream nextPutAll: ''',
    (self argumentAt: ?first), ''', '. ?restCode }) if
generateArgumentCode(?rest,?restCode)

```

Een aantal extra predikaten die modelleren wat de separator en dergelijke is lossen dit op. We zouden bijvoorbeeld aan het model regels kunnen toevoegen als

```

apiOpenFunction({}).
apiArgumentSeparator({,}).
apiCloseFunction({}).

```

In de genererende methoden zouden deze feiten dan opgezocht worden.

```

generateMethodCode(?function,?args,
  { execute
    | stream |
    stream := WriteStream with:
      (?function asString) , '?apiOpenFunction'.
    ?argumentCode
    stream nextPut: '?apiCloseFunction'.
    ^ stream contents }) if
generateArgumentCode(?args,?argumentCode),
apiOpenFunction(?apiOpenFunction),
apiCloseFunction(?apiCloseFunction).

generateArgumentCode(<?el>,
  { stream nextPutAll: ''',
    (self argumentAt: ?el), '''. }).

```

```

generateArgumentCode(<?first|?rest>,
  { stream nextPutAll: '''', (self argumentAt: ?first),
    '''?apiArgumentSeparator'. ?restCode }) if
generateArgumentCode(?rest,?restCode),
apiArgumentSeparator(?apiArgumentSeparator)

```

Het gebruik van de SAX-implementatie is ook wat te hard-gecodeerd op het moment. Bijvoorbeeld

```

saxMethod(TESTPlugin, [#content],
  { characters: aString
    ^ currentAction addArgument: currentElement asSymbol
      value: aString}).

```

Het systeem zou als je een andere SAX-implementatie zou willen gebruiken veel flexibeler kunnen zijn, door een abstractie te maken van de aanroepen van de SAX-methoden (er kan verschil zijn zoals bijvoorbeeld tussen ‘character: aString’ of ‘processCharacters: characters’). De bodies van de methoden blijven altijd hetzelfde, buiten de namen van formele parameters, wat in feite ook in de aanroep staat. Dit is op te lossen met enkele extra SOUL-regels waarbij de aanroepen en gebruikte parameters apart gemodelleerd worden, en later ingevoegd in de code van de methoden. Bijvoorbeeld als we de parameters extra modelleren in SOUL

```

characterParameter({aString})

```

dan kunnen we de methode iets algemener maken.

```

saxMethod(TESTPlugin, [#content], { characters: ?parameter
  ^ currentAction addArgument: currentElement asSymbol
    value: ?param}) if
characterParameter(?parameter).

```

Hetzelfde principe kan doorgetrokken worden voor de volledige header van de SAX-methoden.

Iets wat ook uitgebreid kan worden voor complexere instructies, is dat één instructie in de XML overeenkomt met meer dan één API-functie. Dit kan opgelost worden door in de actionmapping een lijst van API-functies mee te geven, met elk hun argumentenlijst. Laten we als voorbeeld veronderstellen dat de instructie om een playlist een andere naam te geven niet aanwezig is in de API. We lossen dit op door eerst de oude playlist te verwijderen en dan een nieuwe aan te maken met de gewenste naam, wat dus twee API-functies zijn.

```

actionMapping(renamePlaylist, <removePlaylist, newPlaylist>,
  <<OldPlaylistName>, <<NewPlaylistName>>)

```

7.3.4 Hybride specificatie

Een eerder omslachtige wijze wordt nu gebruikt om de ondersteunende klassen te programmeren. Deze worden namelijk opgeschreven in Smalltalk, maar onder de vorm van meerdere logische predikaten in SOUL (zie bijvoorbeeld figuur 6.2). Hierdoor worden de verschillende componenten van de klassen zoals methoden en variabelen van elkaar gescheiden, wat niet bevorderlijk is voor de overzichtelijkheid van hun code. Misschien is het handiger om een hybride vorm van programmeren te gebruiken en de ondersteunende klassen, dus al deze die onveranderd kunnen blijven bij verschillende generaties, in de implementatietaal te programmeren. Dit vermindert wel de eenvoud van de opzet van ons systeem dat alles in de declaratieve laag wordt geprogrammeerd.

7.3.5 Taalafhankelijkheid

In het algemeen geval is flexibiliteit verhogen ten opzichte van taalafhankelijkheid de grootste uitdaging van mogelijke uitbreidingen in deze thesis. In ons experiment hebben we ons toegespitst op het mogelijk maken van de generatie voor verschillende APIs, maar voor systemen die echt in gebruik worden genomen is het belangrijk dat de onderliggende talen onafhankelijk zijn van de sturende modellen en makkelijk vervangen kunnen worden als dit nodig is.

We genereren nu Applescript, maar voor hetzelfde geld genereren we Java-calls of zelfs Smalltalk-code zelf, of laten we rechtstreeks de acties uitvoeren in de implementatietaal, zoals beschreven hierboven in sectie 7.3.1. We hebben dit enkel kunnen uittesten op verschillende APIs, en enkel als ze voldeden aan de vorm die is opgelegd zoals boven beschreven (namelijk API-functies van de vorm `functie(arg1, arg2)`). Om taalafhankelijkheid te voorzien op dieper niveau, zoals vanaf de implementatielaag, zal het systeem uitgebreid moeten worden met een zorgvuldig opgebouwd, abstracter model van de implementatie van de plugin, zodat elke onderliggende taal met de nodige XML-verwerkingscapaciteiten kan gebruikt worden.

Bibliografie

- [1] XML 101. <http://xml101.com>.
- [2] Elizabeth Castro. *XML voor het World Wide Web*. Peachpit Press, 2001.
- [3] World Wide Web Consortium. <http://www.w3.org>.
- [4] Wolfgang de Meuter and Johan Brichau. *SOUL Manual*, 3.2 (draft) edition, 2001.
- [5] The Brighton University Resource Kit for Students (Prolog). <http://burks.bton.ac.uk/burks/language/prolog>.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. AddisonWesley, 1995.
- [7] Elliotte Rusty Harold. *The XML Bible*. Hungry Minds, 2nd edition, 2001.
- [8] YaX homepage. <http://squeaklet.com/Yax>.
- [9] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Expert Systems with Applications*, 2001.
- [10] SAX 2.0 official website. <http://www.saxproject.org>.
- [11] XML Cover Pages. <http://xml.coverpages.org>.
- [12] Squeak SmallTalk: Classes Reference. <http://mucow.com/SqueakClassesRef.html>.
- [13] Leon Sterlin and Ehud Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.
- [14] The Squeak Swiki. <http://minnow.cc.gatech.edu/squeak>.
- [15] The SOUL Logic Meta Programming tool. <http://prog.vub.ac.be/research/DMP/soul/soul2.html>.

- [16] Tom Tourwé, Tom Caljon, and Peter Soetens. D4.1 vrt metadataplugs. Technical report.
- [17] Kris De Volder. *Type Oriented Logic Meta Programming for Java*. PhD thesis, PROG.
- [18] VRT, PROG, IMEC, and ETRO. Mpeg-project: bijlage 1.
- [19] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA98*, pages 112–124, 1998.
- [20] O'Reilly XML.com. <http://www.xml.com>.