Analyse op .Net software

Auteur:	Casper de Lange
Stud. Nr.:	9745467
E-mail:	clange@cs.uu.nl

Een woord van dank gaat uit naar de volgende personen:
- Gert Florijn en Jurriaan Hage voor hun begeleiding tijdens de stage en het schrijven van dit document.
- Jaco Peeman voor de samenwerking gedurende de stage.

Inhoudsopgave

1. Inleiding

Dit document beschrijft de afstudeerstage die ik gelopen heb in het kader van mijn afstuderen aan de Universiteit Utrecht. De stage is gelopen bij het Software Engineering Research Centre (SERC) in de periode november 2002 tot mei 2003.

1.1 Het SERC

SERC is in 1987 opgericht als onderdeel van Stimulerings Projectteam Informaticaonderzoek (SPIN). Dit programma werd door overheid en bedrijfsleven gesponsord en had als doel het verhogen van het kennisniveau op het gebied van Software Engineering in Nederland. Sinds 1992 heeft SERC als zelfstandig bedrijf gefunctioneerd, maar sinds kort is SERC een onderdeel van CIBIT Adviseurs | Opleiders.

SERC adviseert organisaties over het verbeteren van software en softwareontwikkeling, het introduceren van nieuwe technieken en helpt bij het ontwerpen en realiseren van nieuwe systemen (tot en met het prototype).

1.2 De afstudeerstage

De stage is gezamenlijk gelopen met Jaco Peeman. Jaco en ik wilden rond dezelfde tijd beginnen met afstuderen en het leek ons leuk dit op een gezamenlijk onderwerp te doen. Als richting voor onze afstudeerstage hadden we beide besloten iets met object-georiënteerde software te doen. Onze interesse ging vooral uit naar softwareontwikkeling en refactoring van bestaande software.

Als uitgangspunt voor onze stage hebben we het programma RevJava genomen. RevJava is een software-analyse tool die ontwikkeld is door Gert Florijn. Met RevJava is het mogelijk om in Java ontwikkelde software te analyseren met behulp van in RevJava gecodeerde regels. De onderwerpen en opzet van de stage worden hieronder globaal besproken. In hoofdstuk 2 wordt de precieze aard van de stage uitgebreid behandeld en toegelicht.

De stage hebben we ingedeeld in een gezamenlijk en een individueel gedeelte. Het gezamenlijke deel was een voorbereiding op het individuele gedeelte en omvatte de volgende onderwerpen:

- Het onderzoeken van RevJava.
- Een oriënterend onderzoek naar mogelijke uitbreidingen voor RevJava.
- Een onderzoek naar Microsoft .Net.
- Het opstellen van een nieuw metamodel voor RevJava.

Bovenstaande onderwerpen worden respectievelijk beschreven in de hoofdstukken 3 t/m 7 van deze scriptie. Het individuele gedeelte bestond uit het maken van een parser en een reader, zodat software die ontwikkeld was met Microsoft .Net ingelezen kon worden in RevJava. Dit wordt behandeld in de hoofdstukken 8 en 9.

In hoofdstuk 10 wordt er aan de hand van voorbeelden getoond hoe RevJava werkt na de aanpassingen die we gemaakt hebben. Hoofdstuk 11 is een evaluatie van de stage.

2. De stageopdracht

De stageopdracht is onder te verdelen in twee delen: een gezamenlijk deel met Jaco Peeman en een individueel gedeelte. Het individuele gedeelte borduurt voort op het werk dat in het gezamenlijke gedeelte gedaan is.

2.1 Het gezamenlijke deel

2.1.1 Oriënterend onderzoek naar uitbreidingen op RevJava

We zijn begonnen met een onderzoek naar wat er precies met RevJava gedaan kon gaan worden. We hadden een aantal opties welke onderzocht moesten worden, zodat duidelijk zou worden in welke mate ze interessant waren voor RevJava. Het doel was inzicht te krijgen in de mogelijkheden voor uitbreiding van RevJava en dit in een soort handleiding weer te geven. De te bestuderen onderwerpen waren:

1. Microsoft .Net: we moesten uitzoeken wat .Net precies was hoe het werkte en of het nuttig zou kunnen zijn dit in RevJava op te nemen.
2. Extra analyse mogelijkheden aan RevJava toevoegen: hiervoor hebben we metacompilatie en SOUL aan een onderzoek onderworpen, omdat we mogelijk de analyses die daarin toegepast worden ook konden gebruiken. De mogelijk toe te voegen analyses waren:
    a. Type analyse
    b. Alias analyse
3. Het Famix model: Het metamodel van RevJava was door Gert Florijn rond dezelfde tijd ontwikkeld als het Famix model. Het was interessant om te zien wat de verschillen tussen de twee waren. Tevens kon het ons een hint geven hoe wij het metamodel zouden moeten aanpassen mocht dat nodig zijn voor een vervolgopdracht.

Nadat we het eerste onderzoek hadden afgerond hebben we gekozen om .Net aan RevJava toe te voegen. Microsoft .Net leek ons een techniek die in de toekomst een belangrijk aandeel zou gaan hebben in de ontwikkeling van software. Toevoeging van Microsoft .Net aan RevJava leek ons waardevol voor het programma.

De opdracht werd nu RevJava zo aan te passen dat Microsoft .Net ingelezen en geanalyseerd kan worden. Deze opdracht werd gesplitst in een gezamenlijk en een individueel gedeelte. Besloten werd het opstellen van het generieke metamodel gezamenlijk te doen, zodat we allebei goed inzicht hadden in het metamodel. Hierna zijn we individueel verder gegaan en hebben het metamodel als soort 'breekpunt' beschouwd: we waren aan een ander gedeelte van het programma bezig met als overlap het gebruik van het metamodel.

2.1.2 Het opstellen van een generiek metamodel

Het doel van deze opdracht was tot een metamodel te komen, waarin zowel Java als MSIL ingelezen zou kunnen worden. We moesten er echter rekening mee houden dat er op een later tijdstip nog andere talen in het model ingelezen konden gaan worden. Het model moest zo worden opgesteld dat de analyse plaats kon vinden op het generieke niveau van het model.

De opdracht was onderverdeeld in een aantal onderdelen:

1. De MSIL bestuderen, zodat we wisten hoe de MSIL opgebouwd was. We konden toen de verschillen met Java gemakkelijk bepalen.
2. Een metamodel ontwerpen waarin beide talen ingelezen zouden kunnen worden.
3. Implementatie van het ontworpen metamodel.

2.2 Het individuele gedeelte

Voor het individuele gedeelte waren er drie mogelijke onderwerpen:

1. Het inlezen en parsen van de MSIL
2. Het ordenen en aanpassen van de critics
3. Het mogelijk maken van extra analyses (b.v. flow analyse)

We hebben gekozen om de eerste twee opdrachten uit te gaan voeren, aangezien deze dicht op het door ons ontworpen metamodel zaten.

Een uitgebreide specificatie van deze twee opdrachten wordt in hoofdstuk 8 gegeven, omdat deze opdrachten voortbouwen op de resultaten van het gezamenlijke deel.

3. RevJava *

3.1 Wat is RevJava

Wanneer men grote complexe systemen aan het ontwikkelen is, of met meerdere programmeurs tegelijk bezig is aan een systeem, is het belangrijk een goede architectuur voor het systeem opgesteld te hebben. Deze bepaalt de componenten van het systeem en waar welke verantwoordelijkheden binnen het systeem liggen. Daarnaast helpt de architectuur bij het overzichtelijk houden van het systeem en dient het als guide-line voor de constructie en evolutie van het systeem.

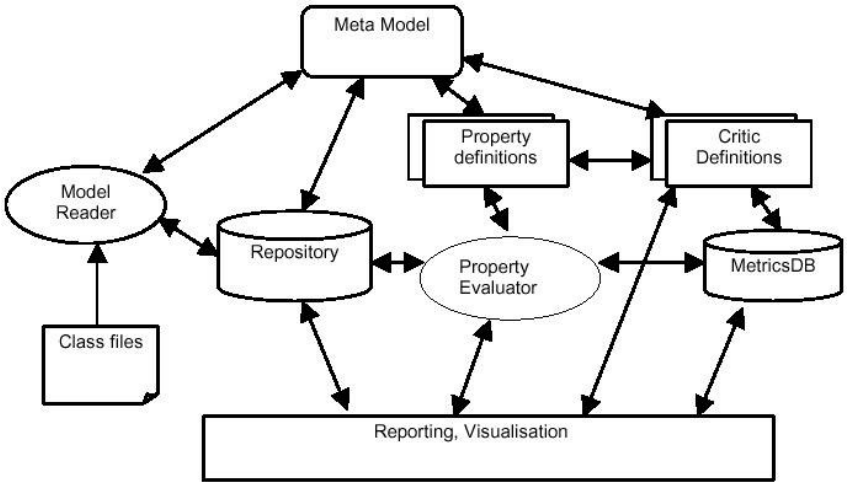
Wanneer men besluit een architectuur voor een systeem te ontwerpen is het erg belangrijk dat de software ook echt conform de architectuur gebouwd wordt. Het is bijvoorbeeld belangrijk dat de verantwoordelijkheden op de juiste plaatsen aanwezig zijn, de hiërarchieën overeenstemmen en andere restricties die gelden volgens de architectuur gerepresenteerd zijn in de software. Wanneer dit niet het geval is gedraagt het systeem zich niet volgens de architectuur en kan men bijvoorbeeld bij het aanpassen of uitbreiden van het systeem niet uitgaan van de architectuur. De architectuur is in dat geval waardeloos.

Een opgestelde architectuur is niet statisch. Tijdens de ontwikkeling en onderhoud van een systeem kan de architectuur ervan zelf ook aangepast worden, omdat er problemen ontdekt worden waar in de architectuur geen rekening mee gehouden was. Dit soort veranderingen brengen met zich mee dat (delen van) het systeem niet meer conform de architectuur zijn.

Om te zorgen dat het systeem werkelijk conform de architectuur is, is het belangrijk regelmatig reviews op de software te doen tijdens de ontwikkeling daarvan. Deze reviews kosten echter veel tijd, zeker wanneer de reviews gedaan worden door mensen die niet geholpen hebben het systeem te ontwikkelen.
RevJava is een tool dat is ontwikkeld om reviews op een snelle en gemakkelijke manier mogelijk te maken, tijdens het ontwikkelen van een systeem of als controle achteraf.

3.2 Wat doet RevJava

RevJava analyseert en bekritiseert bestaande object-georiënteerde software door gebruik te maken van architectuur regels. Deze regels zijn in het programma gecodeerd en er wordt bekeken door RevJava of de software deze regels overtreedt. RevJava werkt op gecompileerde Java code (de `.class` files).

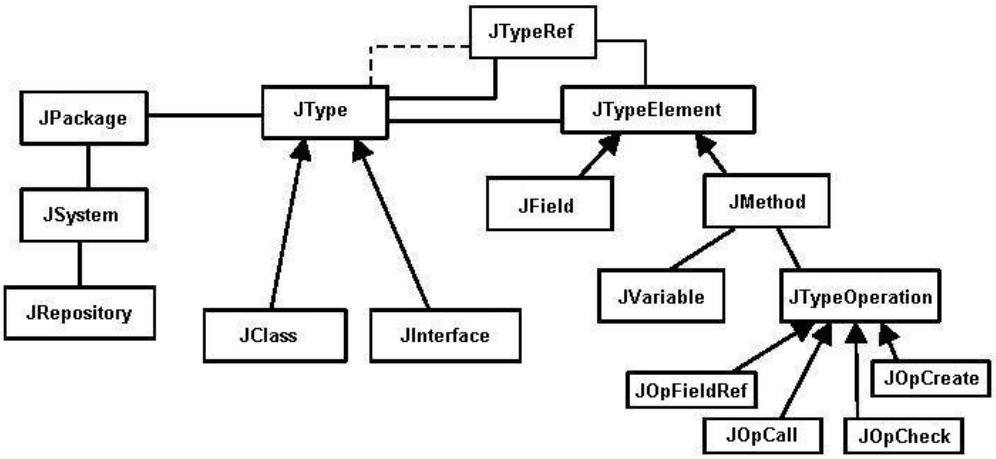


* gebaseerd op RevJava, zoals deze was bij release 0.8.5

Voor dit hoofdstuk is gebruik gemaakt van de literatuur genoemd bij [1] [2] [3]

Figuur 3.2.1: outline van RevJava

De aangegeven classfiles worden geparsed en de informatie die hier uit komt wordt door een reader in een metamodel gezet, wat opgeslagen is in de Repository. Dit metamodel definieert elementen welke in Java software voorkomen, zoals bijvoorbeeld 'Package', 'Class' of 'Interface'. Binnen het metamodel gelden de normale relaties zoals die binnen Java software normaal ook gelden (een class bevat methodes, een methode bevat lokale variabelen e.d.).



Figuur 3.2.2: de 'bevat-relaties' en inheritance in het metamodel

Op ieder model element worden zogenaamde Properties berekend. Bij een methode wordt bijvoorbeeld een set bijgehouden door welke methodes de methode aangeroepen wordt. Sommige van deze properties kunnen als Metric geëxporteerd worden. Een voorbeeld van een Metric is het aantal regels code in een methode bijhouden. Deze houdt bij wat het aantal regels code in een methode is voor alle methodes die in de Repository geladen zijn. Met deze informatie kan dan het gemiddeld aantal regels in een methode in het geladen systeem berekend worden.

De Critics vormen het gedeelte van RevJava wat de architectuur regels checkt. Critics kunnen gebruik maken van de afgeleide Properties van de elementen om te kijken of er (gecodeerde) architectuur regels overtreden worden. Via een user-interface wordt de gebruiker daarvan op de hoogte gesteld en kan hij gepaste actie ondernemen.

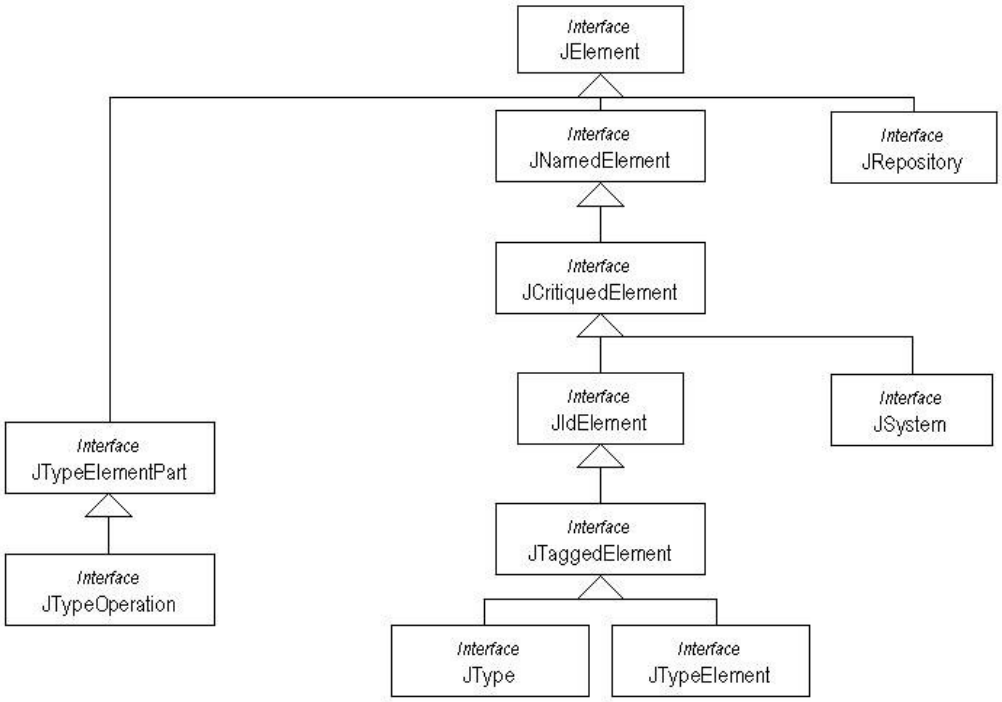
3.3 Het metamodel

Het metamodel vormt de core van RevJava. De elementen van een ingelezen programma worden in het metamodel opgeslagen, waarna er met behulp van cross-referencing een object graaf wordt opgebouwd. Als deze object graaf opgebouwd is kunnen de critics en metrics op de graaf berekend gaan worden.

Tijdens de cross-referencing worden referenties naar types gecheckt. Wanneer zo'n referentie een pointer naat een type betreft dat geladen is, verwijst het TypeRef object daarnaar. Enkele voorbeelden van wat er bij cross-referencing bepaald wordt:

- Inheritance: supertypes en subtypes worden bepaald, evenals interface-implementaties. Deze worden naar beide kanten (upwards en downwards) bijgehouden.
- Methode aanroepen worden gebonden aan de methode declaratie waarnaar ze verwijzen. Tevens worden methodes gebonden aan de methodes die ze zelf herdefiniëren, en aan methodes waardoor ze geherdefinieerd worden.

Het metamodel is enigszins Java georiënteerd van opzet, maar het abstracte deel van het model kan zonder al te veel aanpassingen geschikt zijn voor andere object-georiënteerde programmeertalen. Dit gedeelte van het metamodel wordt hier 'abstract' genoemd, omdat de elementen erin geen elementen zijn die daadwerkelijk in Java voorkomen. Hieronder staat het abstracte deel van het model:

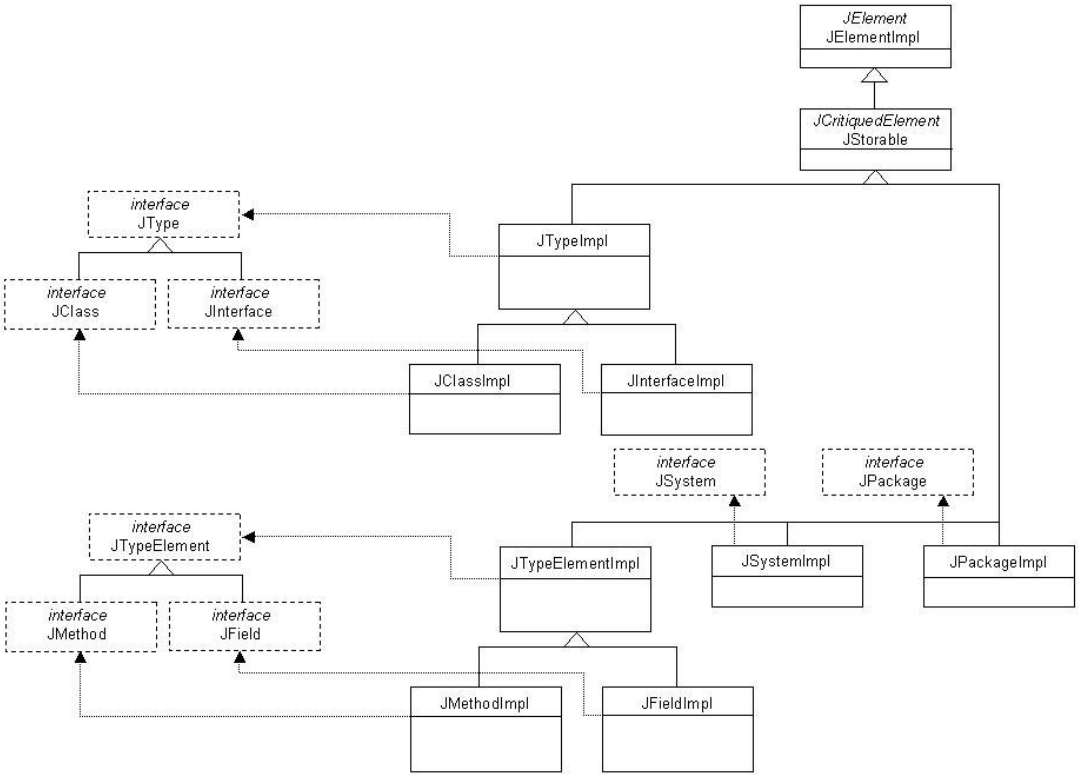


Figuur 3.3.1: het abstracte deel van het RevJava metamodel

De betekenis van de elementen uit het abstracte deel van het model is als volgt:

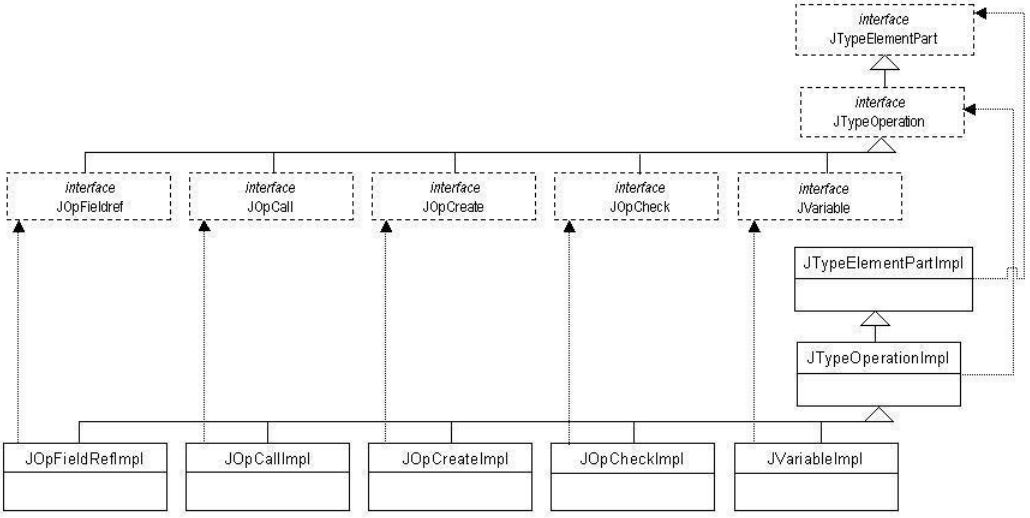
- JElement: dit is het root-type voor alle model elementen.
- JNamedElement: elementen die een naam hebben, op te vragen via getName().
- JCritiquedElement: elementen die critics op zich gedefinieerd kunnen hebben.
- JIdElement: elementen met een unieke id, welke aangemaakt wordt door de JRepository.
- JTaggedElement: elementen die modifiers (tags) op zich gedefinieerd kunnen hebben, zoals 'public', 'private' en 'static', opgeslagen in een JTags.
- JRepository: heeft referenties naar de systemen die geladen zijn en alle types.
- JSystem: heeft referenties naar de packages van het systeem waarop gewerkt wordt.
- JType: classes en interfaces zijn instanties van JType.
- JTypeElement: features van JTypes, zoals methodes en fields.
- JTypeElementPart: operaties en variabelen in een JTypeElement.
- JTypeOperation: operaties die op types werken, zoals manipulatie van een field, of creatie van een nieuw type.

Het concrete deel (dat wil zeggen: de taalelementen als classes en methodes enzovoorts) vormen de bladeren van het metamodel. Hieronder staat een deel van de implementatie van het concrete gedeelte van het metamodel:



Figuur 3.3.2: het (gedeeltelijke) RevJava implementatie metamodel

Het getoonde implementatiemodel beslaat slechts een gedeelte van het gehele model, maar er is goed te zien hoe de taalelementen terug zijn te vinden in het metamodel. Wat betreft JTypeElementPart en JTypeOperation ziet het model er als volgt uit:



Figuur 3.3.3: JTypeElementPart en JTypeOperation implementatiemodel

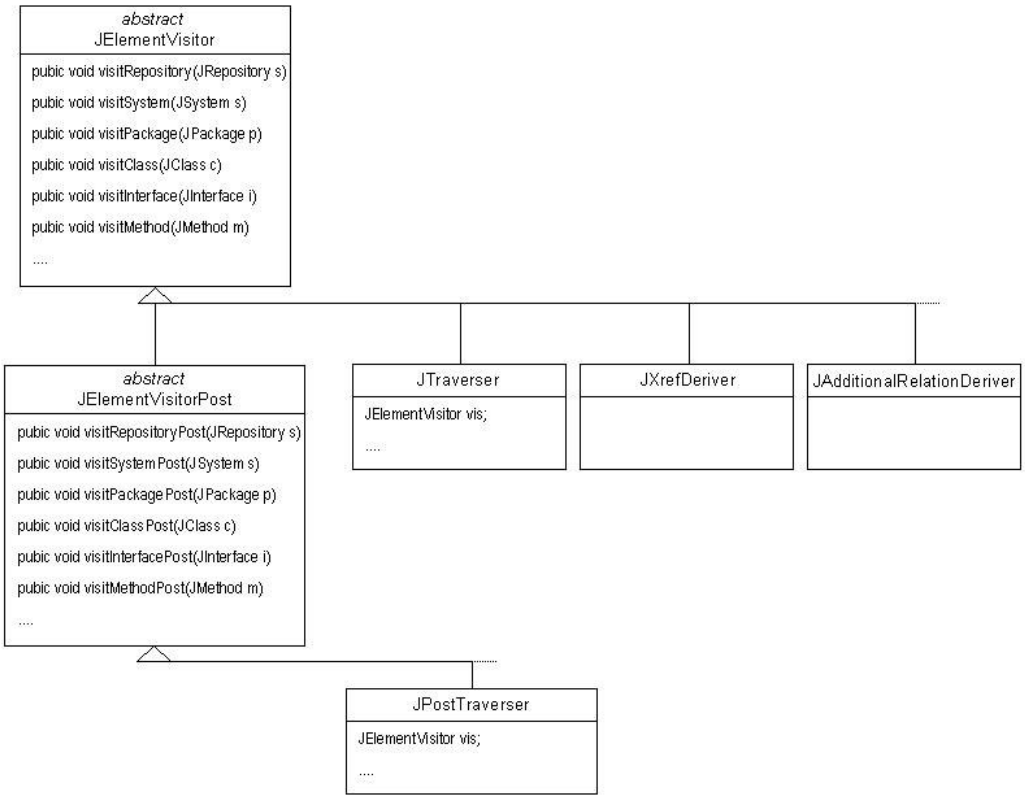
In de hoofdstukken 6 en 7 wordt er dieper ingegaan op het metamodel, de implementatie ervan en de betekenis van de specifieke elementen ervan.

3.4 Visitors

Het opbouwen van de structuur en het afleiden van de properties gebeurt door middel van het zo genaamde visitor-pattern. Dit houdt in dat een bepaalde class (de visitor) de model elementen 'bezoekt' en berekeningen uitvoert.

In RevJava zijn er twee basis visitors aanwezig: JElementVisitor en JElementVisitorPost. De eerste wordt gebruikt om de elementen één keer te bezoeken, namelijk in 'depth-first' volgorde (repository – system – package – type enzovoorts). De JElementVisitorPost bezoekt de elementen ook op een 'depth-first' manier, maar doet dit twee keer: per ouder één keer voordat de kinderen aangeroepen worden en één keer erna (de *Post methoden). JElementVisitor en JElementVisitorPost zijn beiden abstract gedefinieerd, zodat ze gebruikt kunnen worden als superclass voor het definiëren van een eigen visitor.

Naast de visitors die berekeningen op de elementen doen, zijn er ook visitors die bepalen dat de modelelementen op een speciale manier afgelopen moeten worden. Deze visitors noemen we een traversers. Een traverser krijgt een extra JElementVisitor mee welke op aangeven van de traverser bij ieder element berekeningen uitvoert.



Figuur 3.4.1: het visitor model

De visitors zijn binnen RevJava verdeeld over twee packages:

- 'nl.serc.revs.core.model.visitor': hierin staan de traversers en de standaard visitors gedefinieerd..
- 'nl.serc.revs.core.model.xrefs': hierin staan visitors welke gebruikt worden bij de cross-referencing: JXrefDeriver en de JAdditionalRelationDeriver.

3.5 De packages van RevJava

RevJava is opgebouwd via een hiërarchische package-structuur. De structuur wordt in deze paragraaf behandeld, omdat er in het vervolg van deze scriptie van uit wordt gegaan dat deze structuur bekend is.

Het toplevel package is ‘nl.serc.revs’. Dit package bevat de volgende sub-packages:

- core: bevat de classes die het metamodel implementeren, classes voor de metrics en critics en het laden van de classfiles.
- desktop: bevat de classes voor de user interfaces en de belangrijkste functionaliteit die daarachter hangt.
- rules: bevat de definities voor de properties en metrics in RevJava. Deze classes zijn gebaseerd op de definities in het package ‘nl.serc.revs.core’ voor properties en metrics.
- extensions: bevat classes die een extensie vormen op RevJava, zoals het exporteren van de ingeladen informatie naar XML.

Het package ‘nl.serc.revs.core’ is voor ons het belangrijkste package, aangezien we daarin de veranderingen aan het metamodel in moeten doorvoeren. De andere packages hebben we niet of nauwelijks nodig en deze zullen hier niet aan de orde komen, maar pas wanneer dit relevant is voor het begrip van de tekst.

Het package ‘nl.serc.revs.core’ bevat de volgende sub-packages:

- critics: bevat classes voor het beheren, verzamelen en definiëren van critics.
- logging: bevat classes voor het loggen van informatie.
- metrics: bevat classes voor het beheren en definiëren van critics.
- model: bevat de definities van het metamodel. De elementen zijn allen gesplitst in een interface een een implementatie-class, welke bevat is in het sub-package ‘nl.serc.revs.core.model.impl’. Daarnaast bevat dit package twee sub-packages waarin classes voor visitors en het uitvoeren van cross-referencing gedefinieerd kunnen worden: ‘nl.serc.revs.core.model.visitor’ en ‘nl.serc.revs.core.model.xrefs’.
- plugin: bevat classes voor het beheren van plugins en een aantal standaard plugins.
- props: definieert classes voor het beheren, verzamelen en definiëren van properties.
- reader: bevat de classes voor het inlezen van de classfiles. Dit package bevat interfaces, in het sub-package ‘nl.serc.revs.core.reader.javaclasses’ staan de implementatie-classes.
- util: definieert sets, tabellen en andere opslagmechanismen om data in op te slaan.

4. Oriënterend onderzoek naar RevJava

Tijdens de eerste maand van de stage is er besloten een onderzoek te doen naar mogelijke uitbreidingen aan RevJava. Er vijf onderwerpen die we bekeken hebben:

1. SOUL
2. Metacompilatie
3. Famix model

De bedoeling van deze drie onderwerpen was om te kijken of de technieken die gebruikt werden voor RevJava interessant zouden kunnen zijn. Het Famix model zou misschien kunnen helpen wanneer we het metamodel van RevJava aan zouden moeten passen.

Verder zijn de volgende onderwerpen onderzocht om als extra analyses in RevJava toegevoegd te worden:

4. Type analyse
5. Alias analyse

Bij het bestuderen van de onderwerpen moest er rekening worden gehouden met de vereiste performance van RevJava. RevJava is ontwikkeld als evaluatie tool tijdens het ontwikkelen van een systeem, daarom is het nodig dat het inlezen en analyseren weinig tijd kost.

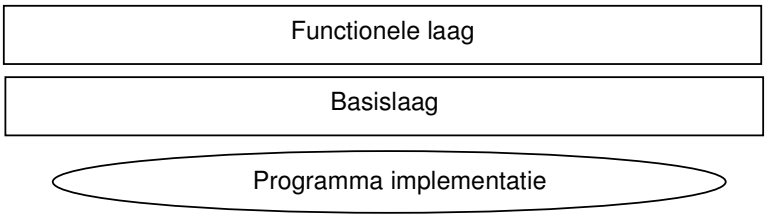
Van dit onderzoek is een verslag geschreven met als filenaam:
'orienterend_onderzoek_revjava.pdf'.
De verdere tekst in dit hoofdstuk is een abstract uit dit verslag. Geïnteresseerden kunnen het gehele verslag downloaden op: www.asskicker.nl/articles.

4.1 SOUL (Smalltalk Open Unification Language)

Wanneer RevJava gebruikt wordt om te kijken of software nog aan de opgestelde architectuur voldoet, moet die architectuur bekend zijn bij de gebruiker. Vanuit RevJava kan er slechts impliciet worden afgeleid wat de architectuur ongeveer kan zijn. Het zou handig zijn als de link tussen de architectuur en de implementatie van een stuk software explicieter aanwezig zou zijn. Voorbeelden van designinformatie die nuttig kunnen zijn om afgeleid te worden:

- Wat voor types een bepaalde Vector bevat
- Is een class onderdeel van een composite pattern

Een bekend project op dit gebied is SOUL [16] [17] [18]. SOUL legt de link tussen design en implementatie met behulp van verschillende lagen. Met behulp van Queries en Rules kunnen bepaalde constraints dan opgevraagd en geanalyseerd worden. SOUL is geïmplementeerd voor Smalltalk, maar het idee is om het ook voor Java en C++ geschikt te maken.

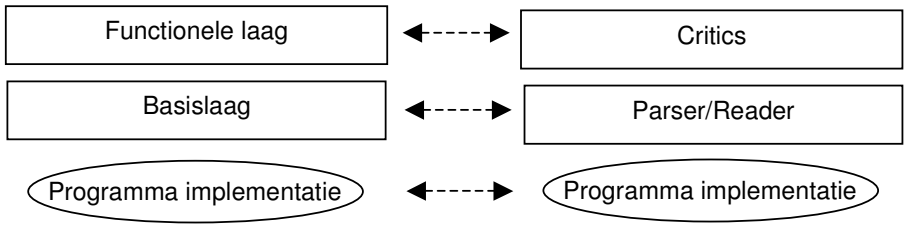


Figuur 4.1.1: Het SOUL lagen model

De basislaag is programmeertaal afhankelijk en doet basisoperaties op de programmacode, zoals alle methoden van een class ophalen. De functionele laag maakt gebruik van de

gegevens in de basislaag om Queries en Rules op te stellen. Deze functionele laag is niet afhankelijk van de taal waarin het programma geïmplementeerd is. De Rules kunnen door de programmeur worden opgesteld naar de eisen waaraan het programma moet voldoen en kunnen gebruikt worden om een design te checken.

Dit model vertoont enige overeenkomst met de werking van RevJava wanneer het RevJava metamodel als generiek model wordt gezien:



Figuur 4.1.2: SOUL (links) versus RevJava (rechts)

De parser en de reader in RevJava zijn programmeertaal specifiek, maar als de informatie vanuit de software ingelezen is werken de critics op het (generieke) metamodel.

4.1.1 RevJava en design informatie

Voor RevJava is het gezien de vereiste performance niet haalbaar om over de gehele programma structuur expliciete design informatie te genereren. De gegevens die nodig zijn hiervoor zijn wel grotendeels aanwezig na het inlezen van de class files, maar het kost te veel tijd om de gegevens uit de structuur te halen en ze te interpreteren. In enkele gevallen zou er zelfs flow-analyse plaats moeten vinden. Toch zou het interessant zijn als RevJava bijvoorbeeld de hieronder staande vragen zou kunnen beantwoorden:

- Is class c onderdeel van een composite pattern?
- Beheert class c objecten, of wordt class c beheerd door een andere class?
- Is class c toegankelijk via een interface, welke classes implementeren die interface ook?

Een optie is om de gebruiker de mogelijkheid te geven dit soort analyses te doen op een subset van de opgebouwde structuur. De gebruiker kan deze subset zelf aangeven, bijvoorbeeld een enkele class of een class en zijn subclasses. Hiervoor zal mogelijk extra informatie uit de class files moeten komen, maar het kost niet veel extra tijd dit te vergaren. Het voordeel van deze methode is dat de performance van RevJava weinig verandert, terwijl er op aangeven van de gebruiker extra analyses plaats kunnen vinden.

4.2. Metacompilatie

Metacompilatie is een techniek die ontwikkeld is om de correctheid en performance van grote systemen te checken aan de hand van opgestelde regels. Het checken van deze regels gebeurt tijdens de compilatie van het systeem, omdat een compiler alle sourcecode naar programma code mapt en het gemakkelijk is een enkele regel op het gehele systeem te laten uitvoeren. Het schrijven van vele testcases, of het opstellen van een complexe formele verificatie van het gehele systeem kan hierdoor overbodig worden.

Het probleem van het willen testen van regels tijdens het compileren is, dat de compiler niet de tools bezit om deze regels op te stellen en te checken. Metacompilatie maakt het mogelijk voor de programmeur compiler extensies te schrijven, waarin de specifieke eigenschappen van het te checken systeem tot uitdrukking kunnen worden gebracht. Door het opstellen van regels over deze eigenschappen kan het systeem gecheckt worden op fouten.

4.2.1. Het MC Framework

Wij hebben ons voornamelijk geconcentreerd op het werk van een groep onderzoekers van de universiteit van Stanford, welke onder leiding van Dawson Engler een Metacompilatie framework hebben ontworpen [19]. Dit MC Framework bestaat uit 2 delen:

- Een taal voor het definiëren van debugging analyses genaamd Metal.
- Een compiler die regels, gedefinieerd in Metal, uit kan voeren, de xgcc.

Het framework is zo opgezet dat het voor programmeurs makkelijk is de compiler te voorzien van nieuwe extensies, geschreven in Metal. Deze extensies worden dynamisch aan de compiler gelinkt en door het gehele programma heen doorgevoerd. De extensies maken gebruik van patterns om operaties die voor hen van belang zijn te herkennen. Wanneer de sourcecode aan één van de patterns voldoet wordt er gekeken of de opgestelde regel overtreden wordt.

Het MC Framework heeft zijn nut bewezen in het checken van grote complexe systemen, waaronder Linux. Hierin zijn een groot aantal fouten gevonden met extensies die vaak nog geen honderd regels code besloegen en geprogrammeerd waren door mensen met een beperkte kennis van het te checken systeem [20] [21].

4.2.2. Metacompilatie versus RevJava

Alhoewel RevJava en het MC Framework beiden gemaakt zijn om complexe systemen te analyseren moet er een duidelijk onderscheid gemaakt worden tussen de twee:

- Het MC Framework, of eigenlijk metacompilatie in zijn algemeenheid, is ontwikkeld om bugs in systemen te vinden. Typische voorbeelden analyses van metacompilatie zijn: ‘wordt een geheugenadres vrijgegeven na een lock’ of ‘wordt een pointer gecheckt op validiteit voordat hij gebruikt wordt’. Het opsporen van dit soort bugs is een moeilijk en tijdrovend karwei wanneer het zou moeten gebeuren met tests of een formele verificatie, maar wanneer er een bug bestaat kan deze het gehele systeem laten crashen.
- RevJava spoort geen directe bugs op, maar analyseert de gecompileerde sourcecode op type niveau. Dit houdt in dat er door de analyses in RevJava geen ‘uitspraken’ worden gedaan over de werking van het programma, maar over de uit de programmatuur afgeleide architectuur ervan.

Het idee van het schrijven van extensies echter is een techniek die in RevJava in de vorm van losse plugins ook zijn nut kan bewijzen. Deze extensies zouden bij het opstarten geladen kunnen worden en met het opbouwen van de boom en bij de analyse meegenomen kunnen worden. Voor deze extensies zou, net als bij metacompilatie, een tool gevonden moeten worden die het opstellen van die plugins op een gemakkelijke manier mogelijk maakt.

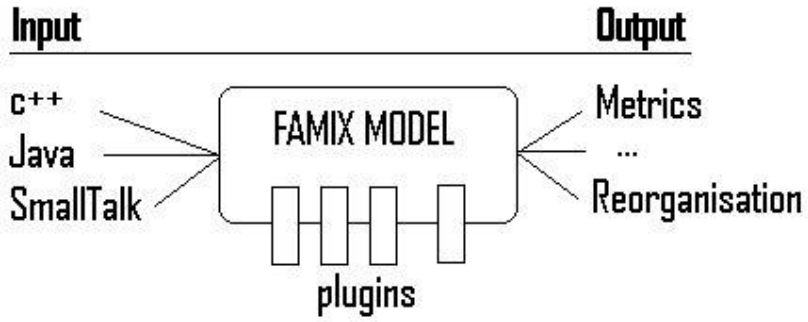
4.3. Het Famix model

Het Famix model [4] [5] is ontwikkeld in het kader van het FAMOOS project, wat is gestart in 1996 en waaraan verschillende grote bedrijven meewerkten. Het FAMOOS project had tot doel bestaande software te verbeteren door het flexibeler van opzet te maken, zodat software gemakkelijk aan te passen is wanneer er veranderingen aan gemaakt moeten worden. Object-georiënteerde software wordt gezien als dé manier om flexibele software te ontwikkelen en daarom richtte het project zich op de re-engineering van bestaande object-georiënteerde systemen.

Dit re-engineering wordt gezien als een evolutionair proces, dat onder te verdelen is in de volgende zes stappen:

1. Requirements Analysis: het doel van de re-engineering vaststellen.
2. Model Capture: het begrijpen en documenteren van de software.
3. Problem Detection: het bepalen van problemen qua kwaliteit en flexibiliteit van de software.
4. Problem Resolution: een oplossing zoeken voor de problemen via een andere architectuur van de software.
5. Reorganisation: het aanpassen van de software voor een nieuwe release.
6. Change Propagation: de aanpassingen voor alle afhankelijke software doorvoeren.

Ter ondersteuning van deze stappen en omdat software in verschillende programmeertalen geschreven kan zijn, is het FAMOOS Information Exchange (Famix) model ontwikkeld. Het Famix model is een generiek model voor object-georiënteerde software, maar het is mogelijk het model met taal specifieke en tool plugins uit te breiden.



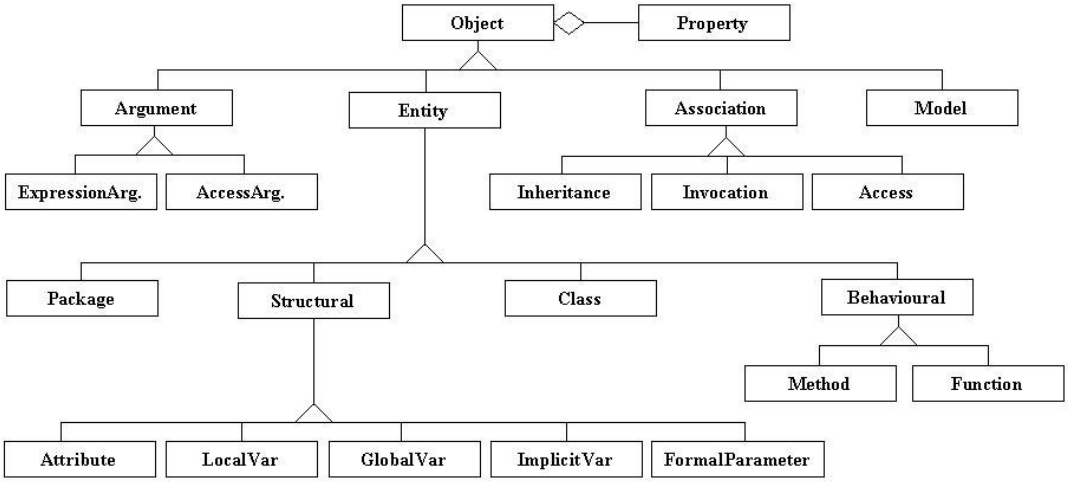
Figuur 4.3.1: het Famix concept model

Het model verschaft in hoofdzaak de basis voor het analyseren van object-georiënteerde software in het algemeen, maar wanneer men taal specifieke analyses wil doen kan men het model uitbreiden met plugins voor die taal. De informatie die opgeslagen kan worden in het model blijft beperkt tot direct uit de code afleidbare gegevens. Interpretatie van de gegevens wordt gezien als de verantwoordelijkheid van de plugins.

4.3.1 Elementen van het Famix model

Het Famix model is onder te verdelen is twee gedeelten: het core model, dat de elementen van object-georiënteerde talen beschrijft (classes, methodes enzovoorts) en een abstract gedeelte dat zorgt dat het model vanuit een generiek niveau opgebouwd is en uitgebreid kan worden.

Het gehele model ziet er als volgt uit:



Figuur 4.3.2: het complete Famix model

Het abstracte gedeelte wordt gevormd door de volgende elementen:

- Object: het root element van het model.
- Entity: een object dat voorzien is van een naam en een gekwantificeerde unieke naam, een class ‘aClass’ in package ‘aPackage’ wordt gekwantificeerd bijvoorbeeld: ‘aPackage::Aclass’.
- Association: geeft een relatie aan tussen twee elementen:
  - Access: een methodemaakt gebruik van een Attribute.
  - Invocation: een methode aanroep naar een andere methode.
  - Inheritance: een ouder-kind relatie tussen classes.
- Property: een eigenschap van een element.
- Argument: een parameter die meegegeven wordt aan bijvoorbeeld een methode.

De eerste vier elementen zijn geschikt om er taal specifieke of tool plugins op te definiëren. Hiervoor kan men taal specifieke objecten in het model introduceren of eigen properties definiëren op objecten.

Het element Model is een singleton en bevat informatie over het systeem dat gemodelleerd wordt, zoals bijvoorbeeld de taal en de naam van het systeem waar de informatie uit afkomstig is.
Behavioural Entities (methodes en functies) geven gedrag aan Entities (packages en classes). Een Structural Entity is een Attribute (field) of een variabele van één of andere vorm:

- Local variable: lokale variabelen in een methode.
- Global variable: in een package gedefinieerde variabele, geldig zolang het systeem runt en globaal toegankelijk.
- Implicit variable: context afhankelijke referentie naar het geheugen zoals ‘this’ in Java.
- Formal parameter: wat een methode verwacht als argument.

4.3.2 Het Famix model versus het RevJava metamodel

Het Famix model en het RevJava metamodel zijn rond dezelfde tijd onafhankelijk van elkaar ontwikkeld met als doel een model te vormen voor object-georiënteerde software, al was bij de ontwikkeling van RevJava al duidelijk dat het programma in eerste instantie slechts voor Java code gebruikt zou gaan worden. De modellen hebben enige opvallende overeenkomsten en verschillen, welke hieronder behandeld worden.

Als overeenkomsten komen de volgende punten duidelijk naar voren:

- Beide modellen hebben een abstract deel in het model, wat zorgt voor een generiek niveau. De taalelementen zijn als subelementen van dit generieke niveau gespecificeerd. De relaties tussen de taalelementen blijven binnen het model gehandhaafd (packages bevatten classes, classes bevatten methodes enzovoorts) en de elementen worden intern gebruikt met hun gekwantificeerde naam.
- Beide modellen maken gebruik van properties op de elementen welke direct uit de code te halen zijn. Hierna worden deze elementen pas werkelijk geïnterpreteerd.

Als verschillen komen de volgende punten duidelijk naar voren:

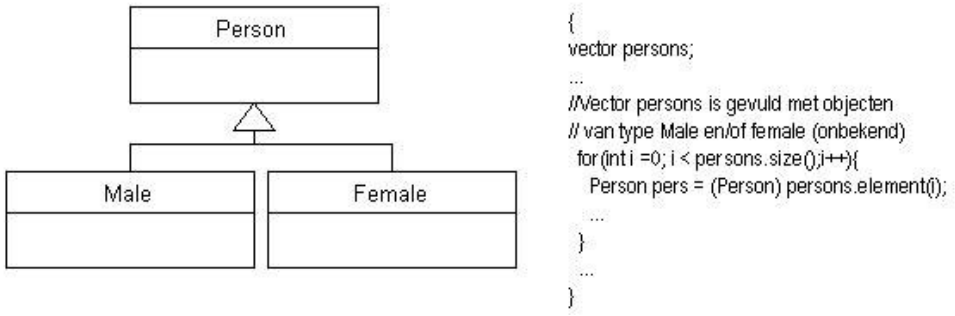
- Het Famix model is ontwikkeld als generiek model, wat taalspecifieker gemaakt kan worden met behulp van taal plugins. Tevens is het Famix model wat betreft functionaliteit uitbreidbaar met tool plugins. RevJava is, alhoewel grotendeels generiek, hier en daar toch wat Java-georiënteerd van opzet. RevJava is in bepaalde mate wel uitbreidbaar met behulp van plugins, maar dit geld voor de functionaliteit en niet voor het uitbreiden van het model zelf.
- Waar in het Famix model gebruik wordt gemaakt van verschillende soorten Associations en Argumenten, kent RevJava eigenlijk alleen een referentie aan een type door middel van TypeRef. Hierbij wordt geen onderscheid gemaakt tussen de verschillende referenties (inheritance, method invocation enzovoorts), maar de referentie wordt gespecificeerd met het roottype van het metamodel (GenElement).
- Inner classes worden in het Famix model niet ondersteund.
- Globale methodes en variabelen worden in het RevJava metamodel niet ondersteund.

Opvallend detail is dat de modellen elk op hun eigen manier niet geheel compleet zijn: het Famix model kent geen definitie van inner classes en is dus niet geheel compleet wat Java code betreft, tenzij het met een omweg op te lossen is door middel van plugins. RevJava kent geen globale methodes en variabelen, wat een probleem kan vormen wanneer men andere talen in het metamodel zou in gaan lezen. Aangezien RevJava niet uitbreidbaar is met objecten door middel van plugins, zal voor ondersteuning van globale methodes en variabelen een aanpassing aan het metamodel onvermijdelijk zijn.

4.4. Type analyse

RevJava analyseert software aan de hand van type informatie. In dit hoofdstuk worden enkele mogelijkheden bekeken waarmee de analyse van RevJava uitgebreid zou kunnen worden. De volgende onderwerpen hebben we onderzocht:

- Nullpointer detectie: nullpointer exceptions zijn ernstige programmeerfouten welke door de compiler niet gedetecteerd worden. Bij een slechte exception-afhandeling kan een nullpointer de applicatie geheel vast laten lopen.
- Overflow errors van bijvoorbeeld arrays: array overflows kunnen wanneer zij niet goed afgevangen zijn de gehele applicatie laten crashen, terwijl ze vrij gemakkelijk te voorkomen zijn.
- Analyse van concurrente programma's: bijvoorbeeld deadlock detectie of te veel synchronisatie (dit kan de applicatie traag maken).
- Typebepaling van objecten die in een Vector opgeslagen worden: het zou handig kunnen zijn om te weten wat voor types er in een Vector worden opgeslagen, zodat wanneer er objecten uit de Vector worden gehaald deze bijvoorbeeld optimaal gecast kunnen worden. In dit geval gaat het er niet om fouten op te sporen, maar om de code te optimaliseren. Zie ook figuur 4.4.1 hieronder:



Figuur 4.4.1: wanneer er in een Vector alleen objecten van type 'Male' worden opgeslagen binnen het systeem, zal de cast bij het lezen uit de Vector optimaal zijn als het object weer naar 'Male' gecast wordt in plaats van naar 'Person'.

De eerste drie onderwerpen hebben we gekozen, omdat er volgens de literatuur nog weinig op het gebied van Java gedaan was of er aangegeven werd dat het belangrijke analyses waren voor object-georiënteerde software. De laatste is een idee dat bij Gert Florijn speelde.

Wat betreft nullpointer detectie en concurrency zijn er programma's die een subset van mogelijke problemen op kunnen sporen, echter met een foutmarge in de analyse. Het is bijvoorbeeld moeilijk aan te geven hoe strikt een bepaalde analyse moest zijn of er worden stukken code als fout aangeduid terwijl ze dat niet zijn. Op het gebied van overflow errors is wel geschreven en geëxperimenteerd, maar er zijn nog geen concrete oplossingen bedacht die ook daadwerkelijk bruikbaar zijn voor RevJava.

De moeilijkheid van deze analyses is, dat de uitkomsten pas runtime bekend zijn en deze afhankelijk kunnen zijn van invoer van een gebruiker. Tevens kunnen er in de code vele paden zijn, bijvoorbeeld door 'if…then…else…' constructies, welke allemaal onderzocht moeten worden. Dit laatste houdt eigenlijk automatisch in dat er een vorm van flow-analyse op het model plaats zal moeten vinden, wat de performance van RevJava negatief zal beïnvloeden. Wat betreft typeanalyse binnen een Vector geldt dat de mogelijke voordelen van de optimalisaties in de code niet echt opwegen tegen de extra tijd die er voor de analyse nodig is. Het lijkt echter onmogelijk om alle optredende gevallen te dekken. Wanneer dan ook nog de performance slechter wordt en er sprake is van een foutmarge in de analyse lijkt het niet erg zinvol de analyses toe te voegen.

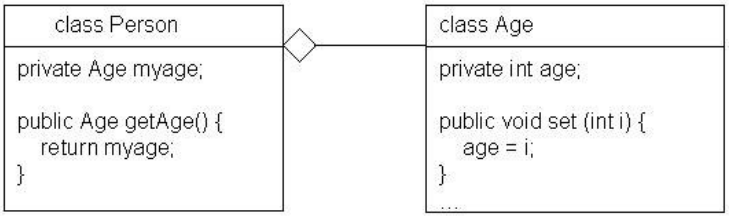
Zie voor type analyse in de literatuur: [10] [11] [12] [13] [14] [15]

4.5 Alias Analyse

Binnen een programma kunnen verschillende objecten een pointer hebben naar één enkele instantie van een bepaald object. We spreken dan van meerdere aliassen van dat object. Het gebruik van aliassen kan heel gemakkelijk zijn, maar er zitten ook grote nadelen aan het gebruik van aliassen:

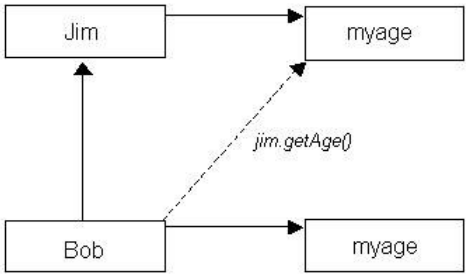
- Het object-georiënteerde programma wordt onoverzichtelijker voor de programmeur.
- Aliassen maken de software gevoelig voor verandering. Aanpassing van software met veel gebruik van aliassen heeft vaak bugs tot gevolg. Software is daardoor minder gemakkelijk aan te passen of uit te breiden.

Met behulp van alias analyse kan er aangegeven worden waar er sprake is van foutgevoelige constructies met betrekking tot een bepaald object. Dit wordt verduidelijkt met een simpel voorbeeld, waarin een object Person een object Age in beheer heeft:



Figuur 4.5.1: een Person object beheert een Age object

Hieronder zien we twee Person objecten: 'Bob' en 'Jim', welke beiden een Age object 'myage' beheren. Aangezien Bob ook een pointer naar Jim heeft, kan Bob via de getAge() methode een pointer naar het 'myage' object van Jim verkrijgen:



Figuur 4.5.2: voorbeeld van het gebruik van Persons en Ages

Wanneer nu de 'set(int i)' methode van 'jim.getAge()' aangeroepen wordt, kan de waarde van de age-variabele daarvan aangepast worden. Mogelijk is dit niet de bedoeling en geeft dit een fout aan in het ontwerp of de implementatie.

Met behulp van alias analyse komen dit soort mogelijke problemen boven. Het metamodel van RevJava bezit grotendeels de informatie die nodig is om alias analyse uit te voeren, maar het kost veel tijd deze gegevens te combineren. Tevens kan het nodig zijn flow-analyse uit te voeren om alle aliassen van een object te analyseren. Alias analyse is derhalve geen critic die standaard meegenomen zou moeten worden, maar eerder een onafhankelijke analysemogelijkheid die op aangeven van de gebruiker uitgevoerd kan worden.

Zie voor alias analyse in de literatuur: [6] [7] [8] [9]

4.6 Conclusies oriënterend onderzoek

Tijdens het onderzoek zijn we drie mogelijke momenten tegen gekomen waarop de analyses plaats kunnen vinden:

1. voor het compileren
2. na het compileren
3. tijdens het compileren

Alhoewel het voor de gebruiker misschien het gemakkelijkst is om de analyse op de sourcecode te doen zonder eerst te hoeven compileren, bevat deze code veel voor de analyse overbodige informatie welke de snelheid van de analyse mogelijk negatief beïnvloedt. Het is vanuit de performance gezien beter om de analyse op de gecompileerde code te doen, aangezien deze door de compiler vaak al geoptimaliseerd is. Een nadeel hiervan is dat bepaalde informatie verloren gaat. Wanneer men bijvoorbeeld een field definieert halverwege een class, is het mogelijk dat de compiler deze verplaatst naar het begin van de code van de class om efficiency redenen.

Om de analyse uit te voeren tijdens de compilatie is enige extra inzet van de programmeur vereist, tenzij er een standaard compiler te vinden is welke al analyses ingebouwd heeft. Dit betekent echter nog steeds dat de programmeur gebruik moet maken van een externe compiler, terwijl er in de ontwikkelomgeving vaak al een compiler meegeleverd wordt.

Er is moeilijk te zeggen welke methode de beste is. De eerste twee methodes ontlopen elkaar niet veel qua functionaliteit, maar wat betreft performance en compleetheid van de informatie verschillen ze enigszins. De laatste methode lijkt het meest geschikt voor het vinden van specifieke bugs in hele complexe systemen als operating systems en dergelijke, waar het zijn nut ook al bewezen heeft. Eén en ander heeft ook te maken met persoonlijke voorkeur en de mogelijkheid de uit te voeren analyses daaraan aan te passen.

Wat betreft het Famix model kunnen we zeggen dat het metamodel van RevJava vergelijkbaar is wat betreft opzet en dat beide modellen generiek genoeg zijn voor meerdere object-georiënteerde talen. De grootste verschillen zijn de uitbreidbaarheid van het Famix model met behulp van plugins en de afhandeling van object relaties (bijvoorbeeld inheritance). Wat wel erg handig is binnen het Famix model, is dat er een verschil gemaakt wordt tussen lokale variabelen en parameters van methoden, dit hebben we meegenomen bij het ontwerpen van een nieuw metamodel voor RevJava.

Wanneer we alias analyse of één van de onderwerpen van de type analyses toe willen voegen aan RevJava zal er minimaal de mogelijkheid tot flow-analyse ingebouwd moeten worden. Dit kan ingrijpende veranderingen voor RevJava inhouden, zoals:

1. De reader zal mogelijk extra informatie moeten inlezen, aangezien RevJava momenteel bijvoorbeeld geen idee heeft van de executievolgorde van operaties of taalconstructies als ‘if…then…else…’.
2. Mogelijk moeten er veranderingen plaatsvinden aan het metamodel om de nieuwe ingelezen informatie op te slaan.
3. Er moeten visitors komen voor het uitvoeren van de flowanalyse, welke alle mogelijke programmapaden langsgaan.
4. Er moeten nieuwe critics gedefinieerd worden.

De analyses zouden kunnen plaatsvinden op een subset van het metamodel, op aangeven van de gebruiker. Desnoods kan er zelfs op het aanroepen van de analyses extra informatie ingelezen worden en hoeft de reader dit niet bij de standaard critics al beschikbaar te hebben. RevJava blijft dan qua performance hetzelfde, maar kan op afroep extra analyses doen die extra tijd kosten. Het is echter niet gemakkelijk deze analyses in te bouwen.

5. Microsoft .Net

Naast de in het vorige hoofdstuk aan bod gekomen onderwerpen, hebben we in de eerste maand ook onderzocht wat Microsoft .Net (afgekort .net) precies is en hoe het in elkaar zit. Aan de hand van dit onderzoek konden we dan beslissen of het haalbaar was RevJava geschikt te maken om ook software die ontwikkeld is met .net te analyseren.

Dit hoofdstuk begint met een inleiding in .net en zal daarna de opbouw van de Microsoft Intermediate Language (MSIL) behandelen. De MSIL is de taal waarnaar de sourcecode van programma's die geschreven zijn voor .net gecompileerd worden. Alhoewel we de MSIL pas in een later stadium onderzocht hebben, zijn de twee onderwerpen vanwege de samenhang samengevoegd in dit hoofdstuk.

5.1 Inleiding in .net

Microsoft .Net is niet echt een specifiek product, maar meer een scala van samenhangende producten welke samen gebruikt kunnen worden om uiteenlopende soorten software te ontwikkelen. De filosofie achter .net is dat ontwikkelaars op een gemakkelijke manier software moeten kunnen ontwikkelen, zonder dat zij zich druk hoeven te maken over welke programmeertaal gebruikt wordt of het operating system waar de software op moet draaien. Tevens is er rekening mee gehouden dat er bij het ontwikkelen van software meerdere teams aan één product kunnen werken en dat geschreven stukken software hergebruikt moeten kunnen worden.

Microsoft .Net bestaat uit een aantal onderdelen:

- .net Framework: het .net framework biedt de basis voor het ontwikkelen van .net software.
- .net Servers: server producten die geoptimaliseerd zijn om met .net te werken, bijvoorbeeld SQL Server 2000.
- .net Web Services: web services maken het mogelijk gedistribueerde applicaties over het internet te ontwikkelen.

Aangezien het .net framework voor het ontwikkelen van .net software de belangrijkste component is hebben we ons hierop geconcentreerd. Informatie over de andere onderwerpen valt te vinden op de website voor .net van Microsoft.

5.1.1. Het .net framework

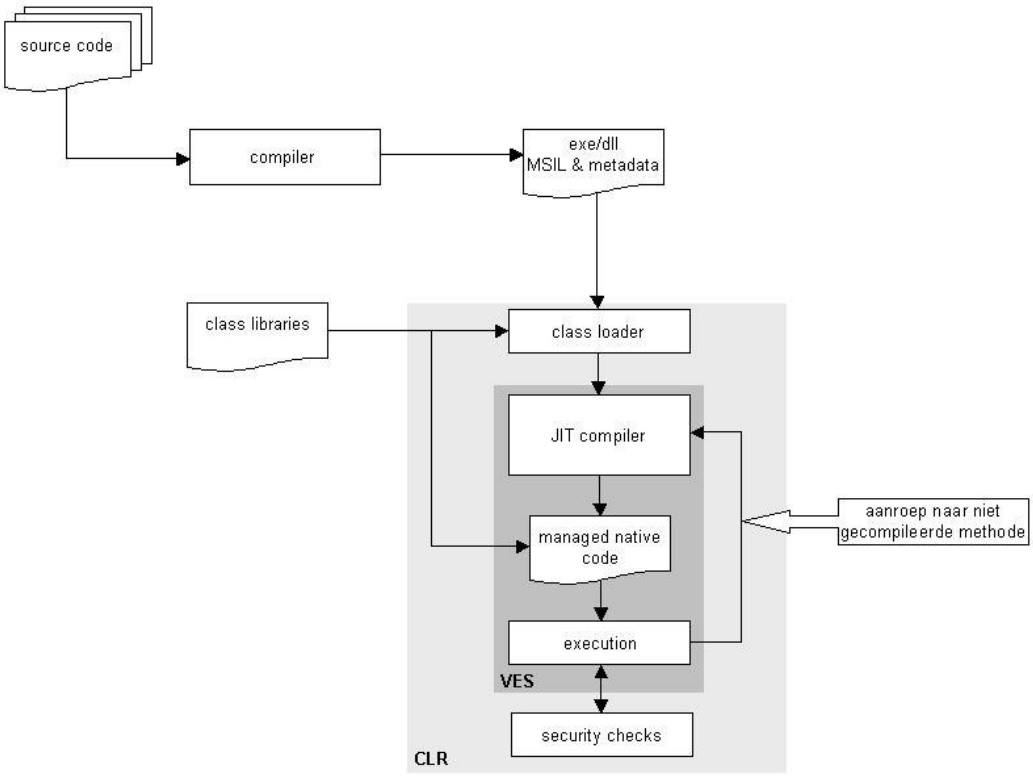
Het .net framework biedt ontwikkelaars de basisfaciliteiten voor het ontwikkelen van software met .net, waaronder verschillende tools voor het compileren en decompileren van sourcecode. Deze sourcecode kan geschreven zijn in verschillende programmeertalen.
De andere onderdelen van het framework zijn:

- Common Language Runtime (CLR): een abstractielaag tussen het operating system en de uit te voeren .net applicatie, qua functie te vergelijken met de virtual machine van Java.
- Unified Class Libraries: een hiërarchische reeks class libraries die ontwikkelaars kunnen gebruiken bij het ontwikkelen van .net software.
- ASP.NET: hiermee kunnen op een gemakkelijke manier webapplicaties gebouwd worden en web services gedefinieerd worden.

De compilers binnen het .net framework compileren de sourcecode van de verschillende .net talen (bijvoorbeeld C#) naar een intermediate language (de MSIL) in plaats van naar executeerbare machinecode. Dit zorgt ervoor dat .net software in verschillende programmeertalen ontwikkeld kan worden en maakt het gebruik van een runtime, de CLR, nodig om een .net applicatie te runnen.

Wanneer een .net applicatie opgestart wordt, zorgt de CLR ervoor dat de MSIL code naar machinecode wordt gecompileerd met behulp van een Just In Time (JIT) compiler. De CLR

bezit hiervoor een apart systeem, genaamd het Virtual Execution System (VES), wat tevens zorgt voor de executie van de code.



Figuur 1.1: een schematische weergave van sourcecode tot de executie van een applicatie. De CLR en het VES zijn duidelijk aangegeven.

Voor ieder operating system is er een aparte CLR. Wanneer men .net op een ander platform wil draaien, zal men zelf voor een CLR voor dat platform moeten zorgen wanneer deze niet door Microsoft is verzorgd. Het voordeel hiervan is dat de JIT compiler de gegenereerde machinecode kan optimaliseren voor het platform waar de CLR op zal gaan draaien.

5.1.2 De Microsoft Intermediate Language

De MSIL wordt door de compilers in het .net framework gegenereerd vanuit de sourcecode van de verschillende programmeertalen voor .net. Door middel van de MSIL is het mogelijk dat men voor het ontwikkelen van een .net applicatie gebruik kan maken van verschillende programmeertalen. Hiertoe wordt er bij de gecompileerde sourcecode metadata opgeslagen.

Een blok gecompileerde sourcecode met de daarbij horende metadata heet een assembly. Programmeurs kunnen nu, zonder de sourcecode te hebben van een assembly, gebruik maken van die assembly door deze in de sourcecode te importeren. Het maakt hierbij niet uit of de assembly oorspronkelijk in een andere programmeertaal geschreven is dan de taal waarin de programmeur zijn applicatie schrijft. Een assembly heeft als extensie .exe of .dll. De precieze opbouw en functie van een assembly komt in paragraaf 5.1.3 aan bod.

Om taalonafhankelijk te kunnen zijn is de MSIL object-georiënteerd gedefinieerd en sterk getypeerd. Deze typering is nodig, omdat alle .net programmeertalen dezelfde datatypes moeten gebruiken voor de uitwisseling van informatie tussen assemblies. Het Common Type System (CTS) bevat deze typering, welke in een hiërarchische objectstructuur is gedefinieerd. Het CTS stelt daarnaast ook regels op voor het gebruik van de types.

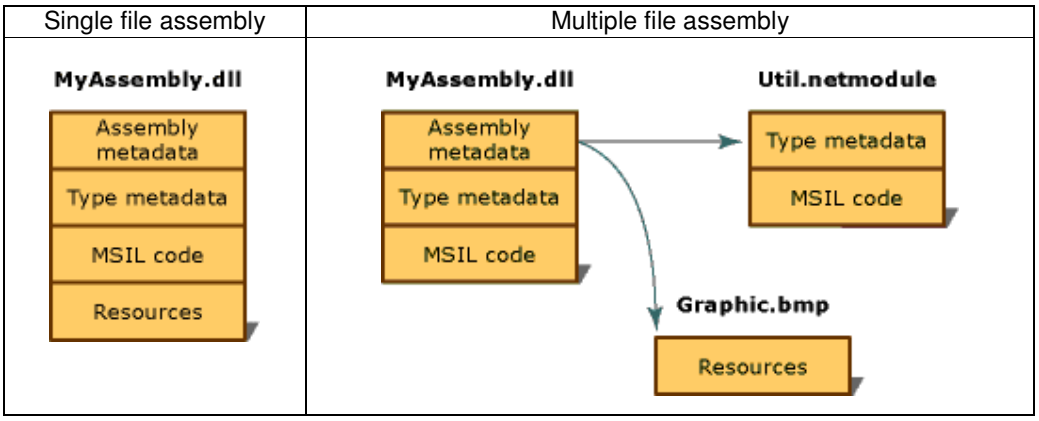
5.1.3 .Net applicaties

Een .net applicatie kan gebruik maken van verschillende assemblies naast degene waarin de applicatie zelf gedefinieerd is. Die assemblies worden externe assemblies genoemd en het systeem is te vergelijken met het import systeem in Java. Externe assemblies moeten geregistreerd zijn op het systeem waar de applicatie op gaat draaien, zodat de .net applicatie deze kan vinden en aanroepen.

Een assembly zelf kan ook bestaan uit meerdere componenten, namelijk uit modules. Een module is, net als een assembly, een gecompileerd stuk sourcecode met metadata, maar met het verschil dat een module niet als aparte applicatie gedraaid kan worden. Slechts wanneer een module bevat is in een assembly kan deze daadwerkelijk gebruikt worden. Een assembly heeft altijd minimaal één module met bijbehorende metadata, namelijk de module met daarin zijn eigen MSIL code.

Naast modules heeft een assembly ook assembly metadata (het manifest) en kunnen er resources (bijvoorbeeld images) in bevat zijn. In het manifest staat informatie over de assembly, zoals bijvoorbeeld welke files er tot een assembly behoren en het versienummer van de assembly.

Er zijn twee manieren waarop een assembly kan zijn opgebouwd: als single file assembly en als multiple file assembly.



Figuur 5.1.3.1: mogelijke manieren om een assembly op te bouwen

In de tabel is te zien dat het verschil tussen een assembly en een module concreet bestaat uit het ontbreken van assembly metadata en resources bij de module ten opzichte van de assembly. Tijdens de compilatie van een multiple file assembly moet aangegeven worden welke modules de assembly gebruikt, deze worden mee gecompileerd met de assembly. Een module heeft als extensie .netmodule.

Een multiple file assembly kan om verschillende redenen de voorkeur hebben boven een single file assembly:

- Er zijn features die weinig gebruikt worden: deze kunnen in aparte modules gezet worden, die pas geladen worden op het moment dat de features nodig zijn en vervolgens in de cache bewaard blijven.
- Er wordt gebruik gemaakt van verschillende programmeertalen om de applicatie te bouwen, of een bepaalde module wordt hergebruikt.
- Er werken veel mensen aan één systeem en door het gebruik van meerdere modules kan het systeem samengevoegd worden tijdens het compileren.

Door het gebruik van de CLR en de JIT compiler zou men verwachten dat een .net applicatie wat betreft performance minder scoort dan een applicatie die geen gebruik maakt van een runtime. Er zijn twee redenen waarom een .net applicatie echter qua performance ongeveer gelijk is aan een normale applicatie in een Windows omgeving.

1. De JIT compiler genereert machinecode die optimaal is voor de processor die op dat moment gebruikt wordt en voor het besturingssysteem. Bij applicaties die geen gebruik maken van een runtime (zoals de CLR) is dit eerste moeilijk te doen, omdat er dan voor iedere processor een aparte versie van de applicatie gemaakt moet worden.
2. Een grote .net applicatie bestaat typisch uit meerdere assemblies. Bij de start van het programma worden de assemblies die initieel nodig zijn geladen en gecompileerd, de andere assemblies pas wanneer zij aangeroepen worden. Geladen assemblies worden in een cache opgeslagen, zodat ze bij een volgende aanroep meteen beschikbaar zijn.

Voor paragraaf 5.1 is gebruik gemaakt van de volgende literatuur: [22] [23] [24] [35] [36]

5.2 Onderzoek naar de MSIL

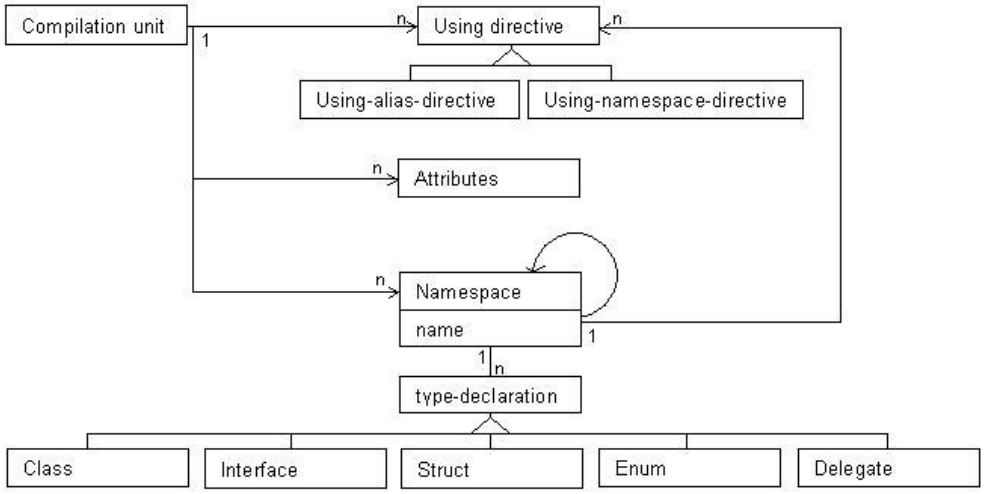
Nadat we besloten hadden het metamodel van RevJava aan te passen zodat er ook MSIL code ingelezen zou kunnen worden, moesten we een compleet beeld krijgen van de opbouw van de MSIL. Wanneer we deze opbouw kenden, zouden we de verschillen tussen de MSIL en Java kunnen bepalen en het metamodel aanpassen waar dit nodig mocht zijn.

5.2.1 De opbouw van C#

Aangezien we in eerste instantie geen grammaticadefinitie konden bemachtigen van de MSIL, zijn we begonnen met het opstellen van een model voor C#. De elementen binnen dit model hebben we vergeleken met de elementen in Java. Tevens konden we ons een beeld vormen over de MSIL vanuit het C# model, aangezien C# speciaal voor .net is ontwikkeld en het grootste deel van de MSIL functionaliteit beslaat.

De gemaakte modellen van C# worden top down behandeld, waarbij belangrijke verschillen met Java aangegeven zullen worden. Hierna worden verschillen tussen C# en de gegenereerde MSIL code behandeld.

<u>Compilation units en namespaces</u>



Figuur 5.2.1.1: outline van een compilation unit (een C# file).

In figuur 5.2.1.1 is het model van een C# file te zien, welke een compilation unit wordt genoemd. Een programma geschreven in C# kan uit meerdere compilation units bestaan. Een compilation unit kan de volgende elementen bevatten:

- Using directives: import statements om aan te geven dat er gebruik wordt gemaakt van types in bepaalde namespaces, andere compilation units of externe assemblies.
- Attributes: deze zijn optioneel. Door een attribute te definiëren is het mogelijk commentaar mee te compileren naar MSIL code. Dit werkt ongeveer hetzelfde als de @depricated tag in Java.
- Namespaces: in een compilation unit kunnen meerdere namespaces gedeclareerd worden. Impliciet bevat een compilation unit altijd al een default namespace, zodat een compilation unit ook direct types kan bevatten. Namespaces kunnen genest worden. Een namespace direct in de compilation unit mag een namespacestructuur aangeven, maar een geneste namespace mag dat niet, zie figuur 5.2.1.2 met een voorbeeld hiervan. Een namespace bevat typedeclaraties zoals classes en interfaces.

```
namespace n1.n2.n3 {
   class test {}
}
```

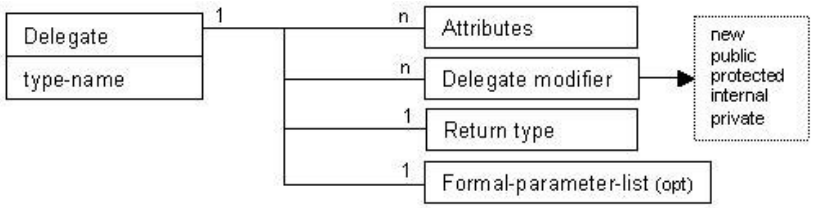
```
namespace n1.n2 {
   namespace n3 {
      class test {}
   } }
```

```
namespace n1 {
   namespace n2 {
      namespace n3 {
         class test {}
      } } }
```

```
namespace n1 {
   namespace n2.n3 {
      class test {}
   } }
```

Figuur 5.2.1.2: vier manieren om class 'test' in namespace n1.n2.n3 te declareren, de eerste drie zijn goed, de laatste is fout. De eerste namespace wordt steeds gedeclareerd in de default namespace van de omvattende compilation unit.

Enumerations (aangeduid als 'Enum' in figuur 5.2.1.1) en delegates zijn speciale constructies binnen C#. Een enumeration kan gebruikt worden om een verzameling van verschillende types aan te geven, ongeveer vergelijkbaar met een Vector in Java. Een delegate is een overblijfsel van functiepointers in C en C++, en zorgt ervoor dat een methode aangeroepen kan worden zonder zijn echte naam te kennen. Een delegate specificeert de signatuur van methodes waarvoor hij bedoeld is:



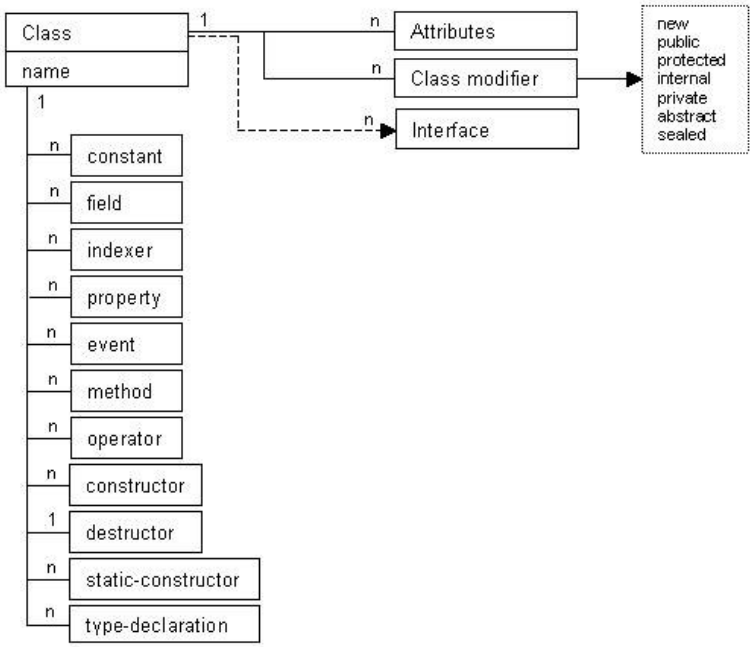
Figuur 5.2.1.3: delegates in C#

Aangezien dit allemaal nogal abstract klinkt volgt hieronder een voorbeeld van het gebruik van een delegate. De delegate word gebruikt voor het printen van een string naar een aantal bestemmingen:

```
...
public delegate void printFuncs(string toprint);

public void printToScreen(string toprint) {...}
public void printToLogFile(string toprint) {...}
public void printToPrinter(string toprint, string printername) {...}

public void printTest() {
   printFuncs print = new printFuncs(printToScreen);
   print += new printFuncs(printToLogFile);
   print("delegate test");
}
...
```

Figuur 5.2.1.4: wanneer de delegate 'print' in printTest() wordt uitgevoerd zal deze printToScreen en printToLogFile methodes uitvoeren met "delegate test" als parameter. De delegate is niet geschikt voor de methode printToPrinter aangezien de parameters van printToPrinter niet overeenkomen met de specificatie van de delegate printFuncs.

Classes

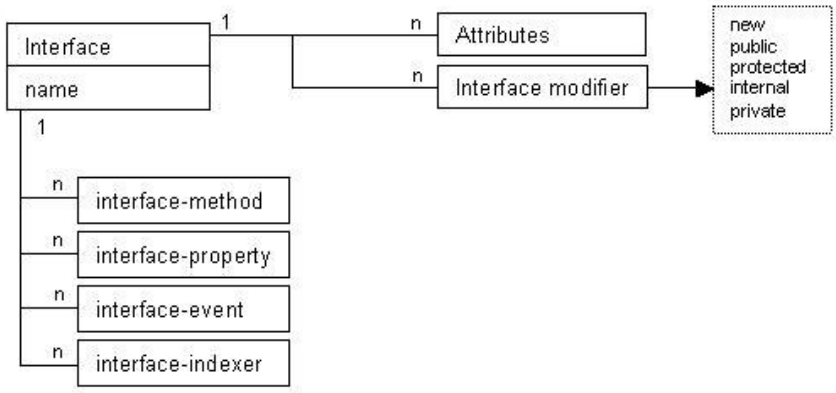


Figuur 5.2.1.5: classes in C#

In het bovenstaande model is te zien hoe een class in C# opgebouwd is. Naast classes kan er in C# ook gebruik gemaakt worden van structs. Een struct kan gebruikt worden als een bepaalde class alleen maar data opslaat en geen gedrag vertoont, zoals bijvoorbeeld het bijhouden van coördinaten van een punt in een vlak. De opbouw van een struct is bijna gelijk aan die van een class, maar een struct kan geen destructor declareren of nieuwe objecten aanmaken.

Java kent twee soorten geneste classes: static en niet static (innerclasses) geneste classes. C# kent alleen het equivalent van static geneste classes van Java.

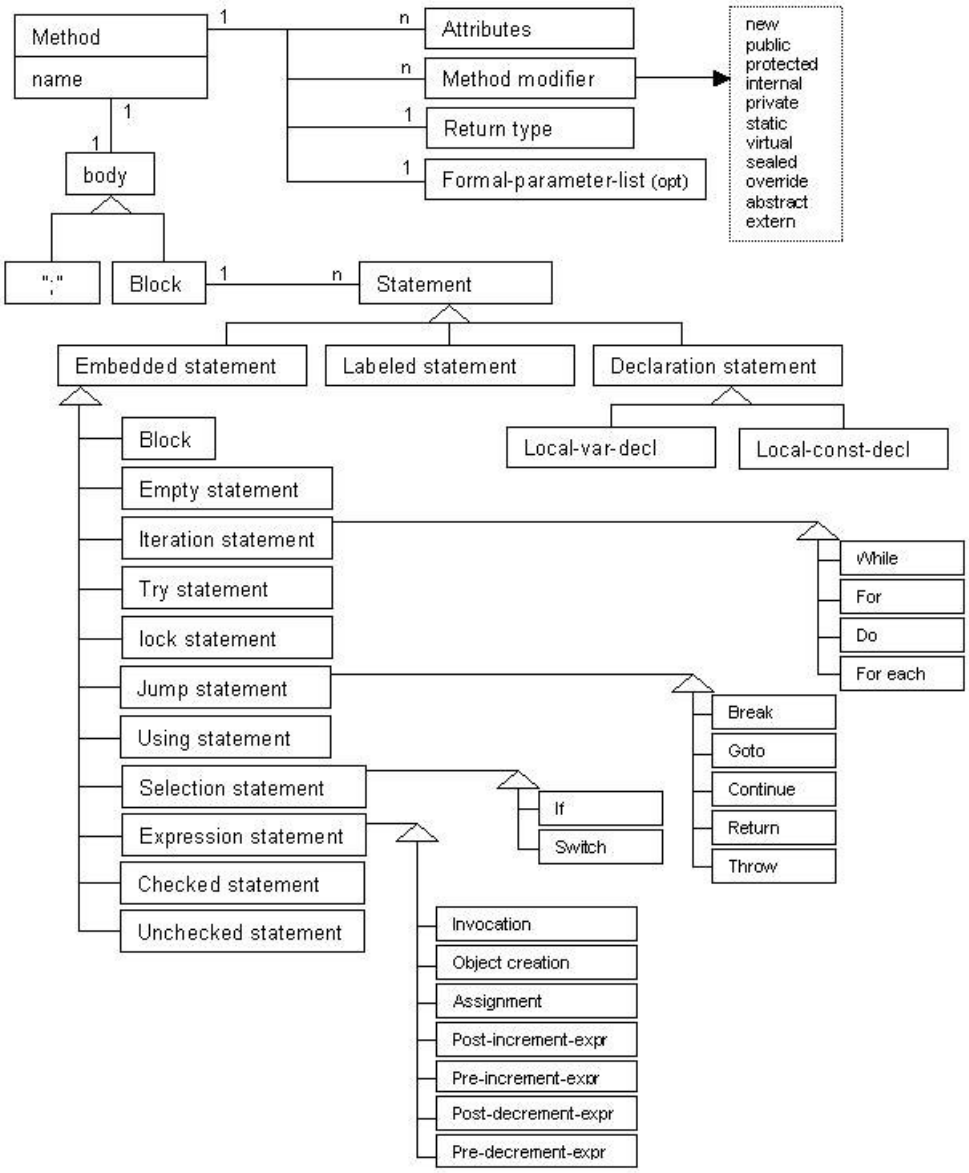
Interfaces



Figuur 5.2.1.6: interfaces in C#

In bovenstaande figuur staat het model van een interface in C#. Interfaces in C# werken hetzelfde als interfaces in Java. De elementen die in een interface gedeclareerd kunnen worden zijn: methodes, events, properties en indexers. Bij Java is het ook mogelijk dat er fields in een interface kunnen staan, maar bij C# kan dit niet. Interfaces kunnen bij C# wel genest zijn, maar zelf geen geneste types definiëren.

Methodes

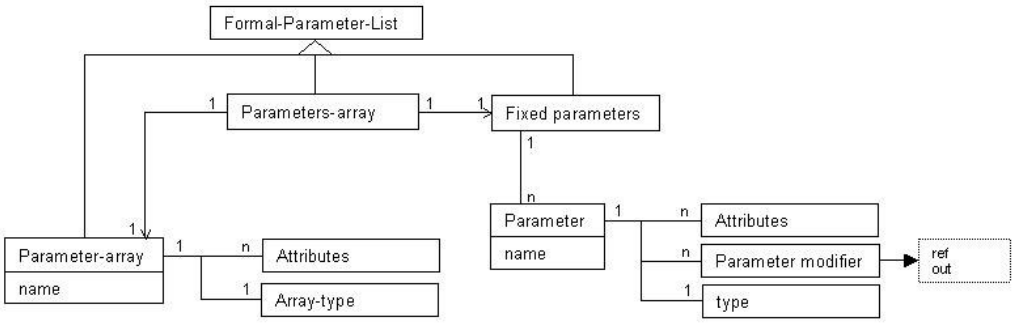


Figuur 5.2.1.7: methodes in C#

Methodes in C# verschillen vrij weinig van methodes in Java. De body van een methode kan leeg zijn of statements en lokale variabele declaraties bevatten. Daarnaast is het ook mogelijk dat de body uit meerdere blokken code bestaat, bijvoorbeeld wanneer een gedeelte van de methodecode door een try-statement beveiligd wordt.

C# kent wel statements die in Java niet bekend zijn, maar de type gerelateerde statements, waarin wij geïnteresseerd zijn, verschillen qua structuur niet van die van Java.

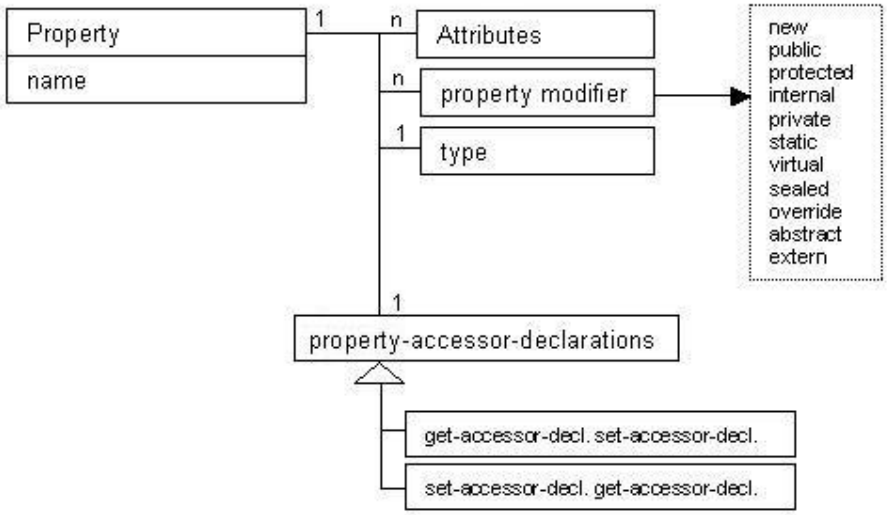
Een formal-parameter-list specificeert welke parameters een methode heeft:



Figuur 5.2.1.8: een formal parameter list

Parameters in C# werken hetzelfde als in Java, afgezien van het feit dat parameters pointers kunnen zijn. Dit wordt aangegeven met de modifiers 'ref' en 'out'. In C# bestaan er ook parameters-arrays. Een parameters-array wordt meegegeven na alle normale parameters, als deze er zijn en heeft als extra modifier 'params'. Met een parameters-array kan een variabel aantal parameters meegegeven worden aan een methode.

Properties en indexers

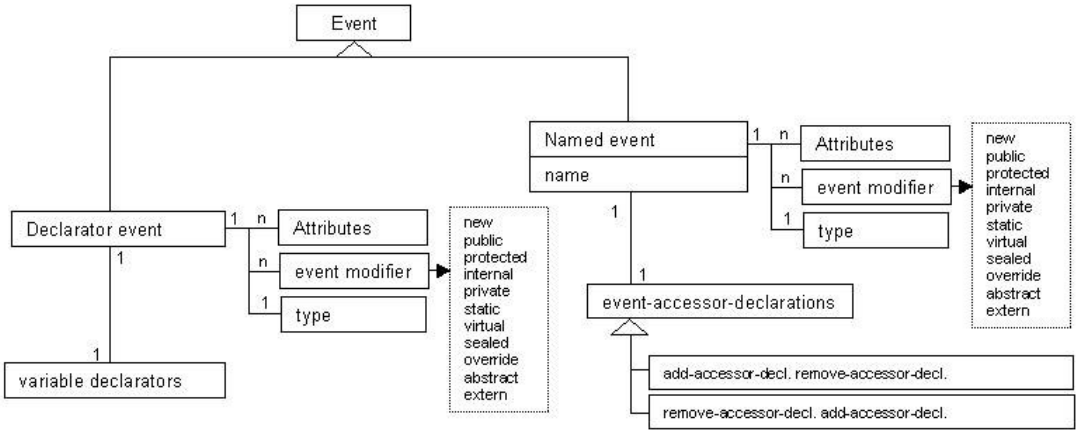


Figuur 5.2.1.9: properties in C#

Binnen C# kan gebruik worden gemaakt van properties. Properties zijn een manier om op een gemakkelijke en overzichtelijke wijze get- en setmethodes binnen een class te definiëren. Deze methodes kunnen gebruikt worden om een field te manipuleren, maar ook om een database te benaderen of een andere niet aan een field gerelateerde operatie. Een property definieert minimaal altijd een get- of setmethode, of beide.

Een indexer is bijna gelijk aan een property, behalve dat een indexer werkt op arrays. Wanneer een class een array declareert, kan een indexer op een zelfde manier als een property dat doet, get- en setmethodes voor de elementen in die array specificeren.

Events



Figuur 5.2.1.10: events in C#

Met behulp van een event is het gemakkelijk om notificaties door te geven vanuit een object. Een event kan gedeclareerd worden met een naam en een aantal methodes of als een instantie van een al bestaand event (bijvoorbeeld een mouse-event). Een event heeft minimaal de add-accessor en remove-accessor methodes en eventueel een extra methode 'fire'.

Verschillen tussen C# en de MSIL

Na het bestuderen van C# en de verschillen daarvan met Java hebben we gekeken wat er met de C# code gebeurde wanneer het naar de MSIL gecompileerd werd. We hebben de volgende grote verschillen gevonden:

- C# statements: de statements in C# zijn in de MSIL terug te vinden als stackcode. Deze code lijkt erg op de machinecode, maar is beter leesbaar en begrijpbaar. Wat betreft de soorten statements waarin we geïnteresseerd zijn, namelijk type gerelateerde statements, is alles goed te begrijpen.
- C# properties: de methodes die een property definieert zijn in de MSIL terug te vinden als normale methodes in de class. Binnen de class staat nog een aparte property-declaratie met daarbij de methodes waarmee hij werkt.
- C# events worden hetzelfde afgehandeld als de properties: de methodes staan in de class en via een aparte event-declaratie is duidelijk van welke methodes het event gebruik maakt.
- C# constructors worden in de MSIL methodes met de naam '.ctor'. Static constructors worden '.cctor()' genoemd. Wanneer er binnen een C# class geen constructor gedefinieerd is, wordt deze standaard aan de MSIL toegevoegd tijdens de compilatie.
- C# delegates worden in de MSIL classes met als superclass System.MulticastDelegate.
- C# structs worden in de MSIL classes met als superclass System.ValueType.
- C# enums in IL worden in de MSIL classes met als superclass System.Enum.
- Wanneer men in C# een constante declareert, staat deze in de MSIL als 'static literal' of 'initonly' variabele.
- Using directives vindt men niet terug in de MSIL. Waar nodig worden volledig gekwantificeerde namen gebruikt.
- Een namespace zonder types erin wordt in de MSIL niet als aparte namespace weergegeven, maar een geneste namespace daarin krijgt dan de naam van de omvattende lege namespace als voorvoegsel voor zijn naam.

Als laatste verschil wil ik de verschillende manieren om vanuit C# een .net applicatie op te bouwen behandelen. Er zijn grofweg drie manieren om vanuit de sourcecode van C# een assembly te compileren, wat te maken heeft met de manieren waarop C# gebruik kan maken van andere files:

- Er wordt gebruik gemaakt van andere C# files.
- Er wordt gebruik gemaakt van een externe module.
- Er wordt gebruik gemaakt van een externe assembly.

In het eerste geval zal er aan de compiler een lijst van C# files meegegeven moeten worden om te compileren. Al deze files worden samengevoegd tot een assembly. Wanneer de assembly gedeassembleerd wordt, staat alle informatie uit de verschillende C# files in één enkele IL-file. In de MSIL is er dus geen informatie dat de code vanuit meerdere files gegenereerd is!

In het tweede geval moet er aan de compiler meegegeven worden welke module er mee gecompileerd moet worden. Deze module wordt een onderdeel van de assembly. Wanneer de assembly echter gedeassembleerd wordt, resten er van de module slechts nog de import statements in de MSIL. Wanneer men ook de MSIL van de module wil hebben, moet deze ook gedeassembleerd worden.

In het derde geval wordt er bij het compileren slechts een referentie naar de externe assembly opgenomen.

Na dit onderzoek hadden we een goed beeld van de MSIL die vanuit C# gecompileerd kon worden. De volgende paragraaf gaat dieper in op de resultaten van dit onderzoek.

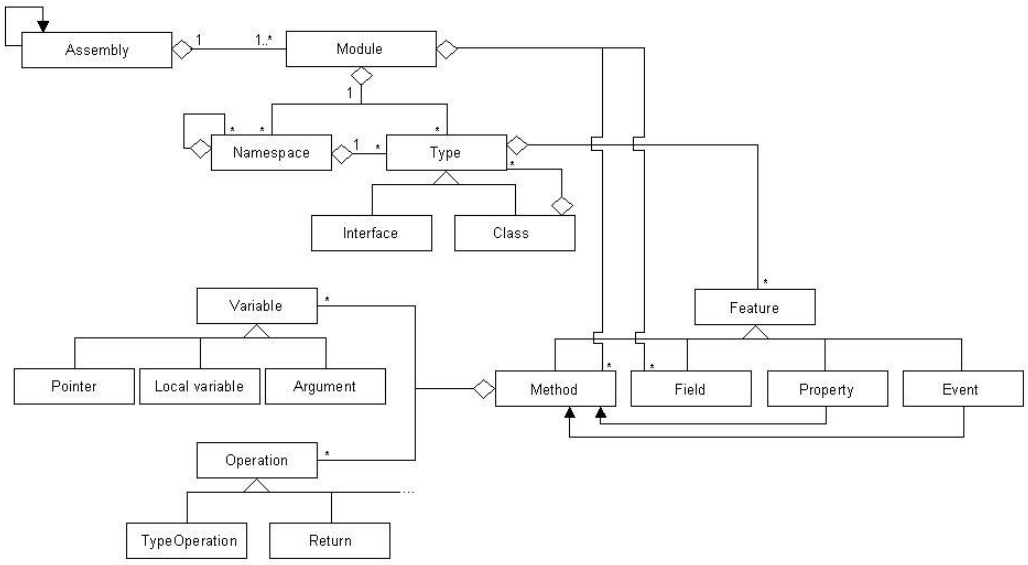
Voor paragraaf 5.2.1 is gebruik gemaakt van de literatuur genoemd bij: [25] [29] [38] [39] [40]

5.2.2 De opbouw van de MSIL

Deze paragraaf is gesplitst in twee onderdelen: eerst wordt de opbouw van een MSIL file behandeld aan de hand van een opgesteld model. Hierna zullen we een blik werpen op de formele grammatica van de MSIL.

De opbouw van een MSIL file

Het onderstaande model geeft aan hoe de MSIL globaal is opgebouwd:



Figuur 5.2.2.1: globaal model van de opbouw van MSIL code

Het valt op dat het model relatief weinig elementen bevat, wat aangeeft dat de compilers van de verschillende .net talen de code naar een generiek niveau compileren. Het model vertoont zelfs overeenkomsten met de manier waarop vanuit het RevJava metamodel de bovenstaande relaties weergegeven zouden worden. Voordat we de verschillen tussen Java en de MSIL behandelen, worden de elementen uit het bovenstaande model eerst kort toegelicht en komt de grammatica van de MSIL aan bod. De elementen uit het model zijn als volgt gedefinieerd:

- Assembly: een assembly is opgebouwd uit één of meerdere modules en kan gebruik maken van andere assemblies, namelijk de externe assemblies.
- Module: een module bezit impliciet de default namespace, waardoor een module eigenlijk direct types bevat. Daarnaast kunnen er in een module andere namespaces gedeclareerd worden en kan een module globale methodes en fields bevatten.
- Namespace: een namespace is mogelijk genest en bevat types, maar mogelijk ook andere geneste namespaces.
- Type: een type is een interface of class en bevat features.
    - Class: een class kan alle soorten features bevatten: methodes, fields, events en properties.
    - Interface: een interface kan geen fields bevatten.
- Feature: features geven gedrag aan types.
    - Method: een methode kan lokale variabelen en operaties declareren.
    - Property en event: properties en events maken gebruik van methodes, waarin het gedrag van die event of property geïmplementeerd is.
- Variable: een variabele is een lokale variabele of een argument dat aan een methode meegegeven wordt.
- Operation: er zijn veel soorten operaties, een paar voorbeelden hiervan:
    - type operations
    - returns
    - branch operations

We hebben verschillende modellen gevonden waar types en namespaces niet aan een module, maar aan de assembly toegewezen worden. Uit hoofdstuk 5.1.3 blijkt al dat er weinig verschil zit tussen een module en een assembly, maar het is toch wel vreemd om het bestaan van een module te negeren in het model.

Een verklaring zou kunnen zijn dat wanneer een assembly gemaakt wordt, informatie uit de modules in de assembly wordt samengevoegd, zodat deze informatie centraal aanwezig is. Wanneer men dan een assembly zou beschrijven, zou de assembly de types en namespaces bevatten. Wanneer een assembly echter gedeassembleerd wordt staat al deze informatie verspreid over de verschillende modules. Aangezien onze interesse uit gaat naar die gedeassembleerde assemblies, zijn in het bovenstaande model types en namespaces bevat in een module.

Grammatica van de MSIL

Toen we een beeld van de MSIL hadden gevormd vanuit C#, vonden we een document dat de precieze grammatica van de MSIL beschreef. We hebben dit document gebruikt om onze resultaten te verifiëren en waar nodig aan te vullen. Een voorbeeld van zo'n aanvulling is het feit dat er globale fields en methodes bestaan. In C# bestaan deze niet en wij hadden ze daarom ook niet in de MSIL gevonden. Om een beeld te geven van de grammatica van de MSIL wordt hieronder een stuk ervan behandeld:

```
<ILFile> :: = <decl>*

<decl> ::=
.assembly <dottedname> { <asmDecl>* }
|.assembly extern <dottedname> { <asmRefDecl>* }
| .class <classHead> { <classMember>* }
| .class extern <exportAttr> <dottedname> {<externClassDecl>*}
| .corflags <int32>
| .custom <customDecl>
| .data <datadecl>
| .field <fieldDecl>
| .file [nometadata] <filename> [.hash = ( <bytes> )] [.entrypoint ]
| .mresource [public | private] <dottedname> [(<QSTRING> )] { <manResDecl>*}
| .method <methodHead> { <methodBodyItem>* }
| .module [<filename>]
| .module extern <filename>
| .namespace { <decl>* }
| .subsystem <int32>
| .vtfixup <vtfixupDecl>
| <externSourceDecl>
| <securityDecl>
```

Een ILFile bestaat uit een lijst van declaraties. In een ILFile kunnen de volgende declaraties voorkomen: assemblies, modules, namespaces, classes, fields en methodes. Verder kan door middel van het keyword *extern* worden aangegeven dat assemblies, modules en classes vanuit een andere file geïmporteerd worden.

Een assembly wordt aangegeven door het keyword *.assembly* en heeft de volgende grammatica:

```
<decl> ::=
.assembly <dottedname> { <asmDecl>* }
| .assembly extern <dottedname> { <asmRefDecl>* }
| .corflags <int32>
| .file [nometadata] <filename> .hash = ( <bytes> )[.entrypoint ]
| .module extern <filename>
| .mresource [public | private] <dottedname> [(<QSTRING>)] {<manResDecl>*}
| .subsystem <int32>
```

Wanneer *.subsystem* aanwezig is betekent dit dat de assembly geen library is, maar uitgevoerd kan worden als applicatie. Het geeft een environment aan voor het programma.

De *.file* geeft een file aan die tot een assembly behoort, bijvoorbeeld een image. Hiervan kan een hash berekend zijn, deze wordt aangegeven met *.hash.* Wanneer de file code bevat die de assembly uit kan voeren, wordt met *.entrypoint* aangegeven waar de assembly de code kan benaderen.

De *.msresource* geeft een datafile aan waarin manifestinformatie staat. Deze informatie kan als private voor de assembly gedefinieerd worden. Over de assembly kan de volgende informatie gespecificeerd worden:

```
<asmDecl> ::=
.custom <customDecl>
| .hash algorithm <int32>
| .publickey = ( <bytes> )
| .ver <int32> : <int32> : <int32> : <int32>
| <securityDecl>
```

De *.custom* kan gebruikt worden voor meerdere soorten informatie, waaronder de attributes die op de assembly gedefinieerd zijn.

Met *.hash algorithm* wordt het algoritme aangegeven waarmee de files die in *.hash* gespecificeerd staan gehashed zijn.

Met *.publickey* wordt de digitale signatuur van de schrijver van de assembly aangegeven. Het versienummer van de assembly wordt aangegeven met *.ver* gevolgd door vier integers. Verder kan een assembly nog zaken over security definiëren, dit is bijvoorbeeld het geval wanneer er in de code pointers worden gebruikt.

Het volgende stuk grammatica geeft de mogelijke declaraties aan binnen een assembly of module:

```
<decl> ::=
| .class <classHead> { <classMember>* }
| .custom <customDecl>
| .data <datadecl>
| .field <fieldDecl>
| .method <methodHead> { <methodBodyItem>* }
| <externSourceDecl>
| <securityDecl>
```

Hierin is te zien dat een assembly of module classes, fields en methodes kan bevatten. De *.data* wordt gebruikt om datafields te creëren binnen een assembly of module. Bijvoorbeeld:

```
.data theInt = int32(123)
```
maakt een data field ‘theInt’ met als waarde 123.

Classes worden als volgt gedeclareerd:

```
<decl> ::= | .class <classHead> { <classMember>* }
```

Deze declaratie kan binnen een namespace staan, maar ook binnen een module of assembly. Dit laatste houdt in dat de class in de default namespace van de module of assembly staat. De classHead bevat de volgende elementen:

```
<classHead> ::=
<classAttr>* <id> [extends <typeReference>] [implements
<typeReference> [, <typeReference>]*]
```

Een class-header bestaat uit de volgende elementen:

- class attributes: tags zoals ‘private’, ‘static’ of ‘abstract’.
- De naam van de class
- Het supertype van de class. MSIL ondersteunt geen multiple inheritance.
- Een lijst van interfaces die de class implementeert.

Een class heeft in de MSIL altijd een supertype. Wanneer dit supertype in de sourcecode niet gespecificeerd is, krijgt de class tijdens de compilatie het roottype System.Object als superclass. De uitzondering hierop is als na *.class* het keyword *interface* volgt. De class moet dan behandeld worden als een interface. Een interface krijgt niet automatisch System.Object als supertype.

Binnen een class kunnen class members gedefinieerd worden:

```
<classMember> ::=
.class <classHead> { <classMember>* }
| .custom <customDecl>
| .data <datadecl>
| .event <eventHead> { <eventMember>* }
| .field <fieldDecl>
| .method <methodHead> { <methodBodyItem>* }
| .override  <typeSpec> :: <methodName> with <callConv> <type>
            <typeSpec> :: <methodName> ( <parameters> )
| .property <propHead> { <propMember>* }
| .size <int32>
| <externSourceDecl>
| <securityDecl>
```

Een nested type wordt hetzelfde aangegeven als een normale class, echter in de class header staat het keyword *nested*. Verder is er te zien dat er methodes, events, properties en fields binnen een class gedefinieerd kunnen worden.

Als laatste worden hier de belangrijkste elementen van de body van een methode behandeld. Een body van een methode bestaat uit items:

```
<methodBodyItem> ::=
.custom <customDecl>
| .data <datadecl>
| .entrypoint
| .locals [init] ( <localsSignature> )
| .maxstack <int32>
| .override <typeSpec>::<methodName>
| .param [ <int32> ] [= <fieldInit>]
| <externSourceDecl>
| <instr>
| <id> :
| <securityDecl>
| <sehBlock>
```

Met de *.entrypoint* kan aangegeven worden of de methode het entrypoint van de assembly is. Deze staat normaal gesproken in de methode Main. Elke assembly heeft precies één entrypoint. Het ontbreken of het dubbel definiëren hiervan in de sourcecode leidt tot een foutmelding van de compiler.

De *.locals* geeft aan welke lokale variabelen er in de methode gedeclareerd worden. De signatuur van een lokale variabele bestaat uit een volgnummer, een type, een naam en mogelijk een toegekende waarde. Met de *.param* wordt aangegeven dat de methode een parameter array meekrijgt.

5.2.3 Verschillen tussen Java en MSIL

Na het bestuderen van de MSIL waren we in staat de verschillen tussen Java en de MSIL te bepalen. Met behulp van deze verschillen konden we in een volgend stadium het metamodel van RevJava aan gaan passen. Tevens konden we waar mogelijk voor bepaalde verschillen generieke oplossingen bedenken, die in het metamodel ingepast zouden moeten worden.

Assemblies en modules

Java kent geen echte equivalenten van assemblies en modules in MSIL. Het enige wat in de buurt komt is een Jar file. Een Jar file kan echter geen andere Jar file bevatten, iets wat tussen assemblies en modules wel kan. Daarbij kan een Jar file geen globale methodes en fields hebben, wat in assemblies en modules wel mogelijk is.

Namespaces

Java kent packages in plaats van namespaces. Beiden hebben eenzelfde functie, namelijk de groepering van types. Er zijn echter een aantal belangrijke verschillen:

1. Packages in Java definiëren een structuur in de directories, namespaces doen dit niet. Lege namespaces (namespaces zonder types erin gedeclareerd) worden zelfs niet meer als aparte namespace weergegeven.
2. Zowel namespaces als packages kunnen een geneste structuur hebben. Bij Java betekent deze nesting echter niets. Wanneer een type vanuit een geneste package een type uit de parent van zijn package wil aanroepen moet deze eerst geïmporteerd worden. Binnen MSIL is dit zonder importstatements wel mogelijk.

Classes en interfaces

Classes en interfaces binnen Java en MSIL zijn eigenlijk vrijwel identiek. Er zijn echter wel een aantal belangrijke verschillen:

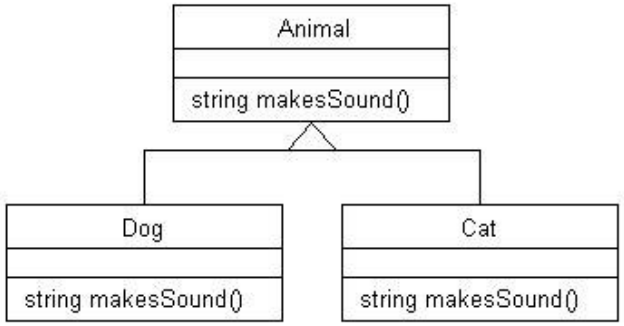
1. Interfaces in MSIL kunnen geen fields bevatten.
2. De tags die op een class of interface gedefinieerd kunnen worden verschillen. Dit komt verderop uitgebreid aan bod.
3. MSIL kent alleen het equivalent van Java ‘static’ geneste classes en niet de niet ‘static’ innerclasses die in Java gedefinieerd kunnen worden.
4. Classes kunnen in MSIL een speciale functie hebben binnen het systeem, wat aangegeven wordt door het supertype. Voorbeelden hiervan zijn de structs, enumerations en attributes.

Features

Een duidelijk verschil is dat Java geen definitie van properties en events kent of iets wat daarop lijkt. Methodes van MSIL en Java komen voor een groot deel overeen, maar er zijn twee belangrijke verschillen:

1. De tags die in Java en MSIL op methodes kunnen worden gedefinieerd verschillen
2. Java methodes zijn standaard als ‘virtual’ gedefinieerd. Dit houdt in dat ze door middel van inheritance tussen types geherdefinieerd kunnen worden. In de MSIL moet echter expliciet staan aangegeven dat een methode geherdefinieerd mag worden, of dat een methode een herdefinitie van een andere methode is. Ter verduidelijking volgt er op de volgende pagina een voorbeeld.

Voorbeeld van virtual methodes in de MSIL:



Figuur 5.2.3.1: Animal is de superclass van Dog en Cat. Beide subclasses definiëren hun eigen makesSound() methode.

De headers van de methodes staan als volgt in MSIL gedefinieerd:

In Animal:
```
.method public hidebysig newslot virtual instance string makesSound()
```

In Dog:
```
.method public hidebysig virtual instance string makesSound()
```

In Cat:
```
.method public hidebysig instance string makesSound()
```

De tags 'newslot' en 'virtual' in Animal geven aan dat de methode makesSound() geherdefinieerd kan worden. In Dog wordt aangegeven door middel van de tag 'virtual', dat zijn methode makesSound() ook daadwerkelijk de methode in zijn superclass herdefinieert. In Cat daarentegen herdefinieert de methode makesSound() de methode in zijn parent niet, maar wordt de methode als een nieuwe methode behandeld.

```
Animal d = new Dog();
Animal c = new Cat();
Console.WriteLine(d.makesSound());
Console.WriteLine(c.makesSound());
```

Voor de bovenstaande code zou dit betekenen dat bij het aanroepen van `d.makesSound()` de makesSound() van Dog aangeroepen wordt, maar bij `c.makesSound()` de makesSound() van Animal.

Fields in Java en MSIL zijn vrijwel identiek, behalve als het gaat om het definiëren van constanten. In Java kunnen constanten aangegeven worden door het field als 'final' of 'final static' te definiëren, in MSIL geeft men dit aan door 'initonly' of 'static literal'. In MSIL kent men twee soorten constanten:

1. Constanten die runtime hun waarde krijgen, zijn als 'static literal' gedefinieerd.
2. Constanten die compile time hun waarde krijgen, zijn als 'initonly' gedefinieerd.

Dit verschil is voor RevJava echter niet interessant. We willen in RevJava op een generieke manier gaan bijhouden of een field een constante is, zodat de critics en properties geen rekening hoeven te houden met de taal waarop ze werken.

Als laatste kan er in MSIL nog gebruik worden gemaakt van pointers. Dit is aan de MSIL toegevoegd zodat ook C++ code naar MSIL gecompileerd kan worden. Java kent geen pointers, we zullen een eenvoudige manier moeten zoeken om pointers in het model toe te voegen, aangezien het wel interessante informatie is om bij te houden.

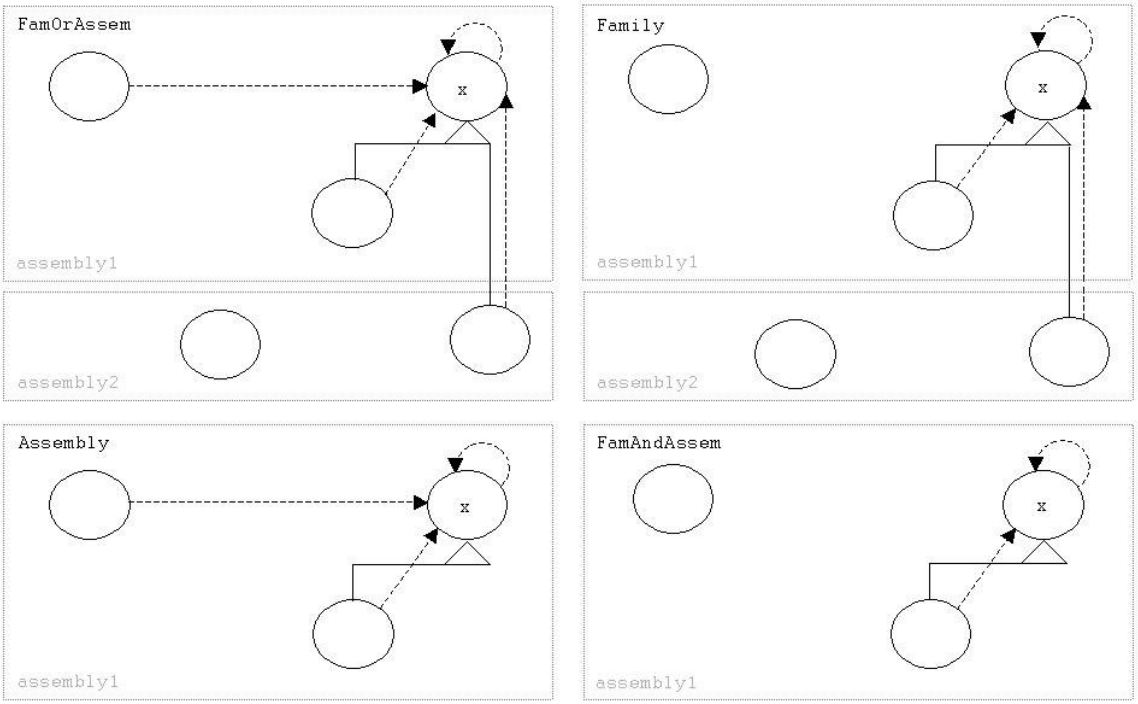
Modifiers

Types en features kunnen tags op zich gedeclareerd hebben in Java en ook in de MSIL. In de MSIL worden deze tags ‘modifiers’ genoemd. In de MSIL komen een groot aantal modifiers overeen met de modifiers die in Java gebruikt worden. De betekenis kan echter een klein beetje afwijken van de betekenis in Java.

De Java modifier 'protected' bestaat in de MSIL niet. Er zijn wel andere modifiers in de MSIL die ongeveer hetzelfde bereiken:

- Family
- Assembly
- FamAndAssem (Family And Assembly)
- FamOrAssem (Family Or Assembly)

De betekenis van deze modifiers wordt hieronder weergegeven.



Figuur 5.2.3.2: Objecten (circels) hebben access tot Feature 'x' wanneer ze een stippelpijl hebben naar het object dat 'x' bevat.

De Java ‘protected’ members komen wat betreft betekenis overeen met ‘FamOrAssem’, ‘package protected’ of ‘package’ members komen overeen met ‘Assembly’. Classes binnen de MSIL kunnen als ‘public’ en ‘private’ gedefinieerd worden. In het eerste geval kunnen ze vanuit een andere assembly gebruikt worden, in het tweede geval niet. Geneste classes kunnen ook als ‘family’,’assembly’,’famorassem’,’famandassem’ gedefinieerd worden. Geneste classes hebben alleen toegang tot de static gedefinieerde features van hun omvattende class.

Een opvallend verschil tussen Java en de MSIL is dat Java het protection level afstemt op packages, terwijl de MSIL dit doet op de assembly. Op dit moment negeren we dit verschil aangezien het voor de critics in RevJava geen verschil maakt.

Wanneer men in Java aan wil geven dat van een type of feature geen overerving mag bestaan kan men dit type ‘final’ definiëren. In de MSIL gaat dit met ‘sealed’ voor types en met ‘final’ voor features.

De verdere verschillen in de betekenis van modifiers komt aan bod als we het nieuwe metamodel opstellen en we bepaald hebben welke modifiers we bij willen houden.

5.3 Een kritische blik op Microsoft .Net

Microsoft introduceerde .net als een kleine revolutie op het gebied van softwareontwikkeling. Met behulp van .net zou men in staat zijn om op een gemakkelijke manier vele soorten software te ontwikkelen, ongeacht in welke programmeertaal men de software wilde schrijven of het platform waar de software op moest draaien.

Achteraf gezien is de revolutie niet zo groot als Microsoft misschien gehoopt had. Het .net platform is inderdaad breed georiënteerd en bevat ondersteuning voor het ontwikkelen van serverapplicaties, webapplicaties en de applicaties voor op de desktop. Het probleem is echter, dat deze veelzijdigheid voor een hoop verwarring zorgt en veel ontwikkelaars niet precies weten wat ze met .net aanmoeten. Ook wordt er veel verwarring veroorzaakt door het feit dat veel bestaande programma's opeens een .net extensie hebben gekregen, maar er niets aan de programma's veranderd lijkt te zijn. Microsoft heeft dit zelf blijkbaar ook ingezien, want de term .net wordt met de introductie van Windows Server 2003 alleen nog gebruikt voor het aangeven van webservices en Visual Studio .Net.

Daarbij is de platformonafhankelijkheid van .net eigenlijk ook maar ten dele waar: wanneer men een applicatie wil draaien op een ander platform dan Windows, zal men zelf de gehele CLR ervoor moeten schrijven, aangezien Microsoft zelf alleen de CLR voor Windows levert.

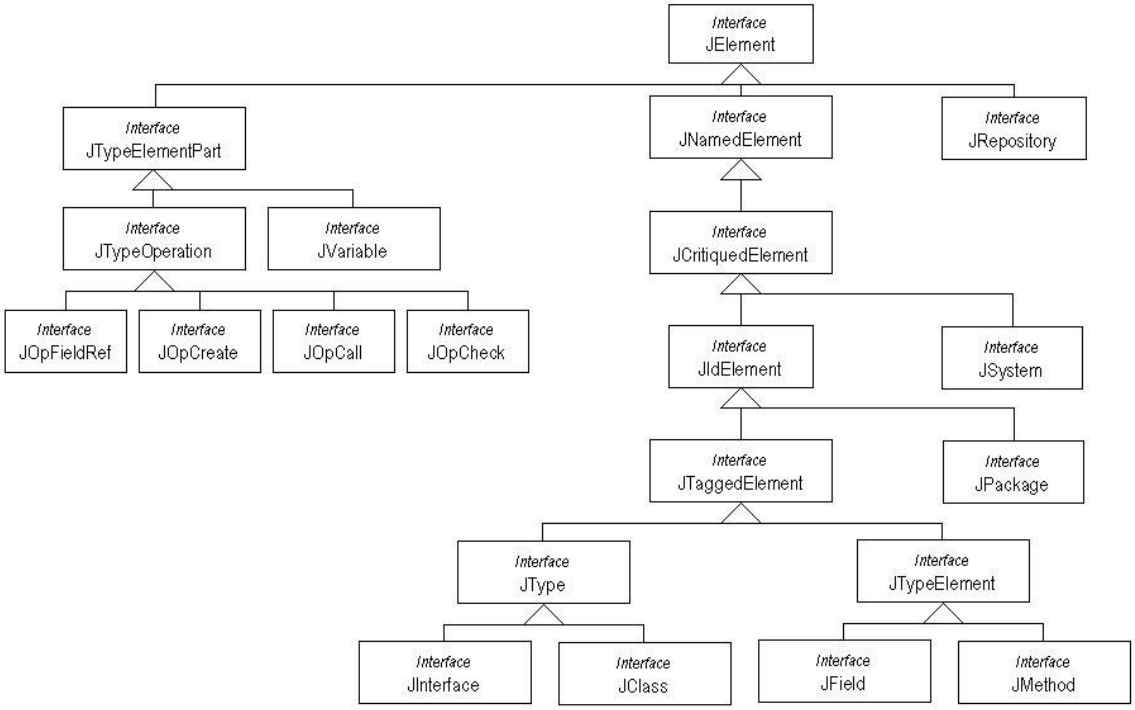
Voor paragraaf 5.2 en 5.3 is gebruik gemaakt van de literatuur genoemd bij: [26] [27] [28] [29] [31] [32] [33] [34] [37]

6. Aanpassingen aan het metamodel

Dit hoofdstuk beschrijft de aanpassingen die we aan het metamodel van RevJava gemaakt hebben om ook de MSIL in te kunnen lezen en te kunnen analyseren. Eerst zal het oude metamodel behandeld worden en aangegeven worden wat we daaraan veranderd hebben. Vervolgens worden de relaties in het nieuwe metamodel beschreven en als laatste zal de implementatie van het nieuwe metamodel uitgebreid aan bod komen.

6.1 Veranderingen aan het oude metamodel

In hoofdstuk 3 is het metamodel al kort behandeld, maar slechts het generieke gedeelte van het model en een gedeelte van de implementatie ervan. Hieronder staat het originele metamodel van RevJava:



Figuur 6.1.1: het originele metamodel van RevJava

We besloten te beginnen met het aanpassen van dit model en later te bepalen welke elementen een implementatie zouden krijgen. We hebben bij het ontwerpen van het metamodel rekening gehouden met de volgende punten:

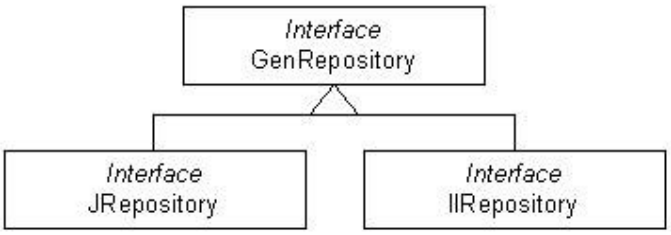
1. We moesten het model zo generiek mogelijk ontwerpen, zodat de critics in RevJava zo veel mogelijk op het generieke niveau uitgevoerd kunnen worden. Op deze manier is het model in de toekomst gemakkelijk uit te breiden met bijvoorbeeld een andere programmeertaal.
2. In de documentatie over RevJava stonden een aantal mogelijke verbeteringen voor het metamodel. We hebben enkelen daarvan doorgevoerd in het nieuwe metamodel.
3. Er moest rekening gehouden worden met het feit dat er in de toekomst misschien ook andere informatie dan typegerelateerde informatie opgeslagen moet kunnen worden.

Aangezien het metamodel niet meer alleen gebruikt zou gaan worden voor Java, hebben we er voor gekozen om het voorvoegsel ‘J’ bij de elementen weg te laten. De elementen die later ook nog gespecialiseerd werden naar een MSIL- en Java-element hebben ‘Gen’ als voorvoegsel gekregen. Een MSIL-element heeft als voorvoegsel ‘Il’ en een Java-element ‘J’.

De aangepaste elementen zullen topdown behandeld worden.

JRepository

De repository houdt alle geladen systemen en de types in die systemen bij. Deze systemen zijn een instantie van JSystem. We hebben een superclass 'GenRepository' gedefinieerd en deze als subclasses 'IlRepository' en 'JRepository' gegeven:

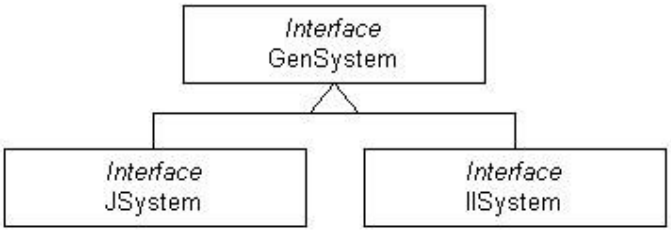


Figuur 6.1.2: GenRepository

De GenRepository zal gebruik maken van andere Gen- elementen. Wanneer het gebruik hiervan niet mogelijk is, maar er specifieke Java of MSIL elementen vereist zijn, zullen de subclasses dit moeten definiëren.

JSystem

We hebben een GenSystem aangemaakt die als subclasses IlSystem en Jsystem heeft:



Figuur 6.1.3: GenSystem

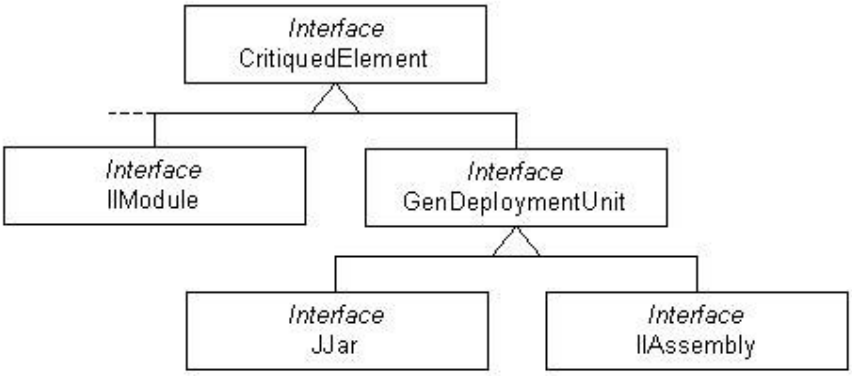
Een system heeft referenties naar packages en deploymentunits. Het GenSystem maakt gebruik van Gen- elementen, de JSystem en IlSystem kunnen methodes definiëren op Java of MSIL specifieke elementen.

GenDeploymentUnit en IlModule

GenDeploymentUnit en IlModule zijn nieuwe elementen in het metamodel. Een deploymentunit moet gezien worden als de file waarin de programma-informatie staat die in het metamodel ingelezen wordt. In Java is dit typisch een Jar-file en in de MSIL een assembly.

We hebben GenDeploymentUnit toegevoegd om het in de toekomst mogelijk te maken dat er ook gemakkelijk sourcecode bij elementen bijgehouden kan worden. Op dit moment heeft de GenDeploymentUnit nog weinig betekenis in het model.

Een GenDeploymentUnit kan GenTypes en GenNamespaces bevatten. Een IlAssembly kan ook IlModules bevatten: één module met zijn eigen MSIL code en een Vector met externe modules. De types in een assembly zijn dan ook een verzameling van alle types in de modules van die assembly. Dit geldt ook voor de namespaces in een assembly. Globale fields en methodes worden niet ingelezen in het metamodel.

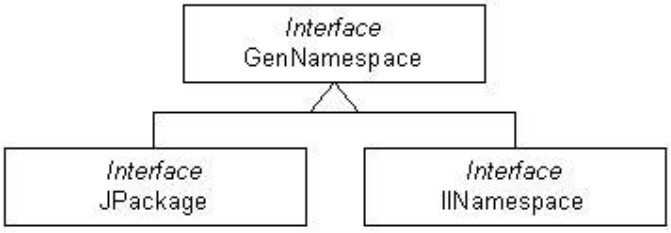


Figuur 6.1.4: GenDeploymentUnit en IlModule

GenDeploymentUnit en IlModule zijn subclasses van CritiquedElement geworden, zodat er ook critics op gedefinieerd zouden kunnen worden. Een IlModule wordt niet beschouwd als een deploymentunit, omdat een losse module niet echt betekenis heeft. Wanneer er een losse module ingelezen wordt, wordt er een default assembly aangemaakt om alle informatie in op te slaan.

JPackage

In Java bestaan packages en in MSIL namespaces. Deze hebben we op een gelijk niveau in het model gezet met als superclass GenNamespace. De naam GenNamespace is gekozen omdat 'package' een Java specifieke benaming is en 'namespace' algemener is.



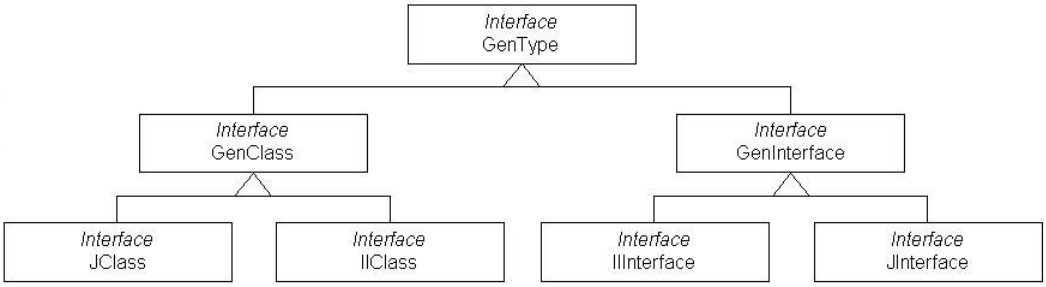
Figuur 6.1.5: GenNamespace

Een GenNamespace kan de volgende elementen bevatten:

- GenTypes
- GenNamespaces

JType

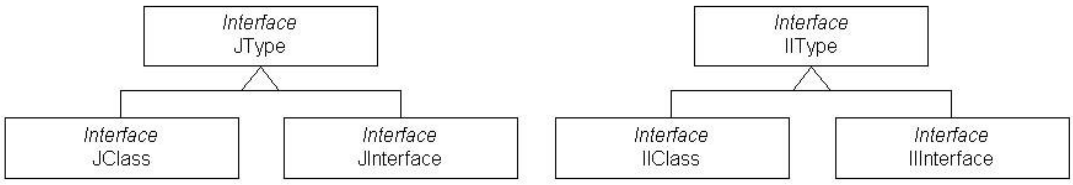
We hebben een class GenType gedefinieerd die als subclasses GenClass en GenInterface heeft. Deze twee subclasses zijn beiden gespecialiseerd naar een Java en een MSIL element.



Figuur 6.1.6: GenType

Door RevJava heen wordt echter intensief gebruik gemaakt van het begrip JType. Aangezien we verwachten dat RevJava op verschillende plekken in het programma zou moeten weten of

het met MSIL of Java elementen aan het werk is, hebben we de begrippen JType en IlType toegevoegd in het model, waardoor Jclass/IlClass en JInterface/IlInterface allen twee supertypes hebben:

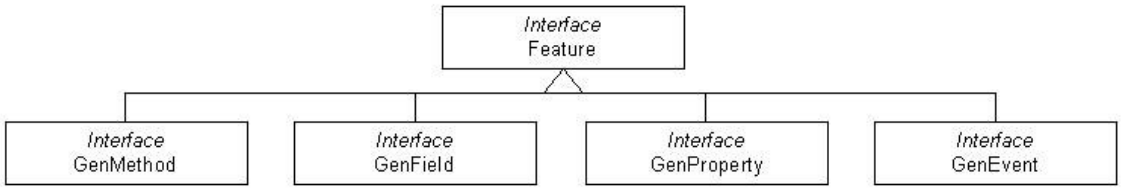


Figuur 6.1.7: JType en IlType

We hadden natuurlijk JType en IlType als subclasses van GenType kunnen definiëren en de GenClass en GenInterface weg kunnen laten. We hebben echter voor deze opzet gekozen, omdat we de mogelijkheid willen openhouden andere types toe te voegen. Dit zou bijvoorbeeld kunnen gebeuren als er een andere taal ingelezen zou worden die gebruik maakt van andere types dan classes en interfaces.

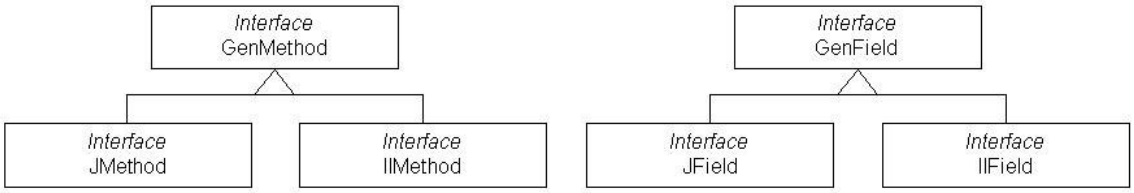
JTypeElement

We hebben ten eerste de naam 'JTypeElement' veranderd naar 'Feature'. Binnen de MSIL kunnen features ook properties of events zijn, daarom hebben we de subclasses GenProperty en GenEvent toegevoegd onder Feature.



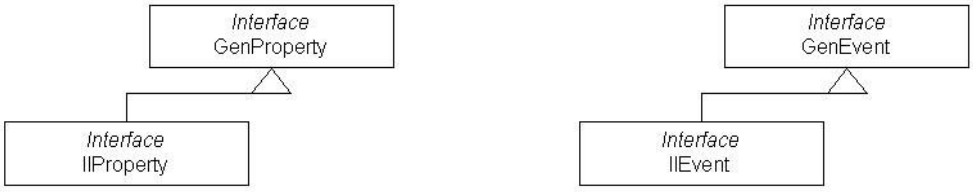
Figuur 6.1.8: Feature

Onder GenMethod en GenField zijn de betreffende Java en MSIL subclasses toegevoegd:



Figuur 6.1.9: GenMethod en GenField

Aangezien Java geen properties en events kent, zijn onder GenProperty en GenEvent alleen subclasses toegevoegd voor de MSIL. Deze subclasses lijken daarom misschien overbodig, maar we houden er rekening mee dat er nog andere talen toegevoegd kunnen worden die properties en events bevatten.

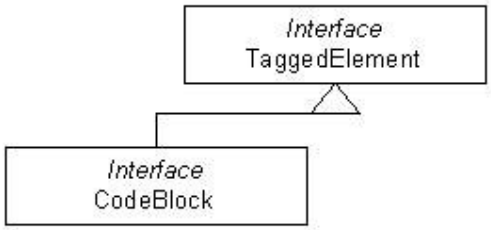


Figuur 6.1.10: GenProperty en GenEvent

JTypeElementPart

De definitie van een JTypeElementPart is geheel verwijderd. We hebben dit gedeelte op een andere manier opgezet:

We hebben een class CodeBlock toegevoegd. Een methode bevat een codeblock, en een codeblock heeft referenties naar de variabelen en operaties van een methode:

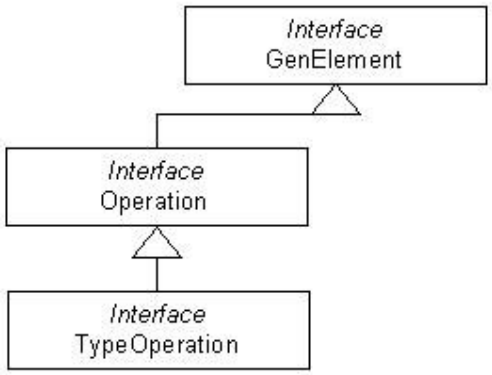


Figuur 6.1.11: CodeBlock

Een codeblock kan bepaalde modifiers op zich gedefinieerd hebben, zoals 'synchronized', en is daarom een subclass van TaggedElement. Hierdoor is CodeBlock via inheritance ook een NamedElement, wat eigenlijk niet juist is aangezien een CodeBlock geen naam heeft. Het leek ons echter belangrijk aan te geven dat een CodeBlock tags kan bevatten, daarom is het toch onder TaggedElement toegevoegd.

In de toekomst kan een codeblock gebruikt gaan worden om sourcecode bij een methode bij te houden of om statische blokken code in types aan te geven.

De class 'JTypeOperation' heet nu 'TypeOperation' en heeft als superclass 'Operation' gekregen:

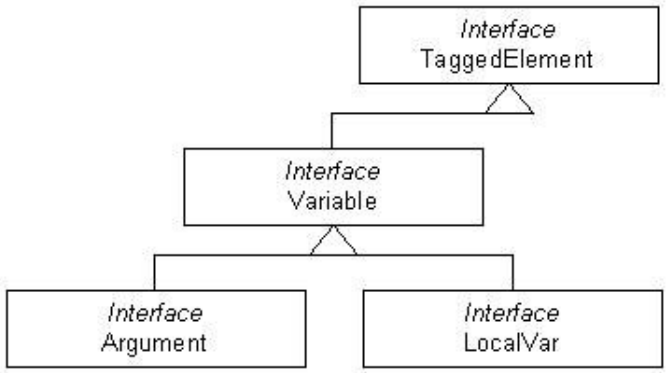


Figuur 6.1.12: Operation en TypeOperation

De class Operation is toegevoegd, omdat er in de toekomst misschien wel andere soorten operaties dan typegerelateerde operaties ingelezen gaan worden. We kunnen hierbij bijvoorbeeld denken aan returnstatements en dergelijke.

JVariable heet nu 'Variable' en heeft twee subclasses: Argument en LocalVar. Tevens is Variable een TaggedElement geworden om de volgende redenen:

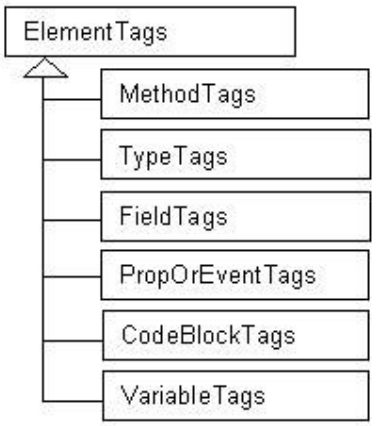
- In de MSIl komen pointers voor.
- In Java kunnen argumenten een modifier meekrijgen. Dit wordt echter door de parser van RevJava niet herkent en opgeslagen.



Figuur 6.1.13: Variable

JTags

TaggedElements zijn elementen die bepaalde modifiers op zich gedefinieerd kunnen hebben, zoals bij voorbeeld 'public' of 'static'. De volgende elementen zijn subclasses van TaggedElement: GenType, Feature, CodeBlock en Variable. In JTags stond een lijst van de tags die in RevJava bijgehouden werd. Deze lijst was voor ieder element gelijk. Niet ieder TaggedElement heeft echter dezelfde modifiers op zich gedefinieerd. We hebben een structuur gedefinieerd waarin de modifiers voor de verschillende TaggedElements bijgehouden kunnen worden:



Figuur 6.1.14: ElementTags

De reorganisatie had verschillende redenen:

- Nu is vanuit het model duidelijk dat niet ieder element dezelfde modifiers kan hebben.
- We willen de modifiers generiek definiëren, zodat ze voor zowel MSIL als Java gelden. Op deze manier kan dat wat overzichtelijker.
- Het kan geheugen besparen als het metamodel opgebouwd wordt, aangezien niet alle lijsten van modifiers even lang zijn.

Het is belangrijk om uit te zoeken wat de verschillen in betekenis en syntax zijn tussen de modifiers in MSIL en Java. Aangezien de critics geacht worden te werken op het generieke deel van het metamodel, moeten de verschillen op een generieke manier in het metamodel gezet worden.

De volgende modifiers zijn gelijkwaardig binnen MSIL en Java:

	Class	Method	Field	Property	Event	Interface
public	x	x	x	x	x	x
private	x	x	x	x	x	x
abstract	x	x		x	x	
static	x	x	x	x	x	
volatile			x			
virtual		x		x	x	
synchronized		x				

Figuur 6.1.15: gelijkwaardige modifiers in Java en MSIL. In de tabel staat ook meteen aangegeven op welke elementen een modifier werkzaam kan zijn door middel van een 'x'.

De volgende modifiers moeten eerst geïnterpreteerd worden:

- Protected members: een Java member kan als protected gedefinieerd worden. De MSIL kent deze modifier niet, maar heeft wel een gelijkwaardige modifier, namelijk 'famorassem'. Members die 'package protected' zijn in Java komen overeen met 'assembly' in de MSIL.
- Constanten: in Java wordt een constante aangegeven door een field 'final static' of 'final' te definiëren. In de MSIL kan dit met 'initonly' of 'static literal'
- Geen inheritance: wanneer in Java aangegeven wordt dat van een element geen inheritance mogelijk is, gebruikt men hiervoor de modifier 'final'. In de MSIL wordt 'sealed' gebruikt om voor classes aan te geven dat er geen inheritance mogelijk is, en 'final' voor methodes.

Het is mogelijk de taalspecifieke verschillen voor de critics te verbergen, door methodes in het metamodel te zetten die zorgen dat de informatie op een generieke manier op te vragen is. Zo kunnen we een methode 'isConst()' definiëren die voor een field ophaalt of het een constante is. De methode 'isConst()' kan dan in JField opvragen of het field de modifier 'final' of 'final' en 'static' heeft en in IlField of het field de modifier 'initonly' of 'static' en 'literal' heeft.

De modifiers zijn toepasbaar op de volgende elementen:

	Class	Method	Field	Property	Event	Interface
constant			x			
final / sealed	X	x		x	x	
assembly / package protected	X	x	x	x	x	x
famorassem / protected	X	x	x	x	x	x

Figuur 6.1.16: modifiers die geïnterpreteerd moeten worden. Een 'x' geeft aan op welk element een modifier werkzaam kan zijn.

Sommige modifiers zijn specifiek voor de programmeertaal.

	Class	Method	Field	Property	Event	Interface
family	x	x	x	x	x	x
famandassem	x	x	x	x	x	x
nested	x					x

Figuur 6.1.17: MSIL specifieke modifiers.

Wanneer een class als 'Assembly', 'FamOrAssem', 'FamAndAssem' of 'Family' wordt aangeduid betekent het dat er sprake is van een geneste class. De modifier 'nested' komt in Java niet voor.

	Class	Method	Field	Property	Event	Interface
transient			x			

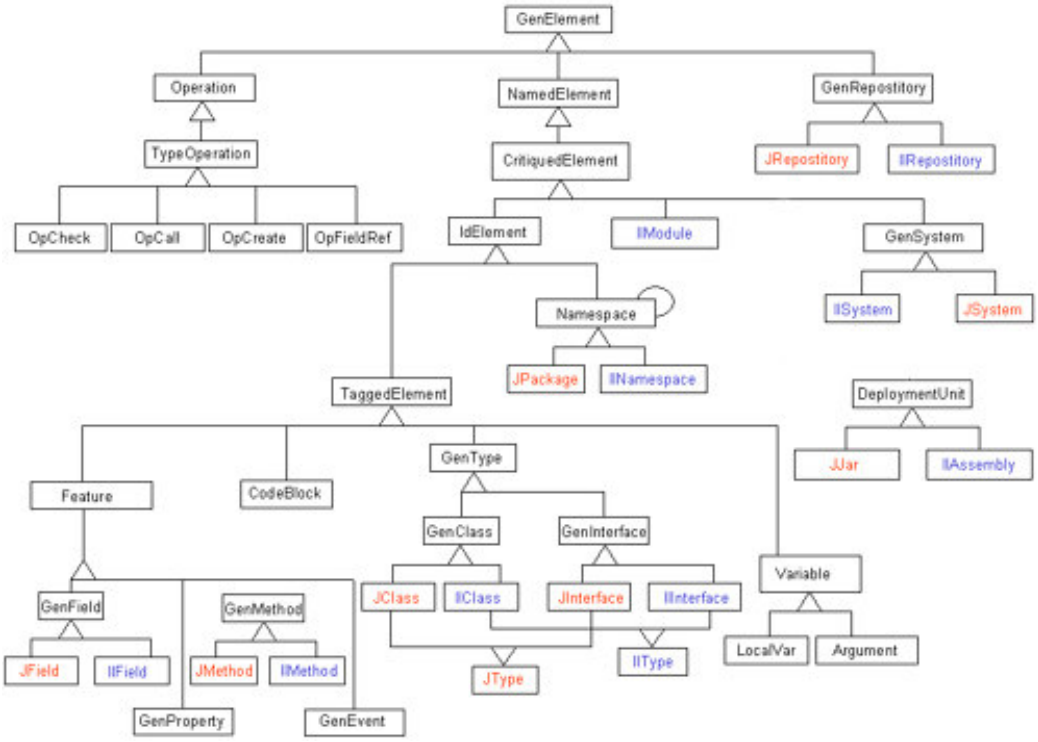
Figuur 6.1.18: Java specifieke modifiers.

De modifier transient geeft aan dat een field niet tot de persistente staat van een object behoort. Transient kan gebruikt worden wanneer de waarde van een field alleen tijdelijk of

lokaal geldt. De waarde ervan wordt niet bewaard na een transfer van de class over het internet of wanneer het object naar een file geschreven wordt.

Het aangepaste metamodel

Hieronder staat het gehele aangepaste metamodel:



Figuur 6.1.19: het aangepaste metamodel

Het abstracte gedeelte van het model is grotendeels gelijk gebleven. De elementen CodeBlock en Variable kunnen als een uitbreiding gezien worden van het abstracte model, het begrip JTypeElement is verloren gegaan. De specifieke taalelementen vormen de bladeren van het model, daarboven zorgen de Gen-elementen ervoor dat de critics op een generiek niveau kunnen werken.

Bij het ontwerp van het nieuwe model zijn er twee items toegevoegd die in de documentatie van RevJava aangegeven stonden als mogelijke verbeteringen voor het metamodel:

- CodeBlock
- Geneste namespaces

Het toevoegen van geneste namespaces in het model was sowieso noodzakelijk, aangezien in de MSIL geneste namespaces voor kunnen komen. Het codeblock zorgt ervoor dat in de toekomst op een gemakkelijke manier extra (niet typegerelateerde) informatie bij methodes opgeslagen kan worden.

De relaties in het nieuwe model worden behandeld in de volgende paragraaf.

Voor deze paragraaf is gebruik gemaakt van de literatuur genoemd bij: [38] [39] [40] [41]

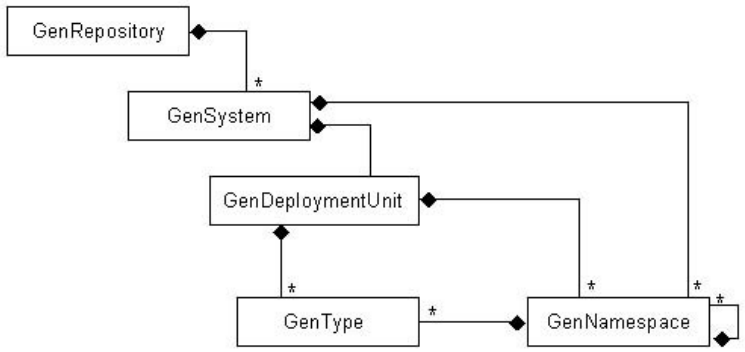
6.2. Relaties in het nieuwe metamodel

Het metamodel is opgebouwd uit generieke en taalspecifieke elementen. De generieke elementen binnen het model specificeren de basisrelaties die er binnen het metamodel gelden. Wanneer er Java code wordt ingelezen worden uitsluitend de basisrelaties gebruikt in het metamodel. De MSIL daarentegen definieert in sommige Il-elementen nog extra relaties. Deze definities zijn gedefinieerd in de taalspecifieke elementen van het metamodel.

6.2.1. Relaties in het generieke model

De relaties in het generieke model worden gevormd door de Gen-elementen van het metamodel. De relaties worden topdown behandeld.

GenDeploymentUnit en GenNamespace



Figuur 6.2.1.1: GenDeploymentUnit en GenNamespace

Een GenRepository heeft een aantal hashtables voor geladen elementen:

- availabletypes: alle geladen types.
- availablenamespaces: alle geladen namespaces.
- availablesystems: de geladen systems.

Een GenSystem heeft referenties naar de deploymentunits en de namespaces daarin.

Een GenDeploymentUnit heeft de volgende members:

- GenType[] types: de types in de deploymentunit.
- GenNamespace[] namespaces: de namespaces in de deploymentunit.
- GenSystem holder: het system waarin de deploymentunit bevat zit.

Een GenNamespace heeft de volgende members:

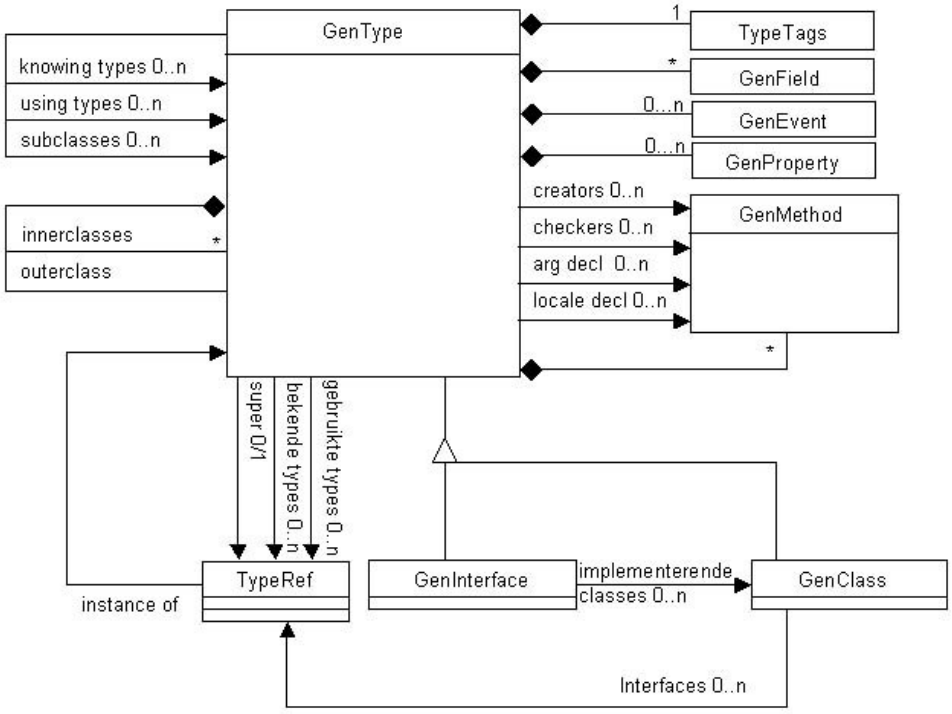
- Vector elements: de types in de namespace.
- GenDeploymentUnit holder: de deploymentunit waarin de namespace zit.
- GenNamespace[] childnamespaces: de namespaces welke in de namespace genest zijn.
- GenNamespace parent: de namespace waarin de namespace zelf gedefinieerd is.

In Java heeft het aangeven dat een package genest is geen echte betekenis.
Childnamespaces kunnen niet zomaar types in hun parent benaderen, maar moeten de types eerst importeren.

Type
Een GenType heeft de volgende members:

- GenNameSpace holder: de namespace waarin het type bevat is.
- GenField[] fields: de fields in het type.
- GenEvent[] events: de events in het type.
- GenProperty[] properties: de properties in het type.

- GenMethod[] methods: de methodes in het type.
- GenMethod[] creators: de methodes die een nieuwe instantie van het type aanmaken.
- GenMethod[] checkers: de methodes die het type aanroepen.
- GenMethod[] argDecls: de argument declaraties in het type.
- GenMethod[] localDecls: de lokale declaraties in het type.
- GenType[] knowingTypes: de andere types die het type kennen.
- GenType[] usingTypes: de andere types die het type gebruiken.
- GenType[] subclasses: de subclasses van een type.
- GenType[] innerClasses: de innerclasses van het type.
- GenType outerClass: de outerclass van het type. Deze is null als het type geen geneste class is.
- TypeRef[] usedTypes: alle andere types die door het type gebruikt worden.
- TypeRef[] knownTypes: alle andere types die het type kent.
- TypeRef superClass: een referentie naar het supertype van het type.
- TypeRef[] interfaces: de interfaces die door het type geïmplementeerd worden.
- JESet implementors: de types waardoor het type geïmplementeerd wordt.
- TypeTags tags: de modifiers die op het type gedefinieerd zijn.

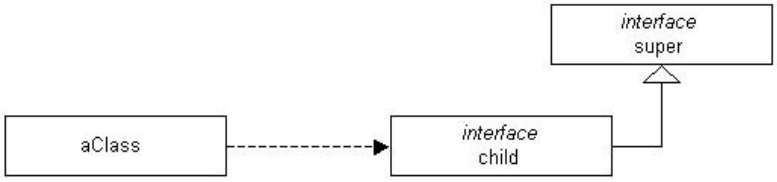


Figuur 6.2.1.2: GenType

Een type in Java kan geen properties en events definiëren, maar omdat we ook properties en events op een generieke manier willen behandelen in het geval er nog andere talen in het model worden toegevoegd, staan ze wel in GenType.

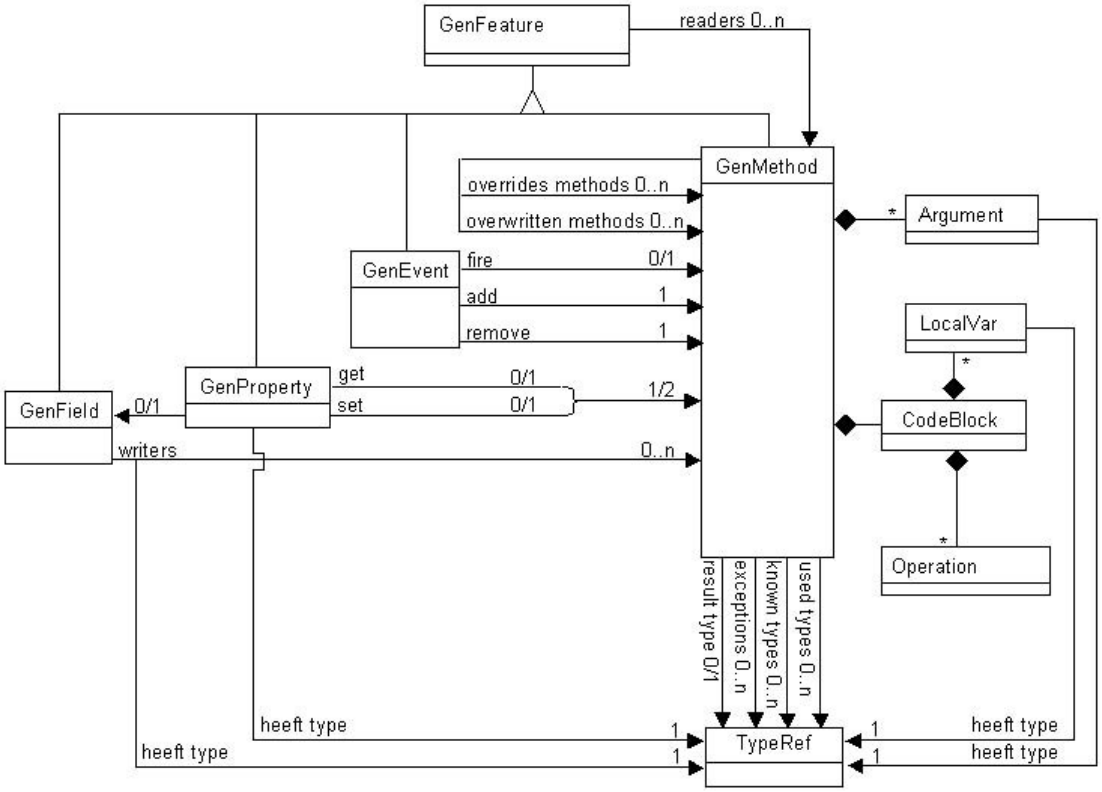
Een GenType kan slechts één supertype hebben, we hebben besloten om multiple inheritance niet toe te voegen aan RevJava.

Het lijkt vreemd dat een GenType definieert welke interfaces er door een type geïmplementeerd worden, want dit houdt in dat ook een interface een type kan implementeren. Dit was echter al zo in het oude metamodel om de volgende reden: als een interface erft van een superinterface, implementeert de interface eigenlijk de methodes in de superinterface.



Figuur 6.2.1.3: class aClass implementeert interface child en super, omdat child erft van super. Class aClass is zich hiervan niet bewust en eigenlijk implementeert child dus super.

GenFeature



Figuur 6.2.1.4: GenFeature

Een Feature heeft de volgende fields:

- GenType holder: het type waarin de feature bevat is.
- GenMethod reader: een feature wordt door 0 of meer methodes gelezen.

GenMethod, GenField, GenEvent en GenProperty zijn subclasses van Feature en bevatten de volgende members:

GenMethod:

- GenMethod overwrittenMethod: de methode die door deze methode wordt geherdefinieerd.
- GenMethod overridesMethod: de methodes die deze methode herdefiniëren.
- TypeRef resultType: het returntype van de methode.
- TypeRef[] exceptions: de exceptions die afgevangen worden in de methode.
- TypeRef[] usedTypes: de types die in deze methode gebruikt worden.
- TypeRef[] knownTypes: de types die bekend zijn in de methode.
- GenArgument[] arguments: de parameters van de methode.

- CodeBlock codeblock: een codeblock bevat de lokale variabelen en operaties van de methode.

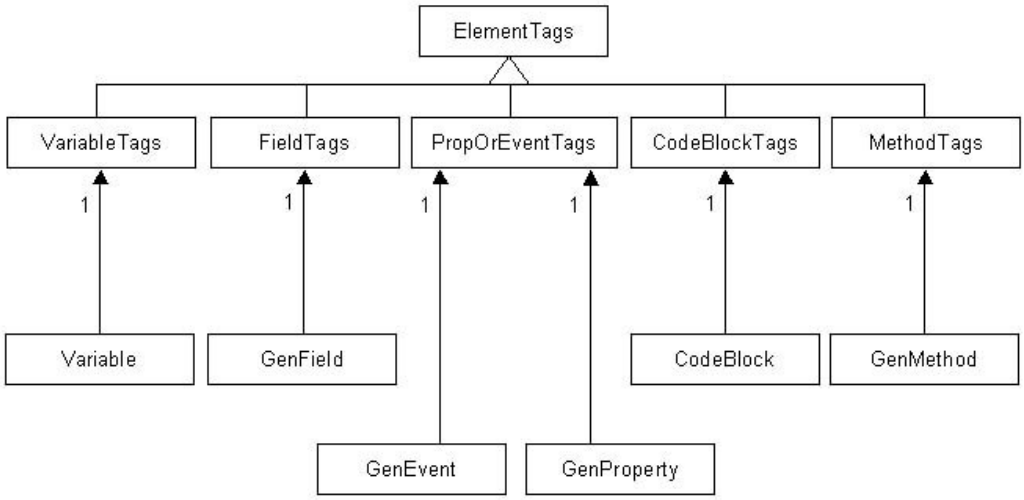
GenField:
- TypeRef type: het type van het field.
- GenMethod[] writers: de methodes die dit field veranderen.

GenProperty:
- TypeRef type: het type van de property.
- GenField field: een property kan gedefinieerd zijn als getter en/of setter voor een field, maar het field kan ook null zijn.
- GenMethod get / GenMethod set: een property moet minimaal één van deze methodes definiëren.

GenEvent:
- TypeRef type: het type van het Event.
- GenMethod addon / GenMethod removeon: deze methode definieert een event altijd.
- GenMethod fire: dit is een methode die een event optioneel kan definiëren.

Properties en events zijn gespecificeerd in het model zoals ze voorkomen in de MSIL. We nemen aan dat er talen zijn met een soortgelijke constructie, die gebruik kunnen maken van GenEvent of GenProperty. Zowel een event als een property kunnen nog een extra methode definiëren in de MSIL, genaamd 'other'. Het is alleen onduidelijk op welke manier dit zou kunnen. Met C# is het definiëren van de 'other' niet mogelijk.

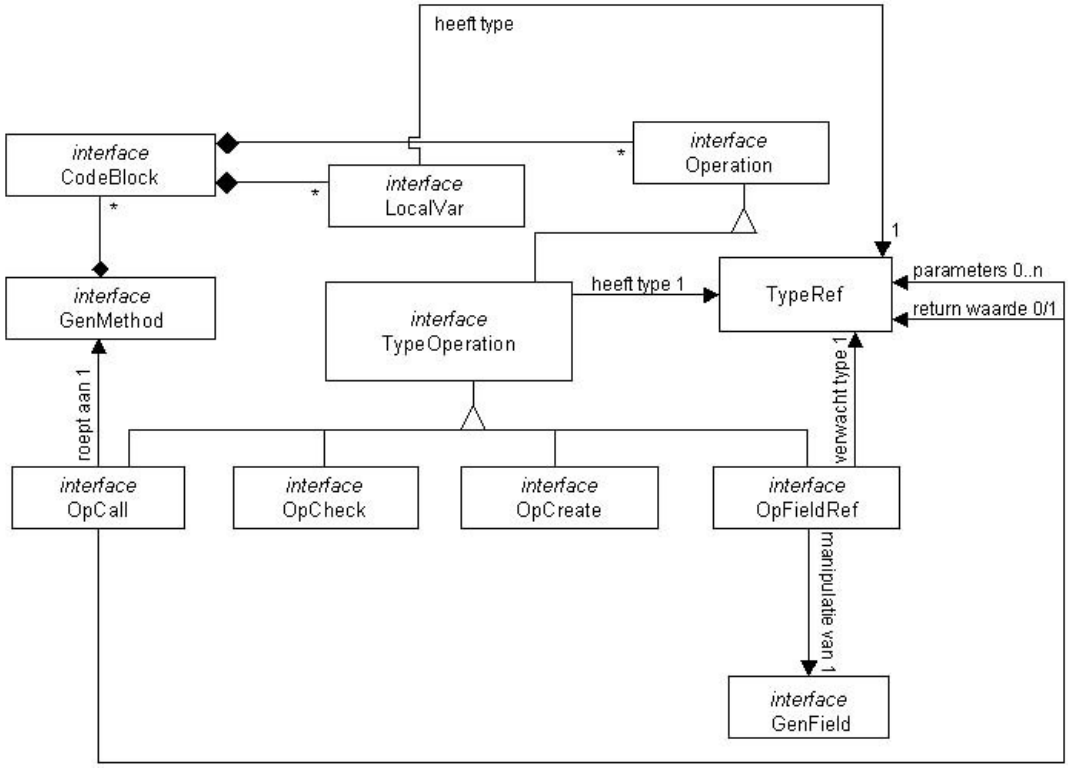
De elementen GenMethod, GenField, GenEvent, GenProperty, CodeBlock en Variable hebben een eigen gespecialiseerde ElementTags, waarin de modifiers op het element bijgehouden worden:



Figuur 6.2.1.5: de modifiers van de verschillende elementen

Generieke elementen
Een aantal elementen is geheel generiek. Deze elementen behoeven geen herdefinitie te krijgen bij het toevoegen van een andere taal. Het betreft de volgende elementen:

- Variable en zijn subclasses Argument en LocalVar
- CodeBlock
- Operation en de subclass TypeOperation
- TypeRef



Figuur 6.2.1.6: CodeBlock, LocalVar, Operation en TypeRef

TypeRef:
Een TypeRef representeert een referentie naar een type. De volgende members zijn in een TypeRef gedefinieerd:

- string name: de naam van het type
- boolean builtin: is het een ingebouwd type (int, boolean enzovoorts)
- boolean ismultiple: is het een collectie
- GenType denotedelement: het type waarnaar de TypeRef verwijst

Variable:
Arguments en LocalVars zijn subclasses van Variable. Een Variable heeft de volgende members:

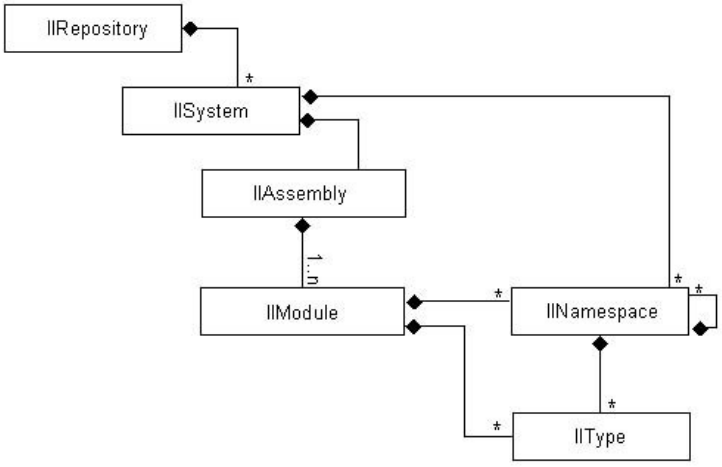
- TypeRef type: het type van de variabele
- GenMethod holder: de methode waarbij de variabele hoort

Een LocalVar heeft daarnaast nog een referentie naar het codeblock van de methode waarin de LocalVar wordt gedeclareerd. In het model staat aangegeven dat een methode meerdere codeblocks kan bevatten. In overleg met Gert hebben we er echter voor gekozen dat wij gebruik zullen maken van één codeblock per methode. Later kan dit gemakkelijk uitgebreid worden, zodat een methode meerdere codeblocks kan bevatten die verschillende modifiers op zich gedefinieerd kunnen hebben.

6.2.2 Het MSIL specifieke gedeelte van het model

De relaties in het metamodel die gelden voor de MSIL elementen wijken op een aantal punten af van de relaties in het generieke model. Het aantal verschillen is echter klein, want we hebben geprobeerd het gehele model zo generiek mogelijk op te zetten.

IlAssembly en IlNamespace



Figuur 6.2.2.1: IlAssembly, IlModule en IlNamespace

Het verschil met de generieke elementen is dat een IlAssembly geen namespaces en types bevat, maar slechts modules. Wanneer aan de assembly de types of namespaces worden opgevraagd, zal hij deze gegevens aan zijn modules opvragen. De types die direct in een IlModule staan zijn eigenlijk bevat in de default namespace.

Een IlNamespace heeft een member van het type IlModule, die bijhoudt in welke module de namespace gedeclareerd is. De members van een IlModule zijn:

- IlAssembly holder: de assembly waarin de module bevat is.
- Vector namespaces: de namespaces in de module.
- Vector types: alle types in de module.

IlClass

In C# kunnen we enumerations, structs en attributes definiëren. In de MSIL zijn deze elementen als classes terug te vinden die als superclass een object uit de System-library hebben, te weten:

- System.Enum voor enumerations.
- System.ValueType voor structs.
- System.Attribute voor attributes.

Om te achterhalen of een class een subclass is van één van deze drie, kunnen we in GenClass methodes definiëren die standaard ‘false’ teruggeven. Deze methodes kunnen we in IlClass herdefiniëren.

6.2.3 Evaluatie van het nieuwe metamodel

Het nieuwe metamodel heeft meer generieke elementen dan het oude metamodel van RevJava. Dit heeft zowel voordelen als nadelen.

De voordelen zijn:

- Het model is gemakkelijk uit te breiden om een extra taal te analyseren, want:
  - de critics werken op het generieke niveau.
  - specifieke taalelementen kunnen als subclass van een generiek element in het metamodel toegevoegd worden.
- Het model is uit te breiden met niet typegerelateerde informatie:
  - Er kan bijvoorbeeld sourcecode van een methode bij een codeblock bijgehouden worden.
- Het model is gemakkelijker uit te breiden met extra elementen, bijvoorbeeld:
  - niet typegerelateerde operaties.
  - meerdere codeblocks bij een methode.

De nadelen zijn:

- Afgeleide properties kunnen voor een bepaalde taal niet van toepassing zijn, zoals Events en Properties voor Java.

Bovenstaande nadelen kunnen overwonnen worden door in de properties en critics een gelaagdheid aan te brengen, zodat er twee soorten properties en critics zijn:

- generieke properties en critcs.
- taalspecifieke properties en critics.

Er moet dan aangegeven worden welke taal er geanalyseerd wordt en met die informatie moet bepaald worden welke properties en critics op de taal van toepassing zijn. Wanneer er dan een extra taal aan het metamodel toegevoegd wordt, kunnen er voor die taal meteen specifieke properties en critics gedefinieerd worden.

7. Implementatie van het aangepaste metamodel

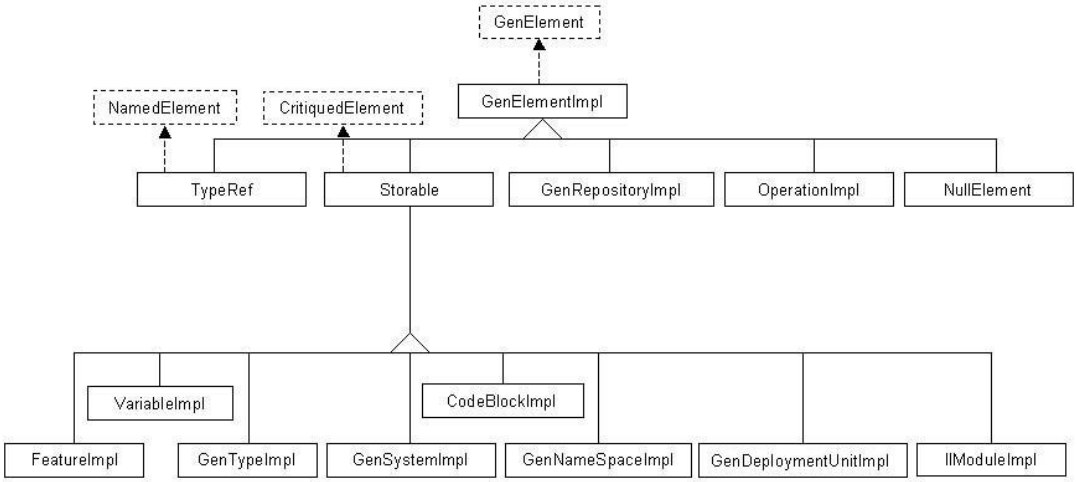
Nadat we het aangepaste metamodel opgesteld hadden, moest dit geïmplementeerd worden. Er moesten een aantal keuzes gemaakt worden:

- Welke elementen krijgen daadwerkelijk een implementatie.
- Hoe verdelen we het gedrag over de elementen.
- In welke packages moeten de elementen komen.

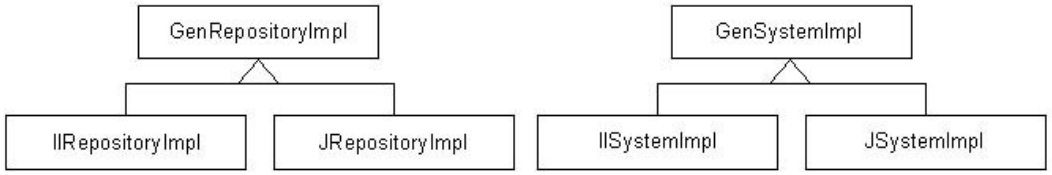
We moesten bij de implementatie opletten dat het geïmplementeerde model gelijk moest zijn aan het ontworpen model. Wanneer dit om praktische redenen niet mogelijk bleek, moest het ontwerp aangepast worden.

7.1 Manier van aanpak

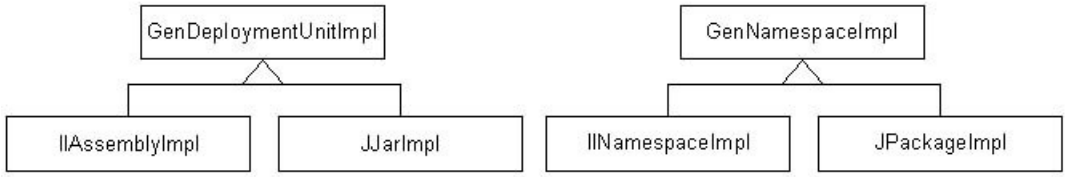
De Gen- elementen uit het metamodel krijgen allen een implementatie-class. We hebben een logische structuur in de naamgeving van de implementatie-classes aangehouden: de naam van de interface die de class implementeert, aangevuld met 'Impl'. Zo wordt bijvoorbeeld de interface 'Operation' geïmplementeerd door 'OperationImpl'.



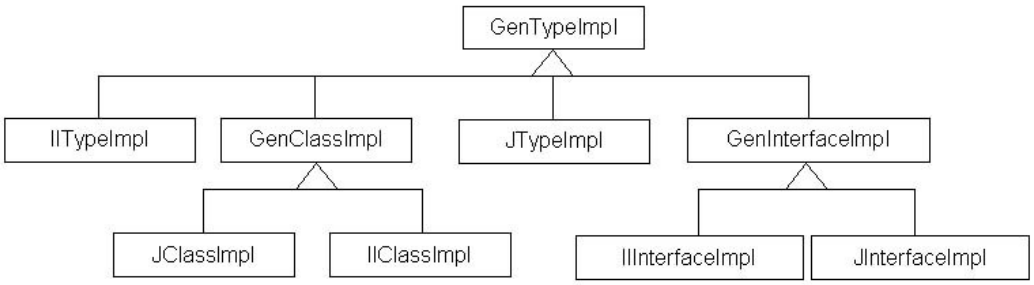
Figuur 7.1.1: het implementatiemodel. Wanneer uit de naam van een class niet blijkt welke interface hij implementeert, staat deze aangegeven. NullElement implementeert geen interface.



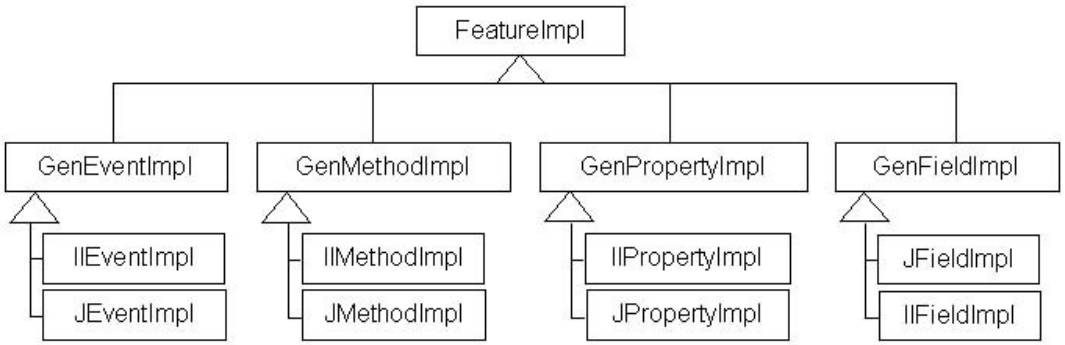
Figuur 7.1.2: de implementatiestructuur van GenRepository en GenSystem met hun subclasses.



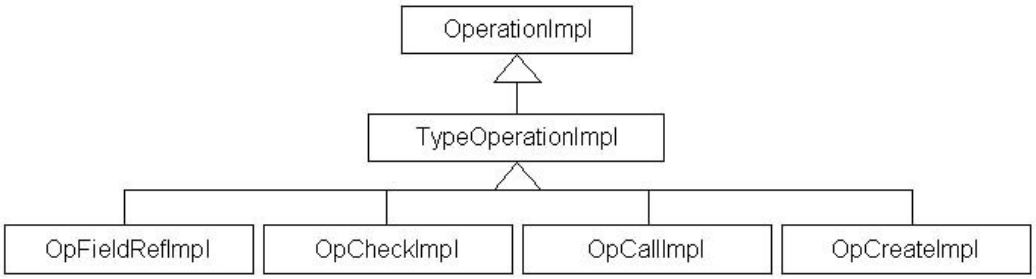
Figuur 7.1.3: de implementatiestructuur van GenDeploymentUnit en GenNamespace met hun subclasses.



Figuur 7.1.4: De implementatiestructuur van GenType en zijn subclasses.



Figuur 7.1.5: de implementatiestructuur van Feature en zijn subclasses.



Figuur 7.1.6: de implementatiestructuur van Operation en zijn subclasses.

Vanuit het opgestelde model was af te leiden dat de Gen- elementen ongeveer dezelfde functionaliteit moesten bieden als de originele J- elementen in het oude model boden. We hebben dan ook de functionaliteit uit de J- elementen in de Gen- elementen gezet. Hierna is RevJava aangepast, zodat het met de Gen- elementen kon werken zoals het met de J- elementen deed. Hierna hebben we de J- en Il- elementen als subclasses van de Gen- elementen toegevoegd aan het programma en waar nodig het programma met de taalspecifieke elementen laten werken.

We hebben de volgende packages gebruikt om de elementen in te definiëren:

- `nl.serc.revs.core.model`: hierin staan de Gen- interfaces gedefinieerd, alsmede de classes die generiek zijn voor het metamodel (TypeRef, Variable).
  - `nl.serc.revs.core.model.impl`: hierin staan de implementatieclasses van `nl.serc.revs.core.model` gedefinieerd.
- `nl.serc.revs.core.model.java`: hierin staan de J- interfaces gedefinieerd.
  - `nl.serc.revs.core.model.java.impl`: hierin staan de implementatieclasses van de J- elementen gedefinieerd.
- `nl.serc.revs.core.model.msil`: hierin staan de Il- interfaces gedefinieerd.

- `nl.serc.revs.core.model.msil.impl`: hierin staan de implementatieclasses van de Il- elementen gedefinieerd.

De eerste twee genoemde packages waren al gedefinieerd. Wij hebben de package-structuur uitgebreid met de taalspecifieke packages.

Toen RevJava naar behoren werkte hebben we de extra elementen toegevoegd, te weten:

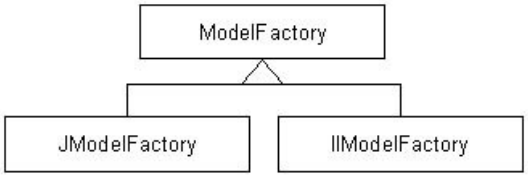
- CodeBlock en CodeBlockImpl.
- GenDeploymentUnit en GendeploymentUnitImpl met hun taalspecifieke subclasses.
- GenProperty en GenPropertyImpl met de taalspecifieke subclasses.
- GenEvent en GenEventImpl met de taalspecifieke subclasses.
- Argument en LocalVar.
- Operation en OperationImpl.
- IlModule.

Het toevoegen van de elementen hield in dat er in de Gen- elementen aanpassingen gemaakt moesten worden, zodat RevJava ook daadwerkelijk gebruik maakte van de toegevoegde elementen. Dit was het gemakkelijkst te realiseren vanuit een werkend programma, vandaar dat we dit pas gedaan hebben toen RevJava werkend was met de Gen- elementen. Tevens is in dit stadium JTags vervangen door ElementTags en zijn subclasses. De elementen waarvoor er een subclass van ElementTags gedefinieerd was hebben deze ook meegekregen als het element aangemaakt wordt.

Na de implementatie van het model hebben we RevJava (versie 0.8.5) gerund en de uitkomsten vergeleken met de uitkomsten die ons programma (met het nieuwe metamodel) gaf. We hebben hiervoor een kleine zelfgemaakte Jar-file gebruikt. De resultaten vertoonden geen verschil.

ModelFactory

Voor het aanmaken van de elementen in het metamodel is er een ModelFactory gedefinieerd. Voor de taalspecifieke elementen zijn er subclasses van ModelFactory gedefinieerd:



Figuur 7.1.7: ModelFactory, JModelFactory en IlModelFactory

De factories staan in de volgende packages gedefinieerd:

- nl.serc.revs.core.model.ModelFactory
- nl.serc.revs.core.model.msil.IlModelFactory
- nl.serc.revs.core.model.java.JModelFactory

In de ModelFactory worden de generieke elementen aangemaakt:

- Variable, Argument en LocalVar
- De TypeOperations :
  - OpCall
  - OpCreate
  - OpCheck
  - OpFieldRef
  - CodeBlock

IlModelFactory en JModelFactory maken de taalspecifieke elementen aan, maar geven deze gecast terug als een GenElement. Dit zorgt ervoor dat de properties en critics op het generieke niveau uitgerekend kunnen worden. De volgende elementen worden aangemaakt in JModelFactory en IlModelFactory:

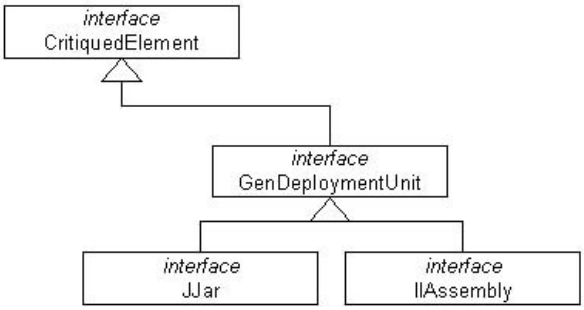
- GenSystems
- GenDeploymentUnits
- GenNamespaces
- GenClasses en GenInterfaces
- GenFields, GenMethods, GenProperties en GenEvents

Wanneer er een andere taal toegevoegd zou worden aan het programma zou deze taal in het package waarin de taalspecifieke modelelementen gedefinieerd staan zijn eigen subclass van ModelFactory kunnen definiëren.

Aanpassingen aan het metamodel tijdens de implementatie

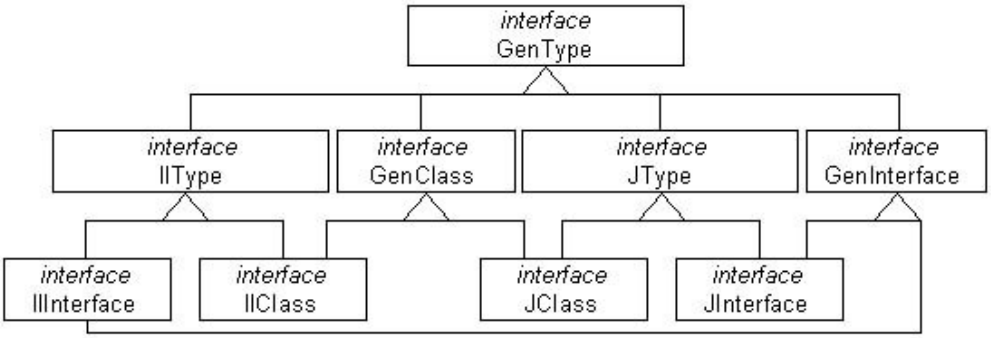
Gedurende de implementatie van het metamodel kwamen er enkele onvolkomenheden in het door ons ontworpen metamodel naar boven. Deze onvolkomenheden zorgden ervoor dat het programma niet goed werkte. We hebben de volgende aanpassingen gemaakt:

- GenDeploymentUnit is een subclass geworden van CritiquedElement:



Figuur 7.1.7: GenDeploymentUnit

- JType en IlType zijn subclasses geworden van GenType. In figuur 7.1.2 was reeds te zien dat de implementatie-classes erven van GenTypeImpl.



Figuur 7.1.8: JType en IlType

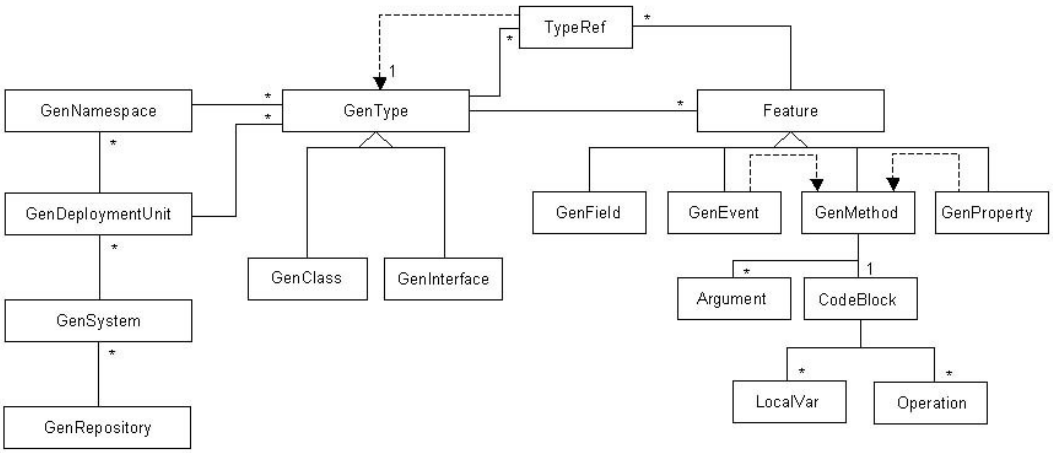
Dubbele code

Tijdens het implementeren van het model hebben we geprobeerd zo min mogelijk dubbele code te introduceren. In enkele gevallen is dit echter niet gelukt:

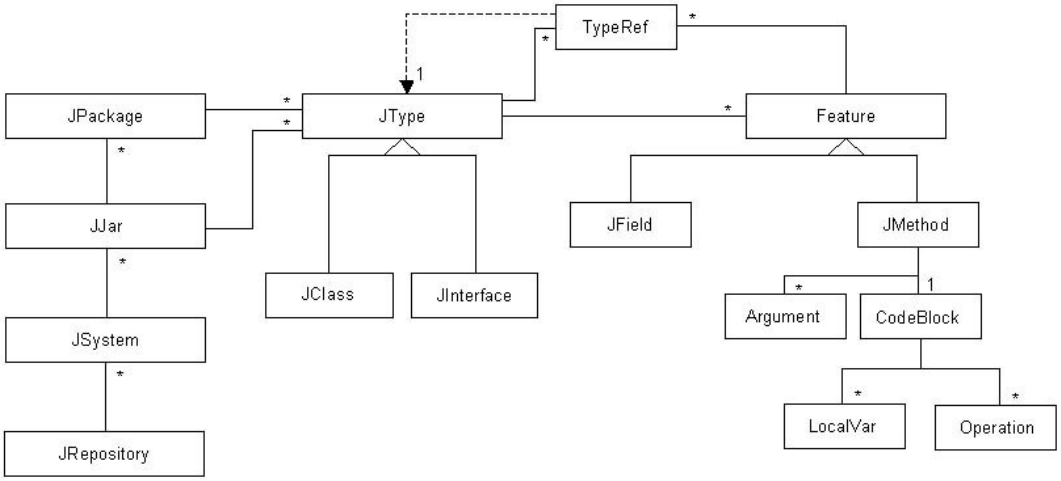
- De methode 'hasTag()' is gedefinieerd in meerdere classes: GenMethodImpl, GenTypeImpl, GenFieldImpl, GenEventImpl, GenPropertyImpl, GenVariableImpl, CodeBlockImpl. Aangezien de subclasses van ElementTags static code bevatten, bleek inheritance niet geheel mogelijk. Dit hield in dat de classes zelf hun eigen ElementTags moesten declareren.
- De methodes die met de methode 'hasTag()' werken, staan in zowel Feature als in GenType door toedoen van het bovenstaande probleem. Methodes die dit aangaat zijn typisch de methodes die het protection-level van een element opvragen, zoals 'isPublic()' en 'isPrivate()'.

7.2 Relaties in het model na de implementatie

De onderstaande diagrammen geven de bevatrelaties en een stukje inheritance in het nieuwe geïmplementeerde metamodel weer. Zie ook figuur 3.2.2 voor een vergelijking van deze relaties in het oude en het nieuwe metamodel.



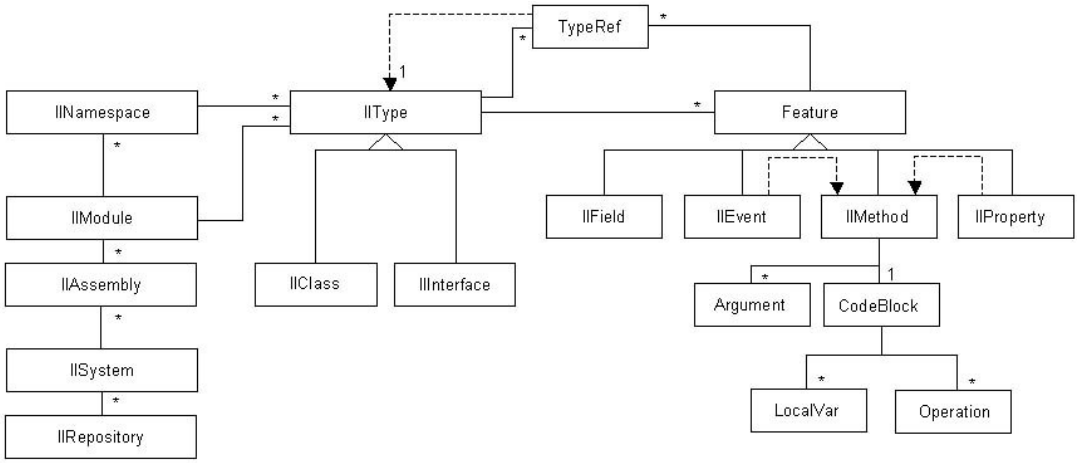
Figuur 7.2.1: de relaties in het generieke deel van het metamodel. GenEvents en GenProperties kunnen links naar meerdere methodes bevatten, een TypeRef heeft een referentie naar één GenType.



Figuur 7.2.2: de relaties in het Java-specifieke gedeelte. Java elementen maken voor het grootste deel gebruik van gedrag dat is gespecificeerd in de Gen- elementen.

Uit bovenstaande modellen blijkt dat Java maar een subset van de mogelijke elementen gebruikt. Java maakt geen gebruik van GenEvent en GenProperty en in het metamodel staan ook geen Java specifieke subclasses hiervan gedefinieerd. Dit geeft aan dat het metamodel meer generiek is geworden en in ieder geval minder Java- georiënteerd van opzet is.

Een methode gebruikt nu een codeblock om de operaties en lokale variabelen in op te slaan en heeft een lijst van argumenten. Als laatste is de JJar geïntroduceerd tussen JSystem en JPackage.



Figuur 7.2.3: de relaties in het Msil-specifieke gedeelte.

Het grootste verschil tussen de generieke elementen en de Il-elementen is het gebruik van een IIModule. Hierdoor herdefinieert de IIAssembly een aantal methodes van zijn superclass GenDeploymentUnit. MSIL kan in tegenstelling tot Java wel Events en Properties definiëren en heeft daarvoor zijn eigen subclasses IIEvent en IIProperty, welke gebruik kunnen maken van 1 of meerdere methodes.

8. Het individuele gedeelte van de stageopdracht

Na het implementeren van het opgestelde metamodel zijn we verder gegaan met de volgende twee opdrachten:

- Het inlezen van MSIL in RevJava. In het vervolg van deze scriptie gebruik ik de term 'RevMsil' om aan te geven dat er MSIL code ingelezen wordt in RevJava.
- Het aanpassen van de metrics en critics.

Het was handig om de MSIL in te lezen, omdat dit een test vormde voor ons ontwikkelde model. Tevens was het gemakkelijker om met behulp van ingelezen MSIL code de metrics en critics aan te passen. In overleg met Gert is besloten dat we gedeassembleerde MSIL code in gaan lezen, aangezien dit een stuk gemakkelijker te realiseren is dan het inlezen van bytecode. In de toekomst zal er wel een bytecode parser komen voor de MSIL, maar dit is een groot project en niet haalbaar voor ons om dit al te realiseren.

8.1 Inlezen van gedeassembleerde MSIL code

Voor het inlezen van de MSIL willen we gebruik gaan maken van een parser welke de informatie uit de IL-files ophaalt en in een model opslaat. Deze informatie moet dan ingelezen worden in RevMsil, waarna de cross-referencing plaats kan vinden. Het coördineren van het opbouwen van het metamodel in RevMsil zal gedaan worden door een reader.

Om gedeassembleerde MSIL code in te lezen in RevJava moet er het volgende gebeuren:

- Er moet een parser komen die de informatie uit de IL-files haalt en in een model opslaat. Aangezien deze parser maar tijdelijk is, kan het beste ons eigen metamodel hiervoor gebruikt worden. Een andere parser zal mogelijk gebruik maken van een ander intern model om de informatie uit de files op te slaan.

- Een reader moet de informatie van de parser in het metamodel van RevJava pushen. Deze reader kan voor een groot deel op dezelfde manier werken zoals nu het inlezen bij Java-bytecode gaat, alleen moet het voor MSIL gespecificeerd worden. Er zal bekeken moeten worden hoe de packages in `nl.serc.revs.core.reader` ingedeeld moeten gaan worden. Nu is er al een package `nl.serc.revs.core.reader.javaclasses`. Voor MSIL zou er een `nl.serc.revs.core.reader.msilclasses` kunnen komen.

  Voor de Java/MSIL reader classes zal er een aangepaste structuur bedacht moeten worden. De class FastAnajaClassReader zal een MSIL tegenhanger moeten krijgen, met als gezamenlijke superclass een Gen-reader class. De interface structuur (nu is AnajaReader de interface voor FastAnajaClassReader) zal dan ook aangepast worden. Misschien kunnen er methodes en/of fields naar het generieke gedeelte. Het aanmaken van de Tags zou misschien in een TagFactory kunnen gebeuren of iets dergelijks, de class JFilteringLoader moet ook voor MSIL gedefinieerd worden (IlFilteringLoader) en een gezamenlijke superclass GenFilteringLoader. Misschien kan JFilteringLoader wel werken op Gen-elementen dan is de opdeling in MSIL en Java niet nodig.

- Er moet op de een of andere manier duidelijk worden dat er MSIL ingelezen gaat worden, het liefst al bij het opstarten van het programma. Hiermee krijgen we een programma dat Java en MSIL aan kan, maar bij het runnen slechts gespecialiseerd is voor één van de twee.

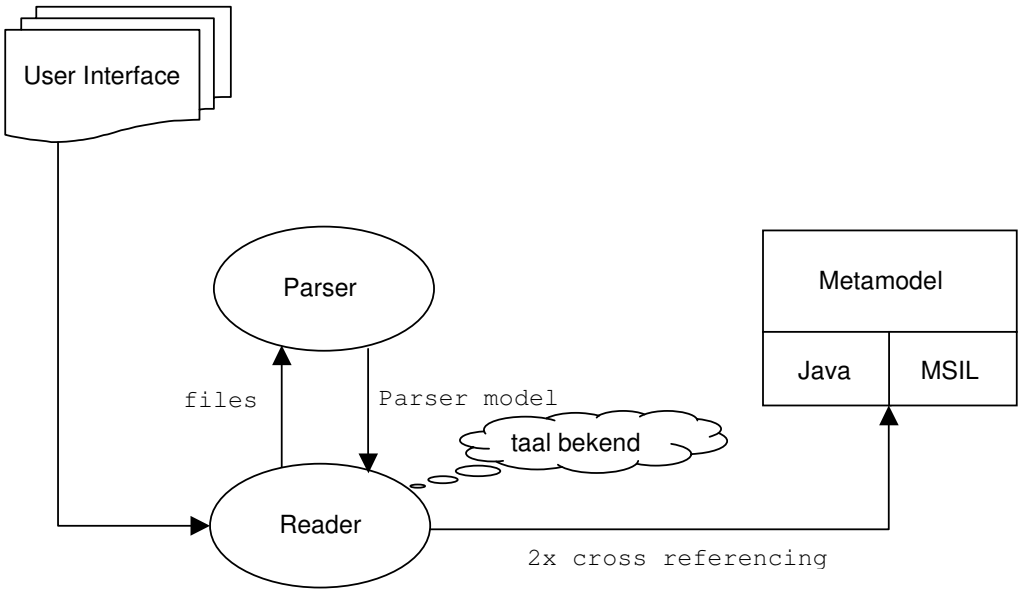
Aangezien ook de parser waarschijnlijk gebruik zal gaan maken van het RevJava metamodel lijkt een aparte reader misschien overbodig, maar wanneer er een andere parser komt scheelt

het werk als de (package) structuur en de class zelf al aanwezig zijn. Daarbij bouwt de parser geen echte model structuur op, wat de reader wel doet d.m.v. cross-referencing. Bij verandering van parser zal de reader opnieuw aangepast moeten worden.

We willen de volgende informatie uit de IL-files ophalen:

- classes, methodes, namespaces, fields, lokale variabelen, argumenten, events en properties. Hierbij moet op de volgende puntn gelet worden:
  - gedistribueerde namespaces moeten mogelijk samengevoegd worden
  - de modifiers op de elementen moeten opgeslagen worden
  - van variabelen moet bepaald worden of het pointers zijn
- bepaalde operaties moeten herkend worden in methodes. We zijn geïnteresseerd in de volgende operaties:
  - OpCreate: het aanmaken van nieuwe types.
  - OpFieldRef: referenties aan een field.
  - OpCall: het aanroepen van een methode.
  - OpCheck: operaties zoals casts en 'instanceof' operaties op een type.
- de modules in een assembly moeten aangegeven worden. Deze kunnen we echter nog niet inlezen aangezien we geen dll's parsen. Wanneer de modules ook gedeassembleerd zijn zouden we deze wel in kunnen lezen.

Hieronder wordt in een schema globaal weergegeven hoe de flow binnen RevJava en RevMsil er uit ziet wanneer er files worden ingelezen.



Figuur 8.1: een globaal schema voor het inlezen van code in RevJava en RevMsil. Uit de user interface wordt aan de reader verteld wat het pad is waar de in te lezen files staan. De parser doorloopt deze files en bouwt een intern model op. Via filtering wordt de juiste informatie door de reader in het metamodel gestopt en wordt er cross-referencing uitgevoerd.

8.2 Metrics en Critics aanpassen

Op het nieuwe metamodel willen we weer de properties kunnen berekenen. Voor het grootste deel gebeurt dit op het generieke deel van het metamodel, maar omdat Java en MSIL beiden specifieke taalelementen hebben is dit niet altijd mogelijk. Het is belangrijk te bepalen welke properties, en daarmee samenhangende metrics en critics, wel en niet op het generieke gedeelte van het metamodel uitgevoerd kunnen worden, omdat het anders voor kan komen dat RevJava of RevMsil onjuiste of nutteloze informatie geeft.

De opdracht hebben we verdeelt in de volgende onderdelen:

1. Welke metrics en critics zijn generiek en welke zijn Java of MSIL specifiek. Tevens moet rekening gehouden worden met het feit dat later andere talen toegevoegd kunnen gaan worden.
2. Hoe gaan we de metrics en critics die niet op alle talen (dus niet voor het generieke model) van toepassing zijn, afleiden en representeren. Dit moet zo intuïtief en eenvoudig mogelijk gedaan kunnen worden.
3. Er moet rekening gehouden worden met het toevoegen van mogelijke analyses op een onderdeel van de datastructuur. Bijvoorbeeld nullpointer analyse op een bepaalde Java class. Dit soort analyses implementeren behoort niet tot de opdracht, echter het moet later op redelijk eenvoudige wijze toegevoegd kunnen worden.
4. Er moet een gelaagdheid toegevoegd worden aan de regels. Het is mogelijk dat als van kritiek *k1* van toepassing is, kritiek *k2* ook altijd geldt. Omdat het de gebruiker dan geen extra informatie geeft, is het beter om dan *k2* buiten beschouwing te laten. Ook dit moet weer zo geïmplementeerd worden dat het toevoegen van regels ‘eenvoudig’ blijft.
5. Er moet gezorgd worden dat de critcs alleen op een CritiquedElement uitgevoerd kunnen worden.

8.3 De verdeling van de opdrachten

De twee opdrachten hebben we als volgt verdeeld:

Jaco is verder gegaan met het aanpassen van de metrics en critics, ik ben verder gegaan met het inlezen van de IL-files in het metamodel.

9. Het inlezen van de MSIL

Voor het inlezen van de informatie uit de class-files maakt RevJava gebruik van een bytecode parser, die de informatie uit de class-files opslaat in een intern model. Deze informatie wordt door RevJava opgevraagd en door de readers in het metamodel opgeslagen.

Voor de MSIL zou er een soortgelijk systeem moeten komen om de informatie uit de IL-files in te kunnen lezen: een parser die de informatie uit de IL-files haalt en readers die deze informatie in het metamodel opslaan. Het voordeel van deze opzet is, dat RevJava zonder al te veel aanpassingen geschikt is voor een andere parser: alleen de readers hoeven aangepast te worden. Dit is erg belangrijk voor ons omdat we in eerste instantie de IL-files in gaan lezen, maar het de bedoeling is dat er in de toekomst een parser komt voor de bytecode van de assemblies.

Er moesten dus twee dingen gebeuren om de informatie uit de IL-files in te lezen in RevJava:

1. Er moest een parser voor de IL-files komen, die de informatie daaruit zou opslaan in een intern model.
2. Er moet een reader voor MSIL gedefinieerd worden in RevJava. Deze reader stuurt de parser aan en slaan de informatie daaruit op in het metamodel van RevJava.

Wanneer de code ingelezen wordt, kan het zijn dat er types zijn die niet ingelezen kunnen worden. Deze types, die we unresolved types noemen, kunnen in een file gedefinieerd staan die niet gevonden wordt of niet beschikbaar is, zoals bijvoorbeeld de core library (mscorlib.dll) van de MSIL. Het zou echter handig zijn in sommige gevallen om toch informatie over deze types te hebben, omdat dit van invloed kan zijn op de critics. Denk hierbij bijvoorbeeld aan het herdefiniëren van methodes of referenties aan een field in een ouder-kind relatie. We willen proberen informatie over deze unresolved types af te leiden aan de hand van het gebruik ervan door andere types in de software. Deze classes komen in paragraaf 9.2.3 aan bod.

9.1 De MSIL-parser

De MSIL-parser haalt de informatie uit de IL-files. Voor het ontwikkelen van de parser waren er een aantal zaken waarmee rekening gehouden moest worden:

- De parser moet schaalbaar zijn: kleine uitbreidingen of het inlezen van extra informatie moet gemakkelijk aan de parser toegevoegd kunnen worden.
- De parser moet snel zijn.
- De parser moet op een gemakkelijke manier te benaderen en aan te sturen zijn: dit is belangrijk om de informatie die de parser inleest op een efficiënte manier in RevJava in te lezen. Tevens moet de parser vanuit de readers aan te sturen zijn.
- De parser moet de informatie uit de IL-files opslaan, zodat deze na het parsen opgevraagd kan worden.

Er moet een model ontwikkeld worden waarin de parser zijn informatie kan opslaan. De informatie in dit model moet compleet zijn en gemakkelijk benaderbaar voor RevJava. Aangezien de parser speciaal geschreven wordt voor RevJava, was het een logische keuze om gebruik te maken van het metamodel van RevJava voor het opslaan van de informatie. Het gebruik van het metamodel had de volgende voordelen:

- Er hoefde geen apart model gespecificeerd te worden voor de parser.
- De parser hoeft alleen die informatie in te lezen die ook daadwerkelijk van belang is voor het metamodel.
- De readers hoeven weinig te doen, aangezien de informatie al in het juiste model zit.

9.1.1 JTopas

Voor de implementatie van de parser had ik besloten gebruik te maken van een tool genaamd 'JTopas'. Met behulp van JTopas is het gemakkelijk om grote tekstfiles te doorlopen aan de hand van te definiëren keywords en speciale tekens. Er waren een aantal redenen waarom ik voor JTopas heb gekozen:

- De schaalbaarheid van de parser is op een gemakkelijke manier te definiëren.
- Complete controle op de informatie die uit de IL-files gehaald wordt door middel van het definiëren van keywords en speciale tekens.
- JTopas definieert veel methodes om de informatie in de IL-files op een transparante manier in te lezen.
- Door middel van flags kan JTopas geheel naar wens geconfigureerd worden.

Aangezien JTopas gebruik maakt van tokenizers om de file te doorlopen, was het moeilijk de performance van de parser te voorspellen. Deze tokenizers zijn echter geoptimaliseerd voor het zoeken naar keywords en speciale tekens. Door deze optimalisaties is JTopas geschikt om snel grote tekstfiles te doorlopen, waardoor de performance in theorie goed zou moeten zijn. Om de werking van JTopas beter te begrijpen volgt hieronder een voorbeeld.

```
1: private TokenizerProperties props = new StandardTokenizerProperties();
2: private Tokenizer tokenizer = new StandardTokenizer();
3:
4: tokenizer.setSource(/* de contents van een IL-file in een stream */);
5:
6: props.addSpecialSequence("{");
7: props.addSpecialSequence("}");
8: props.addKeyword(".class");
9:
10: tokenizer.setTokenizerProperties(props);
11:
12: Token token;
13: while (tokenizer.hasMoreToken()) {
14:    token = tokenizer.nextToken();
15:    switch (token.getType()) {
16:          case Token.KEYWORD: /*operaties/*
17:          case Token.SPECIAL_SEQUENCE: /*operaties/*
18:    }
19: }
```

In het bovenstaande stuk code gebeurt het volgende:

- Regels 1 en 2: declaratie en initialisatie van de tokenizer en de tokenizer-properties waarmee de tokenizer geconfigureerd kan worden. Er wordt gebruik gemaakt van de standaard tokenizers die in JTopas zitten.
- Regel 4: de tokenizer wordt gebonden aan een bepaalde IL-file. Deze file wordt door middel van een stream meegegeven aan de tokenizer.
- Regel 6 tot en met 10: de tokenizer wordt geconfigureerd met de tokenizer-properties, waaraan keywords en speciale tekens toegevoegd zijn. Het verschil tussen een keyword en een special_sequence is dat een keyword een losstaand woord is en een special_sequence kan voorkomen aan een ander token vast. Naast keywords en special_sequences zijn er nog andere definities, maar het gaat te ver deze hier te behandelen.
- Regel 12 tot en met 19: de IL-file wordt met behulp van de tokenizer doorlopen. Wanneer er een token gevonden is kan via het 'switch' statement opgevraagd worden wat het type ervan is, een keyword of special_sequence in dit geval. Per type kan dan een aparte operatie gedefinieerd worden.

De bovenstaande code is geschikt om declaraties van classes uit een file te lezen. Wanneer er in de IL-file het keyword '.class' gevonden wordt, wordt dit token opgeleverd en worden de operaties in het betreffende gedeelte van het 'switch' statement uitgevoerd.

Met behulp van de speciale tekens kan bijvoorbeeld het begin en einde van de code van een gevonden class bepaald worden, wat bij de betreffende instantie van die class opgeslagen

kan worden. Dit is erg handig wanneer de class op een later tijdstip opnieuw ingelezen moet worden, omdat de tokenizer dan kan lezen tussen het begin en einde van de class en niet de hele IL-file doorlopen hoeft te worden op zoek naar de code van de class.

Stel we hebben het hieronder staande stuk code in een IL-file:

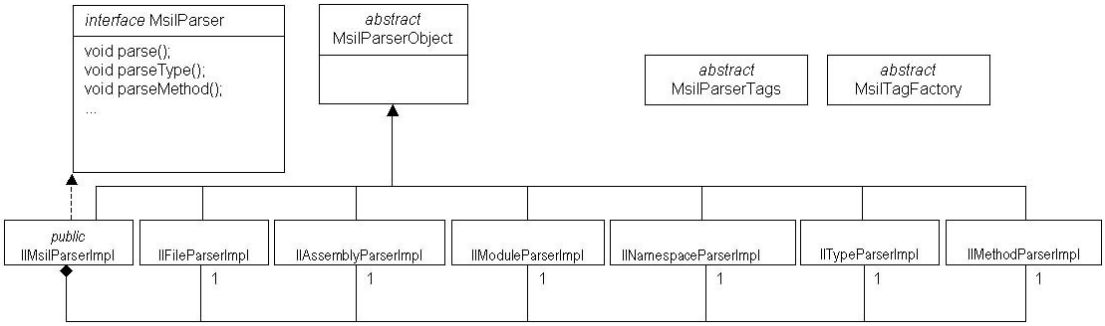
```
.class Person extends [mscorlib]System.Object
{
       .field public string name
       .method public hidebysig specialname instance string
              get_Name() cil managed {}

       …
}
.class Male extends Person
{
       …
}
```

De tokenizer zal hierin de declaraties van de classes Person en Male vinden aan de hand van het keyword '.class'. De beginpositie en eindpositie van de code van de class kunnen gevonden worden door de posities van de '{' en de '}' bij te houden voor de class, waarbij gelet moet worden op het feit dat er binnen de class ook gebruik kan worden gemaakt van deze tekens.

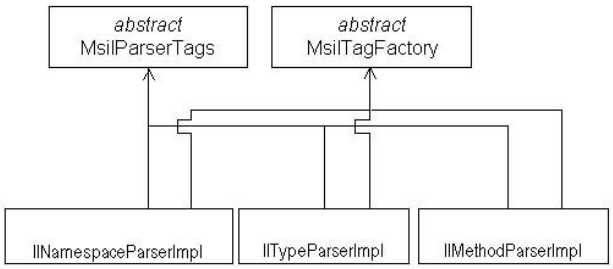
Door het definiëren van de juiste keywords en special_sequences kan met JTopas op een gemakkelijke en duidelijke manier een IL-file geheel doorlopen worden.

9.1.2 Ontwerp en implementatie van de MSIL-parser

Rekening houdend met de eisen die aan de parser gesteld worden en de mogelijkheden die JTopas biedt, is het volgende model voor de parser ontworpen:



Figuur 9.1.2.1: het ontwerp voor de parser.



Figuur 9.1.2.2: IlNamespaceParserImpl, IlTypeParserImpl en IlMethodParserImpl maken gebruik van de classes MsilParserTags en MsilTagFactory.

Het idee was de parser incrementeel op te bouwen. Dit wilde ik bereiken door topdown classes te definiëren, die voor specifieke MSIL- elementen de informatie uit een IL-file haalden. In het bovenstaande model zijn de bouwstenen van de parser duidelijk te zien. Door een instantie aan te maken van een MsilParser kan een gebruiker de parser aanroepen. Deze opbouw van de parser biedt een aantal voordelen:

- Door classes te definiëren voor specifieke MSIL- elementen, is het duidelijk waar bepaalde verantwoordelijkheden binnen de parser liggen. Dit vergemakkelijkt het aanpassen van de parser als dat nodig is.
- Het is eenvoudig te definiëren dat de parser extra informatie kan inlezen op de elementen. Hiervoor is het nodig dat er bekend is op welke posities de elementen in de file staan. Deze informatie wordt door de parser opgeslagen in de elementen.
- De gebruiker hoeft geen inzicht te hebben in de werking van de parser, maar slechts op het gebruik ervan via de MsilParser.

De parser is geïmplementeerd in de volgende packages:

- nl.serc.revs.core.msilparser
- nl.serc.revs.core.msilparser.impl

Alle classes van de parser hebben hun eigen Tokenizer. Deze tokenizer wordt geconfigureerd met keywords en special sequences, zodat de tokenizer geoptimaliseerd is om de elementen te vinden waarnaar de class zoekt. Zo zal de IlTypeParserImpl de fields en methodes in een type willen vinden en daarvoor keywords gebruiken als ".field" en ".method".

Hieronder worden de verschillende classes die tezamen de parser vormen behandeld:

MsilParser en MsilParserImpl

Wanneer men gebruik wil maken van de parser gaat dit via een instantie van MsilParser. Hierin staan de methodes gedefinieerd die een gebruiker op de parser aan kan roepen. MsilParserImpl implementeert de interface MsilParser en de methodes erin.
De belangrijke members van MsilParserImpl zijn:

- `private IlAssembly _assembly`: de assembly die ingelezen wordt. Wanneer er een .netmodule wordt ingelezen, ontbreekt het in de MSIL aan een assembly-declaratie. Er wordt dan een assembly gegenereerd met als naam de filenaam van de ingelezen .netmodule.
- `private IlModule _module`: de module met de MSIL van de ingelezen assembly.
- `private Vector _modules`: de externe modules in de assembly
- `private Vector _assemblies`: de externe assemblies
- `private Vector _namespaces`: de namespaces in alle modules
- `private pSet _resolvedtypes`: alle gevonden types
- `private pSet maybe_unresolvedsupertypes, maybe_unresolvedtypes`: sets van types die mogelijk niet resolved zijn. Dit zijn bijvoorbeeld types die als supertype van een type gedefinieerd staan of returntypes van methodes.
- `public void parse()`: deze methode start het parse-proces. Wanneer deze methode aangeroepen wordt, worden na elkaar de volgende methodes uitgevoerd:
  - `parseFile()`: haalt de assembly op die ingelezen wordt.
  - `parseAssembly(IlAssembly _assembly)`: haalt informatie over de modules, namespaces en types op uit de assembly. Deze informatie staat in de outline van de IL-file en is aan het begin van de IL-file te vinden.
  - `parseModule(IlModule _module)`: haalt informatie uit de IL-file over de structuur van de namespaces en maakt de default namespace aan.
  - `parseNamespaces(IlModule _module)`: bepaalt welke types in welke namespace horen en zet parent/child relaties in de namespaces goed.

  Voor het uitvoeren van de parsings heeft de MsilParserImpl instanties gedefinieerd van de volgende classes: IlFileParserImpl, IlAssemblyParserImpl, IlModuleParserImpl, IlNamespaceParserImpl, IlTypeParserImpl, IlMethodParserImpl.

  Na het aanroepen van de methode 'parse()' kan de gebruiker alle assemblies, modules, namespaces en types die voorkomen in de IL-file opvragen. Met deze informatie kan de gebruiker dan zelf aangeven voor welke types er wel en niet verder geparsed moet worden. De methode roept de volgende methodes aan:
- `public void parseType(GenType gt,IlNamespace ns)`: haalt de methodes en fields uit een meegegeven type op.
- `public void parseMethod(IlMethod m,GenType gt)`: haalt de operaties en locale variabelen uit de meegegeven methode op.
- `private void checkUnresolveable(pSet _unresolveds)`: deze methode checkt of er in de sets met unresolved types nog types voorkomen die al wel door de parser gevonden zijn, deze worden uit de set verwijderd.

MsilParserObject

MsilParserObject definieert generieke methodes die door zijn subclasses gebruikt kunnen worden, waaronder de volgende:

- `protected void addNewFoundType(String name)`: voegt een type toe dat mogelijk niet resolved is.
- `protected pSet getNewFoundTypes()`: geeft de hele lijst met mogelijke niet resolved types terug.
- `protected Tokenizer createTokenizer(String filename)`: maakt een nieuwe tokenizer aan voor het doorlopen van de meegegeven file.

IlFileParserImpl

Deze class leest de naam van de assembly die ingelezen wordt en maakt hiervoor een IlAssembly aan. Ditzelfde gebeurt voor de externe (import) assemblies, die worden bijgehouden in een Vector. De belangrijkste members zijn:

- `private Vector _assemblies`: de externe assemblies die in de IL-file gedefinieerd zijn. Deze vector is op te vragen via getAssemblies().
- `private IlAssembly _assembly`: de assembly die in de IL-file gedefinieerd wordt. Deze is op te vragen via getAssembly().
- `private int max_tokens`: het aantal tokens in de IL-file. De methode getMaxTokens() geeft de max_tokens terug.
- `public void parse()`: deze methode begint de parsing van de file. Met behulp van de tokenizer wordt de file doorlopen op zoek naar de assemblies.

IlAssemblyParserImpl

IlAssemblyParserImpl leest de informatie die in een IlAssembly opgeslagen wordt: de module met daarin de MSIL van de assembly en de import modules. Tevens worden er lege types en namespaces aan de module toegevoegd. Deze types en namespaces worden gevonden in de outline van de MSIL die altijd aan het begin van de MSIL gedefinieerd staat. De members van de class IlAssemblyParserImpl zijn:

- `private Vector _modules`: de modules in de assembly.
- `private IlModule _module`: de module met de MSIL van de assembly.
- `private Vector _types`: de types in de assembly.
- `private Vector _namespaces`: de namespaces in de assembly.
- `public void parse()`: deze methode loopt met de tokenizer de IL-file langs op zoek naar de modules, namespaces en types.

IlModuleParserImpl

IlModuleParserImpl definieert de default namespace voor de module en voegt deze aan de module toe. Hierna wordt de IL-file doorlopen op zoek naar de namespaces die vanuit de AssemblyParserImpl aan de module toegevoegd zijn. Van deze namespaces wordt de beginpositie en eindpositie in de IL-file opgeslagen, zodat bekend is waar de code van de namespaces zich in de IL-file bevindt. De volgende members zijn belangrijk in IlModuleParser:

- `private IlModule _module`: de module die geparst wordt.
- `private IlNamespace _default`: de default namespace die in de module gedefinieerd wordt.
- `public void parse()`: binnen deze methode worden namespaces gevonden in de IL-file. Er wordt bekeken of de namespace al in de module zit, wat normaal gesproken het geval is aangezien dit in de IlAssemblyParser al gedaan is. Vervolgens wordt de beginpositie en eindpositie van de namespace in de IL-file bepaald.

IlNamespaceParserImpl
Aan de hand van de beginpositie en eindpositie van een namespace, zoekt de IlNamespaceParserImpl de types die tot die namespace behoren. Deze types zijn reeds in de omvattende module van de namespace aanwezig, maar worden nu verder aangevuld. Ook bij types worden de beginpositie en eindpositie in de IL-file opgeslagen. De modifiers die op een type gedefinieerd zijn worden opgeslagen bij het type.

Wanneer er in de namespace een geneste namespace gevonden wordt, wordt deze als child-namespace toegevoegd aan de namespace. De geneste namespace krijgt de omvattende namespace als parent.

IlNamespaceParserImpl heeft de volgende members:

- `private Vector _types:` de types die in de namespace gevonden worden.
- `private pSet maybe_unresolvedsupertypes:` de supertypes van de types in de namespace.
- `Private module holder:` de module waarin de namespace bevat is.
- `public void parse():` vindt de types in de MSIL. De types zijn al toegevoegd aan de holder door de IlAssemblyImpl, maar hier worden de modifiers, supertypes en interfaces erin opgeslagen.

IlTypeParserImpl
Aan de hand van de beginpositie en eindpositie van een type worden de features van een type gevonden. Voor de gevonden features worden objecten aangemaakt en toegevoegd aan het type. Tevens worden de modifiers op de features opgeslagen in de daarbij behorende ElementTags.

Voor methodes wordt de beginpositie en eindpositie in de IL-file opgezocht en bijgehouden, tevens worden de argumenten van een methode opgezocht. Voor properties wordt er gekeken of de ‘get_’ en ‘set_’ methodes in de class aanwezig zijn en als dit het geval is krijgt de property een pointer naar de methode. Voor events worden de ‘add_’, ‘remove_’ en ‘fire_’ methodes opgezocht.

Wanneer er een geneste class gevonden wordt, worden de inner- en outer-fields van de classes ge-update.

De volgende members maken deel uit van IlTypeParserImpl:

- `private GenType _type:` het type dat geparst wordt.
- `private IlNamespace _holder:` de namespace die dit type bevat.
- `public void parse():` de methode vindt methodes, fields, events, properties en geneste classes aan de hand van in de tokenizer gedefinieerde keywords.
- `private Variable[] getArguments():` wanneer er een methodedeclaratie gevonden is, worden met behulp van deze methode de argumenten die erbij horen uit de MSIL gehaald.

IlMethodParserImpl
Aan de hand van de beginpositie en eindpositie in de IL-file worden operaties en locale variabelen opgezocht in de body van een methode. Voor de gevonden operaties en locale variabelen worden de betreffende objecten uit het metamodel aangemaakt en aan de methode toegevoegd. Daarnaast wordt er een teller bijgehouden, die het aantal operaties in de methode bijhoudt. Dit wordt in de methode opgeslagen. IlMethodParserImpl heeft de volgende members:

- `private IlMethod _method:` de methode die geparst wordt.
- `private CodeBlock _codeblock:` het codeblock van de methode.
- `private Vector _operations:` de operaties die in de body van de methode gevonden worden.
- `private int _nrofstatements:` het aantal operaties in de methode.

- `public void parse()`: de methode vindt aan de hand van de in de tokenizer gedefinieerde keywords de operaties en locale variabelen die worden gedefinieerd.
- `private LocalVar[] getLocalVars()`: het keyword ".locals" geeft aan dat de lijst met locale variabele declaraties volgt. Deze methode splitst de lijst vervolgens in de individuele variabelen.
- `private TypeRef[] getArgumentsFromCall()`: wanneer een operatie een methodeaanroep is wordt met behulp van 'getArgumentsFromCall' de lijst van argumenten gesplitst in individuele referenties naar de types van de argumenten.

MsilParserTags
In MsilParserTags staan alle modifiers gedefinieerd die in de MSIL kunnen voorkomen. De modifiers zijn gesplitst naar het element waar ze op voor kunnen komen. Tevens staat hierin de lijst van alle ingebouwde types in de MSIL. Deze class is abstract en kan alleen statisch gebruikt worden.

MsilParserTags heeft de volgende members:
- `private final static String[] TYPE_TAGS`: de modifiers die op een type gedefinieerd kunnen worden.
- `private final static String[] FEATURE_TAGS`: de modifiers die op features gedefinieerd kunnen worden.
- `private final static String[] FEATURE_DECLARATION_TAGS`: de modifiers die voor de compiler betekenis hebben.
- `private final static String[] BUILT_IN_TYPES`: de lijst van ingebouwde types. Eigenlijk is de aanwezigheid van deze member dubieus in deze class, omdat het eigenlijk geen array van modifiers is.
- `private final static String[] VARIABLE_TAGS`: de modifiers die op een variabele gedefinieerd kunnen worden.
- `public static boolean isBuiltInType(String key)`: bepaalt of de 'key' een ingebouwd type representeert.
- `public static boolean isVariableTag(String key)`: bepaalt of de 'key' een mogelijke modifier op een Variable representeert.
- `public static boolean isMultipleType(String key)`: bepaalt of de 'key' een array-type representeert.

MsilTagFactory
De MsilTagFactory definieert methodes voor het creëren van de verschillende ElementTags: TypeTags, MethodTags, FieldTags, VariableTags, CodeBlockTags en PropOrEventTags. Ook MsilTagFactory is abstract gedefinieerd en kan alleen statisch gebruikt worden. MsilTagFactory bevat de volgende methodes:
- `public static TypeTags createTypeTags(Vector tagnames)`
- `public static MethodTags createMethodTags(Vector tagnames)`
- `public static PropOrEventTags createPropOrEventTags(Vector tagnames)`
- `public static FieldTags createFieldTags(Vector tagnames)`
- `public static VariableTags createVariableTags(Vector tagnames)`
- `public static CodeBlockTags createCodeBlockTags(Vector tagnames)`

Alle methodes krijgen een Vector binnen waarin de gevonden modifiers op een bepaald element staan. Er wordt gecheckt of de modifiers gedefinieerd zijn in de betreffende ElementTags om opgeslagen te worden. Is dit het geval dan wordt de juiste bit op 1 gezet in de locale variabele 'tagbits' van de methode. Met deze 'tagbits' als parameter wordt dan de juiste ElementTags aangemaakt.

9.1.3 Performance van de parser

RevJava is erop gericht om op een snelle manier tussentijdse reviews op de code te doen. Dit brengt met zich mee dat de parser snel moet zijn, zodat een ontwikkelaar niet lang hoeft te wachten tot de code is ingelezen. Bij het ontwikkelen van de parser was het dan ook van cruciaal belang om de performance in het oog te houden. Dit kon gelukkig vrij gemakkelijk, omdat de parser stapsgewijs werd opgebouwd. Wanneer er extra functionaliteit aan de parser werd toegevoegd kon gemakkelijk bekeken worden wat de invloed daarvan op de parser was.

Tijdens de ontwikkeling van de parser is op de volgende punten gelet ten gunste van de performance:

- Er wordt zo min mogelijk gebruik gemaakt van objecten die slechts dienen om tijdelijk data in op te slaan.
- Vectoren worden aangemaakt met een initiële grootte, zodat er geheugen gereserveerd wordt. Nu hoeft er niet bij ieder element dat wordt toegevoegd steeds opnieuw geheugen gereserveerd te worden.
- Objecten die aangemaakt zijn worden efficiënt gebruikt. De classes die de parser vormen hoeven bijvoorbeeld slechts één keer aangemaakt te worden, waarna ze dan meerdere keren gebruikt worden om elementen te parsen.
- Wanneer er door de IL-file gesprongen moet worden naar een bepaalde positie, wordt dit zo efficiënt mogelijk gedaan.
- Er wordt veel gebruik gemaakt van de mogelijkheden die JTopas biedt op het gebied van keywords en special sequences.

Een belangrijk aspect voor de performance van de parser is ook om minimaal JTopas versie 0.6.2 te gebruiken. De eerdere versies kunnen problemen geven wat betreft performance, omdat het zoeken naar de special sequences daarin nog niet helemaal geoptimaliseerd is. In versie 0.6.2 zijn deze problemen aangepakt en de performance is een stuk beter dan in de eerdere versies.

Voor de implementatie van de parser is gebruik gemaakt van de literatuur genoemd bij: [42]

9.2 De MSIL-readers

RevJava maakt voor het inlezen van Java code gebruik van twee classes om de informatie vanuit de parser in het metamodel in te lezen: FastAnajaClassReader en JFilteringLoader. Deze staan gedefinieerd in het package `nl.serc.revs.core.reader.javaclasses`.

FastAnajaClassReader implementeert de interface AnajaReader in het package `nl.serc.revs.core.reader`. Tijdens het laden van de class-files wordt er gebruik gemaakt van een class die de voortgang daarvan in de gaten houdt: LoadProgressListener.

JFilteringLoader neemt een lijst van class-files en bepaalt aan de hand van filters of het type wel of niet geladen moet worden en of de code in de methodes geladen moet worden. Het laden van een type gebeurt in de FastAnajaClassReader. In de FastAnajaClassReader wordt de meegegeven class naar de parser gestuurd en worden daarna de elementen van het metamodel aangemaakt met de informatie uit de parser terugkomt.

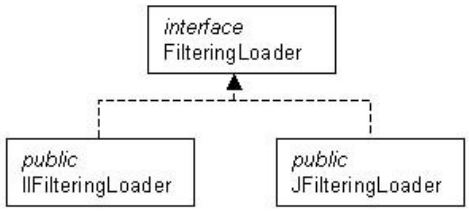
Het voordeel van het definiëren van een aparte reader en loader is, dat de controle over het laden van de files en het daadwerkelijke laden zelf gescheiden is. Deze structuur is daarom ook voor de MSIL gedefinieerd in het package: `nl.serc.revs.core.reader.ilassemblies`. Hieronder worden de elementen die verantwoordelijk zijn voor het laden van de types in het metamodel behandeld.

9.2.1 IlFilteringLoader

De IlFilteringLoader controleert en initieert het laden van de IL-files en alle elementen die daarin gedefinieerd staan. Met deze elementen wordt het metamodel opgebouwd.

De IlFilteringLoader krijgt een lijst van namen van IL-files die ingelezen moeten worden. Deze files moeten expliciet door de gebruiker aangegeven worden, omdat RevJava niet de directory structuur bekijkt op zoek naar meer IL-files in de directories. Dit gebeurt bij het inlezen van Java code wel. De reden dat dit bij het inlezen van MSIL niet gedaan wordt is dat een IL-file op zich al een programma vormt. Wanneer er gebruik wordt gemaakt van externe modules in het programma worden die wel opgezocht tijdens het laden van de IL-file.

De IlFilteringLoader implementeert de interface FilteringLoader, die aangemaakt is in het package `nl.serc.revs.core.reader`. Ook de JFilteringLoader implementeert deze interface nu.



Figuur 9.2.1.1: IlFilteringLoader en JFilteringLoader

De IlFilteringLoader heeft veel members, de belangrijkste zijn:

- `private pSet allunresolvedtypenames`
  Alle unresolved types uit de parser worden hierin opgeslagen.
- `private IlAssemblyReader thereader`
  Een instantie van de IlAssemblyReader, ongeveer het equivalent van de FastAnajaClassReader voor Java classes. Zie ook paragraaf 8.2.2 voor gedetailleerde uitleg over deze class.
- `private DefaultTypeDeriver typederiver`
  Een object dat de unresolved types afleidt aan de hand van operaties en fields. Zie ook paragraaf 9.2.3 voor gedetailleerde uitleg.
- `private pSet modulestoload`

Wanneer een ingelezen assembly aangeeft dat er gebruik wordt gemaakt van externe modules, worden de namen hiervan opgeslagen in de pSet, zodat op een later tijdstip de modules gezocht en geladen kunnen worden. Dit gebeurt niet als de module al geladen is.

- `private Vector loadedfiles`
  De namen van de files die geladen zijn worden opgeslagen in de Vector, zodat er op een later tijdstip gekeken kan worden of een file al geladen is.
- `public void loadAll(pSet inames)`
  Deze methode controleert het laden van de IL-files. De lijst met te laden files wordt langsgelopen en de files worden één voor één ingeladen. Na het inladen van de files en de externe modules worden de methodes aangeroepen voor het creëren van de cross-references.
- `private void loadExternModules(IlSystem system,IlSystem rest)`
  Deze methode zoekt de files op die bij de externe modules in het systeem horen. De files moeten in dezelfde directory staan als de assemblies die ze gebuiken. Vervolgens worden de modules geladen.
- `private void loadDataIntoSystems(IlSystem system,IlSystem rest)`
  Deze methode laadt de informatie na het parsen in de IlSystems. Er worden twee IlSystems bijgehouden: één waarin alle gevonden types en namespaces opgeslagen worden en één voor import types.
- `private void load(String name, IlSystem system, IlSystem rest, boolean withcode)`
  Deze methode verzorgt het laden van het type met de meegegeven naam. Als de boolean 'false' is wordt er geen code geladen van de methodes in het type.
- `private Vector filterUnResolveds(Vector unresolveds,Vector loaded)`
  Vanuit de parser worden alle typenamen die unresolved zijn opgeslagen in de pSet 'allunresolvedtypenames'. Als een assembly echter uit meerdere modules bestaat kan een type als unresolved gemarkeerd zijn, maar wel voorkomen in een andere module. Deze methode checkt of er geladen types voorkomen in de set met alle unresolved types en verwijdert deze dan.

De werking van de IlFilteringLoader is globaal als volgt:
De lijst met de namen van de IL-files die geladen moeten worden wordt afgelopen. Bij iedere file wordt er gekeken of de file bestaat, zo ja dan wordt de IlAssemblyReader geconfigureerd voor het inlezen van deze file. Vervolgens worden de assemblies, modules en namespaces die in de file gedefinieerd zijn opgeslagen in het juiste IlSystem. De types worden allemaal in een Vector geladen en hierdoor als 'unresolved' gemarkeerd. Nu worden de types één voor één geladen met hun members. Een type wordt dan als 'resolved' gemarkeerd en aan de IlRepository toegevoegd.

Wanneer alle types in een file geladen zijn, wordt er gekeken of er nog types uit externe modules geladen moeten worden. Er wordt gekeken of een te laden module al niet in de initiële lijst voorkomt, omdat de module dan al geladen is. Als dit niet het geval is wordt de module geladen.

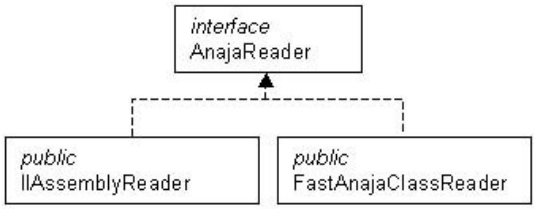
Hierna volgt het aanmaken van de types die unresolved zijn. Dit kunnen supertypes van de resolved types zijn, maar ook returntypes van methodes en types van een field of variabele. De DefaultTypeDeriver krijgt de lijst unresolved types mee. Hierna wordt er in alle OpCall operaties gecheckt of er methodes aangeroepen worden in één van de types in de lijst. Wanneer dit het geval is wordt de methode aangemaakt in het type. Ditzelfde gebeurt er voor fields met een type dat ook in de lijst met unresolved types voorkomt. Op deze manier wordt er een beeld opgebouwd van de types in het import system.

Als laatste wordt de cross-referencing uitgevoerd op de elementen. Door de cross-referencing komt er een echte structuur in het metamodel. Voor de cross-referencing zijn er speciale elementen gedefinieerd voor de MSIL. Als de cross-referencing afgelopen is, zit de taak van de IlFilteringLoader er ook op en is het interne model opgebouwd.

9.2.2 IlAssemblyReader

De IlAssemblyReader zorgt ervoor dat de IlFilteringLoader op een gemakkelijke manier gebruik kan maken van de parser en dat de juiste types geladen worden. Daarnaast beheert de IlAssemblyReader ook alle resolved en unresolved types, zodat deze informatie altijd up-to-date is als de IlFilteringLoader er om vraagt.

De functie van de IlAssemblyReader is anders dan de functie van de FastAnajaClassReader, omdat alle geladen elementen al in het juiste metamodel zitten. In de FastAnajaClassReader moet alle informatie vanuit de parser eerst nog in het metamodel 'gegoten' worden. Als er in de toekomst een andere parser gebruikt gaat worden zal de functie van de IlAssemblyReader mogelijk gelijk worden aan de functie van de FastAnajaClassReader. De naam Anaja komt voort uit een stukje historie van RevJava, dat in de eerste fase van ontwikkeling Anaja heette.



Figuur 9.2.2.1: IlAssemblyReader en FastAnajaClassReader implementeren beide de interface AnajaReader.

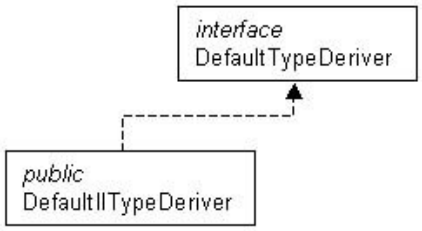
De IlAssemblyReader heeft de volgende members:

- `private MsilParser theparser`
  Een instantie van de parser.
- `public void initialParse()`
  Deze methode zorgt ervoor dat de assemblies, modules, namespaces en de lege types geparst worden. Vervolgens worden de volgende pSets geupdate:
  - private pSet resolvednames
  - private pSet resolveds
  - private pSet unresolveds
  - private pSet unresolvedsupers

  Voor deze methodes zijn er in de IlAssemblyReader getters gedefinieerd.
- `public void parseModule(IlAssembly holder)`
  Deze methode wordt aangeroepen als er een externe module geparst moet worden. De IlAssembly is de holder van de module, dit wordt aan de parser meegegeven.
- `public GenType getType(String name, boolean withoperations)`
  De IlFilteringloader roept deze methode aan als er een type geparst moet worden. Het type wordt opgezocht in de pSet 'resolveds' en door de MsilParser wordt het type gevuld met zijn members.
- `public GenType getType(String name)`
  Deze methode vindt in de pSet 'resolveds' een type met als naam de meegekregen string.
- Er staan ook nog een aantal methodes gedefinieerd om de assemblies en modules vanuit de parser terug te geven naar de IlFilteringLoader:
  - public IlAssembly getAssembly()
  - public Vector getExternModules()
  - public Vector getExternAssemblies()

9.2.3 DefaultTypeDeriver en DefaultIlTypeDeriver

De DefaultTypeDeriver leidt de members van de unresolved types af. Dit gebeurt door te kijken naar het gebruik van de types met betrekking tot zijn members. Voor het afleiden van MSIL types en features is er de class DefaultIlTypeDeriver, voor Java is er nog geen class die dit doet.



Figuur 9.2.3.1: DefaultIlTypeDeriver implementeert DefaultTypeDeriver.

De DefaultTypeDeriver krijgt een Vector met de namen van niet geladen types binnen en een enumeration van externe assemblies. De types worden aangemaakt aan de hand van hun naam en toegevoegd aan de externe assembly waarin ze horen. Er wordt ook gecheckt of een type een interface of een class moet zijn, door te kijken of het type ergens door een interface als supertype gebruikt wordt. Hierna wordt er gekeken naar de volgende features om members te zoeken voor het type:

- methode-aanroepen: wanneer er een methode in één van de unresolved types aangeroepen wordt, wordt deze methode aangemaakt in het type. Het nadeel is dat het niet te bepalen is wat de precieze modifiers op de methode zijn. Het zou bijvoorbeeld handig zijn om te weten of een methode 'virtual' gedefinieerd is, zodat deze hergedefinieerd kan worden. Standaard is dit in de MSIL niet zo, dus heb ik besloten dit ook niet te doen.
- field manipulaties: wanneer er een referentie naar een field in één van de unresolved types is, wordt dit field aangemaakt in het type.

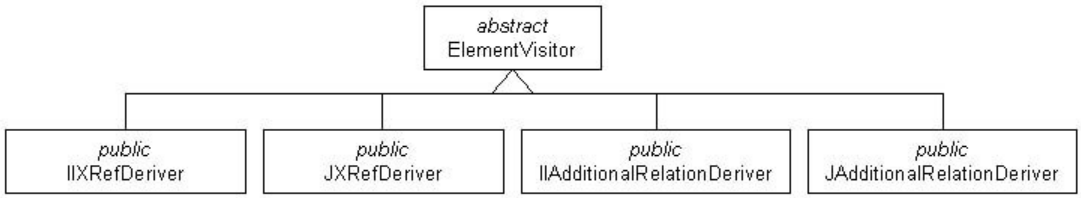
De DefaultIlTypeDeriver implementeert de interface DefaultTypeDeriver en heeft de volgende members gedefinieerd:

- `private GenRepository therepository`
  De respostitory waaraan de types toegevoegd moeten worden.
- `private Vector derivedtypes`
  In deze Vector worden de afgeleide types opgeslagen, zodat deze slechts één keer aangemaakt kunnen worden.
- `private Vector notloadedtypes`
  De lijst met unresolved types die in de DefaultTypeDeriver afgeleid moeten worden.
- `private Vector opcalls`
  De lijst van operaties, die een aanroep op een methode in één van de unresolved types doen.
- `private Vector opfieldrefs`
  De lijst van referenties naar een field in één van de unresolved types.
- `private pSet interfacenames`
  Wanneer een type dat in de respository geladen is een interface is en de superinterface is gezet, wordt de naam van de superinterface opgeslagen in deze pSet. Tevens worden van een type de namen van de interfaces opgeslagen die door de class geïmplementeerd worden. Hiermee kan er gecheckt worden bij het aanmaken van een type of het type een interface moet zijn.
- `public void deriveTypes(Vector unloadablenames,Enumeration extassemblies)`
  Deze methode zorgt dat er voor iedere externe assembly gekeken wordt of er in de Vector 'notloadedtypes' namen van types voorkomen die in de assembly horen. Als dit het geval is wordt het type aangemaakt en in de assembly toegevoegd.

- `private void createType(String name,String qname,IlAssembly assembly)`
  Deze methode maakt een type aan. Het type heeft een naam en een volledig gekwantificeerde naam en wordt toegevoegd in de meegegeven assembly. Binnen deze methode worden ook de methodes en fields die in het type horen aangemaakt. Het type wordt vervolgens aan de repository als geladen gemeld.
- `private Variable[] createArguments(TypeRef[] trefs)`
  Bij het aanroepen van een methode kunnen er argumenten meegegeven worden. Deze methode haalt de types van de argumenten uit de call.
- `private Vector collectOperations(Vector tps)`
  Deze methode verzamelt de operaties in alle geladen types in de repository. Deze worden opgeslagen in de opcalls en de opfieldrefs Vectoren.

9.2.4 Cross-referencing

Voor de cross-referencing zijn er speciale classes geschreven voor de MSIL. De classes staan in het package `nl.serc.revs.core.model.xrefs.msil` gedefinieerd:



Figuur 9.2.4.1: De classes voor de cross-referencing. Om een compleet beeld te krijgen zijn de classes die op Java werken er ook bij gezet.

Uit het model blijkt dat de classes IIXRefDeriver en IlAdditionalRelationDeriver behoren tot de visitors van RevJava. De IIXRefDeriver legt de primaire verbanden tussen de elementen in het model, bijvoorbeeld:

- TypeRefs krijgen een referentie naar het type waarvoor de TypeRef aangemaakt is.
- Voor classes worden subclasses geregistreerd, voor interfaces de types die de interface implementeren.

De IlAdditionalRelationDeriver loopt het opgebouwde model langs nadat de IIXRefDeriver dit gedaan heeft en legt nog extra verbanden. Een van de relaties die door de IlAdditionalRelationDeriver wordt gelegd is bijvoorbeeld de relatie tussen een methode en de methodes die daardoor hergedefinieerd worden of waardoor de methode zelf hergedefinieerd wordt.

<u>Verschillen in cross-referencing tussen Java en MSIL</u>
IIXRefDeriver en IlAdditionalRelationDeriver verschillen beide op enkele punten van de JXRefDeriver en JAdditionalRelationDeriver. Dit heeft in een aantal gevallen te maken met het voorkomen van een type in de IL-file. Een type kan in de IL-file namelijk gerefereerd worden door zijn gekwantificeerde naam of door zijn niet gekwantificeerde naam. Een gekwantificeerde naam kan op 3 manieren voorkomen:

- Een type in een andere namespace: de gekwantificeerde naam ziet er dan als volgt uit: ‘namespace.type’, bijvoorbeeld: ‘n1.test’ is een referentie naar het type ‘test’ in de namespace ‘n1’.
- Een type in een externe assembly: de gekwantificeerde naam is opgebouwd uit een referentie naar de assembly en de gekwantificeerde naam van het type binnen de assembly. Dit ziet er als volgt uit: ‘[assembly]namespace.type’. Bijvoorbeeld ‘[testassembly]n1.test’ geeft het type ‘test’ aan in de namespace ‘n1’ in de externe assembly ‘testassembly’.
- Een type in een externe module: de gekwantificeerde naam is opgebouwd uit een referentie naar de module en de gekwantificeerde naam van het type binnen die

module: ‘[.module module-naam]namespace.type’. Bijvoorbeeld: ‘[.module testmodule.netmodule]n1.test’ geeft het type ‘test’ aan in de namespace ‘n1’ in de externe module testmodule.netmodule.

In eerste instantie zorgde dit voor nogal wat problemen. Deze zijn opgelost door de gekwantificeerde naam bij te houden in de TypeRef en de types. Wanneer de IlXRefDeriver nu een TypeRef bezoekt, wordt de naam van het type van de TypeRef altijd veranderd naar de gekwantificeerde naam zonder assembly- of module-referentie ervoor.

Een ander verschil is dat in de IlAdditionalRelationDeriver er bij het zoeken naar de supermethodes van een methode gecheckt wordt of er wel de juiste modifiers op de methodes gedefinieerd staan:

- De supermethode moet ‘newslot’ en ‘virtual’ gedefinieerd zijn.
- De herdefiniërende methode moet ‘virtual’ gedefinieerd zijn.

In Java is de check op de modifiers niet nodig, aangezien methodes standaard al als ‘virtual’ gedefinieerd zijn.

Als laatste zijn de methodes ‘visitProperty’ en ‘visitEvent’ in de IlXrefDeriver hergedefinieerd. In deze methodes wordt er gekeken of een event of een property voor een field gedefinieerd is. In dat geval worden er in het field flags gezet, zodat bekend is dat er een event of property voor het field gedefinieerd is.

9.3 Veranderingen aan het metamodel

Gedurende de implementatie van de parser en readers zijn er nog enkele aanpassingen gemaakt aan de elementen van het metamodel. Het betreft hier geen aanpassingen van de relaties in het metamodel, maar veelal het toevoegen of verplaatsen van functionaliteit.

Extra methodes om incomplete informatie aan te vullen

Aan de elementen in het metamodel zijn veel methodes toegevoegd voor het zetten van de fields erin. De meeste fields in de elementen werden bij het aanmaken van een element meteen gezet. Hiervoor werden de waarden ervan aan de constructor meegegeven. Dit was mogelijk, omdat bij het inlezen van Java alle informatie uit de parser over een element beschikbaar is. De MSIL-parser gebruikt echter het metamodel als intern model en heeft niet meteen alle informatie over de elementen beschikbaar. Dit maakt het noodzakelijk dat het zetten van de informatie achteraf plaats kan vinden.

Een voorbeeld hiervan is het aanmaken van een type. Een type krijgt in de constructor een lijst van methodes mee die members van dat type zijn. Als in de MSIL-parser echter een type wordt aangemaakt, is er nog niet bekend welke methodes het type heeft. In een later stadium worden deze methodes gevonden en één voor één toegevoegd aan het type via de methode 'addMethod(GenMethod m)', die de methode toevoegt aan de lijst van methodes.

Herdefiniëren van methodes

Op een aantal punten zijn methodes uit de Gen- elementen van het model hergedefinieerd in de J- en Il- elementen. Een voorbeeld hiervan is de 'isRoot()' methode in GenTypeImpl. Deze methode bepaalt aan de hand van de naam van het type of het type de root in de objecthiërarchie van de ingelezen programmeertaal. In JClassImpl en IlClassImpl wordt deze methode hergedefinieerd, omdat Java en MSIL een verschillende root hebben in hun objecthiërarchie:

- Voor de MSIL is dit: 'System.Object' of '[mscorlib]System.Object'.
- Voor Java is dit: 'java.lang.Object'.

Toevoegen van extra members

Aan een aantal elementen zijn extra members toegevoegd die bij het ontwerp en de implementatie van het metamodel ontbraken. Het gaat hier om de volgende elementen:

- GenEvent
- GenPropery
- GenField

Wanneer een event of een property werkzaam is op een field, wordt dit field bijgehouden in het event of de property. Aan een field wordt in dat geval meegegeven welke acties er door het event of de property gedefinieerd worden waarbij het field mogelijk betrokken is. In GenField zijn er daarvoor de volgende elementen toegevoegd:

- `protected HashTable properties`
  In de HashTable wordt opgeslagen welke methodes er door de property die het field manipuleert, gedefinieerd worden. De opties zijn: get, set en other.
- `protected HashTable events`
  In de HashTable wordt opgeslagen welke methodes er door het event, die met het field werkt, gedefinieerd worden. De opties zijn: add, remove, fire en other.

  Ik heb ervoor gekozen hastables te gebruiken, omdat het aantal methodes dat door een event of property wordt gedefinieerd kan verschillen. Tevens is het mogelijk dat er andere soorten events en properties in andere programmeertalen gedefinieerd kunnen worden op fields, op deze wijze kunnen deze gemakkelijk toegevoegd worden.
- `public void setFieldEvent(String s, boolean b)`
  Voegt in de hashtable 'events' de string als key en de boolean als value toe.
- `public void setFieldProperty(String s, boolean b)`
  Voegt in de hashtable 'properties' de string als key en de boolean als value toe.

- `public boolean hasFieldProperty(String s)`
  Geeft aan of de property met de parameter als key mogelijk gedefinieerd is voor dit field.
- `public boolean hasFieldEvent(String s)`
  Geeft aan of het event met de parameter als key mogelijk gedefinieerd is voor dit field.

Het verwijderen van IlRepository en JRepositiory
We hebben besloten de IlRepository en de JRepository uit het model te verwijderen. De classes definieerden geen andere of extra functionaliteit dan de GenRepository. Wanneer er nu dus een repository aangemaakt wordt is dit een GenRepository, ongeacht de taal die ingelezen wordt.

Afgezien van de bovenstaande punten voldeed het nieuwe metamodel aan de vereisten om de MSIL in te kunnen lezen. Het lijkt er dus op dat we in ons onderzoek naar de MSIL een redelijk compleet beeld hebben weten te krijgen. Voor het testen van de parser, de readers en het metamodel hebben we gebruik gemaakt van de voorbeelden die Microsoft verschaft bij de .Net FrameworkSDK. Deze voorbeelden waren behoorlijk compleet wat betreft het gebruik van de MSIL. Voor de ontbrekende elementen zijn deze voorbeeldfiles zo nodig doelgericht aangevuld.

10. RevMsil

We hebben besloten per programmeertaal een apart programma te maken: RevJava en RevMsil (of RevDotNet). Voor het maken van de verschillende programma's hoeft er slechts het volgende te gebeuren:

- In nl.serc.revs.core.desktop.project.JEApplicationContext moet de string 'resourcebundle' geinitialiseerd worden als 'Project' (voor Msil)of 'Project1' (voor Java). Men kan ook de files die hiermee geassocieerd zijn (Project.properties en Project1.properties) van naam laten verwisselen.

Na compilatie kan er een Jar-file aangemaakt worden met de class-files die de gebruiker op kan starten. We hadden natuurlijk ook kunnen beslissen om runtime te bepalen wat voor taal er ingelezen zou worden, maar dit had als nadeel dat een gebruiker een pad op zou kunnen geven waar zowel class-files als IL-files in zouden staan. Het programma zou dan wel eens vast kunnen lopen doordat er files van de verschillende talen ingelezen worden.

10.1 Voorbeelden van het inlezen van MSIL

Hieronder volgen een aantal voorbeelden van het inlezen van IL-files om een beeld te geven van hoe het programma werkt. Hiervoor zijn twee C# files aangemaakt, die op verschillende manier met elkaar kunnen samenwerken. De twee files zijn:

- MainClient.cs
- FirstClass.cs

De sourcecode van de files wordt hieronder weergegeven:

```
public class MainClient
{
  public static void Main()
  {
    FirstClass fc = new FirstClass();
    fc.Firstclass = ("firstclass called from mainclient");
    fc.PrintMessage();
  }
}
```

Figuur 10.1: class MainClient in Mainclient.cs

```
using System;
public class FirstClass
{
  private string firstclass = "";
  public string Firstclass
  {
      get{return firstclass;}
      set{firstclass = value;}
  }
  public void PrintMessage()
  {
    Console.WriteLine(firstclass);
  }
}
```

Figuur 10.2: class FirstClass in FirstClass.cs

FirstClass definieert een string 'firstclass' met daarbij horende get- en set-properties. Tevens definieert de class een methode PrintMessage die de string 'firstclass' naar het scherm print. MainClient maakt een instantie van een FirstClass aan, zet de string 'firstclass' via de set-property en laat deze naar het scherm printen.

De files worden op twee manieren gecompileerd:

1. Als een single file assembly
2. Als een multiple file assembly

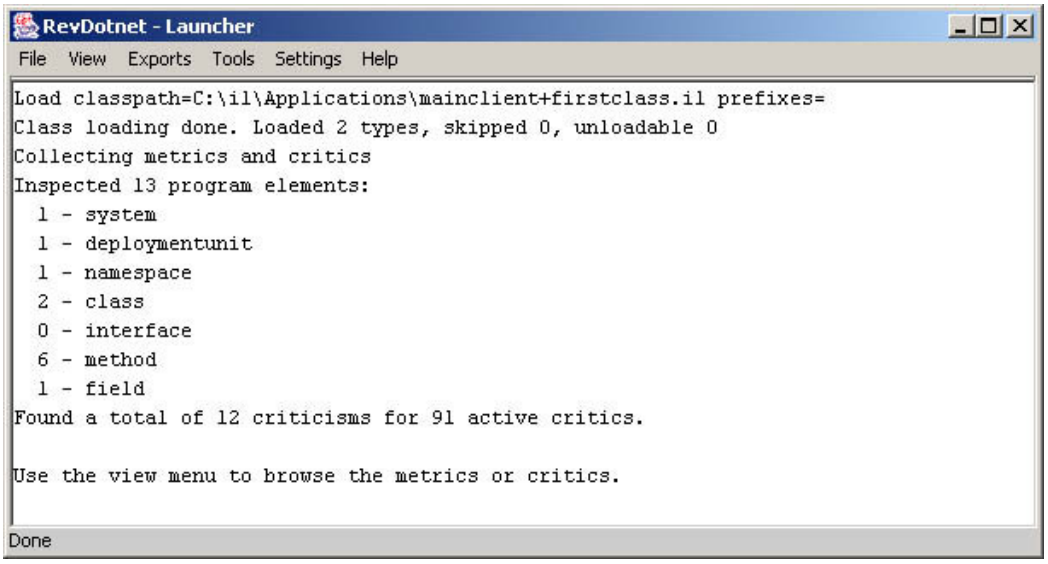
10.1.1 Het inlezen van een single file assembly

De files MainClient.cs en FirstClass.cs worden op de volgende manier gecompileerd:

- `csc mainclient.cs firstclass.cs.`

Het resultaat is een assembly genaamd MainClient.exe. Bij het uitvoeren ervan zal er 'firstclass called from mainclient' komen te staan op de command line. Wanneer men deze assembly deassembleert, zal er een IL-file aangemaakt worden die ik mainclient+firstclass.il heb genoemd. In deze IL-file staat de MSIL van de beide classes gedefinieerd.

We gaan de file mainclient+firstclass.il nu inlezen in RevMsil. Het resultaat wordt door het volgende scherm aangegeven:

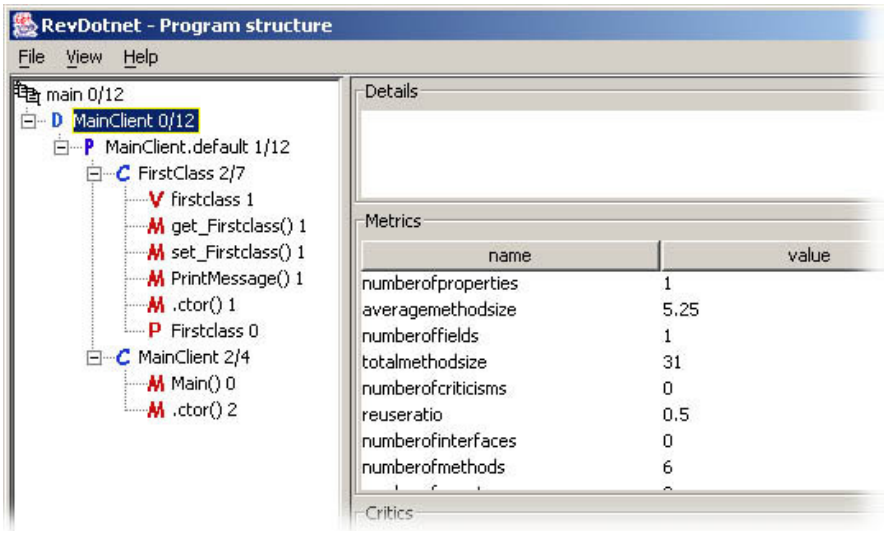


We kunnen ervoor kiezen de ingelezen elementen een aantal verschillende manieren te bekijken. De volgende passeren hier de revue:

- Het bekijken van de programmastructuur.
- Het doorlopen van de inhoud van de repository.
- Het bekijken van de kritieken op de elementen.

De Programmastructuur

Wanneer we nu de programmastructuur bekijken ('View' – 'View program structure') zien we het volgende als we de structuur uitvouwen:



De programmastructuur laat de inhoud van het system 'main' zien. We zien dat de assembly MainClient ingelezen wordt en dat deze de volgende elementen bevat:

- Een namespace default, met daarin:
  - Een class FirstClass, welke de volgende elementen bevat:
    - Een Field firstclass
    - Een methode get_Firstlass()
    - Een methode setFirstClass()
    - Een methode PrintMessage()
    - Een constructor genaamd '.ctor'
    - Een Property Firstclass
  - Een class MainClient, welke de volgende elementen bevat:
    - Een methode Main()
    - Een constructor genaamd '.ctor'

Let op dat de beide classes tijdens de compilatie naar MSIL een constructor hebben gekregen, welke in C# niet gedefinieerd was!

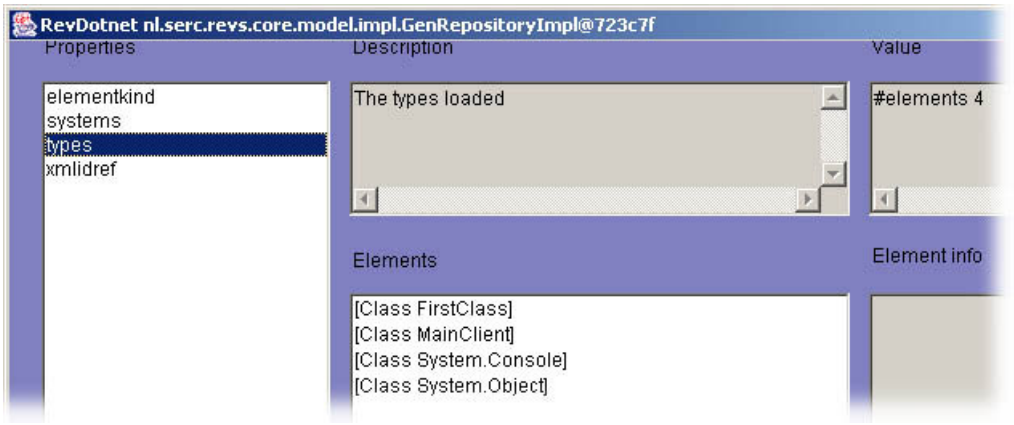
Een namespace wordt aangegeven met 'P', dit is nog de notatie die door het originele RevJava gebruikt werd voor het aangeven van packages. De naam van de namespace bestaat uit de naam van de module waarin hij bevat is en de naam van de namespace zelf.

Fields worden aangegeven met de 'V' van variable, omdat een field gezien kan worden als een variabele die binnen een object geldt. Een 'static' gedefinieerd field kan ook gebruikt worden als er geen instantie van het bevattende object aangemaakt is.

Methodes worden aangegeven met 'M', properties met een 'P'. Wanneer men een element in het menu links selecteert worden de eigenschappen van het object rechts in het scherm weergegeven.

Inhoud van de repository

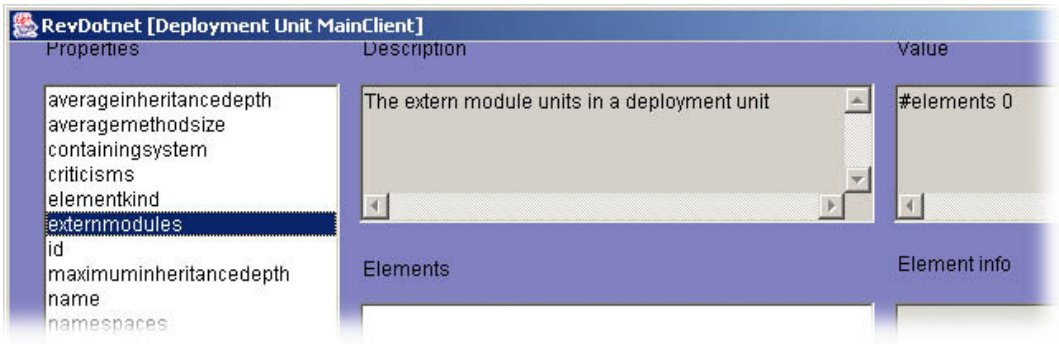
Wanneer we in het hoofdscherm kiezen voor het inspecteren van de elementen in de repository ('View' – 'Inspect repository') krijgen we het volgende te zien:



Bij het selecteren van 'systems' in de lijst links zien we dat er twee systems geladen zijn in de repository: [System main] en [System import]. In de Repository zijn vier types geladen, waarvan er twee in het import systeem gedefinieerd zijn.

We selecteren het onder item 'systems' het systeem [System main] en zien bij het selecteren van 'deploymentunits' dat het system slechts één deploymentunit bevat. In het geval van het inlezen van de MSIL is dit een instantie van een IlAssembly.

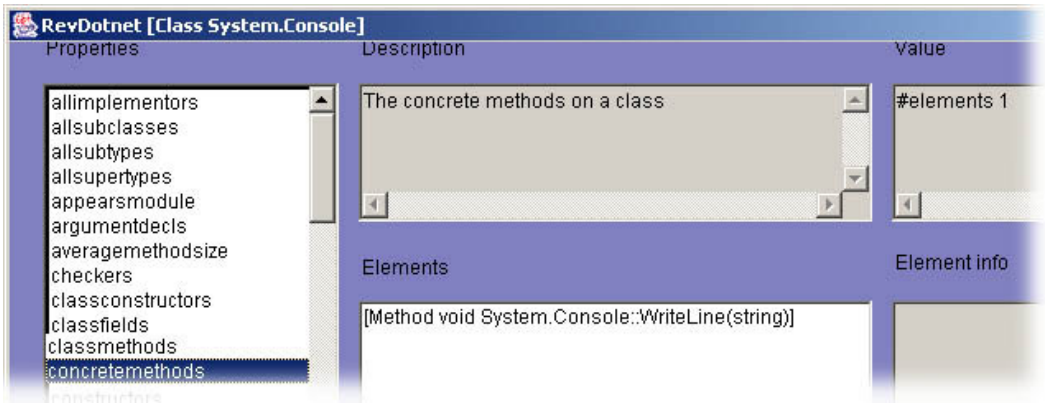
We selecteren de deploymentunit (met als naam MainClient) en zien bij het selecteren van het item 'externmodules' dat de deploymentunit geen gebruik maakt van externe modules:



Door het item 'namespaces' te selecteren kunnen we de rest van de elementen van de ingelezen assembly doorlopen.

Als laatste van het inspecteren van de repository is het wel interessant om een type te laten zien in het import systeem: deze zijn immers aangemaakt door de DefaultTypeDeriver.

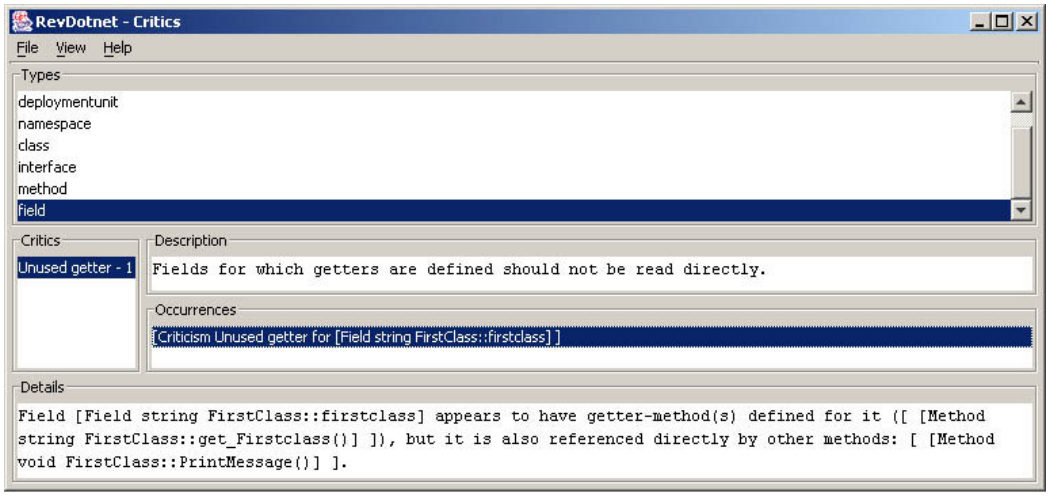
We kiezen voor de class System.Console. We zien dan de onderstaande figuur:



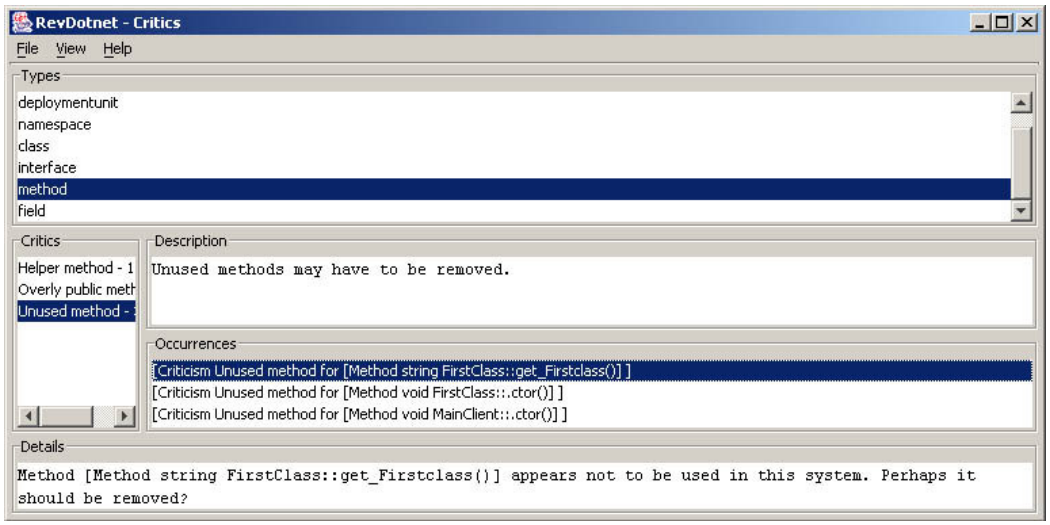
In System.Console is één methode aangemaakt, namelijk methode WriteLine die als parameter een string meekrijgt. Van deze methode is evenwel geen implementatie bekend. Het type is bevat de default namespace van zijn deploymentunit, in dit geval is dat de namespace 'mscorlib.default' in de deploymentunit 'mscorlib'. Dit is de system library van de MSIL, waarin onder andere ook de System.Object gedefinieerd is.

Kritieken op de elementen

In het hoofdscherm kunnen we ook kiezen om de kritieken op de elementen te bekijken ('View' – 'View crtics'). Deze zijn gesorteerd per elementsoort.



Het item 'field' is geselecteerd. We zien dat er op de fields in de ingelezen assembly slechts één kritiek is: het field wordt direct aangeroepen, terwijl er een get-methode voor gedefinieerd is. Dit gebeurt in de methode PrintMessage() van FirstClass.



Wanneer we het item 'method' selecteren zien we onder de critic 'Unused method' dat de betreffende get-methode aangeduid staat als niet gebruikt. Dit is in overeenstemming met de kritiek op het field.

We zien in dit scherm de properties en events niet terug, omdat er op die elementen geen critics zijn gedefinieerd. Wanneer er een critics voor properties en events toegevoegd worden komen deze ook in de lijst te staan.

10.1.2 Het inlezen van een multiple file assembly

Om een multiple file assembly te maken compileren we de files FirstClass.cs eerst naar een .net module:

- `csc /t:module FirstClass.cs.`

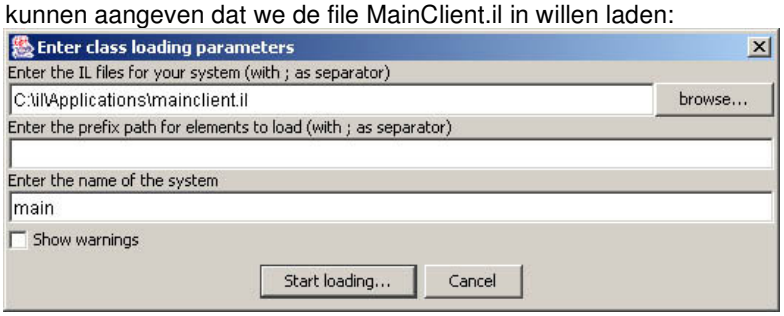
We krijgen dan een file FirstClass.netmodule. Vervolgens compileren we deze module mee wanneer we de MainClient.cs gaan compileren:

- `csc /addmodule:FirstClass.netmodule MainClient.cs`

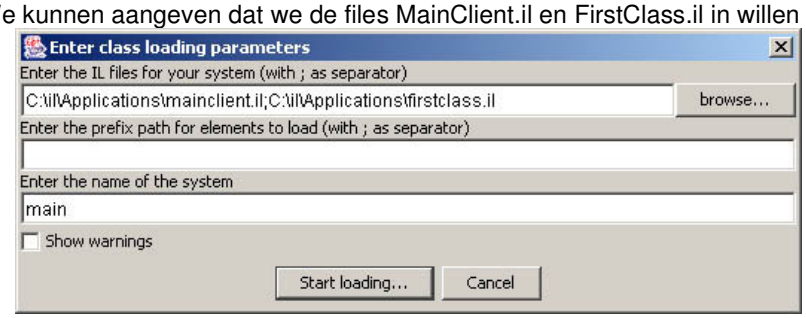
Er is nu een file MainClient.exe aangemaakt, die we deassembleren naar de file MainClient.il. Tevens deassembleren we FirstClass.netmodule naar FirstClass.il in dezelfde directory als MainClient.il staat.

Er zijn nu een aantal manieren om de IL-files in RevMsil in te laden:

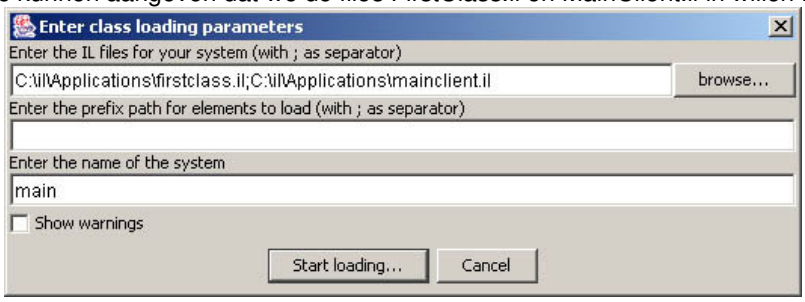
1. We kunnen aangeven dat we de file MainClient.il in willen laden:



2. We kunnen aangeven dat we de files MainClient.il en FirstClass.il in willen laden:

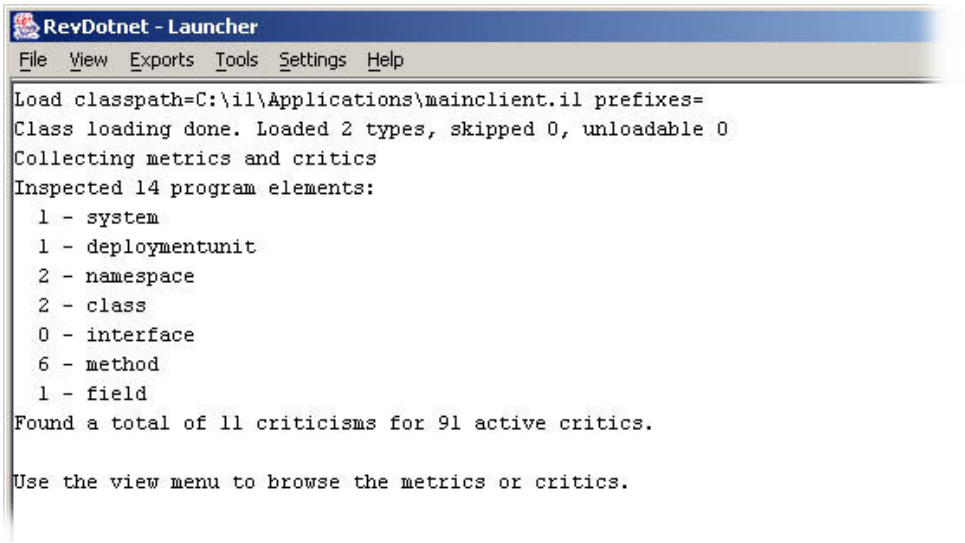


3. We kunnen aangeven dat we de files FirstClass.il en MainClient.il in willen laden:

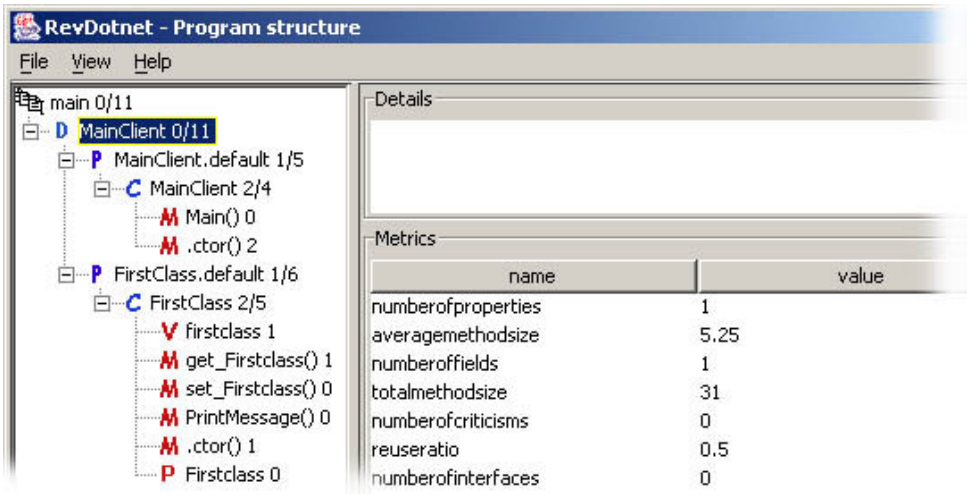


Het maakt voor de opgebouwde structuur in RevMsil niet uit of we de eerste of tweede manier gebruiken, mits de IL-files in het eerste geval in dezelfde directory staan. Immers, wanneer er een externe module gebruikt wordt, wordt er gekeken of deze ook geladen kan worden. Een IL-file wordt echter slechts één keer geladen. Wanneer de module al geladen is al als externe module bij een assembly zal hij niet nogmaals geladen worden.

Het hoofdscherm ziet er na het laden volgens de eerste twee manieren als volgt uit:



Wanneer we het scherm met de programmastructuur openen en de threads uitklappen zijn de volgende elementen zichtbaar:



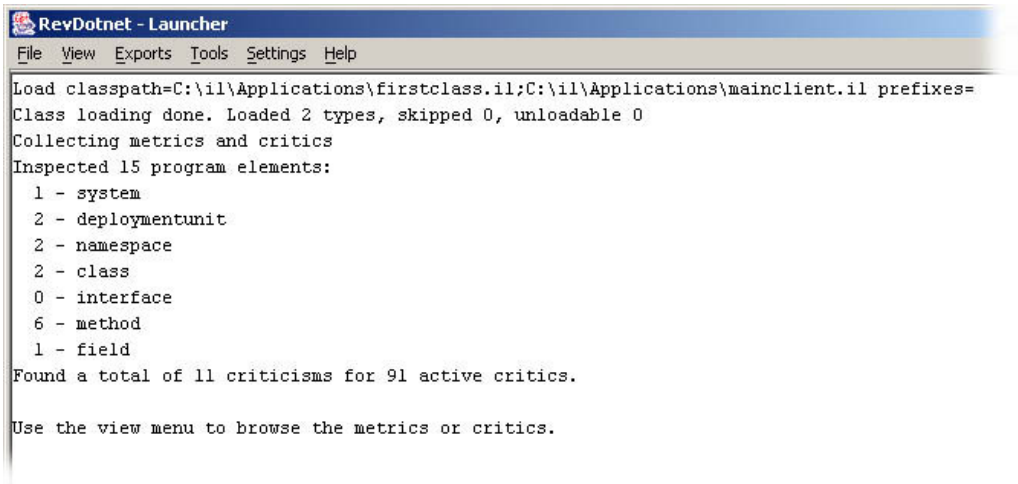
De deploymentunit heeft nu twee namespaces, namelijk de default namespace van zichzelf en de default namespace van de module.

In vergelijking met het laden van de single file assembly is er een kritiek minder op de elementen. Dit heeft te maken met de volgende elementen:

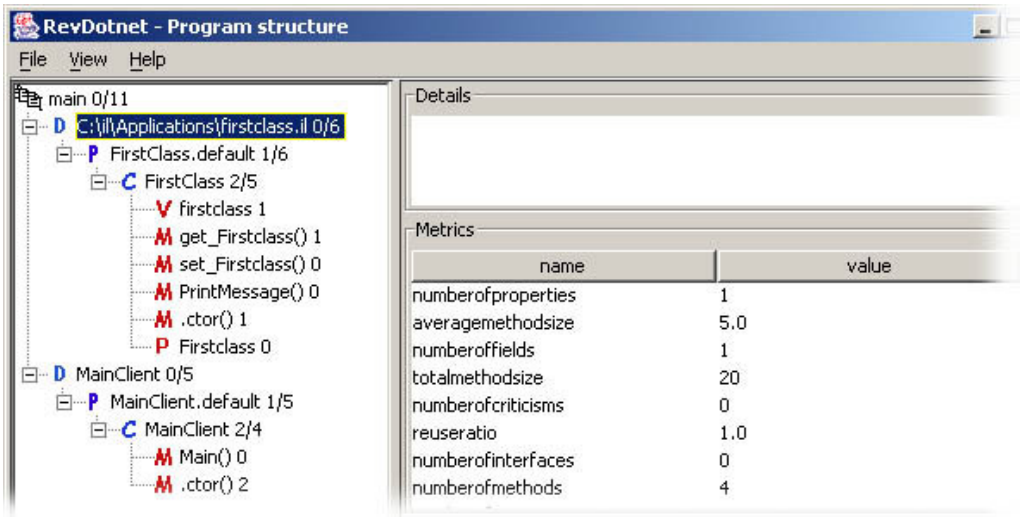
- De methodes set_FirstClass() en PrintMessage() worden nu in een andere namespace gebruikt dan degene waarin zij gedefinieerd zijn, omdat het type MainClient nu niet meer tot de namespace behoort. Hierdoor zijn er twee kritieken minder op methodes.
- Er is een extra namespace bij gekomen waarop een kritiek van toepassing is.

Wanneer we de IL-files inladen door éérst de .netmodule en dan de assembly in te laden, gebeurt er het volgende:

De .netmodule wordt geladen, maar aangezien deze geen assembly-definitie bevat wordt er een default assembly aangemaakt. Dit gebeurt, omdat een module binnen een assembly gedefinieerd moet zijn om betekenis te hebben. Wanneer nu MainClient.il geladen wordt is zijn externe module (FirstClass.il) al geladen, maar in een andere aangemaakt assembly. Dit ziet er als volgt uit:



Er wordt nu inderdaad een extra deploymentunit geladen, maar het aantal kritieken is gelijk aan dat van wanneer we de .netmodule inladen na de assembly. De programmastructuur ziet er als volgt uit:

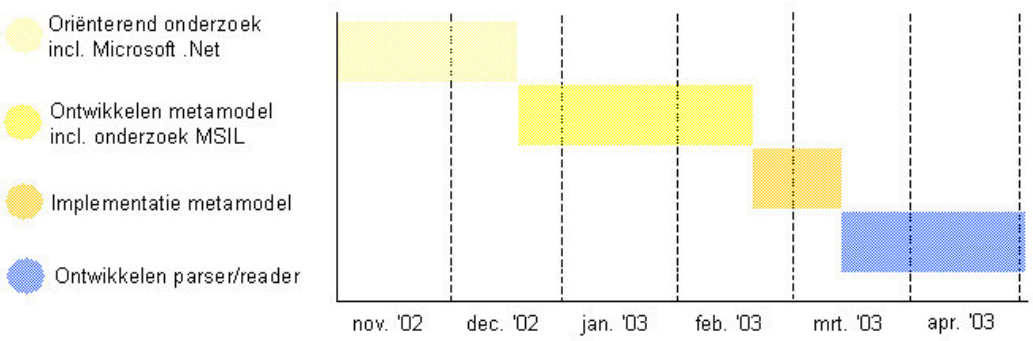


De extra deploymentunit waar de externe module in gedefinieerd is, heeft als naam zijn complete filenaam gekregen. De kritieken wijken niet af van die van wanneer we de .netmodule inladen na de assembly.

Het eerst inladen van een module is handig, als er meerdere assemblies zijn die op een ietwat verschillende wijze gebruik maken van de module. De kritieken op de module zijn dan de combinatie van alle kritieken die voortvloeien uit het gebruik van alle assemblies tezamen.

11. Evaluatie

Gedurende de afstudeerstage zijn er een aantal onderwerpen bestudeerd, met als eindresultaat het ontwikkelde metamodel en de aanpassingen aan RevJava om met dit metamodel te kunnen werken. Hiernaast is het programma aangepast om ook de MSIL te kunnen analyseren. De tijdsindeling van onze stage zag er ongeveer als volgt uit:



Figuur 11.1: tijdsindeling van de afstudeerstage.

11.1 Het oriënterende onderzoek

Het oriënterende onderzoek vormde de basis voor het verdere verloop van het afstuderen. Op basis van de uitkomsten van dit onderzoek hebben we besloten om Microsoft .Net beter te bestuderen en om analyse van de MSIL aan RevJava toe te voegen. Dit hield automatisch in dat we het metamodel zouden moeten aanpassen.

Met het oriënterende onderzoek hebben we een beeld gecreëerd van de mogelijke uitbreidingen voor RevJava. Aan de hand van het verslag dat we hiervan gemaakt hebben kan in de toekomst besloten worden of bepaalde onderdelen die wij niet hebben uitgewerkt nog toegevoegd gaan worden.

Tijdens het oriënterende onderzoek heeft het Famix model ons een ander generiek object-georiënteerd metamodel laten zien. Een interessant item was de splitsing van argumenten en variabelen. We hebben dit ook toegevoegd aan het metamodel van RevJava.

11.2 Het ontwikkelen en de implementatie van het metamodel

Het ontwikkelen van een generiek metamodel vormde een belangrijk deel van de stage. Het metamodel zou de splitsing vormen tussen de twee individuele opdrachten:

1. Het aanpassen van de critics en metrics, zodat deze generiek zouden werken.
2. Het ontwikkelen van de parser en reader voor de MSIL.

De implementatie van het metamodel is eigenlijk onder te verdelen in een aantal fasen:

1. Het definiëren van de Gen- elementen boven de Java-elementen.
2. Het werkend maken van het gehele programma met de Gen- elementen.
3. Het metamodel aanpassen aan de opgestelde architectuur.

De eerlijkheid gebiedt me te zeggen dat Jaco het tweede deel grotendeels heeft gedaan, omdat ik wegens vakantie vijf dagen niet aanwezig was. Het eerste en het derde gedeelte is wel geheel gezamenlijk gedaan.

Het ontwerp van het metamodel is, na een paar kleine aanpassingen tijdens de implementatie, generiek genoeg gebleken om zowel Java als de MSIL in te kunnen lezen. De verwachting is dat wanneer er nog andere talen in het model ingelezen gaan worden het model weinig tot geen aanpassingen hoeft te ondergaan.

11.3 Het inlezen van de MSIL

Het opgestelde model van de parser, met aparte classes voor het parsen van verschillende elementen in het model, bleek achteraf een goede keuze te zijn geweest. Gedurende de implementatie van de parser bleek het vrij eenvoudig het bestaande gedeelte uit te breiden met nieuwe functionaliteit. Tevens was het eenvoudig om definiëren dat elementen op afroep van de readers geparst konden worden. De tool JTopas die gebruikt wordt voor het efficiënt doorlopen van de IL-files werkt naar behoren, mits er minimaal versie 0.6.2 wordt gebruikt. Bij eerdere versies is er al heel snel sprake van performanceproblemen.

Ondanks het goed werken van de parser, kunnen er vrij snel geheugenproblemen optreden bij het draaien van het programma. Dit lijkt te gebeuren door een combinatie van een aantal factoren:

- Veel methodes per type en veel code per methode in de MSIL.
- Veel gebruik van types in externe assemblies in de MSIL.

Wanneer er veel types aangemaakt moeten worden door de DefaultTypeDeriver uit de externe assemblies treden er vrij snel performance problemen op. Ditzelfde geldt wanneer er veel methodes in een type gedefinieerd zijn en deze methodes veel statements bevatten.

De parser dekt het grootste gedeelte van de MSIL, maar nog niet alle mogelijke code wordt herkend. Dit heeft mede te maken met het feit dat deze code meestal niet uit C# te compileren is en het erg moeilijk is om echt complete voorbeelden van MSIL code te vinden.

Ondanks de bovenstaande performanceproblemen en het niet geheel in kunnen lezen van de MSIL, is er een werkend programma ontwikkeld. Dit programma kan zowel de MSIL als Java bytecode op een generieke manier opslaan en analyseren. Hiermee is aangetoond dat ons ontwikkelde metamodel generiek genoeg is om minimaal geschikt te zijn voor Java en MSIL.

Het ontwikkelde programma moet niet direct worden gezien als een product dat in aanmerking komt voor een release. Mogelijk kunnen docenten het gebruiken om kleine practicumopdrachten in .net op een snelle manier te scannen op de ingebouwde kritieken, maar voor grote applicaties is het zeker nog niet geschikt. Het meest voor de hand liggend is het programma te beschouwen als een werkend prototype, dat met een andere (MSIL bytecode) parser en aangepaste readers een rol kan gaan spelen in het ontwikkelen van .net software.

11.4 Still to do

Er zijn nog een aantal punten waaraan in het programma verbeteringen kunnen of moeten plaatsvinden, maar waar wij geen tijd meer voor hadden:

- Het aanmaken van JJars: er wordt in het programma voor het inlezen van Java slechts één JJar aangemaakt voor alle classes die ingelezen worden, zelfs als de classes eigenlijk in verschillende Jar-files zitten.
- Een JTagFactory om voor Java ElementTags aan te maken: voor MSIL is er de MsilTagFactory die voor de elementen de juiste ElementTags aan maakt. Dit zou er eigenlijk ook voor Java moeten komen.
- Een DefaultJTypeDeriver: het kan ook interessant zijn om voor Java de unresolved types aan te maken. Hiervoor moet er een DefaultJTypeDeriver gedefinieerd worden.

Het toevoegen van de verbeteringen kan ook een herziening van de packagestructuur met zich mee brengen. Het kan nuttig zijn dit samen te laten vallen met het toevoegen van de nieuwe MSIL-parser en het aanpassen van de readers, zodat het programma optimaal opgebouwd is.

Verklarende woordenlijst

Critic	Een critic is een soort functie die gebruik maakt van eigenschappen van een element in het model en checkt of de regel, die door de functie gerepresenteerd wordt, overschreden wordt.
Field	Een variabele die bestaat in de context van een type. Een static gedefinieerd field kan ook gebruikt worden zonder dat er een instantie van het bevattende type is aangemaakt.
IL-file	Een file die gedeassembleerde MSIL code bevat.
Metric	Een aantal properties die gebruikt kunnen worden door de critics om bepaalde situaties in de software op te sporen.
MSIL	De Microsoft Intermediate Language. Sourcecode welke gecompileerd wordt door de compilers welke bij het .Net Framework horen, wordt gecompileerd naar MSIL code.
Property	Een property kan in twee contexten voorkomen: - in context tot RevJava / RevMSIL. - in context tot de programmeertaal die ingelezen wordt. In het eerste geval kan een property gezien worden als een eigenschap die bij een bepaald element in het metamodel hoort. In het tweede geval betreft het een manier om elementen van een type te benaderen.
.net applicatie	Een programma dat is ontwikkeld met behulp van het .Net Framework.
Resolved types	Bij het inlezen van gecompileerde code worden er types gevonden en ingelezen in het metamodel, deze types noemen we ‘resolved’.
Token	Eén of meerdere characters waarop een tokenizer een tekstfile doorloopt.
Tokenizer	Met behulp van een tokenizer kan een tekstfile doorlopen worden aan de hand van op te geven tokens. Op de positie waar een opgegeven token zich bevind is de inhoud van de tekstfile op te vragen.
Unresolved types	Bij het inlezen van gecompileerde sourcecode kan het zijn dat er types niet gevonden worden, deze types noemen we ‘unresolved’.

Literatuur

RevJava
Website: http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm

[1] RevJava Design Overview:
Gert Florijn, februari 2001.
[2] RevJava whitepaper:
Website: http://www.serc.nl/people/florijn/papers/RevJava-overview-recent.pdf
Gert Florijn.
[3] Documentatie voor de BCEL class load library:
Website: http://bcel.sourceforge.net
Markus Dahm.

Oriënterend onderzoek
Famix model
Website: http://iamwww.unibe.ch/~famoos/

[4] FAMIX 2.0. The FAMOOS Information Exchange Model:
Demeyer, Tichelaar en Steyaert, 1999.
[5] FAMIX Java language plug-in 1.0:
Sander Tichelaar, 1999.

Alias analyse
[6] Flexible Alias Protection:
Ecoop 1998: http://www.ifs.uni-linz.ac.at/~ecoop/cd/html/contents.htm
James Noble, Jan Vitek, John Potter
[7] Demand Driven Pointer analysis:
Website: http://citeseer.nj.nec.com/heintze01demanddriven.html
Nevin Heintze, Olivier Tardieu.
[8] Scale: A Scalable Compiler for Analytical Experiments:
Website: http://www-ali.cs.umass.edu/Scale/
[9] Points-to Analysis in Almost Linear Time:
Bjarne Steensgaard, POPL 1996.

Type analyse
[10] Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler:
Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani.
[11] Improving Java ™ Programming Language Code Using Static Analysis:
Tom Ball.
[12] Applying Static Analysis to Large-scale, Multi-threaded Java Programs:
Cyrille Artho and Armin Biere.
[13] The Complexity of Type Analysis of Object Oriented Programs:
Joseph (Yossi) Gil and Alon Itai.
[14] A generic approach of static analysis for detecting runtime errors in Java Programs:
Xiaoping Jia and Sotiris Skevoulis.
[15] A system and language for building system-specific static analysis:
Seth Hallem, Benjamin Chelf, Yichen Xie, Dawson Engler.

SOUL
Website: http://www.iam.unibe.ch/~wuyts/Soul/

[16] Understanding Object-Oriented programs with declarative event analysis: Tamar Richner, Stéphane Ducasse, Roel Wuyts
[17] Declaritive reasoning about the structure of object-oriented systems: Roel Wuyts
[18] Synchronising changes to design and implementation using a declaritive meta-programming language: Roel Wuyts

Meta compilatie
Website: http://www.stanford.edu/~engler/

[19] Checking System Rules using System-Specific, programmer-Written Compiler Extensions: Engler, Chelf, Chou, Hallem.
[20] Using meta-level Compilation to Check FLASH Protocol Code: Chou, Chelf, Engler, Heinrich.
[21] Using Programmer-Written Compiler Extensions to Catch Security Holes: Ken Ashcraft, Dawson Engler

Microsoft .Net

Website: http://www.microsoft.com/net

[22] http://www.microsoft.com/library
[23] http://msdn.microsoft.com
[24] http://www.microsoft.com/india/indiadev/projects/
[25] http://msdn.microsoft.com/library/en-us/csspec/html/vclrfcsharpspec_2_3.asp

[26] http://www.ecma.ch
[27] http://www.gotdotnet.com
[28] http://www.dotnetextreme.com
[29] http://www.mastercsharp.com
[30] http://www.iunknown.com/Weblog/fog0000000110.html
[31] http://www.devx.com/dotnet/articles/lp100901/lp100901.asp
[32] http://www.oreillynet.com/pub/a/dotnet/2001/10/15/jsharp.html
[33] http://www.javaworld.com/javaworld/jw-11-2001/jw-1121-iw-jsharp.html
[34] http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double-p2.html

[35] SDGN (Software Developers Group Netherlands), magazine nr. 70, maart 2002: '.Net? IT opnieuw op rails gezet', Patrick Vletter.
[36] Microsoft .Net Magazine: .Net Framework
[37] Computable Magazine, uitgave april 2003

Java versus .Net
[38] http://www.soften.ktu.lt/~mockus/gmcsharp/csharp/c-sharp-vs-java.html
[39] http://www.unf.edu/~rzucker/cop3540dir/modifiers.html
[40] http://www.jot.fm/issues/issue_2002_11/article4

Ontwerp van het nieuwe metamodel
Voor het opstellen van het nieuwe metamodel hebben we kennis gebruikt die we in de voorgaande onderzoeken opgedaan hadden. Tevens is er nog een extra model bekeken:

[41] Models for reverse-engineered artifacts:
http://www.uni-koblenz.de/~riediger/public_files/research/dagstuhl01041/lethbridge-talk.pdf
Timothy C. Lethbridge, januari 2001.

Het inlezen van de MSIL

Voor het inlezen van de MSIL is er gebruik gemaakt van de tool JTopas en de documentatie die daarbij hoorde:

[42] Website: http://jtopas.sourceforge.net/jtopas/index.html