universiteit hasselt

2e bachelor informatica-ict

Academiejaar 2005–2006

Trimester overschreidend project
.NET YAML Parser

Groep 3

Jonathan Slenders (0421645)

—

Christophe Lambrechts (0421611)

Inhoudsopgave

In dit eind verslag zullen we delen uit het *Analyse verslag* terug aanhalen (geindenteerd), aangevuld met commentaar en gegevens die we hebben opgedaan tijdens het maken van het project.

1 Opgave

Een korte samenvatting van de opgave:

> Yaml [1] [10] is een zeer leesbaar en makkelijk te bewerken data serialisatieformaat. Het kan gezien worden als een light-weight alternatief voor XML. Yaml is opgebouwd rond het idee dat alle data voorgesteld kan worden met combinaties van lijsten, hashtables en scalaire data. Yaml kan makkelijk gemapt worden op data types van de meeste high-level programmeertalen. Er zijn parsers beschikbaar voor scripting talen zoals Python, PHP, Perl en Ruby, alsook een Java implementatie [3]. Deze implementaties gebruiken vaak onderliggend ook Syck [9], een snelle implementatie in C. Voor het .NET platform is er echter nog geen native parser beschikbaar. De studenten doen ervaring op met het .NET framework en de programmeertaal C# en leren een specificatie te bestuderen en te implementeren. Het is de bedoeling het uiteindelijke resultaat als een open source library vrij te geven.

2 Contact personen

- Jo Vermeulen (jo.vermeulen@uhasselt.be)
- Tom Van Laerhoven (tom.vanlaerhoven@uhasselt.be)

3 Diepgaande beschrijving van het onderwerp

> Zoals reeds vermeld in de opgave (sectie 1), zullen we een parser implementeren in C# voor Yaml (klinkt als 'camel'). We zullen ons baseren op bestaande parser in Ruby die gebruik maakt van Syck. Dit houdt in dat we trachten in grote maten de toegang tot de library en de functies te uniformeren. Bij de implementatie van het project zullen we van nul beginnen. De hoofddoelstelling is propere en goed leesbare code te schrijven via de object-geöorienteerde principes van C#, dit in tegenstelling tot Syck, die sterk is geoptimaliseerd, met als nadeel dat men moet in boeten tegen netheid en leesbaarheid. Syck gebruikt bijvoorbeeld veel sprong instructies, iets wat sterk indruist tegen een goede programmeer stijl. C# is een sterk objectgeoriënteerde taal, we zullen dus steeds zoveel mogelijk van deze OO-structuren gebruik maken.

We zijn er van overtuigd dat de code goed gestructureerd is en dat deze uitermate geschikt is om als Open Source verder te leven. Onze belofte om de Ruby interface aan te houden is niet 100% bereikt daar we niet voldoende tijd hebben gehad om Ruby aan te leren. Het serializeren en deserializeren is

[1] YAML Ain't Markup Language

echter zo eenvoudig dat het naar ons inziens geen noodzaak is dat alle Yaml parsers eenzelfde interface hebben. Verder heeft elke programmeertaal eigen specifieke structuren die best kunnen worden gebruikt. De doorgedreven OO-structuur is goed zichtbaar in het UML-diagram (sectie 4.1).

> Bij het ontwerp van de parser zullen we zoveel mogelijk de officiële Yaml 1.1 specificatie [11] volgen. We stellen echter niet als doel deze volledig te realiseren, temeer omdat er momenteel discussies op gang zijn gekomen om Yaml te vereenvoudigen. De belangrijkste elementen zullen uiteraard werken, later kunnen nog details worden toegevoegd.

De basis functionaliteit van een Yaml parser is naar ons inziens bereikt. Wel moeten we hierbij opmerken dat we op bepaalde punten keuzes hebben moeten maken waarover geen duidelijkheid is/was in de 1.1 specificatie. We denken hierbij aan de al dan niet verplichte spatie achter de ':' van een mapping *(Post: 2006-04-14 19:19)*. Verder is er op de mailing list [7] een discussie over de compatibiliteit met JSON. Tenslotte gaan er nu geruchten op om een meer definitieve versie van 1.1 te maken die gebaseerd is op de huidige parsers voor Yaml *(Post: 2006-05-24 14:11)*. Hopelijk mag onze ook zijn steentje bijdragen.

> Onze Parser zal uit twee delen bestaan. Ten eerste kan deze Yaml bestanden inlezen uit een bestand (of uit een string) en vervolgens *parsen*. Ten tweede zal deze ook een gecreëerde of geweizigde datastructuur terug naar een Yaml-bestand (of naar een string) kunnen uitschrijven (*generator*).

Door voor alle types een eigen class te schrijven (wrapper class voor de native types) beschikken we over extra info die het deserializeren vereenvoudigd. De serialisatie procedure kon op deze manier zeer snel worden geïmplementeerd.

> Verder zullen we ook een beperkte test applicatie schrijven waarbij eenvoudig de werking van de parser geëvalueerd kan worden. Vanzelfsprekend voorzien we in een set van voorbeeld Yaml documenten. Om ook het praktische nut te kunnen aantonen van Yaml zullen we onze activiteiten logs en onderlinge communicatie opmaken in Yaml.

De test applicatie is gewijzigd in een verzameling van voorbeeld toepassingen. Dit geeft ons de mogelijk meerdere kleine programma's aan te bieden. Deze helpen bij het leren kennen van de parser als ook bij het verdere ontwikkelen. Voor het resultaat van onze Yaml experience (logs en onderlinge communicatie) verwijzen we naar de bijlage A.2 en A.3 (Deze logs zijn niet volledig. Door nu en dan te vergeten de logs te gebruiken beschrijven ze maar gedeeltelijk wanneer we wat gedaan hebben en kunnen deze een vertekend beeld van onze inspanning geven.)

> Buiten deze technische details willen we ook graag aandacht vestigen op andere aspecten van programmeer projecten. Zo gebruiken we CVS om code en

project gerelateerde documenten te centraliseren. De ingebouwde documentatie mogelijkheden van C# en het .NET Framework, gebaseerd op XML zullen we benutten, dit ter vervanging van Doxygen in eerdere projecten. De 'automatisch' gegenereerde documentatie zal ook van pas komen in het Open Source aspect van dit project. We zijn ons er echter van bewust dat deze nooit volledig is en enkel een hulp biedt voor de developpers. We zullen de documentatie daarom uitbreiden met een eigen handleiding. Om het Open Source project zoveel mogelijk kansen te bieden zullen we trachten de code compatibel te maken met zowel het .NET Framework (Windows) als met Mono (voor Linux, Mac OS, ...). Mogelijk zal deze ook in het .NET Compact Framework werken.

Reeds in een heel vroeg stadium hebben we besloten om alle commentaar en documentatie in het Engels te schrijven. In plaats van de ingebouwde .NET documenter te gebruiken hebben we gekozen voor NDoc [6]. De documentatie is te vinden op `http://lumumba.uhasselt.be/~christophe/YAML/`, samen met een manual/overview van het project. Bij de NDoc documentatie zijn ook all private members inbegrepen.

De platform onafhankelijkheid is in zekere maten bereikt. Zo heeft Jonathan het volledige project onder Linux gemaakt, gebruik makend van Mono. Christophe heeft gewerkt onder Visual Studio en zo kunnen we garanderen dat de parser onder deze twee platformen werkt. Het .NET Compact Framework is nooit getest, maar we verwachten hier geen problemen.

4 Analyse

4.1 Klasse-diagram

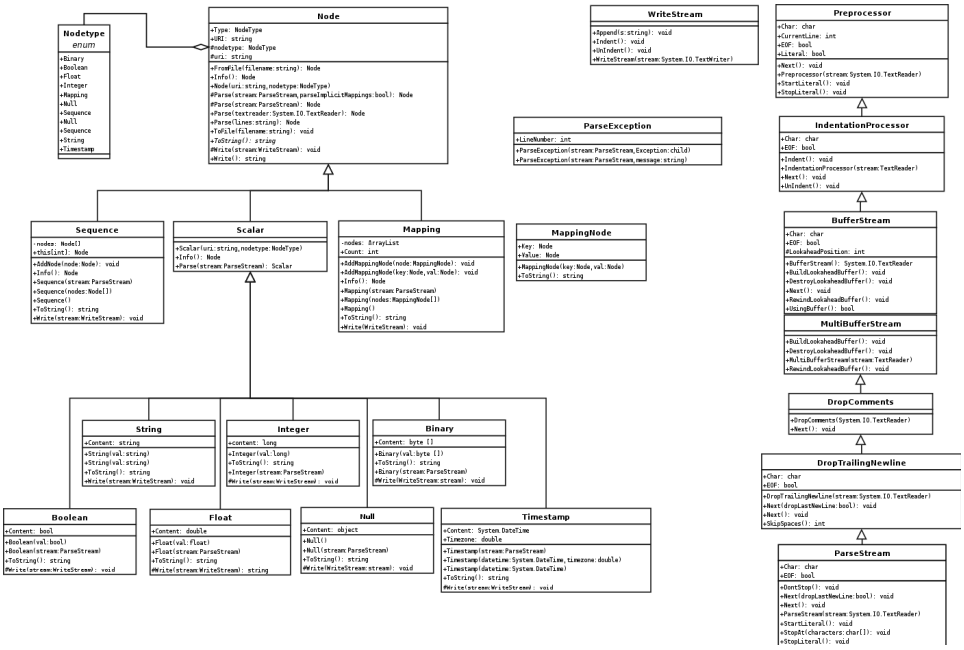
Buiten de algemene boomstructuur merken we ook een *stream* classe die eigenlijk de toegang is voor de gebruiker. Belangrijk is hierbij op te merken dat er zowel 'communicatie' mogelijk is via file I/O als via strings.

Verder zijn de classe voor de test applicatie hier niet gerepresenteerd. Deze zullen minimaal van opbouw zijn en behoren strikt gezien ook niet tot de parser.

De *parse stream* class is niet meer direct toegankelijk voor de gebruiker, deze moet nu enkel nog met de class *node*, of een afgeleide class daarvan communiceren. De *parse stream* vormt intern voor de parser een interface tot de stream, welke via een node zou kunnen zijn meegegeven om te parsen. De *parse stream* voorziet verschillende transformaties die het parsen kunnen abstraheren.

Een definitief UML diagram kan gevonden worden in figuur 1 en een grotere versie in bijlage op blz. 15. Bij de evaluatie van het Analyse verslag werden enkele suggesties gedaan die de structuur zouden kunnen verbeteren.

De opmerking om de (Parse)Stream class afteleiden van IO.stream hebben we in overweging genomen. Maar gezien onze specifieke noden was dit



Figuur 1: Definitieve UML diagram voor YAML .NET Parser (Grote versie in bijlage, blz. 15)

niet voor de hand liggend. Wel is er voldoende compatibiliteit voorzien zodat van een StreamReader kan worden geparst.

Verder heeft Jo ons een Set class aangeboden. Wegens het gebrek aan tijd hebben we deze niet nodig gehad.

Er werd ons ook aangeraden de anchors in een aparte klasse te abstraheren. Daar we ook hier niet aan toe zijn gekomen zien we hier niets van terug in het definitieve UML diagram. Het is wel duidelijk dat er geen beperkingen zijn om dit later alsnog te doen.

Tenslotte merken we enkele hulp classes op zoals *ParseException* en *Nodetype* die oorspronkelijk niet waren voorzien. De test classes waarover we sproken in het analyse verslag hebben we gegroepeerd onder de 'Examples'.

4.2 Verantwoording van de keuze van ADT's

4.2.1 Algemene datastructuur

> Onze datastructuur van een Yaml document zal uit een boom van nodes bestaan. De wortel is het Yaml document zelf, dit kan een mapping, sequence of scalar zijn. (Een bestand kan meerdere Yaml documenten bevatten, in dat geval hebben we een ArrayList[2] van documenten.
>
> Elk element wordt voorgesteld door een classe die van de class *Node* erft. In het geval van een *Mapping* zal elk item in een ArrayList opgeslagen worden

[2]het .NET alternatief voor een vector

waarbij voor elk item zowel een key als de data gespecifieerd worden. Zowel de key als de data kan eender welk type *Node* zijn, zelfs een andere *Mapping*.

Een *Sequence* zal ook opgeslagen worden als ArrayList, maar in dit geval is er enkel een value, ook van het type *Node*. *Scalars* zijn primaire types, de bladeren van de boom, deze erven wel van *Node*, maar kunnen zelf geen kinderen hebben. We gaan verschillende klassen gebruiken die van *Scalar* erven, om onder andere types zoals date, string en numerieke waarden te implementeren.

4.3 Algoritmes

We zullen trachten een one-pass parser te maken, vermits bij het ontwerpen van Yaml hier extra aandacht aan besteed werd. Het maakt de implementatie misschien niet alleen eenvoudiger, maar ook sneller. De latere implementatie kan mogelijk afwijken omwille van een beter inzicht in de structuur die aantoont dat een ander algoritme efficiënter is of meer volledig.

Onze parser is nog steeds zoals gepland een one-pass parser.

4.3.1 Enkele niveaus in de parse stream

De Parse stream heeft een gelaagde opbouw, elke laag bouwt verder op de vorige laag en voegt extra functionaliteit toe.

1. Preprocessor
2. Indentatieprocessor
3. Buffer
4. Geneste buffer
5. Comments verwijderen
6. De laatste newline verwijderen
7. Inline processor (Stopt op een gegeven karakter)
8. Literal parsing (verbatim)

De preprocessor houdt bij op welke regel dat we zitten, dit kan later gebruikt worden om bij het uitschrijven van een fout het regelnummer te vermelden. Voor het uitschrijven van fouten hebben we onze eigen Exception class voorzien.

We gebruiken veel recursie om alles te parsen. Het tweede niveau, de indentatieprocessor, zorgt ervoor dat een genest of geindenteerd stuk Yaml net op dezelfde manier kan geparsed worden als een stuk code op het root-niveau.

Onze parser is bijna volledig een one pass parser. We lezen van een stream en we bewegen ons enkel voorwaarts in de stream. Enkele Yaml

structuren vereisen wel een beperkte vorm van lookahead. Soms moet parser even proberen of hij de volgende node als een bepaald type kan parsen, zo niet moet hij de buffer kunnen terugspoelen en een ander type proberen. Wij voorzien een circulaire buffer met een grootte van 1024 karakters wat volgens de documentatie is toegelaten. Omwille van de vaste buffergrootte en de werking van de buffer kunnen we de parser nog steeds een one-pass parser noemen.

Soms hebben we zelfs een bufer nodig op een moment dat we al binnen een buffer aan het werken zijn. De vierde laag maakt geen buffer bij maar onthoudt de startpositie binnen de huidige buffer.

We gebruiken een extra laag die de laatste newline verwijderd. Dit karakter behoort niet tot de content maar is enkel nodig omwille van de stuctuur waarmee Yaml is opgebouwd.

Om eenvoudig inline structuren te parsen hebben we nog een extra laag toegevoegd. Hierin kan ingesteld worden dat op een bepaald karakter gestopt moet worden. Inline sequences bijvoorbeeld starten met een "[" en eindigen met "]".

Strings die tussen enkele quotes staan moeten letterlijk genomen worden, dit houdt in dat newlines, tabs en spaties die hiertussen staan ook tot de werkelijke content van de string behoren. Het gedrag van de bovenliggende lagen moet ongedaan gemaakt worden. Dit gebeurt hier met de methoden StartLiteral en StopLiteral. Deze funtionaliteit is in werkelijkheid niet in een aparte laag geimplementeerd, maar wordt door de verschillende lagen geimplementeerd.

We gebruiken veel samenhorende methoden, deze moeten altijd samen aangeroepen worden. Elke aanroep van Indent moet gepaard gaan met een latere aanroep van UnIndent, hetzelfde voor StopAt en DontStop, StartLiteral en StopLiteral.

De meeste van deze lagen definiëren de volgende functies:

Next Ga naar het volgende karakter

Char Geef het huidige karakter

EOF (End of file) Geeft true terug indien we op het einde van deze (deel)stream zitten.

4.3.2 Indentatie processor

Indentatie is heel belangrijk in Yaml. Een regel die geen voorafgaande spaties heeft stelt een node voor, regels die volgen én meer geïndenteerd zijn dan de vorige regel, die zijn op de één of andere manier geassocieerd met die node. Mogelijk vormen deze een blok dat een child van die node vormt. In dat geval zal de indentatie van deze child node verwijderd worden. Het aantal spaties dat verwijderd moet worden is het maximaal aantal spaties dat bij iedere regel van dat blok voorkomt, sommige regels zullen dus nog steeds

voorafgaande spaties bevatten. Op een recursieve manier kunnen we nu dit blok parsen.

Dit is niet volledig waar. Enkel de eerste regel van een node bepaald hoeveel spaties deel uitmaken van de indentatie.

We hebben twee belangrijke methoden: Indent en UnIndent. De eerste geeft aan dat we in een kind-node gaan welke waarschijnlijk maar niet noodzakelijk meer geïndenteerd is dan de parent node. Zoals geïllustreerd in figuur 2.

```
key:
  value
```

Figuur 2: Voorbeeld van indentatie

Bij het aanroepen van de *Indent* functie - wat steeds gebeurt bij een overgang van parent naar child node - wordt een *indentation-request* variabele geset. Wanneer we ons later verder verplaatsen in de stream en een newline karakter tegenkomen, dan gaan we naar de volgende regel en tellen het aantal indentatie karakters. Indien dit er meer zijn dan het aantal van de vorige regel, dan zitten we nog steeds in de child node. Zoniet (evenveel of minder indentatie), dan stopt hier het child blok en wordt EOF hier voorlopig op true gezet. Bij de aanroep van Indent wordt het huidige indentatie niveau op een stack geplaatst, zodat dit bij een *UnIndent* terug kan worden hersteld. Het aanroepen van *UnUndent* zal in indentation-request ongedaan maken indien we nooit een newline tegenkwamen, of anders het laatste indentatie niveau van de stack poppen. OEF wordt hierbij mogelijk ook terug of false geset.

4.3.3 De buffer

We gebruiken een circulaire buffer van één kilochars. Een lookahead van meer dan 1024 karakters is niet mogelijk, maar dit is niet vereist indien we de ristrictie kunnen opleggen dat een key van een impliciete mapping niet groter mag worden dan 1024. We hebben drie balangrijke integer variabelen: de rotatie, de positie en de grootte. De rotatie wijst naar de eerste cel van het stukje gebruikte buffer. Positie is de offset binnen dit stukje waar we momenteel aan het parsen zijn. Grootte is de grootte van dat stukje buffer.

Wanneer we *BuildLookaheadBuffer* aanroepen dan wordt de buffer geinitialiseerd, tenzij deze al eerder in gebruik was. Een *DestroyLookaheadBuffer* verwijdert de buffer niet meteen maar gaat de buffer enkel alle karakters voor het huidige karakter laten vergeten. Hiermee kunnen we het toestaan om de destroy functie aan te roepen terwijl we nog midden in de buffer zitten. Na een aanroep van destroy zitten we nog steeds op hetzelfde karakter. Elke aanroep van Next zal telkens de positie pointer een plaats in de buffer doen opschuiven en weer het vorige karakter doen vergeten. Dit blijft zich

herhalen totdat we aan het einde van de buffer komen en terug in de normale flow van de stream komen.

Geneste buffers realiseren we door bij elke aanroep van BuildLookaheadBuffer de huidige positie op een stack te plaatsen indien de buffer al in gebruik was. Een rewind zal dan naar deze positie, en niet naar de allereerste positie springen. De Destroy functie zal de laatste positie van de stack poppen.

4.3.4 Inline processor

Een geopende accolade geeft het begin van een inline mapping aan. Als we vanaf daar verder willer parsen moet de recursieve functie die de childnode gaat parsen stoppen op een komma, wat het begin van een nieuw item is, maar ook aan bij de bijhorende gesloten accolade. Aan de functie *StopAt* kunnen we een karakter array meegeven. Deze zal telkens dat we op één van deze karakters komen de EOF op true plaatsen, totdat de *DontStop* functie wordt aangeroepen. We gebruiken weer een stack om het gedrag van de parent te overschrijven. Hierbij is het ook mogelijk - in tegenstelling met het aanroepen van een Indent - om de stream te verlengen door bijvoorbeeld een leeg array mee te geven.

4.3.5 Het type node bepalen

> Het herkennen van een sequence is eenvoudig omdat deze met een "–" (liggend streepje of dash) begint. Een mapping bevat altijd een ":" (dubbele punt). In moeilijke gevallen (Bijvoorbeeld wanneer de key zelf een mapping is) wordt de key door een "?" (vraagteken) voorafgegaan wat het parsen vereenvoudigt. Voor het parsen is nog niet bekend om welk type node het gaat, dus indien dit geen mapping of sequence is, kan het net zo goed nog een scalar zijn.

Bepaalde types van nodes kunnen we al onmiddellijk aan de eerste karakters herkennen. Bij nodes die beginnen met twee uitroeptekens gevolgd door de typenaam kan meteen de parse functie van dat type nood aangeroepen worden. Het liggend streepje aan het begin duidt op het volgen van een sequence, net zoals een vraagteken wijst op een mapping. Andere types vereisen lookahead door middel van een buffer. Verder wijzen een open accolade of vierkantig haakje op respectievelijk een inline mapping en inline sequence.

Het eerste wat we nu moeten controleren zijn inline mappings. De enige manier waar deze aan te herkennen zijn is de dubbele punt die na de eerstvolgende scalar volgt. Dit houdt in dat we eerst deze scalar moeten parsen, en daarna zien of er een dubbele punt volgt. Volgt er geen dubbele punt, dan moeten we terugspoelen tot voor deze scalar en terug alle soorten scalars uitproberen.

4.3.6 Scalars

> Indien na het parsen van de structuur bekend is dat een bepaald stukje tekst een scalar voorstelt, moet dit geparst worden. Hiervoor hebben we een methode in de class *scalar* die bepaalt om welk type scalar het gaat (datum, float, ...), en daarna een instantie van de gepaste subclass teruggeeft.
>
> Veel kan al bepaald worden door het eerste karakter. Scalars die met een enkele quote — ' — beginnen zijn ongeescapete strings, een dubbele quote — " — geeft het begin aan van een geescapte string. ">" en "|" zijn het begin van respectievelijk een *folded block* en een *literal block*. (Een folded block is een blok waar regeleindes door spaties vervangen worden.) Andere scalars kunnen eenvoudig met reguliere expressies herkend worden.

Bij het parsen van scalars wordt intensief gebruik gemaakt van de voorziene buffer uit de *ParseStream*. Een uitzondering hierop is indien er gebruik wordt gemaakt van tags zoals *!!str*, *!!float*, ..., dan was al eerder bekend om welk type het gaat. We proberen telkens een bepaald type, indien deze node niet geldig is volgens dat type, wordt er een error geworpen die maakt dat we het volgende type proberen. Als laatste wordt het type string geprobeerd omdat bijna alles als geldige string kan worden geinterpreteerd. Er is ook ondersteuning voor folded en literal blocks, maar deze zijn nog niet volledig; onderandere + en − chomps zijn niet ondersteund.

Volgende scalars worden ondersteund:

1. Strings
2. Booleans
3. Null values
4. Integers
5. Floats
6. Timestamps
7. Binary data

Scalars worden niet gemapped op native C# datatypes zoals we zouden doen bij het echt (de)serialiseren via reflectie. Wij gebruiken de native types wel, maar deze steken we telkens in een eigen container afgeleid van een node. Hierdoor is het eenvoudig deze in boomstucturen als mappings en sequences te steken. Timestamps zijn een geval apart; Yaml biedt de mogelijkheid om een tijdzone te definiëren, maar dit is niet aanwezig in de 1.0 versie van .NET. Daarom komen we niet toe met het native datatype alleen en hebben we een extra variabele om de tijdzone te bewaren opgeslagen.

Bij het parsen van de scalars hebben we het gebruik van reguliere expressies vermeden. Aangezien we een one-pass parser hebben trachten te implementeren was het logisch om geen reguliere expressies te gebruiken. Ook zouden reguliere expressies de efficiëntie niet ten goede komen in ons geval.

4.3.7 Ordered maps, pairs en sets

> Dit zijn datastructuren die van sequences en mappings afgeleid zijn. Ordered maps en pairs worden voorgesteld door een sequence van maps, het verschil is dat bij pairs wel duplicaten worden toegelaten en bij ordered maps niet. Sets worden voorgesteld als mappings waarbij de value telkens gelijk is aan nul.

Zoals reeds eerder aangehaald bij het bespreken van het klasse-diagram (sectie 4.1) hebben we deze functionaliteit niet geïmplementeerd. Het aanbieden van deze structuren aan de parser geeft geen problemen aangezien we deze mappen op de reeds geïmplementeerde types.

Momenteel worden sets nog als normale mappings gezien bestaande uit keys zonder value, maar deze constraint wordt nog niet gecontroleerd. Op dezelde manier worden pairs geparsed als sequences zonder de pair-specifieke constraints te controleren.

4.3.8 Integratie van het .NET Framework

> Na het parsen verkrijgen we een boom die opgebouwd is uit onze eigen datatypes, dit in tegenstelling tot sommige implementaties (Ruby) waar dat de 'native' datatypes van de taal gebruikt worden. Het voordeel van de tweede methode is dat dit het gebruik van de library volledig transparant maakt. De gebruiker hoeft niet meer te weten dan de syntax van de parse en write functies en kan verder werken met de standaard datatypes zoals arrays en hashes welke in de taal geimplementeerd zijn. We onderzoeken nog in hoeverre deze manier van werken mogelijk zal zijn voor .NET.

We zijn bij ons eerste idee gebleven om eigen wrapper classes te implementeren voor de verschillende datatypes.

De class Node heeft een statische *Parse* functie die een stream of string meekrijgt en één van zijn kinderen teruggeeft afhankelijk van het type node. De *Write* functie van node maakt van een node terug een Yaml string en de *ToString* functie geeft de inhoud in string vorm terug. Deze laatste dient enkel om eenvoudig te controleren of Yaml code daadwerkelijk juist wordt geparsed.

> Het zal niet vanzelfsprekend zijn (misschien onmogelijk) om de inhoud van een willekeurige class uit te schrijven naar een Yaml stream. Ruby, python, perl en PHP zijn scriptingtalen, welke flexibele mogelijkheden ondersteunen voor het het omgaan met datatypes.

Bij het bespreken van het analyse verslag werd het gebruik van reflection besproken. Dit zou een gedeeltelijke oplossing kunnen zijn. Wegens gebrek aan tijd en ervaring op dat gebied zien we hier niet aan toe kunnen komen.

5 Het programmeren

Aanvankelijk was C# voor Christophe een nieuwe programmeertaal. Het was een motivatie om dit project te doen, omdat we op deze manier iets nieuw konden leren.

Als referentie hebben we [5], [1] en [8] gebruikt. In mindere maten hebben we ook de website van het Mono Project geraadpleegd [4]. Ook de hulp van Gert van Gool en Ingo Berben stellen we erg op prijs. Ook zij hebben hun trimester overschreidend project gemaakt in C# wat zorgde voor een interessante wisselwerking.

5.1 Probleempjes

Tijdens het programmeren hebben we geen spectaculaire problemen gehad. Enkele puntjes die we hier misschien willen aanhalen zijn het boxing en unboxing probleem en het omgaan met binaire data.

Op een gegeven moment wenste we een *int* als parameter mee te geven als een reference. Dit bleek echter niet te lukken We hebben hier boxing en unboxing voor trachten te gebruiken, wat niet het gewenste effect gaf. Het gebruik van pointers wordt in C# wordt afgeraden, maar we vonden de *out* constuctie waarmee we functie parameters konden veranderen.

Ook hebben we ons verdiept in een nieuw aspect, binaire data. Zo wordt binaire data in Yaml gerepresenteerd in Base64. Afhankelijk wilde we dit volledig zelf gaan parsen. Echter met het gebruik van de juiste library *System.Convert* is het relatief eenvoudig. Deze voorziet ook in de functionaliteit om een byte array terug om te zetten naar Base64.

Aangezien ons project intensief steunt op het appenden van string hebben we gebruik gemaakt van *System.Text.StringBuilder*. Dit is efficienter dan rechtstreeks op een string te werken aangezien er bij elke wijziging van een gewone string steeds een nieuwe instantie wordt gecreëerd.

Vaak hebben we problemen gahad dat nieuwe code interfereerde met al eerder geschreven code. Door alles goed op splitsen zijn de meeste problemen verholpen.

We hebben nog enkele Yaml specifieke problemen. De integer *-1* wordt als een sequence gezien omdat we de spatie na het liggend streepje nog niet verplichten. *!!int -1* gebruiken is voorlopig een oplossing. Een ander gelijkaardig probleem hebben we bij timestamps: *2006-06-01 8:45* hier wordt de dubbele punt als mapping gezien, tenzij de timestamp door een *!!timestamp* tag voorafgegaan wordt.

5.2 Library

Het is logisch dat ons project als library werd gepubliceerd. In plaats van een normale executable file voorzien wij in een DLL file. In het vak *Gea-*

vanceerde programmeer technieken zijn de technieken voor het maken van libraries uitgelegd voor C en C++. In het .NET platform zijn enkele zaken vereenvoudigd en is er rekening gehouden met de problemen uit het verleden. Zo wordt de developper gemotiveerd om zijn code te *signen*. Het heeft ons enig zoekwerk gekost, maar uiteindelijk hebben we deze redelijk eenvoudige klus toch geklaard.

6 Open Source

Zoals beloofd publiceren we ons project als Open Source. We waren hier niet vertrouwt mee maar gelukkig zijn we hier goed in geholpen. We hebben nauwgezet de richtlijnen van de GPL [2] website gevolgd [3]. We hebben verder ook gekozen om de Lesser GPL licentie te gebruiken. Dit geeft de mogelijkheid om de library te gebruiken in commerciële pakketten, op voorwaarde dat de library vrij beschikbaar blijft.

7 Taakverdeling en planning

> Dit trimester hopen we nog voldoende research te kunnen doen. Verder trachten we ook in communicatie te gaan met de hoofdleden en community van Yaml. Enerzijds om hen te informeren rond ons project en anderzijds eventuele ondersteuning te verkrijgen.

Ons verdiepen in Yaml (en C#) heeft zoals gepland het volledige tweede trimester ingenomen. We hebben enige test code geschreven en heel beperkt de Ruby parser bestudeerd. Achteraf bekeken was het niet nodig de Yaml specificatie zo diepgaand te bestuderen aangezien we slecht een beperkt deel hebben geimplementeerd. Het had misschien verstandiger geweest om sneller te beginnen programeren en indien nodig later de Yaml documentatie terug ter hand te nemen. De vraag die we ons kunnen stellen is echter of we dan geen cruciale fouten hadden gemaakt in onze structuur?

We hebben in de laatste week van het project onze parser aangekondigd op de Yaml mailing list. We dachten hier heel wat reacties op te ontvangen, dit is spijtig genoeg niet het geval. Onze algemene indruk is dat Yaml communitie aan het stilvallen is. In het begin van het project waren er hevige discussies, nu wordt er slechts sporadisch gepost.

> Onze Yaml parser zal zoals eerder gezegd een one-pass parser worden, we kunnen dus geen taakverdeling maken op basis van de passes. Een goede verdeling is dat één iemand de individuele scalars gaat parsen en iemand anders de grotere structuren zoals sequences en mappings. Voor we dit kunnen beslissen moeten we ons beide verdiepen in de details van Yaml,

[3] Het onderwijsteam was bij het ter perse gaan van dit verslag nog bezig met het verzorgen van de nodige gegevens betreffende de 'supporting school'. Uiteraard wordt dit aangepast zodra dit bekend is.

> dit is iets dat we niet kunnen opdelen. Tenslotte zal afhankelijk van wie het verst gevorderd is bepaald worden wie de Yaml generator gaat schrijven. Als tegen gewicht op de Yaml generator zijn er nog talloze andere hulp functies nodig, die dan de andere persoon voor zijn rekening kan nemen.

In het analyse verslag hadden we slecht een beperkt idee van de taakverdeling. We hebben wel op aanraden van het onderwijsteam getracht een duidelijkere taakverdeling te maken. Van de moment dat we zijn beginnen programmeren hebben we de Yaml structuur in twee delen opgedeeld. Enerzijds de Collection types (mappings, sequence, ...) en anderzijds de Scalar types. Aangezien de Scalar types aanvankelijk eenvoudig leken heeft Christophe dit voor zijn rekening genomen. Volgens de oorspronkelijke planning zou dit niet de volledige ontwikkelingstijd innemen en zou er zo een reserve zijn om aan andere delen of extra's te werken. Dit is echter niet gelukt en zo heeft Jonathan zowel de Collections types als de onderliggende stream structuur uitgewerkt.

Een overzicht van onze logs is te vinden in bijlage A.2, blz. 16.

8 Slot opmerking

> Het is duidelijk dat dit geen standaard project zal worden. Dit is ook onze belangrijkste motivatie om te kiezen voor iets wat we nog niet kennen en hopelijk veel van kunnen bijleren. We verwachten ons aan moeilijke momenten en gaan ervan uit dat niet alles even vlot zal verlopen als we willen. We hopen dat we kunnen rekenen op de nodige steun van uit het onderwijs team.

8.1 Ons standpunt t.o.v. Yaml

Yaml is goed leesbaar en verstaanbaar. Verder is het ook eenvoudig om te schrijven, maar naar onze mening heeft de gebruiker te veel vrijheid. Zo kunnen dezelfde dingen op verschillende manieren worden geschreven. Dit resulteert in meer verwarring dan dat het helpt. We zijn het dan ook volledig eens met de communitie dat er een vereenvoudiging van Yaml nodig is.

Yaml is een mooi alternatief voor XML. Het grote voordeel van Yaml is de ondersteuning voor ankers en referenties. Dit maakt, buiten boomstructuren, ook circulaire en accyclische datastructuren mogelijk.

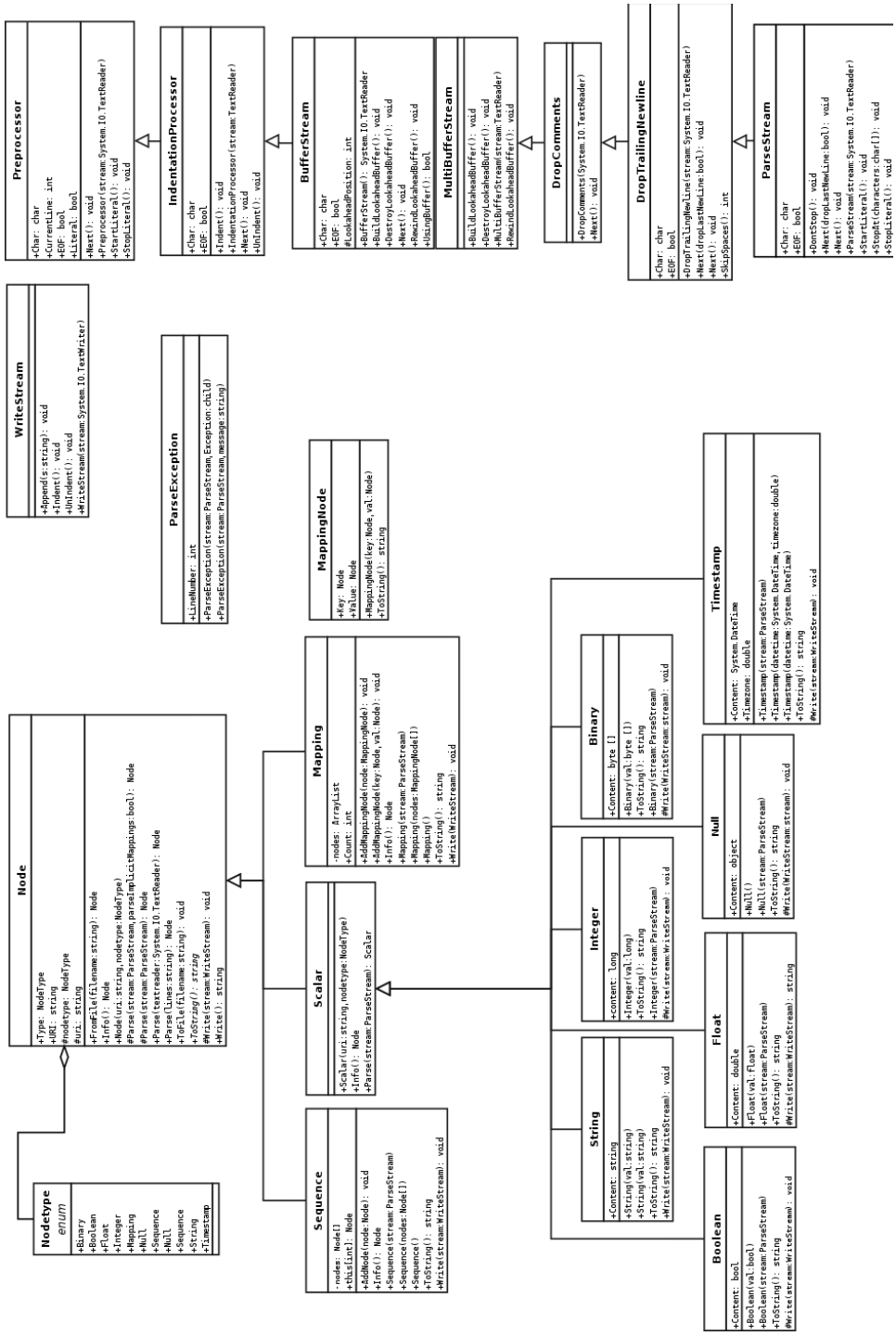
Yaml moet vooral nog heel hard groeien! (Of inkrimpen?)

8.2 Het project in het algemeen

Zoals verwacht hebben we veel bijgeleerd. We hebben tot onze verbazing nooit de hulp van het onderwijs team moeten inroepen. Yaml en C# zijn dan ook twee heel toegankelijke items die met een beetje doorzettings vermogen goed te verwerken zijn.

A Bijlage

A.1 UML Diagram



A.2 Logs in Yaml

A.2.1 Jonathan Slenders

```
name: Jonathan Slenders
studentennummer: 0421645
log:
  - "24-01-06":
      - duur: dag
      - omschrijving:
          - CVS opzetten
          - C# initiatie
          - lezen in YAML Specification
          - Nadenken over het parsen van yaml code
  - "28-01-06":
      - duur: avond
      - omschrijving:
      - Stukje verslag
  - "4-02-06":
      - duur: half uurtje
      - omschrijving:
        - UML diagrom gedeeltelijk geschreven in Yaml
  - "29-02-06":
      - duur: onbekend
      - omschrijving:
         - Omdat ik zoveel keer hier ben vergeten te loggen,
           hier een entry als placeholder
           voor de vergeten items.
  - "29-02-06":
      - duur: een uur
      - omschrijving:
          - Begonnen aan her herschrijven van het
            begin van de parser.
            Reduceren naar 1 a 2 passes.
  - "18-04-06":
      - duur: bijna hele dag
      - omschrijving:
          - hele layout van parsestream en nog enkele
            andere classes herschrijven,
            eerste versie van de buffer
  - "23-04-06":
      - duur: enkele uren
      - omschrijving:
          - Parsestream aan het herschrijven om
```

```
            efficienter met de buffer om te
            gaan, bug fixes
            (nu zou er niet onnodig readahead gelezen worden)
  - "26-04-06":
      - duur: 2 uur
      - omschrijving:
          - Bugfixes in het buffer mechanisme
            (extra spatie teveel)
```

A.2.2 Christophe Lambrechts

```
---
name: Christophe Lambrechts
studentennummer: 0421611
log:
  - "Begin":
      - omschrijving:
          - Sporadisch lezen in YAML Specification
          - Inschrijven op YAML mailing list
  - "24-01-06":
      - duur: dag
      - omschrijving:
          - CVS opzetten
          - C# initiatie
          - lezen in YAML Specification
          - basis voor het analyse verslag leggen

  - "27-01-06"
      - duur: avond
      - omschrijving:
           - Lezen in C# voor professionals
  - "29-01-06"
      - duur: "40 min."
      - omschrijving:
          - Schrijven aan analyse verslag
  - "01-02-06"
      - duur: "5 min."
      - omschrijving:
          - Beginnetje voor UML diagram
  - "04-02-06"
      - duur: "60 min."
      - omschrijving:
           - Lezen in C# voor professionals
           - Werken aan verslag
```

```
- "06-02-06"
    - duur: avond
    - omschrijving:
            - Verder werken aan verslag
            - Voorlopig UML diagram uit Yaml omzetten in Dia
- "12-02-06"
    - duur: "1.5 uur"
    - omschrijving:
            - Verslag afwerken
- "01-04-06"
    - duur: "2 uur"
    - omschrijving:
            - Lezen Yaml Cookbook
- "04-04-06"
    - duur: "4 uur"
    - omschrijving:
            - Samen de verdere krijtlijnen
              uitzetten voor het project
- "11-04-06"
    - duur: "dag"
    - omschrijving:
            - Integer.cs afwerken en testen
            - Yaml reference verder bestuderen
- "18-04-06"
    - duur: "dag"
    - omschrijving:
            - ParseStream ontwerpen
            - Huidige Scalar afgeleide classe omzetten
              naar nieuwe ParseStream
- "25-04-06"
    - duur: "dag"
    - omschrijving:
            - Combibiliteits problemen oplossen
            - NDoc configureren en opzetten
            - Binary data (bestuderen en
              documentatie zoeken)
- "01-05-06"
    - duur: "1/2 dag"
    - omschrijving:
            - Binary.cs implementeren
- "02-05-06"
    - duur: "dag"
    - omschrijving:
            - Binary.cs proberen aftewerken
```

```
            - Timestamp.cs implementeren
  - "07-05-06"
      - duur: "dag"
      - omschrijving:
              - Place holder voor alle kleine
                werkjes die nog moesten gebeuren
              -
                 - Verslag
                 - Debuggen
                 - ...
```

A.3 Onderinge communicatie in Yaml

```
chat:
  - 2006-02-02:
      naam: Christophe
      message: >
        Hey Jonathan, hier heb ik een TXT waar we op
        lange termijn wat ideen kunnen uitwisselen.
        Ik dacht er aan om misschien onze logs en deze file
        misschien ook, direct in YAML formaat te zetten?
        Dan hebben we direct een leuk test document voor op
        de presentatie. Alleen moeten we dan eens denken over
        de structuur hé...
  - 2006-02-02:
       naam: Jonathan
       message: >
         Cool! Alles staat ondertussen in Yaml Formaat.
         Als je het bestand de .yaml
         extensie geeft geeft vim er zelfs een mooie
         syntax highlighting aan.
  - 2006-02-04:
      naam: Christophe
      message:
        - Ik heb wat aanpassingen gedaan aan het verslag.
          Kijk jij daar eens naar en pas
          maar aan als je iets weet.
          Misschien moeten we die dingen maandag wat
          verder uitspitten nog als we er niet uitkomen.
        - Leuk van vim, misschien moeten we onze logs
          ook maar een andere extensie geven.
        - Jonathan, jij vergeet ook niet je log aan te passen?
        - Mag deze opmaak van deze message??
  - 2006-02-05
```

```
    naam: Jonathan
    message: >
      De opmaak van je vorige message is verkeerd.
      het > teken geeft
      aan dat hetgeen volgt een folded textblock is.
      (newlines worden vervangen door spaties,...)
      Een sequence binnen dit
      block gaat niet, de streepjes zouden als gewone tekst
      gezien worden. Beste is daar gewoon de > weg
      te laten.
- 2006-02-08
    naam: Christophe
    message: |
      Inderdaad, ik had een foutje gemaakt.
      Bij deze is het aangepast.
      Ik heb nog eens gekeken naar het UML diagram,
      ik kan er niet zoveel aan doen denk ik.
      We moeten die zaken misschien nog eens
      bespreken
      (zit met nog wat vragen omtrent types enzo.)
      Verder staat er ergens in het details
      voor het synthese verslag
      dat we algoritmes moeten voorstellen
      met behulp van speudocode.
      Wat de boomstructuur betreft,
      hoe gaan we de verschillende kinderen
      voorstellen? Arrays/Vectors,
      kinderen naar rechter buren laten
      verwijzen, ... Wat ik zeker zou doen,
      dat is de parent bijhouden.
- 2006-02-09
    naam: Christophe
    message: >
      Ik heb jouw boeken hier nog liggen.
      Als jij die nodig hebt
      moet je het maar laten weten.
      Ik denk niet dat ik in
      het kort nog ga kunnen lezen.
      Maar ik wil er zeker nog
      wel dingen uit halen als ik begin te coden.
- 2006-02-10
    naam: Christophe
    message: >
      Bij Node in het UML diagram heb je
```

```
      nog steeds de parent
      staan, moet dit niet een child worden?.
      Kan jij ook eens bij gelegenheid die ruby test
      filetjes op CVS zetten.
- 2006-02-11
    naam: Jonathan
    message: >
      Ik vergeet dus steeds die log in te vullen.
- 2006-02-11
    naam: Christophe
    message:
       - Je bent het nu weer vergeten denk ik ...
         (je log invullen)
       - Misschien moet je jouw log ook hernoemen
         naar .yaml (is iets consistenter)
- 2006-02-18
    naam: Christophe
    message:
       - Jonathan, denk jij er nog aan om de xml commentaar
         in jouw classes aan te vullen?
- 2006-06-01
    naam: Jonathan
    message:
       - Christophe, enkele opmerkingen ivm.
         je programmeerstijl die
         misschien kunnen helpen
       -
         - Plaats een spatie aan de
           buitenkant van je haakjes
         - Niet te veel onnodig commentaar,
           en als je commentaar
           plaats, becommentarieer dan
           het werken van de code,
           bespreek nooit hoe een
           taal-specifieke constructie
           werkt, vermits een lezer van
           de code hier normaal
           al bekend mee is.
         - Enkel <summary> voor publieke
           elementen lijkt me genoeg.
         - Begin commentaar met een hoofdletter.
         - Gebruik slash-slash commentaar, nooit slash-ster
           commentaar. Bij het eerste kan
           je eenvoudig grote
```

```
              stukken code uitstuiten.
            - Als je een woord niet weet kan
              je het snel opzoeken,
              bijvoorbeeld via interglot.com.
              Een 'Colon' is een
              dubbele punt, geen kolom, wat
              'Column' is in het
              Engels. Excepted bestaat niet, het is expected.
            - Het heeft geen zin een exception op te
              vangen met
              een catch als je hem onmiddellijk verdoorgooit
              zonder iets met de error-info te doen.
              Het is niet verkeerd deze tot enkele
              niveaus of zelfs tot het
              'root'-level verder te gooien,
              door het opzettelijk
              laten ontbreken van een catch.
            - Nest niet meer dan nodig: '
if (! eof)
{
...
if (! eof)
{
...
if (! eof)
{
...
}
}
}
kan best  vervangen worden door:

if (! eof)
{
...
}
if (! eof)
{
...
}
if (! eof)
{
...
}
```

```
            - Double.Parse ("0,087860"): lijkt me
              regio afhankelijk
              Een decimaal kan evengoed een punt zijn,
              dat hangt van de taalinstellingen af.
              Ik heb het in DateTime vervangen
              door een simpele loop.
            - Probeer je totale regellengte te beperken
              tot 80 karaketers
              en het gebruikt gedeelte, dus zonder
              (plaatselijke marges)
              tot 50 karakters. Dit geeft een veel beter
              overzicht, lange
              regels zijn vermoeiend om te lezen.
- 2006-06-06
     naam: Christophe
     message:
        - Heel hartelijk bedankt voor de opmerkingen.
          Ik heb best veel geleerd van jouw, dank je wel.
        - PS: Je hebt hierboven gebruik gemaakt van tabs,
          ik heb ze alvast verwijderd.
        - Op naar de finish ...
```

A.4 Manual

Voor een actuele versie van de manual verwijzen we naar de project website op `http://lumumba.uhasselt.be/~christophe/YAML/`. We voegen hier een versie toe van 8 juni 2006.

Referenties

[1] Tom Archer. *Inside C#*. Microsoft Press A Division of Microsoft Corporation, Redmond, Washington 98052-6399, 2001.

[2] How to use the GPL. Website. `http://www.gnu.org/licenses/gpl-howto.html`.

[3] Jyaml - Yaml library for the Java language. Website. `http://jyaml.sourceforge.net/`.

[4] Mono CSharp Compiler. Website. `http://www.mono-project.com/CSharp_Compiler`.

[5] MSDN - .NET reference. Website. `http://msdn2.microsoft.com/`.

[6] NDoc - XML Commenting system. Website. `http://ndoc.sourceforge.net/`.

[7] Official Yaml mailing list archive. Website. `http://sourceforge.net/mailarchive/forum.php?forum_id=1771`.

[8] Simon Robinson, Ollie Cornes, Jay Glynn, Burton Harvey, Craig McQueen, Jerod Moemeka, Christian Nagel, Morgan Skinner, and Karli Watson. *C# voor professionals*. Wrox Programmer to Programmer. Academic Service, Schoonhoven, 2002. Nederlands vertaling door Fontline, Nijmegen.

[9] Syck - Yaml for Ruby, Python, PHP and OCaml. Website. `http://whytheluckystiff.net/syck/`.

[10] Yaml Homepage. Website. `http://www.yaml.org/`.

[11] Yaml Specification. Website. `http://www.yaml.org/spec/`.

[3]Bij het ter perse gaan van dit eind verslag zijn alle URL's voor een laatste maal bezocht en geverifieerd.