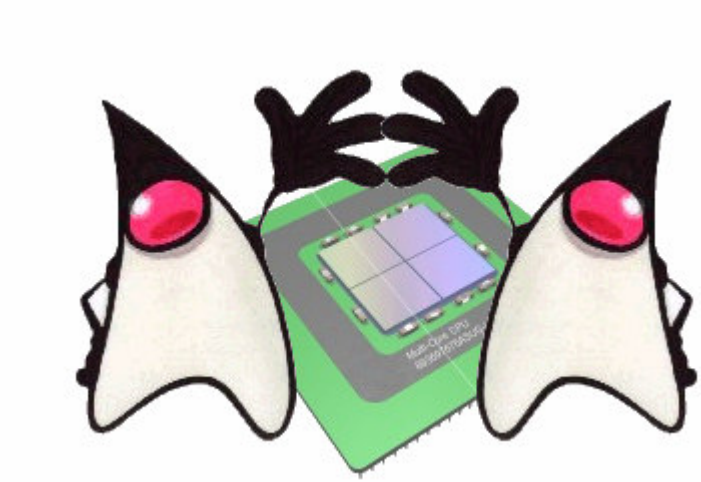
Masterscriptie Software Engineering

“Programming for a parallel future”



Arjen van Schie

Masterscriptie Software Engineering

"Programming for a parallel future"

"Improving the modularity and encapsulation for the implementation of concurrency concerns."

Arjen van Schie

28-8-2007

Publicatiestatus: Openbaar

UNIVERSITEIT VAN AMSTERDAM

LogicaCMG

Onderwijsinstelling : Universiteit van Amsterdam Opleiding : Eénjarige Master Software Engineering	Opdrachtgever: LogicaCMG Rotterdam Divisie: Industry, Distribution and Transport Competentie: Java Programma: Working Tomorrow
Afstudeercoördinator: Jurgen Vinju	Programmabegeleider: Wouter Kers Programma-architect: Martin van Amersfoorth

Samenvatting

Met de opkomst van multi-core processoren is er een toenemende aandacht ontstaan voor het ontwikkelen van software met concurrency. Concurrency is een techniek waarmee in de software meerdere taken gelijktijdig, parallel, uitgevoerd kunnen worden. Deze toenemende aandacht ontstaat omdat het toepassen van concurrency essentieel is om multi-core processoren goed te benutten. Echter het implementeren van concurrency in de huidige programmeertalen zoals Java en C# is complex. Dit onderzoek tracht het implementeren te vereenvoudigen.

Dijkstra gaf al in 1974 aan dat het scheiden van verantwoordelijkheden, *Separation of Concerns*, één van de belangrijke pijlers is voor het beheersen van de complexiteit. Voor dit scheiden van verantwoordelijkheden (concerns) is het van belang een goede *modularity* (opdeling in modulen met één specifiek concern) en goede *encapsulation* (verbergen van implementatiedetails) te waarborgen.

Binnen dit onderzoek wordt in kaart gebracht hoe goed het mogelijk is om met objectoriëntatie een scheiding van de concurrency concerns te realiseren. Wanneer concerns slecht te scheiden zijn met objectoriëntatie noemen we deze *crosscutting concerns*. Binnen dit onderzoek is een aantal concurrency concerns vastgesteld voor een applicatie waarin concurrency een grote rol speelt. Vervolgens is van deze concerns onderzocht of er sprake is van *crosscutting*. Daarnaast zijn deze *crosscutting concerns* geclassificeerd volgens een model met bekende typen *crosscutting concerns*. Dit maakt het mogelijk deze concerns binnen andere applicaties eenvoudiger te herkennen en geeft indicaties over mogelijke oplossingen, vergelijkbaar met de voordelen van *design patterns*.

Nadat van enkele concurrency concerns vastgesteld is dat er inderdaad sprake was van *crosscutting,* is met behulp van een alternatief voor objectoriëntatie(OO), namelijk aspectoriëntatie(AO), een alternatieve implementatie van een deel van de applicatie gerealiseerd. Aspectoriëntatie is een techniek waarmee *crosscutting concerns* op een natuurlijker wijze geïmplementeerd kunnen worden. Hierdoor is het dus mogelijk om betere *modularity* en *encapsulation* te realiseren.

Om te kunnen bepalen of de concerns met AspectJ daadwerkelijk beter geïmplementeerd zijn, is besloten bekende software metrieken over *modularity* en *encapsulation* te gebruiken. Omdat we te maken hebben met meerdere programmeerparadigma's (objectoriëntatie en aspectoriëntatie) is besloten metrieken uit beide werelden te selecteren en deze op beide oplossingen los te laten. Daarnaast is nog op basis van onderhoudsscenario's, van concurrency gerelateerde wijzigingen, bepaald hoe goed de principes door ieder van de oplossing ondersteund worden.

Op basis van de gegevens uit de software metrieken kan vastgesteld worden dat er inderdaad een betere *modularity* en *encapsulation* is te vinden voor de concurrency concerns bij de aspectgeoriënteerde oplossing. Zowel bij de AO-metrieken en OO-metrieken als bij de scenario's blijkt dat de aspectgeoriënteerde oplossing beter scoort.

Tot slot is nog gevalideerd hoe representatief de gevonden concurrency concerns uit de onderzochte applicatie zijn. Hierbij is vastgesteld dat de gevonden concurrency concerns inderdaad bij andere applicaties ook een rol speelden en dat de implementatie daar ook *crosscutting* is.

Op basis van dit onderzoek kan geconcludeerd worden dat er concurrency concerns bestaan waarvan is vastgesteld dat ze in meerdere applicaties *crosscutting* zijn. En dat de *modularity* en *encapsulation* hiervan verbeterd kan worden met behulp van aspectoriëntatie. Kortom objectoriëntatie is niet optimaal wanneer het gaat om *modularity* en *encapsulation* van concurrency concerns

Summary

As of the last two years, the consumer market of CPU's is dominated by a new type of CPU: the multi-core CPU. Multi-core CPU's differ from the regular ones in their ability to execute multiple statements parallel to each other, whereas regular CPU's only allow one statement to be executed at a time. To use these new CPU's to their fullest potential, it will be mandatory to include concurrency in the software. Concurrency is a general term for the ability of a program to indicate that multiple threads of execution might be executed at the same time. But implementing concurrency in, for example, Java or C# programs, is considered to be very complex. The goal of this project is to lower this complexity.

One of the tools used to lower the complexity of source code is called separation of concerns (SoC), an idea introduced by Dijkstra. With this separation, one can focus on the individual parts, thus resulting in lower complexity at a time. Two principles for maintaining this SoC are: modularity (each concern in a separate module) and encapsulation (each module should hide its implementation). This thesis investigates whether or not object orientation is suited for realizing SoC for concurrency related concerns, as a way of lowering the complexity of implementing them.

For this we investigated the existence of so called crosscutting concurrency concerns (Aspect mining). Crosscutting concurrency concerns are concurrency related concerns with bad SoC. This resulted in a list of concurrency concerns with a crosscutting implementation. Next we classified them with a model of well known crosscutting concern sorts. This model supplies us with known ways of improvements for the sorts of the model.

After the identification and classification of the crosscutting concurrency concerns, we went on to investigate a way to improve the situation. For this we used AspectJ, an aspect-oriented programming language. Aspect orientation is a methodology similar to object orientation, but in this case the decomposition is not based on objects but on aspects. Aspects are the natural modular implementation form of crosscutting concerns. With AspectJ we developed new implementations of the concerns and refactored the application from which the concerns where derived.

Building these new implementations was not enough though; we also had to evaluate whether or not the implementations really improved the SoC for concurrency. The problem with this is that the solutions where built on different paradigms (aspect orientation and object orientation). This makes it difficult to compare them to each other.
To accomplish this we designed a method based on known aspect oriented and object oriented software metrics, completed with maintenance scenarios. This method will allow us to measure and compare the modularity and encapsulation of the solutions.

The results of these measurements indicated that the use of AspectJ indeed increased the SoC. The AO metrics, the OO metrics and the scenarios all indicated that the Aspect solution performed better with respect to modularity and encapsulation.

One of the pitfalls of these promising results was the commonness of the chosen concurrency concerns. Therefore it was validated whether or not these concerns where unique or common, by investigating the existence of them among other applications. As it turned out, all the initially investigated concurrency concerns had a role in one or more of the other applications. We found that their implementation was also crosscutting in a similar way.

Based on this thesis it can be concluded that there are crosscutting concurrency concerns shared among multiple applications and that the modularity and encapsulation of those can be raised by the use of aspect orientation, AspectJ. So object orientation is less then optimal for the implementation of concurrency concerns with respect to SoC.

Voorwoord

Deze scriptie beschrijft het afstudeerproject van Arjen van Schie. Het project is onderdeel van de laatste fase van de opleiding Master Software Engineering aan de Universiteit van Amsterdam en is uitgevoerd als onderdeel van het Working Tomorrow Programma, een Landelijk afstudeerprogramma binnen LogicaCMG. De opdracht is bij de Java Competence te Rotterdam uitgevoerd.

Het doel van dit document is het aantonen van de capaciteiten van de student op het gebied van gestructureerd wetenschappelijk onderzoek. Daarnaast is een subdoel het wekken van interesse en aandacht binnen LogicaCMG voor de onderwerpen die binnen deze scriptie behandeld worden.

Dankwoord

Allereerst wil ik een aantal personen bedanken voor het beschikbaar stellen van een afstudeerplek bij LogicaCMG, te weten: Eddy de Ridder (Projectleider WT Landelijk), Wouter Kers (Projectleider WT Rotterdam) en Rob Krassenburg (Competence Manager). Zij stelden mij in staat om dit onderzoek uit te voeren binnen een gezellige en innovatieve werksfeer. Bij deze wil ik ook mijn collega-studenten binnen het WT-programma voor deze sfeer bedanken.

Daarnaast gaat mijn dank uit naar de volgende personen voor hun medewerking en inspiratie: Marius Marin (TUDelft), Michael Suess (University of Kassel), Peter Hofstee (IBM), Marco Pas (LogicaCMG), Okke van 't Verlaat (LogicaCMG), Marco Hayes (LogicaCMG), Martin van Amersfoorth (LogicaCMG), Klaas van der Ploeg (LogicaCMG), Marius van Hal (LogicaCMG), Christiaan Tilman (LogicaCMG), Paul Gardner (Azureus), Michal Stochmialek (AOPMetrics Team).

Tot slot wil ik natuurlijk nog de docenten van de Universiteit van Amsterdam bedanken die mij geholpen hebben gedurende het jaar en in het bijzonder mijn afstudeerbegeleider Jurgen Vinju.

Veel plezier met het lezen van mijn scriptie.

Arjen van Schie,
Augustus 2007,
Honselersdijk.

Inhoudsopgave

1 Inleiding

Op dit moment lijkt er een einde te komen aan de stijgende lijn in de klokfrequenties van processoren. Om de klokfrequentie van de huidige generatie processoren van Intel en AMD nog verder te verhogen is zodanig veel koeling vereist om de warmteproductie op te vangen, dat er een andere richting in wordt gegaan door deze fabrikanten: Multi-core.
De fabrikanten hebben daarom allen de overstap gemaakt naar multi-core CPU's, CPU's waarbij meerdere CPU-cores op één chip geplaatst worden. De rekenkracht is hierbij dus afhankelijk van de cumulatieve rekenkracht van de cores op een chip. Eén probleem is, om deze rekenkracht te benutten dient de software aangepast te worden voor multi-core CPU's en dit (*parallel programming*) is lastig. Deze situatie wordt dan ook wel de *multi-core crisis* genoemd *[O'Reilly,2007][Bader, 2007].*

Eén van de zaken die het ontwikkelen van software voor multi-core machines zou vereenvoudigen, is het verhogen van de abstractie niveaus waarop de ontwikkelaar bezig is met het probleem. Waarbij in dit geval het probleem bestaat uit het toevoegen van parallellisme, de mogelijkheid om meerdere zaken tegelijk uit te voeren.
Binnen de literatuur zien we vaak dat het scheiden van verantwoordelijkheden in aparte modulen als oplossing wordt aangewezen voor het verminderen van de complexiteit *[McConnel,2004].* Dit doordat de ontwikkelaar zich dan beter kan focussen op de afzonderlijke zaken. Dit scheiden wordt ook wel *Separation of Concerns* genoemd *[Dijkstra,1974]*.
Een systeem beschikt over een goede *Separation of Concerns* wanneer het systeem is opgedeeld in modulen met elk één eigen taak (*modularity*) en wanneer het gebruik van deze module onafhankelijk is van de implementatie ervan, dat de interne werking verborgen is (*encapsulation*).

Deze opdracht richt zich op de toepasbaarheid van de principes van *encapsulation* en *modularity* voor parallellisme. De vraag is hoe goede deze principes kunnen worden toegepast hiervoor.

Een meer uitgebreide beschrijving van de opdracht, de motivatie en de context is terug te vinden in *hoofdstuk 2 Probleemanalyse.*

Belang opdrachtgever bij dit onderzoek

De opdrachtgever waarvoor dit onderzoek wordt uitgevoerd heeft een grote ervaring in het ontwikkelen van omvangrijke (gedistribueerde) systemen. Echter de ervaring met het bewust ontwikkelen van multi-threaded applicaties is nog vrij klein. Maar de opdrachtgever ziet wel in dat de overstap naar multi-core CPU's voor hem een rol zal gaan spelen bij het ontwikkelen van deze applicaties.
Dit vormt hun motivatie voor dit onderzoek, zie ook *Bijlage E: Context van de opdracht*.
De opdrachtgever is daarnaast ook geïnteresseerd in de mogelijkheden van AspectJ. Ze hebben inmiddels positieve ervaringen met het inzetten hiervan bij verschillende projecten. Daarom zal onderzocht worden of AspectJ ook voordelen heeft bij het ontwikkelen en onderhouden van multi-threaded applicaties, door bijvoorbeeld het abstractie niveau van bepaalde constructies te verhogen.

Geleverde bijdragen

Hieronder is een opsomming te vinden van de bijdragen die geleverd zijn binnen deze scriptie.

- Een overzicht van (crosscutting) concurrency concerns in multi-threaded software
- Een *toolkit* voor multi-threading in AspectJ
- Een methode voor het vergelijken van AOP en OOP systemen op *modularity* en *encapsulation*
- Een gestructureerde aanpak voor het verbeteren van de implementatie van concurrency concerns in multi-threaded software

In de conclusie (*H8*) zullen deze verder worden toegelicht.

Daarnaast is er nog een bijdrage die gericht is op de opdrachtgever, deze is terug te vinden in de *Bijlage E: .* Dit is een analyse van de stand van zaken bij de opdrachtgever op het gebied van concurrency en parallellisme.

Doelgroep

Deze scriptie is geschreven met als doelgroep: lezers met een gedegen kennis van concurrency, parallel programming en object / aspect oriëntatie. Voor lezers die (een deel van) deze kennis missen is in de bijlage *Bijlage D: Appendix introductie parallel programming* meer informatie over parallellisme terug te vinden. Dit maakt de scriptie toegankelijk voor een brede doelgroep.

Bovendien is in *Bijlage B: Begrippenlijst* een lijst te vinden met de belangrijkste begrippen uit deze scriptie met hun betekenis.

Leeswijzer

De structuur van de scriptie is als volgt:

Achtergrond:

H1: Deze inleiding
H2: Dit hoofdstuk beschrijft de motivatie en achtergrond van het onderzoek
H3: Dit hoofdstuk omvat een beschrijving van de gehanteerde onderzoeksmethode.
H4: Hierin komt de benodigde achtergrondinformatie over aspectoriëntatie en *parallel programming* naar voren. Dit hoofdstuk is tevens een antwoord op de eerste en tweede onderzoeksvraag.

Onderzoek:

H5: Binnen dit hoofdstuk wordt onderzocht op welke wijze een bestaande applicatie parallellisme gebruikt en in hoeverre hier sprake is van goede *modularity* en *encapsulation*.
H6: Beschrijft de resultaten van een aantal experimenten om vast te stellen of andere technieken het mogelijk maken om een betere *modularity* en *encapsulation* te bereiken van parallellisme binnen de applicatie van H5.
H7: Dit hoofdstuk brengt in kaart in welke mate de onderzoeksresultaten uit H5 generaliseerbaar zijn.

Afronding:

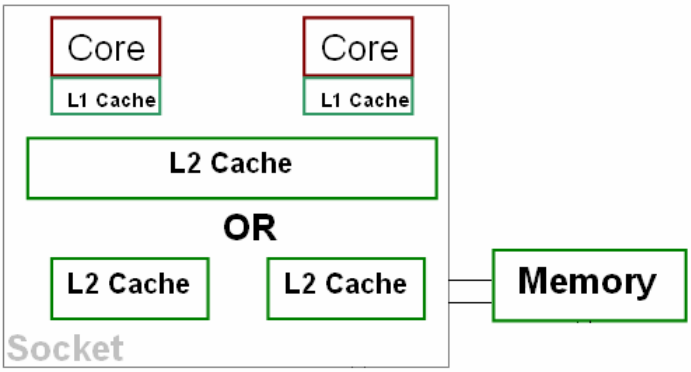
H8: Dit laatste hoofdstuk beantwoordt de vraag of goede *modularity* en *encapsulation* van parallellisme haalbaar zijn. Daarnaast geeft het een beeld van de geleverde bijdrage, de aanbevelingen en *future work*.

2 Probleemanalyse

Dit hoofdstuk geeft een introductie op de opdracht. Het hoofdstuk bestaat uit een uitgebreide probleemanalyse waarin naar voren komt wat de situatie is rondom het probleem dat binnen deze opdracht aangekaart wordt, wat *de trigger* van de opdracht is geweest en waarom deze opdracht relevant is voor de opdrachtgever en de buitenwereld.

Opkomst multi-core

Deze opdracht richt zich op de problemen die ontstaan bij het programmeren van parallelle verwerking in systemen. Dit is noodzakelijk om beter gebruik te maken van de multi-core CPU's. Multi-core CPU's zijn eigenlijk meerdere CPU's in één. Bij een multi-core CPU zijn meerdere verwerkings-/rekeneenheden op één CPU geplaatst. Fysiek is het dus één CPU-package dat in één *socket* op het moederbord geplaatst wordt, maar deze bevat meerdere verwerkingseenheden. Het besturingssysteem ziet deze als verschillende logische processoren, voor de gebruiker is het alsof er meerdere processoren in het systeem aanwezig zijn. *[Intel,2007]*



Figuur 1: Schematisch overzicht multi-core[5]

In deze figuur is verder te zien dat elke *core* zijn eigen L1 *cache* heeft, maar mogelijk een gedeelde L2 *cache*. Intel Core Duo maakt gebruik van een gedeelde L2 *cache*, AMD niet.

Definities:

Package, verpakking waarin processorchips geplaatst worden. Deze *package* wordt in de slots/sockets van het moederbord geplaatst.

Single-core processor, de processor die in 'traditionele' desktop computers is terug te vinden. Deze bezit over één CPU-core in zijn eigen *package*.

Multi-core processor, is een processor waarbij meerdere CPU-cores in één CPU-*package* geplaatst zijn. Deze kunnen verschillende taken tegelijkertijd uitvoeren.

In de oktober uitgave van het IEEE Spectrum magazine van 1989 stond een artikel met de titel *"Microprocessors Circa 2000".* Hierin werd voorspeld dat rond het jaar 2000 de overgang naar multi-core CPU's zou plaatsvinden.
Nu zeventien jaar later blijkt dat de Intel wetenschappers die deze voorspelling deden gelijk hebben gekregen. Meerdere grote CPU-ontwikkelaars hebben producten op de markt gebracht met multi-core techniek. Bedrijven als Intel en AMD richten zich momenteel op de duo- en quad-core, met respectievelijk twee en vier cores, STI (een samenwerking van Sony, Toshiba en IBM) hebben een CPU, '*Cell Broadband Engine*' genaamd, op de markt gebracht met 9 cores. En er worden zelfs al producten voor korte termijn aangekondigd met 16 cores (Sun) en 48 cores (Azul). *[Intel,2007] [Shankland,2006] [Riske, 2005] [AMD, 2005]*

Hiermee is de trend op de CPU-markt gezet, meerdere cores samen bundelen tot één processor. Deze nieuwe generatie processoren kent geen enorme groei in hoeveelheid megahertz, maar haalt zijn potentiële prestatiewinst uit de aanwezigheid van meerdere cores. Het nog verder

verhogen van het aantal megahertz was niet meer mogelijk doordat o.a. de warmte productie voor te veel problemen zou gaan zorgen.

Definities:

Proces, een programma in executie. In principe beschikken processen ieder over hun eigen resources.*[Silberschatz, 2003]*

Thread, onderdeel van een proces. Threads van één proces delen hun code-, datasectie en overige resources. Dit maakt onderlinge communicatie relatief goedkoop met betrekking tot performance*. [Silberschatz, 2003]*

Multi-threaded proces, een proces bestaande uit meerdere threads.*[Silberschatz, 2003]*

Binnen deze scriptie wordt het concept multi-threading gebruikt. Echter in de meeste gevallen is dit inwisselbaar met samenwerkende processen i.p.v. samenwerkend threads.

Parallelle verwerking, een systeem beschikt over parallelle verwerking wanneer er meerdere taken tegelijk uitgevoerd kunnen worden. *[Nakhimovski, 2001]*

Parallel programmeren, het ontwikkelen van systemen welke beschikken over parallelle verwerking.*[Süß,2005]*

Om de potentiële prestatiewinst van multi-core CPU's te benutten dient er bij het ontwikkelen van systemen rekening te worden gehouden met deze nieuwe situatie. Dit betekent dat het potentieel alleen gerealiseerd kan worden door het rekenwerk van het systeem zo goed mogelijk te verdelen over deze cores. Hiervoor zal bij het ontwikkelen van het systeem aandacht besteed moeten worden aan parallelle verwerking en zogenaamde parallelle algoritmen. Hier wordt ook door de fabrikanten van deze CPU's op gewezen. *[Intel,2007][Shankland,2006] [Riske, 2005] [AMD, 2005][Intel,2005]*

> *Even with the added difficulty and complexity of developing multi-threaded applications, by using this design method software vendors will have new opportunities to better differentiate their solutions from competitors and better optimize their software for better performance. As a result, AMD believes a significant number of multi-threaded applications will become available over the next few years, expanding the possibilities of the next "killer app." [AMD, 2005]*

Echter het merendeel van de ontwikkelaars denkt en ontwikkelt sequentiële algoritmen, waarbij de algoritmen het probleem stap-voor-stap oplossen, ook wanneer die stappen parallel uitgevoerd zouden kunnen worden. Dit betekent dat er rekenkracht onbenut zal blijven wanneer geen parallelle verwerking gerealiseerd wordt. *[Sutter,2005]*

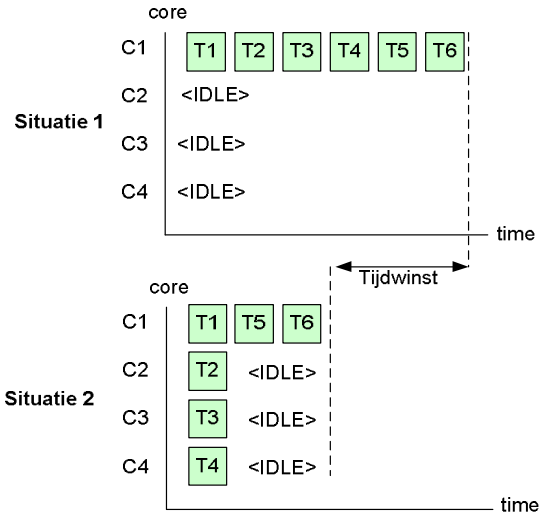
Wanneer we nog iets verder vooruit kijken zien we het *post multi-core* tijdperk al aandringen, *many-core*. Deze term duidt eigenlijk ook op multi-core processoren, maar dan met een veel groter aantal cores (>100). Bij deze processoren zal een *niet* multi-threaded applicatie slechts een minimaal deel van totale potentiële rekenkracht kunnen gebruiken en is de noodzaak van parallelle verwerking dus nog vele malen groter. *[Intel,2005]*

Onderstaande figuur *(Figuur 2)* toont deze situaties schematisch. Het toont een applicatie die bestaat uit vijf onafhankelijke taken en één taak die na afronding van deze vijf kan worden uitgevoerd.

- Situatie 1: weerspiegelt de applicatie waarbij de taken niet parallel, maar sequentieel worden uitgevoerd en de overige rekeneenheden niet benut worden. Hierbij worden geen

taken tegelijkertijd uitgevoerd en moeten ze op elkaar wachten tot de voorgaande is afgerond.

- Situatie 2: weerspiegelt eenzelfde applicatie, maar dan één waarbij de taken *wel* parallel worden uitgevoerd. Doordat het nu niet meer één grote sequentiële taak is kan deze verdeeld worden over de cores, met uitzondering van de taak die achteraf wordt uitgevoerd.



Figuur 2: Vergelijking CPU-utilization single-threaded vs multi-threaded

Zoals al in de inleiding is aangegeven, heeft de opdrachtgever nog weinig ervaring met omvangrijke multi-threaded systemen en vormt dit onderzoek één van de verkenningspaden op dit onderwerp. Zie ook *Bijlage E: Context van de opdracht.*

Complexiteit van parallel programming

In bovenstaand verhaal is duidelijk naar voren gekomen waarom systemen meer gebruik dienen te maken van parallelle verwerking en hoe dit de programma *flow* beïnvloedt. Wat echter nog buiten beschouwing is gelaten zijn de problemen die ontstaan als gevolg van de benodigde extra communicatie / coördinatie als gevolg van de invoering van parallelle verwerking. *[Dijkstra,1968]*
Het is voor ontwikkelaars niet alleen lastig om te zien wat parallel uitgevoerd kan worden, maar ook hoe dit dan in de code gerealiseerd dient te worden. Hierbij ontstaan allerlei communicatie/coördinatie problemen waarvan de oplossingen complex zijn. Dankzij deze complexiteit worden er veel fouten gemaakt, wat slecht werkende multi-threaded programmatuur oplevert met bugs zoals *deadlocks*, *race-conditions* en *priority inversion (zie Bijlage D: Appendix introductie parallel programming).* Deze opdracht besteed aandacht hieraan: *het verlagen van de complexiteit voor het invoeren van parallelle verwerking in systemen*.
Het belang van dit punt wordt ook vaak in de literatuur onderstreept o.a. in *[Robison,2007] [Lee,2006]*.

Traditionele aanpak voor verlagen complexiteit

Er zijn verschillende manieren om de complexiteit van het ontwikkelen van software te verlagen. Echter al jaren wordt het goed scheiden van verantwoordelijkheden gezien als één van de belangrijkste pilaren voor het beheersen van complexiteit. Ook wel *Separation of Concerns[Dijkstra,1974]* genoemd, een term die toegeschreven wordt aan E.W. Dijkstra en D. Parnas *[Constantinides,2002]*.
In 1972 werd al aangegeven door D. Parnas dat de beste manier om dit te bereiken was door modulen te creëren(*modularity*) die beslissingen voor andere modulen verbergen(*encapsulation*). *[Laddad,2003] [Parnas,1972]*

Definities:

Module, gegroepeerde verzameling van codestatements, bijvoorbeeld in de vorm van klassen of aspecten.

Modularity, de mate waarin het systeem de verantwoordelijkheden voor taken heeft ondergebracht in elementen/modulen die ontkoppeld zijn van de overige elementen/modulen. *[Parnas,1972][Silva, 1997]* De functionaliteit in een module dient toe te behoren aan één taak, en één taak zou binnen één module moeten worden ingevuld.*[Yourdon,1989]*

Encapsulation, de mate waarin de implementatiedetails van een module verborgen zijn voor de modulen waarmee hij communiceert, vaak verborgen achter interfaces *[Parnas,1972] [Silva, 1997].* Wijzigingen in een module zouden niet tot wijzigingen in een ander module moeten leiden.*[Yourdon,1989]*

Concern, een functionele of technische eigenschap van het systeem waarvoor een (of meerdere) element(en) uit het systeem verantwoordelijk zijn. Veelal als onderdeel van één of meerdere requirements. *[Laddad,2003] [Marin,2005] [Silva, 1997] [Colyer,2004]*

Een oplossing die de ontwikkelaar in staat stelt om parallelle verwerking zo goed mogelijk in een module onder te brengen en de implementatiedetails verbergt, zou het ontwikkelen zeer vereenvoudigen. De scheiding tussen declaratie en implementatie door middel van interfaces is een hulpmiddel bij het scheiden van verantwoordelijkheden. Een ander bekend middel voor *separation of concerns* is objectoriëntatie, waarbij het opstellen van klassen en het gebruik van overerving gebruikt wordt voor het realiseren van de scheiding van de concerns.
In *[Mattson,2007]* wordt op basis van het model van Thomas Green ook benadrukt dat voor parallel programmeren *encapsulation* en *modularity* belangrijk is.

Belang van modularity en encapsulation

Programmeertalen richten zich dan ook al jaren op het zo goed mogelijk implementeren van deze principes. Binnen *[Laddad,2003]* wordt een aantal negatieve gevolgen opgesomd die in verband staat met slechte *modularity* en *encapsulation*. Te weten: slechte herleidbaarheid van oorzaak en gevolg in de code, verminderde productiviteit, verminderde mogelijkheden voor hergebruik, slechtere codekwaliteit, complexere systeem evolutie. *[Kiczales,2001] [Laddad,2003]*

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.
>
> Dijkstra, On the role of scientific thought (1974)

Met deze opmerking geeft Dijkstra aan dat het verstandig is om, waar mogelijk, de verschillende aspecten van software onafhankelijk van elkaar te behandelen, omdat men zich dan beter kan focussen waardoor het minder complex wordt. Wel is zijn definitie van een ‘concern’ breder.

> We would like the modularity of a system to reflect the way "we want to think about it" rather than the way the language or other tools force us to think about it. In software, Parnas is generally credited with this idea. Kiczales, An Overview of AspectJ (2001)

Twee symptomen van slechte *modularity* en *encapsulation* in objectgeoriënteerde systemen zijn: *scattering* en *tangling* van concerns.

Definities:

Tangling, is het verweven zijn van de implementatie van meerdere concerns in één module. Hierdoor is het lastig aan te wijzen wat het doel is van een specifiek statement in de code. Dus is het lastig te bepalen tot welk concern hij behoort. *[Laddad,2003]*

Scattering, is het verspreid zijn van de implementatie van een concern over meerdere modulen. Als gevolg hiervan kan het zijn dat dezelfde implementatiecode in meerdere modulen terugkomt(duplicaten), of dat iedere module verschillende code omvat om samen het doel te bereiken(complementair). Dus is het lastig te overzien waar het concern geïmplementeerd is.*[Laddad,2003]*

Beperkingen van objectoriëntatie met betrekking tot modularity en encapsulation

Voor sommige concerns geldt dat ze met objectgeoriënteerd-programmeren niet opgelost kunnen worden met een goede invulling van de principes van *modularity* en *encapsulation*. Zoals in *[Kiczales,1997][Constantinides,2002]* is aangegeven bestaat er niet één programmeer paradigma dat goed aansluit bij alle mogelijke concerns. Ieder concern heeft zijn eigen natuurlijke verschijningsvorm en deze verschijningsvormen zijn niet optimaal aan één specifiek paradigma te koppelen. Wanneer concerns niet goed gescheiden zijn noemen we ze *crosscutting concerns*.

Voorbeelden van concerns welke niet goed bij het objectgeoriënteerde programmeerparadigma passen zijn: *logging, business rules, authenticatie, resource pooling, administration, performance, storage management, data persistence, security, multithread safety, transaction integrity, error checking* en *policy enforcement [Marin,2005][Laddad,2003][Kiczales,1997]*

Binnen dit onderzoek wordt voornamelijk aandacht besteed aan concerns die een relatie hebben met de parallelle verwerking van een applicatie. Bijvoorbeeld concerns die betrekking hebben op het delen van *resources* of het starten van *threads.* We noemen dit concurrency concerns. *[Cunha,2006] [Soares, 2004]*

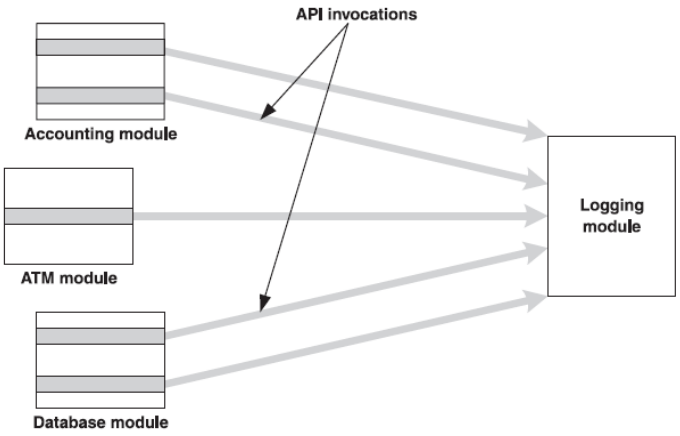
Definities:

Crosscutting concern, een concern dat niet goed ontkoppeld kan worden van andere concerns. Vaak betekent dit dat de implementatie verweven (*tangling*) is met andere concerns en verspreid (*scattering*) is over het systeem. Deze worden ook wel *system-wide* concerns genoemd omdat ze over meerdere modulen verspreid zijn. *[Marin,2005][Laddad,2003]*

Concurrency concern, een concern gerelateerd aan de implementatie van parallelle verwerking.

Vaak wordt gezegd dat het opstellen van interfaces en het gebruik van overerving en polymorphisme het mogelijk maken deze concerns zonder *crosscutting* te implementeren. Dit is deels waar. Het is bijvoorbeeld wel mogelijk om de implementatie van een logging-algoritme in een module onder te brengen en via een API/interface beschikbaar te stellen. Echter ook dan geldt nog steeds dat om het concern te implementeren in alle andere modulen aanroepen naar

deze module moeten worden gemaakt, waardoor nog steeds *scattering* plaatsvindt. Het algoritme is soms wel modulair, het concern blijft echter *crosscutting*. *[Laddad,2003]* *Figuur 3* toont dit.



Figuur 3: Logging als module levert scattering van het concern

Binnen *[Marin,2005]* is een classificatie opgesteld met daarin beschrijvingen van gegeneraliseerde *crosscutting concerns*. De elementen binnen deze classificatie worden *sorts* genoemd. Waarbij zowel beschreven wordt hoe deze zich in objectgeoriënteerde systemen manifesteren, als hoe deze met aspectoriëntatie beter geïmplementeerd zouden kunnen worden.

In *[Constantinides, 2002]* wordt ook aangestipt dat meerdere concurrency concerns waarschijnlijk tot de categorie *crosscutting* behoren. Hierbij wordt aangegeven dat al in de jaren 80 en 90 de ervaring leerde dat het hergebruik en onderhoud op concurrency aspecten niet goed aansloten op de filosofie van objectoriëntatie.

Technieken voor verbetering modularity en encapsulation

Objectoriëntatie levert goede technieken voor *modularity* en *encapsulation,* voor vele concerns, in de vorm van *inheritance*, *interfaces* en *polymorphisme*. Echter zoals al eerder aangegeven is het niet in staat hetzelfde te doen voor *crosscutting* concerns. *[Constantinides, 2002] [Laddad,2003]*
Andere technieken zijn hier mogelijk beter geschikt voor: *generative programming, meta-programming, reflective programming, compositional filtering, adaptive programming, subject-oriented programming, aspect-oriented programming,* en *intentional programming*. *[Laddad,2003]*

Aspect-oriented programming is van deze paradigma's momenteel het populairst *[Laddad,2003]*. Aspect oriëntatie (en aspect georiënteerd programmeren) is een reactie op vaststellen van het bestaan van *crosscutting* concerns in OO-applicaties. De concerns worden in hun natuurlijke vorm in gescheiden modulen beschreven (aspect) en de *weaver* zorgt er voor dat deze bij het *compilen* met elkaar verbonden worden. Om hiermee de voordelen van goede *modularity* en *encapsulation* te bewaren. *[Kiczales, 2001] [Laddad,2003]*
Figuur 4 toont het zelfde *crosscutting* concern als *Figuur 3,* maar dan in AOP geïmplementeerd. Hierbij zijn de verschillen goed te zijn ten aanzien van de *modularity*.

Definities:

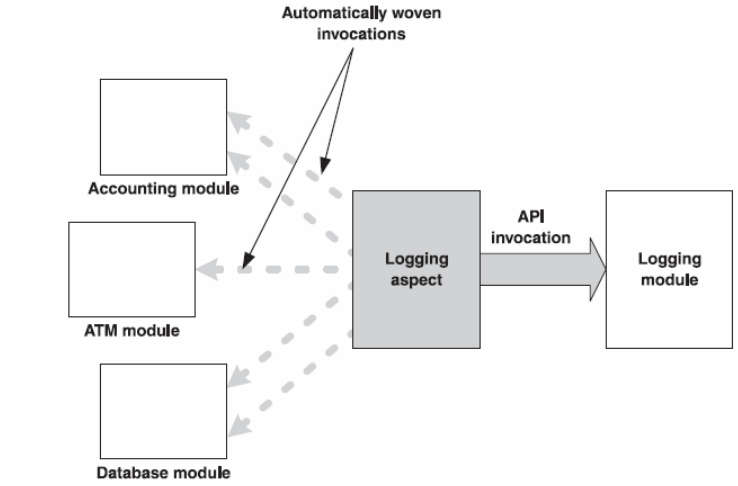
Aspect
Vergelijkbaar met een (*crosscutting*) concern, maar binnen deze scriptie wordt deze term gebruikt voor de implementatie ervan (in AspectJ). *[Kiczales, 2001]*

Aspect oriëntatie (AO)
AO is een (probleem-) analyse paradigma waarbij het probleem wordt opgelost op basis van elementen van het type Aspect. Het levert elementen voor het op een natuurlijke (modulaire)

manier implementeren van *crosscutting concerns*, waardoor deze in de code van elkaar gescheiden blijven. *[Kiczales, 1997] [Kiczales, 2001]*

Aspect georiënteerd programmeren (AOP)
AOP is het ontwikkelen van software in een taal welke gebaseerd is op het AO principe. AspectJ is een voorbeeld van een aspectgeoriënteerde taal. *[Kiczales, 1997] [Kiczales, 2001]*

Weaving
AOP maakt het mogelijk concerns gescheiden van elkaar te beschrijven. Echter tijdens het uitvoeren dienen deze concerns weer automatisch te worden samengevoegd om dezelfde functionaliteit te behouden. Dit samenvoegen van concerns wordt Weaving genoemd en kan zowel tijdens het uitvoeren als het compileren.*[Kiczales, 1997]*



Figuur 4: Logging in AOP

Literatuur over AOP voor scheiding concurrency concerns

Binnen de literatuur is al eerder getracht om AOP in te zetten voor *crosscutting concurrency concerns*. Binnen *[Laddad,2003] [Marin,2005]* zijn enkele concurrency concerns aangemerkt als mogelijk *crosscutting* en in enkele gevallen is ook getracht een AO-oplossing op te stellen. Echter niet altijd is hierbij de AO-oplossing als beter ervaren.

Binnen *[Kienzle,2002]* is met succes vastgesteld dat er sprake was van *crosscutting* bij het gekozen concurrency concern(transacties). Echter de conclusie van dit werk was dat de inzet van AOP geen verbeteringen mogelijk maakte. Eén van de redenen was dat geen volledige *obliviousness* gerealiseerd kon worden, waardoor volgens de schrijver van het artikel geen meerwaarde voor AOP was weggelegd. Als een concern *oblivious* is, dan betekent dit dat de OO code niet aangepast hoeft te worden als de AO code wijzigt, dus dat de OO code en AO code onafhankelijk zijn. Hierdoor zou de AO code ook herbruikbaar zijn voor meerdere applicaties zonder deze aan te hoeven passen.
Het lijkt aannemelijk dat *obliviousness* inderdaad niet altijd mogelijk is voor de implementatie van concurrency concerns in AOP, omdat concurrency te veel verbonden is met de applicatie. Binnen dit onderzoek zal dan ook geen poging gedaan worden om volledige *obliviousness* te realiseren. Echter de verwachting is dat ook zonder *obliviousness* de mate van *modularity* en *encapsulation* verbeterd kan worden.

In *[Soares,2004]* wordt ook AOP ingezet voor het verbeteren van een applicatie, hierbij worden ook enkele concurrency concerns aangepakt. Hierbij is de conclusie op basis van AO-metrieken

dat de AO implementatie beter presteert op de eigenschappen: *Separation of Concerns, Coupling, Cohesion*, en omvang. Echter hierbij is geen verschil gemaakt tussen de onderlinge concerns, dit maakt het onmogelijk om op basis van deze studie uitspraken te doen over specifieke concurrency concerns en *modularity* en *encapsulation* ervan in OO en AO implementaties.
Kortom in *[Soares,2004]* wordt de inzet positief ervaren maar kan dit niet direct worden toegewezen aan specifieke concurrency concerns. In *[Kienzle,2002]* wordt de inzet als mislukt ervaren, maar wordt dit niet gemeten.

[Voelter,2000][Kienzle,2002][Tsang,2003][Laddad,2003][Soares,2004][Cunha,2006] omvatten allen de implementatie van één of meerdere concurrency concerns. Echter het meten van verbeteringen voor bijvoorbeeld de *modularity* en *encapsulation* is minimaal of afwezig.

Inzichtelijk maken van verbetering voor modularity en encapsulation

Het doel van de programma-aanpassingen is het verlagen van de complexiteit van de implementatie van het parallelle gedrag op basis van *modularity* en *encapsulation*. Echter om te meten in hoeverre dit doel behaald is zullen de verbeteringen voor deze principes inzichtelijk gemaakt moeten worden.

Om de mate van *modularity* en *encapsulation* inzichtelijk te maken kunnen broncode-metrieken gebruikt worden. Een probleem is echter dat metrieken vaak gekoppeld zijn aan een programmeerparadigma, er zijn metrieken voor aspectgeoriënteerde software en voor objectgeoriënteerde software. Aangezien er sprake is van vergelijking tussen paradigma's zal niet met één set metrieken kunnen worden afgedaan, metrieken voor object oriëntatie zijn niet direct toepasbaar voor aspect georiënteerde software *[Tsang, 2003][Ceccato,2004]*.

OO-metrieken en AO-programmeerparadigma

Omdat OO metrieken niet direct toepasbaar zijn voor aspect georiënteerde software, zijn er alternatieve metrieken aangedragen in de literatuur*[Tsang, 2003][Ceccato,2004] [Soares,2004]*. In deze werken zijn vervangende sets van metrieken opgesteld specifiek voor software gebaseerd op het AO programmeerparadigma. Hierbij zijn de metrieken voor OO (uit *[Chidamber,1994]*) deels hergebruikt na aanpassing voor AO. Hierbij zijn vaak de definities van de metrieken zo aangepast dat klassen en aspecten als gelijkwaardig gezien worden, onder de noemer module. Doordat de papers dezelfde bron metrieken gebruikten, vertonen de geïdentificeerde metrieken voor AO uit deze papers grote overeenkomst.
Naast de aangepaste OO metrieken, zijn ook enkele nieuwe metrieken geïntroduceerd die alleen voor AO software van toepassing zijn.

Vergelijken van AO implementaties

Uiteraard kunnen voor het vergelijken van AO implementaties de metrieken voor AO uit de vorige deelparagraaf gebruikt worden. Echter binnen *[Kiczales,2005]* worden onderhoudsscenario's gebruikt om verschillende AO implementaties aan elkaar te relateren.
Een nadeel van scenario's ten opzichte van metrieken is dat ze applicatiespecifiek zijn. Dankzij de relatie tussen scenario en applicatie is het niet mogelijk een generieke set scenario's op te stellen die herbruikbaar is voor metingen aan meerdere applicaties.

Kortom er zijn verschillende mogelijkheden voor het in kaart brengen van de *modularity* en *encapsulation* van applicaties; OO-metrieken, AO-metrieken en onderhoudsscenario's.

In het volgende hoofdstuk zal beschreven worden welk deel van het probleemgebied aangepakt gaat worden binnen dit onderzoek, wat de scope van het onderzoek is en wat de onderzoeksmethode is.

3 Onderzoeksmethode

In hoofdstuk 2 is uiteengezet wat het probleemkader is van deze opdracht en wat de *trigger* is van dit onderzoek. Dit hoofdstuk beschrijft hoe het onderzoek uitgevoerd zal worden.

Trigger:

In de toekomst zal de rol van concurrency bij de ontwikkeling van applicaties steeds belangrijker worden als gevolg van de introductie van multi-core CPU's. De implementatie van concurrency (parallel programming) is echter lastig. Twee bekende principes voor het in kaart brengen van code complexiteit zijn modularity en encapsulation. Het is dus interessant om te weten hoe deze principes toepasbaar zijn voor concurrency om de complexiteit te beheersen.

3.1 Probleemstelling en onderzoekvragen

Het onderzoek is ingericht rond de volgende centrale probleemstelling.

Probleemstelling:

In welke mate zijn de principes van modularity en encapsulation toepasbaar voor de implementatie van concurrency (parallel programming) in objectgeoriënteerde systemen?

Deze probleemstelling zal worden aangepakt in twee hypothesen, één om aan te tonen hoe slecht objectoriëntatie presteert m.b.t. deze principes (*het bestaan van het probleem*), en één om aan te tonen dat er verbetering mogelijk is met andere programmeerparadigma's (*het bestaan van een oplossing*).

Hypothesen:

1) "Binnen bestaande objectgeoriënteerde multi-threaded software is sprake van slechte modularity en encapsulation (crosscutting) bij de implementatie van concurrency gerelateerde concerns."
2) "Met behulp van AspectJ en aspect refactoring is het mogelijk de concurrency concerns met betere modularity en encapsulation te implementeren."

Ter onderbouwing van deze probleemstelling zijn de volgende onderzoeksvragen vastgesteld:

Onderzoeksvragen:

1) *Wat is parallellisme en parallelle verwerking en wat zijn de verschillen met sequentiële verwerking ? Welke zaken spelen een rol bij de implementatie hiervan ?*
2) *Wat is Aspect georiënteerd programmeren (AOP) ? Hoe is AOP toepasbaar ?*
3) *Gegeven een applicatie met concurrency, welke concerns met betrekking tot concurrency kunnen we identificeren in deze applicatie ? In welke mate is er sprake van crosscutting (tangling en scattering) van deze concerns?*
4) *Is het mogelijk crosscutting concurrency concerns in AspectJ te implementeren ? Levert de AspectJ-oplossing een betere toepassing van modularity en encapsulation ?*

3.2 Toelichting onderzoeksvragen

Onderzoeksvraag 1:
Wat is parallellisme en parallelle verwerking en wat zijn de verschillen met sequentiële verwerking? Welke zaken spelen een rol bij de implementatie hiervan ?
Motivatie:
Het antwoord op deze vraag gaat ons helpen met het opbouwen van een visie over dit complexe onderwerp en bij het bedenken van mogelijke concurrency concerns waarvan we later de *modularity* en *encapsulation* willen vaststellen/verbeteren (in onderzoeksvraag 3 en 4). Bovendien is de kennis die wordt opgedaan, interessant voor de opdrachtgever en deze zal daarom aan hem worden overgedragen via bijv. een workshop.
Methode:
Om deze vraag te beantwoorden zal een literatuurstudie plaatsvinden. Waarbij ook nadrukkelijk wordt gekeken naar zaken die het ontwikkelen van applicaties met parallelle verwerking complex maken. Daarnaast zullen enkele interviews worden gehouden om de rol van multi-threading voor de opdrachtgever in kaart te brengen.
Uitgewerkt in: H4.1 Introductie parallel programming en Bijlage D: Appendix introductie parallel programming

Onderzoeksvraag 2:
Wat is Aspect georiënteerd programmeren (AOP) ? Hoe is AOP toepasbaar ?
Motivatie:
Het antwoord op deze vraag dient ter verkenning van AOP. AOP zal als alternatief worden ingezet van OOP bij het achterhalen van de mogelijkheden voor het verbeteren van de *modularity* en *encapsulation* van concurrency concerns (in onderzoeksvraag 4).
Methode:
Om deze vraag te beantwoorden zal een literatuurstudie plaatsvinden. Daarnaast zullen ook enkele AspectJ programma's bekeken (en gemaakt) worden om kennis op te doen van AOP.
Uitgewerkt in: H2 Probleemanalyse en H4.2 Introductie aspect mining en refactoring

Onderzoeksvraag 3:
Gegeven een applicatie met concurrency, welke concerns met betrekking tot concurrency kunnen we identificeren in deze applicatie? In welke mate is er sprake van crosscutting (tangling en scattering) van deze concerns?
Motivatie:
Het antwoord op deze vraag geeft ons inzicht in de mate waarin de *modularity* en *encapsulation* principes zijn toegepast op de concurrency concerns. Hierbij wordt de eerste hypothese getoetst.
Methode:
De volgende stappen vormen de aanpak van dit deel van het onderzoek: • Stap 0: Selecteren van een te onderzoeken applicatie *(Uitgewerkt in H5.1)* • Stap 1: Identificatie en classificatie van *concurrency concerns* *(Uitgewerkt in H5.2)* • Stap 2: In kaart brengen *tangling* van de concerns *(Uitgewerkt in H5.3)* • Stap 3: In kaart brengen *scattering* van de concerns *(Uitgewerkt in H5.4)* • Stap 4: Controle classificatie van de concerns *(Uitgewerkt in H5.5 & H5.6)* • Stap 5: Conclusie en reflectie *(Uitgewerkt in H5.7)*

Als eerste zal een applicatie geselecteerd worden waarvan de implementatie van concurrency concerns onderzocht kan worden. Deze applicatie zal aan een aantal vooraf opgestelde eisen moeten voldoen om het kader waarin de resultaten geldig zijn vast te leggen, zie *scope*. ***(Stap 0)***

Vervolgens zal in kaart worden gebracht welke concurrency concerns aanwezig zijn binnen de geselecteerde applicatie, *aspect mining*. Dit vindt plaats door te zoeken naar bekende concurrency concerns (op basis van kennis uit onderzoeksvraag 1) en een *fan-in* analyse. Deze analyse helpt ook bij het bepalen van de prioriteiten van de concerns. *[Marin,2006]*
Naast de identificatie vindt ook classificatie van concerns plaats op basis van het model uit *[Marin,2005].* Het maken van deze koppeling is interessant omdat voor deze standaard typeringen ook standaard oplossingsrichtingen bekend zijn. Zie *H4.2* voor meer informatie over *aspect mining* en classificatie. ***(Stap 1)***

Hierna zal van enkele concurrency concerns onderzocht worden in welke mate zij de principes van *modularity* en *encapsulation* invullen. Dit zal gedaan worden door in kaart te brengen of er sprake is van *tangling (indicatie slechte encapsulation)* en *scattering (indicatie slechte modularity)* van deze concerns. ***(Stap 2)*** **(*Stap 3*)**

Vervolgens wordt gecontroleerd of de concernclassificatie die is vastgesteld bij *stap 1* en de *tangling* uit *stap 2* correct is. Hiervoor wordt de code van de applicatie doorlopen en wordt van de implementaties van de concerns onderzocht of ze aan hun concernclassificatie voldoen. **(*Stap 4*)**

Tot slot wordt op basis van de voorgaande stappen een conclusie getrokken en volgt een korte reflectie op enkele aspecten uit de aanpak. ***(Stap 5)***

Scope:

Correctheid van de applicatie
Binnen dit onderzoek wordt geen aandacht besteed aan de correctheid van de applicatie, het doel is niet om bugs te vinden. Bijvoorbeeld, er zal niet gecontroleerd worden of er concurrency bugs zoals *deadlocks* op kunnen treden. Dit heeft ook geen invloed op het antwoord van de onderzoeksvraag, er is geen directe relatie met *modularity* en *encapsulation*.

Aantal onderzochte applicaties (en bijbehorende selectie criteria)
Bij de casestudy, zal van één applicatie in kaart gebracht worden welke (*crosscutting*) concurrency concerns hij omvat. Een aanname hierbij is dat deze applicatie (en de selectie van de concerns) representatief is voor objectgeoriënteerde multi-threaded applicaties. Er is daarom in overleg met de opdrachtgever het volgende kader opgesteld:

- *De applicatie is geschreven in Java en beschikt over multi-threading*
- *De applicatie beschikt over geëvolueerde introductie van concurrency*
- *Broncode van de applicatie is vrij toegankelijk*
- *De applicatie heeft baat bij multi-threading*

Deze eisen zijn bepaald door de context van de opdracht, namelijk de Java-afdeling van de opdrachtgever. Waar een opkomende interesse is voor multi-threading. Echter ze hebben nog geen ervaring met het bewust ontwikkelen van multi-threading in applicaties. Het te onderzoeken systeem zou daarom een voorbeeldfunctie kunnen vervullen (voor nieuwbouw of evolutie van bestaande systemen). Er dient daarom dus ook een duidelijke rol voor concurrency te zijn.

Validatie:

De analyse vindt plaats op één applicatie, de waarde van de resultaten uit dit onderzoek zijn dus sterk afhankelijk van de representativiteit van deze applicatie. Deze representativiteit zal daarom gevalideerd worden om generalisatie van de resultaten mogelijk te maken. De uitwerking hiervan is terug te vinden in *H7 Generalisatie.*

Uitgewerkt in: H5 Aspect mining van concurrency concerns

Onderzoeksvraag 4:
Is het mogelijk crosscutting concurrency concerns in AspectJ te implementeren? Levert de AspectJ-oplossing een betere toepassing van modularity en encapsulation?
Motivatie:
Deze vraag dient om ons inzicht te verschaffen in de mogelijkheden van AOP ten opzichte van OOP met betrekking tot de verbetering van *modularity* en *encapsulation* voor concurrency concerns. Hierbij wordt de tweede hypothese getoetst.
Methode:
De volgende stappen vormen de aanpak van dit deel van het onderzoek. • Stap 1: Verzamelen van locaties waar de concern implementaties zich bevinden. • Stap 2: Opstellen testcases op basis van de huidige concern implementatie. • Stap 3: Opstellen AspectJ implementatie van het concern. *(Uitgewerkt in H6.2)* • Stap 4: Applicatie-refactoring in verschillende AspectJ-stijlen. *(Uitgewerkt in H6.2)* Voor iedere locatie waar de implementatie van het concern voorkomt: a. *Verwijderen objectgeoriënteerde implementatie voor de locatie* b. *Implementeren van concern in AspectJ voor de locatie* • Stap 5: Meten van *modularity* en *encapsulation* van de concerns *(Uitgewerkt in H6.3)* • Stap 6: Analyse meetresultaten *(Uitgewerkt in H6.4)* • Stap 7: Conclusie en reflectie *(Uitgewerkt in H6.5)* Eerst zal onderzocht worden hoe en waar de concerns geïmplementeerd zijn ***(Stap 1)***, hiervoor worden de resultaten uit onderzoeksvraag 3 gebruikt. Vervolgens zal in een aantal experimenten getracht worden om een goede en volledig werkende aspectgeoriënteerde implementatie te realiseren van de concerns. Binnen deze experimenten zullen verschillende stijlen (*zie H4.2.4*) van ontwikkelen in AspectJ getest worden. Dit omdat de stijl mogelijk invloed heeft op de *modularity* en *encapsulation*. Daarnaast wordt ook de performance van de OO- en AO-oplossingen met elkaar vergeleken. Immers wanneer de invoering van AspectJ een grote performance vermindering zou opleveren, is het de vraag wat de waarde is van de behaalde onderzoeksresultaten. ***(Stap 2 en 3)*** Hierna worden enkele *branches* gemaakt van de onderzochte applicatie. Vervolgens wordt in iedere *branch* één concern vervangen op basis van één AOP oplossingstijl. Hierdoor kunnen later bij de analyse de gevolgen per concern en per stijl afzonderlijk bepaald worden. Bovendien wordt ook een baseline gemaakt voor elk concern, dit houdt in dat er een *branch* wordt aangemaakt waarin alle code voor het concern verwijderd is. Hierdoor is te zien wat de waarde is voor de metrieken in de afwezigheid van het concern. Dit maakt het mogelijk een indicatie te krijgen van de significantie van de verbetering tussen de AO en OO *branches* ***(Stap 4)*** Nadat deze aanpassingen gerealiseerd zijn, zal gemeten worden of de *encapsulation* en *modularity* daadwerkelijk verbeterd is. Om dit in kaart te brengen zijn metrieken geselecteerd die inzicht geven hierin. Meer hierover in de paragraaf *Scope*. ***(Stap 5)*** Met behulp van de scores van de oplossingen bij deze metrieken wordt geanalyseerd of er verbeteringen gerealiseerd worden met het gebruik van AOP voor de *modularity* en *encapsulation*. ***(Stap 6)*** Tot slot wordt op basis van de voorgaande stappen een conclusie getrokken en volgt een korte reflectie op enkele aspecten uit de aanpak.***(Stap 7)***

Scope:
Paradigma's en talen Het antwoord op de onderzoeksvraag is afhankelijk van zowel het programmeerparadigma als de implementatie van dit paradigma in een programmeertaal. In verband met de doorlooptijd van de opdracht, zullen twee programmeerparadigma's met elkaar vergeleken worden, ieder op basis van één taal. Java als taal voor OO en AspectJ als taal voor AO, gebaseerd op de voorkeur van de opdrachtgever. **Metrieken** In de probleemanalyse(*H2*) is reeds aangegeven dat het meten van *modularity* en *encapsulation* verbeteringen over meerdere programmeerparadigma's lastig is. Binnen dat hoofdstuk is aangegeven dat er drie richtingen hiervoor zijn (OO Metrieken, AO Metrieken en scenario's). Alle drie zijn toegepast binnen dit onderzoek, hiermee wordt een te eenzijdige blik op de materie voorkomen. Een uitgebreide beschrijving van de metrieken is te vinden in *H6.1*. De geselecteerde metrieken en scenario's geven inzicht in de *modularity*, *encapsulation* en daarnaast ook de complexiteit. Immers we wilden de complexiteit aanpakken op basis van deze principes. Bij de OO metrieken is gekozen voor metrieken uit de set van *[Chidamber,2004]*. De metrieken uit dit artikel zijn algemeen geaccepteerde metrieken voor objectoriëntatie *[Tsang, 2003]*. Bij de AO metrieken is geselecteerd uit de set van *[Soares,2004][Tsang,2003]*. De metrieken hieruit zijn deels gebaseerd op *[Chidamber,2004]*, maar dan aangepast voor AO. Voor het opstellen van scenario's is gebruik gemaakt van de voorbeelden in *[Kiczales,2005]* en input van collega's. Hierbij wordt per scenario gemeten hoeveel eenheden (modulen en regelscode) aangepast moeten worden als indicatie van *modularity* en *encapsulation*.
Uitgewerkt in: H6 Aspect refactoring van concurrency control constructs

3.3 Samenvatting

Er zijn binnen dit hoofdstuk een aantal onderzoeksvragen en hypothesen opgesteld die samen het antwoord zouden moeten leveren op de probleemstelling.

De eerste en tweede onderzoeksvraag zorgen voor de nodige ondersteuning uit de literatuur om het onderwerp te bevatten en het onderzoek uit te voeren.

Onderzoeksvraag drie richt zich op het eerste gedeelte van de probleemstelling (de eerste hypothese), namelijk het in kaart brengen van de (slechte) *modularity* en *encapsulation* van de implementatie van concurrency concerns in een applicatie.

Onderzoeksvraag vier vult dit aan door de mogelijkheden voor verbetering van de *modularity* en *encapsulation* van de implementatie van concurrency concerns te onderzoeken. Hiermee wordt het tweede deel van de probleemstelling (de tweede hypothese) aangepakt.

Op basis van de resultaten uit deze onderzoeksvragen en hypothesen kan een conclusie getrokken worden over de probleemstelling (binnen de gestelde scope).

4 Achtergrondinformatie

Binnen dit hoofdstuk zal achtergrondinformatie verschaft worden op het gebied van parallellisme. Hiermee kan de lezer een beeld vormen van de verschillen tussen systemen met sequentiële verwerking en systemen met parallelle verwerking. De informatie uit dit hoofdstuk is grotendeels verzameld gedurende de literatuurstudie en vormt het antwoord op de eerste onderzoeksvraag. In *Bijlage D: Appendix introductie parallel programming* is meer achtergrondinformatie terug te vinden over dit onderwerp.

Daarnaast worden in het tweede deel van dit hoofdstuk technieken uit de wereld van aspectoriëntatie aangehaald die later in dit onderzoek (*H5, H6*) toegepast worden. Dit is tevens ook het antwoord op de tweede onderzoeksvraag.

4.1 Introductie parallel programming

Binnen deze sectie zal een korte introductie plaatsvinden op het vakgebied van parallel programming.

4.1.1 Wat is parallel gedrag

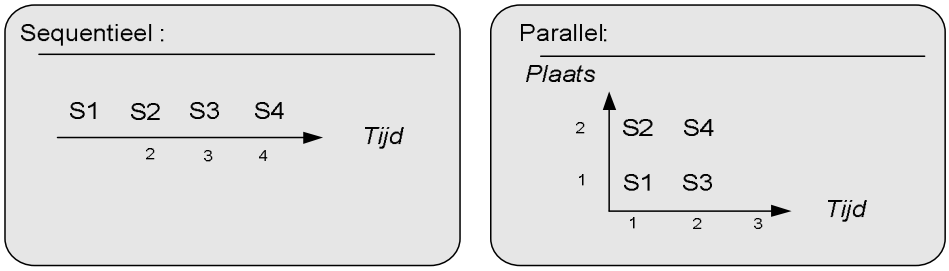
De literatuur *[Nakhimovski, 2001] [Lea,1999]* maakt een onderscheid tussen concurrent, parallel en sequentieel gedrag.

- Een applicatie bevat *sequentieel gedrag* wanneer de statements verdeeld zijn over ***tijd***, de statements worden na elkaar uitgevoerd. *[Silberschatz, 2003]*

- Een applicatie bevat *concurrent gedrag (concurrency)* wanneer statements tegelijkertijd over meerdere componenten uitgevoerd *kunnen* worden. Waarbij minimaal twee threads ***voortgang*** maken, maar niet noodzakelijk tegelijkertijd (door afwisselend een statement uit te voeren op één CPU). Er kunnen dus statements tegelijkertijd op verschillende plaatsen (verschillende *cores* of CPU's) uitgevoerd worden, maar soms ook op dezelfde plaats snel achterelkaar. *[Nakhimovski,2001] [Lea,1999][Silberschatz, 2003]*

- Een applicatie bevat *parallel gedrag* wanneer statements over *plaats* en *tijd* verdeeld zijn. Parallellisme wordt gekenmerkt door het ***simultaan uitvoeren*** van minimaal twee threads. Bij concurrent is dit optioneel, bij parallel verplicht. Bij parallelle verwerking geldt dat de statements ook *daadwerkelijk* op een andere plaats worden uitgevoerd. Of een applicatie concurrent of parallel gedrag bevat is dus onder andere afhankelijk van de omgeving(*run-time)* niet van de code (*compile time*).*[Nakhimovski,2001] [Lea,1999][Silberschatz, 2003]*

De figuur hieronder (*figuur 5*) geeft het verschil in voortgang weer tussen een sequentieel programma (als een ééndimensionaal model met statements op basis van tijd) en een parallelprogramma (als een tweedimensionaal model). De eenheid van de *plaats-as* is dan het aantal cores van een multi-core CPU.



Figuur 5: Vergelijking van verdeling statements over dimensies

4.1.2 De onvoorspelbaarheid van concurrency

Bij sequentiële applicaties is de volgorde waarin statements uitgevoerd worden een vastgesteld feit, bij parallel kan de volgorde verschillen per *run*. De volgorde van statements binnen threads blijft gelijk, maar de volgorde van statements tussen threads niet. Bijvoorbeeld in *Figuur* 5 is stap 1 altijd voor 3, maar 1 is niet altijd voor 4. De volgorde waarin threads gebruik mogen maken van de CPU wordt *scheduling* genoemd. Er zijn meerdere algoritmen voor het bepalen van deze volgorde (o.a. op basis van prioriteit en/of wachttijd). Het moment van wisselen kan op twee wijzen bepaald worden, soms mogen de threads zelf bepalen wanneer ze de processor afstaan (*non pre-emptive*), maar soms bepaald het OS dat een thread gepauzeerd wordt (*pre-emptive*). Het wisselen van de actieve thread op een CPU wordt een *context-switch* genoemd. Wanneer twee threads *pre-emptive* een CPU delen zijn er een enorme variatie aan afwisselschema's mogelijk(*interleavings*). *[Silberschatz,2003]*

Deze *interleavings* zorgen voor onvoorspelbaarheid (non-determinisme) bij bijvoorbeeld het testen, een bug kan optreden bij bepaalde *interleaving(s)* en afwezig zijn bij andere. *[Lea,1999] [Lee,2006]* Echter de kosten voor *context-switching* zijn hoog waardoor in de praktijk de variatie meevalt (hoge granulariteit).
Echter bij architecturen zoals multi-core zijn er twee threads tegelijk bezig en zou men dus kunnen spreken van interleaving met minimale granulariteit. Een gevaar hierbij is dat bugs die tot nu verborgen bleven, met de komst van multi-core toch te voorschijn komen.*[Lee,2006]*

4.1.3 Motivaties voor concurrency

Hieronder staan een aantal redenen opgesomd die, los of in combinatie, als motivatie kunnen dienen om concurrency te implementeren*[Silberschatz,2003][Amarasinghe,2007][Sutter,2007]*

- ***Performance/Utilization (Computation speedup)***
  Het opdelen van een systeem in subsystemen/taken kan een snelheidsverbetering opleveren doordat deze subsystemen tegelijkertijd voortgang kunnen maken.
  Met name deze motivatie zal een grotere rol spelen met de introductie van multi-core CPU's. Zonder concurrency zal de applicatie maar gebruik kunnen maken van slechts één van de cores, wat zonde is van de overige rekencapaciteit.

- ***Modularity***
  Het opdelen van een systeem in meerdere deelsystemen kan de complexiteit verlagen. Eén groot systeem is soms veel lastiger te overzien dan enkele kleine.

- ***Convenience (Following Natural Application Structure)***
  Ondersteuning van de gebruiker, deze wil soms meerdere zaken tegelijk uitvoeren, printen, spelling checken, typen etc. De wereld, die we ondersteunen met software, is niet sequentieel, dus de software zou dit ook niet 'moeten' zijn. Wanneer bijvoorbeeld het ondersteunde bedrijfsproces uit meerdere parallelle taken bestaat, dan zou de applicatie structuur deze 'natuurgetrouw' moeten volgen.

- ***Responsiveness***
  Door taken op de achtergrond als subtaak uit te voeren kan de response tijd van het systeem verlaagd worden. Door bijvoorbeeld binnenkomende request bij een webserver in een aparte thread af te laten handelen, blijft de server beschikbaar/*responsive*.

- ***Resource Sharing***
  Bijvoorbeeld door gedeelde toegang op een apparaat of gedeeld geheugen.

Voor meer informatie over de mogelijkheden en beperkingen van concurrency met betrekking tot performance, zie *Bijlage D: Appendix introductie parallel programming.*

4.1.4 Welke applicaties zijn geschikt voor concurrency

In de voorgaande paragraaf is naar voren gekomen dat er gegronde redenen zijn om een applicatie van parallel karakter te voorzien. De vraag hierbij is: *is elke applicatie geschikt?*

Sommige problemen kennen van nature een sterk parallel karakter, waardoor de applicaties waarin deze problemen opgelost worden vrijwel ongelimiteerd (zonder extra inspanning) opschalen (met meer threads). Dit kan vaak doordat de threads nauwelijks data delen. Deze problemen noemen we *embarrassingly parallel problems [Lee,2006]*. Een voorbeeld is 3d projectie, waarbij pixels onafhankelijk van elkaar berekend kunnen worden. Deze zijn dus vrijwel optimaal geschikt voor concurrency.

Maar in principe kan voor bijna iedere applicatie één van de eerder genoemde redenen als motivatie gebruikt worden. Het is echter niet zo dat iedere reden voor iedere applicatie een motivatie kan zijn. Bijvoorbeeld sommige applicaties zullen geen of nauwelijks prestatiewinst kunnen behalen, maar mogelijk dat andere reden wel een geschikte motivatie vormt. *[Lea,1999] [Goetz,2006]*
Het toevoegen van parallel gedrag kan dus vrijwel altijd, er is bijna altijd wel één van de motivaties van toepassing. Of het zinvol is, is vooral afhankelijk van de kosten die er met meekomen. Deze kosten uiten zich in bijvoorbeeld extra ontwikkelkosten als gevolg van gevaren en problemen die ontstaan bij het toevoegen van parallel gedrag.

Op hoog abstractie niveau gezien, zou men kunnen spreken van twee vormen van parallellisme binnen applicaties *[Rabbah, 2007] [Larus,2007]*:

- ***Data parallelism (Data decomposition)***
  Het uitvoeren van een bewerking op een grote dataset. Hierbij wordt de data in blokken opgedeeld die parallel aan elkaar dezelfde bewerking kunnen ondergaan.

  Bijvoorbeeld: Beeldbewerking, waarbij dezelfde bewerking op ieder frame uit een film wordt toegepast.

- ***Control parallelism (Task decomposition)***
  Het uitvoeren van verschillende bewerking naast elkaar. Verschillende subtaken die samen de hoofdtaak vormen worden dan tegelijkertijd uitgevoerd.

  Bijvoorbeeld: Tekstverwerker, waarbij de taken spellingcontrole en tekstinvoer verwerken gelijktijdig worden uitgevoerd.

Binnen *[Intel,2005] [Manocha,2007]* worden applicaties (typen van applicaties) genoemd die nu mogelijk uitvoerbaar worden op werkstations dankzij de introductie van parallelle hardware(multi-core).

4.1.5 Communicatie-modellen

Dijkstra *[Dijkstra, 1968]* geeft aan dat als gevolg van proces opdeling vaak communicatie tussen deze delen nodig is. Op het moment dat een ‘probleem’ parallel opgelost wordt, ontstaat de behoefte aan communicatie tussen deze oplossingseenheden. In essentie is het ook mogelijk om sommige problemen parallel op te lossen zonder communicatie, maar voor de meeste problemen geldt toch dat uiteindelijk ergens in de verwerking communicatie nodig is. Deze communicatie kan gebruikt worden om synchronisatie te laten plaatsvinden voor bijvoorbeeld het delen van data of het afstemmen van de volgorde van uitvoeren, of om af te spreken wie wanneer gebruik mag maken van gedeelde resources. Het uiteindelijk doel is om de toestand van de applicatie correct te houden en omdat de toestand van de applicatie verdeeld is over meerdere processen is er sprake van een *shared state. [Dijkstra,1968]*

Binnen de literatuur *[Per Brinch Hansen,1989]* wordt dit als volgt gekarakteriseerd. Het abstracte model voor berekeningen wordt *process* genoemd. Een sequentieel proces voert hierbij stap voor stap zijn taak uit, een parallel proces doet meerdere stappen tegelijk en kan gezien worden als een set van sequentiële processen. Communicatie kan dan omschreven worden als de overdacht van data tussen meerdere processen.
De meest voorkomende modellen zijn *message passing* en *shared memory*.*[Needham,1979]*

Voor meer achtergrondinformatie zie *Bijlage D: Appendix introductie parallel programming.*

4.1.6 Hulpmiddelen voor consistent en correct houden shared state

Echter met de introductie van communicatie ontstaan ook een aantal problemen en gevaren. Men dient de *shared state* consistent te houden, deze bestaat uit de data die gebruikt wordt door meerdere *threads*. Om dit goed te laten verlopen worden allerhande hulpmiddelen gebruikt, ook wel *concurrency control constructs* genoemd. Voorbeelden hiervan zijn: *Mutexen, locks*, monitoren en semaphoren.

Deze hulpmiddelen stellen de ontwikkelaar in staat om één (of meerdere) stuk(ken) programma zo te coördineren dat deze slechts vanuit één (of een beperkt aantal) thread(s) tegelijkertijd benaderd kan worden. Wat men eigenlijk wil is tijdelijk exclusieve toegang tot één bepaalde (of meerdere) datastructuren. De code waarin we exclusieve toegang willen, noemen we een *critical region.* En het beschermen van zo'n *critical region* voor gelijktijdig gebruik, noemen we *mutual exclusion*. Al kan het soms zijn dat, opzettelijk of per ongeluk, de regio die beschermd wordt door *mutual exclusion* groter is dan de *critical region*. *[Dijkstra,1969]*

Voor meer achtergrondinformatie zie *Bijlage D: Appendix introductie parallel programming.*

4.1.7 Gevaren en problemen

Foutief gebruik van de hulpmiddelen uit de vorige sectie kunnen zorgen voor fouten in de werking van de applicatie. Bijvoorbeeld er kan een fout gemaakt worden bij het afspreken wie er wanneer gebruik mag maken van gedeelde resources. Een gevolg van deze fout kan zijn dat er *race-conditions* of *deadly embrace (deadlocks)* optreden. *[Dijkstra,1968]*

Andere bugs/gevaren die kenmerkend zijn voor systemen met concurrency zijn:
Race-conditions, starvation, unfair load balancing, priority inversion en contention convoying.Zie o.a. [Dijkstra,1969][Silberschatz,2003][Lea,1999][Goetz,2006]

Een groot nadeel van deze bugs ten opzichte van 'normale' bugs is dat ze vaak afhankelijk zijn van de *scheduling* en daardoor lastig te vinden of repliceren zijn. Hierdoor kunnen ze jarenlang verborgen blijven en op het minst geschikte moment naar boven komen.*[Amarasinghe,2007]*

Om het werk eenvoudiger te maken worden er oplossingen gezocht in de richtingen van tools, frameworks en programmeertalen met meer aandacht voor concurrency. Eén van de bekendste technieken die in opkomst zijn is het *Software Transactional Memory(H9.2.3)*, waarbij *shared memory* wordt voorzien van transactioneel gedrag.

Voor meer achtergrondinformatie zie *Bijlage D: Appendix introductie parallel programming*

4.2 Introductie aspect mining en refactoring

Binnen deze scriptie worden ook methoden en technieken uit de wereld van aspectoriëntatie toegepast (*in H5, H6*). Hieronder is achtergrondinformatie te vinden over de belangrijkste technieken rondom aspectoriëntatie voor deze scriptie; *Concern Classificatie*, *Aspect Mining* en *Aspect Refactoring*.

4.2.1 Concern Classificatie

Om communicatie over veel voorkomende *crosscutting concerns* eenvoudig te maken is in *[Marin, 2005] [Marin,2005b]* een classificatiesysteem (idioom) opgesteld van *crosscutting concern* typen. Binnen dit idioom zijn generieke beschrijvingen terug te vinden van de kernmerken van verschillende veel voorkomende concerntypen. Zo'n concerntype wordt ook wel *sort* genoemd.

Het doel van dit idioom is tweezijdig, enerzijds het vergemakkelijken van het herkennen en vaststellen van de *crosscuttingness* van een concern, anderzijds om helder erover te kunnen communiceren en het vastleggen van herbruikbare oplossingsstrategieën. Hierbij zou een *crosscutting concern* idioom een vergelijkbare rol kunnen spelen als het idioom van *design patterns [Gamma,1995]* .

Binnen deze scriptie spelen de volgende *sorts* een rol:

- *consistent behavior*, een type dat bestaat uit vast een patroon van methode-aanroepen, bijvoorbeeld aan het begin en/of einde van bepaalde methoden.
- *add variablity*, een type waarbij een object van een *anonymous* klasse gebruikt wordt om een methode (functionaliteit) als parameter door te geven.
- *dynamic behavior enforcement*, een type waarbij het gebruik van methoden van een object door 'regels' gedefinieerd wordt, maar deze regels niet in de code verwerkt zijn.

Definities:

Sort, in *[Marin,2005]* is een classificatie opgesteld van veel voorkomende *crosscutting concern* typen. Zo'n cross-cutting concern type wordt een *sort* genoemd.

Consistent behavior, wanneer een concern bestaat uit het consistent toevoegen van een rol of gedrag aan een systeem, noemt men dit een *consistent behavior*. Het gedrag moet als een vast patroon van stappen omschreven kunnen worden en de locaties waar het gedrag wordt toegevoegd moeten voorspelbaar zijn. Bijvoorbeeld altijd aan het begin en/of eind van bepaalde methoden*[Marin,2005].*

Add variability, dit is de classificatie van het objectgeoriënteerde patroon waarbij objecten van *anonymous* klassen gecreëerd worden om een methode (functionaliteit) als parameter te kunnen gebruiken. *[Marin,2005].*

Dynamic behavior enforcement, wanneer het gebruik van methoden van een object door 'regels' gedefinieerd wordt, maar deze niet in de code verwerkt zijn. Bijvoorbeeld door een bepaalde aanroepvolgorde te vereisen voor het gebruik van een aantal methoden (eerst initialization, dan functionaliteit, dan finalization). Alleen zijn deze eisen dan niet in code, maar in bijvoorbeeld commentaar verwerkt *[Marin,2005].*

Een extra nadeel van *add variability* is dat de syntax onduidelijk is, binnen *[Sierra, 2006] (instructie boek voor officiële Java certificering)* omschreven als :'*the most unusual syntax you might see in java*'. Als gevolg hiervan is de intentie van de code slecht zichtbaar, terwijl dit erg belangrijk is voor het beperken van de complexiteit *[McConnel,2004]*

4.2.2 Aspect Mining

Het zoeken naar *crosscutting concerns* binnen software wordt ook wel *aspect mining* genoemd. Hierbij wordt de code van het systeem onderzocht op de implementatie van concerns, om vast te stellen in welke mate er sprake is van *crosscutting* concerns. Het doel hierbij is het documenteren van deze concerns.

Er zijn verschillende strategieën voor *aspect mining*. Binnen *[Marin, 2006]* worden de volgende twee voorgesteld: *query based* en *generative based*.

- *Query based* is het doorzoeken van de code met een opgegeven patroon.
- *Generative based* is het automatisch zoeken van patronen (bijvoorbeeld via *fan-in*).

Binnen deze scriptie wordt bij *aspect mining* niet alleen de *crosscutting concerns* geïdentificeerd, maar ook geclassificeerd volgens het eerder beschreven model. Omdat dit zoals al eerder is aangegeven, voordelen heeft bij het herkennen van dit concern in andere applicaties en het een indicatie geeft voor het implementeren ervan in AspectJ.

> ***Definities:***
>
> **Aspect Mining,** Aspect Mining is het zoeken naar en in kaart brengen van *crosscutting concerns* binnen software. Dit kan zowel *generative* (bijvoorbeeld met een *fan-in* analyse) als op basis van *queries* (waarbij gericht gezocht wordt naar een patroon). *[Marin,2005] [Marin,2006]*

4.2.3 Aspect Refactoring

Wanneer van een concern is vastgesteld dat hij *crosscutting* is, ontstaat de mogelijkheid om naast het documenteren hiervan (Aspect Mining) verbetering aan te brengen. Een methode hiervoor is aspect refactoring.

Aspect refactoring is het aanpassen van een systeem waarbij de *crosscutting* concerns met aspect- georiënteerd programmeren omgezet worden in goed gescheiden concerns. De interne structuur van de code wordt aangepast, maar met behoud van het externe gedrag. *[Laddad,2003b] [Fowler,2007] [Marin,2005b]*
Men zou kunnen zeggen dat het *intent* van het concern, het doel dat het concern heeft, gelijk blijft, maar dat het *extent*, de wijze waarop dit geïmplementeerd wordt, verbeterd wordt.

Aspect refactoring maakt het mogelijk de volgende voordelen te realiseren*[Laddad,2003]*:

- Centraal onderhoud mogelijk op het concern.
- Consistente uitvoering van het concern
- Betere scheiding van de concerns
- Minder code duplicaten

Een extra voordeel van refactoring is dat code-intentie ook beter zichtbaar wordt. Grote blokken code waarvan nu de intenties onduidelijk zijn, worden opgesplitst in kleinere methoden met een naam die de intentie duidelijk kan maken. Dit kan deels ook met OO-refactoring, maar het is wel een bijkomend voordeel.

Binnen deze scriptie zal AspectJ, een uitbreiding op Java voor aspect georiënteerd programmeren, ingezet worden voor het toepassen van *aspect refactoring*. Binnen deze taal lijkt het ontwikkelen van aspecten op dat van klassen. Een *aspect* bestaat uit een beschrijving van de locaties in de broncode waar het *aspect* een rol speelt en een constructie waarmee beschreven wordt welke functionaliteit op deze locaties uitgevoerd dient te worden. De locaties worden in AspectJ *joinpoints* genoemd, een verzameling van *joinpoints* een *pointcut* en de koppeling van

functionaliteit aan de locaties een *advice*. In *6.2 Gerealiseerde oplossingen* zijn voorbeelden van *joinpoints, pointcuts* en *advices* te zien.

Definities:

Aspect Refactoring (Aspect oriented refactoring), het aanpassen van een systeem waarbij de *crosscutting concerns* met aspectgeoriënteerd programmeren omgezet worden in goed gescheiden concerns. De interne structuur van de code wordt aangepast, maar met behoud van het externe gedrag. *[Laddad,2003b] [Fowler,2007][Marin,2005b]*

AspectJ, een aspectgeoriënteerde programmeertaal (als uitbreiding op Java). *[Laddad,2003]*

Joinpoint, een locatie in de broncode, binnen AspectJ kan dit op klasse en methode niveau. *[Laddad,2003b]*

Pointcut, één of meerdere joinpoints vormen samen een pointcut. *[Laddad,2003b]*

Advice, Een *advice* bestaat uit de combinatie van het wat (*action*) waar (*pointcut*) en hoe (*around/before/after*) van een concern in AspectJ. *[Laddad,2003b]*

4.2.4 Beschikbare pointcut mechanismen

Bij AOP zijn er verschillende mechanismen voor het realiseren van een *pointcut*. Deze verschillende mechanisme worden binnen deze scriptie ook AspectJ-stijlen genoemd. De volgende methoden voor het beschrijven zijn beschikbaar:

- ***enumeration***, hierbij wordt een lijst van elementen van de volgende structuur gebruikt: <package>.<klasse>.<methodenaam>.<parameters> voor het beschrijven van de bindingslocaties
- ***pattern matching,*** hierbij wordt een pattern zoals bijv *com.logicacmg.util.** (..) gebruikt
- ***attributes***, hierbij worden bijvoorbeeld annotaties in de code toegevoegd waarmee de bindingslocaties worden aangegeven.

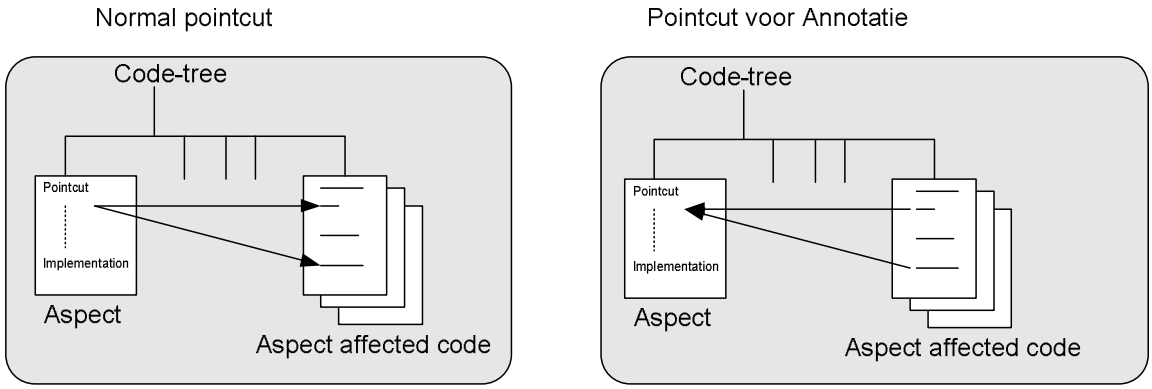
[Kiczales,2005]
Binnen dit onderzoek worden alleen de *enumeration* en *attribute pointcut mechanismen* toegepast. Ten eerste omdat op basis van *[Kiczales,2005]* deze twee mechanismen het beste passen bij dit onderzoek. We richten ons niet op *obliviousness* en dat is vooral het punt waarop *pattern matching* goed tot zijn recht komt.
Ten tweede is het opstellen van *enumeration* en *attribute* pointcuts eenvoudiger, omdat exact aangegeven wordt welke *joinpoints* in een *pointcut* behoren. Bij een *pattern* is dit niet en kan als gevolg van een verkeerd *pattern* slechts een deel van de *joinpoints* in het *pointcut* terecht komen, wat onvoorspelbaar gedrag lijkt op te leveren.

Definities:

AspectJ-stijl, een AspectJ oplossing kan gebruik maken van verschillende *pointcut* mechanismen, deze pointcut mechanismen bepalen de stijl waarin de AspectJ-code (*pointcut*) geschreven is.

4.2.5 Enumeration-pointcuts versus attribute-pointcuts

Deze *pointcut* mechanismen verschillen met name op het gebied van lokaliteit en expliciteit. Het grote verschil tussen beiden is de afstand tussen het aspect en de code waarop hij van toepassing is. De afstand tussen *pointcut* en implementatie/*advice* is in beide gevallen klein, maar de afstand tussen *advice*(aspect) en de code waarop het van toepassing is, verschilt. Bij *enumeration pointcuts* is in de code geen relatie met een aspect te zien, bij annotaties is in de code wel degelijk expliciet een relatie zichtbaar. Hierdoor zou men kunnen zeggen dat de relatie de andere kant op gericht is. Onderstaande figuur (*Figuur 6*) toont dit.



Figuur 6: Enumeration-pointcuts versus attribute-pointcuts

4.3 Samenvatting

Binnen dit hoofdstuk is duidelijk uiteengezet wat we verstaan onder sequentieel, parallel en concurrent gedrag. Het grootste verschil is hierbij dat er een extra dimensie aan de programmatoestand wordt toegevoegd. Bij sequentieel worden de statements alleen over de tijd verspreid, bij parallel ook over verschillende plaatsen(bijvoorbeeld CPU-cores). Er is dus een extra *plaats-dimensie.*

Een ander groot verschil tussen sequentiële en parallelle verwerking is het non-determinisme. Als gevolg van *scheduling* en *interleaving,* kan de volgorde waarin statements (tussen *threads*) uitgevoerd worden per executie van de applicatie verschillen. Hierdoor kan een bug soms wel optreden en soms niet, bij dezelfde data-invoer en systeemomgeving.

Binnen het hoofdstuk is ook aangegeven dat voor vrijwel iedere applicatie wel een reden te vinden is om concurrency toe te voegen. Maar ook dat dit de nodige risico's op bugs met zich meebrengt. Het is aan de ontwikkelaar om de afweging hiertussen te maken; of hij kiest voor bijvoorbeeld betere performance met als *trade-off* de kans op *race-conditions* of *deadlocks*.
De beschreven kenmerken van parallellisme zoals de rol van communicatie en synchronisatie en de hulpmiddelen voor het implementeren, dienen als indicatie voor mogelijke concurrency concerns die we in het volgende hoofdstuk kunnen onderzoeken.

Daarnaast is aangegeven wat concern classificatie is en welk model voor classificatie binnen deze scriptie zal worden toegepast. En de technieken *aspect mining* en *aspect refactoring* zijn behandeld. Hiermee kan gezocht worden naar *crosscutting* concerns en kunnen deze beter geïmplementeerd worden. Tot slot is nog even ingegaan op de mogelijkheden die AspectJ biedt om aan te geven op welke plaatsen een *crosscutting* concern geïmplementeerd moet worden, de *pointcut mechanismen/AspectJ-stijlen*.

Hiermee zijn de eerste twee onderzoeksvragen beantwoord en kunnen we de kennis toepassen bij het uitvoeren van *aspect mining* en *refactoring* van *crosscutting concurrency concerns*.

5 Aspect mining van concurrency concerns

Dit hoofdstuk gaat in op de al eerder beschreven principes die de complexiteit van software beïnvloeden: *modularity* en *encapsulation*. Om de mate vast te stellen waarin een applicatie deze principes toepast voor concurrency aangelegenheden, zal Aspect Mining gebruikt worden. Op deze manier wordt in kaart gebracht hoe goed de scheiding is tussen de concerns

Binnen dit hoofdstuk zal een antwoord gezocht worden op onderzoeksvraag 3 (*zie H3 Onderzoeksmethode*). Hiervoor zal de volgende hypothese nader onderzocht worden:

"Binnen bestaande objectgeoriënteerde multi-threaded software is sprake van slechte modularity en encapsulation (crosscutting) bij de implementatie van concurrency gerelateerde concerns."

Om deze hypothese te onderzoeken zal eerst een applicatie geselecteerd worden *(zie H5.1)*. Vervolgens worden hiervan de concurrency concerns vastgesteld (*zie H5.2*). Enkele van deze concerns zullen vervolgens onderzocht worden op de mate waarin deze *crosscutting* zijn *(zie H5.3 en H5.4)*.
Vervolgens zal nog bepaald worden in welke categorie van het model uit *[Marin,2005]* deze concerns vallen, om hiermee ten eerste op een heldere wijze te kunnen communiceren over de concerneigenschappen en uiteraard om alvast een richting in te slaan voor het vinden van mogelijke verbeteringen in de AOP-wereld *(zie H5.5 en H5.6)*.
Hierna volgt nog een conclusie *(zie H5.7)* over de hypothese is er nog een sectie (*zie H5.8)* waarin gereflecteerd wordt op de benadering van het probleem uit dit hoofdstuk.

De volgende stappen vormen de aanpak van dit deel van het onderzoek:

- Stap 0: Selecteren van een te onderzoeken applicatie *(Uitgewerkt in H5.1)*
- Stap 1: Identificatie en classificatie van *concurrency concerns* *(Uitgewerkt in H5.2)*
- Stap 2: In kaart brengen *tangling* van de concerns *(Uitgewerkt in H5.3)*
- Stap 3: In kaart brengen *scattering* van de concerns *(Uitgewerkt in H5.4)*
- Stap 4: Controle classificatie van de concerns *(Uitgewerkt in H5.5 & H5.6)*
- Stap 5: Conclusie *(Uitgewerkt in H5.7)*

5.1 Applicatieselectie

Binnen sectie *H3.2 Toelichting onderzoeksvragen* kwam naar voren welke criteria gebruikt zouden worden bij het selecteren van een te onderzoeken systeem. De keuze is gevallen op Azureus (versie 3.0.1.2) de volgende alinea zal ingaan op deze applicatie.

Wat is Azureus?

Azureus is een zeer populaire open-source Java-applicatie (zie de Sourceforge downloadstatistieken) voor het *peer-2-peer* uitwisselen van bestanden via het BitTorrent (http://www.bittorrent.com/what-is-bittorrent) netwerk. De onderstaande tabel toont enkele statistieken over de applicatie. Hierin is te zien dat het om een grote Multi-threaded applicatie gaat.

Kenmerk	Waarde
Aantal packages	~ 420
Aantal klassen	~ 1.737
Aantal regels code	~ 300.000
Aantal threads na het opstarten	~ 80

Tabel 1: Kenmerken Azureus

Hoe voldoet Azureus aan de gestelde criteria?

Hieronder wordt beschreven hoe de applicatie aan de eerder gestelde criteria (*sectie 3.2 Toelichting onderzoeksvragen*) voldoet.

- ***De applicatie is geschreven in Java en beschikt over multi-threading***

Zoals eerder al aangegeven is Azureus een Java applicatie met veel multi-threading. Zie bovenstaande tabel (*Tabel 1*).

- ***De applicatie beschikt over geëvolueerde introductie van concurrency***

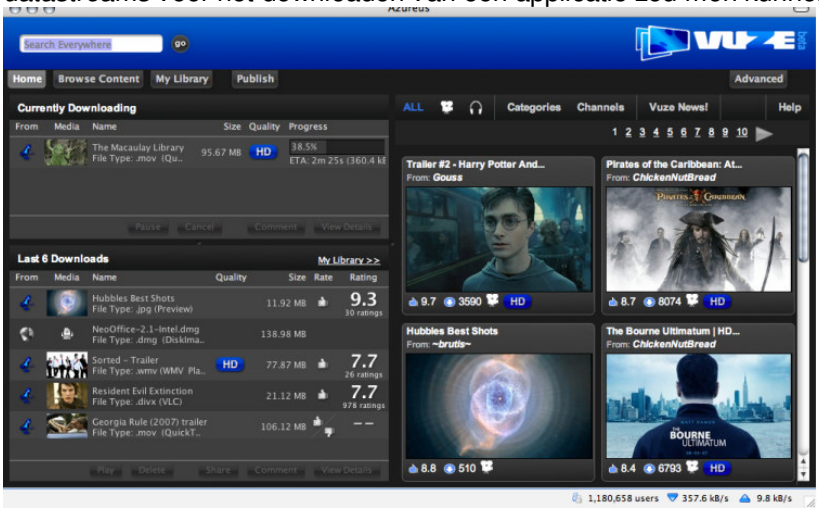
Bij de selectie heeft ook overleg plaatsgevonden met de *lead-developer* die zich met name richt op het in goede banen leiden van de ontwikkeling van het concurrency aspect van de applicatie. Hierbij kwam naar voren dat bij de ontwikkeling van de applicatie in het begin nauwelijks aandacht besteed werd aan concurrency en het pas later in het systeem is opgenomen. Met als gevolg dat er allerlei problemen en bugs optraden, o.a. omdat men het overzicht verloor van de implementatie van concurrency.

- ***Broncode van de applicatie is vrij toegankelijk***

De applicatie is te downloaden op SourceForge (http://azureus.sourceforge.net/). Hier wordt ook de broncode beschikbaar gesteld.

- ***De applicatie heeft baat bij multi-threading***

De hoofdmotivatie voor het gebruik van concurrency binnen Azureus is: *responsiveness*. De andere motivaties spelen ook een rol, maar de hoofdreden is het scheiden van: *GUI*, binnenkomende connectieaanvragen, binnenkomende datapakketten en validatie van pakketten. Daarnaast is er zowel sprake van *task- parallellisme als data- parallellisme. Task- parallellisme* om deze zaken gescheiden parallel uit te voeren. Maar bijvoorbeeld het verkrijgen van meerdere *datastreams* voor het downloaden van één applicatie zou men kunnen zien als *data-parallellisme*.



Figuur 7: Screenshot van Azureus 3 (Vuze)

5.2 Identificatie en classificatie van concurrency concerns

Binnen deze paragraaf wordt uiteengezet welke concurrency concerns binnen Azureus aangetroffen kunnen worden. Bij de identificatie speelde o.a. de informatie verkregen uit gesprekken met de ontwikkelaars van Azureus een rol.
Vervolgens worden deze concerns geclassificeerd volgens het model opgesteld in *[Marin,2005]*. Het relevante deel van dit model is al deels beschreven in sectie *4.2.1 Concern Classificatie*.

5.2.1 Identificatie concurrency concerns

Hieronder staat een beschrijving van de concurrency concerns die binnen de geselecteerde applicatie, Azureus een rol spelen. Zij zijn kort beschreven op hun *intent* (wat is hun doel) en hun *extent* (hoe ziet de implementatie er uit). Binnen dit onderzoek zijn niet alle concerns uit dit model onderzocht op de mate van *crosscutting*. De reden is dat dit te veel tijd zou gaan kosten. De selectie van de te onderzoeken concerns is op basis van de grote van hun rol binnen de applicatie. De grote van de rol is bepaald aan de hand van een gesprek met één van de hoofdontwikkelaars van Azureus.

In paragraaf *5.3 Tangling* zijn codefragmenten (het *extent*) te zien voor ieder van de concerns die hieronder beschreven zijn.

- Concurrency control constructs

Om de *shared state* van de applicatie consistent te houden zijn *concurrency control constructs* nodig (zoals beschreven in *H4.1.7*). Binnen Azureus worden de volgende twee klassen gebruikt voor concurrency control: *AEMonitor* en *AESemaphore*. Daarnaast wordt ook het taalelement *synchronized* gebruikt. Echter *synchronized* zal niet worden opgenomen in de onderzoeksresultaten. Dit omdat het onderzoek zich richt op de mogelijkheden van objectoriëntatie en *synchronized* is geen klasse of object, al zou men het wel als speciale variant van een methode-aanroep kunnen zien. Wel zal het gebruik en de *scattering* van *synchronized* gemeten worden om te bepalen hoe groot de rol is van het *concurrency control concern*.

- Asynchrone methoden

Binnen Azureus is nog een ander concern dat wijdverspreid in de applicatie terugkomt: parallelle uitvoering van methoden. Het gaat hierbij om het uitvoeren van een methode (of deel ervan) in een nieuwe thread, waarbij niet gewacht wordt op het resultaat. Het gaat dus om asynchrone (parallelle) methoden, welke soms éénmaal en soms als *daemon* worden gestart.
Daarnaast zien we terug in welke *threadpool* een asynchrone methode thuishoort, op basis van zijn prioriteit. Het nadeel van de implementatie is dat er geen centraal overzicht is van de pools en bijbehorende threads en prioriteiten.

- User interface updating (in eigen thread)

Een aanverwant concern aan *asynchrone methoden* is het updaten van de *user interface*. Het updaten van de user interface moet plaatsvinden binnen de thread die eigenaar is van de user interface. Binnen de code van Azureus zien we dit terug doordat op verschillende plaatsen code naar de thread van de *GUI*-eigenaar gestuurd wordt. Dit lijkt erg op een asynchrone methode waarbij ook code verplaatst wordt naar een andere thread, maar dan naar een nieuwe. Het kan dus als specialisatie ervan gezien worden.

Welke concerns zijn er nog meer

Naast de al eerder genoemde concerns zijn er nog andere concerns te vinden, zoals te zien in het concern model zie *Bijlage F: Concurrency model van de onderzochte applicatie*. Deze richten zich bijvoorbeeld op synchronisatie tussen threads. Zoals het gebruik van *Thread.sleep,* dit maakt deel uit van een concern voor het synchroniseren tussen threads op basis van tijd.

5.2.2 Classificatie van de gevonden concerns in sorts

Hieronder is beschreven tot welk concerntype (*sort*) de *crosscutting concurrency concerns* behoren.

Concurrency control constructs als vorm van consistent behavior

Het gebruik van *concurrency control constructs* zal voornamelijk plaatsvinden volgens een vastgesteld patroon. Het principe van een monitor*(zie 9.2.1)* is dat het een constructie is waarbij toegang tot een *critical region* afgeschermd wordt op basis van een conditie of object. Hierbij is het in principe zo dat vooraf aan de sectie een monitor wordt betreden, en achteraf wordt

verlaten. Het patroon zou dus dit consistente gedrag moeten vertonen om een monitor genoemd te kunnen worden.

User interface updating als vorm van add variability
Het user interface updaten vindt plaats middels een aanroep van de methode *ExecSWTThread*. De eerste parameter van deze methode verwacht een instantie van het type *Runnable*. Omdat het nogal omslachtig is om voor iedere *user-interface update* een klasse aan te maken die *Runnable* implementeert, is het aannemelijk dat hier gebruik gemaakt wordt van methode-objecten en dus dat er regelmatig sprake is van het *add variability sort*.

Asynchrone methoden als vorm van add variability
Het asynchroon uitvoeren van methoden vindt plaats met behulp van een *AEThread*-instantie. Hierbij is niet sprake van het doorgeven van een methode als argument, maar wel van methode objecten. Er is dus wel sprake van *add variability*, in plaats van het doorgeven als argument wordt de methode via een overervingrelatie beschikbaar gesteld aan een ander object. Vaak via anonieme objecten en dus is het een vorm van *add variability.*

Asynchrone methoden als vorm van dynamic behavior enforcement
Bij de asynchrone methoden is een patroon voor het aanroepen van methoden terug te vinden welke in de categorie valt van initialisatie. De eerste is het aanmaken van een *Runnable*-instantie (al dan niet als instantie van een anonieme klasse), het tweede is het instellen van het *runnable*-object als *daemon* (of juist niet), het derde is het instellen van de prioriteit van het *runnable*-object, het vierde is het starten. Hierbij is niet altijd iedere methode-aanroep nodig, soms worden de default-waarden toegepast en de methode aanroep overgeslagen.

5.3 Tangling

Binnen deze paragraaf wordt beschreven in welke mate er sprake is van *tangling* van de concerns beschreven in *H5.2 Identificatie concurrency concerns.* De *tangling* bepaalt samen met de *scattering* of er sprake is van een *crosscutting concern*.
Hieronder is voor ieder van de concerns een fragment uit de code van de applicatie gekopieerd om aan te geven hoe deze verweven zijn.
Om de mate van *tangling* van de concerns aan te geven zou ook in kaart gebracht moeten worden hoe vaak en consistent dit *tangling*-patroon voorkomt. Dit zal impliciet gedaan worden bij het controleren van de classificatie van de concerns.
De vorm van verwevenheid is dus binnen dit hoofdstuk aangekaart, de consistentie ervan in *H5.5* en *H5.6.*

- Concurrency control constructs

In onderstaand voorbeeld (*Figuur 8*) zien we twee concerns met elkaar verweven, aan de ene kant het *concurrency control concern*, aan de andere kant het concern voor het afhandelen van nieuwe connecties.

```
public void addPeerConnection( NetworkConnectionBase connection ) {
  try {  connections_mon.enter();
    //copy-on-write
    ArrayList conn_new = new ArrayList( connections.size() + 1 );
    conn_new.addAll( connections_cow );
    conn_new.add( connection );
    connections_cow = conn_new;                    Ander concern
  }
  finally{ connections_mon.exit();  }              Concurrency control concern
}
```

Figuur 8: Voorbeeld van verweven 'concurrency control' concern

- Asynchrone methoden

In *Figuur 9* is te zien hoe de code voor het concern 'asynchrone methoden' verweven is met het concern van *file-IO*. Waarbij het concern *file-IO* verpakt zit in één methode, meestal is de inhoud *inline*, alleen dan zou het voorbeeld wat onoverzichtelijk worden.

```
//start read handler processing
Thread read_processor_thread = new AEThread( "ReadController" ){
  public void runSupport() {
    readProcessorLoop();                                  Ander concern
  }
};
read_processor_thread.setDaemon( true );
read_processor_thread.setPriority( Thread.MAX_PRIORITY - 1 );
read_processor_thread.start();                        Asynchrone methoden
```

Figuur 9: Voorbeeld van verweven 'asynchrone methoden' concern

- User interface updating (in aparte thread)

In *Figuur 10* is te zien hoe het concern van de weergave van de webbrowser verweven is met het concern voor het verwerken van gebruiksinterface wijzigingen in een apart thread(*SWTTread*). Het gebruik van *anonymous classes* maakt het geheel nog onoverzichtelijker.

```
if (browser != null && !browser.isDisposed()) {
    Utils.execSWTThread(new AERunnable() {
        public void runSupport() {
            if (browser != null && !browser.isDisposed()
                    && !browser.isVisible()) {
                browser.setVisible(true);
            }                                             Ander concern
        }
    });                                                   'UI updating' concern
}
```

Figuur 10: Voorbeeld van verweven 'UI updating' concern

5.4 Scattering

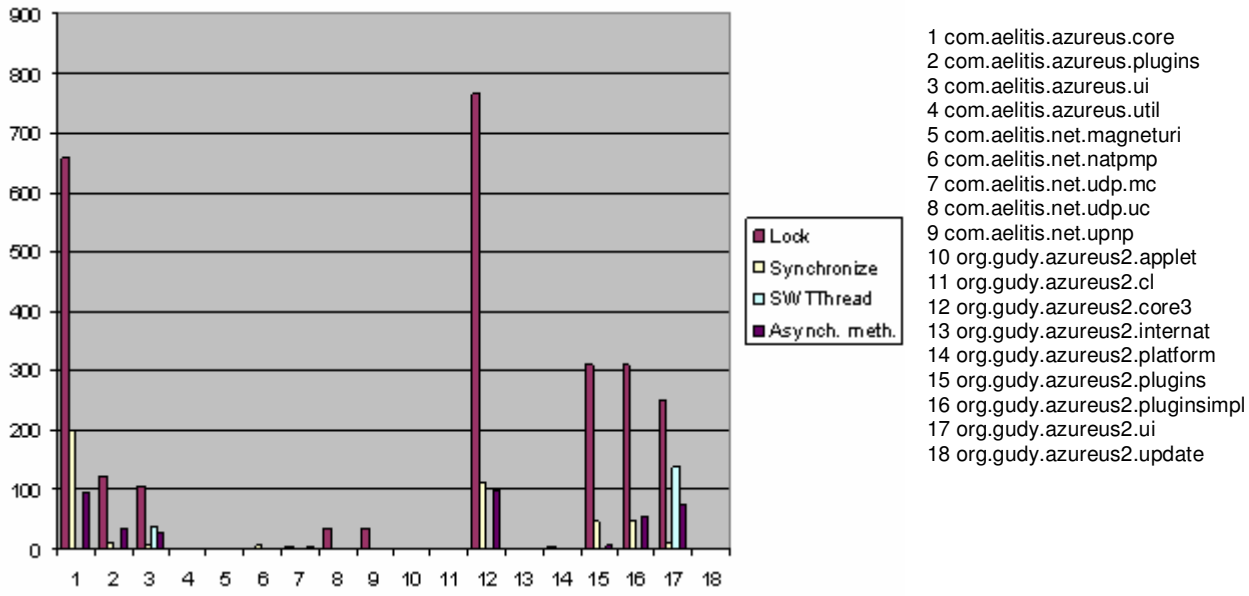
Deze paragraaf behandelt de verspreiding van de concerns over de code van het systeem. De tabel hieronder(*Tabel 2*) toont statistieken over het gebruik van de concerns. De *scattering* is vooral afhankelijk van het aantal aanroepen en het aantal *packages* waaruit deze aanroepen plaatsvinden.
Echter een groot aantal 'aanroepen' wil niet zeggen dat het concern ook daadwerkelijk over het gehele systeem verspreid is, het zegt alleen iets over de omvang. Om de spreiding van de aanroepen over het systeem te tonen is van elk van de aanroepen bepaald in welke *package* deze plaatsvond.

Tabel 2 toont per concern welke methoden betrokken zijn en hoe vaak ze worden aangeroepen en vanuit hoeveel packages. In *Figuur 11* is de spreiding te zien, hierbij is per package-groep het aantal aanroepen te zien (packages zijn gegroepeerd per 3 niveaus *nesting*). Het is niet verassend dat de twee grootste uitschieters hierbij tot de grootste *packages* van het systeem behoren.

Tabel 2: Concern statistieken uit Azureus

Concern	Methode naam	# Aanroepen	# Betrokken packages
Concurrency control	- AEMonitor.Enter	1043	122
	- AEMonitor.Exit	1042	122
	- AESemaphore.Reserve	74	47
	- AESemaphore.Release	139	47
	- Synchronized *)	404	56
Asynchrone methoden	- AEThread.start	158	101
	- AEThread.setDeamon	70	37
	- AEThread.setPriority	20	12
User interface updating	- Utils.execSWTThread	163	28

*) niet onderdeel van het concern maar wel betrokken



Figuur 11: Spreiding van de concerns in Azureus

Om een weergave met nog meer precisie te realiseren is gebruik gemaakt van de *Aspect Visualizer* van Eclipse om het gebruik in kaart te brengen. Deze toont per *class* of *package* een balk waarvan de grote overeenkomt met de grote van de module (*class*/*package*). Op deze balk is met blauw aangegeven wanneer er een referentie is naar de AEMonitor. Een volledige plattegrond van het hele systeem (op *package* niveau) is te vinden in de *Bijlage G: Lock scattering* . Hierin is duidelijk te zien dat over het hele systeem aanroepen terug te vinden zijn.

5.5 Metingen

Binnen deze paragraaf wordt de tangling en classificatie van de concerns gecontroleerd. Hiervoor wordt gemeten hoe vaak de methoden van het concern gebruikt worden in de vorm van het *sort* wat er aan toegekend is in *H5.2.*

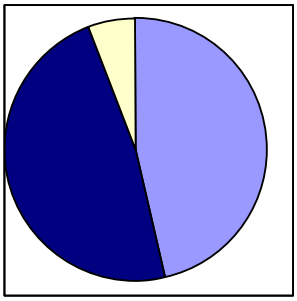
5.5.1 Concurrency control concern als consistent behavior

AEMonitor

Er zullen 100 methoden, waarin *AEMonitor.enter()* en *AEMonitor.exit()* gebruikt zijn, onderzocht worden op dit patroon. Hierbij worden een aantal willekeurige klassen geselecteerd waarvan de methoden onderzocht worden en geregistreerd wordt of ze aan het patroon voldoen. Maar ook

wordt bijgehouden of de afwijking groot (+4 statements) is. Een kleine afwijking betekent vaak dat met een kleine refactoring een 'pure' implementatie van het *sort* gemaakt kan worden. Daarom kunnen instanties met een kleine afwijking ook meegeteld worden als consistent behavior.

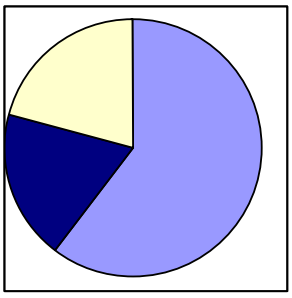
AEMonitor resultaten
Onderstaande tabel toont de resultaten van het onderzoek naar consistent behavior.

Patroon overeenkomst	Percentage	Weergave
Puur/Exact (Paars)	53 %	
Met minimale afwijking (blauw)	42 %	
Rest (geel)	5 %	

AESemaphore

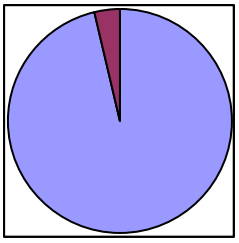
Alle methoden waarin *AESemaphore.reserve()* en *AESemaphore.release()* gebruikt zijn, worden onderzocht op het patroon. Hierbij wordt geregistreerd of ze exact aan het patroon voldoen. Wanneer er geen exacte match is, wordt bepaald of de afwijking groot (+2 statements) is.

AESemaphore resultaten
Onderstaande tabel toont de resultaten van het onderzoek naar consistent behavior.

Patroon overeenkomst	Percentage	Weergave
Puur/Exact (Paars)	61 %	
Met minimale afwijking (blauw)	19 %	
Rest (geel)	20 %	

5.5.2 UI updaten als add variability sort

Binnen deze paragraaf wordt onderzocht in welke mate het gebruik van *execSWTThread* plaatsvindt als onderdeel van een *add variability sort*. Onderstaande tabel toont van de 163 onderzochte aanroepen van exec*SWTThread* hoe vaak er sprake is van *add variability*

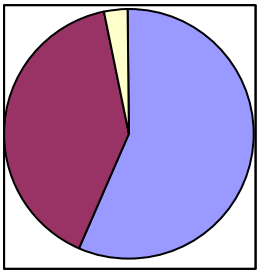
Patroon overeenkomst	Percentage	Weergave
Ja	96 %	
Nee	4 %	

5.5.3 Asynchrone methoden als combinatie van add variability en dynamic behavior sort

Binnen deze paragraaf wordt onderzocht in welke mate het gebruik van *AEThread* / *AERunnable* een vorm is van *add variability* en *dynamic behavior enforcement.*

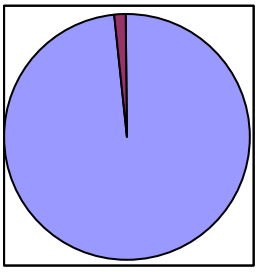
Onderstaande tabel toont de metingen voor het *dynamic behavior enforcement* gedeelte.

- volledig patroon = *constructor* + *setDeamon* +*setPriority* + *start*
- onvolledig patroon = *constructor* + *start* (evt. *setDeamon of setPriority*)
- geen patroon = alleen *constructor*

Dynamic behavior enforcement	Percentage	Weergave
volledig patroon	56%	
onvolledig patroon	41%	
geen patroon	3%	

Onderstaande tabel toont de metingen voor het *add variability* gedeelte.

- Ja = anonieme klasse voor *thread* of *runnable* met daarin code voor bijwerken gebruikersinterface
- Nee = geen anonieme klasse voor *thread* of *runnable*

Add variability	Percentage	Weergave
ja	99%	
nee	2%	

5.6 Evaluatie

5.6.1 Evaluatie concurrency control concern als consistent behavior

AEMonitor
Van alle onderzochte methoden bevat net iets meer dan de helft het exacte patroon (53%). Daarnaast is er 42% waarbij het patroon met een minimale afwijking geïmplementeerd is. Slechts bij 5% was de afwijking groter dan minimaal. Bij 95% is dus dit patroon terug te vinden, er is dus sprake van *consistent behavior*.

Oorzaken

Hieronder volgt een opsomming van de waargenomen oorzaken voor het niet exact voldoen aan het consistent behavior patroon.

- **Initialisatie / exit**
  Bijvoorbeeld voor het initialiseren van variabelen voor het *try*-blok of het opruimen van resources na afloop van het *finaly*-blok en het return statement bij niet *void*-methoden.

- **Timing**
  Bijvoorbeeld door het gebruik van timers/sleep voor synchronisatie.

- **Overig concurrency control**
  Bijvoorbeeld door genest monitor gebruik.

- **Bugs**
  Bijvoorbeeld het gebruik van de monitor zonder het *try/finally* blok wat mogelijk bugs zoals *deadlocks* op kan leveren.

AESemaphore
Van de onderzochte methoden bevat net meer dan de helft het exacte patroon (61%), 20% van de methoden bevat een afwijking van meer dan 4 statements. Het patroon is dus maar in 80% van de gevallen waargenomen, waarbij de oorzaak er bovendien vaak op duidde dat het patroon niet correct was.
Zo kwam het ook vaak voor dat juist aan het begin van een methode *release()* en aan het eind *reserve()*. Voor het creëren van synchronisatie wachten van andere threads.

Er is dus wel sprake van *crosscutting,* maar niet van *consistent behavior*, althans niet volgens het patroon dat onderzocht is.

5.6.2 Evaluatie user interface updating als add variability sort

Van alle onderzochte methoden wordt in 96% van de gevallen de methode *execSWTThread* alleen gebruikt om de user interface te updaten. In de overige 4% blijkt dat naast de user interface ook nog andere functionaliteit wordt uitgevoerd. Hieronder is opgesomd wat dan wel de reden was om *execSWTThread* te gebruiken
Er is dus consistent sprake van *add variability* voor deze concern implementatie.

Oorzaken afwijkingen

- **Clipboard toegang**
  Soms verliep ook toegang tot het clipboard via de SWT thread.

- **File Access**
  In één geval werd ook file-access geregeld via de SWT Thread.

5.6.3 Evaluatie asynchrone methoden als combinatie van add variability en dynamic behavior sort

Dynamic behavior
Van alle onderzochte methoden bevat net iets meer dan de helft het exacte patroon (56%). Daarnaast is er 41% waarbij een deel van het patroon geïmplementeerd is. Slechts bij drie procent werd er geen patroon waargenomen.
Er is dus consistent sprake van *dynamic behavior* voor deze concern implementatie.

Oorzaken afwijking dynamic behavior sort:

- **Gebruik standaard prioriteit**
  In het opgestelde patroon zit een stap voor het instellen van de prioriteit. Echter soms wordt deze stap overgeslagen, waardoor er een afwijking is op het patroon.

- **Alternatieve manier voor thread starten**
  In enkele gevallen werd het starten van de thread gedaan via een *speciale* hulpklasse.

Add variability
Vrijwel alle onderzochte instanties van asynchrone methoden bleken gebruik te maken van methode-objecten in de vorm van *add variability.* Slechts bij twee procent werd er geen patroon waargenomen.
Er is dus consistent sprake van *add variability* voor deze concern implementatie.

Oorzaken afwijking add variability:

- **Gebruik 'echte klasse'**
  Een enkele keer was er sprake van het gebruik van een 'volledige' klasse in plaats van een anonieme klasse.

5.7 Conclusie

Binnen dit hoofdstuk is in kaart gebracht hoe goed de principes van *modularity* en *encapsulation* van toepassing zijn op de applicatie Azureus. Hierbij zijn de volgende *crosscutting* concerns geïdentificeerd en geclassificeerd.

Tabel 3: Geïdentificeerde en geclassificeerde crosscutting concurrency concerns

Crosscutting concern	Type/Sort
Concurrency control	*consistent behavior*
Asynchrone methoden	*add variability* en *dynamic behavior*
GUI Updating	*add variability*

Deze concerns spelen alle drie een grote rol binnen de onderzochte applicatie.

Samengevat:

- De applicatie beschikt over meerdere concurrency concerns (zie Tabel 3)
- De implementatie van de onderzochte concurrency concerns bleek *crosscutting* te zijn.
- De meeste concurrency concerns konden geclassificeerd worden (met het model beschreven in *[Marin,2005])*
- De onderzochte concurrency concerns zijn classificeerbare *crosscutting concerns* en beschikken derhalve over slechte *modularity* en *encapsulation*, welke mogelijk verbeterd zou kunnen worden.

Op basis van deze resultaten kan een onderzoek gestart worden naar de mogelijkheden voor verbetering van de *modularity* en *encapsulation* van deze concerns. In het volgende hoofdstuk zal hier verder op worden ingegaan.

5.8 Reflectie

Er is een aantal aspecten dat de validiteit van het onderzoek in gevaar kan brengen. De belangrijkste zijn hieronder opgesomd en behandeld. Hierbij is aangegeven wat gedaan is om de risico's voor de validiteit van het onderzoek te beperken.

5.8.1 False positives / negatives

Binnen dit onderzoek zijn verschillende tools ingezet om de concerns te documenteren en classificeren. Hierbij is het mogelijk dat bij de controle van de classificatie zich *false positives* of *false negatives* voordoen. Echter het gaat hier om eenvoudige patronen die duidelijk te herkennen zijn, waardoor de kans op *false positives* en *negatives* klein is. *False positives* en *negatives* treden vaak op wanneer de analyse complex is, wat hier niet het geval is.

5.8.2 Steekproef

Bij dit onderzoek is een steekproeftelling gedaan voor het onderzoeken van *consistent behavior* van *concurrency control*. Dit omdat het onderzoeken van alle code-eenheden te omvangrijk was. Belangrijke eigenschappen van een steekproef is dat deze representatief en betrouwbaar is[1]. De populatie bij de steekproeven is de verzameling klassen uit Azureus waarin methoden met het onderzochte patroon voorkomt. De elementen zijn uiteraard de klassen met deze methoden.

Representatief,
betekent dat de elementen uit de steekproef een afspiegeling vormen van de gehele populatie waar ze toe behoren. Wanneer dit het geval is, is het onderzoek generaliseerbaar naar de populatie. Hierbij is het vooral van belang dat de elementen uit de populatie aselect gekozen worden. Ieder element uit de populatie heeft een gelijke kans om in de steekproef opgenomen te worden.
In dit geval is dit gerealiseerd door willekeurig klassen te selecteren (uit de klassen die populatie vormen/ het concern bevatten).De elementen worden dus willekeurig uit de populatie genomen(aselect). Hierbij zijn uiteindelijk uit elke ('level 3') package klassen geselecteerd. De klassen zijn dus van verschillende auteurs en systeemonderdelen.

Betrouwbaarheid,
betekent dat wanneer de steekproef herhaald wordt de resultaten globaal hetzelfde zijn. Hier spelen onder andere de methode van waarneming, de variabiliteit uit de populatie en de omvang van de steekproef een rol. Zoals al eerder aangegeven is de methode van waarneming niet destructief, de steekproef verandert de populatie niet. Hierdoor is het mogelijk om de elementen uit de test herhaald te controleren, waar dan hetzelfde resultaat uit zou moeten komen.
Daarnaast is de variabiliteit laag, bij de steekproef bleek dat de verhouding 'match' en 'geen match', 95% om 5% is.

[1] Dit is gebaseerd op informatie uit *Kennisbasis Statistiek,* http://www.wynneconsult.com

6 Aspect refactoring van concurrency control constructs

In het vorige hoofdstuk is in kaart gebracht welke concurrency concerns aanwezig zijn binnen de onderzochte applicatie. Hierbij kwam naar voren dat bij deze concerns sprake was van *crosscutting* en dus een slechte *modularity* en *encapsulation*. Dit hoofdstuk bouwt hierop voort door met enkele experimenten te onderzoeken hoe dit verbeterd kan worden.

Binnen dit hoofdstuk zal een antwoord gezocht worden op onderzoeksvraag 4 (*zie H3 Onderzoeksmethode*). Hiervoor zal de volgende hypothese nader onderzocht worden:

"Met behulp van AspectJ en aspect refactoring is het mogelijk de concurrency concerns met betere modularity en encapsulation te implementeren"

Om deze hypothese te testen zal een aantal oplossingen met AspectJ ontwikkeld worden (resultaten beschreven in *H6.2*). Op basis van metingen aan deze oplossingen (*H6.3*) en de originele versie van Azureus, zal onderzocht worden of er sprake is van verbetering op het gebied van *modularity* en *encapsulation*. Dit meten wordt gedaan aan de hand van software metrieken en scenario's.
Op basis van de analyse (*zie H6.4*) van deze meetresultaten zal geconcludeerd worden of er daadwerkelijk sprake is van verbetering (*zie H6.5*).

De volgende stappen vormen de aanpak van dit deel van het onderzoek.

- Stap 1: Verzamelen van locaties waar de concern implementaties zich bevinden.
- Stap 2: Opstellen testcases op basis van de huidige concern implementatie.
- Stap 3: Opstellen AspectJ implementatie van het concern. *(Uitgewerkt in H6.2)*
- Stap 4: Applicatie-refactoring in verschillende AspectJ-stijlen. *(Uitgewerkt in H6.2)*
  Voor iedere locatie waar de implementatie van het concern voorkomt:
  c. *Verwijderen objectgeoriënteerde implementatie voor de locatie*
  d. *Implementeren van concern in AspectJ voor de locatie*
- Stap 5: Meten van *modularity* en *encapsulation* van de concerns *(Uitgewerkt in H6.3)*
- Stap 6: Analyse meetresultaten *(Uitgewerkt in H6.4)*
- Stap 7: Conclusie en reflectie *(Uitgewerkt in H6.5)*

6.1 Metrieken

Al eerder was aangegeven (*H3 Onderzoeksmethode*) dat voor het in kaart brengen van de verschillen in *modularity* en *encapsulation* van de verschillende *branches* van de applicatie gebruik gemaakt zou worden van metingen op basis van drie strategieën: OO-metrieken, AO-metrieken en scenario's. Hierbij zijn alleen de relevante metrieken geselecteerd op basis van hun relatie met complexiteit, *modularity* en *encapsulation*.

6.1.1 Object- georiënteerde metrieken

Hieronder is een opsomming te zien van de OO-metrieken die binnen dit onderzoek gebruikt zullen worden, aangevuld met een korte beschrijving van het *wat* en *waarom*.

Method lines of code (LOC) / Number of Methods (NOM)	***Modularity***
De omvang van een methode in aantal regelscode en het aantal methoden	
De omvang van een methode zegt iets over de transparantie ervan, kleine methoden zijn vaak beter te begrijpen. Bovendien hebben kleine methoden vaak een betere *modularity* dan grote. Een grote methode omvat vaak subtaken die beter in een aparte methoden geplaatst hadden kunnen worden*[McConnell,2004].*	

Nested Block Depth (NBD)	***Complexiteit***
Het maximale niveau waarmee de statements binnen een methode genest zijn.	
Het niveau van *nesting* van code binnen methoden heeft een invloed op de complexiteit ervan. Hoe dieper genest hoe complexer de code. *[McConnell,2004]*	

McCabe Cyclomatic Complexity Index per method (McCabe CC)	***Complexiteit***
Deze metriek weerspiegelt de structurele complexiteit van de methode en is gebaseerd op het aantal unieke paden daarbinnen. Dit is dus onder andere afhankelijk van het aantal conditionele statements(if else) en lussen (while). [Rosenberg,1998]	
Dit is een algemeen geaccepteerde metriek voor het verkrijgen van inzicht in de complexiteit van een systeem. *[McConnell,2004]*	

Coupling Between Object Classes (CBO)	***Modularity en Encapsulation***
Het aantal andere klassen waarmee een klasse een relatie heeft. Hoe hoger de waarde, hoe hoger de koppeling. Een hoge waarde voor deze metriek is een indicatie van slechte modularity en encapsulation. [Chidamber,2004]	
Deze metriek heeft een directe band met de principes van *modularity* en *encapsulation*. *[Chidamber,2004]*	

Lack of Cohesion (LCOM)	***Encapsulation***
Hierbij wordt gemeten in hoeverre de methoden van een klasse gebruik maken van de attributen van dezelfde klasse. Een getal rond de één impliceert dat het mogelijk verstandig is de klasse op te splitsen. [Chidamber,1994]	
Hoe meer gebruik ze maken hoe beter de cohesie is tussen de methoden en de attributen. Een goede waarde voor deze metriek impliceert een goede *encapsulation.[Chidamber,1994]*	

6.1.2 Aspect- georiënteerde metrieken

Hieronder is een opsomming te zien van de geselecteerde AO-metrieken en een korte beschrijving van *wat* het is. De reden dat ze geselecteerd zijn is hun directe relatie met de *modularity* en/of *encapsulation*. De metrieken zijn afkomstig uit *[Ceccato,2004]*, maar binnen *[Tsang, 2003]* en *[Soares,2004]* worden vergelijkbare metrieken genoemd.

Lines of Code (LOC)	***Complexiteit en Modularity***
De omvang van een systeem staat in verband met de complexiteit en modularity. Een systeem met goede modularity zal vaak meer hergebruik van stukken code plaatsvinden op basis van aanroep t.o.v. kopieën. Al zijn er ook andere oorzaken te bedenken voor lagere LOC.	

Coupling on Method Call (CMC)	***Modularity en Encapsulation***
Dit is het aantal modulen waarvan methoden door een (andere) module worden aangeroepen. Dit is vergelijkbaar met CBO uit het OO-metrieken hoofdstuk. Een hoge waarde betekent een hoge koppeling.	

Coupling on Field Access (CFA)	***Modularity en Encapsulation***
Het aantal modulen dat velden declareert die door een module benaderd worden. Bij OO metrieken is dit onderdeel van CBO, bij AO-metrieken is dit opgedeeld in CFA en CMC. De reden is dat dit bij OO systemen vaak laag of bijna nul is en bij AO systemen vaak hoog omdat de aspects velden benaderen van een module.	

Coupling between Modules (CBM)	***Modularity en Encapsulation***
Dit is een combinatie van CMC en CFA en dus direct vergelijkbaar met CBO uit de OO-metrieken set. Een lage waarde betekent een lage koppeling en is dus beter.	

Coupling on Intercepted Modules (CIM)	***Encapsulation***
CIM is het aantal modulen dat letterlijk genoemd wordt in een pointcut van een aspect. Sub-modulen, modulen die interfaces implementeren of modulen welke via wildcards een match maken met een pointcut, worden niet meegeteld. CIM staat dus voor een sterke koppeling tussen een aspect en het systeem en betekent een lage herbruikbaarheid.	

Crosscutting degree of an aspect (CDA)	***Modularity en Encapsulation***
Het aantal modulen dat beïnvloed wordt door een pointcut. Een verschil met CIM is dat die alleen expliciet genoemde relaties meetelt. Het verschil tussen CIM en CDA is dus het aantal modulen dat beïnvloed wordt, zonder letterlijk als pointcut opgenomen te zijn. Hoge waarden van CDA en lage waarden van CIM zijn wenselijk.	

Coupling on Advice Execution (CAE)	***Modularity en Encapsulation***
Het aantal aspecten dat advices bevat die mogelijk betrokken zijn bij de executie van methoden in een module. Deze metriek meet de koppeling tussen een methode en het aspect (een koppeling die afwezig is bij OO systemen).	

Lack of Cohesion in Operations (LCO)	***Encapsulation***
Hierbij wordt gemeten in hoeverre de methoden van een module gebruik maken van de attributen van dezelfde module.	

CIM, CDA en CAE zijn AO specifiek, voor OO systemen leveren deze geen waarde op omdat er metingen worden gedaan op basis van pointcuts en advices.

6.1.3 Onderhoudsscenario's

De volgende scenario's zijn opgesteld om de *modularity* en *encapsulation* in kaart te brengen. Het idee van scenario's is afkomstig uit *[Kiczales,2005].* Bij het opstellen van deze scenario's is getracht om de scenario's dicht bij de realiteit te houden en ze zijn daarom afgestemd op de geselecteerde concerns. De scenario´s zijn afkomstig uit een overlegsessie met collega´s bij de opdrachtgever en op basis van informatie van het Azureus Team.

Bij de evaluatie hiervan zal een tweetal kenmerken gemeten worden, het aantal regels dat gewijzigd dient te worden, en het aantal modulen dat betrokken is bij de wijziging. Het aantal modulen en aantal regels code dat aangepast dient te worden zegt iets over de mate waarin het concern zijn implementatie verbergt (*encapsulation*) en hoe een concern verspreid is (*modularity*). Wanneer een wijziging tot gevolg heeft dat er meerdere modulen en veel regels code aangepast moeten worden, betekent dit dat de implementatie van een concern over slechte *modularity* en *encapsulation* beschikt.

Tabel 4: Onderhoudsscenario's

Concurrency control concern	
1	Toevoegen/verwijderen van een monitor aan een methode.
2	Hernoemen van een door een monitor beschermde methode.
3	Wijzigen van patroon voor toepassen van de monitor.
4	Toevoegen van een tweede monitor aan een klasse.
5	Invoeren van gescheiden Read/write-monitoren voor drie methoden.
Asynchrone methoden concern	
6	Toevoegen/verwijderen van een asynchroon uit te voeren methoden.
7	Hernoemen van een methode met asynchroon gedrag.
8	Hernoemen methode die de *GUI update*.
9	Threads voor verwerken binnenkomende connecties hogere prioriteit geven dan threads voor bijwerken user interface.
10	Beperken van het aantal threads voor IO-controles.
11	Aanpassen implementatie van het '*user interface update* concern'.
12	Aanpassen implementatie van *asynchrone-methoden* concern.

6.2 Gerealiseerde oplossingen

Binnen dit hoofdstuk worden de AOP implementaties van de concurrency concerns besproken voor beide stijlen. Dit wordt gevolgd door een sectie waarin wordt aangegeven hoe het gebruik van de AOP implementatie gecontroleerd kan worden. Hierna wordt aangetoond dat de performance van de AOP oplossing gelijkwaardig is aan de OOP oplossing. En tot slot een beschrijving van de uitgevoerde *Aspect Refactoring.*

6.2.1 Oplossing op basis van enumeration pointcuts

Deze oplossingen maken gebruik van *enumeration* van *joinpoints* als *pointcut*. Hieronder is te zien hoe zo'n *pointcut* er dan uitziet en hoe een *around advice* toegepast kan worden op de *joinpoints* uit het *pointcut, zie Figuur 12*. Het voorbeeld is een versimpelde versie van de AspectJ implementatie voor het *concurrency control concern.*

In *Figuur 13 en Figuur 14* zijn versimpelde implementaties van respectievelijk het *Asynchrone methoden concern* en het *GUI-updating concern.*

De reden dat deze hieronder getoond worden is om te laten zien hoe in deze implementaties de concerns gescheiden zijn. Op de plek waar normaal gesproken *tangling* plaatsvindt, is nu een aanroep van *proceed* terug te vinden. Hierdoor is de *tangling* uit de code verdwenen, de implementaties van de concerns zijn gescheiden, maar de functionaliteit is gelijk gebleven. De aanroep van *proceed* en het *around advice* zorgen hiervoor.

```
pointcut ProtectedMethods(Object operation) :(
    call( public static AzureusCore AzureusCoreImplE.create(..) )
    // more joinpoints
) && this(operation);

Object around(Object operation) : ProtectedMethods(operation) {
      Object returnValue = null;
      AEMonitor monitor = getMonitorForObject( thisJoinPoint );

      try{
          monitor.enter();
          returnValue = proceed(operation);
      }finally{
          monitor.exit();
      }
      return returnValue;
}
```

Figuur 12: Versimpelde implementatie concurrency control concern

```
void around() :  MethodsUpdatingGUI() {
    org.gudy.azureus2.ui.swt.Utils.execSWTThread(
            new org.gudy.azureus2.core3.util.AERunnable() {
                public void runSupport() {
                    proceed();
                }
            }
    );
}
```

Figuur 13: Versimpelde implementatie GUI-update concern

```
void around()    : AsynchronousMethodsLowPrio() {
    AEThreadPool pool = AEThreadPoolFactory.getInstance( "lowPrio" );
    AERunnable runner =
        new AERunnable(){
            public void runSupport(){
                do{
                    proceed();
                }while(deamon);
            }
        };
    pool.setThreadPriority( low );
    pool.run(runner);
}
```

Figuur 14: Versimpelde implementatie Asynchrone methode concern

Beperkingen:

- Bij het *concurrency control concern* werden soms meerdere monitor-objecten binnen een methode gebruikt. Dit is met deze AspectJ-implementatie niet mogelijk zonder grote structurele wijzigingen.
- Bij het *asynchrone methoden concern* geldt hetzelfde, men wil per methode aangeven welke prioriteit het is (laag, medium, hoog) en of het een *daemon -thread* is (thread die herhaald uitgevoerd wordt). Hierdoor moet voor elke combinatie hiervan een *pointcut* en *advice* gemaakt worden. Dit betekent dat er een grote onoverzichtelijke verzameling *pointcuts* en *advices* dreigt te ontstaan.
- Een derde beperking is dat de *joinpoints* die gebruikt worden binnen objecten van *anonymous-classes* niet vervangen kunnen worden, omdat het onmogelijk is een *pointcut* hiervoor te definiëren, annotaties kennen dit probleem niet.

6.2.2 Oplossing op basis van annotatie pointcuts

Deze oplossingen maken gebruik van *attributes(annotaties)* als *joinpoints* binnen een *pointcut*. Hieronder is te zien hoe zo'n *pointcut* er dan uitziet, *Figuur 15*. In *Figuur 16,* is een versimpelde versie van de AspectJ implementatie voor het *concurrency control concern* te zien. Deze lijkt op de voorgaande alleen is het nu mogelijk om via de annotatie een *monitorId* mee te geven waardoor een beperking van de *enumeration* oplossing wordt opgeheven.

```
pointcut  ProtectedMethods(
            ProtectedByMonitor monitorAnnotation,
            Object operation)
    : execution( @ProtectedByMonitor * *.*(..) )
        && @annotation( monitorAnnotation )&& this( operation );
```

Figuur 15: Voorbeeldpointcut op basis van annotaties (ProtectedByMonitor annotatie)

```
Object around(ProtectedByMonitor monitorAnnotation, Object operation)
                    : ProtectedMethods(monitorAnnotation,operation) {
      Object returnObject;
      String monitorId = monitorAnnotation.name();
      AEMonitor monitor = getMonitorForObject( thisJoinPoint, monitorId );

      try{
          monitor.enter();
          returnObject = proceed( monitorAnnotation, operation );
      }finally{
          monitor.exit();
      }

      return returnObject;
}
```

Figuur 16: Versimpelde implementatie concurrency control concern met annotaties

In *Figuur 17* en *Figuur 18* zijn versimpelde implementaties te zien van respectievelijk; het *Asynchrone methoden concern* en het *GUI-updating concern.* De implementatie van het *GUI-updating concern* is vrijwel gelijk gebleven. Echter de implementatie van het *asynchrone methoden concern* is wel flink aangepast. Er kan nu centraal ingesteld worden welke *thread pools* bij welke threads (bij welke klassen) horen. Dit was bij de *enumeration* oplossing lastig. Bovendien kan nu ook in de annotatie worden verwerkt of er sprake is van een *daemon thread*. Bij deze oplossing is dit eenvoudig te realiseren doordat er nu slechts één *advice* nodig is dat zijn informatie haalt uit een tweetal annotaties. De ene annotatie staat lokaal in de code, de andere wordt vanuit een centrale locatie verdeeld (hier wordt AspectJ zowel gebruikt om annotaties toe te voegen aan methoden *en* om ze te verwerken). Nu is het dus voor de ontwikkelaars eenvoudig om lokaal aan te geven welke methoden asynchroon verwerkt dienen te worden en of dit herhaald dient te worden. En centraal bestaat het overzicht van de threads en hun *thread pools* met bijbehorende prioriteit. Dit maakt het beheer en de implementatie van asynchroon gedrag vele malen eenvoudiger.

```
void around(UpdatesGUI ug) :  MethodsUpdatingGUI(ug) {
    final UpdatesGUI annotation = ug;
    org.gudy.azureus2.ui.swt.Utils.execSWTThread(
            new AERunnable() {
                public void runSupport() {
                    proceed(annotation);
                }
            }
    );
}
```

Figuur 17: Versimpelde implementatie GUI-updating concern met annotaties

```
void around(RunsAsynchronous RA,BelongsToPool BTP)
        : AsynchronousMethods( RA, BTP) {
    final RunsAsynchronous runsAnno = RA;
    final BelongsToPool belongsAnno = BTP;
    final boolean deamon = runsAnno.deamon();
    final String poolName = belongsAnno.name();

    AEThreadPool pool = AEThreadPoolFactory.getInstance(poolName);

    AERunnable runner =
        new AERunnable(){
            public void runSupport(){
                do{
                    proceed( runsAnno, belongsAnno );
                }while(deamon);
            }
    };
    pool.run(runner);
}
```

Figuur 18: Versimpelde implementatie Asynchrone methoden concern met annotaties

6.2.3 Enforcement van de oplossingen

Deze paragraaf beschrijft hoe voorkomen kan worden dat ontwikkelaars weer in hun 'oude patroon' vervallen. We hebben nu immers een oplossing waarmee het systeem ge-*refactored* kan worden, maar wat nog ontbreekt is een hulpmiddel om ervoor te zorgen dat de ontwikkelaars niet toch de concerns handmatig op de OO manier gaan implementeren (*enforcement*).

Ook hiervoor is een oplossing gevonden binnen de 'AspectJ toolbox'. AspectJ stelt de ontwikkelaar namelijk in staat om op basis van *pointcuts* door de AspectJ-compiler waarschuwingen of foutmeldingen te laten genereren. Hier wordt AspectJ dus niet als codegenerator ingezet maar als compiler/parser uitbreiding. Dit is ook een aanpak die binnen *[Laddad,2003]* wordt voorgesteld.

```
declare warning : unallowedAEMonitorUsage()
    : "Please use the 'AEMonitor Aspect for concurrency " +
      "control' to enforce Consistent Behaviour";
```

Figuur 19: Voorbeeldcode voor het declareren van compiler warning voor een pointcut (unallowedAEMonitorUsage)

Bovenstaand codefragment (*Figuur 19*) zal de ontwikkelaars van het Azureus-team helpen bij het in stand houden van de AO-Refactoring door ze te waarschuwen wanneer AEMonitor gebruikt wordt buiten de AspectJ-code.

6.2.4 Performance

Ondanks het feit dat dit onderzoek zich richt op *modularity* en *encapsulation* is ook performance van belang. Wanneer de AspectJ oplossingen een slechte performance zou leveren, dan is het de vraag of de inzet wel een verstandige zet is.

Onderstaande tabel toont in hoeverre de performance verandert bij het gebruik van AspectJ voor het Asynchrone methoden concern. Hierbij is een vergelijking gemaakt tussen de normale

implementatie voor het starten van asynchrone methoden en de AspectJ versie ervan. Hierbij zijn uiteraard dezelfde methoden asynchroon uitgevoerd alleen het mechanisme voor asynchroon gedrag is aangepast.
Daarnaast is het algoritme voor de functionaliteit die asynchroon uitgevoerd wordt, zo eenvoudig mogelijk gehouden. Hierdoor is de doorlooptijd (gemeten in ms) afhankelijk van de overhead voor het creëren van threads en dat is wat we willen meten voor beide oplossingen. Bovendien wordt er gemeten per grote hoeveelheden threads om het effect van stoorfactoren zoals *garbage collectors* te minimaliseren. Tot slot zijn de tests 10 maal uitgevoerd en is het gemiddelde genomen van de waarden om dezelfde reden.

Tabel 5: Performance (overhead) concurrency control concern

	1000 threads	2000 threads
Normaal	0,248 ms per thread	0,248 ms per thread
Aspect	0,250 ms per thread	0,248 ms per thread
Verschil	0,81%	0,00%

Hetzelfde is gedaan voor het *concurrency control concern* hierbij is van twee situaties gemeten wat de overhead is van de normale code voor *concurrency control* en de AspectJ implementatie. De twee situaties verschillen in de mate van *contention*, het wachten van een thread op een *lock*. De eerste situatie is zonder *contention*, dus wanneer de threads vrije doorgang krijgen. De tweede situatie is één waarbij de threads wel *locks* delen en op elkaar moeten wachten.

Tabel 6: Performance (overhead) asynchrone methoden concern

	Zonder contention	Met contention
Normaal	0,66 ms per monitor	1,57 ms per monitor
AspectJ	0,66 ms per monitor	1,58 ms per monitor
Verschil	0,00%	0,64%

De conclusie die we kunnen trekken uit deze meetwaarden is dat de performance nauwelijks (minder dan 1%) te lijden heeft als gevolg van het gebruik van AspectJ. Het verschil zal in ieder geval voor de meeste desktop applicaties acceptabel zijn, aangezien de granulariteit van concurrency hier nog laag is. Bijvoorbeeld bij desktop applicaties met veel threading zoals Azureus hebben we het nog steeds maar over een kleine honderd threads en dus minder dan een milliseconde overhead.

6.2.5 Toepassing van aspect refactoring op Azureus

Het aanpassen van de volledige Azureus-applicatie op basis van beide AspectJ-oplossingen is te veel werk. Er is daarom gekozen om per concern een tiental klassen willekeurig te selecteren waarin het concern voorkomt. Een tiental klassen levert voldoende data om te kunnen analyseren wat de gevolgen zijn van het gebruik van AspectJ.

Er zijn twee typen refactoring uitgevoerd:
- *Extract method*
- *Aspect refactoring*

Om te controleren dat de werking van de code gelijk is gebleven, zoals zou moeten volgens de definitie van *refactoring [Fowler,2007] [Laddad,2003b],* zijn unit-tests geschreven.

Er zijn dus verschillende *branches* gemaakt voor de applicatie waarbij per *branch* één concern is ge-*refactored* in één stijl, zodat per concern en per stijl gemeten kan worden hoe ze de metrieken beïnvloeden (hun impact).

Dit heeft geresulteerd in de volgende branches:

Tabel 7: Gerealiseerde branches

1	Originele oplossing voor concurrency control
2	Originele oplossing voor asynchrone methode
3	Annotatie AO oplossing voor concurrency control
4	Annotatie AO oplossing voor asynchrone methode
5	Enumeratie AO oplossing voor concurrency control
6	Enumeratie AO oplossing voor asynchrone methode
7	OO baseline voor concurrency control
8	OO baseline voor asynchrone methode

6.3 Metingen aan het resultaat

De volgende secties tonen de bij dit onderzoek verzamelde meetwaarden voor *modularity* en *encapsulation* van de gerealiseerde *branches* van Azureus. Deze meetwaarden worden per type meting (OO-metrieken, AO-metrieken, scenario's) in tabellen getoond. Hierbij zijn de metingen uitgesplitst per concern voor de OO oplossingen, de AO oplossingen en de Baseline oplossingen. Baseline oplossingen zijn versie van de applicatie zonder het concurrency concern, waardoor de significantie van de meetwaardeverschillen beter bepaald kan worden.

6.3.1 OOP Metrieken

Onderstaande tabellen tonen de meetwaarden voor de OO-metrieken voor de oplossingen. De kollommen van de '*AO enumeratie branch*' en de '*AO annotatie branch*' zijn samengevoegd omdat de meetwaarden hetzelfde zijn.

Tabel 8: OO metrieken voor concurrency control concern

	OOP oplossing			AOP oplossing			Baseline		
	Total	Avg	Max	Total	Avg	Max	Total	Avg	Max
Method lines of code (LOC)	2185	9,9	158	1991	8,5	146	1980	9,5	150
Number of Methods (NOM)	209	6,5	57	228	6,8	56	209	6,5	57
Nested Block Depth (NBD)		1,89	8		1,77	7		1,70	7
McCabe Cyclomatic complexity per method (MCCabe CC)		2,53	38		2,44	23		2,43	25
Coupling between objects (CBO)		17,6	64		15,5	63		15,5	63
Lack of Cohesion of methods (LCOM)		0,60	0,34		0,49	0,91		0,73	0,96

Tabel 9: OO Metrieken voor het asynchrone methoden concern

	OOP oplossing			AOP oplossing			Baseline		
	Total	Avg	Max	Total	Avg	Max	Total	Avg	Max
Method lines of code (LOC)	6870	19,3	266	6770	18,6	261	6737	18,9	261
Number of Methods (NOM)	356	3,1	63	370	3,8	63	356	3,1	63
Nested Block Depth (NBD)		2,77	8		2,55	8		2,55	8
McCabe Cyclomatic complexity per method (MCCabe CC)		4,10	47		3,57	45		3,45	45
Coupling between objects (CBO)		34,0	125		30,3	121		27,7	121
Lack of Cohesion of methods (LCOM)		0,72	0,97		0,73	0,97		0,79	1

6.3.2 AOP Metrieken

Onderstaande tabellen tonen de meetwaarden voor de OO-metrieken voor de oplossingen. De kollommen van de '*AO enumeratie branch*' en de '*AO annotatie branch*' zijn samengevoegd omdat de meetwaarden hetzelfde zijn. Alleen bij CIM zien we verschillende waarden terug dus is hier tussen haakjes de waarden voor de annotatie-oplossing weergegeven.

Tabel 10: AO metrieken voor concurrency control concern

	OOP oplossing			AOP oplossing			Baseline		
	Total	Avg	Max	Total	Avg	Max	Total	Avg	Max
Lines of Code (LOC)	3534			3240			3194		
Coupling on Field Access (CFA)	23	0,61	9	21	0,53	9	21	0,58	9
Coupling on Method Call (CMC)	178	4,68	56	175	4,00	55	172	4,18	55
Coupling between Modules (CBM)	187	4,92	59	190	4,19	58	181	4,25	58
Coupling on Intercepted Modules (CIM)	0	0	0	43 (3,0)	21,5 (1,5)	42 (2)	0	0	0
Crosscutting degree of an aspect (CDA)	0	0	0	10	5	10	0	0	0
Coupling on Advice Execution (CAE)	0	0	0	10	1	1	0	0	0
Lack of Cohesion in Operations (LCO)	4494	118	2177	4978	105	2312	4977	115	2312

Tabel 11: AO Metrieken voor het asynchrone methoden concern

	OOP oplossing			AOP oplossing			Baseline		
	Total	Avg	Max	Total	Avg	Max	Total	Avg	Max
Lines of Code (LOC)	6782			6597			6541		
Coupling on Field Access (CFA)	140	1,21	24	135	1,15	24	135	1,16	24
Coupling on Method Call (CMC)	587	5,00	88	480	4,10	87	476	4,21	87
Coupling between Modules (CBM)	690	5,90	105	579	4,95	104	575	5,07	104
Coupling on Intercepted Modules (CIM)	0	0	0	38(4)	19(2)	34(2)	0	0	0
Crosscutting degree of an aspect (CDA)	0	0	0	12	0,10	12	0	0	0
Coupling on Advice Execution (CAE)	0	0	0	12	0,10	1	0	0	0
Lack of Cohesion in Operations (LCO)	5920	50,6	2570	5533	47,3	2531	5533	62,9	2531

6.3.3 Code statistieken bij onderhoud; Java oplossing vs. AspectJ oplossing

Tabel 12: Onderhoudsmetrieken voor concurrency control concern

	Origineel	Aspect	Aspect met annotaties
Onderhoudsscenario 1: *Toevoegen/verwijderen van een monitor aan een methode.*			
- Aantal regels	4, volledige consistent behavior implementeren	1, pointcut aanpassen	1, annotatie toevoegen
- Aantal modulen	1, de klasse	1, het aspect	1, de klasse
Onderhoudsscenario 2: *Hernoemen van een door een monitor beschermde methode.*			
- Aantal regels	1, methode declaratie aanpassen	2, pointcut en methode declaratie aanpassen	1, methode declaratie
- Aantal modulen	1, de klasse	2, aspect en klasse	1, de klasse

Onderhoudsscenario 3: *Wijzigen van patroon voor toepassen van de monitor.*			
- Aantal regels	Aantal te wijzigen regels * ieder voorkomen (1000+ in huidige versie)	Aantal te wijzigen regels	Aantal te wijzigen regels
- Aantal modulen	Iedere module waarin een monitor gebruikt wordt	1, het aspect	1, het aspect
Onderhoudsscenario 4: *Toevoegen van een tweede monitor aan een klasse.*			
- Aantal regels	5, volledige consistent behavior implementeren (4) en 1 voor declaratie monitor	Niet mogelijk	1, annotatie met andere naam als parameter
- Aantal modulen	1, de klasse	Niet mogelijk	1, de klasse
Onderhoudsscenario 5: *Invoeren van gescheiden Read/write-monitoren voor drie methoden.*			
- Aantal regels	Aantal te wijzigen regels * ieder voorkomen (3, bij dit scenario)	Aantal te wijzigen regels + 3 (1 voor elke wijziging)	Aantal te wijzigen regels + 3 (3 annotaties moeten aangepast worden)
- Aantal modulen	Max 3, bij dit scenario	1, het aspect	4, zowel het aspect als de code waarop het wordt toegepast (Max 3 in dit scenario)

Tabel 13: Onderhoudsmetrieken voor de asynchrone methoden & GUI Updating concerns

	Origineel	**Aspect**	**Aspect met annotaties**
Onderhoudsscenario 6: *Toevoegen/verwijderen van een asynchroon uit te voeren methoden.*			
- Aantal regels	2-4 voor het creëren van een anonieme klasse en thread	1, pointcut aanpassen	1, annotatie toevoegen
- Aantal modulen	1, de klasse	1, het aspect	1, de klasse
Onderhoudsscenario 7: *Hernoemen van een methode met asynchroon gedrag.*			
- Aantal regels	1 regel voor de methode	2, pointcut en methode declaratie aanpassen	1 regel voor de methode
- Aantal modulen	1, de klasse	2, het aspect en klasse	1, de klasse
Onderhoudsscenario 8: *Hernoemen methode die de GUI update.*			
- Aantal regels	1 voor het voor het creëren van een anonieme klasse	1, pointcut aanpassen	1, annotatie toevoegen
- Aantal modulen	1, de klasse	1, het aspect	1, de klasse
Onderhoudsscenario 9: *Threads voor verwerken binnenkomende connecties hogere prioriteit geven dan threads voor bijwerken user interface.*			
- Aantal regels	Onbekend	NVT	2, pointcut aanpassen en thread pool declareren
- Aantal modulen	Onbekend, men zal alle klassen moeten doorzoeken om te zien waar deze zich bevinden en welke threads er nog meer zijn	NVT	1, het aspect
Onderhoudsscenario 10: *Beperken van het aantal threads voor IO-controles.*			
- Aantal regels	2, voor aanmaken thread pool en toepassen	NVT	2, pointcut aanpassen en thread pool declareren
- Aantal modulen	1, de klasse	NVT	1, het aspect
Onderhoudsscenario 11: *Aanpassen implementatie van het 'user interface update concern'.*			
- Aantal regels	Aantal te wijzigen regels * ieder voorkomen	Aantal te wijzigen regels	Aantal te wijzigen regels
- Aantal modulen	Iedere module waarin het voorkomt	1, het aspect	1, het aspect

Onderhoudsscenario 12: *Aanpassen implementatie van asynchrone-methoden concern.*			
- Aantal regels	Aantal te wijzigen regels * ieder voorkomen	Aantal te wijzigen regels	Aantal te wijzigen regels
- Aantal modulen	Iedere module waarin het voorkomt	1, het aspect	1, het aspect

6.4 Analyse meetresultaten

Hieronder volgt een analyse/interpretatie van de meetresultaten uit de vorige paragraaf. Deze worden per concern behandeld.

6.4.1 Concurrency control concern

Hieronder wordt voor het *concurrency control concern* beschreven hoe de verschillende branches presteren per type meting (OO-metrieken, AO-metrieken en onderhoudsmetrieken).

OO-metrieken
Wanneer we in *Tabel 8* de metrieken voor *modularity* en *encapsulation* (*Coupling Between Objects, Lack of Cohesion, Method Count and Size*) bekijken, zien we dat de AspectJ-oplossing vrijwel overal beter scoort. Het verwijderen van de referenties naar de AEMonitor in alle klassen waar het concern geïmplementeerd was, is de belangrijkste oorzaak hiervoor. Dit speelt met name een positieve rol bij de *Coupling Between Objects,* hier zien we ook dat de score gelijk is aan de baseline.
Tegelijkertijd is hierdoor de maximale waarde bij de *Lack of Cohesion* metriek verhoogd wat slecht is. De oorzaak hiervoor is dat deze metriek gebaseerd is op de verhouding tussen het aantal methoden van een klasse en het gebruik van membervariabelen van deze klasse. Het feit dat we de relatie met AEMonitor verwijderen betekent dat er één membervariabele minder is en zal deze waarde dus negatief beïnvloeden.
Voor de metrieken van *method count* en *size* blijkt het toepassen van refactoring een positieve bijdrage te leveren, doordat hierdoor kleinere methoden ontstaan.

Daarnaast zien we in *Tabel 8* ook dat de resultaten voor metrieken op het gebied van complexiteit (*Nested Block Depth* en *McCabe complexity*) positief beïnvloed zijn bij de wijziging naar AspectJ. Voor de *Nested Block Depth* geldt dat het toepassen van refactoring ervoor zorgt dat de *nesting* minder wordt(bijna vergelijkbaar met de baseline), maar ook het feit dat er een *try-finaly* blok verdwijnt beïnvloedt deze metriek positief. Dit laatste is ook de grootste invloed voor de verbeterde McCabe complexity(vrijwel gelijk aan de baseline), het verwijderen van het *try/finaly* blok betekent dat er minder mogelijke *control-flows* in de code aanwezig zijn. In werkelijkheid worden deze later weer door de AspectJ Weaver teruggebracht, maar dat is tijdens het compileren en is dus niet voor de ontwikkelaar zichtbaar.
Maar ook het opsplitsen van methoden zorgt voor lagere McCabe complexity per methode, maar het totaal per klasse verandert hierdoor niet.

AO-metrieken
Op basis van de metrieken in *Tabel 10* zien we hier dat de AspectJ-oplossing beter is. Bij de metrieken voor koppeling (*Coupling on Field Acces (CFA), Coupling on Method Call (CMC), Coupling between Modules (CBM)*) zien we dat de waarden bij de AOP oplossing lager (beter) zijn. Dit betekent dat hier minder koppeling tussen de modulen is. Dit zagen we eerder ook al terug bij de OO-metrieken (*Tabel 8*), waar bijvoorbeeld voor de *Coupling Between Objects* metriek te zien was dat de koppeling lager is bij de AspectJ-oplossing.
De waarde van CFA is bij beide oplossingen laag (goed). Dit is opvallend want normaal gesproken scoort een AspectJ-oplossing hier slechter dan de OO oplossing. Een verklaring hiervoor is dat deze AspectJ-oplossing geen attributen/velden uit de module gebruikt waarop de aspecten worden toegepast. Hierdoor blijft de waarde van CFA min of meer gelijk.

Bovendien zien we ook hier weer dat het aantal regels code flink verminderd is. Niet altijd geldt dat minder regels code beter is, echter in dit geval is de oorzaak beter herbruikbare (kleinere) 'methoden' en is dit dus wel het geval.

Op basis van de metriek *Coupling on Intercepted Modules (CIM)* kunnen we concluderen dat de oplossing op basis van annotaties beter is. De op annotaties gebaseerde oplossing heeft een lage waarde voor CIM, immers het aantal modulen dat letterlijk genoemd wordt in de Aspecten is één (alleen de annotatie). Bij de 'normale' AspectJ-oplossing worden er evenveel modulen genoemd als dat er modulen zijn waarin het concern aanwezig is. Dit betekent dat de koppeling tussen Aspect en het systeem bij annotaties veel lager is. De op annotaties gebaseerde oplossing is daarom beter herbruikbaar voor andere systemen vanwege de lage koppeling met het systeem.

Het laatste dat opvalt, is dat de metrieken voor AOP positiever zijn dan van de baseline. De oorzaak hiervoor kan o.a. gevonden worden in het feit dat de AspectJ code zelf het aantal modulen verhoogt, maar zelf een lagere koppeling per module kent dan het originele systeem.

Scenario´s
In *Tabel 12* zien we duidelijk één grote winnaar, namelijk de oplossing op basis van AspectJ en annotaties. Deze oplossing laat over de scenario´s heen de beste resultaten zien voor het aantal te wijzigen modulen en regels.

Bij de meeste wijzigingen blijkt het aantal modulen dat aangepast dient te worden niet veel te verschillen. Meestal geldt dat de wijziging tot één of twee modulen beperkt blijft. Het grote verschil ontstaat bij wijziging 3, hierbij scoort de originele oplossing erg slecht. Bij het aanpassen van het patroon dienen alle modulen die er gebruik van maken aangepast te worden. Dit zijn er honderden en dat betekent erg veel werk. Deze wijziging toont duidelijk dat het voordeel van het onderbrengen van het *consistent behavior* in één module, waardoor het gedrag verborgen is voor de gebruikende modulen.

Het aantal te wijzigen regels code tussen de AspectJ-oplossingen onderling verschilt niet veel, afwisselend doen ze het net iets beter. Wel blijkt dat het verschil tussen de AspectJ-oplossingen en het origineel erg groot is in enkele gevallen. Bij wijziging 3 en 5 is het aantal regels dat aangepast dient te worden van een hele andere orde. Dit toont duidelijk dat de *encapsulation* van het 'probleem' in de originele oplossing slecht gerealiseerd is. Een wijziging in het interne gedrag betekent daardoor dat er veel code aangepast dient te worden.

6.4.2 Asynchrone methode concern (incl. GUI updating)

Hieronder is te zien hoe de *mogelijke* verbeteringen verdeeld zijn over de drie categorieën (OO-metrieken, AO-metrieken en onderhoudsmetrieken).

OO-metrieken
Wanneer we in *Tabel 9* de metrieken voor *modularity* en *encapsulation* (*Coupling Between Objects, Lack of Cohesion, Method Count and Size*) bekijken, zien we dat de AspectJ-oplossing wederom vrijwel overal beter scoort(met scores vrijwel gelijk aan de baseline), net als bij het vorige concern. Ook hier is verwijderen van de link tussen de gewone klassen en concurrency gerelateerde klassen, in dit geval *AEThread* en *AERunnable*, de bron voor verbeterde resultaten op het gebied van *coupling between objects*.
Wat wel opvalt, is dat de *Lack of Cohesion* metriek gelijk is gebleven, bij het andere concern zagen we een verslechtering, hier blijft het resultaat gelijk. Dit komt omdat bij deze aanpassingen geen wijziging zijn gedaan aan member-variabelen en de relatie dus ongewijzigd blijft.
Voor de metrieken van *method count and size* blijkt het toepassen van refactoring op de methoden waar het concern geïmplementeerd werd wederom een positieve bijdrage te leveren, hierdoor ontstaan kleinere methoden.

Daarnaast zien we in *Tabel 9* dat de resultaten voor metrieken op het gebied van complexiteit (*Nested Block Depth* en *McCabe complexity*) ook positief beïnvloed zijn bij de wijziging in AspectJ. Voor de *Nested Block Depth* geldt, net als bij *method count and size,* dat het toepassen van refactoring ervoor zorgt dat de *nesting* minder wordt, maar uiteraard ook het feit dat er een anonieme klasse verdwijnt beïnvloedt deze metriek positief. Deze anonieme klassen zorgen voor een extra diepe *nesting*.
Het opsplitsen van methoden zorgt voor lagere *McCabe complexity* per methode.

AO-metrieken
Bij de AO-metrieken in *Tabel 11* zien we exact hetzelfde beeld als bij het vorige concern. Bij de metrieken voor koppeling (*Coupling on Field Access (CFA), Coupling on Method Call (CMC), Coupling between Modules (CBM)*) zien we dat de waarden bij de AspectJ-oplossing lager zijn. Dit betekent dat hier minder koppeling tussen de modulen is. Bovendien zien we ook hier weer dat het aantal regels code flink vermindert is.
En ook hier kunnen we op basis van de metriek *Coupling on Intercepted Modules (CIM)* concluderen dat de oplossing op basis van annotaties beter is.

Het laatste dat ook hier weer opvalt, is dat de metrieken voor AOP positiever zijn dan van de baseline. De oorzaak hiervoor kan o.a. gevonden worden in het feit dat de AspectJ code zelf het aantal modulen verhoogt, maar zelf een lagere koppeling per module kent dan het originele systeem.

Scenario´s
Ook hier (*Tabel 13*) zien we weer eenzelfde beeld als bij het andere concern. Hier zien we ook weer duidelijk één grote winnaar, namelijk de oplossing op basis van AspectJ en annotaties. Deze oplossing laat over de scenario´s heen de beste resultaten zien voor het aantal te wijzigen modulen en regels.

6.4.3 Samenvatting

Voor beide concerns zien we min of meer dezelfde trend, de overgang van de originele oplossing naar een AspectJ oplossing levert betere resultaten op. Zowel de OO- als de AO-metrieken spreken in het voordeel van de AspectJ-oplossing. De belangrijkste metrieken voor *modularity* en *encapsulation* spreken allen in het voordeel van de AspectJ-oplossing. Ook de resultaten bij de scenario´s spreken in het voordeel van de AspectJ-oplossing.

Wanneer we de resultaten van de AspectJ-stijlen onderling vergelijken zien we dat de metrieken, met uitzondering van CIM, geen onderscheid maken. Echter voor de CIM metriek geldt dat de annotatie-oplossing beter scoort en ook bij de onderhoudsmetrieken scoort deze veel beter. Op basis hiervan kan geconcludeerd worden dat de oplossing op basis van annotaties de beste is.

6.5 Conclusie

Binnen dit hoofdstuk is in kaart gebracht of met behulp van AspectJ een betere implementatie van de crosscutting concurrency concerns uit Azureus gerealiseerd kan worden. Hiervoor is een deel van Azureus aangepast met AspectJ volgens twee stijlen: met *attribute pointcut mechanismen(annotaties)* en *enumeration pointcut mechanisme.* Er is vervolgens per stijl per concern vastgesteld wat de *modularity* en *encapsulation* is middels de volgende drie methoden:OO Metrieken, AO Metrieken en onderhoudsscenario's.

Waarbij voor beide concerns bleek dat de oplossing met *attribute pointcut mechanismen (annotaties)* het beste resultaat leverde, beter dan de OO oplossing en de oplossing met het *enumeration pointcut mechanisme*.

Op basis hiervan kan dus geconcludeerd worden dat het gebruik van AspectJ de implementatie van de concurrency concerns verbeterd ten aanzien van *modularity* en *encapsulation.*

Samengevat:

- Er is een tweetal AspectJ oplossingen gerealiseerd, één op basis van *enumeration pointcuts* en één op basis van *attribute poincuts* (annotaties).
- De oplossing op basis van *enumeration* kon geen volledige oplossing opleveren.
- De metrieken (aspectgeoriënteerd en objectgeorienteerd) en scenario's toonden aan dat de AspectJ oplossingen betere *modularity* en *encapsulation* mogelijk maken van *crosscutting concurrency concerns*.
- Beide concurrency concerns toonden een zelfde verbetering.
- De AspectJ-oplossing op basis van *attribute pointcut mechanisme* leverde de beste resultaten. De AspectJ-stijl heeft dus wel invloed.

6.6 Reflectie

Er is een aantal aspecten dat de validiteit van het onderzoek in gevaar kan brengen. De belangrijkste zijn hieronder opgesomd en behandeld. Hierbij is aangegeven wat eventueel gedaan is om de risico's voor de validiteit van het onderzoek te beperken.

6.6.1 Correctheid van de metingen

Een aanname die vooraf gemaakt werd is dat de tools correct werken, dat de opgeleverde metrieken correct zijn. In werkelijkheid zal dit niet altijd het geval zijn. Om dit te ondervangen zijn voor de OO-metrieken twee tools gebruikt en zijn de waarden vergeleken. De waarden bleken gelijk te zijn en op basis hiervan kan aangenomen worden dat ze ook correct zijn.
Voor de AO-metrieken is op basis van beredeneren van te voren in kaart gebracht wat de te verwachten waarden ongeveer zouden zijn. Vervolgens zijn deze daadwerkelijk berekend en vergeleken met de verwachtingen. De waarden bleken in de lijn der verwachting te liggen en op basis daarvan kan aangenomen worden dat deze in grote lijnen correct zijn.

6.6.2 Generaliseerbaarheid

Binnen dit onderzoek is aangetoond dat voor de geselecteerde concurrency concerns verbetering gerealiseerd kon worden. De vraag is in hoeverre dit betekent dat voor alle *concurrency concerns* geldt dat AspectJ een verbetering kan opleveren. Dit is iets waarop deze scriptie geen antwoord kan geven.
Wat in het voordeel spreekt van deze aanname is dat de concerns (concurrency control, en asynchrone methoden) niet op elkaar lijken voor wat betreft hun intenties. Zowel de implementaties als het doel zijn verschillend. Maar het is maar de vraag of op basis hiervan aan te nemen is dat de combinatie van beide representatief is voor alle concurrency concerns.

6.6.3 Omvang van refactoring

Een beperking van dit onderzoeksdeel is dat er een meting is gedaan op basis van een deel van de applicatie, hierbij is een aanname gedaan dat dit voldoende is om de effecten van AspectJ te meten voor de concerns. Bij een volledige refactoring zal de verhouding AO tot OO code veranderen. De hoeveelheid AO code zal bij de volledige refactoring nauwelijks toenemen t.o.v. de AO code van de nu uitgevoerde refactoring, terwijl er wel veel meer (OO) code in de volledige applicatie zit ten opzichte van het deel dat nu is aangepakt. De validatie van de aanname dat dit geen negatieve invloed op de meetwaarden heeft valt buiten de scope van het onderzoek omdat hier niet voldoende tijd voor is.

7 Generalisatie

Binnen dit onderzoek is tot nu toe vastgesteld welke concurrency gerelateerde concerns zich binnen één applicatie bevinden en of concurrency concerns *crosscutting* zijn. Vervolgens is een aanpak opgesteld en toegepast om deze concerns beter te implementeren, met betrekking tot de principes van *modularity* en *encapsulation*.
Echter hiermee is het nog lastig om een brede conclusie te trekken over concurrency gerelateerde concerns in het algemeen. Daarom zal in dit hoofdstuk nader onderzocht worden of de concurrency concerns die binnen Azureus zijn vastgesteld ook voorkomen in andere applicaties en in hoe verre daar ook sprake is van *crosscutting*. Oftewel, dit hoofdstuk is bedoeld om aan te tonen dat de gevonden concurrency concerns een bredere scope omvatten dan alleen Azureus, zodat er ook een bredere conclusie getrokken kan worden over concurrency concerns in het algemeen.

7.1 De applicaties

Bij de selectie is gebruik gemaakt van *SourceForge* voor het vinden van Java applicaties met multi-threading. Hierbij is gebruik gemaakt van het overzicht *Most Active Java Applications*[2]. Daarnaast is de software van een intern project van LogicaCMG onderzocht.
De volgende applicaties zijn geselecteerd:

Tabel 14: Geselecteerde applicaties voor controle generalisatie

Castor	*Een framework voor het koppelen van Java-objecten, XML Documenten en relationele (database) tabellen. Zie http://www.castor.org/*
FreeCol	*Open source implementatie van het spel Colonization, een strategie spel, inclusief multi-player opties. Zie http://freecol.org/*
Apache Lucene	*Search engine library voor het, met goede performance, doorzoeken (full-text search) van tekst. Zie http://lucene.apache.org*
Rapid Miner	*Rapid miner is de meest gebruikte open source applicatie voor data mining www.rapidminer.com/*
LogicaCMG	*Het gaat hier om een intern project van LogicaCMG waarbij een Multimedia-applicatie ontwikkeld wordt. Dit project valt onder een Non Disclosure Agreement*

7.2 Resultaten

Tabel 15 (op de volgende pagina) toont per applicatie welke van de concurrency concerns uit dit onderzoek ook een rol spelen in die applicatie en of er sprake is van *crosscutting* (*tangling*, *scattering*).

Opmerkingen bij de tabel:
- **LCMG**: Veel parallellisme op klasse i.p.v. methode niveau en veel *event-based* parallellisme.
- **Castor**: Gebruikt *tasks* ipv *threads* waardoor de asynchrone concerns heel anders zijn.
- **RapidMind en FreeCol**: Deze projecten gebruiken *synchronized* voor concurrency control.
- **Azureus**: Waarden voor concurrency control zijn berekend op basis van steekproefresultaten

7.3 Evaluatie

In *Tabel 15* zien we dat de concerns van Azureus een subset vormen van de concerns uit de andere applicaties. Opmerkelijk is dat de concerns uit Azureus wel voorkomen binnen de andere applicaties, maar in mindere mate. De oorzaak hiervan kan vermoedelijk gevonden worden in het feit dat zij andere concerns bezitten die een hoofdrol spelen. Bij het LogicaCMG project bleek bijvoorbeeld dat *events* een veel grotere rol spelen dan bijvoorbeeld asynchrone methoden.
Daarnaast valt de kleinere rol voor concurrency control op. Hier is een vrij logische verklaring voor te vinden in het feit dat binnen de meeste applicaties dit opgelost wordt met het taalelement

[2] http://sourceforge.net/softwaremap/trove_list.php?form_cat=198

synchronized. Zoals in *Tabel 15* te zien is blijkt dat deze wel veel gebruikt wordt en ook over een grote mate van *scattering* beschikt.

7.4 Conclusie

De *crosscutting concurrency concerns* uit Azureus blijken een bredere scope te omvatten dan alleen Azureus. Van alle drie de concerns blijkt dat op zijn minst één van de geselecteerde applicaties dit concern ook bevat en dat er daarbij ook sprake is van *crosscutting*.
Of de verbeteringen uit H6 ook generaliseerbaar zijn is niet gevalideerd, maar wel aannemelijk. Immers, wanneer hetzelfde probleem zich voordoet (dezelfde *crosscutting concerns* met dezelfde classificatie in het model van Marin *[Marin,2005]*) en deze dezelfde oorzaak hebben (de tekortkomingen van objectdecompositie) dan lijken de 'problemen' zodanig op elkaar dat het aannemelijk is dat een oplossing in beide gevallen mogelijk een min of meer vergelijkbaar effect zal hebben. Echter het bewijzen hiervan valt buiten de scope van deze scriptie.

	Azureus	Castor	FreeCol	LCMG	Lucene	Rapid Miner
Algemeen						
# Packages	420	87	27	93	22	113
# Classes	1.737	855	375	876	447	1451
Concurrency Control Concern						
* Scattering (calls/pkgs)	1043 / 122	30 / 4	NVT	14 / 3	2 / 22	NVT
* Scattering (incl synchronized)	404 / 56	220 / 25	59 / 12	610 / 53	138 / 10	77 / 13
* Tangling	Ja	Ja	NVT	Ja	Ja	NVT
Asynchrone Methods Concern						
* Scattering (calls/pkgs)	158 / 101	NVT	13 / 9	17 / 9	4 / 22	10 / 5
* Tangling	Ja	NVT	Ja	Ja	Ja	Ja
Userinterface Concern						
* Scattering (calls/pkgs)	135 / 28	NVT	20 / 4	NVT	NVT	4 / 2
* Tangling	Ja	NVT	Ja	NVT	NVT	Ja
Concurrency control concern als consistent behavior						
* Puur	552	16	NVT	2	0	NVT
* Minimaal afwijking	439	12	NVT	9	4	NVT
* Niet	52	2	NVT	3	0	NVT
Asynchrone als variability						
* Ja	155	NVT	11	17	1	5
* Nee	3	NVT	2	0	10	5
Asynchrone als behavior						
* Volledig patroon	89	NVT	0	0	0	0
* Onvolledig patroon	64	NVT	13	12	11	9
* Geen patroon	5	NVT	0	4	0	1
GUI als variability						
* Ja	130	NVT	18	NVT	NVT	4
* Nee	5	NVT	2	NVT	NVT	0

Tabel 15: Meetresultaten aanwezigheid concerns in andere applicaties

8 Evaluatie en Conclusies

Binnen deze scriptie is een antwoord gegeven op de vraag:

In welke mate zijn de principes van modularity en encapsulation toepasbaar voor de implementatie van concurrency in objectgeoriënteerde systemen?

Hiervoor is van één applicatie onderzocht welke concerns er vastgesteld konden worden op het gebied van *concurrency* en in welke mate deze in objectgeoriënteerde implementatie beschikte over goede *modularity* en *encapsulation*. Hierbij kon geconstateerd worden dat binnen de applicatie sprake was van een slechte *modularity* en *encapsulation* van de onderzochte concurrency concerns, er was sprake van *crosscutting* (*zie H5 Aspect mining van concurrency concerns*).

Vervolgens zijn experimenten uitgevoerd met AspectJ om te onderzoeken of AOP een betere *modularity* en *encapsulation* kon leveren voor de concurrency concerns. Op basis van de metrieken bleek dit inderdaad het geval. Voor de onderzochte concerns kon een betere implementatie opgesteld worden met behulp van AspectJ en annotaties. Waarbij wel gold dat er toch nog enige mate van spreiding(*scattering*) van het concern plaatsvond, al was dit niet het geval voor de AspectJ-oplossing op basis van *enumeration* (*zie H6 Aspect refactoring van concurrency control constructs*).

Op basis hiervan kon geconcludeerd worden dat de principes van *modularity* en *encapsulation* voor deze concurrency concerns binnen deze objectgeoriënteerde applicatie niet optimaal waren.

Vervolgens is onderzocht of dit resultaat generaliseerbaar was door te valideren dat de onderzochte concerns ook in andere applicaties een rol spelen. Dit bleek inderdaad het geval. Hiermee is gevalideerd dat het probleem (het bestaan van crosscutting concurrency concerns) generaliseerbaar is. Maar het is alleen aannemelijk gemaakt dat AspectJ ook bij deze applicaties dezelfde verbeteringen oplevert, dit is niet gevalideerd *(zie H7 Generalisatie).*

In hoeverre de gerealiseerde *modularity* en *encapsulation* optimaal is kan niet vastgesteld worden, mogelijk dat andere technieken of programmeerparadigma´s nog beter in staat zijn de concurrency concerns te implementeren voor deze principes. Wat ook niet vaststaat, is dat de behaalde resultaten ook voor andere concurrency concerns gelden. Dit zou in een vervolg onderzoek aangepakt kunnen worden.

De conclusie is dus dat de implementatie van meerdere concurrency concerns in verschillende applicaties niet optimaal was met betrekking tot *modularity* en *encapsulation* (*crosscutting*) en dat AspectJ hier verbeteringen voor kon realiseren bij één van de onderzochte applicaties.

8.1 Geleverde bijdragen

1 Een overzicht van (crosscutting) concurrency concerns in multi-threaded software

Binnen dit onderzoek is voor een groot bestaand systeem, Azureus, in kaart gebracht hoe de concurrency concerns geïmplementeerd zijn met betrekking tot *modularity* en *encapsulation*.
Het in kaart brengen van *concurrency concerns* is iets dat nog weinig aandacht heeft gekregen, zeker in deze omvang van geclassificeerde concerns over meerdere applicaties.

2 Een toolkit voor multi-threading in AspectJ

Er is in AspectJ een *toolkit* ontwikkeld met hulpmiddelen voor het beter implementeren van enkele *crosscutting concurrency concerns*. Dit soort *toolkits* voor concurrency in AspectJ zijn nog vrij uniek. Al is wel al eerder *[Cunha,2006]* aangetoond dat concurrency *design patterns* in AspectJ geïmplementeerd kunnen worden in een *toolkit*.

3 Een methode voor het vergelijken van AOP en OOP systemen op modularity en encapsulation
Bovendien is een methode ontwikkeld om objectgeoriënteerde en aspectgeoriënteerde systemen te vergelijken op hun *modularity* en *encapsulation*. Deze methode is toegepast binnen dit onderzoek. Het vergelijken tussen deze twee systemen met verschillende paradigma's is een redelijk nieuw onderzoeksgebied. Al is de methode uiteraard wel gebaseerd op bestaande methoden, *zie H3 Onderzoeksmethode*. Wel dient deze methode nog gevalideerd te worden.

4 Een gestructureerde aanpak voor het verbeteren van de implementatie van concurrency concerns in multi-threaded software
Tot slot is de beschreven gestructureerde aanpak, voor het analyseren van multi-threaded software op verbeterpunten van concurrency concerns en de transitie naar een verbeterde oplossing, redelijk nieuw. Binnen deze opdracht is een duidelijk stappenplan opgesteld waarin beschreven staat hoe de implementatie van de concurrency concerns in multi-threaded software met een gestructureerde aanpak verbeterd kan worden. Ook dit is gebaseerd op bestaande methoden, *H3 Onderzoeksmethode,* en kan nog beter gevalideerd worden.

8.2 Future work

FW 1) Opstellen van een lijst met veel gebruikte crosscutting concurrency concerns
Opstellen van een verzameling van (*crosscutting*) concerns voor concurrency die binnen meerdere applicaties worden waargenomen. Dus een verdere verdieping in *bijdrage 1.*

FW 2) Ontwikkelen van toolkit met implementaties van veel voorkomende crosscutting concurrency concerns
Ontwikkelen van een generieke *toolkit* waarin de AspectJ implementatie van *crosscutting concurrency concerns* als herbruikbaar *building block* worden opgenomen. Dus een verdere uitwerking van *bijdrage 2.* En onderzoek naar de rol hiervan voor 'nieuwbouw'.

FW 3) Onderzoek naar mogelijkheden van andere technieken
Zoals in de conclusie al werd aangegeven is het de vraag of aspectoriëntatie de meest optimale *modularity* en *encapsulation* mogelijk maakt. Het zou interessant zijn om te kijken of andere technieken, zoals genoemd in de *H2 Probleemanalyse* nog verdere verbeteringen zouden kunnen bieden.

FW 4) Onderzoeken generaliseerbaarheid van verbeterde modularity en encapsulation
Zoals in *H7* is aangegeven dient nog gevalideerd te worden dat verbeteringen niet applicatiespecifiek zijn, maar breder toepasbaar. Nu is dat alleen nog aannemelijk gemaakt.

FW 5) Onderzoeken of ontwikkelaars daadwerkelijk een verminderde complexiteit ervaren
Met ontwikkelaars in de praktijk testen of de complexiteit van de broncode daadwerkelijk vermindert is als gevolg van het gebruik van AspectJ.

8.3 Aanbevelingen

Aanpassingen Java
Binnen dit onderzoek is aangegeven (*zie o.a. H4.2.1*) dat enkele problemen aangepakt zouden kunnen worden door de Java programmeertaal aan te passen. Bijvoorbeeld door een oplossing te bieden in de vorm van een *keyword* voor asynchrone methoden (vergelijkbaar met *synchronized*), in plaats van het gebruik van objecten van *anonymous* klassen. Of ontwikkelaars de mogelijkheid bieden zelf *keywords* toe te voegen, dit is nu deels mogelijk met annotaties.

Ontwikkelen van tools voor AO
Er zijn op dit moment nog weinig goede tools voor AO (AspectJ). Het zou aan te bevelen zijn dat er tools ontwikkeld worden voor het meten van AOP metrieken, de huidige tools zijn zeer ongebruiksvriendelijk. Maar ook tools voor het vergelijken van OO en AO implementaties, waarmee een deel van dit onderzoek door tools uitgevoerd zouden kunnen worden.

Bijlage A: Referentielijst

Boeken

[Burns,2001]	***Real-Time Systems and Programming Languages**,third edition,* 2001, Alan Burns en Andy Wellings
[Gamma,1995]	***Design Patterns: Elements of Reusable Object-Oriented Software***, 1995, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
[Goetz,2006]	***Java Concurrency in Practice (Paperback)***, 2006, Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea
[Laddad,2003]	***AspectJ in action, practical aspect-oriented programming***, Ramnivas Laddad, 2003
[Lea,1999]	***Concurrent Programming in Java Second Edition; design principles and patterns,*** 1999, Doug Lea
[McConnell,2004]	***Code Complete, Second Edition***, Steve McConnell, 2004
[Sierra, 2006]	***Sun Certified Programmer for Java Study Guide***, Kathy Sierra, Bert Bates, 2006
[Silberschatz, 2003]	***Operating System Concepts 6e editie***, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, 2003
[Yourdon,1989]	***Modern Structured Analysis***, Edward Yourdon, 1989 (Nederlandstalige druk: 2001)

Papers

[Alexandrescu,2004]	***Lock-Free Data Structures***,2004, Andrei Alexandrescu *(Bijlage D)*
[Armstrong, 1996]	***Erlang - A survey of the language and its industrial applications***, 1996, Joe Armstrong *(Bijlage D)*
[Bader,2007]	***Enabling Programmers to Design Efficient Parallel Algorithms for Many-Core Processors***, 2007, David A. Bader
[Baker, 1977]	***The Incremental Garbage Collection of Processes***, 1977, Henry G. Baker, Jr. and Carl Hewitt *(Bijlage D)*
[Ceccato,2004]	***Measuring the Effects of Software Aspectization***, 2004, Mariano Ceccato, Paolo Tonella
[Chidamber,1994]	***A Metrics Suite for Object Oriented Design***, 1994, Shyam R. Chidamber, Chris Kemerer
[Coffman,1971]	***System Deadlocks,*** 1971, E. G. Coffman, M. J. Elphick, A. Shoshani
[Colyer,2004]	***On the Separation of Concerns in Program Families***, 2004, Adrian Colyer, Awais Rashid, Gordon Blair
[Constantinides, 2002]	***Beyond objects: Improving the modularity of complex software***, 2002, Constantinos Constantinides, Youssef Hassoun
[Cunha,2006]	***Reusable Apect-Oriented Implementations of Concurrency Patterns and Mechanisms***, 2006, Carlos Cunha, João Sobral, Miguel Monteiro
[Dijkstra,1968]	***Cooperating sequential processes (EWD 123),*** 1968, Dijkstra

[Dijkstra,1974]	***On the role of scientific thought***, 1974, Dijkstra
[Fraser,2004]	***Concurrency without locks***, 2004, K Fraser T Harris *(Bijlage D)*
[Herlihy,1993]	***Wait-Free synchronization***, 1993, Maurice Herlihy *(Bijlage D)*
[Herlihy,2007]	***Taking Concurrency Seriously: New Directions in Multiprocessor Synchronization***, 2007, Maurice Herlihy *(Bijlage D)*
[Kiczales, 1997]	***Aspect-Oriented Programming***, 1997, Gregor Kiczales, John Irwin , John Lamping, Jean Marc Cloingtier, Cristina Videira Lopes, Chris Meada, Anurag Mendhekar
[Kiczales,2001]	***An Overview of AspectJ***, 2001, Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm en William G. Griswold
[Kiczales,2005]	***Separation of Concerns with Procedures, Annotation, Advice and pointcuts***, 2005, Gregor Kiczales, Mira Mezini
[Kienzle,2002]	***AOP: Does it make sense ? The Case of Concurrency and Failures***, 2002, Jörg Kienzle, Rachid Guerraoui
[Lea,2000]	***A Java Fork/Join Framework,*** 2000, Doug Lea
[Lee,2006]	***The problem with threads,*** 2006, Edward A. Lee
[Manocha,2007]	***GPGPU to Many-Core Processing:Higher Performance for Mass Market Applications,*** 2007, Dinesh Manocha & Ming C. Lin Naga Govindaraju , *(Bijlage D)*
[Marin, 2006]	***Identifying crosscutting concerns using fan-in analysis***, 2006, Marius Marin, Arie van Deursen and Leon Moonen
[Marin, 2005]	***A Classification of Crosscutting Concerns***, 2005, Marius Marin, Leon Moonen, Arie van Deursen
[Marin, 2005b]	***An Approach to Aspect Refactoring Based on Crosscutting Concern Types***, 2005, Marius Marin, Arie van Deursen and Leon Moonen
[Needham,1979]	***On the Duality of Operating System Structures,*** 1979, Roger M. Needham, Hugh C. Lauer
[O'Brien,2007]	***Following the Maturity Model Thread***, 2007, SD Times, Larry O'Brien *(Bijlage E)*
[Parnas,1972]	***On the Criteria To Be Used in Decomposing Systems into Modules,*** 1972, D.L. Parnas,
[Per Brinch Hansen, 2001]	***The invention of concurrent programming,*** 2001, Per Brinch Hansen *(Bijlage D)*
[Per Brinch Hansen, 1989]	***The nature of parallel programming***, 1989, Per Brinch Hansen *(Bijlage D)*
[Robison,2007]	***Allow Concurrency, Don't Mandate It***, 2007, Arch D. Robison
[Rosenberg,1998]	***Applying and Interpreting Object Oriented Metrics***, 1998, Linda H. Rosenberg
[Samoladas,2004]	***Open Source Software Development Should Strive for even greater code maintainability***, 2004, Loannis Samoladas, Loannis Stamelos, Lefteris Angelis, Apostolos Oikonomou
[Silva, 1997]	***Framework, Design Patterns and Pattern Language for Object Concurrency,*** 1997 António Rito Silva
[Soares, 2004]	***An Aspect-Oriented Implementation Method***, 2004, Sérgio Castelo Branco Soares

[Sutter,2005]	***The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software***, 2005, Herb Sutter
[Sutter,2007]	***The Pillars of Concurrency,*** 2007, Herb Sutter
[Süß,2005]	***Evaluating the state of the art of parallel programming systems,*** 2005 Michael Süß, Claudia Leopold
[Tsang,2003]	***Object Metrics for Aspect Systems: Limiting Empirical Inference Base on Modularity***, 2003, Shiu Lun Tsang, Siobhan Clarke, Elisa Baniassed
[Valois,1995]	***Lock-free linked-list using compare and swap***, 1995, Valois *(Bijlage D)*
[Voelter, 2000]	***Aspect-Oriented Programming in Java***, 2000, Markus Voelter
[Zeichick,2007]	***Presenting the Threading Maturity Model***, 2007, SD Times, Alan Zeichick *(Bijlage E)*

Overig (White-papers, presentaties, webartikelen) (alle urls zijn op 13 augustus gecontroleerd)

[Amarasinghe, 2007]	***Lecture 4: Introduction to Concurrent Programming***, 2007, Saman Amarasinghe, MIT http://cag.csail.mit.edu/ps3/lectures/6.189-lecture4-concurrency.pdf
[AMD, 2005]	***Multi-core processors: the next evolution in computing,*** 2005, AMD http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf
[Fowler,2007]	**Refactoring Home Page**, Martin Fowler http://www.refactoring.com/
[Intel,2005]	***Platform 2015 Software Enabling Innovation in Parallelism for the Next Decade,*** 2005, Intel http://download.intel.com/technology/computing/archinnov/platform2015/download/Parallelism.pdf
[Intel,2007]	***Intel® Multi-Core Processor Architecture Development***, 2007, Intel Corporation http://cache-www.intel.com/cd/00/00/20/57/205707_205707.pdf
[Kent,2007]	***Multi-Core Programming Concepts in Commodity Systems***, Kent, 2007 www.tacc.utexas.edu/cluster2006/Kent_milfeld.ppt
[Laddad,2003b]	***Aspect Oriented Refactoring Series Part 1***, 2003, Ramnivas Laddad www.theserverside.com/tt/articles/article.tss?l=AspectOrientedRefactoringPart1
[Larus,2007]	***Parallel Thoughts***, 2007, James Larus, Microsoft Research http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/Jim%20Larus%20-%20Position%20Paper.ppt
[Mattson,2007]	***Defining the foundations of Programmability Research***, 2007, Tim Mattson, Intel http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/Tim%20Mattson.pp
[O'Reilly,2007]	***Google's Folding@Home on the "Multi-Core Crisis"***, 2007, Tim O'Reilly http://radar.oreilly.com/archives/2007/06/googles_folding.html
[Quinn, 2001]	***Parallel Programming with MPI and OpenMP chapter 7***, 2001, Michael J. Quinn http://polaris.cs.uiuc.edu/~padua/cs420/Chapter7.ppt
[Rabbah, 2007]	***Lecture 5: Parallel Programming Concepts***, 2007, Rodric Rabbah,IBM http://cag.csail.mit.edu/ps3/lectures/6.189-lecture5-parallelism.pdf
[Riske, 2005]	***The Multicore Advantage***, 2005, Sun, Al Riske http://research.sun.com/minds/2005-0902/
[Shah, 2007]	***Interviewing the Parallel Programming Idols***, 2007, Joe Armstrong,

William Gropp, Sanjiv Shah, David Butenhof *(Bijlage D)*, www.thinkingparallel.com/2007/03/14/interviewing-the-parallel-programming-idols/

[Shankland,2006] ***Sun puts 16 cores on its 'Rock' chip***, 2006, Stephen Shankland, http://news.zdnet.com/2100-9584_22-6141961.html

[Suess, 2006] ***Mutual Exclusion with Locks - an Introduction,*** 2006, Michael Suess http://www.thinkingparallel.com/2006/09/09/mutual-exclusion-with-locks-an-introduction

[Sun, 2004] ***New Features and Enhancements J2SE 5.0***, 2004, Sun MicroSystems, http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[Sun, 2006] ***Java EE Application Model, 2006***, Sun MicroSystems, http://java.sun.com/javaee/5/docs/tutorial/doc/

[Sun, 2007] ***Fortress FAQ***, 2007, Sun http://research.sun.com/projects/plrg/faq/index.html *(Bijlage D)*

[Sweeney, 2007] ***The Next Mainstream Programming Language: A Game Developer's Perspective***, Tim Sweeney, 2007 www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/sweeny.pdf

Bijlage B: Begrippenlijst

Add variability — Dit is de classificatie van het objectgeoriënteerde patroon waarbij objecten van *anonymous* klassen gecreëerd worden om een methode (functionaliteit) als parameter te kunnen gebruiken. *[Marin,2005].*

Advice, — Een *advice* bestaat uit de combinatie van het wat (*action*) waar (*pointcut*) en hoe (*around/before/after*) van een concern in AspectJ. *[Laddad,2003b]*

Aspect — Vergelijkbaar met een (*crosscutting*) concern, maar binnen deze scriptie wordt deze term gebruikt voor de implementatie ervan (in AspectJ). *[Kiczales, 2001]*

Aspect Oriented Programming (AOP) — AOP is het ontwikkelen van software in een taal welke gebaseerd is op het AO principe. AspectJ is een voorbeeld van een aspectgeoriënteerde taal. *[Kiczales, 1997] [Kiczales, 2001]*

Aspect Mining — Aspect Mining is het zoeken naar en in kaart brengen van *crosscutting concerns* binnen software. Dit kan zowel *generative* (bijvoorbeeld met een *fan-in* analyse) als op basis van *queries* (waarbij gericht gezocht wordt naar een patroon). *[Marin,2005] [Marin,2006]*

Aspect Oriëntatie (AO) — AO is een (probleem-) analyse paradigma waarbij het probleem wordt opgelost op basis van elementen van het type Aspect. Het levert elementen voor het op een natuurlijke (modulaire) manier implementeren van *crosscutting concerns*, waardoor deze in de code van elkaar gescheiden blijven. *[Kiczales, 1997] [Kiczales, 2001]*

Aspect Refactoring (Aspect Oriented Refactoring) — Het aanpassen van een systeem waarbij de *crosscutting concerns* met aspectgeoriënteerd programmeren omgezet worden in goed gescheiden concerns. De interne structuur van de code wordt aangepast, maar met behoud van het externe gedrag. *[Laddad,2003b] [Fowler,2007][Marin,2005b]*

AspectJ — Een aspectgeoriënteerde programmeertaal (als uitbreiding op Java). *[Laddad,2003]*

AspectJ-stijl — Een AspectJ oplossing kan gebruik maken van verschillende *pointcut* mechanismen, deze pointcut mechanismen bepalen de stijl waarin de AspectJ-code (*pointcut*) geschreven is.

Concern — Een functionele of technische eigenschap van het systeem waarvoor een (of meerdere) element(en) uit het systeem verantwoordelijk zijn. Veelal als onderdeel van één of meerdere requirements. *[Laddad,2003] [Marin,2005] [Silva, 1997] [Colyer,2004]*

Consistent behavior — Wanneer een concern bestaat uit het consistent toevoegen van een rol of gedrag aan een systeem, noemt men dit een *consistent behavior*. Het gedrag moet als een vast patroon van stappen omschreven kunnen worden en de locaties waar het gedrag wordt toegevoegd moeten voorspelbaar zijn. Bijvoorbeeld altijd aan het begin en/of eind van bepaalde methoden*[Marin,2005].*

Concurrency concern — Een concern gerelateerd aan de implementatie van parallelle verwerking.

Crosscutting concern — Een concern dat niet goed ontkoppeld kan worden van andere concerns. Vaak betekent dit dat de implementatie verweven (*tangling*) is met andere concerns en verspreid (*scattering*) is over het systeem. Deze worden ook wel

	system-wide concerns genoemd omdat ze over meerdere modulen verspreid zijn. *[Marin,2005][Laddad,2003]*
Dynamic behavior enforcement	Wanneer het gebruik van methoden van een object door 'regels' gedefinieerd wordt, maar deze niet in de code verwerkt zijn. Bijvoorbeeld door een bepaalde aanroepvolgorde te vereisen voor het gebruik van een aantal methoden (eerst initialization, dan functionaliteit, dan finalization). Alleen zijn deze eisen dan niet in code, maar in bijvoorbeeld commentaar verwerkt *[Marin,2005].*
Encapsulation	De mate waarin de implementatiedetails van een module verborgen zijn voor de modulen waarmee hij communiceert, vaak verborgen achter interfaces *[Parnas,1972] [Silva, 1997].* Wijzigingen in een module zouden niet tot wijzigingen in een ander module moeten leiden.*[Yourdon,1989]*
Joinpoint	Een locatie in de broncode, binnen AspectJ kan dit op klasse en methode niveau. *[Laddad,2003b]*
Modularity	De mate waarin het systeem de verantwoordelijkheden voor taken heeft ondergebracht in elementen/modulen die ontkoppeld zijn van de overige elementen/modulen. *[Parnas,1972][Silva, 1997]* De functionaliteit in een module dient toe te behoren aan één taak, en één taak zou binnen één module moeten worden ingevuld.*[Yourdon,1989]*
Module	Gegroepeerde verzameling van codestatements, bijvoorbeeld in de vorm van klassen of aspecten.
Multi-core processor	Is een processor waarbij meerdere CPU-cores in één CPU-*package* geplaatst zijn. Deze kunnen verschillende taken tegelijkertijd uitvoeren.
Multi-threaded proces	Een proces bestaande uit meerdere threads.*[Silberschatz, 2003]*
Package	Verpakking waarin processorchips geplaatst worden. Deze *package* wordt in de slots/sockets van het moederbord geplaatst.
Parallel programmeren	Het ontwikkelen van systemen welke beschikken over parallelle verwerking.*[Süß,2005]*
Parallelle verwerking	Een systeem beschikt over parallelle verwerking wanneer er meerdere taken tegelijk uitgevoerd kunnen worden. *[Nakhimovski, 2001]*
Pointcut	Eén of meerdere joinpoints vormen samen een pointcut. *[Laddad,2003b]*
Proces	Een programma in executie. In principe beschikken processen ieder over hun eigen resources.*[Silberschatz, 2003]*
Scattering	Is het verspreid zijn van de implementatie van een concern over meerdere modulen. Als gevolg hiervan kan het zijn dat dezelfde implementatiecode in meerdere modulen terugkomt(duplicaten), of dat iedere module verschillende code omvat om samen het doel te bereiken(complementair). Dus is het lastig te overzien waar het concern geïmplementeerd is.*[Laddad,2003]*
Single-core processor	De processor die in 'traditionele' desktop computers is terug te vinden. Deze bezit over één CPU-core in zijn eigen *package*.
Sort	In *[Marin,2005]* is een classificatie opgesteld van veel voorkomende *crosscutting concern* typen. Zo'n cross-cutting concern type wordt een *sort* genoemd.

Tangling Is het verweven zijn van de implementatie van meerdere concerns in één module. Hierdoor is het lastig aan te wijzen wat het doel is van een specifiek statement in de code. Dus is het lastig te bepalen tot welk concern hij behoort. *[Laddad,2003]*

Thread Onderdeel van een proces. Threads van één proces delen hun code-, datasectie en overige resources. Dit maakt onderlinge communicatie relatief goedkoop met betrekking tot performance*. [Silberschatz, 2003]*

Weaving AOP maakt het mogelijk concerns gescheiden van elkaar te beschrijven. Echter tijdens het uitvoeren dienen deze concerns weer automatisch te worden samengevoegd om dezelfde functionaliteit te behouden. Dit samenvoegen van concerns wordt Weaving genoemd en kan zowel tijdens het uitvoeren als het compileren.*[Kiczales, 1997]*

Bijlage C: Gebruikte tools

Deze paragraaf geeft een overzicht van de tools die gebruikt zijn bij het uitvoeren van het onderzoek dat in deze scriptie beschreven is. De tools zijn verdeeld over twee categorieën. Hierbij is kort aangegeven wat het is en waar het voor gebruikt is. Om het onderzoek te kunnen reproduceren zou men dus deze tools kunnen gebruiken.

Aspect Mining en Refactoring

- **Eclipse 3.2 voor Windows , icm Java SE 1.6**
  Deze Java IDE is gebruikt voor zowel het uitvoeren van de analyse als de refactoring.

- **AspectJ 1.5 (Eclipse AJDT)**
  AspectJ en de *AspectJ Development Tools* zijn gebruikt bij de experimenten om de *modularity* en *encapsulation* te verhogen*(refactoring)*. Daarnaast is de *Aspect Weaver* gebruikt voor het in beeld brengen van concern *scattering (*analyse*)*.

- **SoQuet 0.2**
  Met deze tool is een concern-model opgesteld. Hiermee zijn de concerns dus gedocumenteerd.

- **Sort (search)**
  Deze tool is ingezet bij het documenteren en analyseren van de *sorts*

- **FINT 0.6**
  Deze tool maakt het eenvoudig om een *Fan-in* analyse te maken van het systeem. Deze is gebruikt om *seeds* (mogelijke concerns) te vinden in het onderzochte systeem.

- **Ultra-edit 13.00**
  Deze tool is gebruikt om de broncode te doorzoeken, voor het in kaart brengen van concern *scattering*.

Evaluatie tools

- **AOPmetrics**
  AOPmetrics maakt het mogelijk om metrieken van AspectJ code te verkrijgen. Het is echter geen eenvoudige tool om werkend te krijgen.

- **Metrics Plugin 1.3.6 (metrics.sourceforge.net)**
  Deze plugin heeft een bijdrage geleverd door de meeste standaard OO-metrieken te verzamelen. Het is een tool waarmee metrieken voor code complexiteit van Java software kunnen worden verzameld. De implementatie van de tool is gebaseerd op *"Object-Oriented Metrics, measures of Complexity" by Brian Henderson-Sellers, Prentice Hall, 1996.*

- **Analyst4J**
  Deze plugin heeft een bijdrage geleverd door de meeste standaard OO-metrieken te verzamelen.

Bijlage D: Appendix introductie parallel programming

Binnen deze bijlage is achtergrond informatie te vinden van enkele paragraven uit de *sectie 4.1 Introductie parallel programming*

9.1 Mogelijkheden voor prestatiewinst met multi-core CPU's

Wanneer software van parallelle verwerking voorzien wordt is het mogelijk om flinke prestatiewinst te realiseren voor multi-core CPU's. Er is echter wel een beperking op de mogelijke prestatiewinst. Helaas kan er meestal geen lineaire snelheidswinst behaald worden bij het parallelliseren van software. De oorzaak hiervoor is het bestaan van inherent seriële operaties, communicatie overhead, *process startup* kosten, onevenwichtige workload verdeling en eventuele architectuur beperkingen. *[Amarasinghe, 2007] [Lea,1999] [Quinn, 2001][Goetz,2006]*

Sommige taken kunnen bovendien gewoon niet sneller uitgevoerd worden. Een bekend voorbeeld is de observatie van Brooks *"The bearing of a child takes nine months no matter how many women are assigned"*. Waarmee wordt aangegeven dat de doorlooptijd van dit proces niet korter kan. Echter wanneer het doel is zoveel mogelijk kinderen te maken, helpt het toevoegen van vrouwen wel.

Om de mogelijke prestatiewinst te beschrijven zijn een aantal modellen/'wetten' opgesteld waarvan Amdahls Law de bekendste is *[Amarasinghe, 2007] [Quinn, 2001]*

> *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*

De volgende formule kan gebruikt worden voor het berekenen van de te behalen optimalisatie. De variabelen betekenen hierin het volgende: *f* percentage sequentiële code, *p* aantal cores

$$\psi \leq \frac{1}{f + (1-f)/p}$$

Voorbeeld als 95% van een applicatie parallel kan worden uitgevoerd over 8 CPU-cores dan is de optimale prestatiewinst: 1 / (0,05 + 0.95 / 8) = +- 6x *[Quinn, 2001]*

Andere, minder bekende, methoden voor het in kaart brengen van schaalbaarheid zijn *Karp-Flatt Metric* en de *Gustafson-Barsis' Law*. *[Quinn, 2001]*

9.1 Communicatie-modellen

De volgende paragrafen zullen ingaan op de twee standaarden voor expliciete communicatie: *Messages Passing* en *Shared Memory*. Daarnaast zijn er ook nog impliciete vormen van communicatie zoals *futures* / *promises*.

9.1.1 Message Passing vs Shared Memory

Message passing:
Bij deze communicatievorm communiceren de processen met behulp van berichten. Deze berichten verspreiden de informatie meestal als kopie van het origineel. De berichten infrastructuur zorgt ervoor dat het gebruik van de gedeelde informatie duidelijk (expliciet) zichtbaar is. *[Needham,1979][Burns,2001]*

Shared Memory:
Bij deze communicatievorm delen de processen het ge-alloceerde geheugen. Hierdoor kunnen ze ieder wanneer ze maar willen dit geheugen aanspreken en delen ze de informatie dus direct met elkaar. Doordat het gebruik hier impliciet is, is het minder duidelijk wie wanneer de gedeelde informatie gebruikt. *[Needham,1979] [Burns,2001]*

Vergelijking
Welke van deze vormen beter is, is een groot strijdpunt zonder echte winnaar. Binnen *[Needham,1979]* wordt beargumenteerd dat beide modellen dermate veel gelijkenis vertonen dat ze vrijwel altijd inwisselbaar zijn.
Maar meestal wordt *shared memory* toegepast binnen applicaties op één machine omdat dit minder overhead omvat en sneller is. *Message passing* wordt vaker toegepast binnen applicaties die over meerdere systemen samenwerken. Al zijn voor beide ook uitzonderingen te vinden op deze regel.
De hardware en de architectuur spelen dus een grote rol bij het selecteren van het communicatiemodel. Daarnaast bepaald de omvang van de te delen objecten (die de *shared state* vormen) soms het model. *Message passing* verdient de voorkeur bij een *shared state* die gedefinieerd kan worden als kleine berichten, grote objecten (grote *shared state*) ligt beter voor het *shared memory* model. *[Needham,1979]*

Meer over deze communicatievormen is te vinden in *[Burns,2001][Needham,1979] [Silberschatz,2003]*

Voorbeeld toepassingen
Voor applicaties zoals SETI@home[3] geldt dat *message passing* goed past. Er is sprake van een kleine *shared state*, de rekeneenheden kunnen goed afzonderlijk hun werk doen. Voor applicaties zoals games, webbrowsers, en office-applicaties geldt dat *shared memory* een betere optie is omdat deze een grotere *shared state* bezitten. *[Shah, 2007]*

Een variant op *shared memory* is STM, *Software Transactional Memory*. In hoofdstuk 9.2.3 zal hier meer op ingegaan worden

9.1.2 Futures / promises

Een *Future* is een hulpmiddel voor het asynchroon uitvoeren van (langdurige)bewerkingen. Het stelt de ontwikkelaar instaat om de bewerking asynchroon uit te laten. Hierdoor kan het hoofdprogramma verder zonder te wachten op de langdurige bewerking. Hiervoor wordt een Proxy gebruikt naar het object dat terugkomt uit de bewerking.

Nu zijn er twee situaties mogelijk: het programma heeft het object uit de bewerking nodig *voordat* deze bewerking klaar is of *nadat* de bewerking klaar is. Wanneer de bewerking al klaar is wordt het resultaat direct via de Proxy beschikbaar gesteld. Wanneer de bewerking nog niet klaar is zal alsnog gewacht moeten worden door het hoofdprogramma totdat de asynchrone bewerking afgerond is. Echter doordat deze wel al enige tijd loopt is er een tijdwinst behaalt. Mits er sprake is van parallellisme in de hardwarearchitectuur, zoals een multi-core CPU. Deze vorm van communicatie / synchronisatie is meer impliciet dan de vorige. De ontwikkelaar geeft hierbij geen expliciete opdracht tot synchronisatie waardoor het abstractieniveau iets hoger is dan bij de vorige communicatie modellen. *[Baker, 1977][Lea,1999]*

[3] http://setiathome.berkeley.edu/

9.2 Hulpmiddelen voor consistent en correct houden shared state

In de onderstaande paragraven komen technieken naar voren voor het beschermen van *critical regions*.

9.2.1 Mutex, Locks, Semaphoren, Monitoren

De termen uit de titel van deze paragraaf zijn de meest gebruikte constructies voor het inrichten van *mutual exclusion*. Hieronder staan ze opgesomd met een korte beschrijving van hun kernmerken.

Mutex,
Een *mutex* is een term die meerdere betekenissen gekregen heeft. Oorspronkelijk is *mutex* een afkorting van *mutual exclusion,* de theorie van het afschermen van gelijktijdige toegang tot een *resource*. Echter het woord wordt vaak ook gebruikt voor implementaties van deze theorie bijvoorbeeld als synoniem van semaphoren. *[Per Brinch Hansen, 2001] [Silberschatz, 2003] [Suess, 2006] [Lea,1999]*

Lock,
Verzamelnaam voor het mechanisme voor het controleren van toegang tot een resource. Het mechanisme van *Locks* is dus bruikbaar voor het realiseren van *mutual exclusion*. Bij de meeste *lock* implementaties geldt dat de threads in een slaapstand terechtkomen wanneer zij moeten wachten op een *resource* die bezet is. Hierdoor wordt zo min mogelijk rekenkracht verspilt. Het alternatief zou zijn om continu toegang te vragen wat rekenkracht kost. *[Per Brinch Hansen, 2001] [Silberschatz, 2003] [Suess, 2006] [Lea,1999]*

Er zijn verschillende typen *locks*, de bekendste zijn hieronder opgesomd en kort beschreven, uiteraard zijn ook combinaties van typen mogelijk:

- *Spinlocks,*
  Bij deze type *locks* komen de *threads* niet in een slaapstand, maar blijven deze hun verzoek tot toegang herhalen, ook wel *polling / busy waiting* genoemd. Het voordeel hiervan is dat mogelijk sneller gereageerd wordt op het vrijkomen van de *resource*. *[Silberschatz, 2003]*

- *Re-entrant lock*
  Bij sommige *lock*-typen ontstaat een probleem wanneer een *thread* twee keer dezelfde *lock* probeert te verkrijgen. Immers de tweede keer is deze al bezet en zal hij in de wachtstand terecht komen. Echter de *lock* zal nooit vrijgegeven worden, omdat hij hem immers zelf bezit en zelf in de slaapstand terecht is gekomen. De *Re-entrant lock* kent dit probleem niet, wanneer hij de tweede *lock* aanvraag krijgt controleert hij of dat de eigenaar niet toevallig dezelfde is die de aanvraag doet, zodat hij in dat geval verder kan. Dit wordt ook wel *recursive locking* genoemd. De meeste moderne talen (Java bijvoorbeeld) en bibliotheken gebruiken deze variant. *[Suess, 2006] [Lea,1999]*

- *Read/write lock*
  Soms bestaat de situatie dat meerdere threads wel tegelijk leestoegang tot een *resource* mogen krijgen, maar dat alleen het schrijven exclusief dient te zijn. In dat geval is een read-*write lock* een oplossing, hierbij kan bij het aanvragen van de *lock* worden aangegeven of het om lees- of schrijftoegang gaat en kunnen meerdere lezers tegelijk toegang krijgen. In applicaties waar veel threads simultaan leestoegang nodig hebben en slechts af en toe schrijftoegang kan deze oplossing een betere performance leveren. *[Suess, 2006] [Lea,1999]*

- *Timed Locks*

Dit zijn *locks* waarbij de aanvraag van een *time-out* voorzien wordt. Op het moment dat de *time-out* verstreken is zal de thread stoppen met wachten ongeacht het vrijkomen van de *lock*. *[Suess, 2006] [Lea,1999]*

Semaphore

De klassieke methode voor het afschermen van toegang tot een resource, bedacht door Dijkstra *[Dijkstra,1968].* De waarde van de *semaphore* wordt vooraf ingesteld op het aantal processen dat tegelijkertijd toegang mag krijgen en biedt twee methoden *verlaag* en *verhoog.* Verlagen wordt gebruikt bij het verkrijgen van toegang, verhogen bij het vrijgeven van de toegang. Wanneer de waarde op 0 staat zal de aanvragende thread in de wachtstand terechtkomen, totdat een andere thread hem weggeeft. Deze vorm van *semaphores* wordt *counting semaphores* genoemd, een speciaal geval hiervoor is de *binary semaphore* waarbij het aantal processen dat toegang krijgt één is. *[Dijkstra,1968] [Silberschatz, 2003]*

Monitor

Monitors zijn met conditie variabelen beschermde *critical regions*. In feite is dit dus een combinatie van *semaphore* en *critical region*, waarbij de conditie variabele bepaald wat de waarde van de semaphore is en of de *critical region* betreden mag worden. Dit idee is oorspronkelijk van Hoare, maar er zijn naderhand meerdere implementaties/variaties op dit idee ontstaan en geïmplementeerd. *[Per Brinch Hansen, 2001]*

9.2.2 Lock Free/Wait Free

Eerder is al reeds beschreven dat het noodzakelijk is om maatregelen te nemen voor het consistent houden van de gedeelde data. Hierbij is aangegeven dat *locking* ingezet kan worden voor het beveiligen van data. Een nadeel is dat *locking* een pessimistische insteek heeft bij het beschermen van data. Het gaat er van uit dat de data beveiligd *moet* worden, doormiddel van het uitsluiten van anderen en hun te laten wachten (*blocking*), terwijl dit niet altijd het geval is. Soms is het ook mogelijk om het probleem op te lossen zonder dat threads toegang wordt ontzegt en ze dus gewoon verder kunnen met hun verwerking (*non-blocking*).
Blocking heeft dus als gevolg dat soms onnodige wachtrijen ontstaan voor het verkrijgen van toegang tot de gedeelde data. *[Valois,1995]*

Daarom zijn er alternatieven: *Lock free & wait free.*

Lock-free.

Zoals de naam al aangeeft wordt bij deze techniek geen gebruik gemaakt van *locking* om data te beveiligen. Vaak met als doel om de processen of threads *non-blocking* te laten zijn. Dit houdt in dat een vertraging in een proces geen invloed kan hebben op de verwerkingstijd van een ander proces. *[Valois,1995]*
Dit heeft onder andere tot gevolg dat *liveness* beter voorspelbaar wordt. Immers wanneer processen *non-blocking* zijn, kan geen *deadlock* optreden. Bovendien zou de performance en schaalbaarheid ook ten goede kunnen komen. Bijvoorbeeld doordat twee entiteiten tegelijkertijd een niet overlappend deel geheugen (stack, queue) kunnen aanpassen. *[Fraser,2004]*

Wait-Free

Wait-Free is een variant op *lock-free* waarbij ook nog eens gegarandeerd kan worden dat de processen binnen een eindige tijd zullen worden afgerond. Echter deze eisen maken het geheel nog ingewikkelder. *[Valois,1995] [Herlihy,1993]*

Voor- en nadelen

De voordelen van *lock-free* en *wait-free* zijn dat deze methoden enkele van de standaard problemen met het gebruik van concurrency control constructs niet kennen, zoals *deadlocks* en *priority inversion [Alexandrescu,2004] (deze problemen worden toegelicht in 9.3).*
Het nadeel is dat er complexe algoritmen en datastructuren moeten worden bedacht om te voorkomen dat de data in inconsistente staat terecht komt. Het opstellen van dit soort algoritme is echter een zeer complexe zaak, vele malen complexer dan met locking. *[Sutter,2005]*

Hardware
Lock-free implementaties maken vaak gebruik van een CAS (*Compare-and-swap) of MCAS(multi-word Compare-and-swap)* constructie. Dit is een actie waarbij eerst gecontroleerd wordt of de huidige waarde van een element gelijk is aan de verwachte waarde (*compare*) en wanneer dit het geval is zal de waarde vervangen worden (*swap*). Voor echt efficiënte *lock-free* of *wait-free* constructies is de aanwezigheid van CAS (en MCAS) erg belangrijk. *[Valois,1995] [Fraser,2004]*

9.2.3 STM

STM, *Software Transactional Memory* wordt door sommige gezien als *de oplossing* voor de concurrency implementatie problemen *[Sweeney, 2007] . Software transactional memory* is een concept waarmee *shared memory* operaties gegroepeerd kunnen worden in atomaire transacties. De term is geïntroduceerd door *[Herlihy,1993].*

De reden voor de introductie van STM is dat *threads* en *locking* volgens Herlihy niet goed samen gaan, ze schalen niet goed genoeg op. Het idee is dat een transactionele API het bovendien eenvoudiger zou moeten maken om met concurrency om te gaan. Zie ook de database wereld, ontwikkelaars lijken minder moeite te hebben met de concurrency binnen databases dan binnen *shared memory*. Dit is doordat men de synchronisatie op een hoger abstractie niveau kan plaatsen, niet meer op het fysieke niveau van *locks*, maar op het meer hogere niveau van (samengestelde) acties/transacties.

Voor de implementatie wordt gebruik gemaakt van *optimistic concurrency control*, zoals ook bij databases voorkomt. De threads mogen tegelijkertijd voortgang maken en kunnen na afloop controleren of de transactie doormag gaan(*commit*) of afgebroken dient te worden (*abort*). Dit wordt vaak gerealiseerd met *lock-free* datastructuren.

Maar zijn ook nadelen, ten eerste is het zo dat algoritmen die ontworpen zijn voor locking concurrency control, niet altijd goed werken bij een STM implementatie. Dit betekent dat bestaande algoritmen aangepast moeten worden.
En er is een kans dat threads/transacties wel voortgang maken, maar telkens opnieuw uitgevoerd moeten worden doordat er iedere keer een rollback plaats moet vinden *[Herlihy,2007].*

9.3 Gevaren en problemen

De volgende paragrafen geven inzicht in een aantal van de problemen die op kunnen treden bij het gebruik van concurrency

9.3.1 Deadlocks

Deadly embrace (*deadlock*) is het probleem dat sommige deelprocessen nooit de kans krijgen om hun verwerking te voltooien, tenzij de andere processen gestopt worden. *[Dijkstra,1968]*

De volgende voorwaarden (*Coffman conditions*) moeten gelden voor het optreden van een deadlock:

- *Mutual Exclusion (*er is een gedeelde bron die maar door één proces tegelijkertijd benaderbaar is, de andere moeten wachten*)*
- *Hold and wait* (minimaal één van de processen moet in het bezit zijn van één of meerdere bronnen en wachten op toegang tot een andere bron)
- *No Pre-emption* (de bronnen komen alleen vrij wanneer de huidige eigenaar hem vrijgeeft, ze kunnen niet worden opgeist door een proces)
- *Circular wait* (Er is een set processen waarvoor geldt ze een bron bezitten die de ander wilt en omgekeerd, een bron willen die de ander heeft.)

[Silberschatz,2003] [Coffman,1971]

Bijvoorbeeld: Proces A vraagt resource 1 en 2, Proces B vraagt resource 2 en 1. Uiteindelijk krijgen beiden hun eerste resource toegewezen en kunnen ze niet verder omdat ze daarvoor de resource nodig hebben die door het andere proces bezet gehouden wordt.

Het elimineren van één of meer van deze voorwaarden kan *deadlocks* voorkomen/onmogelijk maken. *[Amarasinghe, 2007] [Silberschatz,2003]*

9.3.2 Livelock

Een variant op de *deadlock* is de *live-lock.* Het verschil is dat bij *deadlock* alle processen staan te wachten en niet verder kunnen, bij een *live-lock* kunnen de processen nog wel verder maar bereiken nooit hun uiteindelijk doel, ze gaan 'zinloos' verder. Bijvoorbeeld wanneer twee mensen in de gang staan en allebei uit beleefdheid opzij stappen, maar in dezelfde richting waardoor eenzelfde situatie ontstaat. De personen staan niet op elkaar te wachten maar blijven bewegen, alleen ze komen niet bij het eind van de gang omdat ze in elkaars doorgang blijven staan *[Amarasinghe, 2007]*

9.3.3 Race Conditions

Een *race-condition* is het optreden van een schrijfactie van een thread die er voor zorgt dat de *shared-state* inconsistent wordt. De processen delen samen een stuk geheugen en door samenloop van omstandigheden kunnen schrijfacties plaatsvinden op verkeerde momenten wanneer de *critical regions* niet goed beschermd worden.
Bijvoorbeeld als een actie transactioneel / atomair zou moeten zijn, maar dit niet is. Het standaard voorbeeld hierbij is dat twee processen een banksaldo aflezen, wijzigen en terugschrijven. Eén van de twee wijzigingen raakt hierbij verloren omdat de resultaten van de eerste wijzinge overschreven worden.
[Dijkstra, 1968]

9.3.4 (Un)fairness

Fairness is de mate waarin de processen ieder in de juiste mate en met de juiste frequentie toegang krijgen tot de CPU. *Unfairness* treedt dus op wanneer een proces minder toegang krijgt tot de processor dan dat hij recht op heeft, of te lang moet wachten voordat hij eindelijk toegang krijgt. Dit hoeft uiteraard niet noodzakelijk te betekenen dat ieder proces evenveel toegang krijgt, sommige processen hebben nu eenmaal *meer* of *eerder* toegang nodig.

Twee specialisaties hiervan zijn: het nooit uitgevoerd worden van een deelproces (*Starvation, [Silberschatz,2003] [Amarasinghe, 2007]*) en et verkeerd verdelen van de *workload (Unfair Loadbalancing, [Per Brinch Hansen,1989]*).

9.3.5 Priority inversion

Een thread met lage prioriteit kan zich in een *mutual exclusion block* bevinden en daarmee een hoge prioriteit thread blokkeren. De hoge prioriteit thread krijgt dan voorrang van de *scheduler*, maar heeft hier niks aan omdat hij moet wachten op een thread met lage prioriteit. Het resultaat is dat de thread met lage prioriteit in de praktijk voorrang heeft op een thread met hoge prioriteit, of te wel *priority inversion*. *[Alexandrescu,2004]*

9.3.6 Contention convoying

Contention convying (*lock contention)* is het verzamelen van wachtende threads rondom een code-blok dat beveiligd wordt met een *mutual exclusion*. Wanneer de granulariteit van de beveiligde code groter wordt, dus grote lappen code in *mutual exlcusion* blokken, dan is de kans groter dat threads op elkaar wachten. Immers threads zullen dan meer tijd doorbrengen in dit stuk code en dus moeten de andere threads langer wachten. *Contention convoying* heeft hiermee te maken, het houdt in dat veel threads wachten om dezelfde *critical region* binnen te gaan, omdat andere threads deze lang bezet houden. Er ontstaat dan een wachtrij. Kleinere *mutual exclusion* voorkomt dat deze lang bezet gehouden worden maar heeft als nadeel dat er meer processortijd verloren gaat. Het realiseren van een mutual exclusion kost uiteraard ook klok-cycli. *[Fraser,2004]*

9.4 Java

Java en concurrency
Het Java-platform kent al vanaf het ontstaan hulpmiddelen voor concurrency. Met het uitbrengen van versie 1.5 van het framework zijn hier nog verschillende nieuwe elementen aan toegevoegd, op basis van enkele bekende concurrency *design patterns* zoals *latches* en *futures*. Bovendien zijn ook enkele *non-blocking* datastructuren toegevoegd. *[Sun, 2004][Lea, 1999][Goetz,2006]*
Daarnaast is de verwachting dat met Java 1.7 weer een nieuw onderdeel voor concurrency wordt toegevoegd, het zogenaamde fork/join framework, waarmee enkele bewerkingen op standaard datastructuren zoals arrays eenvoudiger parallel uitgevoerd kunnen worden.*[Lea, 2000]*

Een nadeel van Java ten opzichte van bijvoorbeeld Erlang*[Armstrong, 1996]* is dat er veel werk nodig is om de *shared state* te beschermen. Eén van de gevaarlijke kenmerken van Java is dat iedere thread in principe iedere *state* van iedere andere thread kan aanpassen, wanneer deze niet afgeschermd wordt. Daarnaast is er geen compile time controle voor het detecteren van *race-conditions* en *deadlocks*, of hulp hierbij. Dit zijn kenmerken van Java die verbeterd zouden moeten worden.*[Sweeney,2007][Goetz,2006]*

Multi-core en Java
Deze opdracht richt zich op parallellisme in de vorm van gelijktijdige executie van threads op meerdere cores van een multi-core CPU. Het is daarom interessant om te weten hoe threads in Java door het besturingssysteem worden toegewezen aan de fysieke cores.

Bij Java bepaalt de JVM uiteindelijk via het OS door middel van *scheduling* welke thread op welke *core* wordt uitgevoerd. Hier heeft de Java-ontwikkelaar geen controle over, o.a. omdat hij communiceert via de JVM. De combinatie van OS en JVM bepaalt de *scheduling* over de cores, niet de ontwikkelaar. Wat hij wel kan doen is prioriteiten toekennen aan threads om de *scheduling* te sturen *[Silberschatz, 2003]*.

Bijlage E: Context van de opdracht

Een korte beschrijving van de context van opdracht. Hierin komt naar voren hoe de opdrachtgever met concurrency omgaat.

10.1 Kennis van multi-threading en concurrency van collega's

De opdracht wordt uitgevoerd bij de Java Competence van LogicaCMG Rotterdam. Binnen dit hoofdstuk wordt over LogicaCMG gesproken vanuit het oogpunt van de collega-ontwikkelaars uit deze afdeling. Het gaat om ervaren Java ontwikkelaars, met momenteel een basiskennis van zaken als threading en concurrency. De meeste ontwikkelaars hebben weinig hands-on ervaring met concurrency en kennen concurrency/parallellisme vooral vanuit het oogpunt van webapplicaties. Waarbij de webserver het concurrency aspect invult door de requests een eigen thread in de webserver te geven. Echter uit deze beschrijving blijkt ook dat de ontwikkelaars hier niet zelf in aanraking komen met het parallelle karakter en hier mogelijk ook niet van bewust zijn.

Binnen de *literatuur* zijn modellen te vinden*[Zeichick,2007] [O'Brien,2007]* voor het indelen van individuen of bedrijven op een schaal voor adoptie van concurrency in de development omgeving. Echter dit zijn geen algemeen geaccepteerde modellen zoals *Capability Maturity Model (CMM)* van het *Software Engineering Institute (SEI)* dat is voor procesverbetering. Desondanks geven ze wel een indicatie van het niveau van adoptie.

Het *Threading Maturity Model [Zeichick,2007]* bevat een indeling op zes niveaus (0-5). Hieronder is een beschrijving van het level 1 en level 2 niveau opgenomen.

> **Level 1. Awareness.** General awareness of the potential benefits of threading desktop or server applications, but unsure of the specific benefits of different techniques for threading. Threading not incorporated into the development process. Developers trust that compilers, libraries and runtimes are handling threading automatically. No serious consideration of threading as a way to solve performance problems. No testing of applications against platforms with different cores/processors to identify runtime issues.
>
> **Level 2. Experimentation.** Some developers have studied threading, such as by reading articles, but are not trained. Simple tests of threading conducted with trivial or ad-hoc projects; some tests will appear to indicate failure, because the tests weren't properly designed or executed. Some allocation of enthusiast team members' time is channeled into exploring threading, often as a "skunkworks" project. Some understanding of the different threading models. Minimal incorporation of threading into production code. No testing of applications against platforms with different cores/processors to identify runtime issues.

LogicaCMG bevindt zich op het *awareness* niveau van het *Threading Maturity Model (ThMM).* De meeste ontwikkelaars weten wel enigszins dat sommige taken parallel uitgevoerd zouden kunnen worden, maar vertrouwen vrijwel volledig daarvoor op de mogelijkheden van de gebruikte frameworks en besteden er zelf geen of nauwelijks aandacht aan. [4]
Voor een hoger niveau zouden meer ontwikkelaars moeten experimenteren met threading. Nu zijn er slechts enkelen die dit doen. Maar er zijn geen ontwikkelaars die dit dragen binnen de competence.

Wanneer we de beschrijving van het hierop volgende level bekijken kunnen we vaststellen dat deze niet van toepassing is op de Java competence van LogicaCMG. Binnen dit deel van de organisatie wordt nauwelijks geëxperimenteerd met threading en de leden zijn nauwelijks op de hoogte van de ontwikkeling op het gebied van multi-threading in Java. De kennis blijft beperkt tot het gedeelte dat benodigd is voor het behalen van de Sun-certificatie.

[4] Zoals al eerder aangegeven is dit beeld op basis van de Java competentie, de competentie Technical software engineering zal zich op een hoger level bevinden

10.2 Systemen

Het merendeel van de applicaties die ontwikkeld worden zijn webapplicaties. Echter daarnaast worden er ook grotere systemen ontwikkeld voor bijvoorbeeld *backends* bij de NS en DSM. Het kenmerkende van deze systemen is dat ze vaak op een J2EE (Enterprise Beans) architectuur gebaseerd zijn.

Voorbeeldsystemen:

- Communicatiesystemen
- RFID Mobile ticketing
- Digitale brievenbus
- Internet-betalen Postbank
- Reisinformatiesystemen NS
- Cell-broadcast waarschuwingensysteem (milieugevaar, terreurdreigingen etc.)
- Digitaal Factureren.
- Desktop applicaties zoals voor Wehkamp

Deze informatie over de ontwikkelde systemen is afkomstig van consultants binnen LogicaCMG en het 'LogicaCMG Management Consulting Jaarbericht 2006'.

10.3 Visie op multi-threading

LogicaCMG ziet in dat op termijn de rol van concurrency als gevolg van de intrede van de multi-core processor niet genegeerd kan worden. Daarom zal er in de komende jaren meer aandacht besteedt worden aan dit onderwerp, o.a. in de vorm van afstudeeropdrachten.

Van de motivaties die benoemd zijn in *(H4.1.3 Motivaties voor concurrency)* is 'vergroten performance' de grote drijfveer voor de interesse vanuit het bedrijf. Wat ook voor de hand ligt, aangezien de opkomst van multi-core processoren de *trigger* is van de interesse.

Hierbij zal in het begin vooral aandacht zijn voor multi-threading bij het opzetten van nieuwe systemen. Waarschijnlijk eerst de kleinere en naarmate de ervaring met dit onderwerp groeit, ook de grotere systemen.
Daarnaast zou men ook bij lopende projecten mogelijk multi-threading in kunnen voeren wanneer blijkt dat er sprake is van performance problemen (die met concurrency opgelost kunnen worden). Hierbij zou men dan met behulp van *profiling* tools de bottlenecks kunnen opzoeken en gericht concurrency kunnen invoeren.

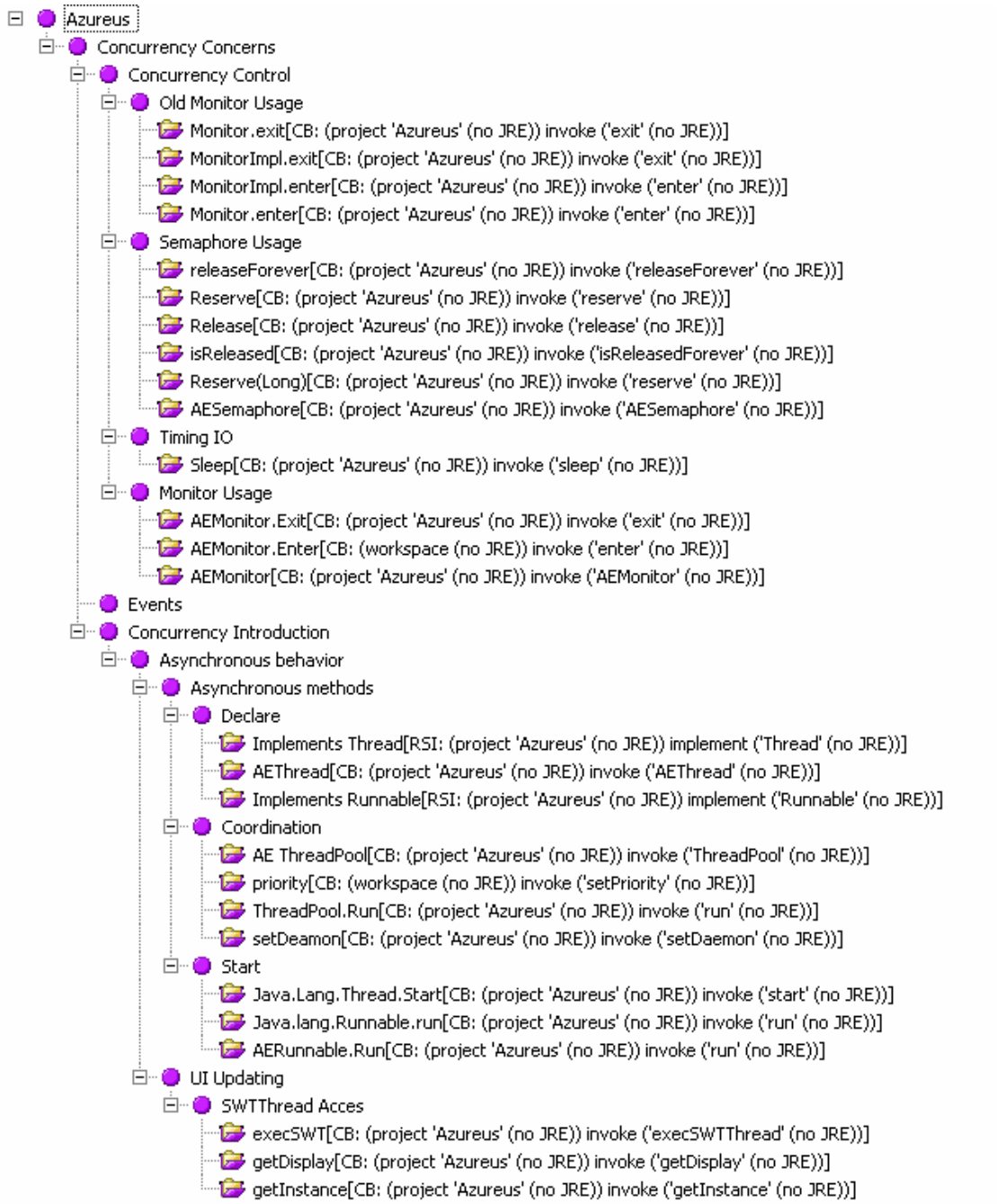
Om dit alles goed te laten verlopen zal scholing van de ontwikkelaars plaats moeten vinden, maar ook zal er de juiste *tooling* aangeschaft moeten worden. Op basis van de gesprekken en de literatuurstudie blijkt dat hierbij met name tooling voor het documenteren van de multi-threading implementatie van belang is. De reden is dat deze documentatie een grote rol zou spelen bij het beheersbaar houden van de problematiek van concurrency zoals beschreven in *H9.3 Gevaren en problemen*. Bijvoorbeeld door het in kaart brengen van de locaties waar concurrency speelt en welke maatregelen genomen zijn om foute verwerking (als gevolg van bijvoorbeeld *race-conditions*) te voorkomen. Daarnaast zou men ook hulp willen bij het testen van het systeem na het uitvoeren van onderhoud.

10.4 Samenvatting

Binnen dit hoofdstuk is uiteengezet wat de context is waarin de afstudeeropdracht uitgevoerd wordt. Hierbij hebben we gezien dat de opdrachtgever nog weinig ervaring heeft met het bewust implementeren van concurrency, maar hier tegelijkertijd wel een toekomst voor ziet.

Bijlage F: Concurrency model van de onderzochte applicatie

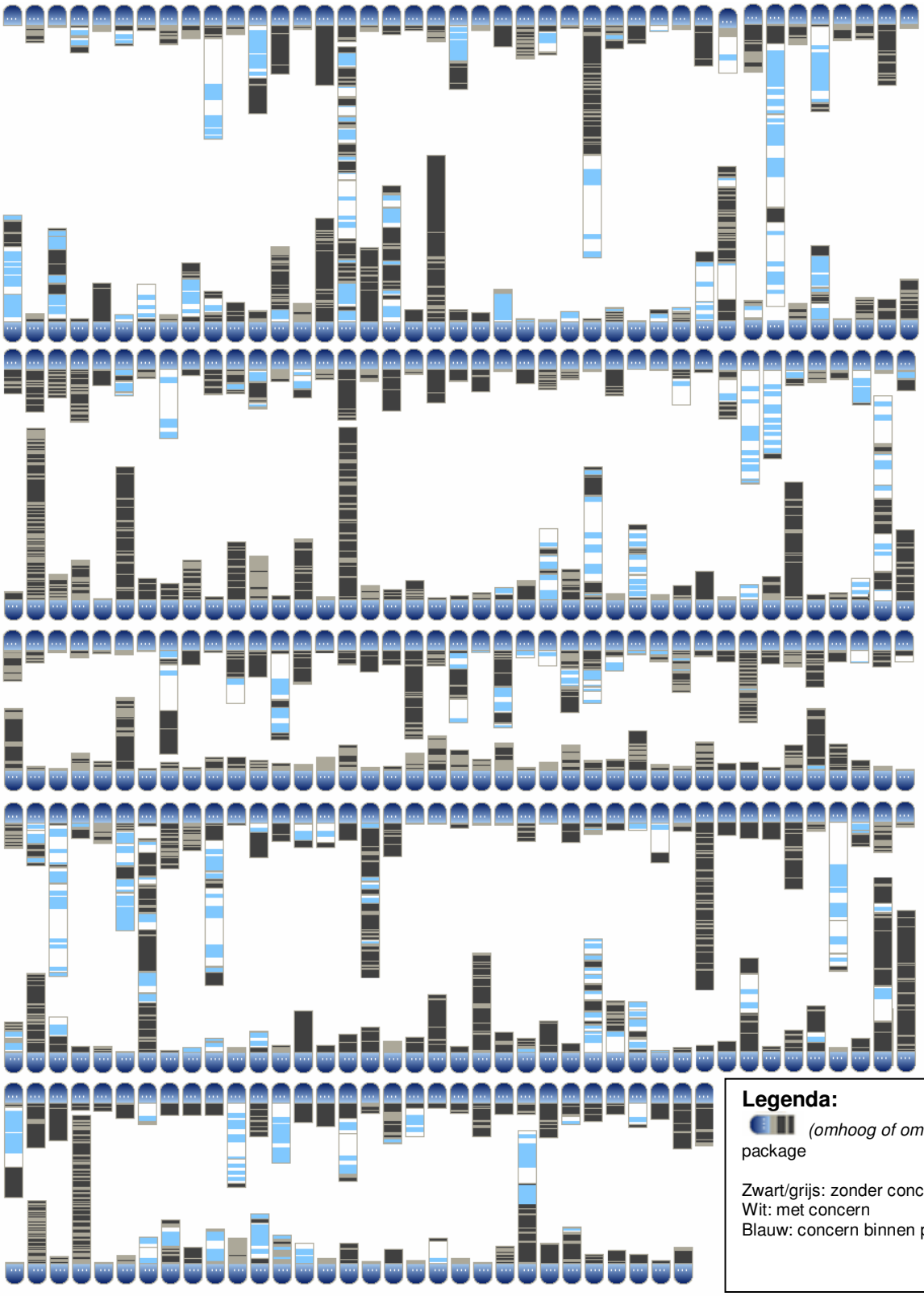
Onderstaande figuur toont het *concurrency concern model* voor Azureus. In de figuur zijn zowel de concerns met betrekking tot het introduceren van concurrency als het beschermen van de *shared state* opgenomen.



Opmerkingen:

1. In de figuur ontbreekt het gebruik van *synchronized*.
2. In de figuur is het voor het *sort* CB (*consistent behavior*) ingevuld bij ieder concern, het daadwerkelijke *sort* zal echter onderzocht moeten worden.

Bijlage G: Lock scattering in de onderzochte applicatie



Legenda:

(omhoog of omlaag): package

Zwart/grijs: zonder concern
Wit: met concern
Blauw: concern binnen package