

Katholieke
Universiteit
Leuven

Faculteiten Wetenschappen
en Toegepaste Wetenschappen



BEVEILIGING VAN MEDISCHE INFORMATIESYSTEMEN

Tim DENOULET
Mathieu DESMET

Eindverhandeling aangeboden tot het
behalen van de graad van licentiaat
in de informatica

2003–2004

Promotor : Prof. Dr. ir. F. PIESENS

Naam en voornaam studenten: Denoulet Tim en Desmet Mathieu

Titel:

Beveiliging van Medische Informatiesystemen

Engelse vertaling:

Securing Medical Information Systems

ACM Classificatie: J.3

AMS Classificatie: 68N19

Korte inhoud:

De informatisering zet zich steeds sterker door, ook in de medische wereld. Aangezien medische informatie een zeer confidencieel karakter heeft, is het van groot belang om te waken over de veiligheid van medische informatiesystemen.

Na een studie van de wetgeving en vereisten omtrent de verwerking van medische gegevens, zien we dat we met deze complexiteit kunnen omgaan door de beveiliging als een afgescheiden belang te zien. De techniek die we gebruiken om tot het scheiden van belangen te komen, is *aspectgeoriënteerd programmeren*. Voor de implementatie zelf gebruiken we het raamwerk *JAC (Java Aspect Components)*. Mede door het bekijken van twee bestaande beleidstalen komen we tot een eigen taal. Het algemeen model van onze beveiliging bestaat uit twee aspecten: één voor de toegangscontrole en één voor de authenticatie. Het aspect dat de toegangscontrole regelt, maakt gebruik van het Uitgebreid View-Connectormodel. Dit model verbindt de beleidstaal met een concrete applicatie. De beveiliging is in de eerste plaats ontworpen specifiek voor medische applicaties, maar zou ook moeten bruikbaar zijn in andere toepassingen zoals die voor bankinstellingen.

We geven ook een aanzet tot enkele uitbreidingen van de ontworpen beveiliging en tonen hoe we ze in een gedistribueerde omgeving kunnen inzetten. Tenslotte formuleren we enkele voor- en nadelen van het beveiligen via aspectgeoriënteerd programmeren en van JAC in het bijzonder.

*Eindverhandeling aangeboden tot het behalen
van de graad van licentiaat in de informatica*

Promotor: Prof. Dr. ir. F. Piessens Departement Computerwetenschappen

Assessoren: Prof. Dr. ir. M. Bruynooghe
Dr. B. De Win

Begeleider: ir. T. Verhanneman

Inhoudsopgave

Inhoudsopgave	1
Lijst van figuren	4
Inleiding	5
1 Beveiliging van medische gegevens	7
1.1 Klinische informatiesystemen	8
1.1.1 Vereisten	8
1.1.2 Computer- en Communicatiebeveiliging	10
1.2 Wettelijk kader	10
1.2.1 Belgische wetgeving inzake verwerking van medische gegevens	10
1.2.2 Interpretatie van de wet	12
1.2.3 Omzetten van de wetgeving in de praktijk	13
1.3 Toegangscontrole	15
1.3.1 Beveiligingsbeleid	15
1.3.2 Aanpasbare toegangscontrole	16
1.4 Uitdaging	17
2 Op AOP gebaseerde softwarebeveiliging	18
2.1 Oorzaken hoge complexiteit softwarebeveiliging	18
2.1.1 Alomtegenwoordigheid van beveiliging	18
2.1.2 Onverwachte gevaren en veranderingen	20
2.2 Scheiden van belangen in software	20
2.2.1 Perspectieven van scheiden van belangen	20
2.2.2 Geavanceerde scheiding van belangen	21
2.3 Aspectgeoriënteerd programmeren	21
2.3.1 Nood aan een hoger abstractieniveau	22
2.3.2 Aspecten bovenop objecten	22
2.3.3 Weaving	23
2.4 Beveiliging met AOP	23
2.4.1 Uittekenen krijtlijnen	23
2.4.2 Beveiliging als afgescheiden belang	24
2.4.3 Voordelen	25
2.5 Conclusie	26

3	Beleidsstalen	27
3.1	Declaratieve beleidstaal	27
3.1.1	Beleidsstaal voor flexibele toegangscontrole	28
3.1.2	Integratie declaratieve beleidstaal	29
3.2	Bestaande beleidsstalen	31
3.2.1	Ponder	31
3.2.2	SPL	34
3.2.3	Conclusie	37
3.3	Binding via het Uitgebreid View-connectormodel	38
3.3.1	Het View-connectormodel	38
3.3.2	Uitgebreid View-connectormodel	39
3.3.3	Regels volgens Uitgebreid View-connectormodel	40
3.4	Besluit	44
4	Java Aspect Components	45
4.1	Aspect componenten	45
4.1.1	Programmeren	46
4.1.2	Configureren	47
4.2	Dynamische wrappers	47
4.2.1	Programmeren	48
4.2.2	Samenstellen	49
4.3	Architectuur	51
4.3.1	Werking	51
4.3.2	Performantie	53
4.4	Conclusie	53
5	Algemeen ontwerp beveiliging	54
5.1	Kandidaataspecten	54
5.1.1	Authenticatie	55
5.1.2	Sessie	55
5.1.3	Toegangscontrole	56
5.1.4	De interactie tussen kandidaataspecten	56
5.2	Communicatie tussen aspecten	56
5.2.1	Mogelijke ontwerpen	57
5.2.2	Conclusie	62
5.3	Design	64
5.3.1	Applicatiemodel	64
5.3.2	Beveiligingsmodel	64
5.4	Besluit	65
6	Access-control aspect	66
6.1	Algemeen ontwerp	66
6.1.1	Client	67
6.1.2	Server	69
6.2	Nieuwe gegevensstructuren	70
6.2.1	MetaObject	70
6.2.2	MetaSubject	70

6.2.3	MetaPolicyRule	71
6.3	Beslissingsmechanisme voor toegangscontrole in JAC	71
6.3.1	Implementatie clientzijde	71
6.3.2	Implementatie serverzijde	72
6.4	Voorbeeld beleid voor toegangscontrole	74
6.4.1	Opstellen van beleid	74
6.4.2	Binden met applicatie	76
6.5	Conclusie	76
7	Verfijningen en bevindingen	77
7.1	Verfijningen toegangscontrole	77
7.1.1	Beslissingsmechanisme kan efficiënter	77
7.1.2	Statisch informatie ophalen	78
7.1.3	Uitschakelen beveiliging tijdens beveiliging	78
7.1.4	Controleren van inconsistentie in regels en connectoren	79
7.1.5	Functionaliteiten beleidstaal uitbreiden	79
7.2	Bevindingen	80
7.2.1	Beveiliging via AOP	80
7.2.2	AOP via JAC	81
7.3	Distributie	83
7.3.1	JAC en distributie	83
7.3.2	Gedistribueerde applicatie	84
7.3.3	Schaalbaarheid van onze oplossing	84
7.4	Conclusie	86
	Besluit	87
A	Binding applicatie	89
A.1	Domeinen	89
A.2	Interfaces	90
A.2.1	Access-interfaces	90
A.2.2	Subject-interface	90
A.3	View-connectoren	91
A.3.1	Object	91
A.3.2	Subject	92
B	Het programma	93
B.1	Installatie	93
B.2	Werking	93
	Bibliografie	96

Lijst van figuren

1.1	Overzicht toegangscontrole	15
3.1	Ponder Object Meta Model	31
3.2	Syntax autorisatieregel	32
3.3	Voorbeeld autorisatieregel	32
3.4	Voorbeeld verplichting	33
3.5	Voorbeeld basisconstraint	34
3.6	Voorbeeld rol	34
3.7	Een eenvoudige rol	36
3.8	Toepassing van rol	37
3.9	View-connectormodel	40
4.1	Voorbeeld aspect component	46
4.2	Voorbeeld wrapper	48
4.3	Onderscheppen van een methode	49
4.4	Wrappers samenstellen	50
4.5	De JAC architectuur	52
5.1	JAC container	55
5.2	Samengesteld aspect	58
5.3	Hoofd- en deelaspect	59
5.4	Uitbreiding interaction	60
5.5	Wrappen rond aspect	61
5.6	Attribuut aan thread toevoegen	62
5.7	Model van de applicatie	64
5.8	Model van de beveiliging	65
6.1	Model van beslissingsmechanisme	67
6.2	Overzicht client-kant	72
6.3	Overzicht server-kant	73
7.1	Gedistribueerd model	85

Inleiding

Net zoals in de meeste sectoren sijpelt ook in de medische wereld de informatisering door. Het tijdperk van de fichebakken met de patiëntengegevens loopt op zijn eind. In de plaats komen grote netwerken waar veel gegevens beschikbaar worden gesteld voor veel gebruikers.

In het Verenigd Koninkrijk heeft het opzetten van een nationaal netwerk van patiëntengegevens, namelijk het *National Health Security Network (NHS)*, veel bezorgdheid opgewekt bij artsen en ander medisch personeel. Ze vrezen voor de vertrouwelijkheid van de patiëntengegevens, gezien die nu op veel plaatsen toegankelijk zijn. We citeren in dezelfde context een artikel uit De Standaard van 1 april 2004 [1, 2]: *“Die gebrekkige beveiliging is verontrustend, want steeds meer ziekenhuizen zetten de dossiers met medische gegevens van hun patiënten op een intern computernetwerk. En die medische dossiers bevatten hoe langer hoe meer gedetailleerde informatie over de medische geschiedenis van de patiënt, informatie die door verzekeraars en werkgevers gemakkelijk misbruikt kan worden.”* De aanleiding van deze artikels was een jongere die zonder probleem enkele draadloze netwerken van ziekenhuizen kon binnendringen en zo patiëntengegevens kon opvragen.

Het beveiligen van medische gegevens is een complex gegeven. Ten eerste is er tegenwoordig een strenge wetgeving die zware beperkingen oplegt omtrent de toegankelijkheid tot klinische gegevens. Bovendien heeft een medische instelling een ingewikkeld personeelsbeleid. Men heeft specialisten die een bepaalde zelfstandigheid bezitten, verplegers en hoofdverplegers, ander medisch personeel (zoals kinesisten) en het administratief personeel. Tenslotte bestaat een ziekenhuis vaak uit afzonderlijke departementen met hun eigen intern systeem die communiceren via een al dan niet intern netwerk. Dit is nu zeker het geval met de huidige golf van fusies tussen ziekenhuizen.

Het is duidelijk dat bijvoorbeeld beveiliging via het besturingssysteem of een beveiliging die is ingebouwd in een gegevensbank niet toereikend zijn voor de toegangscontrole van dit complex probleem. Daarom is het aangewezen om in dit geval een softwarebeveiliging te gebruiken bovenop de besturingssystemen of databanken. Deze softwarebeveiliging moet in de eerste plaats een uitgebreide toegangscontrole aanbieden maar moet ook overweg kunnen met de veelheid van verschillende systemen. Met andere woorden een en dezelfde beveiliging moet kunnen gebonden worden aan verschillende applicaties (onderliggende systemen).

Het doel van deze thesis is een dergelijke softwarebeveiliging te ontwerpen met de recente technologie van aspectgeoriënteerd programmeren. We willen ook de beveiliging die we voor deze toepassing ontwerpen, kunnen gebruiken in andere sectoren, bijvoorbeeld in een bankinstelling.

We beginnen in hoofdstuk 1 met een *studie van medische gegevens* en de vereisten waaraan deze moeten voldoen. In de medische wereld wordt beveiliging namelijk steeds belangrijker en kritischer ten gevolge van de opmars van de informatisering. In hoofdstuk 2 bestuderen we hoe we een beveiliging kunnen realiseren die zo onafhankelijk mogelijk is van de toepassing zelf. Hiervoor bekijken we het concept van *aspectgeoriënteerd programmeren*. Voor de beveiliging van gegevens zijn uiteraard regels nodig, die worden gebundeld in een beleid. Een beleid wordt uitgedrukt in een specifieke *beleidstaal*. We bekijken in hoofdstuk 3 wat de eigenschappen van zo'n taal zijn en komen tot een model met een taal die aan onze eisen voldoet.

Om uiteindelijk tot een implementatie van een beveiliging te komen zullen we gebruik maken van een recent ontwikkeld raamwerk, met name *Java Aspect Components*, waarmee dynamisch aan aspectgeoriënteerd programmeren kan worden gedaan. We bestuderen het raamwerk in hoofdstuk 4. Alvorens we overgaan tot de eigenlijke implementatie, maken we eerst een *ontwerp van de beveiliging* die we wensen. Hiervoor moeten we onder andere nagaan hoe verschillende aspecten kunnen interageren met elkaar. In hoofdstuk 5 werken we dit ontwerp uit. Het belangrijkste onderdeel van onze beveiliging is de *toegangscontrole*. Omdat die redelijk omvangrijk is, hebben we daar een volledig hoofdstuk aan gewijd, meerbepaald hoofdstuk 6. In hoofdstuk 7 sluiten we af met enkele *bevindingen* die we in de loop van deze thesis ervaren hebben.

Hoofdstuk 1

Beveiliging van medische gegevens

Een kwart eeuw geleden werd alle medische informatie bijgehouden op papier. Iedere arts had dan een fichebak met steekkaarten per patiënt. Wanneer een andere arts gegevens over een patiënt wou opvragen, vroeg hij gewoon inzage in de fiche. Langzamerhand sijpelt de informatisering ook door in de medische wereld, waardoor er steeds minder op papier wordt bijgehouden. In eerste instantie ontstaan er kleinere lokale systemen (alleenstaande pc's) die geen extra gevaren teweeg brengen: men kan de informatie enkel verkrijgen via de arts, inbraak of diefstal (net zoals tevoren). Volgens Anderson [3] zijn er twee mogelijke implementaties van elektronische klinische records:

1. Een afspiegeling van het bestaande *fichebakstelsel*, namelijk iedere zorgverstreker (*definitie in 1.1*) houdt op zijn eigen pc een record bij. Uitwisselen van data gebeurt via samenvattingen.
2. Iedere patiënt heeft maar één record met alle klinische informatie, dat wordt aangemaakt bij geboorte en vernietigd na autopsie.

Tegenwoordig is er een tendens die gaat van de eerste naar de tweede implementatie omdat netwerken hun intrede doen in medische systemen. De patiëntengegevens worden nu gecentraliseerd¹ of gedistribueerd² bijgehouden. Daarbij komt ook nog dat veel ziekenhuizen fusioneren waardoor veel personen toegang krijgen tot veel data. Het spreekt voor zich dat er van die situatie veel misbruik kan worden gemaakt.

In dit hoofdstuk bespreken we eerst de vereisten waaraan medische systemen moeten voldoen, evenals mogelijke bedreigingen en gevaren die hiermee gepaard gaan (1.1). Daarna bekijken we de Belgische wetgeving inzake verwerking van medische gegevens en halen we er een aantal mogelijke principes uit die kunnen helpen bij het opstellen van een beleid (1.2). Tenslotte bespreken we een toegangscontrole samen met een beveiligingsbeleid dat nodig is voor medische gegevens (1.3).

¹Per patiënt is er één record met alle data.

²Per patiënt is er een hoofdrecord dat referenties bijhoudt naar locaties waar er informatie staat over die patiënt.

1.1 Klinische informatiesystemen

Voor we verdergaan zouden we graag enkele definities introduceren:

Persoonlijke medische gegevens: de informatie die handelt over de gezondheid, medische geschiedenis of medische behandelingen van een persoon. Die informatie wordt zo bijgehouden dat de persoon kan geïdentificeerd worden door een *zorgverstreker* die de behandeling niet uitvoert.

Zorgverstreker: (in het Engels *clinician*) een gediplomeerd persoon zoals een arts, verpleger, tandarts, fysiotherapeut of apotheker die toegang heeft tot records.

Patiënt: representatie van een individu. In de meeste gevallen slaat dit op de patiënt zelf, maar in sommige gevallen kan “patiënt” verwijzen naar diegene die verantwoordelijk is voor de opgenomen persoon. Zo worden bijvoorbeeld de ouders van een minderjarige patiënt bedoeld, of ook de verantwoordelijke van een mentaal mindervalide.

Systeem: het geheel van hardware, software en communicatie.

1.1.1 Vereisten

Er zijn verschillende vereisten waaraan moet voldaan worden in een medisch informatiesysteem. Zo is er de nood aan vertrouwelijkheid (1.1.1.1), evenals aan integriteit en beschikbaarheid (1.1.1.2) van de gegevens van een patiënt.

1.1.1.1 Vertrouwelijkheid

De *eed van Hippocrates* (ongeveer 400 v. Chr.) was een eerste stap van medische vertrouwelijkheid in de ethiek van het beroep van een arts. Omdat die eed al millennia meegaat heeft men een moderne uitvoering opgesteld, namelijk de *Good Medical Practice* [4]. Wanneer men deze kort samenvat, vindt men het volgende:

“Patiënten hebben het recht om te verwachten dat je geen enkele informatie doorgeeft die je leerde in het kader van professionele activiteiten.”

De autoriteiten en gezondheidszorg zijn het eens dat elektronische medische gegevens ten minste even goed beschermd moeten worden als die op papieren fiches. Het basisprincipe in de Europese Unie is dat de patiënt ervan op de hoogte moet gebracht worden wanneer zijn gegevens worden doorgegeven. Een aantal uitzonderingen zijn in de loop der tijden ontwikkeld, zowel op wettelijke basis als op grond van pragmatisme. Deze bevatten de bekendmaking van abortus, geboortes, sommige sterfgevallen, bepaalde ziektes, tegenreacties op geneesmiddelen, rijverbod en vrijgeven van gegevens aan een advocaat bij een dispuut.

Ten laatste moet een patiënt op de hoogte gebracht worden van het feit dat zijn record met gegevens in een computersysteem wordt bijgehouden. Soms kan het gebeuren dat een persoon (bijvoorbeeld een publiek figuur) expliciet vraagt dat zijn gegevens enkel op papier worden bijgehouden. Dit kan opgelost worden door gebruik te maken van pseudoniemen zodat de ware identiteit van de patiënt niet in het systeem voorkomt.

Later, in 1.2.1, gaan we dieper in op de wetgeving.

1.1.1.2 Bedreigingen en zwakheden

Naast vertrouwelijkheid moet er ook gelet worden op de *integriteit* en *beschikbaarheid* van medische gegevens. Wanneer de data onbetrouwbaar zijn, dan zijn ze waardeloos voor medische beslissingen. Er zijn veel discussies aan de gang of men een actie kan justificeren aan de hand van elektronische data en of men al dan niet een back-up op papier of in de vorm van microfiches moet bijhouden.

Daarbij komt ook dat wanneer die elektronische gegevens niet altijd beschikbaar zijn door een systeemfout of sabotage, de waarde van die gegevens vermindert en er voorzichtig mee moet omgesprongen worden.

De kans dat data op een ongeoorloofde manier gebruikt worden hangt van twee factoren af: ten eerste de waarde van de informatie zelf en ten tweede het aantal personen dat toegang heeft tot de data. Het is maar logisch dat hoe groter de groep personen is die toegang heeft tot de data, hoe hoger de kans is dat er iemand bij is die malafide bedoelingen heeft. De waarde van medische data blijkt uit de volgende misbruiken:

- Medische gegevens die geraadpleegd worden bij aanwervingen of andere personeelsbeslissingen.
- Farmaceutische bedrijven die personen met een ziekte rechtstreeks aanschrijven.
- Banken die medische gegevens opvragen bij toewijzen van leningen.

Uit studies in de Verenigde Staten [5] blijkt dat de meeste bedreigingen van binnenuit komen. Dit wordt in de hand gewerkt door het gebruik van netwerkgebaseerde systemen. Het mag bijvoorbeeld niet dat administratief personeel in een ziekenhuis alle gegevens van een patiënt kan inkijken. Anderzijds mag een dokter enkel gegevens doorgeven aan een collega als die relevant zijn voor de behandeling door die tweede dokter. Vaak claimen verzekeraars, sociaal assistenten, mensen van de politiediensten en administratief personeel de *need-to-know* van persoonlijke gegevens. Een gerechtelijke uitspraak in het Verenigd Koninkrijk vond zelfs dat de HIV-status van een dokter niet mag vrijgegeven worden [6].

Er bestaan nog allerhande andere bedreigingen voor klinische informatie. Hierna volgen er enkele.

- Softwarebugs en fouten in hardware kunnen boodschappen corrupt maken.
- Een hoog aantal fouten kan voorkomen bij het automatisch verwerken van data.
- Virussen kunnen data vernietigen.
- Aanvallers manipuleren boodschappen.
- Interne aanvallen, zoals het vernietigen en toevoegen van informatie, diefstal en fraude, kunnen voorkomen. Dit gebeurde vroeger eveneens bij papieren fiches.
- Wanneer het publieke vertrouwen geschonden is, verzwijgen patiënten bepaalde gevoelige zaken.

Een mogelijke opsplitsing op hoog niveau van beveiliging zou kunnen zijn dat men de communicatiebeveiliging splitst van de beveiliging van computers zelf.

1.1.2 Computer- en Communicatiebeveiliging

Bij *computerbeveiliging* (*Compusec*) gaat het erom te beveiligen wat er op een computer staat. Die beveiliging kan ingebouwd zijn in het besturingssysteem of andere software.

In een kleiner systeem kan men bijvoorbeeld paswoord-authenticatie als enige techniek gebruiken. Wanneer men via een netwerk toegang kan krijgen moet men andere technieken introduceren. Toegangscontrolelijsten (in het Engels *access control lists*) zijn lijsten die de personen bevatten die toegang mogen krijgen tot een object en worden door veel besturingssystemen ondersteund (bv. door UNIX). Wanneer men een systeem voor databankbeheer gebruikt dan moet de toegangscontrole met de granulariteit van een record in beheersysteem van de gegevensbank zelf geïmplementeerd worden. Bij gedistribueerde systemen kan men encryptie gebruiken en de toegangscontrole ligt dan vooral in het mechanisme dat de sleutels beheert.

De beveiliging die besturingssystemen voorzien is vaak te beperkt wanneer men ingewikkelde toegangscontrole wil voorzien. Een belangrijke oorzaak hiervan is dat een besturingssysteem geen informatie heeft over de interne toestand van de toepassing, terwijl die wel nodig is voor de beveiliging ervan. Complexere beveiliging moet voorzien worden, op applicatieniveau, die op zijn beurt wel gebruik kan maken van de beveiliging van onderliggende lagen (besturingssysteem, gegevensbank,...).

Communicatiebeveiliging (*Comsec*) heeft als hoofddoel dat de toegangscontrole niet wordt omzeild bij het overbrengen van gegevens van de ene computer naar de andere. Verder moet de integriteit van data die migreren, gewaarborgd worden. Daarbij komt dat in de geneeskunde de autoriteit niet echt hiërarchisch is: ze is eerder lokaal en collegiaal dan gecentraliseerd en bureaucratisch, wat de toegangscontrole complex maakt.

In dit werk concentreren we ons vooral op het controleren van de toegang tot gegevens. We zullen ons niet bezig houden met encryptie voor transport of bijhouden van gegevens. In een realistische omgeving zal dit natuurlijk wel nodig zijn.

1.2 Wettelijk kader

In deze sectie geven we een extract uit de Belgische wetgeving met betrekking tot het verwerken van medische gegevens (1.2.1). Daarna, in 1.2.2, bespreken we dat extract en tenslotte geven we een voorbeeld van hoe de wetgeving kan omgezet worden in een concreet beleid (1.2.3).

1.2.1 Belgische wetgeving inzake verwerking van medische gegevens

In deze sectie beschouwen we de voor ons relevante stukken van de Wet van 8 december 1992 tot bescherming van de persoonlijke levenssfeer ten opzichte van de verwerking van persoonsgegevens (gecoördineerde versie 2003), **Hoofdstuk II, artikel 7** [7].

§ 1. De verwerking van persoonsgegevens die de gezondheid betreffen, is verboden.

§ 2. Het verbod om de in §1 bedoelde persoonsgegevens te verwerken, is niet van toepassing in de volgende gevallen:

a) wanneer de betrokkene schriftelijk heeft toegestemd in een dergelijke verwerking met dien verstande dat deze toestemming te allen tijde door de betrokkene kan worden ingetrokken; de Koning kan, bij een in Ministerraad overlegd besluit na advies van de Commissie voor de bescherming van de persoonlijke levenssfeer, bepalen in welke gevallen het verbod om gegevens betreffende de gezondheid te verwerken niet door de schriftelijke toestemming van de betrokkene ongedaan kan worden gemaakt;

...

f) wanneer de verwerking noodzakelijk is ter verdediging van vitale belangen van de betrokkene of van een andere persoon indien de betrokkene fysiek of juridisch niet in staat is om zijn toestemming te geven;

...

j) wanneer de verwerking noodzakelijk is voor doeleinden van preventieve geneeskunde of medische diagnose, het verstrekken van zorg of behandelingen aan de betrokkene of een verwant, of het beheer van de gezondheidsdiensten handelend in het belang van de betrokkene en de gegevens worden verwerkt onder het toezicht van een beroepsbeoefenaar in de gezondheidszorg;

§ 3. De Koning legt, bij een in Ministerraad overlegd besluit en na advies van de Commissie voor de bescherming van de persoonlijke levenssfeer, bijzondere voorwaarden vast waaraan de verwerking van de in dit artikel bedoelde persoonsgegevens moeten voldoen.

§ 4. Persoonsgegevens betreffende de gezondheid mogen, behoudens schriftelijke toestemming van de betrokkene of wanneer de verwerking noodzakelijk is voor het voorkomen van een dringend gevaar of voor de beteugeling van een bepaalde strafrechtelijke inbreuk, enkel worden verwerkt onder de verantwoordelijkheid van een beroepsbeoefenaar in de gezondheidszorg.

De Koning kan, na advies van de Commissie voor de bescherming van de persoonlijke levenssfeer en bij een in Ministerraad overlegd besluit, bepalen welke categorieën van personen als beroepsbeoefenaars in de gezondheidszorg in de zin van deze wet worden beschouwd.

Bij de verwerking van de in dit artikel bedoelde persoonsgegevens zijn de beroepsbeoefenaar in de gezondheidszorg en zijn aangestelden of gemachtigden, tot geheimhouding verplicht.

§ 5. Persoonsgegevens betreffende de gezondheid moeten worden ingezameld bij de betrokkene.

Zij kunnen slechts via andere bronnen worden ingezameld op voorwaarde dat dit in overeenstemming is met de paragrafen 3 en 4 van dit artikel en dat dit noodzakelijk is voor de doeleinden van de verwerking of de betrokkene niet in staat is om de gegevens te bezorgen.

In dezelfde wet staat in **Hoofdstuk IV, Artikel 16, § 4** het volgende over beveiliging van persoonsgegevens in het algemeen:

Om de veiligheid van de persoonsgegevens te waarborgen, moeten de verantwoordelijke van de verwerking, en in voorkomend geval zijn vertegenwoordiger in België, alsmede de verwerker, de gepaste technische en organisatorische maatregelen treffen die nodig zijn voor de bescherming van de persoonsgegevens tegen toevallige of ongeoorloofde vernietiging, tegen toevallig verlies, evenals tegen de wijziging van of de toegang tot, en iedere andere niet toegelaten verwerking van persoonsgegevens.

Deze maatregelen moeten een passend beveiligingsniveau verzekeren, rekening houdend, enerzijds, met de stand van de techniek terzake en de kosten voor het toepassen van de maatregelen en, anderzijds, met de aard van de te beveiligen gegevens en de potentiële risico's. Op advies van de Commissie voor de bescherming van de persoonlijke levenssfeer kan de Koning voor alle of voor bepaalde categorieën van verwerkingen aangepaste normen inzake informaticaveiligheid uitvaardigen.

1.2.2 Interpretatie van de wet

Het is duidelijk dat de wet streng is op vlak van de verwerking van medische gegevens. Het is bijgevolg ook logisch dat medische instellingen zo weinig mogelijk risico's willen lopen op rechtszaken wanneer bijvoorbeeld aan het licht komt dat een interne of externe persoon (al dan niet met kwaad opzet) ongeautoriseerde toegang heeft verkregen tot gevoelige gegevens. Daarom zijn medische instellingen bereid te investeren in de beveiliging van medische data.

Wanneer we nu artikel 7 van 1.2.1 beschouwen, kunnen we de volgende zaken eruit distilleren. In § 2 worden de gevallen beschreven wanneer het toegelaten is persoonlijke data betreffende de gezondheid te verwerken. In punt a) wordt duidelijk gesteld dat de persoon op wie de gegevens betrekking hebben zijn schriftelijke toestemming moet geven. Punt f) stelt een uitzondering op a) waarbij de persoon niet in staat is zijn toestemming te geven, denk maar aan de situatie waarbij een persoon bewusteloos de spoedgevallendienst wordt binnen gebracht. In punt j) wordt beschreven dat enkel als de persoonlijke gegevens onder het toezicht van een beroepsbeoefenaar uit de gezondheidszorg staan, deze mogen verwerkt worden om diagnoses te stellen of andere medische handelingen uit te voeren.

§ 4 vat eerst § 2 kort samen, maar gaat dan verder in op de term van *beroepsbeoefenaars in de gezondheidszorg*. Zowel de beroepsbeoefenaar als de personen die hij aanstelde, moeten de informatie geheim houden.

In § 5 wordt gezegd dat de medische gegevens enkel via de persoon zelf mogen ingezameld worden behalve wanneer paragrafen 3 en 4 voldaan zijn. Daarbij komt dat die uitzonderingen toegelaten zijn ofwel als ze noodzakelijk zijn voor de verwerking ofwel als de persoon zelf niet in staat is de gegevens te geven. Wanneer de persoon bijvoorbeeld een mentaal mindervalide is, mag de nodige informatie aan de voogd worden gevraagd.

In Hoofdstuk IV wordt er gesteld dat men naast organisatorische ook technische maatregelen moet treffen om de gegevens te beveiligen. Een organisatorische maatregel is bijvoorbeeld het grondig *screenen* van personeel dat toegang heeft tot de data terwijl technische voorzorgen eerder misbruik van die toegang moeten tegengaan. Hoofdstuk IV is van toepassing voor het beveiligen van persoonlijk data, het is evident dat medische gegevens ook onder deze noemer vallen.

In de volgende sectie geven we een mogelijke beveiligingspolitiek die rekening houdt met de facetten van de wetgeving die hier werden besproken.

1.2.3 Omzetten van de wetgeving in de praktijk

Zoals we eerder hebben gezien in 1.1.1.1 is het de bedoeling om digitale medische data minstens even goed te beveiligen als gegevens op papier. De wetgeving legt dit sinds 1992 ook op (zie 1.2.1 en 1.2.2). De volgende principes, die door Anderson beschreven worden in [3, p. 11–22] kunnen een mogelijkheid bieden om dit te realiseren.

We veronderstellen dat een record uit meerdere delen bestaat: een algemeen record, een record voor psychologische begeleiding, een record voor hartkwalen, ... Anderson heeft zich voor deze principes gebaseerd op een systeem dat gebruik maakt van *toegangscontrolelijsten* (*access control lists*).

Principe 1: Ieder record moet een lijst hebben van personen of groepen die de data mogen lezen en wijzigen. Het systeem geeft enkel toegang aan de personen op die lijst.

Toegangs informatie kan opgeslagen worden per subject of per object. Met *object* bedoelen we de entiteit die moet beveiligd worden, en met *subject* degene die probeert toegang te verkrijgen tot zo'n beveiligde entiteit. In het laatste geval noemt men het meestal *toegangscontrolelijsten*. zo'n lijst bevat alle personen die toegang kunnen krijgen tot het object in kwestie.

Principe 2: Een zorgverstrekker mag een record openen met zichzelf en de patiënt op de toegangscontrolelijst. Wanneer de patiënt is doorverwezen naar de zorgverstrekker in kwestie, mag deze laatste een record openen met zichzelf, de patiënt en de zorgverstrekker door wie de patiënt is doorverwezen op de toegangscontrolelijst.

Bij de aanmaak van een record mag men enkel toegangsrechten verlenen aan de maker van het record, de patiënt zelf en eventueel een andere zorgverstrekker die nauw betrokken is.

Principe 3: Eén van de zorgverstrekkers op de toegangscontrolelijst moet aangeduid worden als verantwoordelijke die het alleenrecht heeft om de lijst te wijzigen.

Het is belangrijk dat het beheer van een toegangscontrolelijst toegewezen wordt aan een persoon die de verantwoordelijkheid ervoor draagt. Hij alleen mag personen toevoegen of verwijderen.

Principe 4: De verantwoordelijke zorgverstrekker moet de patiënt melden welke namen op de toegangscontrolelijst staan bij het aanmaken van een record, bij toevoegingen en wanneer de verantwoordelijkheid is doorgegeven. De patiënt moet altijd zijn toestemming geven, behalve in noodgevallen en bij wettelijke vrijstelling.

De beheerder van de toegangscontrolelijst moet de patiënt melden wanneer de lijst wordt gewijzigd. Er zijn uitzonderingen waarbij men zonder de toestemming informatie mag uitwisselen (in noodgevallen), maar dan moet er een notificatie gestuurd worden naar de patiënt.

Principe 5: Niemand mag klinische informatie verwijderen voor een vastgelegde periode is verstreken.

Ook persistentie is belangrijk, maar enkele opmerkingen omtrent dit principe zijn mogelijk.

- Na verloop van de vereiste periode mag men het record vernietigen, maar er wordt niets verplicht.
- Het zesde principe van de Data Protection Act [8] zegt dat men persoonlijke informatie niet langer dan nodig mag houden. Er moet een mechanisme uitgedacht worden voor het vernietigen van het record.
- Een patiënt kan de toestemming doorgeven aan bijvoorbeeld een zorgverstrekker.
- Bij tijdelijke kopieën moet de vastgestelde periode kleiner zijn.

Principe 6: Iedere toegang tot een record moet in het record vermeld worden met de naam van het subject (degene die toegang heeft verkregen), datum en tijdstip. Hetzelfde moet gebeuren met het verwijderen van gegevens uit een record.

Dit is het principe van het loggen van alle operaties, uitgevoerd op een record, zodat men achteraf kan controleren wie er wat allemaal heeft uitgevoerd.

Principe 7: Informatie, afgeleid van een record A, mag aan een ander record B gehangen worden op voorwaarde dat de toegangscontrolelijst van B een deel is van die van A.

Hier stelt men een duidelijke regel bij het toevoegen van informatie van een record aan een ander. Het vermijdt dat men gegevens aan een record toevoegt wanneer er personen op de toegangscontrolelijst van het record staan maar niet op die van de toegevoegde gegevens.

Principe 8: Er moeten maatregelen genomen worden om het verzamelen van medische informatie tegen te gaan. Concreet moeten patiënten op de hoogte gebracht worden wanneer iemand die op de toegangscontrolelijst wil staan al toegang heeft tot persoonlijke gegevens van een groot aantal mensen.

Dit principe houdt verband met aggregatiecontrole. Men kan data natuurlijk ook gebruiken voor onderzoek en boekhoudkundige zaken, maar dan moet die voldoende anoniem gemaakt worden. Dat is erg moeilijk, want wanneer bijvoorbeeld iemand zeer specifieke informatie opvraagt dan kan hij/zij één record als resultaat krijgen. Omdat veel informatie met elkaar is verbonden moet er een *statistische beveiliging* zijn om te vermijden dat men uit algemene informatie persoonlijke gegevens kan halen.

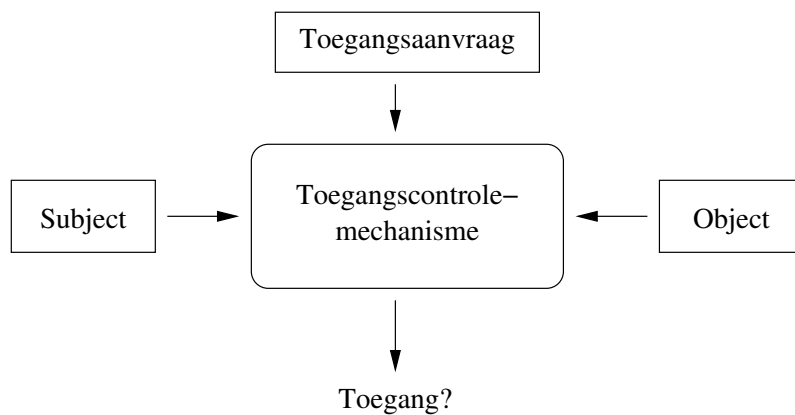
Principe 9: Computersystemen die persoonlijke medische data beheren, moeten een subsysteem hebben dat alle bovenstaande principes oplegt. De doeltreffendheid ervan moet geëvalueerd worden door onafhankelijke experts.

Dit laatste principe spreekt voor zich. Alle voorgaande principes moeten gerealiseerd worden en dit alles moet dan door objectieve specialisten ter zake geëvalueerd worden.

We gaan ons zeker niet vastpinnen op deze negen principes, ze zijn eerder een illustratie van hoe een concreet beleid er kan uitzien, bovendien vertrekken ze vanuit de techniek van *toegangscontrolelijsten*.

1.3 Toegangscontrole

Toegangscontrole laat ons toe te controleren of een subject (in veel gevallen een persoon) dat wil toegang krijgen tot een object (hier is dat een patiëntenrecord), effectief toegang heeft tot dat object. Dit wordt in figuur 1.1 schematisch voorgesteld.



Figuur 1.1: Overzicht toegangscontrole

Om te kunnen bepalen wie nu effectief toegang kan verkrijgen tot welke objecten, is het noodzakelijk om regels op te stellen. Deze regels worden gebundeld in een beveiligingsbeleid (1.3.1). We baseren ons hiervoor op wat de wetgeving hieromtrent oplegt (gezien in 1.2). Tenslotte zien we in 1.3.2 de nood aan een aanpasbare toegangscontrole.

1.3.1 Beveiligingsbeleid

Het eenvoudigste beleid bestaat erin iedereen toegang te verlenen tot alle objecten. Dit kan uiteraard niet als een echt beleid worden beschouwd. In de realiteit zullen er verschillende restricties zijn die moeten worden opgelegd aan de toegang tot gegevens.

Deze toegangscontrole kan op twee manieren verkregen worden. Enerzijds kan men stellen wat iemand mag doen, anderzijds kan men aangeven wat iemand niet mag doen. Deze twee soorten toegangsregels kunnen verwezenlijkt worden met respectievelijk positieve en negatieve regels. Een *positieve regel* geeft aan in welk geval het subject toegang krijgt tot het object. Als voorbeeld kan de volgende regel worden beschouwd: “*Wanneer arts A patiënt B behandelt, mag hij het record van B lezen*”.

Een *negatieve regel* daarentegen geeft aan in welk geval het subject geen toegang krijgt tot het object. Een voorbeeld hier kan zijn: “*Een psycholoog heeft geen toegang tot cardiovasculaire gegevens*”.

Verwerking van medische gegevens is in principe verboden, behalve in de uitzonderingsgevallen die in de wet worden geformuleerd (1.2.1). Er moet dus worden bepaald in welke gevallen iemand wel toegang krijgt tot de gevraagde informatie. In medische toepassingen geeft men eerder toegangsrechten dan dat men ze ontnemt omdat men ervan uitgaat dat een persoon geen toegang heeft. Het is daarom het meest aangewezen om positieve regels te gebruiken. Vandaar dat het beleid van onze medische toepassing uit positieve regels zal bestaan. In de beveiliging die we in de loop van deze thesis voorstellen gaan we geen gebruik maken van negatieve regels.

Laat ons eerst kijken waaraan een toegangscontrole zou moeten voldoen.

1.3.2 Aanpasbare toegangscontrole

Een algemeen beveiligingsbeleid kan op twee manieren opgelegd worden [9], ook de wetgeving gebruikt dezelfde indeling (zie 1.2)

1. via organisatorische maatregelen
2. m.b.v. technische maatregelen

Organisatorische maatregelen kunnen onder meer inhouden dat men het ziekenhuispersoneel grondig screent of het personeel attent maakt op mogelijke bedreigingen.

Een technische maatregel kan zijn: het opleggen van de toegangsregels via Informatie Technologie (IT).

1.3.2.1 Regels opleggen via IT

Vroeger was er een tendens om de verantwoordelijkheid te leggen bij de zorgverstrekker (*trust based model*). Door zowel de specialisatie van zorgverstrekkers (bv. orthopedist, kinesist en fysiotherapeut gebruiken vaak dezelfde informatie, terwijl er vroeger één zorgverstrekker was die deze functies allemaal zelf deed), als de groeiende complexiteit van verzorging zelf (bv. een psycholoog moet soms weten wat de recente medische achtergrond is van een patiënt), stijgt het aantal personen dat data van een patiënt raadpleegt.

Daarbij komt dat het gebruik van IT in ziekenhuizen stijgt en dat afdelingen vaak in aparte gebouwen zitten, telkens met een eigen administratie. Hierdoor worden gegevens niet meer centraal bijgehouden. Om die reden is er communicatie nodig, en die gebeurt onder andere via fysisch externe kanalen.

De regels die via IT worden opgelegd moeten dynamisch zijn om verschillende redenen:

1. Verandering in het algemene beveiligingsbeleid door wetgeving, andere interpretatie van de wetgeving, enzovoort moet kunnen ingebouwd worden.
2. Beveiliging is noodzakelijk dynamisch, het opleggen van toegangscontrole moet wettelijk regelmatig kunnen herbekeken worden.
3. IT beveiliging moet zich aanpassen wanneer er veranderingen optreden in het systeem.

Met een flexibele toegangscontrole kunnen we aan bovenstaande vereisten voldoen.

1.3.2.2 Ondersteuning voor flexibele toegangscontrole

Flexibele toegangscontrole kunnen we verkrijgen door gebruik te maken van een taal die de toegangsregels beschrijft. Bij een groot systeem kunnen we moeilijk voor ieder object een aparte regel maken. De taal waarin men de regels schrijft moet de objecten die dezelfde controle- of toegangsrechten hebben, kunnen groeperen.

Groeperen van objecten wordt meestal verkregen met behulp van *beleidsdomeinen* (*policy domains*). Een object kan tot een of meerdere domeinen behoren. Een krachtige taal voor het opstellen van de regels zou zowel verfijnde beschrijvingen als generaliserende definities moeten kunnen ondersteunen.

Daarnaast zou er ook een groepering mogelijk moeten zijn volgens *subjecten*. Het duidelijkste voorbeeld hierbij is de notie van rollen. Meer geavanceerd zou men dynamisch kunnen groeperen aan de hand van eigenschappen van een subject. Een subject mag maar een beperkte toegang hebben tot een object. Hier zullen we verder in dit werk uitgebreid op terug komen. In medische informatiesystemen lijkt het dat men de privileges in termen van *bekijken en wijzigen van data* kan uitdrukken. Wanneer de enige acties op een object expliciete lees- en schrijfoperaties zijn, zit de de complexiteit niet zozeer in de operaties die men uitvoert op records (objecten), maar eerder in het grote aantal en de verscheidenheid van objecten en subjecten (de uitvoerders van de operaties).

1.4 Uitdaging

Het beveiligen van medische gegevens is een complex probleem. Ten eerste zijn er veel gebruikers van de applicatie die vaak totaal verschillende belangen hebben, zoals dokters, verplegers, patiënten, ... Ten tweede is er veel informatie die moet beschermd worden tegen zowel externen als internen. Daarenboven is de context dynamisch: zowel de wetgeving als de systemen zelf zijn onderhevig aan veranderingen.

We hebben gemerkt dat de toegang tot medische gegevens bepaald wordt door een veelheid aan regels. Daarbij komt nog dat die regels kunnen veranderen (door wetswijziging, fusie, verandering van beleid, ...).

De uitdaging bestaat erin een beveiliging te ontwerpen die rekening houdt met dit alles. Het komt er voor ons dus op neer om een flexibele toegangscontrole te ontwikkelen.

Hoofdstuk 2

Op AOP gebaseerde softwarebeveiliging

Vroeger wou men al van bij de start een veilig systeem ontwikkelen, maar dat is utopisch voor (middel-)grote en complexe systemen. Pas tijdens de levensloop van een programma komen vele bedreigingen aan het licht, eventueel door fouten of onvolledigheden in de analyse van de bedreigingen of door veranderingen in de omgeving. Door het toepassen van het principe van het *scheiden van belangen* (*separation-of-concerns*) kan men de graad van beveiliging verhogen: men voert de beveiliging als het ware uit in een laag boven de toepassing zelf [10].

We focussen ons op de beveiligingselementen bij het ontwikkelen van software. Vaak wordt benadrukt dat beveiliging een eigenschap is van een systeem en dat het onmogelijk is om het beveiligingsaspect te scheiden van het functionele deel van het systeem. Hoewel beveiliging een alomtegenwoordige eigenschap is, moeten we toch proberen de beveiligingslogica af te schermen van de rest.

In sectie 2.1 gaan we op zoek naar de oorzaken van de problemen bij de beveiliging van software en in 2.2 gaan we dieper in op het scheiden van belangen. Daarna hebben we het in sectie 2.3 over aspectgeoriënteerd programmeren (AOP). Op het einde van dit hoofdstuk, in sectie 2.4, bespreken we tenslotte beveiliging met behulp van aspecten.

2.1 Oorzaken hoge complexiteit softwarebeveiliging

In deze sectie proberen we een ruwe analyse te maken van de problemen in softwarebeveiliging. We zullen later zien dat er geen *silver bullet* bestaat om deze problemen tegelijk op te lossen. Er zijn wel enkele technieken, zoals geavanceerde scheiding van belangen, die in hoge mate kunnen omgaan met deze complexiteit.

2.1.1 Alomtegenwoordigheid van beveiliging

Het probleem is dat men de beveiliging niet volledig kan loskoppelen. Omdat beveiliging zo ruim is (encryptie, toegangscontrole, invoercontrole, ...) kan men het niet als aparte module zien. Men kan ook niet een deel van de code beveiligen en voor de rest van de code er geen rekening mee houden.

Er is eveneens het probleem dat de stukken van de beveiliging doorheen heel de code verspreid zitten. Het is moeilijk (zomet onmogelijk) om de beveiliging te isoleren omdat een zwakte eender waar in het programma kan voorkomen.

Bijgevolg moet iedere programma-ontwikkelaar enige ervaring hebben met beveiliging, maar jammer genoeg is het onrealistisch te verwachten dat bij een groot project alle ontwikkelaars een aanvaardbare expertise hebben op vlak van beveiliging. We kunnen twee soorten van het alomtegenwoordig zijn van beveiliging onderscheiden: ten eerste is er het principe van veilig coderen (2.1.1.1), ten tweede is er de vaststelling dat beveiligingsgerelateerde belangen verstrengeld en verspreid zitten in de applicatiecode (2.1.1.2).

2.1.1.1 Veilige codering van alle onderdelen

De reflex om bij het implementeren te denken aan beveiliging is er vaak niet. Als men een deel van de applicatie onvoorzichtig ontwikkelt (op het eerste zicht niet gerelateerd met beveiliging), kunnen er al grote beveiligingsproblemen in het programma sluipen. Typerende voorbeelden hierbij zijn het niet controleren van invoer en bufferoverloop. Soms kunnen deze problemen verholpen worden door ondersteuning tijdens de uitvoering of door de compiler. Het doorschuiven van bepaalde verantwoordelijkheden naar een compiler, naar een runtime-omgeving of zelfs het opleggen van coderingsregels kunnen de beveiliging significant verbeteren maar dit zal nooit alle problemen van deze categorie oplossen.

Veilig coderen vereist dus een defensieve houding van een programmeur: naast nadenken over het realiseren van de opgelegde functionaliteit moet men er ook voor zorgen dat de code niet misbruikt kan worden om ongewilde resultaten te krijgen.

In deze thesis gaan we niet verder in op deze problematiek.

2.1.1.2 Beveiligingsgerelateerde belangen zijn verstrengeld en verspreid

Hier bekijken we de verspreiding in het programma van de beveiliging zelf. De logica van de beveiliging is als het ware een rode draad in de applicatie maar niet iets dat samenvalt met de structuur van het programma. Wanneer men een objectgerichte taal gebruikt, dan is de code die een toegangscontrolemodel implementeert vaak verspreid over vele klassen. Op deze manier is beveiliging een *verstrengeld en verspreid belang* (*crosscutting concern*) want ze snijdt doorheen vele klassen of functies in een programma.

Beveiliging is niet alleen verspreid door de diversiteit van de plaatsen waar het beveiligingsmechanisme wordt opgeroepen. Bepaalde beveiligingsmechanismen (zoals voor medische gegevens) hebben informatie nodig die niet in de applicatie zelf zit. Een geëncrypteerde communicatie in een applicatie heeft bijvoorbeeld verschillende gegevens nodig over het communicatiekanaal, de connecties en de host. In het domein van medische gegevens kan men bij de toegangscontrole naast informatie uit de applicatie (zoals gegevens over de sessie) ook informatie nodig hebben die niet (onmiddellijk) in de toepassing beschikbaar is, zoals tijdsafhankelijke informatie. Het is de nood aan applicatie-specifieke informatie die tot gevolg heeft dat toegangscontrole regels in de functionaliteit van de toepassing worden vervat.

2.1.2 Onverwachte gevaren en veranderingen

Het is naïef te denken dat men onmiddellijk een veilig programma kan ontwikkelen. Onverwachte gevaren steken de kop op tijdens de levensduur van een programma vaak omdat de *bedreigingsanalyse* (*threat analysis*, kijken wat de mogelijke gevaren zijn voor een applicatie) onvolledig was. Het is quasi onmogelijk om in de ontwikkelingsfase alle mogelijke gevaren te detecteren en er een sluitende oplossing voor te vinden. Ook de omgeving waarin het programma werkt, kan veranderen. Het kan bijvoorbeeld zijn dat een applicatie die oorspronkelijk draaide op een intern netwerk, nu moet communiceren via een extern en onbeveiligd kanaal.

Daarnaast is er altijd enige vorm van updating van een systeem nodig. Ook moet men rekening houden met veranderingen van de applicatie zelf of het uitbreiden van functionaliteiten. Daarbij komt nog dat als men van in het begin een erg veilig systeem wil, dat het initieel programma dan erg complex en dus duur wordt om te bouwen.

2.2 Scheiden van belangen in software

Scheiden van belangen kan men omschrijven als het opsplitsen van de probleemstelling in meerdere delen. Men lost ieder deel apart op zonder dat men veel rekening houdt met de andere. Tenslotte voegt men de verschillende delen samen tot een werkend geheel.

Over de jaren heen zijn er verschillende technieken ontstaan om scheiding van belangen te bekomen. Twee voorbeelden hiervan zijn functionele decompositie en objectoriëntatie. Deze technieken helpen vooral de functionaliteiten van programma's te scheiden, terwijl de niet-functionele vereisten (zoals beveiliging) zeer moeilijk te scheiden zijn in één aparte module van klassen of methoden.

We moeten dus op zoek gaan naar een meer geavanceerde techniek die ook tot op een bepaald niveau de niet-functionele belangen kan scheiden. Voor we een concrete oplossing bekijken gaan we eerst na vanuit welke perspectieven we de scheiding van belangen kunnen bekijken.

2.2.1 Perspectieven van scheiden van belangen

Om belangen te scheiden, zijn er verschillende opties. Deze mogelijkheden kunnen vanuit verschillende standpunten gekarakteriseerd worden (zie tabel 2.1). Wanneer we hier beveiliging als belang benaderen, bekijken we het zeker niet in de eerste plaats vanuit het prestatievermogen. In deze thesis is het namelijk niet de bedoeling om een snel werkende beveiliging te ontwikkelen maar voor een reëel systeem is dit zeker een issue. Met de rest houden we des te meer rekening.

Schaalbaarheid betekent dat onze beveiliging geen probleem mag hebben met het grote aantal gebruikers en records. De expressiviteit is ook erg belangrijk, hierbij denken we vooral aan de toegangscontrole die met complexe regels van het beveiligingsbeleid overweg moet kunnen. Samenstelbaarheid zou kunnen betekenen dat verschillende aspecten (bv. authenticatie en toegangscontrole) samengesteld worden om zo tot een volledige beveiliging te komen. Tenslotte moet de beveiliging aanpasbaar zijn, wat enerzijds betekent dat de basisapplicatie zou moeten kunnen werken zowel zonder de beveiliging als met een andere of herwerkte beveiliging en anderzijds dat de beveiliging zelf gewijzigd kan worden tijdens de levensloop van de toepassing.

Standpunt	Beschrijving
Prestatievermogen	Hoe eenvoudig kan het systeem geoptimaliseerd worden?
Schaalbaarheid	Is de oplossing gemakkelijk toepasbaar bij een groot aantal gebruikers en/of gegevens?
Expressiviteit	Laat de oplossing toe om de belangen expressief te specificeren?
Samenstelbaarheid	Kan de oplossing eenvoudig op hetzelfde moment meerdere belangen ondersteunen?
Aanpasbaarheid	Kan een belang eenvoudig worden verwijderd, toegevoegd of vervangen?

Tabel 2.1: Perspectieven van scheiding van belangen [11]

2.2.2 Geavanceerde scheiding van belangen

Men ging op zoek naar meer geavanceerde technieken om belangen te scheiden, waarvan men er heel wat onder de noemer van *Aspect Oriented Programming (AOP)* kan plaatsen [11].

De meeste toepassingen van beveiliging, zoals toegangscontrole, kan men herleiden tot een bepaald mechanisme op een aantal plaatsen in de toepassing. Via AOP zullen we dit beveiligingsaspect proberen te scheiden van de toepassing zelf. Met de state-of-the-art in AOP is dit gedeeltelijk mogelijk.

We zullen later zien dat men via AOP toegangscontrole kan scheiden van de applicatie zelf maar dat men het niet kan beschouwen als een aspect dat volledig onafhankelijk is van andere beveiligingsaspecten.

Volgens Filman en Friedman [12] bestaat een mogelijke classificatie van AOP uit het onderscheiden van *blackbox*- van *clearbox*-technieken. Een *blackbox*-techniek maakt enkel gebruik van de publieke interface van componenten. Met interface worden hier bijvoorbeeld publieke functies en objectmethodes bedoeld. *Clearbox* kijkt echter ook naar de interne structuur van een component, niet enkel naar de publieke interface.

2.3 Aspectgeoriënteerd programmeren

Volgens Pawlak [11] zijn er twee grote stromingen om aan scheiding van belangen te doen. Ze kunnen worden gezien als twee verschillende manieren om tot hetzelfde resultaat te komen:

top-bottom: gefundeerd op oplossingen die gebaseerd zijn op transformatie en verfijning van een hoog-niveau naar een concrete specificatie.

bottom-up: op niveau van de taal, gebruikt meer gevarieerde technieken in functie van de context en van de omgevings- en taalgebonden noden.

In dit kader wordt essentieel gekeken naar bottom-up oplossingen, gezien deze pragmatischer zijn en toelaten om het geheel van concrete problemen, die in reële toepassingen voorkomen, te omvatten.

2.3.1 Nood aan een hoger abstractieniveau

Het gebruik van nieuwe bronnen (een gevolg van de opkomst van het Internet en andere netwerken) is een van de grote uitdagingen van de moderne informatica. Door de natuurlijke verscheidenheid van gehanteerde informatie, de mogelijkheden van de verspreiding van informatie en van programma's, en door de heterogeniteit van bruikbare systemen of applicaties, wordt het programmeren van een toepassing die het geheel van bronnen maximaal uitbuit extreem complex. Om deze complexiteit onder controle te krijgen, verkrijgen informatici beetje bij beetje de middelen (formalismen, talen en protocollen) om functionaliteiten abstracter uit te drukken. Hoe hoger het abstractieniveau, hoe eenvoudiger, duidelijker, vlugger invoerbaar en beter onderhoudbaar de functionele expressie van de toepassing wordt. Maar om zo'n hoog niveau te bereiken, heeft men ook geavanceerde technieken nodig.

2.3.2 Aspecten bovenop objecten

De instantiatie van een abstracte architectuur voor het scheiden van belangen in een concreet programma leidt tot een specifiek type van programmeren dat *aspectgeoriënteerd* heet en dat als volgt kan gedefiniëerd worden [11]:

Aspectgeoriënteerd programmeren: in een concreet programma of systeem een verzameling constructies, eigenschappen of functionaliteiten voorop zetten en ze hergroeperen in modules of in gescheiden en coherente objecten zodat deze niet meer tot het initiële programma behoren. Het onderliggende resultaat van dit proces leidt tot een particuliere vorm van scheiding van belangen.

Er is een fundamenteel verschil tussen klassiek objectgericht en aspectgeoriënteerd programmeren. Het klassieke objectgericht programmeren (OO) is gebaseerd op een proces van factorisatie en abstractie van problematieken. Hierdoor laat OO niet toe om afhankelijkheden zodanig te elimineren, dat het lijkt "alsof ze niet meer tot het initiële programma behoren".

Er zijn een aantal problemen verbonden aan objectgericht programmeren:

- De nood aan herbruikbaarheid en onderhoudbaarheid leidt tot minimalisatie van afhankelijkheden die een verhoging van indirecties, en dus een merkbare daling van de performantie, met zich meebrengen.
- Wat het gekozen abstractieniveau ook is, er blijven altijd afhankelijkheden over.
- De overblijvende afhankelijkheden creëren een mix van belangen die niet wenselijk is binnen het kader van complexe en open ontwikkeling.
- Problemen met herbruikbaarheid en aanpasbaarheid komen voor, waardoor de wijziging van een van de belangen dikwijls leidt tot invalidatie van de abstractie in zijn geheel.

AOP introduceert een ander soort organisatie dan objectgericht programmeren. Deze organisatie bestaat erin de richting van de afhankelijkheid tussen het cliëntprogramma en de diensten, relatief aan de omgeving, om te keren. Deze inversie laat toe om de code volledig te scheiden van de afhankelijkheden met een gegeven belang (zoals bijvoorbeeld beveiliging).

Het mechanisme dat de afhankelijkheden tussen het programma en het aspect moet elimineren heeft niet voldoende aan een eenvoudige inversie van de afhankelijkheden. Het aspectprogramma wordt geen cliënt van het functionele programma, er wordt een nieuw soort afhankelijkheid ingevoerd: de *aspectafhankelijkheid*.

2.3.3 Weaving

AOP is vaak gebaseerd op de notie van weaving (letterlijk “weven”). Het is daarom aangegeven om een goede definitie ervoor te formuleren.

Weaving: het invoegen van supplementaire gegevens en/of verwerking in functionele componenten zonder dat die componenten die toevoegingen expliciteren in hun oorspronkelijke specificatie. De functionele componenten zijn zich bijgevolg niet bewust van de aanwezigheid van die bijkomende gegevens en/of verwerkingen.

Er moet worden gespecificeerd waar het weven moet uitgevoerd worden in de originele code. Het is precies de specificatie van deze locaties die aspectgeoriënteerd van objectgericht programmeren onderscheidt.

2.4 Beveiliging met AOP

In de vorige sectie hebben we kort uitgelegd waarvoor AOP staat. Hier gaan we nu na of AOP daadwerkelijk geschikt is voor beveiliging. We beginnen in 2.4.1 met het uittekenen van enkele krijtlijnen waaraan beveiliging zou moeten voldoen. Vervolgens gaan we in 2.4.2 na of AOP ervoor kan zorgen dat de doelen, zoals die in de krijtlijnen geformuleerd zijn, bereikt worden. Tenslotte, in 2.4.3, bekijken we enkele voordelen van het gebruik van aspecten.

2.4.1 Uittekenen krijtlijnen

In dit hoofdstuk gaan we op zoek naar een techniek die een (aanvaardbare) oplossing biedt voor de opgesomde problemen bij beveiliging. Daarbij komt nog de extra uitdaging om bij het beveiligen van medische data om te gaan met de complexiteit van een flexibele toegangscontrole. Ook in andere toepassingen (zoals bankinstellingen) kan een beveiligingsbeleid erg ingewikkeld zijn.

Als we dan concreet de lijnen uittekenen, krijgen we:

1. beheren van verstrengeling en verspreiding van de code
2. kunnen aanpassen aan onverwachte gevaren en veranderingen
3. mogelijkheid om een beveiligingsbeleid op een hoog niveau te realiseren

De vraag is nu of aspectgeoriënteerd programmeren ervoor kan zorgen dat deze doelen bereikt worden. Een antwoord op deze vraag kan in 2.4.2 gevonden worden.

2.4.2 Beveiliging als afgescheiden belang

We willen graag gebruik maken van geavanceerde scheiding van het belang beveiliging van de rest van de applicatie. Hierboven hebben we de lijnen uitgetekend voor de beveiliging waar we naartoe willen werken. We bekijken hier of we met AOP die doelen kunnen bereiken.

2.4.2.1 Beheren van verstrengeling en verspreiding van de code

In de vorige sectie zagen we dat AOP via *weaving* gelijk waar in de code een bijkomende functionaliteit kan toevoegen. Die bijgevoegde code zelf wordt dan centraal beheerd in de aspecten. Met andere woorden, wanneer men aspecten gebruikt voor de beveiliging dan kan men overweg met de verstrengeling van de code.

In 2.1.1 zagen we dat de code voor toegangscontrole op verschillende plaatsen (klassen) in een programma staat. We zouden dus een aspect kunnen ontwerpen dat de toegangscontrole verzorgt en ervoor zorgt dat de juiste functionaliteit op de juiste plaatsen wordt ingeweven in de uitvoering van het programma. Dat inweven kan zowel statisch als dynamisch gebeuren. Statisch betekent dat *tijdens de compilatie* de extra functionaliteit van het aspect in de code wordt geplaatst, terwijl bij dynamisch weven de functionaliteiten tijdens de uitvoering worden toegevoegd. De laatste optie is natuurlijk het meest interessant.

In het volgend hoofdstuk introduceren we *JAC (Java Aspect Components)*, een raamwerk dat dynamisch aspectgeoriënteerd programmeren toelaat en bekijken we in detail hoe deze onder andere het weven verwezenlijkt.

2.4.2.2 Aanpassen aan onverwachte gevaren en veranderingen

AOP helpt niet om de volledigheid van de bedreigingsanalyse te verbeteren, maar zorgt ervoor dat een systeem beter aanpasbaar wordt. Bij een verandering of toevoeging van een of ander (beveiligings-)mechanisme tijdens de levensduur van een programma moet men nu niet meer de volledige code bekijken.

Het grote voordeel van AOP hier is dat men met een aspect alles kan centraliseren. Voor de beveiliging kan men verschillende aspecten aanmaken, zoals bijvoorbeeld een aspect voor de authenticatie en een voor het beheren van de sessies. Als men de authenticatie wil verbeteren, dan moet men enkel dat aspect beschouwen. Het komt er dus op neer dat men de beveiliging zou moeten opdelen in aspecten zodat die aspecten volledig afgescheiden zijn van zowel de basisapplicatie als andere aspecten. Later in dit werk zullen we zien dat beveiliging bijna volledig kan afgescheiden worden maar dat er tussen de aspecten zelf in de beveiliging afhankelijkheden ontstaan (zie hoofdstuk 5).

2.4.2.3 Beveiligingsbeleid op hoog niveau

Op het eerste zicht biedt AOP niet onmiddellijk extra troeven die het mogelijk maken om een beleid op hoog niveau te ondersteunen. zo'n beleid vereist immers een expressieve taal. *Expressief* betekent dat het mogelijk moet zijn om een beleid uit te breiden met andere (vaak complexere) regels. Die complexere regels maken dan op hun beurt vaak gebruik van gegevens die niet onmiddellijk in de applicatie te vinden zijn. In hoofdstuk 3 proberen we via AOP een beleid te implementeren dat later gebruik maakt van een expressieve declaratieve beleidstaal die we in hoofdstuk 6 zullen uitwerken.

2.4.3 Voordelen

Wat volgt zijn enkele redenen uit de literatuur [10] waarom technieken op basis van AOP moeten toegepast en nog verder uitgebreid worden.

- Aanpasbare beveiliging zorgt voor eenvoudige ontwikkeling en beheer. Men kan bijvoorbeeld eerst een matig beveiligd systeem ontwikkelen en het daarna telkens beter en beter beveiligen. (van *get-it-all-right-the-first-time* naar *monitor-and-evolve*)
- Het zorgt ook voor betere begrijpelijkheid en het concentreren van de inspanning leidt tot minder bugs.
- Wanneer beveiligingsaspecten ontkoppeld zijn van andere aspecten kan er een grotere flexibiliteit - zowel onderhoudbaarheid (*maintainability*) als beheersbaarheid (*manageability*) - aan de dag gelegd worden.
- Het resulteert in minder ontwerp- en implementatiefouten door de duidelijke scheiding tussen de *programmeur (application developer)*, die met de ontwikkeling van de applicatie zelf bezig is, de *beveiligingsbeheerder (security manager)*, die het beleid opstelt, en diegene die de applicatie daarna gebruikt (*de deployer*).
- Daarnaast zijn er nog voordelen: betere herbruikbaarheid, gemakkelijker parallele ontwikkeling, ...

De voordelen die hierboven vermeld werden, zijn zeker een aanzet om aan aspectgeoriënteerd programmeren te doen. Op lange termijn zal een programmeur van een applicatie zich enkel moeten bezig houden met de functionaliteit ervan. Hij zal in een taal moeten werken die beveiligingsprincipes in se bevat of voor meer ingewikkelde beveiliging, een taal waarin een beveiligingsingenieur kan werken.

2.5 Conclusie

In dit hoofdstuk werden twee oorzaken voor de complexiteit van softwarebeveiliging besproken. Naast veilig coderen is er ook het feit dat de beveiligingslogica verstrengeld en verspreid is in de code van de onderliggende toepassing. We focussen ons in deze thesis op de tweede oorzaak. Om deze complexiteit te verminderen is er een principe beschikbaar, meerbepaald dat van scheiding van belangen. Die scheiding kan (grotendeels) bekomen worden door aspecten te gebruiken voor de implementatie.

Het uiteindelijke doel van deze thesis bestaat uit drie onderdelen: ten eerste moeten we de verstrengeling en verspreiding van de beveiligingslogica beheren, daarnaast moeten we de beveiliging kunnen aanpassen aan veranderende omstandigheden en tenslotte moeten we een beveiligingsbeleid op een hoog niveau kunnen realiseren.

Hoofdstuk 3

Beleidsstalen

In ons eerste hoofdstuk beschreven we waaraan medische gegevens moeten voldoen. Die vereisten zouden op een of andere manier *verwoord* moeten worden. Het beveiligingsbeleid moet dus kunnen vertaald worden naar een taal die tussen gebruiker en beveiliging staat. Wij zullen gebruik maken van een declaratieve beleidstaal om de regels van een flexibele toegangscontrole vorm te geven (restricties op toegang van een subject tot een object).

In 3.1 wordt uitgelegd wat van een beleidstaal allemaal verwacht mag worden. We beschrijven ook hoe we zo'n declaratieve taal kunnen integreren in een beveiligingsmechanisme zodat de beveiliging totaal onafhankelijk wordt van de eigenlijke applicatie. Om een indruk te geven van wat een beleidstaal concreet inhoudt, geven we in 3.2 twee voorbeelden van bestaande talen en hun tekortkomingen voor onze toepassing. Als mogelijke oplossing voor deze tekortkomingen stellen we de taal voor die wij in deze thesis zullen gebruiken voor de toegangscontrole. Dit gebeurt in sectie 3.3.

3.1 Declaratieve beleidstaal

Een beleidstaal is als het ware een interface tussen de beleidsmaker en de beveiliging zelf. Met beleidsmaker bedoelen we de persoon die de regels maakt, niet noodzakelijk iemand met een IT-achtergrond. Een beleidsmaker kan dus zowel personeelschef als programmeur zijn. Hier moeten we in het vervolg rekening mee houden. Het declaratieve karakter van een taal bestaat erin dat men duidelijk kan specificeren *wat* er moet gebeuren zonder dat men rekening moet houden met *hoe* alles concreet zal gerealiseerd worden.

In deze sectie gaan we eerst bekijken waarmee men moet rekening houden bij een beleidstaal voor flexibele toegangscontrole (3.1.1). Daarna wordt uitgelegd waarom precies een declaratieve taal gebruikt wordt en hoe die dan concreet kan worden ingebed (3.1.2).

3.1.1 Beleidstaal voor flexibele toegangscontrole

Hier gaan we dieper in op enkele aspecten van een beleidstaal voor flexibele toegangscontrole [9]. Expressiviteit en leesbaarheid (3.1.1.1) en groeperingen (3.1.1.2) zijn functionele vereisten, terwijl uitbreidbaarheid en herconfigureerbaarheid (3.1.1.3) eerder niet-functioneel zijn.

3.1.1.1 Expressiviteit versus leesbaarheid

Expressiviteit en leesbaarheid (of begrijpelijkheid) van een taal staan niet onmiddellijk in contrast met elkaar. Het is niet omdat een taal leesbaar is, dat ze totaal niet expressief is of omgekeerd. Maar om een hoge graad van expressiviteit te kunnen bereiken moet men meestal complexere structuren in de taal introduceren.

Zeker als men al op voorhand weet dat de taal zal gebruikt worden door mensen die leek zijn op vlak van programmeren, is de gebruiksvriendelijkheid van de taal erg belangrijk. Anderzijds kunnen regels van het beleid erg ingewikkeld zijn, als voorbeeld bij medische data : *een verpleger kan maar gegevens van een record lezen als de patiënt op de afdeling ligt waar de verpleger werkt én als hij hoofdverpleger is*. Een taal is nutteloos wanneer men bepaalde regels niet kan beschrijven. Er moet dus gestreefd worden naar een taal die de meeste (complexe) regels kan beschrijven zonder dat die al te ingewikkeld wordt.

Niet triviale regels bevatten ook *informatie omtrent de context*, zoals bij de regel in de vorige alinea het departement van de patiënt en de verpleger. Het moet dus mogelijk zijn op een bepaald niveau extra informatie uit het systeem te kunnen halen. Dit kan gaan over tijdafhankelijke gegevens (bv. al dan niet shift van de nachtploeg), eigenschappen van object of subject of andere kenmerken van het systeem.

3.1.1.2 Groeperen van objecten en subjecten

Een taal moet zowel overweg kunnen met specifieke regels als met algemene regels. Daarom is het handig om concepten met dezelfde eigenschappen te kunnen groeperen. In het geval van toegangscontrole gelden de meeste regels voor een verzameling van objecten of verzameling van subjecten.

Het groeperen van objecten doet men meestal in *domeinen (domains)*. Een domein bestaat dus uit objecten met gemeenschappelijke kenmerken. Wat die kenmerken precies zijn, is niet echt van belang. Het kunnen objecten zijn van bepaalde types, of met gelijke eigenschappen (bv. de waarde van bepaalde attributen), of objecten die in een bepaalde periode gecreëerd zijn, ...

Ook subjecten kan men groeperen. Die groeperingen gebeuren aan de hand van eigenschappen van de subjecten en worden vaak *rollen* genoemd. Ofwel wijst men rollen aan subjecten toe, ofwel krijgen subjecten rollen (eventueel dynamisch) toegewezen aan de hand van eigenschappen van die subjecten. Men zou natuurlijk ook kunnen domeinen maken voor subjecten, net zoals voor objecten, maar in de praktijk verkiest men rollen boven domeinen van subjecten. Met een rol bedoelen we bijvoorbeeld een verpleger, iemand van de administratie, ...

3.1.1.3 Uitbreidbaarheid en herconfigureerbaarheid

Bij het ontwikkelen van een taal moet men rekening houden met eventuele uitbreidingen van de taal zelf. Wanneer men bijvoorbeeld nieuwe beveiligingsconcepten wil introduceren, zou men deze zonder al te veel moeite in de taal moeten kunnen inpluggen. Bij *Ponder* bijvoorbeeld is dat geen probleem omdat de taal objectgeoriënteerd is; men kan van de bestaande objecten overerven (zie verder in 3.2.1).

Meestal is het niet nodig om bij het veranderen van het beleid een aanpassing door te voeren aan de taal zelf. Flexibele toegangscontrole is een vereiste, zeker in medische toepassingen omdat hier het beleid van nature dynamisch is (zie Hoofdstuk 1). Flexibele toegangscontrole houdt in dat men zonder veel moeite de toegangsregels kan herconfigureren.

3.1.2 Integratie declaratieve beleidstaal

Een beleid zegt *wat* er moet gebeuren in welke situaties, of met andere woorden, een beleid is van nature declaratief. Voor de gebruiksvriendelijkheid en de leesbaarheid is het aangeraden om de beleidstaal zo dicht mogelijk te laten aanleunen bij het beleid zelf. We hebben in 3.1.1 besproken welke functionaliteiten een beleidstaal voor toegangscontrole zou moeten hebben, maar zo'n taal moet ook geïntegreerd worden in een applicatie.

3.1.2.1 Binding tussen beleid en applicatie

Wij willen ervoor zorgen dat de beveiliging niet tussen de applicatie staat en omgekeerd, zodat de logica van de beveiliging zich niets moet aantrekken van de implementatie van de toepassing zelf.

In een declaratieve beleidstaal kan een gebruiker zeggen wat er moet gebeuren terwijl de feitelijke implementatie transparant is voor hem. Dit impliceert dat er een onderliggend mechanisme verantwoordelijk is voor het omzetten en uitvoeren van wat de gebruiker specificeerde. Dit onderliggend mechanisme kan men als een *connector* aanzien die een hoog-niveau beleidstaal uitvoert in een “lagere” taal (programmeertaal of eventueel vraagtaal bij gegevensbanken). Met *uitvoeren* bedoelen we controleren of het om een geoorloofde operatie gaat en zo ja het oproepen van de methode in de lagere taal. In het verdere verloop van dit hoofdstuk bedoelen we met *connector* wat we zojuist vermeld hebben. Dit is duidelijk een mogelijke oplossing, niet dé oplossing.

In concreto moet de connector dan bijvoorbeeld de acties die in de regels van het beleid voorkomen, linken aan concrete methodes in de applicatie. Niet alleen acties moeten gelinkt worden, het kan ook zijn dat er in het beleid attributen gebruikt worden. Hier moet dan de connector de juiste gegevens uit de applicatie halen.

3.1.2.2 Herconfigureerbaarheid beveiliging

Het herconfigureerbare dat we hiervoor in 3.1.1 wilden bereiken, wordt hier opgelost door de connector die telkens de regels verwerkt voor de toegangscontrole. Als men de regels wijzigt dan moet men er enkel voor zorgen dat de connector de gewijzigde regels gebruikt. Dit herconfigureren kan zowel statisch als dynamisch gebeuren. Statisch betekent dat men telkens

de beveiliging (eventueel enkel de toegangscontrole) moet inladen. Daarentegen bij een dynamisch herconfigureerbare toegangscontrole kan men de wijzigingen *at runtime* doorvoeren maar dit kan ervoor zorgen dat de toegangscontrole zelf minder performant werkt.

Het is duidelijk dat op deze manier enkel het onderliggend mechanisme (de connector dus) afhankelijk is van de implementatie. Wij werken zoals eerder gezien volgens het *bottom-up*-principe (dit werd uitgelegd in 2.3). We starten vanuit beveiliging van medische gegevens, maar willen eerder een algemene beveiliging ontwerpen. Een medische toepassing is een concrete implementatie maar de beveiliging die we ervoor schrijven willen we ook kunnen gebruiken voor andere toepassingen: zowel voor andere medische applicaties als voor systemen in een andere sector (zoals bijvoorbeeld in het bankwezen).

3.1.2.3 Naar totale onafhankelijkheid

Stel dat we twee applicaties hebben, bijvoorbeeld twee ziekenhuizen met elk een verschillend *business model*. Nu willen we één en hetzelfde beleid (via de taal) toepassen voor beide ziekenhuizen omdat ze zopas gefusioneerd zijn. Het zou erg handig zijn dat de taal er rekening mee houdt dat ze kan gebruikt worden voor verschillende implementaties. Het concept van de connector dat we introduceerden, zorgt wel voor de onafhankelijkheid tussen beleid en implementatie maar niet onmiddellijk voor de herbruikbaarheid.

Het is logisch dat voor twee ziekenhuizen met een totaal verschillend systeem niet dezelfde connector kan worden gebruikt. Als ze nu wel beiden bijvoorbeeld een Java-interface hebben dan zou het handig zijn dat men een en dezelfde herconfigureerbare connector heeft die men kan aanpassen aan de verschillende modellen. Dit laatste gaan we concreet uitwerken in het volgende hoofdstuk.

We gaan dus op zoek naar een beveiliging met een declaratieve taal die zo'n herconfigureerbare connector gebruikt. Het grote voordeel hiervan is dat er voor verschillende toepassingen hetzelfde consistente beleid kan worden gebruikt. Op deze manier kan men niet alleen één beveiliging voor meerdere implementaties gebruiken, maar ook wanneer de implementatie zelf verandert, dan moet men enkel de connector herconfigureren. Dit betekent dat de beveiliging (taal en connector) dan in principe totaal onafhankelijk is van de implementatie.

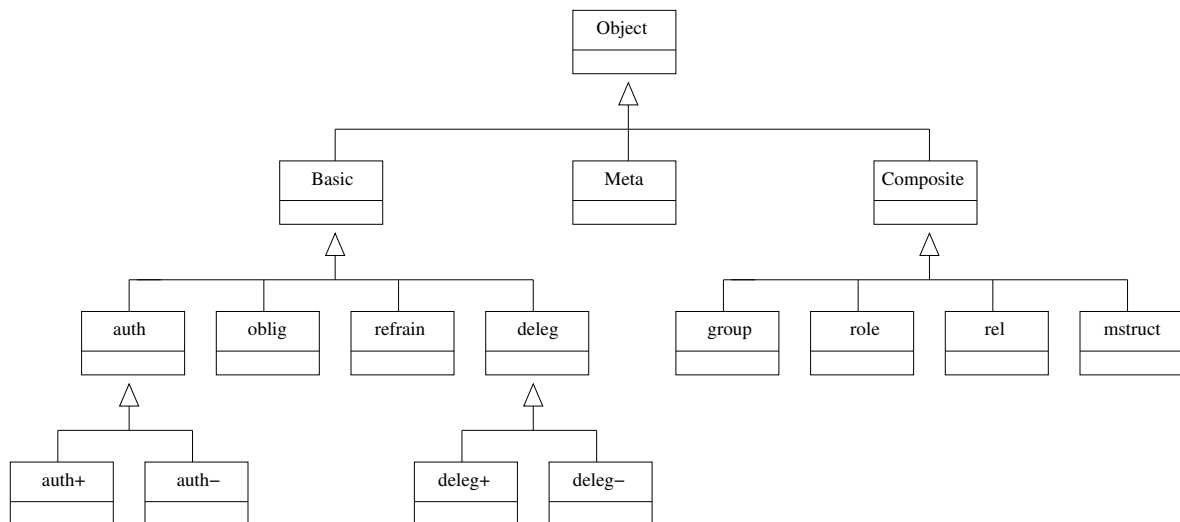
In de taal die wij in 3.3 voorstellen, trachten we de configureerbare binding en de daaruit volgende totale onafhankelijkheid te integreren. Maar eerst beschouwen we twee bestaande beleidstalen, met name *Ponder* en *SPL*.

3.2 Bestaande beleidstalen

Om te verduidelijken wat een beleidstaal juist inhoudt, bespreken we kort twee bestaande talen. We kozen voor Ponder omdat deze veel gebruikt wordt voor het beleid en de toegangscontrole in gedistribueerde toepassingen, en SPL omdat het ook een taal is die geschikt is voor toegangscontrole.

3.2.1 Ponder

Ponder [13] is een declaratieve, objectgerichte taal waarin men het beveiligingsbeleid en -beheer kan specificeren. We introduceren eerst enkele termen. Ponder gebruikt de term *subject* om alle gebruikers of geautomatiseerde componenten (zoals een onderdeel van de beveiliging dat zelf toegang wil tot een beveiligd object) aan te duiden. Een subject roept dan zichtbare methodes op op een *doelobject* (*target*) terwijl een *domein* een groepering is van objecten.



Figuur 3.1: Ponder Object Meta Model [13]

Het objectgerichte van Ponder zit in de structuur van de beveiligingsprimitieven (zie figuur 3.1). Het grote voordeel hiervan is dat men eenvoudig nieuwe uitbreiding kan toevoegen door gewoon over te erven van een bestaande type. Hier gaan we nu kort de voor ons belangrijkste beveiligingsprimitieven verduidelijken. We delen ze op in vier categorieën: regels voor toegangscontrole (3.2.1.1), beleid van verplichtingen (3.2.1.2), beperkingen (3.2.1.3) en groepen (3.2.1.4).

3.2.1.1 Regels voor toegangcontrole

Voor toegangscontrole voorziet Ponder regels voor autorisatie, delegatie, filteren van informatie en opleggen van verboden. We vermelden van al deze regels kort wat ze doen. Maar eerst gaan we wat dieper in op autorisatieregels.

In Ponder kan men zowel positieve als negatieve *autorisatieregels* opstellen. In principe kan een volledig beleid worden opgesteld met enkel positieve regels, maar negatieve geven de taal extra expressiviteit. Daarenboven gebruiken we nu en dan ook in het dagelijks leven negatieve regels. Het gebruik van negatieve regels brengt wel met zich mee dat er conflicten kunnen ontstaan met positieve regels. De algemene vorm van een autorisatieregel ziet er uit zoals in figuur 3.2.

```

1 inst (auth+ | auth-) policyName "{"
2     subject      [<type>]    domain-Scope-Expression;
3     target       [<type>]    domain-Scope-Expression;
4     action       action-list;
5     [ when      constraint-Expression; ]
6 "}"

```

Figuur 3.2: Syntax autorisatieregel

Eerst specificeert men of het om een positieve respectievelijk negatieve regel gaat waarna de regel een naam krijgt (lijn 1). Bij het subject (lijn 2) geeft men het domein van de subject en eventueel het type op, bij het target (lijn 3) wordt gewerkt volgens hetzelfde stramien. Verder geeft men de lijst op van acties (lijn 4), en tenslotte volgt eventueel een beperking (lijn 5).

De regel in figuur 3.3 geeft aan ieder subject uit het domein */NetworkAdmin* het recht om op alle objecten van type *PolicyT* uit het domein */Nregion/switches* de operaties *load()* en *remove()* uit te voeren.

```

inst auth+ switchPolicyOps {
    subject      /NetworkAdmin;
    target <PolicyT> /Nregion/switches;
    action       load(), remove();
}

```

Figuur 3.3: Voorbeeld autorisatieregel

Via een *delegatieregel* kan een subject de rechten die het zelf heeft, doorgeven aan een ander subject. Een subject kan enkel maar de rechten die het bezit (of een aantal ervan) doorgeven. Men kan ook de geldigheidsduur van de delegatie beperken. *Filteren van informatie* komt er op neer om aan de hand van de input- en outputparameters van een actie te beslissen of ze al dan niet mag uitgevoerd worden, men kan ook de teruggeefwaarde aanpassen.

Verbodsregels (refrains) bepalen welke acties subjecten niet mogen uitvoeren. Hun syntax is gelijkaardig aan die van de negatieve autorisatieregels. Het verschil is echter dat een verbod eerder zegt wat niet toegelaten is aan de kant van het subject, terwijl een negatieve autorisatieregule vanuit het object uitgaat.

3.2.1.2 Beleid van verplichtingen

Regels in dit beleid specificeren welke acties door de beheerders moeten genomen worden wanneer bepaalde gebeurtenissen plaats vinden. Ze reageren op veranderende omstandigheden. Men moet definiëren welke activiteiten subjecten (meestal beheerders van het systeem) moeten uitvoeren op objecten in het doeldomein.

We tonen de werking aan de hand van een voorbeeld (zie figuur 3.4). Wanneer een gebruiker probeert in te loggen en daar drie maal op rij niet in slaagt (lijn 2), dan moet de beheerder van de beveiliging (lijn 3) ervoor zorgen dat het ID van die gebruiker (lijn 4) wordt uitgeschakeld, en wordt gelogd (lijn 5). Op die manier kan later nog worden nagegaan waarom die gebruiker geen geldig ID meer heeft.

```

1  inst oblig loginFailure {
2      on          3*loginfail(userid);
3      subject     s = /Nregion/SecAdmin;
4      target <userT> t = /Nregion/users ^ {userid};
5      do          t.disable() -> s.log(userid);
6  }
```

Figuur 3.4: Voorbeeld verplichting

3.2.1.3 Beperkingen

Beperkingen leggen condities op onder dewelke het beleid geldig is. Er zijn in Ponder twee soorten beperkingen, namelijk *basisconstraints* en *metabeleidsregels*. Een deelverzameling van de *Object Constraint Language (OCL)* wordt gebruikt om de beperkingen uit te schrijven.

Basisconstraints zijn zelf een conjunctie van basisuitdrukkingen. Alles is uitgedrukt in termen van een predicaat en als de beperking naar *true* evalueert, dan is het beleid toepasbaar. Er zijn verschillende types van basisconstraints te onderscheiden: deze die afhankelijk zijn van de toestand van het subject of van het doelobject, van de parameters van de actie of event en van de geldigheidsduur (tijdsbeperking). De regel in figuur 3.5 beschrijft dat testingenieurs (lijn 2) geen performantietest (lijn 3) mogen uitvoeren op routers (lijn 4) als de ingenieur nog in opleiding en jonger dan 25 is (lijn 5).

Naast de basisconstraints zijn er ook beperkingen die van toepassing zijn op een groep van regels, men noemt deze *metabeleidsregels (Meta-policies)*. Via deze beperkingen kan men bijvoorbeeld simultane uitvoering van conflicterende regels vermijden. In deze tekst gaan we hier niet verder op in.

```
1 inst auth- testRouters {
2   subject      s = /testEngineers;
3   action       performance_test();
4   target       /routers;
5   when         s.role = "trainee" & s.age < "25";
6 }
```

Figuur 3.5: Voorbeeld basisconstraint

3.2.1.4 Groeperen van regels

In Ponder is het mogelijk om gerelateerde regels te groeperen. Op die manier kan men ze beter beheren en wordt zo de uitbreidbaarheid verhoogd. De criteria om regels te groeperen zijn niet opgelegd. Men kan bijvoorbeeld alle regels voor hetzelfde domein van objecten (targets) samennemen, of alle regels voor het inloggen in een groep brengen. Zo'n verzameling wordt dan een *groep* genoemd. De herbruikbaarheid hier bereikt men als volgt. Men kan de groepen specificeren als types met daarin alle regels en dan kan men dat type telkens instantiëren om de gehele groep te beschrijven.

Een *rol* is in Ponder een speciaal geval van een groep. Een rol is een semantische verzameling (groep) van regels met één en hetzelfde domein voor het subject. Een rol kan net als een groep zowel toegangsregels, verplichtingen, verbodsregels als delegaties bevatten. De volgende rol (figuur 3.6) beschrijft wat een subject uit het domein allemaal als regels heeft.

```
Type role ServiceEngineer (CallsDB callsDb) {
  inst oblig serviceComplaint { ... }
  inst oblig deactivateAccount { ... }
  inst auth+ serviceActionsAuth { ... }
  // other policies
}
```

Figuur 3.6: Voorbeeld rol

3.2.2 SPL

Een andere beleidstaal die we bekeken hebben is SPL. We bespreken eerst de structuur van de taal in 3.2.2.1, daarna zeggen we iets over het oplossen van conflicten (3.2.2.2) en we eindigen in 3.2.2.3 met het bekijken hoe rollen in SPL kunnen worden voorgesteld.

3.2.2.1 Structuur

SPL [14] staat voor *Security Policy Language*, een beleidgeoriënteerde constraintgebaseerde taal. Ze bestaat uit vier basisblokken: entiteiten, verzamelingen (*sets*), regels en beleiden (*policies*). Het fundamentele element van de taal is de regel. Deze drukt beperkingen uit in termen van relaties tussen entiteiten en verzamelingen. Beleiden zijn complexe beperkingen die voortkomen uit het samenstellen van regels en verzamelingen in logische eenheden.

a) *Entiteiten*

Entiteiten zijn getypeerde objecten met een expliciete interface, via dewelke eigenschappen kunnen opgevraagd worden. Entiteiten kunnen zich zowel binnen als buiten de beveiligings-service bevinden. De eerste worden mee beveiligd, terwijl de andere niet worden beschouwd bij het toepassen van de beveiliging.

Het risico bij het werken met externe entiteiten bestaat erin dat het opvragen ervan niet altijd als veilig kan beschouwd worden. Om die reden moet geverifieerd worden dat elke operatie, afhankelijk van eigenschappen van externe entiteiten, enkel toegelaten is als het opvragen van die eigenschappen geautoriseerd is. Voorbeelden van externe entiteiten zijn gebruikers, bestanden en gebeurtenissen. SPL legt geen beperkingen op voor de eigenschappen van entiteiten, integendeel: de taal buit dit uit om de kracht van haar beleiden te verhogen.

b) *Verzamelingen*

In 3.1.1.2 zagen we dat een declaratieve taal nood heeft aan enige vorm van groepering om de nodige abstractie van compactheid, generalisatie en schaalbaarheid te verkrijgen. In SPL introduceerde men daarom verzamelingen van identiteiten. Zonder verzamelingen zou iedere regel voor elke entiteit waarop die van toepassing is, moeten herhaald worden. Net als entiteiten kunnen ook verzamelingen zowel intern als extern zijn. Sommige externe verzamelingen kunnen uiterst nuttig blijken voor de definitie van beleiden, zo is er bijvoorbeeld de verzameling van alle gebruikers van het systeem.

In SPL worden twee soorten verzamelingen ondersteund: categorieën en groepen. *Categorieën* zijn gedefinieerd a.h.v. de classificatie van entiteiten volgens hun eigenschappen. *Groepen* daarentegen worden bepaald door het expliciet toevoegen en verwijderen van hun elementen.

c) *Regels die beperkingen opleggen (constraint rules)*

De taal is een samenstelling van individuele regels die allen logische uitdrukkingen zijn. Deze kunnen drie mogelijke waarden aannemen: *allow* (toegelaten), *deny* (niet toegelaten) en *notapply* (niet van toepassing). Een regel kan bovendien eenvoudig of samengesteld zijn.

Een *eenvoudige regel* bestaat uit twee logische uitdrukkingen, de ene om het toepassingsdomein van de regel vast te leggen en de andere om te beslissen over de aanvaardbaarheid van de event:

```
deny: true :: false
```

Een samengestelde regel bestaat uit verschillende eenvoudige regels die met elkaar verbonden zijn m.b.v. drie logische operatoren: conjunctie (AND), disjunctie (OR) en negatie (NOT). Dit alles wordt uitgedrukt in een tri-waarde algebra, waarin de waarde *'notapply'* als neutraal element optreedt. Een voorbeeld van een samengestelde regel (*OwnerRule* en *DutySep* zijn eenvoudige regels):

```
OwnerRule AND DutySep OR deny;
```

d) *Beleiden*

Een SPL-beleid is een groep regels en verzamelingen die een bepaald domein van events beheren. Elk beleid heeft een *QueryRule* die alle regels in het beleid relateert. Deze regel gebruikt de gedefinieerde algebra om te specificeren welke regels moeten opgelegd worden en hoe dit moet gebeuren.

Sommige verzamelingen kunnen als parameter meegegeven worden aan het beleid. Hierdoor kunnen abstracte beleiden geconstrueerd worden, die meerdere keren geïnstantieerd worden met andere parameterwaarden. Op die manier kan men generische beleiden opstellen.

3.2.2.2 Oplossen van conflicten

In SPL kunnen tegelijk positieve en negatieve beperkingen uitgedrukt worden. Deze mogelijkheid bevordert de expressiviteit van het beleid. Het probleem hiermee is dat er conflicten ontstaan tussen tegenstrijdige regels. Dit kan worden opgelost door het introduceren van impliciete voorrangsalgoritmes, die bepalen welke regel domineert over de andere.

In SPL wordt de beveiligingsadministrator verplicht beleiden te combineren tot een unieke structuur die vrij is van conflicten. Elk actief beveiligingsbeleid moet in de *hiërarchische delegatieboom* van beleiden zitten. Wanneer dan twee regels conflicteren, is er in die boom (op een hoger niveau) een tri-waarde uitdrukking die deze regels bevat en het conflict oplost.

3.2.2.3 Rollen

SPL ondersteunt ook rollen om aan RBAC (Role Based Access Control) te kunnen doen. Rollen in SPL zijn beleiden op zich die in andere beleiden gebruikt kunnen worden. Rollen kunnen samengesteld zijn uit verschillende verzamelingen en beperkingen. De eenvoudigste vorm heeft maar twee verzamelingen: een met de gebruikers die de rol kunnen hebben, en een andere met de gebruikers die de rol effectief hebben. Enkel de gebruikers in de eerste verzameling kunnen dan in de tweede terecht komen. Om de nodige autorisaties te hebben om de rol te mogen spelen, moet de gebruiker zich in de tweede verzameling bevinden.

De specificatie van zo'n eenvoudige rol kan er uit zien zoals in figuur 3.7. Die geeft aan dat enkel de gebruikers die in de Active-verzameling zitten de nodige autorisaties mogen hebben om de rol te spelen. Die rol kan dan gebruikt worden bij toegangscontrole (zie figuur 3.8). Het voorbeeld geeft aan dat gebruikers van "localhost" de rol "Clerk" mogen spelen en dat elke "Clerk" toegang kan krijgen tot "invoices"

```
policy simpleRole (user set Authorized, user set Active)
{
    // Events die een gebruiker in de Active-verzameling toevoegen
    // worden enkel toegelaten als die gebruiker zich in de
    // Authorized-verzameling bevindt.
    ?simpleRole: ce.action.name = "insert" & ce.target = Active
                :: ce.parameter[1] IN Authorized;
}
```

Figuur 3.7: Een eenvoudige rol

```
policy Clerk
{
  // Alle gebruikers van localhost zijn leden van RoleUsers
  user set RoleUsers = AllUsers@{.host = localhost};

  // Invoices zijn alle objecten van het type invoice
  object set Invoices = AllObjects@{.doctype = "invoice"};

  // De verzameling met gebruikers die de rol spelen begint leeg
  user set ActiveGroup = {};

  // Leden van RoleUsers mogen de Clerk-rol spelen
  ClerkRule: new simpleRole(RoleUsers, ActiveGroup);

  // Alle leden uit ActiveGroup kunnen toegang krijgen tot Invoices
  InvoiceRule: new ACL(ActiveGroup, Invoices, AllActions);

  ?Clerk: ClerkRule AND InvoiceRule;
}
```

Figuur 3.8: Toepassing van rol

3.2.3 Conclusie

Ponder en SPL zijn expressieve talen. Het kunnen opstellen van zowel positieve als negatieve regels werkt dat nog in de hand. Anderzijds kan dit voor contradicties zorgen waardoor het beslissingsmechanisme zelf gecompliceerder wordt. In medische toepassingen spreekt men niet onmiddellijk van negatieve toegangsregels, in tegendeel, men verleent toegangsrechten meestal afhankelijk van de rol van de persoon. Zoals wettelijk opgelegd is, zijn medische gegevens strikt privé behalve in bepaalde gevallen. Die uitzondering kan men het best beschrijven met enkel positieve regels.

Beide talen voldoen ruimschoots aan de voorwaarden voor een flexibele toegangscontrole-taal (deze werden beschreven in 3.1.1). Naast expressiviteit bieden ze concepten van groeperingen aan. De configureerbaarheid van de talen is geen probleem terwijl hun gebruik eenduidig blijft. Het grote probleem voor onze toepassing is echter dat ze geen binding van toegangslogica aan een applicatie aanbieden en dat daardoor ook de beveiliging niet (bijna volledig) onafhankelijk is van de applicatie. Het ontbreken van die binding is de reden dat we op zoek gaan naar een nieuwe taal met bijhorende connector die dan wel aan onze eisen voldoet.

3.3 Binding via het Uitgebreid View-connectormodel

In deze sectie stellen we niet alleen een taal voor, maar kijken we vooral hoe we die connector die de beveiligingslogica met de applicatie verbindt, kunnen modelleren. Hierbij willen we al duidelijk stellen dat de taal die we zullen gebruiken zeker niet de mogelijkheden bezit als bijvoorbeeld Ponder. We concentreren ons op de connector, in hoeverre men deze als één geheel kan beschouwen.

Eerst introduceren we het View-connectormodel (3.3.1) en bouwen we dit verder uit in 3.3.2. Tenslotte gaan we voor een eenvoudige applicatie enkele regels opstellen volgens het voorgestelde model (3.3.3).

3.3.1 Het View-connectormodel

De taal die we gebruiken werkt met het concept van *view-connectoren* zoals het wordt voorgesteld in [15]. Het probleem van bestaande systemen die gecentraliseerde toegangscontrole aanbieden is dat ze niet opgeven hoe de opgestelde regels in een applicatie moeten geïntegreerd worden. Het gebruik van *access-interfaces* en bijhorende *view-connectoren* kan dit verhelpen.

3.3.1.1 Access-interface

Om ervoor te zorgen dat de regels volledig implementatie-onafhankelijk zijn, introduceren we het concept van een *access-interface*. Een access-interface baseert zich enkel op informatie die relevant is voor de toegangscontrole. Zo een interface is meestal hetzelfde binnen één instelling maar kan wel afhankelijk zijn van het applicatiedomein. Voor de beveiliging van een bankinstelling zal men normaal andere access-interfaces gebruiken dan bijvoorbeeld voor een ziekenhuis. De regels worden dan ook uitgedrukt in termen van access-interfaces.

Een access-interface zorgt voor een abstracte benadering van de applicatie. Bijvoorbeeld “klant met lage credibiliteit” voor het toewijzen van een lening in een bankinstelling, of “toegang in noodgeval” in een medische toepassing kunnen in een access-interface bijgehouden worden. In eerste instantie bevat de access-interface attributen. Met *attributen* bedoelen we hier niet attributen van de applicatie maar wel attributen voor de toegangscontrole. Het zijn gegevens die het beslissingsmechanisme van de toegangscontrole kunnen beïnvloeden, meer nog, de toegangsregels kunnen enkel deze attributen gebruiken. *Actions* daarentegen zijn een opsomming van wat toegelaten is uit te voeren op die interface. Die acties zijn van belang voor de eigenlijke toegangsregels.

3.3.1.2 View-connector

Een *view-connector* implementeert een access-interface net zoals een klasse in Java een bepaalde interface implementeert. Een access-interface biedt maar een abstracte benadering, terwijl een view-connector de acties en attributen concreet invult. View-connectoren verbinden de access-interface met de verschillende implementaties. Per interface kunnen er meerdere connectoren zijn: ze worden per object ingevuld. Later zullen we in detail zien hoe dat gebeurt, maar hier vermelden we enkel dat op deze manier de regels onafhankelijk blijven van de eigenlijke implementatie.

Daarnaast kan men ook, zoals het hoort bij een beleidstaal, objecten groeperen in domeinen. *Domeinen* (*domains*) worden gebruikt om een verzameling van doelobjecten van een regel voor te stellen. Gezien het feit dat in een domein specifiek het type van de objecten moet worden meegegeven, zijn ook domeinen implementatieafhankelijk.

3.3.1.3 Overzicht model

Voor de duidelijkheid schetsen we nog heel kort eens wat er volgens het View-connectormodel gebeurt. De regels van de toegangscontrole worden op een hoger niveau geplaatst. Ze zijn dus van toepassing op access-interfaces en niet op de applicatie zelf. De view-connectoren binden dan de access-interfaces met de eigenlijke toepassing.

Het View-connectormodel biedt enkel maar interfaces aan voor het object. Dit betekent concreet dat men in de regels niets kan gebruiken over het subject, tenzij men net zoals in Ponder of SPL in de regels alle implementatieafhankelijke gegevens invult. Dit is mede de reden waarom we het View-connectormodel willen uitbreiden. Naast interfaces voor subjecten, willen we ook rollen en *vereisten* (*requirements*) introduceren zodat onze taal expressiever wordt.

3.3.2 Uitgebreid View-connectormodel

Er is een access-interface aan de objectzijde, maar wat als men ook gegevens die relevant zijn voor de toegangscontrole aan de kant van het subject wil introduceren? Denk maar bijvoorbeeld aan gevallen waar men wil weten in welke afdeling de verpleger werkt die gegevens van een patiënt wil lezen. We zien ook de nood om de notie van rollen in de taal te integreren.

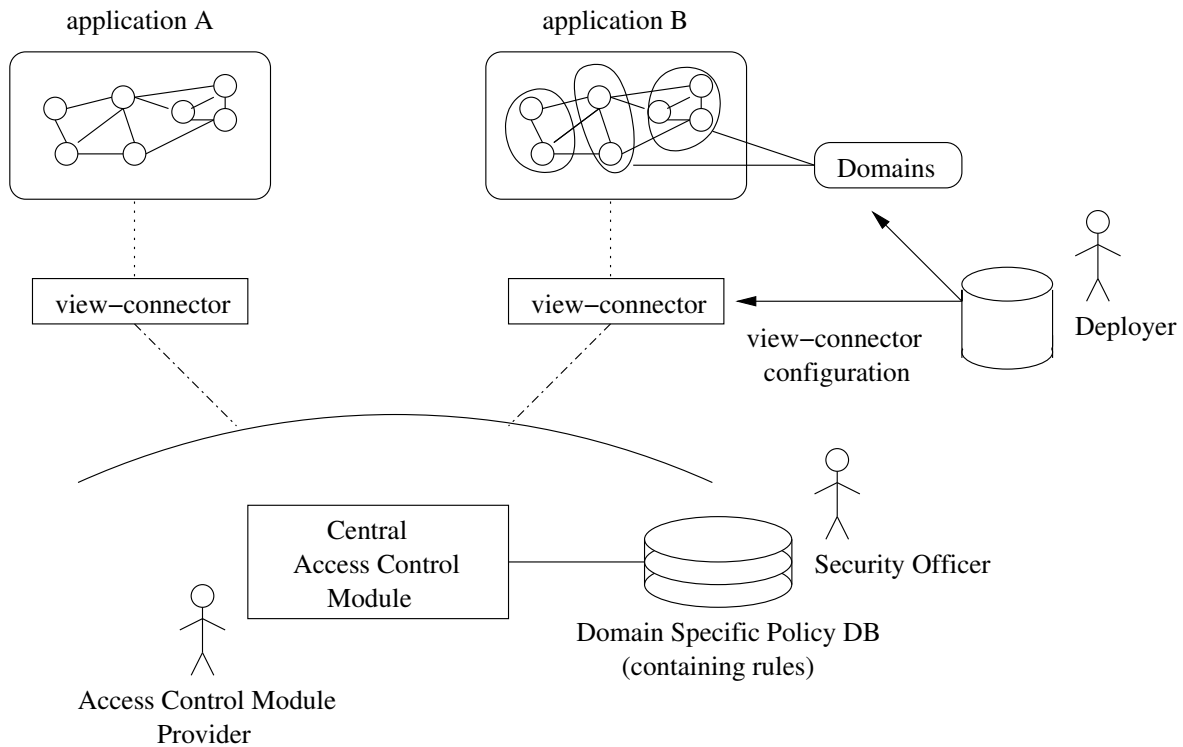
Bijkomende gegevens opvragen van het subject is niet mogelijk in [15], daarom introduceren we aan de subjectzijde ook een interface, namelijk de *subject-interface*. In tegenstelling tot access-interfaces moet deze interface geen acties voorzien omdat juist het subject de acties uitvoert. De subject-interface bevat bijgevolg enkel attributen. Omdat in werkelijkheid de toegangscontrole vaak gebaseerd is op rollen, is het dan ook aangewezen om in onze taal rollen in te passen zodat men aan rollen bepaalde rechten kan toewijzen.

Wanneer we nu alles op een rijtje zetten, dan moet een regel rekening houden met de access-interface, subject-interface, het domein van het object en de rol van het subject. Daarbij kunnen we in de regel informatie van zowel subject als object gebruiken.

3.3.2.1 Overzicht werking

In figuur 3.9 visualiseren we de werking van het (Uitgebreid) View-connectormodel. In het model zien we drie beheerders, een *deployer*, een *security manager* en een *access-control module provider*. Een *security manager* is de persoon die verantwoordelijk is voor het opstellen van de regels, niet noodzakelijk iemand met een informatica-achtergrond (bv. een personeelschef). Eventueel kan deze persoon ook de rollen toewijzen. De *access-control module provider* is dan de maker van het eigenlijke controlemechanisme. De *deployer* vult dan voor een bepaalde applicatie alle connectoren en domeinen in. In een kleinere onderneming kunnen deze drie functies door één persoon ingevuld worden, met als gevolg dat die voldoende kent van zowel de applicatie als de beveiliging. Bij grotere instanties, zoals een ziekenhuis is dit zeker niet

altijd het geval. Daardoor is het best mogelijk dat bijvoorbeeld een deployer (bijna) niets afweet van de beveiliging, wat het voor hem moeilijk maakt om er dan ook rekening mee te houden bij het ontplooiën van de toepassing.



Figuur 3.9: View-connectormodel [15]

De figuur toont ook dat men voor verschillende applicaties (A en B) telkens de interfaces via de connectoren moeten implementeren, maar op deze manier wel gebruik kunnen maken van hetzelfde centrale controlemechanisme.

3.3.3 Regels volgens Uitgebreid View-connectormodel

Hier kijken we hoe we regels kunnen opstellen volgens het Uitgebreid View-connectormodel. We zullen dit doen aan de hand van een beveiliging voor een eenvoudige applicatie. Een overzicht van de toepassing kan in figuur 5.7, op p. 64 gevonden worden.

We specificeren enkel positieve regels, omdat die het meest relevant zijn voor de in hoofdstuk 5 voorgestelde applicatie. Toegang tot medische gegevens is namelijk in principe verboden, behalve wanneer uitdrukkelijk wordt vermeld dat de gebruiker wel toegang krijgt. Die vermeldingen zijn in positieve regels gegoten (zie eerder in 1.3.1 op p. 15).

3.3.3.1 Opstellen van de interfaces

Er zijn dus zowel interfaces aan de kant van het subject (in dit geval `Person`) als voor het object (`Record`), maar bij de interface maken we nog abstractie van de implementatie.

```
<access-interface>
  <interface>
    PatientInterface
  </interface>
  <attributes>
    <attribute>
      owner
    </attribute>
  </attributes>
  <actions>
    <action>
      readAccess
    </action>
  </actions>
</access-interface>
```

In onderstaand voorbeeld hebben we een access-interface met naam `PatientInterface`. Die interface heeft een attribuut `owner` dat de maker van het record moet voorstellen. Men kan maar één actie uitvoeren op de interface, namelijk `readAccess`. Het spreekt voor zich dat we meerdere acties en attributen kunnen hebben per interface. Als subject-interface hebben we `ClinicianInterface` met het attribuut `department`:

```
<subject-interface>
  <interface>
    ClinicianInterface
  </interface>
  <attributes>
    <attributes>
      <attribute>
        department
      </attribute>
    </attributes>
  </subject-interface>
```

3.3.3.2 Maken van een regel

We maken onze beveiliging zeer eenvoudig: ze bestaat uit maar één regel (zie hieronder) die zegt dat enkel dokters die eigenaar zijn van een record, dat record mogen lezen. Dat “eigenaar zijn” werd gespecificeerd in de access-interface `PatientInterface`.

De regel is enkel van toepassing voor de actie `readAccess` door een subject waarvoor er een connector is die `ClinicianInterface` implementeert, in dit geval dus voor `Person`. Het object moet in het domein `PatientDomain` zitten en een connector hebben van `PatientInterface`. Daarenboven moet het subject dan nog de rol `Doctor` hebben. Alleen als alle voorgaande voorwaarden zijn vervuld, worden de vereisten (`requirements`) gecontroleerd. In dit concrete geval moet het subject de eigenaar zijn van het object.

```

<policy-rule>
  <domain>
    PatientDomain
  </domain>
  <access-interface>
    PatientInterface
  </access-interface>
  <subject-interface>
    ClinicianInterface
  </subject-interface>
  <role>
    Doctor
  </role>
  <requirements>
    <requirement>
      ClinicianInterface = PatientInterface.owner
    </requirement>
  </requirements>
  <actionList>
    <action>
      readAccess
    </action>
  </actionList>
</policy-rule>

```

3.3.3.3 Opstellen van connectoren en domeinen

We gaan nu bovenstaande interfaces invullen aan de hand van de eenvoudige applicatie. We hebben hier voor de duidelijkheid een eenvoudige applicatie genomen, maar in het volgende hoofdstuk zullen we connectoren opstellen van complexere modellen, dit om nog maar eens te duiden op het feit dat de interfaces volledig onafhankelijk zijn van concrete implementaties.

Hieronder wordt een view-connector voorgesteld voor `PatientInterface` voor een object van het type `Record`. Het attribuut `owner` is het resultaat van de inspector `getResponsibleDoctor()`, terwijl de actie `readAccess` de operatie `read()` voorstelt.

```

<view-connector>
  <type>
    application.record.Record
  </type>
  <access-interface>
    PatientInterface
  </access-interface>
  <attributes>
    <attribute>
      <name>
        owner
      </name>
      <value>
        getResponsibleDoctor()
      </value>
    </attribute>
  </attributes>

```

```

        <actions>
            <action>
                <name>
                    readAccess
                </name>
                <value>
                    read():void
                </value>
            </action>
        </actions>
    </view-connector>

```

Hierna wordt de subject-interface `ClinicianInterface` gespecificeerd voor het type `Person` met als waarde voor het attribuut `department` gewoon de afdeling van de hoofdrol van de oproeper.

```

<view-connector>
    <type>
        application.Person
    </type>
    <subject-interface>
        ClinicianInterface
    </subject-interface>
    <attributes>
        <attribute>
            <name>
                department
            </name>
            <value>
                getPrincipalRole().getDepartment()
            </attribute>
        </attributes>
    </view-connector>

```

Vervolgens maken we een domein aan met daarin de types `CardioRecord` en `Patient`:

```

<domain>
    <name>
        PatientDomain
    </name>
    <typeList>
        <type>
            application.Patient
        </type>
        <type>
            application.record.CardioRecord
        </type>
    </typeList>
</domain>

```

3.4 Besluit

We zagen dat bestaande talen als Ponder en SPL geen sluitende oplossing bieden voor onze toepassing omdat ze voor hun regels niet aangeven hoe ze kunnen toegepast worden op de onderliggende applicatie. De taal die we in de loop van dit hoofdstuk ontwikkelden is daarentegen wel totaal onafhankelijk maar zeker nog niet zo geavanceerd als de bestaande talen. We hebben bijvoorbeeld nog geen delegatie of kunnen geen verplichtingen opleggen. Het belangrijkste is dat we een raamwerk, namelijk het Uitgebreid View-connectormodel, hebben dat de algemene beleidstaal verbindt met de specifieke applicatie(s). Later kan de voorgestelde taal verder uitgebreid worden om dezelfde functionaliteiten te bevatten als de huidige gebruikte systemen, maar dit valt buiten het doel van onze thesis.

Hoofdstuk 4

Java Aspect Components

Java Aspect Components (JAC) [16] is een raamwerk dat een verzameling concepten aanbiedt die gedistribueerd en dynamisch AOP (Aspect Oriented Programming) toelaat. JAC is door Renaud Pawlak tijdens zijn doctoraat ontwikkeld [17]. De reden waarom we hier JAC bespreken, is dat we dit raamwerk willen gebruiken om tot een aspectgeoriënteerde implementatie te komen.

Er zijn twee niveaus om met JAC aan aspectgeoriënteerd programmeren te doen. Enerzijds is er het *programmeerniveau* waar nieuwe aspecten geprogrammeerd worden. Anderzijds kan ook op het *configuratieniveau* gewerkt worden. Bestaande aspecten worden dan aangepast om ze te laten werken in applicaties. Dit niveau wordt ondersteund door een configuratietaal met een generische syntax die de programmeur toelaat configuratiemethodes op te roepen op bestaande aspecten.

In wat volgt worden eerst twee basisconcepten van JAC uitgelegd, namelijk aspect componenten, in sectie 4.1, en dynamische wrappers, in sectie 4.2. Daarna wordt in sectie 4.3 de architectuur van het raamwerk bestudeerd.

4.1 Aspect componenten

Aspect componenten zijn implementatie-eenheden die bijkomende karakteristieken, die doorheen een verzameling basisobjecten voorkomen, definiëren. Een programmeur van aspect componenten kan het basisprogramma op drie manieren beïnvloeden:

1. Hij kan de semantiek van de basisklassen uitbreiden met definities van structurele meta-informatie. In JAC is er een runtime meta-model gedefinieerd voor elke applicatie: *RTTI (Run-Time Type Information)*. Elk element van de RTTI is een element van het basisprogramma (zoals klassen, methodes, collecties, ...) en wordt door JAC zelf gegenereerd.
2. Hij kan ook een intern gedefinieerde Meta-Object Protocol (MOP)-interface implementeren die toelaat dat aspect componenten reageren op sommige events die binnen het systeem plaats vinden (zoals de lancering van een applicatie, een system shutdown, e.a.).

3. Tenslotte kan hij pointcuts construeren, meerbepaald door extra handelingen toe te voegen voor, na of rond de uitvoering van basismethodes. Pointcuts controleren het dynamische deel van de applicatie, in tegenstelling tot de RTTI die structurele wijzigingen toelaat per klasse.

Hierna wordt uitgelegd hoe aspect componenten geprogrammeerd worden (4.1.1), en op het einde van deze sectie wordt getoond hoe ze geconfigureerd kunnen worden (4.1.2).

4.1.1 Programmeren

We beschouwen het voorbeeld in figuur 4.1.

```
1 public class AccessAC extends AspectComponent {
2     public AccessAC() {
3         aw = new AccessWrapper(this);
4     }
5
6     public void restrictAccess(String objects, String classes,
7                               String methods, String wrappingMethod) {
8         this.pointcut(objects, classes, methods, aw, wrappingMethod, null);
9     }
10
11    public void readConnectorList(String connectorsFilename) {
12        aw.setInterfaceConnectorList(getConnectorList(connectorsFilename));
13    }
14
15    private AccessWrapper aw;
16 }
```

Figuur 4.1: Voorbeeld aspect component

Elke aspect component erft over van de klasse `AspectComponent` (lijn 1). Bij de constructie kan de programmeur de semantiek van het basisprogramma wijzigen door de klassen attributen te geven of door pointcuts te construeren. De eigenlijke globale wijziging van de code van het basisprogramma gebeurt met de definitie van een pointcut (lijn 8), die extra code toevoegt voor, na of rond methodes.

Er zijn twee methodes gedefinieerd, met name een die een nieuwe pointcut definieert (lijn 6) en dan nog een methode die een extra instelling doet (lijn 11).

De methode `restrictAccess` definieert dus een pointcut. Deze wordt geplaatst op de methodes `methods` van de objecten `objects` die van een van de types `classes` zijn. Wanneer een methode dan onderschept wordt, zal de `wrappingMethod` uit de wrapper van de aspect component uitgevoerd worden. Wat een wrapper is en doet, wordt in 4.2 besproken.

4.1.2 Configureren

De configuratie van aspect componenten gebeurt in **.acc*-bestanden. In deze bestanden worden onder andere de locaties waar pointcuts van aspect componenten moeten worden geplaatst, gedefinieerd.

Stel dat we een aspect component hebben zoals in 4.1.1.

Wanneer men nu een pointcut wil plaatsen op de methode `read` van de klasse `GeneralRecord`, dan kan men dat als volgt in het overeenkomstige configuratiebestand (`access.acc`) definiëren:

```
restrictAccess ALL application.record.GeneralRecord read.* controlAccess;
```

parameters

Het gevolg van deze lijn code is dat bij elke oproep van de methode `read` van alle objecten van de klasse `GeneralRecord`, de methode `controlAccess` van de wrapper `AccessWrapper` wordt geïnvocerd. Pas wanneer daar de methode `proceed` wordt uitgevoerd, zal worden verdergegaan met de uitvoering van de methode `read`. Daarna kan de wrapper eventueel nog extra zaken uitvoeren.

Behalve pointcuts kunnen ook andere zaken geconfigureerd worden in een **.acc*-bestand. Zo definiëren we in `access.acc` ook in welk bestand de connectoren, nodig voor het beleid, te vinden zijn. Dit gebeurt als volgt:

```
readConnectorsList "files/policy/view-connectors.pol"
```

parameter

4.2 Dynamische wrappers

Een *dynamische wrapper* is een gewoon alleenstaand object (met attributen die zijn toestand vormen en methodes die zijn functionaliteiten definiëren) dat verschillende methodes kan implementeren met speciale semantiek. Zo een wrapper kan worden gezien als een object dat gedragingen definieert om het gedrag van normale objecten uit te breiden (het basisobject wordt *gewrapped* door een wrapper). Dit kan worden gezien als een implementatie van het *decorator-patroon* [18].

Eerst wordt er getoond hoe dynamische wrappers geprogrammeerd kunnen worden (4.2.1), waarna wordt uitgelegd hoe wrappers kunnen worden samengesteld (4.2.2).

4.2.1 Programmeren

Dynamische wrappers kunnen naast de normale methodes nog drie andere soorten definiëren. Ze worden in tabel 4.1 kort uitgelegd.

Methode	Beschrijving
Wrapping methode	Kan handelingen uitvoeren voor en na methodes van normale objecten (zelfde als <i>around</i> in AspectJ)
Rol methode	Kan interfaces van normale objecten uitbreiden (gelijkaardig aan <i>introduce</i> in AspectJ)
Exception handler	Kan uitzonderingen behandelen die in het object waarop de wrapper wordt toegepast, voorkomen

Tabel 4.1: Extra methodes van wrappers [16]

De code in figuur 4.2 toont een wrapper die een methode van de eerste soort bevat (*controlAccess*).

```
public class AccessWrapper extends Wrapper {
    public AccessWrapper(AspectComponent ac) {
        super(ac);
        adf = new AccessDecisionFunction();
    }

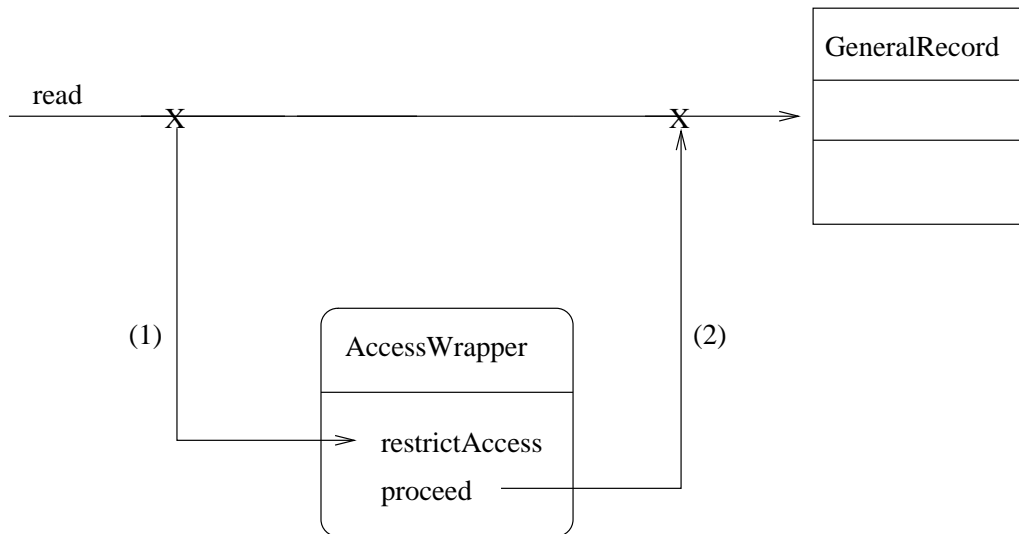
    public void controlAccess(Interaction i) {
        boolean accessGranted = ...;

        if (accessGranted)
            proceed(i);
        else
            System.out.println("Access denied");
    }
}
```

Figuur 4.2: Voorbeeld wrapper

Wanneer een object gewrapped wordt door een wrapper (dit object wordt dan *wrappee* genoemd) kunnen sommige methodes gewrapped worden door wrapping-methodes. Een inkomende call wordt onderschept door de wrapping-methodes die de huidige interactie ontvangen als enige parameter.

Concreet wordt een *GeneralRecord* gewrapped door een *AccessWrapper*. Dan onderschept *controlAccess* de methode *read* ((1) in figuur 4.3), die wordt voortgezet na de *proceed* in de wrapping-methode ((2) in figuur 4.3).



Figuur 4.3: Onderscheppen van een methode

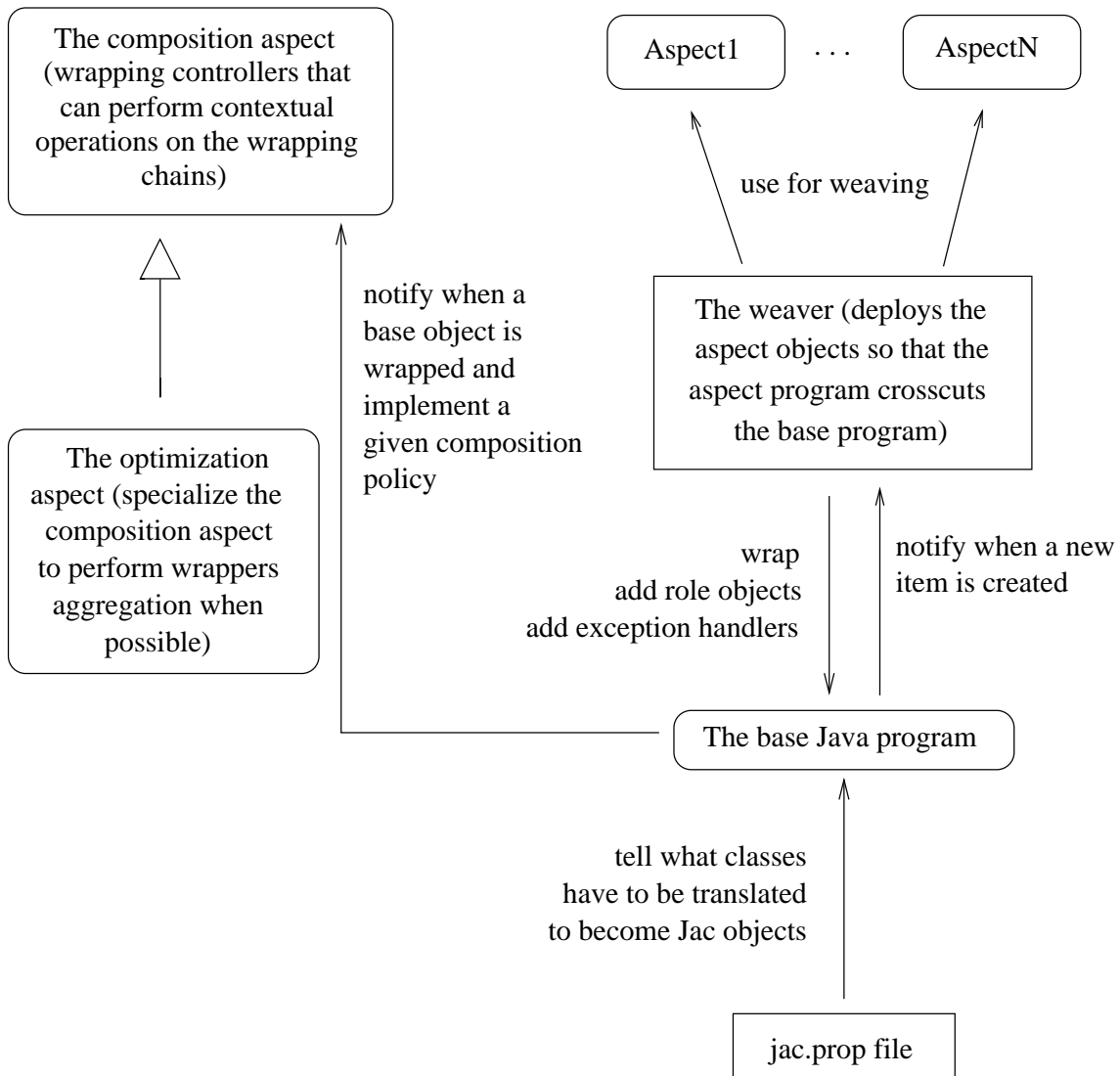
De *interactie*, een concept dat in JAC is ingebouwd, bevat de wrappee, de huidig opgeroepen methode en de parameters die werden meegegeven met de methode (hier zijn dat dus het *GeneralRecord* en *read*, die geen parameters heeft).

Een aspectgeoriënteerde toepassing programmeren komt neer op het kiezen van de juiste aspecten en ze zo configureren dat ze zich gedragen zoals het moet voor het programma. De volgorde van wrappen is belangrijk om het gewenste resultaat te bekomen. Het composition aspect moet dus zo worden geconfigureerd dat de volgorde van de wrappers het gewilde interactieschema implementeert (zie 4.2.2).

4.2.2 Samenstellen

Een wrapper kan worden gezien als de samenstellingseenheid bij het samenstellen van verschillende aspecten (wanneer verschillende wrapping-methodes dezelfde wrappee-methode wrappen). JAC voorziet een *composition aspect* (linksboven op figuur 4.4) dat geconfigureerd kan worden om samenstellingsregels te activeren. Deze regels specificeren onder meer in welke volgorde de wrappers op de basisobjecten moeten toegepast worden. Ze kunnen ook enkele afhankelijkheden of incompatibiliteiten tussen de wrappers definiëren. Door deze samenstellingsregels te gebruiken, kan het JAC-systeem voor elke methode met bijhorende verzameling wrappers een correcte wrap-volgorde definiëren. Deze ordening wordt de ‘*wrapping chain*’ genoemd. Wanneer de *proceed*-methode wordt opgeroepen, dan roept het systeem, als de wrapping chain niet eindigt, de volgende wrapping-methode op. Uiteindelijk zal de *proceed* in de laatste wrapping-methode de methode van de wrappee oproepen.

Het standaard aspect dat in JAC wordt gebruikt om aspecten samen te stellen is *org.objectweb.jac.core.CompositionAspect*. Dit aspect wordt gebruikt om te weten waar een wrapper in de wrapping-chain van een methode moet worden toegevoegd (zie methode *getWeaveTimeRank()* in het samenstellingsaspect). Het standaard compositie-aspect gebruikt hiervoor



Figuur 4.4: Wrappers samenstellen

jac.comp.wrappingOrder, die een strikte volgorde oplegt voor alle gekende wrappers. Wanneer je een nieuw aspect met nieuwe wrappers wil toevoegen, zou je dat in *jac.comp.wrappingOrder* moeten doen, zoniet wordt het standaard achteraan geplaatst.

Er is in JAC maar één samenstellingsaspect beschikbaar, maar men kan wel een ander schrijven dat dan bepaalde regels (zoals 'AspectA moet voor AspectB komen') bevat om zo de volgorde van wrappers te bepalen. Dit zelfgeschreven aspect moet wel overerven van het standaard aspect *org.objectweb.jac.core.CompositionAspect*.

Het *jac.prop*-bestand (onderaan figuur 4.4) definieert enkele algemene configuraties:

1. Het geeft aan welke klassen (met behulp van *jac.toNotAdapt* *jac.toAdapt*), en welke methodes (met behulp van *jac.wrappableMethods*) wrappable moeten gemaakt worden.

2. Het zegt aan de bytecode-vertaler om sommige collectievelden niet te vertalen (anders wordt *java.util.<collectie>* vertaald naar *org.objectweb.jac.lib.java.util.<collectie>*). Dit gebeurt met *jac.dontTranslateField*.
3. Het bepaalt welke klasse moet worden gebruikt om aspecten samen te stellen (*jac.comp.compositionAspect*).
4. Het bepaalt de volgorde van de wrappers met behulp van *jac.comp.wrappingOrder*.
5. Het registreert aspecten en geeft ze een korte naam.
6. Het bepaalt de topologie voor gedistribueerde configuraties met meerdere servers met *jac.topology*.

De volgorde van de wrappers, zoals ze door *jac.prop* bepaald is, wordt gelezen bij het opstarten van het systeem en wordt dan doorgegeven aan het samenstellingsaspect. De verdere werking van het raamwerk wordt in de volgende sectie uitgelegd.

4.3 Architectuur

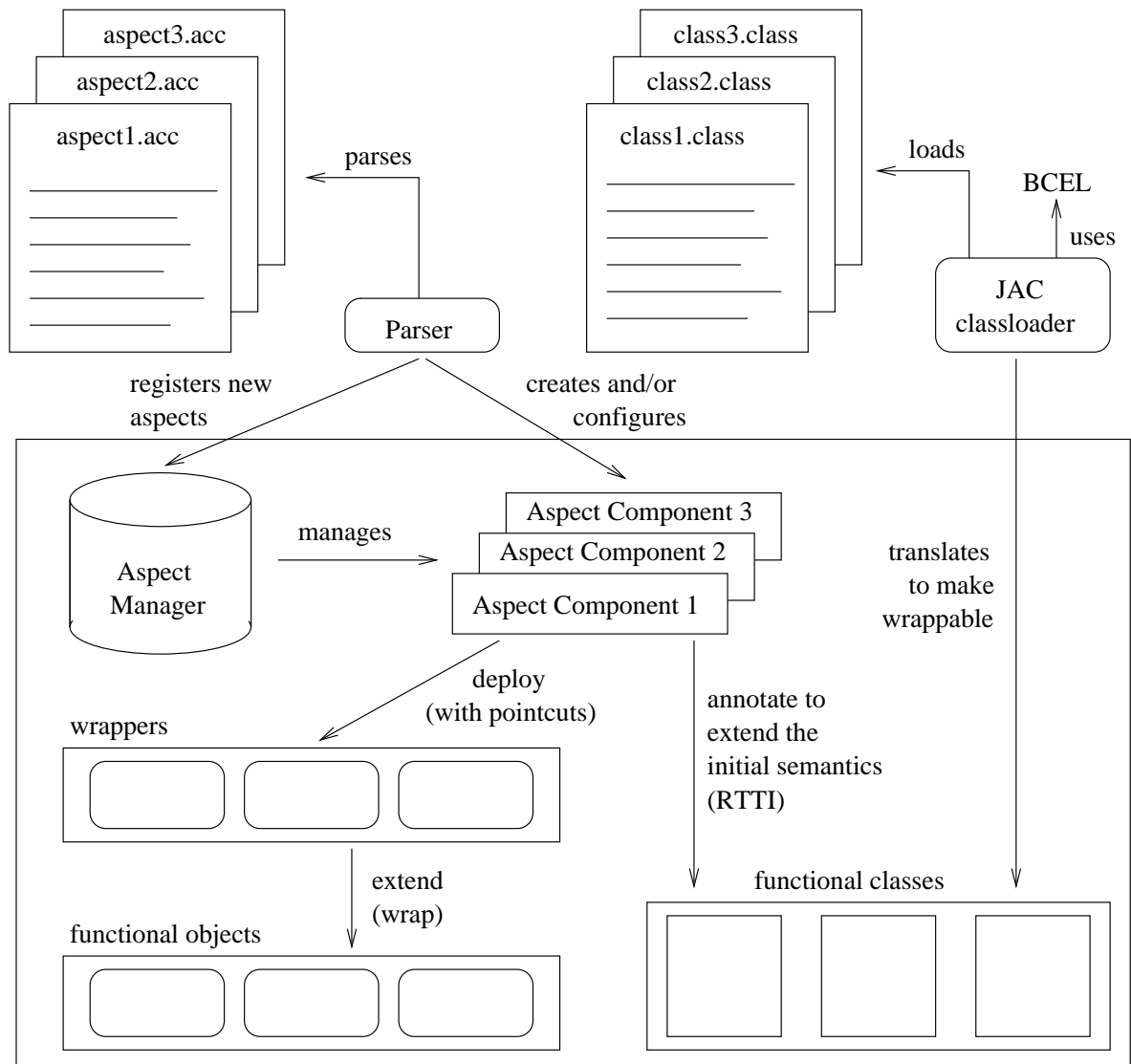
De architectuur van JAC wordt in *figuur 4.5* schematisch voorgesteld. De werking van JAC wordt eerst uitgelegd (4.3.1), daarna volgt een woordje over de performantie van dit systeem (4.3.2).

4.3.1 Werking

Rechts op *figuur 4.5* kan men zien dat de functionele klassen op bytecode-niveau gewijzigd worden om hun instanties wrappable te maken. Deze vertaling vindt plaats in de JAC classloader en gebeurt met behulp van BCEL (Byte Code Engineering Library). Tijdens de vertaling worden er drie verschillende handelingen uitgevoerd:

- De vertaler herbenoemt de originele methodes en creëert stub methodes die de methode *Wrapping.nextWrapper* oproepen om de wrapping chain toe te passen, als die er is.
- De bytecode van de methodes wordt geanalyseerd om in de RTTI nuttige informatie in te vullen: welke methodes getters, setters, lezers, accessoren, ... zijn.
- De collectietypes van *java.util.** worden vervangen door die van *jac.lib.java.util.** zodat ook collecties wrappable worden. De klassen van *java.** zijn namelijk niet wrappable omdat *Sun* er om veiligheidsredenen een hardgecodeerde test heeft ingebouwd.

Eenmaal de klassen vertaald zijn, zijn ze klaar om gewrapped te worden door de aspect componenten. Wanneer een aspect in een gegeven applicatie wordt geweven, leest het JAC-systeem eerst de beschikbare configuratie van de aspect component (de **.acc*-bestanden, linksboven in *figuur 4.5*). De parser roept dan op zijn beurt een verzameling configuratiemethodes op op de nieuw geïnstantieerde aspect component. Deze oproepen veroorzaken de creatie van pointcuts en het toevoegen van meta-data aan de functionele klassen (met behulp van de RTTI).



Figuur 4.5: De JAC architectuur [16]

Wanneer een nieuwe instantie gecreëerd is, verwittigt de *Aspect-Component Manager* (op de figuur Aspect Manager genoemd) automatisch alle geregistreerde aspecten zodat de pointcuts hun methodes wrappen volgens de overeenkomstige aspectconfiguratie. Belangrijk hierbij is dat elke aspect component op elk moment kan in- of uitgeweven worden. Meer nog, het configuratiebestand van een aspect component kan opnieuw geparsed worden tijdens de uitvoering van de applicatie. Wanneer een aspect wordt uitgeweven of een nieuwe configuratie wordt ingelezen, worden alle toegevoegde meta-data en pointcuts die geassocieerd zijn met dat aspect, door het systeem vernietigd. Dit is mogelijk dankzij de implementatie van het wrapping mechanisme voor dynamisch wrappen/unwrappen.

Dynamische wrappers steunen op een Meta-Object Protocol (MOP) dat voor zijn implementatie reflectie gebruikt. De belangrijkste vertaling is het herbenoemen van de originele in een verborgen methode zodat de originele methode niet rechtstreeks wordt opgeroepen. De originele methode wordt vervangen door een stub die de originele methode wrapt door gebruik te maken van *Wrapping.nextWrapper(Interaction)*. De initiële interactie wordt gecreëerd in de stub methode. Die heeft de wrapping chain die intern wordt gebruikt, de referentie van de wrappee, de huidige methode en haar parameters, en een initiële rang in de wrapping chain nodig. Wrappen of unwrappen van een gegeven object at runtime is eenvoudig gezien het neerkomt op het toevoegen of verwijderen van een element in de wrapping chain van de methode.

4.3.2 Performantie

In JAC kunnen we twee performantie-issues onderscheiden:

Loadtime overhead: Is te wijten aan de bytecode-analyse en de vertaling bij het laden van de functionele klassen. Het laden van een klasse in JAC is ongeveer 8 keer trager dan voor een gewone klasse.

Runtime overhead: Wordt veroorzaakt door de stub methodes en de evaluatie van de volgende wrappers in de wrapping chain (vooral geïmplementeerd in de *Wrapping.nextWrapper* methode).

4.4 Conclusie

In dit hoofdstuk werd *JAC (Java Aspect Components)* geïntroduceerd, een raamwerk voor aspectgeoriënteerd programmeren in Java. Het verschil met andere aspectgeoriënteerde systemen zoals AspectJ, is het dynamische karakter. In JAC kan men aspecten namelijk at runtime in- en uitweven, en zelfs herconfigureren. Het feit dat JAC aspectgeoriënteerd is, maakt het raamwerk geschikt om een hoge graad van scheiding van belangen te verkrijgen. Ook het dynamische karakter is een groot voordeel, omdat de beveiliging (toegangscontrole, authenticatie, ...) op die manier eenvoudig kan worden aangepast wanneer de situatie dit vereist. Bovenstaande voordelen zijn de belangrijkste redenen waarom we JAC gaan gebruiken om onze toepassing te beveiligen.

Hoofdstuk 5

Algemeen ontwerp beveiliging

In hoofdstuk 2 hebben we gezien dat AOP een mogelijke oplossing biedt om niet-functionele belangen af te scheiden van de rest van het programma. In dit hoofdstuk doen we een poging om via AOP de beveiliging in zijn geheel af te zonderen van de basisapplicatie.

In de eerste plaats gaan we in 5.1 na wat de vereisten zijn voor een medische toepassing en trachten we uit de vereisten de mogelijke aspecten halen. We zullen zien dat het onmogelijk is om alle aspecten afzonderlijk van elkaar te ontwerpen, daarom is er nood aan communicatie tussen hen. In 5.2 stellen we enkele vormen van communicatie voor in AOP. Tenslotte gaan we in 5.3 een ontwerp van onze applicatie en van de verschillende aspecten voorstellen.

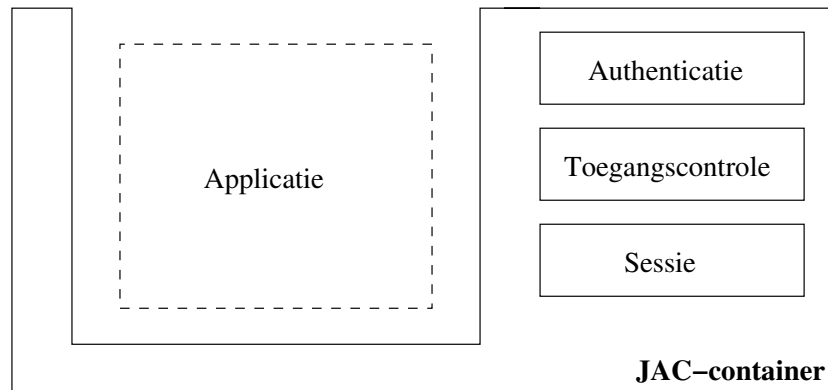
5.1 Kandidaataspecten

Het is nu het moment om eens terug te kijken naar hoofdstuk 1. We zagen er wat de wetgeving oplegt en hoe we die wetgeving in de praktijk konden brengen. Op het einde stelden we dat we voor de beveiliging best een flexibele toegangscontrole gebruiken.

Een flexibele toegangscontrole gebruiken is een vaag begrip. Toegangscontrole is ook niet iets dat volledig apart staat. Het is evident dat als iemand het systeem gebruikt, zich eerst bekend maakt aan het systeem. Het zou ook niet mogen dat telkens als een persoon een actie wil uitvoeren, hij zich moet identificeren. Het is duidelijk dat we hieruit al verschillende aspecten kunnen halen, maar dit betekent niet dat die aspecten effectief als aspect zullen geïmplementeerd worden (zie later 5.3).

Op het eerste zicht kunnen we 3 aspecten afzonderen: één voor authenticatie, één voor het beheren van de sessie en één voor de toegangscontrole. We gaan nu meer in detail ieder kandidaataspect bekijken.

JAC is eigenlijk een application-container die zich rond een applicatie bevindt. Dit wordt in figuur 5.1 schematisch voorgesteld. Een groot voordeel van JAC, in tegenstelling tot bijvoorbeeld J2EE, is dat de aspecten uit de container door de gebruiker kunnen uitgebreid worden. We kunnen zelfs enkele aspecten toevoegen aan die container, zodat de functionaliteit ervan aan onze noden voldoet. De JAC-container is bijgevolg veel flexibeler dan die van J2EE, waar enkel aan de configuratie kan gesleuteld worden.



Figuur 5.1: JAC container

5.1.1 Authenticatie

Niet iedereen mag zomaar toegang hebben tot een systeem, en vooral: het systeem moet ook kunnen bepalen wie er juist probeert in te loggen. Een eerste mogelijk aspect zou voor de authenticatie verantwoordelijk kunnen zijn.

Authenticatie komt er op neer te achterhalen wat de herkomst is van een toegangsaanvraag tot het systeem. Wanneer de authenticatie bestaat uit een loginmechanisme en een sessie, komt het erop neer de herkomst van de actieve sessie - de informatie die bij de login is opgevraagd - te controleren.

Dit is eigenlijk ook een vorm van toegangscontrole. Men kijkt namelijk of een persoon het systeem mag gebruiken. Wij echter zien toegangscontrole eerder als het beslissen of een reeds aangemeld persoon een operatie mag uitvoeren of niet.

5.1.2 Sessie

Het is absurd dat bij iedere actie die een persoon wil uitvoeren, er een login moet gebeuren. Het zou handig zijn om een soort sessie te starten die dan voor een bepaalde tijd geldig is.

Het sessieaspect zou dus kunnen bijhouden wie er precies is ingelogd. Dit brengt met zich mee dat men de sessie niet volledig kan afzonderen van de authenticatie, maar daar komen we verder op terug in 5.1.4. Een sessie kan gestart worden als iemand is ingelogd en afgesloten worden als diezelfde persoon uitlogt. Men zou ook kunnen zeggen dat een sessie maar een bepaalde geldigheid bezit, bijvoorbeeld dat men na een half uur automatisch wordt afgemeld door de sessie af te breken. Dit vermijdt dat door een onvoorzichtig persoon die zijn machine onbewaakt achterlaat, een indringer ongeoorloofde toegang zou hebben.

5.1.3 Toegangscontrole

In punt 5.1.1 legden we kort uit wat wij onder toegangscontrole verstaan. Toegangscontrole is hier het mechanisme dat aan de hand van het beveiligingsbeleid uitmaakt of een ingelogd persoon een operatie mag uitvoeren. Bij medische gegevens bestaan de meeste operaties uit het lezen en wijzigen van records. Hier beperken we ons op methoden op objecten.

Dit aspect zal de meeste complexiteit bevatten. Hier wordt immers het beveiligingsbeleid toegepast. Er moet soms bijkomende informatie verzameld worden om de beslissingen te kunnen uitvoeren. In de eerste plaats moet de toegangscontrole weten wie er juist de operatie uitvoert, met andere woorden wie er ingelogd is. Dit betekent dat er weer communicatie nodig is met de authenticatie en/of de sessie. Vaak moeten er ook bijkomende gegevens uit het systeem gehaald worden, zoals *"Is het tijdens een spoedgeval?"* of *"Is de persoon die het record wil lezen een behandelende zorgverstrekker?"*.

We mogen ook niet vergeten dat de toegangscontrole (lieft dynamisch) aanpasbaar moet zijn, wat nog voor een extra moeilijkheid zorgt.

5.1.4 De interactie tussen kandidaataspecten

De kandidaataspecten kunnen niet volledig onafhankelijk functioneren. Het authenticatieaspect kan op zichzelf werken, maar de twee andere (sessie en toegangscontrole) kunnen niet rechtstreeks of onrechtstreeks zonder het authenticatieaspect. Dit komt omdat ze beiden moeten weten wie er juist ingelogd is.

Men zou kunnen stellen dat de authenticatie eigenlijk een deel is van de sessie, bij het starten van een sessie gebeurt er eerst een authenticatie en voor de rest werkt de sessie alleen verder omdat het weet welke persoon is ingelogd. We kunnen het ook omgekeerd zien, namelijk dat de sessie eigenlijk een deel is van de authenticatie. Bij het inloggen wordt een sessie gestart, zodat de controle van identiteit van de persoon bij latere operaties niet meer gecontroleerd wordt. Hoedanook hebben we hier met een eerste afhankelijkheid te maken.

Zoals al eerder aangehaald is de toegangscontrole waardeloos als men niet weet wie de oproeper is van de methode (het *subject*). Het subject is normaal de persoon die ingelogd is en bijgehouden wordt in de sessie. Op een of andere manier moet die persoon doorgegeven worden aan het toegangscontrole-aspect.

Voor we kijken naar oplossingen voor deze twee afhankelijkheden kijken we eerst in het volgende hoofdstuk hoe men tussen twee aspecten kan communiceren.

5.2 Communicatie tussen aspecten

Het is de bedoeling dat een aspect los staat van de functionaliteiten van het basisprogramma, maar het kan natuurlijk wel voorkomen dat een aspect informatie nodig heeft van een ander aspect. In onze toepassing bijvoorbeeld heeft het Access-aspect nood aan informatie over de herkomst van de degene die toegang wil. Die persoon is beschikbaar via het Authenticatie-aspect, vandaar dat communicatie tussen deze twee aspecten noodzakelijk is.

Hier botsen we op een moeilijkheid, want in JAC (en meer algemeen in AOP) bestaan geen richtlijnen die stellen hoe communicatie tussen aspecten moet verlopen. Om die reden zijn we zelf op zoek gegaan naar mogelijke manieren om dit op te lossen.

5.2.1 Mogelijke ontwerpen

We willen dus communicatie die gaat van het Authenticatie-aspect naar het Access-aspect. Om dit te realiseren zijn er verschillende opties, elk met hun voor- en nadelen. In deze sectie zullen we voor elke optie de voor- en nadelen afwegen, en gaan we ook eens kijken of de realisatie ervan in JAC wel mogelijk is. De opties die we geïdentificeerd hebben zijn:

- Gebruik maken van een samengesteld aspect dat dan de twee aspecten (Access en Authenticatie) omvat.
- Het ene aspect in het andere verwerken. Concreet voor ons betekent dat dat het Access-aspect het Authenticatie-aspect omvat.
- Het in JAC ingebouwde concept van *Interaction* uitbreiden, zodat daar de nodige informatie in verwerkt kan worden.
- Het Authenticatie-aspect wrappen rond het Access-aspect.
- De nodige gegevens in een thread bijhouden die voor beide aspecten toegankelijk is.

Al deze opties worden hierna in meer detail besproken.

5.2.1.1 Samengesteld aspect

Ten eerste kunnen we een samengesteld aspect gebruiken. Een aspect bevat dan meerdere deelaspecten. In JAC bestaat er een speciale aspect component die dit mogelijk maakt. Deze component wordt *CompositeAspectComponent* genoemd, en kan verschillende kind-aspect componenten bevatten. Het is nodig om de methodes die op het samengesteld aspect worden opgeroepen te delegeren naar de overeenkomstige methodes van zijn kinderen. Het is best mogelijk dat de samengestelde aspect component nogal omslachtig wordt, vooral als die veel sub-aspect componenten bevat. In dat geval zullen er namelijk veel methoden zijn die moeten gedelegeerd worden.

Een korte illustratie van hoe het kan, is hier misschien wel op zijn plaats. We hebben dus twee aspecten *AccessAspect* en *AuthenticationAspect*, die dan worden samengebracht in *CompositeAspect*. In deze laatste kan dan informatie worden bijgehouden die door alle kinderen kan worden gebruikt (*'information'* in *CompositeAspect*).

```
public class CompositeAspect extends AspectComponent {
    public CompositeAspect() {
        access = new AccessAC();
        authentication = new AuthenticateAC();
    }

    public void restrictAccess(...) {
        access.restrictAccess(...);
    }

    public void authenticate(...) {
```

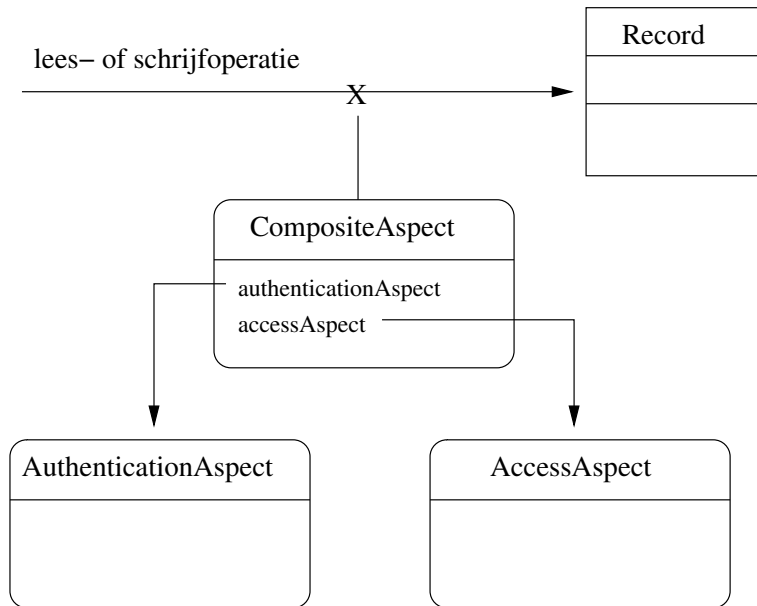
```

    authentication.authenticate(...);
}

private AccessAC    access;
private AuthenticateAC authentication;

private Object information;
}

```



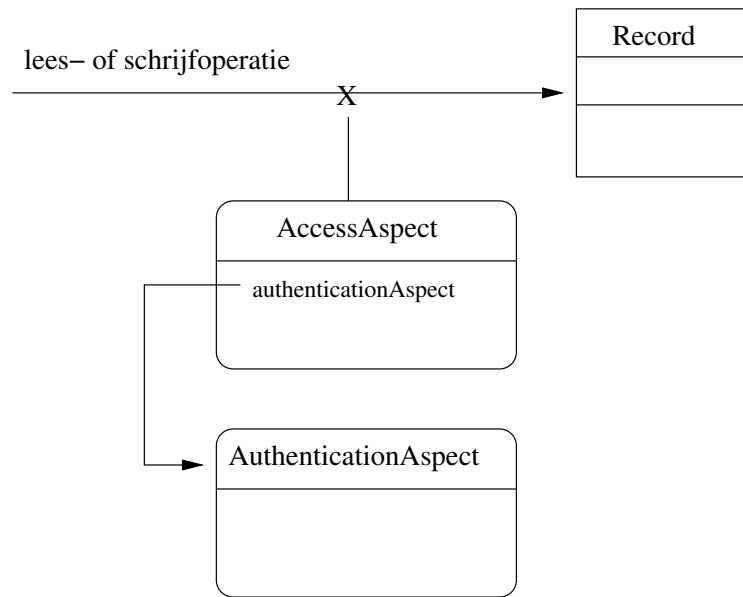
Figuur 5.2: Samengesteld aspect

Het grote voordeel van deze methode is dat de controle over de aspecten gecentraliseerd verloopt. Men moet zich maar op één plaats concentreren op het gedrag van de aspecten. Nadelig aan deze oplossing is dat de twee kind-aspecten expliciet van elkaar afhankelijk zijn. In deze opstelling zou men kunnen denken dat de twee aspecten van elkaar afhangen - Authenticatie beïnvloedt Access en vice versa - terwijl dit in onze toepassing maar in één richting zou gelden: Access is wel afhankelijk van Authenticatie, maar dit geldt niet in de andere richting. Problematisch hier is ook dat geen van de kinderen nog alleen kan gebruikt worden, ze komen altijd samen voor. De implementatie ervan kan wel herbruikt worden in andere componenten, maar de kinderen op hun geheel zijn gebonden aan het samengesteld aspect.

5.2.1.2 Hoofd- en deelaspect

Een andere mogelijke oplossing is dat een (hoofd-)aspect een ander (deel-)aspect bevat. In dit geval kan het hoofdaspect de methoden van het deelaspect gebruiken, terwijl dit in de omgekeerde richting niet mogelijk is.

Dit ontwerp is in feite een speciaal geval van een samengesteld aspect, meerbepaald het geval waarin één aspect afhankelijk is van een ander. Hier kan - in tegenstelling tot de oplossing hierboven - geen verwarring ontstaan over het al dan niet wederzijds zijn van de



Figuur 5.3: Hoofd- en deelaspect

afhankelijkheid. Het is namelijk overduidelijk dat die enkel in één richting geldig is: het hoofdaspect hangt af van het deelaspect, maar niet andersom. Het deelaspect kan dus ook op zichzelf gebruikt worden, zonder tussenkomst van het andere.

Voor ons betekent dit ontwerp dat het Access-aspect het Authenticatie-aspect zou omvatten (zie figuur 5.3). En dit om de eenvoudige reden dat het Access-aspect informatie van het Authenticatie-aspect nodig heeft, meerbepaald de persoon die toegang wil krijgen tot een record.

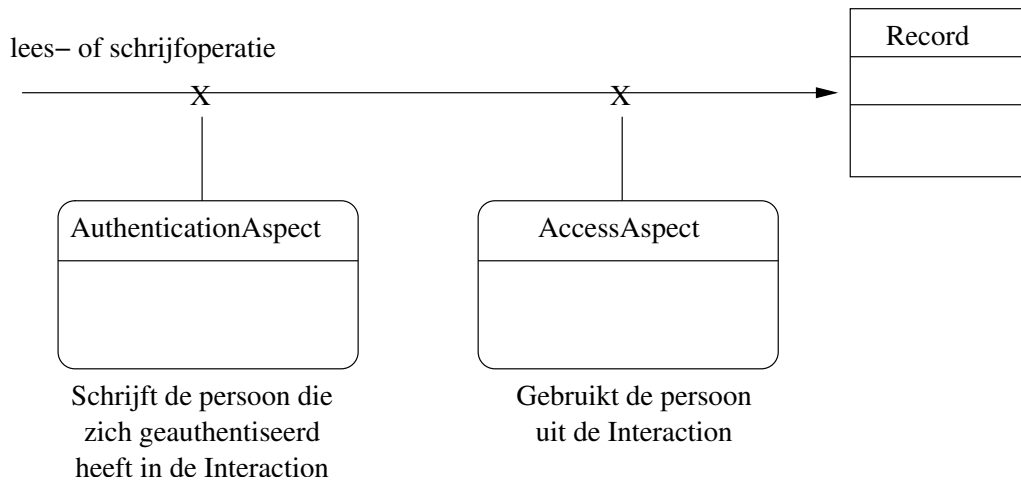
Een voordeel ten opzichte van de vorige oplossing is dat hier expliciet wordt weergegeven dat de afhankelijkheid maar in één richting geldt. Dit betekent dat het Authenticatie-aspect ook zelfstandig kan werken, zonder tussenkomst van het Access-aspect.

Het Access-aspect daarentegen kan dit niet, omdat het afhankelijk is van de authenticatie. Op dit gebied kan er misschien een nog betere oplossing worden gevonden die ervoor zorgt dat de twee aspecten volledig onafhankelijk van elkaar *kunnen* werken.

5.2.1.3 Uitbreiding van de interaction

Vervolgens kunnen we ook informatie toevoegen aan de Interaction. De interactie bevat al het object en de methode die gewrapped worden, samen met de argumenten van die methode. Het concept van Interaction zou nu kunnen worden uitgebreid. Het ene aspect schrijft er extra informatie in, die dan door het andere aspect kan gelezen worden (zie figuur 5.4).

In ons concrete programma zou het erop neerkomen dat we het Authenticatie-aspect de persoon die toegang tot een record wil krijgen, aan de interactie laten toevoegen wanneer zijn authenticatie geslaagd is. Op die manier kan het Access-aspect die informatie gebruiken. Dit is nodig omdat de toegangscontrole concreet per persoon moet gebeuren.



Figuur 5.4: Uitbreiding interaction

De volgorde van de wrappers is hier van groot belang, omdat de authenticatie zeker voor de toegangscontrole moet komen. Zoniet zal het Access-aspect onvoldoende informatie hebben om een beslissing te kunnen nemen.

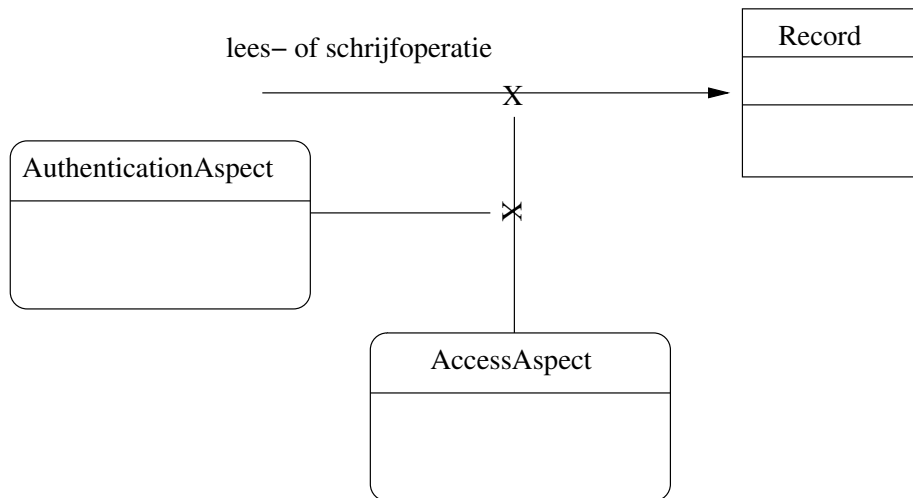
Het voordeel van deze methode is dat de communicatie tussen de aspecten gescheiden gebeurt. Het ene aspect geeft geen informatie rechtstreeks aan het andere door. Eenmaal de informatie in de Interaction is bijgevoegd, moet het Authenticatie-aspect zich er niets meer van aantrekken, zijn taak zit er op. Wanneer het Access-aspect dan de informatie ophaalt, wordt die uit de interactie gehaald, ze moet dus niet uit het Authenticatie-aspect worden gehaald.

Gevaarlijk aan deze oplossing is wel het feit dat de afhankelijkheid tussen de aspecten *impliciet* is. De toegangscontrole verwacht namelijk dat de authenticatie de nodige informatie in de interactie geschreven heeft. Wanneer dit niet gebeurt is, zal er van alles foutlopen bij de toegangscontrole. Om deze reden is het belangrijk dat dit goed gedocumenteerd wordt, zodat andere programmeurs, die de code willen hergebruiken, weten dat er wel degelijk een afhankelijkheid tussen de twee bestaat.

Deze optie mag worden afgeschreven, omdat dit niet blijkt te werken in JAC. Wanneer een uitgebreide interactie wordt gebruikt, wordt die niet meer gezien als de oorspronkelijke interactie. Dit zorgt er voor dat de *WrappingChain* (die weergeeft in welke volgorde de wrappers moeten worden opgeroepen) wordt onderbroken, zodat na de Authenticatie de toegangscontrole niet gestart wordt.

5.2.1.4 Een aspect rond een ander aspect wrappen

Een andere optie is dan weer dat we een aspect rond een ander laten wrappen. Het ene aspect onderschept methodeoproepen van het basisprogramma. Dit aspect roept dan de nodige methodes op in zijn wrapper. Hier komt dan het tweede aspect tussen, en onderschept een van de methodeoproepen in deze wrapper.



Figuur 5.5: Wrappen rond aspect

Bij ons zou het er op neerkomen dat het Authenticatie-aspect gewrapped wordt rond het Access-aspect (zie figuur 5.5). Wanneer de toegangscntrole dan wordt gestart, zal de gebruiker eerst zijn identiteit moeten kenbaar maken. Het Authenticatie-aspect kan dan die identiteit doorgeven aan het Access-aspect (eventueel via een uitgebreide interactie).

Deze oplossing lijkt een volledige onafhankelijkheid te verwezenlijken, maar toch is dit niet het geval. De toegangscntrole verwacht namelijk dat de authenticatie de identiteit van de gebruiker doorgegeven heeft. We hebben hier dus hetzelfde probleem als in het vorige geval, namelijk dat we hier te maken hebben met een *impliciete* afhankelijkheid.

Een voordeel hier is wel dat deze oplossing leidt tot een ‘proper’ ontwerp. Dit dubbele wrappen is in een conceptueel model eenvoudig voor te stellen.

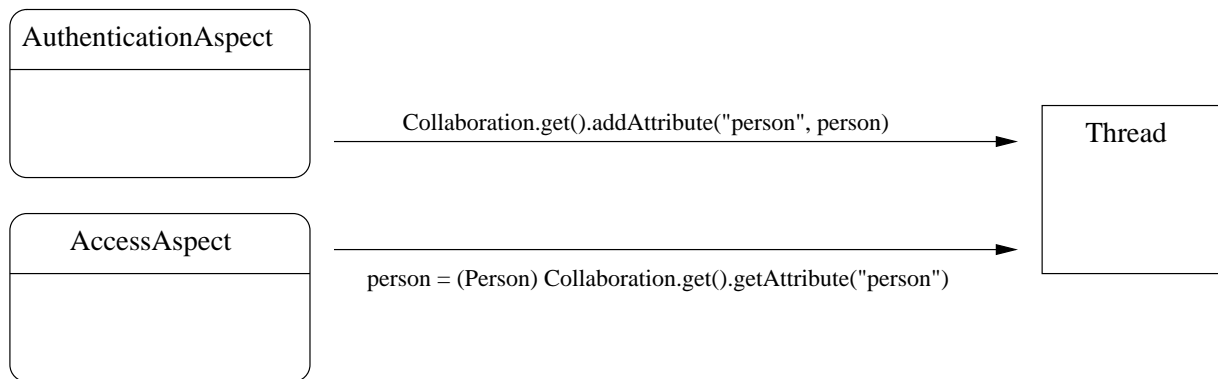
Dit zou een kanshebber kunnen zijn voor onze toepassing, ware het niet dat dit niet werkt in JAC. De reden hiervoor is dat JAC de klassen van het basisprogramma vertaalt op bytecode-niveau om ze wrappable te maken. Gezien wrappers niet tot deze klassen worden gerekend, worden ze ook niet vertaald, waardoor ze niet kunnen gewrapped worden. Een mogelijk alternatief is dat de wrapper van het Access-aspect een methode in een andere klasse oproept, en dat het Authenticatie-aspect zich dan rond deze methode wrapt. Dit hebben we onmiddellijk afgeschreven als optie, omdat zo de onafhankelijkheid van de beveiliging ten opzichte van het basisprogramma in het gedrang komt.

5.2.1.5 Attributen in thread

Tenslotte kunnen we ook in een thread, die voor de aspecten in kwestie toegankelijk is, attributen per object gaan bijhouden (zie figuur 5.6). Dit kan met behulp van twee methoden die in JAC vervat zitten.

Voor het schrijven van het attribuut in de thread kan volgende code worden gebruikt:

```
Collaboration.get().addAttribute("myAttributeName",myValueObject);
```



Figuur 5.6: Attribuut aan thread toevoegen

Het lezen van die informatie kan dan als volgt geïmplementeerd worden:

```
info = Collaboration.get().getAttribute("myAttributeName");
```

Concreet kunnen we in het Authenticatie-aspect volgende code invoegen:

```
Collaboration.get().addAttribute("person", person);
```

In het Access-aspect kunnen we dan als volgt die informatie ophalen:

```
person = (Person) Collaboration.get().getAttribute("person");
```

Nadeel hierbij is dat de afhankelijkheid tussen de aspecten opnieuw *impliciet* is. Ook hier is een degelijke documentatie dus vereist. De naamgeving van de attributen kan ook voor problemen zorgen. Bijvoorbeeld als er meerdere gebruikers tegelijk ingelogd zijn, dan zullen er verschillende attributen worden toegevoegd aan de thread. Er moet dus een consistente naamgeving gebruikt worden, zodat duidelijk is welk attribuut bij welke sessie behoort. Het grote voordeel is dat het heel eenvoudig is om de communicatie tussen de aspecten te verwezenlijken. Er moeten maar twee lijnen code worden toegevoegd aan de bestaande aspecten.

5.2.2 Conclusie

Voor een overzicht van de voor- en nadelen van de besproken opties: zie tabel 5.1

Gezien in ons geval enkel het Access-aspect afhankelijk is van het Authenticatie-aspect, kunnen we het CompositeAspect als oplossing al verwerpen. Twee andere oplossingen - uitbreiding van het concept van Interaction en het wrappen rond een aspect - kunnen ook worden afgeschreven, omdat ze niet te implementeren zijn in JAC. Er blijven dus nog twee mogelijkheden over: een aspect dat een ander bevat en het gebruiken van een thread om daar de

Keuze	Voordelen	Nadelen
Samengesteld aspect	Gecentraliseerde controle	Expliciete, wederzijdse afhankelijkheid
Hoofd- en deelaspect	Enkelvoudige afhankelijkheid, deelaspect kan autonoom werken	Hoofdaspect kan niet zonder deelaspect
Uitbreiding interaction	Communicatie afgescheiden	Impliciete afhankelijkheden, controle, implementatie uitbreiding, werkt niet in JAC
Wrappen rond aspect	Makkelijk gebruik	Impliciete afhankelijkheid, werkt niet in JAC
Variabelen per thread	Eenvoudig	Keuze namen van variabelen, impliciete afhankelijkheid

Tabel 5.1: Voor- en nadelen van de verschillende ontwerpen

nodige informatie in weg te schrijven.

Wanneer we een hoofd- en deelaspect gebruiken, betekent dit dat het deelaspect niet meer als een echt aspect kan gezien worden. Het heeft nog wel de functionaliteiten van een aspect maar onderschept zelf geen methodes meer, alles gebeurt via het hoofdaspect. De wrapper van het deelaspect wordt niet meer gebruikt om rond een methode te wrappen, maar gewoon om functionaliteiten toe te voegen. Het gevolg hiervan is dat we een deel van onze dynamiek verliezen. Enkel het hoofdaspect kan nog dynamisch in- of uitgeweven worden, omdat enkel voor dat aspect pointcuts gedefinieerd zijn.

Wanneer we nu een *thread* gebruiken, behouden we de dynamiek van beide aspecten. Wanneer het ene aspect onafhankelijk werkt van het ander wordt er wel nutteloze informatie weggeschreven door dat aspect. In ons geval zou dit voorkomen wanneer we bijvoorbeeld enkel authenticatie doen, zonder toegangscontrole.

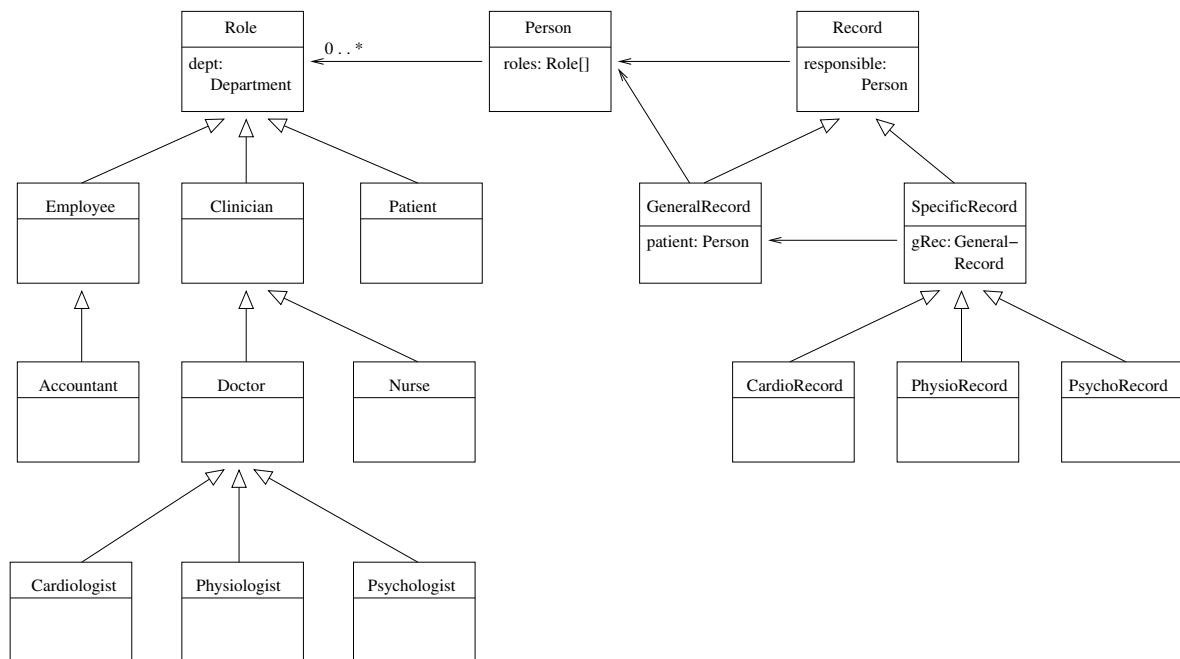
We hebben beslist om te werken met een thread, ondanks het feit dat dit een risico inhoudt: er is een impliciete afhankelijkheid tussen beide aspecten. Om duidelijk te maken dat men met deze afhankelijkheid moet rekening houden, is het noodzakelijk om dit in de documentatie uitgebreid ter sprake te brengen.

5.3 Design

In deze sectie bespreken we het ontwerp van zowel de applicatie als de beveiliging. Dit gebeurt respectievelijk in 5.3.1 en in 5.3.2.

5.3.1 Applicatiemodel

In figuur 5.7 kan men het model vinden van de applicatie die wij gebruiken om onze beveiliging op toe te passen.



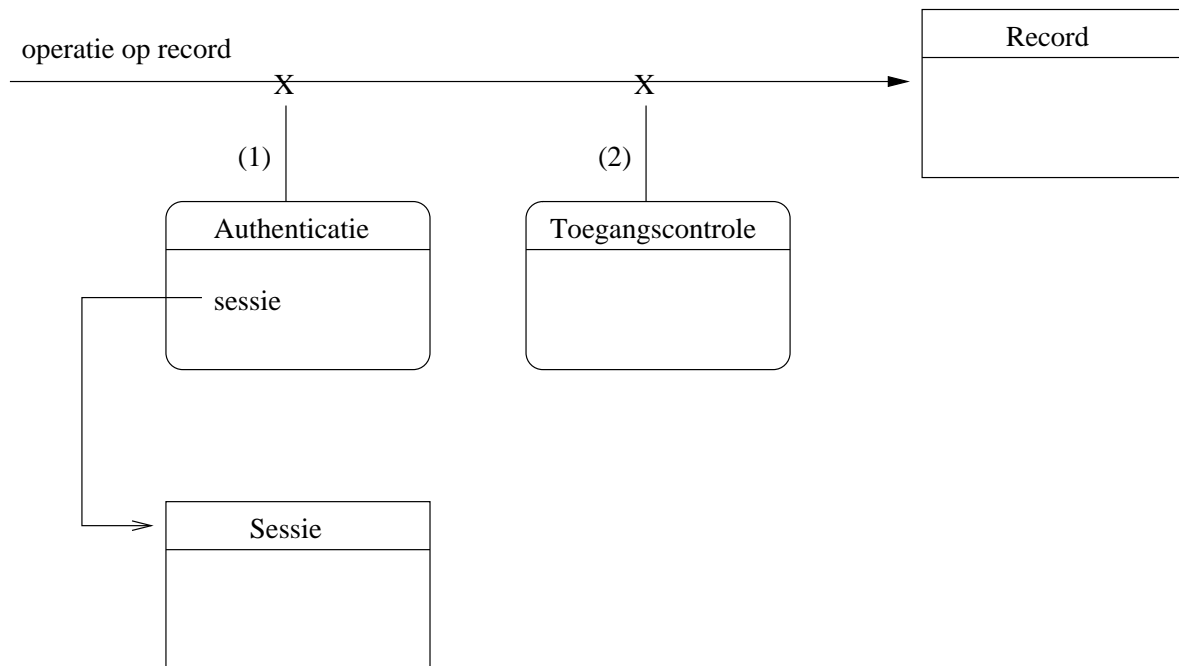
Figuur 5.7: Model van de applicatie

Een gebruiker van het programma (*Person* in de figuur) kan tijdens de uitvoering verschillende rollen spelen. Elke rol wordt geassocieerd met één afdeling. Zo kan een gebruiker bijvoorbeeld tegelijk *cardioloog* zijn op de afdeling *Cardiologie* en *verpleger* op de afdeling *Fysiologie*.

Van elke patiënt worden er verschillende records bijgehouden: een *GeneralRecord* met algemene informatie (zoals naam en adres van de patiënt) en een of meer records met specifieke informatie (zoals bv. cardiologische informatie in een *CardioRecord*).

5.3.2 Beveiligingsmodel

Een model op hoog niveau van de beveiliging die we ontwikkeld hebben, kan worden gevonden in figuur 5.8. Daar kan men zien dat de beveiliging bestaat uit een authenticatie- en een toegangscontrole-gedeelte.



Figuur 5.8: Model van de beveiliging

Voor de authenticatie ((1) in de figuur) wordt er extra informatie bijgehouden nadat een gebruiker zich succesvol geauthentiseerd heeft. Er wordt dan namelijk een *sessie* gecreëerd zodat de gebruiker, zolang hij niet uitgelogd is uit het systeem, niet opnieuw moet inloggen wanneer hij nog eens toegang wil tot een record. Op deze manier wordt het beheren van een sessie niet aanzien als een aspect. De functionaliteit is wel afgescheiden van die van de authenticatie. Het is eigenlijk onnodig om een sessie als aspect te modelleren, gezien ze nauw verbonden is met het authenticatie-aspect en bovendien op dezelfde plaatsen gebruikt wordt. Vandaar dat we in het authenticatie-aspect een sessie vervat hebben.

Voor de toegangscontrole ((2) in de figuur) is er informatie nodig over de gebruiker die toegang wil verkrijgen, vandaar dat in onze beveiliging de authenticatie eerst komt. Op die manier is de nodige informatie beschikbaar wanneer de toegangscontrole start.

5.4 Besluit

In dit hoofdstuk hebben we het ontwerp van zowel de applicatie als de beveiliging die we gaan gebruiken, besproken. De beveiliging zal uit een authenticatie- en een toegangscontrole-gedeelte bestaan. Om die twee te laten samenwerken zijn we op zoek geweest naar mogelijke manieren om communicatie tussen aspecten te verwezenlijken. Bij de authenticatie wordt ook een sessie gebruikt, maar die wordt niet als aspect gemodelleerd. Omdat het toegangscontrolemechanisme dat we ontwikkeld hebben nogal complex is, hebben we daar in deze tekst een volledig hoofdstuk aan gewijd, meerbepaald hoofdstuk 6.

Hoofdstuk 6

Access-control aspect

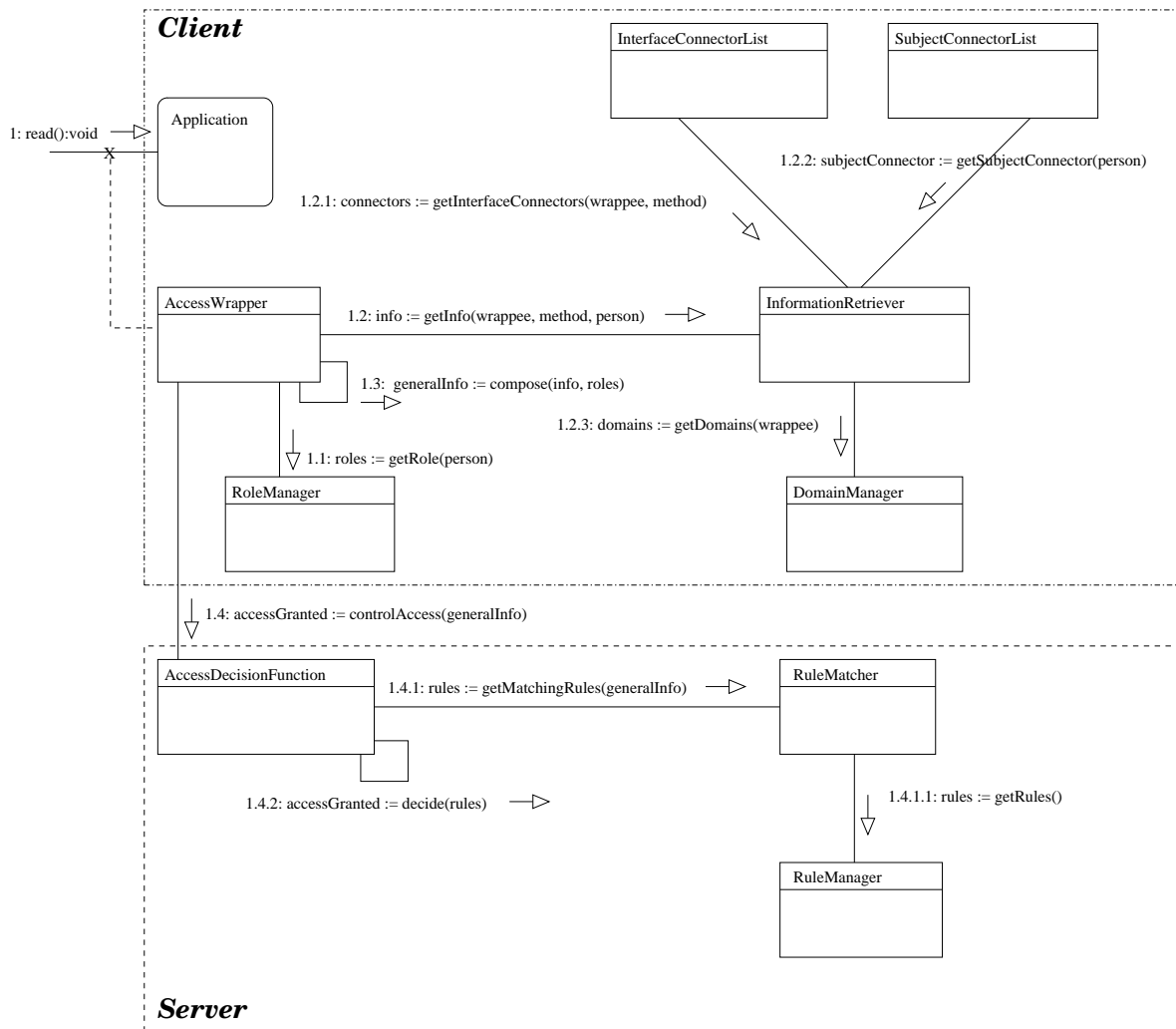
In hoofdstuk 5 zijn we snel over het access-control aspect gegaan. Dit aspect voert het meest gecompliceerde van de beveiliging uit, namelijk de toegangscontrole. We zijn eerder in hoofdstuk 3 tot een taal gekomen die gebruik maakte van het Uitgebreid View-connectormodel. Dit model maakte gebruik van een soort connector zodat op deze manier de taal werd gebonden aan de applicatie zonder dat ze er afhankelijk van werd. Hier trachten we volgens dit laatste model een aspect dat toegangscontrole regelt, te ontwerpen.

We beginnen in 6.1 met het geven van een algemene structuur van hoe een access-control aspect eruit kan zien. Omdat de interactie tussen de onderlinge delen van de toegangscontrole nogal complex verloopt, introduceren we in 6.2 nieuwe structuren die de transparantie van de werking van het aspect verhogen. In 6.3 bespreken we de implementatie van het access-control aspect in JAC volgens het algemeen ontwerp uit 6.1. Tenslotte stellen we in 6.4 een beleid op voor onze toepassing.

6.1 Algemeen ontwerp

We bekijken eerst eens het model dat we voorstellen. We geven hier een algemene oplossing voor het betreffende aspect. Later in dit hoofdstuk zullen we dit dan concreet in JAC implementeren voor onze toepassing.

Op figuur 6.1 zien we duidelijk een server en een client. Deze opdeling mag men letterlijk nemen. De client verzamelt zijn specifieke gegevens en geeft die dan door naar de centrale server die dan alles verwerkt. In deze sectie bespreken we uitgebreider de componenten zowel in de client (6.1.1) als in de server (6.1.2).



Figuur 6.1: Model van beslissingsmechanisme

6.1.1 Client

De taak van de client bestaat er in om ten eerste de methoden te onderscheppen en alle informatie die van belang kan zijn voor de toegangscontrole te verzamelen. Daarna worden die gegevens doorgestuurd naar de server. In de client wordt de binding verzorgd tussen de concrete applicatie op de client en de algemene toegangsbeveiliging op de server. De twee belangrijkste componenten hier zijn de **AccessWrapper** (6.1.1.1) en de **InformationRetriever** (6.1.1.2).

6.1.1.1 De AccessWrapper

In figuur 6.1 zien we dat de **AccessWrapper** de `read():void` onderschept. Dan worden eerst aan de **RoleManager** de rollen van het subject gevraagd (1.1 op de figuur). Het subject is in dit geval natuurlijk de persoon die de methode oproept. Via 1.2 wordt aan de **InformationRetriever** alle informatie opgevraagd die relevant kan zijn voor de toegangscontrole.

1.3 brengt alle gegevens samen en giet deze in een andere vorm, wat die vorm inhoudt, zien we later in 6.2. Tenslotte wordt dit alles naar de **AccessDecisionFunction** in de server doorgestuurd (1.4).

Wij hebben ervoor geopteerd om de **AccessWrapper** rechtstreeks met de **RoleManager** te laten communiceren. Eigenlijk kon de **InformationRetriever** de **RoleManager** aanspreken maar wij hebben ervoor gekozen dat de **InformationRetriever** enkel gegevens uit de interfaces verzamelijk zoals we onmiddellijk zullen zien. Als men bijvoorbeeld aan rolactivatie doet, dan moet de **Rolemanager** dit allemaal beheren en dit alles heeft dan totaal niets meer te maken met de interfaces.

6.1.1.2 De **InformationRetriever**

De taak van de **InformationRetriever** is zoals we daarpas kort hebben aangehaald het verzamelen van alle gegevens die te vinden zijn in de interface van zowel het subject als het object.

Het is misschien eens handig om terug te kijken naar de werking van het Uitgebreid View-Connectormodel (3.3.2). Volgens het model zijn de regels uit de beleidstaal vooral gebaseerd op interfaces en domeinen. De interfaces voor subjecten bevatten enkel attributen, terwijl die voor objecten zowel attributen als operaties, die op de objecten kunnen uitgevoerd worden, beschrijven. Beide soorten interfaces moeten dan per type geïmplementeerd worden door connectoren. Domeinen zijn dan gewoon een groep van objecten.

De **InformationRetriever** doet beroep op een **InterfaceConnectorList** en een **SubjectConnectorList**, respectievelijk om de connectoren voor het object en die voor het subject te verkrijgen. De **InterfaceConnectorList** houdt alle connectoren bij voor objecten die voorkomen op de client. Om de passende connectoren te verkrijgen moet men het object (de **wrappee**) en de methode meegeven (in figuur 6.1 operatie 1.2.1). Dan wordt een lijst teruggegeven van alle connectoren die van toepassing zijn (het kunnen er dus meerdere zijn maar in 6.2.1 komen we daar uitgebreider op terug). Aan de **SubjectConnectorList** wordt de connector van de aangemelde persoon gevraagd (1.2.2). Voor de eenvoud leggen we op dat er maar één connector is per type (in dit geval **person**), een uitbreiding van dit ontwerp zou zijn om net zoals bij het object meerdere connectoren per type toe te laten. Deze vereenvoudiging zorgt er mede voor dat het eigenlijke beslissingsmechanisme niet nog ingewikkelder wordt.

De **InformationRetriever** verkrijgt de connectoren via de twee **ConnectorLists**. De domeinen waarin de **wrappee** zich bevindt, worden opgevraagd via de **DomainManager** (1.2.3). Deze laatste weet tot welke domeinen een bepaald type behoort. Alle informatie (de connector voor de persoon, de lijst van connectoren en lijst van de domeinen van het object) wordt als teruggeefwaarde aan de **AccessWrapper** bezorgd.

6.1.2 Server

De server-kant van onze applicatie bevat alle benodigdheden om de toegangscontrole tot een goed einde te kunnen brengen: we hebben een **RuleManager**, een beheerder voor de beleidsregels; een **RuleMatcher** die alle regels die van toepassing zijn voor het concrete geval ophaalt uit de RuleManager; en ook een **AccessDecisionFunction** die beslist over de toegang op basis van die opgehaalde regels en van informatie over het object en subject.

De werking van de drie bovenstaande componenten wordt hierna beschreven: die van de RuleManager in 6.1.2.1, de werking van de RuleMatcher in 6.1.2.2 en tenslotte wordt in 6.1.2.3 getoond hoe de AccessDecisionFunction te werk gaat.

6.1.2.1 De RuleManager

RuleManager is de beheerder van de beleidsregels van ons programma. Bijvoorbeeld bij de start van het programma worden de regels ingelezen uit de beleidsbestanden en in RuleManager bijgehouden. Op die manier is het eenvoudig om de regels op te halen die de het toegangscontrolemechanisme nodig heeft om een beslissing te nemen.

Het eindresultaat van deze opeenvolging van methodes is dat enkel de regels die van toepassing zijn op alle meegegeven informatie, teruggegeven worden aan de *RuleMatcher* (1.4.1.1 in figuur 6.1). Wat die dan met die regels doet, wordt in het volgende punt beschreven.

6.1.2.2 De RuleMatcher

De rol van de RuleMatcher houdt in dat hij de juiste gegevens samenvoegt met de juiste regels. De regels worden verkregen van de RuleManager. Wat die verwerking precies inhoudt zien we in 6.2.3. Deze informatie per regel is namelijk nodig in de **AccessDecisionFunction**, waar de uiteindelijke beslissing over het verlenen van toegang valt.

6.1.2.3 AccessDecisionFunction

Eenmaal alle benodigde regels met hun bijhorende informatie beschikbaar zijn (via 1.4.1), kan het eigenlijke beslissingsmechanisme in gang treden. Dit mechanisme bevindt zich, zoals eerder al vermeld, in de AccessDecisionFunction. De uiteindelijke beslissing is afhankelijk van hoe de individuele regels geëvalueerd worden (1.4.2). Wanneer één van de regels positief geëvalueerd wordt, dan wordt er toegang verleend aan het subject.

Ook hier opteren we voor een eenvoudige oplossing. Bijkomend kan er bijvoorbeeld eerst gecontroleerd worden of er inconsistenties zijn tussen de regels. Dit kan natuurlijk gebeuren als men negatieve regels toelaat (wat bij ons niet het geval is, want onze taal biedt deze mogelijkheid niet aan). Als er zich conflicten voordoen, dan moeten die eerst opgelost worden.

6.2 Nieuwe gegevensstructuren

Het is de bedoeling alle gegevens die relevant zijn voor de beslissingen in de toegangscontrole gestructureerd te verzamelen en te encapsuleren. In de vorige sectie zagen we dat er heel wat informatie verzameld en doorgegeven wordt. Het encapsuleren gebeurt bij ons in java-objecten. We hebben in een eerder hoofdstuk (meer bepaald in 3.1.2) besproken dat de “schakel tussen beleidstaal en applicatie” afhankelijk is van het platform waar de applicatie draait. De concepten worden hier wel in Java geïmplementeerd, maar kunnen zonder problemen ook in andere (objectgerichte) talen gemaakt worden. Het is een algemeen toepasbare techniek die we proberen voor te stellen. Het gaat meer bepaald over drie nieuwe concepten: `MetaObject`, `MetaSubject` en `MetaPolicyRule`.

6.2.1 `MetaObject`

Het metaobject bevat bijkomende informatie van het doelobject en bestaat uit de volgende velden:

1. een lijst met alle mogelijke domeinen waartoe het object behoort
2. de attributen van de `access-interface` met hun waarde volgens de `view-connector`
3. de *action* waarop de *methode-invocatie* is gemapt in de `view-connector`

Het is natuurlijk mogelijk dat er voor één methode op een bepaald type object meerdere `access-interfaces` zijn. Het is belangrijk dat de informatie per interface verzameld wordt en telkens in één metaobject wordt gegroepeerd. Is dat niet zo, dan zou het erg moeilijk worden als er meerdere interfaces zijn om te zien welke info bij welk interface past. Als we al vooruit lopen op de werking van het beslissingsmechanisme dan wordt per metaobject gekeken of er regels voldoen. Dit is ook de reden waarom in ieder metaobject apart alle domeinen zitten. Een alternatief zou kunnen zijn die domeinen niet in het metaobject te stoppen, maar dan zitten niet alle gegevens omtrent één object geëncapsuleerd in het metaobject.

Tenslotte kan het nuttig zijn op te merken dat er in het metaobject op geen enkele manier nog een verwijzing te vinden is naar het object, met andere woorden in het uiteindelijk metaobject is er letterlijk en figuurlijk geen verwijzing naar de applicatie op de client. Natuurlijk zou men wel eventueel een link kunnen behouden naar het oorspronkelijke object. Dan moet men natuurlijk rekening houden met het feit dat het metaobject vaak in reële omgevingen over netwerken wordt doorgegeven, wat meebrengt dat de link vanop een andere machine moet kunnen gevolgd worden. Er zou ook een kopie kunnen geplaatst worden in het metaobject maar dan kan men nooit meer de oorspronkelijke link controleren.

6.2.2 `MetaSubject`

Aan de kant van het subject ligt het enigzins anders dan bij het object. Oorspronkelijk kon men in het `view-connector`model van [15] enkel de referentie naar het subject gebruiken bij het opstellen van een regel. Bij het uitgebreid `view-connector`model (3.3.2) introduceerden we ook een interface voor het subject zodat men aan een subject rollen kan toewijzen en zijn eventuele attributen kan gebruiken.

Op deze manier is de implementatie van het subject volledig transparant voor de beveiliging.

Subjecten groepeer men volgens rol en niet in domeinen zoals bij objecten. De verantwoordelijkheid van de rollen ligt volledig in handen van de `RoleManager`. Naast rollen kan men ook in het uitgebreid view-connector model ook attributen van het subject gebruiken. Een metasubject bevat de attributen die ingevuld worden via de view-connector en alle rollen van het subject. Eerder in dit hoofdstuk (6.1.1.2) hebben we de beperking besproken die oplegde dat er maar één connector per subject mag zijn. Op deze manier is er telkens maar één metasubject per subject.

6.2.3 MetaPolicyRule

Alle relevante gegeven van object en subject zitten nu respectievelijk in het metaobject en -subject. Aan de hand van deze laatste twee kan men regels ophalen. Men zoekt de regels aan de hand van de rol en interface van het subject én aan de hand van het domein en de interface van het object. Het is evident dat er meerdere regels kunnen voldoen (straks geven we een voorbeeld). Als er een regel is gevonden, dan wordt die regel samen met het overeenkomstig metaobject en -subject in een metaregel gestopt. Die metaregel bevat dan alle informatie die nodig is om over de regel te beslissen (zie werking 6.1.2.3).

In tegenstelling tot het metaobject wordt er wel een referentie bijgehouden in het metasubject omdat het oorspronkelijke View-Connectormodel die referentie kon gebruiken. Dit betekent dat men hiermee rekening moet houden als men de client en server op een verschillende machine laat draaien.

6.3 Beslissingsmechanisme voor toegangscontrole in JAC

In deze sectie is het de bedoeling om het algemeen model van 6.1 met behulp van de structuren uit 6.2 concreet om te zetten in JAC.

Het algemeen model had een client-serverarchitectuur. De componenten die daar werden voorgesteld gaan we nu implementeren.

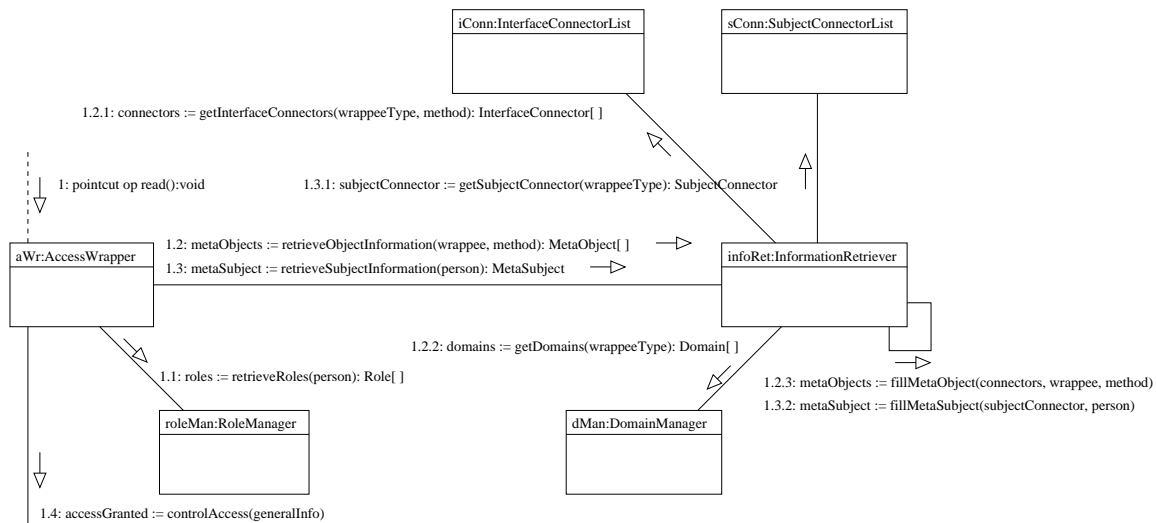
Een overzicht van hoe het beslissingsmechanisme werkt, vindt men in figuur 6.1.

6.3.1 Implementatie clientzijde

De implementatie van de clientzijde (zie figuur 6.2) is volledig gebaseerd op de client zoals voorgesteld in het algemeen ontwerp (figuur 6.1). Op het eerste zicht valt ook natuurlijk op dat de informatie nu in termen van metaobjecten of -subject wordt teruggegeven.

Als een methode onderschept wordt door de `AccessWrapper`, stuurt deze de wrappee (het object dus) en de naam van de methode door naar de `InformationRetriever` (1.2). De `InterfaceConnectorList` vraagt dan alle view-connectoren die passen voor het type van de wrappee en de methode die de gebruiker wil uitvoeren (1.2.1). Met behulp van reflectie worden dan de waarden van de attributen volgens de connectoren ingevuld. Daarnaast vraagt de `InformationRetriever` alle domeinen van de wrappee op (1.2.2) en voegt deze toe aan de metaobjecten

die dan op hun beurt aan de AccessWrapper worden teruggegeven.



Figuur 6.2: Overzicht client-kant

Om het ene metasubject te verkrijgen wordt weer beroep gedaan op de InformationRetriever (1.3). Die vraagt dan aan de SubjectConnectorList de overeenkomstige connector op (1.3.1). Opnieuw worden met behulp van reflectie de attributen ingevuld en wordt alles in het metasubject gestopt.

De AccessWrapper heeft nu alle nodige gegevens verzameld en geeft het metasubject en de één of meerdere metaobjecten door aan de serverlaag, meerbepaald de AccessDecisionFunction.

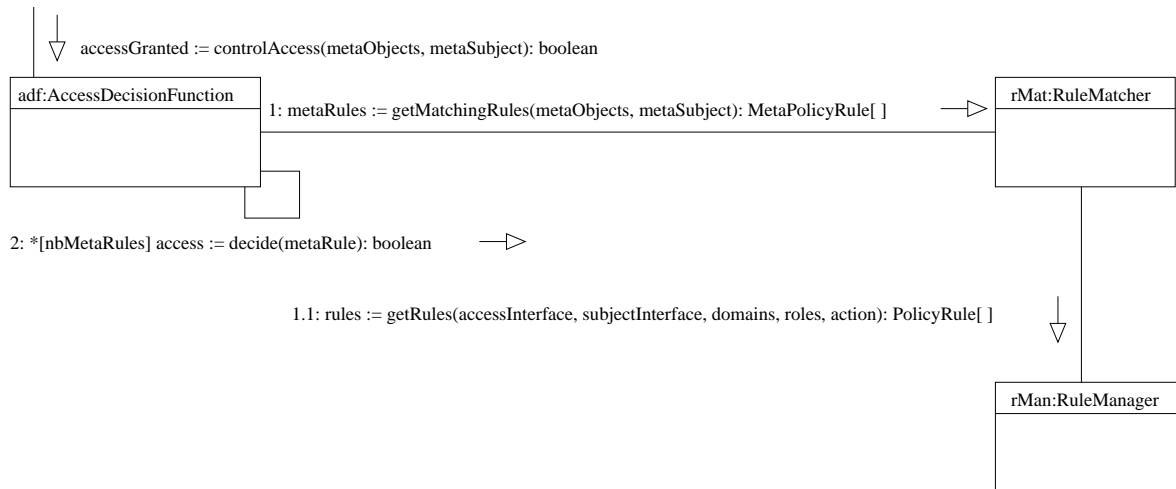
6.3.2 Implementatie serverzijde

Naar analogie van het algemeen model in figuur 6.1, bestaat de server uit drie delen. Een overzicht van hoe deze drie componenten samenwerken, kan in figuur 6.3 gevonden worden.

6.3.2.1 Implementatie van de RuleMatcher

Aan de RuleMatcher wordt door de AccessDecisionFunction gevraagd alle mogelijke metaregels te verkrijgen aan de hand van een metasubject en een aantal metaobjecten.

Concreet zal de RuleMatcher één voor één de metaobjecten overlopen en telkens de access-interface, het domein en de actie eruit halen. Deze drie gegevens worden dan iedere keer samen met de rollen en subject-interface doorgestuurd naar de RoleManager die dan de regels teruggeeft die slaan op deze vijf argumenten die werden meegegeven (1.1). Dan worden per verkregen regel de regel zelf met het overeenkomstige metaobject en het subject in een nieuwe metaregel geplaatst. Het resultaat van 1 uit de figuur is de lijst van alle metaregels.



Figuur 6.3: Overzicht server-kant

6.3.2.2 Implementatie RuleManager

De RuleManager krijgt de opdracht van de RuleMatcher om alle regels te geven die van toepassing zijn bij een gegeven access-interface, subject-interface, een aantal domeinen, een aantal rollen en één action. Wanneer aan de RuleManager gevraagd wordt de passende regels terug te geven (m.b.v. de methode `getRules` (1.1)), dan worden de methodes uit tabel 6.1, in respectievelijke volgorde, opgeroepen. Na `getRulesWithAction` moeten enkel nog de vereisten uit de beleidsregel gecontroleerd worden.

Methode	Beschrijving
<code>getRulesWithInterface</code>	Haalt alle regels op die van toepassing zijn op de <i>access-interface</i> in kwestie.
<code>getRulesWithSubjectInterface</code>	Haalt de regels op die van toepassing zijn op de <i>subject-interface</i> in kwestie. Hier worden enkel de regels die uit de vorige methode resulteren beschouwd.
<code>getRulesWithDomains</code>	Uit de regels van 2. worden die regels gehaald die van toepassing zijn op de <i>domeinen</i> in kwestie.
<code>getRulesWithRoles</code>	Geeft de regels terug die van toepassing zijn op de <i>rollen</i> in kwestie. Opnieuw worden enkel de regels uit het vorige punt beschouwd.
<code>getRulesWithAction</code>	Tenslotte worden de regels die de <i>actie</i> in kwestie bevatten, teruggegeven. Ook hier weer worden enkel die uit 4. bekeken.

Tabel 6.1: Opvragen regels

6.3.2.3 Implementatie AccessDecisionFunction

De AccessDecisionFunction verkrijgt alle metaregels van de RuleMatcher(1). In het algemeen model zeiden we dat de werkwijze van de AccessDecisionFunction erin bestaat een voor een

de regels te overlopen, en van zodra er één slaagt, er melding van te maken dat er toegang mag verleend worden.

Het beslissingsproces per regel bestaat erin de extra vereisten die in de tag “<requirements>” staan, te controleren. Die vereisten zijn allemaal van toepassing op eenerzijds het subject dat toegang wil verkrijgen tot een beveiligd object en anderzijds het object zelf. Sommige van die eisen leggen beperkingen op aan het subject zelf (zoals “`SubjectData = ReadPatientData.owner`”), andere vereisten bepalen dan weer dat een attribuut van het subject een bepaalde eigenschap moet hebben (zoals “`SubjectData.department = ReadPatientData.department-Patient`”).

De uiteindelijk genomen beslissing wordt tenslotte doorgegeven aan de `AccessWrapper`. Die laat dan afhankelijk van het resultaat de oorspronkelijk opgeroepen methode al dan niet uitvoeren. Wanneer de `AccessDecisionFunction` `true` teruggeeft, mag de methode uitgevoerd worden en roept de `AccessWrapper` dus de methode *proceed* op. In het andere geval, wanneer de `AccessDecisionFunction` `false` teruggeeft, wordt de uitvoering niet toegelaten en wordt dit aan de gebruiker meegedeeld.

6.4 Voorbeeld beleid voor toegangscontrole

We hebben nu alles om effectief een beleid uit te voeren op een applicatie. We nemen hier opnieuw de applicatie uit 5.3.1. Eerst stellen we de regels van het beleid op (6.4.1) waarna we dan de binding tussen de beveiling en de applicatie doorvoeren (6.4.2).

6.4.1 Opstellen van beleid

Uit wat we in hoofdstuk 1 te weten kwamen, proberen we een concreet beleid te halen. We stellen de volgende regels voor:

1. Een arts kan de patiëntgegevens van een record lezen én schrijven als hij zelf de verantwoordelijke is (de eigenaar ervan).
2. Een patiënt kan zijn eigen gegevens lezen.
3. Een verpleger kan maar patiëntgegevens lezen van patiënten die liggen op de afdeling waar de verpleger werkt.
4. Iemand van de administratieve dienst kan de administratieve gegevens van een patiënt lezen.

We willen al onmiddellijk erop duiden dat ons beleid niet volledig is. Het toont wel dat we niet-triviale regels in ons beleid kunnen gebruiken en hoe we die dan moeten omzetten. Hieronder kan men het bestand vinden dat de vier regels bevat (in dezelfde volgorde zoals we ze hierboven informeel beschreven). Dit bestand wordt ingelezen door de `RuleManager`.

De eerste regel is in het bestand te vinden vanaf lijn 2 tot 29. De regel is van toepassing op het domein *PatientDomain* en access-interface *PatientInterface*. Het subject moet de rol *Doctor*

hebben en bijkomende voorwaarde is dat het subject de *owner* moet zijn van het object. Als alle opgesomde condities vervuld zijn, dan mag het subject zowel lezen als schrijven (Read- en WriteAccess). De tweede regel (lijnen 30-54) gebruikt weer dezelfde interfaces. Nu moet het subject de rol van *Patient* hebben en volgens de voorwaarde zelf de patiënt van het object zijn. De actie die toegelaten is, is *ReadAccess*. De volgende regel (55-79) gebruikt weer dezelfde interfaces en domein. Nu wordt opgelegd dat een persoon met rol verpleger enkel gegevens kan lezen van een patiënt die ligt op de afdeling waar de verpleger werkt. De laatste regel (80-100) is van toepassing op een ander domein namelijk *AdministrationDomain* en ook een andere access-interface *AdministrationInterface*. Deze eenvoudige regel zegt dat een persoon met rol van *accountant* objecten uit zopas vernoemd domein mag lezen.

```

1 <file>
2 <policy-rule>
3 <domain>
4 PatientDomain
5 </domain>
6 <access-interface>
7 PatientInterface
8 </access-interface>
9 <subject-interface>
10 ClinicianInterface
11 </subject-interface>
12 <role>
13 Doctor
14 </role>
15 <requirements>
16 <requirement>
17 ClinicianInterface =
18 PatientInterface.owner
19 </requirement>
20 </requirements>
21 <actionList>
22 <action>
23 readAccess
24 </action>
25 <action>
26 writeAccess
27 </action>
28 </actionList>
29 </policy-rule>
30 <policy-rule>
31 <domain>
32 PatientDomain
33 </domain>
34 <access-interface>
35 PatientInterface
36 </access-interface>
37 <subject-interface>
38 ClinicianInterface
39 </subject-interface>
40 <role>
41 Patient
42 </role>
43 <requirements>
44 <requirement>
45 ClinicianInterface =
46 PatientInterface.patient
47 </requirement>
48 </requirements>
49 <actionList>
50 <action>
51 readAccess
52 </action>
53 </actionList>
54 </policy-rule>
55 <policy-rule>
56 <domain>
57 PatientDomain
58 </domain>
59 <access-interface>
60 PatientInterface
61 </access-interface>
62 <subject-interface>
63 ClinicianInterface
64 </subject-interface>
65 <role>
66 Nurse
67 </role>
68 <requirements>
69 <requirement>
70 ClinicianInterface.department =
71 PatientInterface.departmentPatient
72 </requirement>
73 </requirements>
74 <actionList>
75 <action>
76 readAccess
77 </action>
78 </actionList>
79 </policy-rule>
80 <policy-rule>
81 <domain>
82 AdministrationDomain
83 </domain>
84 <access-interface>
85 AdministrationInterface
86 </access-interface>
87 <subject-interface>
88 SubjectInterface
89 </subject-interface>
90 <role>
91 Accountant
92 </role>
93 <requirements>
94 </requirements>
95 <actionList>
96 <action>
97 readAccess
98 </action>
99 </actionList>
100 </policy-rule>
101 </file>

```

6.4.2 Binden met applicatie

Het binden met de applicatie (te vinden in 5.3.1) houdt in dat men de interfaces moet invullen en bepalen welke types tot welke domeinen behoren. Dit gebeurt, zoals we eerder vermeld hebben, in view-connectoren. We nemen dit alles niet in de tekst op maar wel in de bijlage omdat we in hoofdstuk 3 al een gelijkaardige binding hebben uitgevoerd. In bijlage A.1 zien we welke types tot welke domeinen behoren. Onze regels slaan op twee interfaces, die in A.2 gedefinieerd zijn, die dan ingevuld worden door de connectoren uit A.3.

6.5 Conclusie

We hebben in dit hoofdstuk besproken hoe ons toegangscontrolemechanisme precies in elkaar steekt. Het bestaat uit twee grote delen: een client die beveiligde oproepen onderschept en dan de nodige informatie verzamelt, en een server waar het beslissingsmechanisme zich op bevindt. Door het invoeren van nieuwe gegevensstructuren kunnen we de verzamelde informatie eenvoudiger doorgeven van de client naar de server. Een ander voordeel van die invoering is dat het beslissingsmechanisme op die manier onafhankelijk wordt van de applicatie. Tenslotte hebben we een voorbeeld gegeven van een beleid, en hebben we getoond hoe dat beleid door onze beveiliging kan worden gebruikt.

Hoofdstuk 7

Verfijningen en bevindingen

In dit afrondend hoofdstuk analyseren we ons werk. In sectie 7.1 geven we enkele mogelijke uitbreidingen en verfijningen van de toegangscontrole van onze beveiliging. In 7.2 schrijven we onze bevindingen neer over AOP en JAC die we ervaren hebben doorheen onze thesis. Als afsluiter ga we in 7.3 na hoe ons model kan gebruikt worden in een gedistribueerde omgeving, wat we hiervoor moeten doen en wat de mogelijke complicaties zijn.

7.1 Verfijningen toegangscontrole

In hoofdstuk 6 hebben we uitvoerig het aspect besproken dat de toegangscontrole uitvoert. Onze oplossing hebben we bewust eerst eenvoudig gehouden. Hier volgen enkele mogelijke verfijningen of alternatieven zonder dat we al te ver in detail treden.

Een eerste punt is eventueel een performanter beslissingsmechanisme te gebruiken omdat nu het controleren van de regels nogal primitief gebeurt (7.1.1). Men kan ook opteren voor een meer statische werking van de toegangscontrole door meer specifiek een statische *InformationRetriever* te gebruiken (7.1.2). Voorts kan het natuurlijk zijn dat de beveiliging tijdens de uitvoering zelf een beveiligde methode wil uitvoeren. Dan moet natuurlijk de toegang verleend worden als men er zeker van is dat deze enkel en alleen gebruikt wordt voor de beveiligingsmodule zelf (7.1.3). Verder stellen we dat het kan aangewezen zijn om te controleren of gebruikte regels en connectoren consistent zijn (7.1.4). Tenslotte bespreken we wat de gevolgen kunnen zijn van het gebruik van een complexere taal (7.1.5).

7.1.1 Beslissingsmechanisme kan efficiënter

Het hele proces dat nodig is om te beslissen over het al dan niet toegang krijgen tot de gewenste gegevens, is in ons programma zeker niet optimaal qua performantie. Het is dan ook niet het doel van deze thesis om een performante beveiliging te maken, het is voor ons vooral de bedoeling een onafhankelijke beveiliging te realiseren. Andere aspecten zoals efficiëntie komen zo op de achtergrond terecht. We schetsen hier kort enkele mogelijke optimalisaties

Bij het ophalen van de nodige informatie (in *InformationRetriever*) bijvoorbeeld, gebruiken we *reflectie*. Het is algemeen gekend dat dit niet de meest performante manier is, maar dit bespreken we verder in 7.1.2 iets gedetailleerder.

Een gerelateerd aspect is het bijhouden van de beleidsregels in *RuleManager*. De manier waarop de regels bijgehouden worden kan zeker beter, zodat het opvragen ervan efficiënter kan gebeuren. We zouden ze bijvoorbeeld in een soort van geïndexeerd systeem kunnen bijhouden, zodat het zoeken naar de regels die van toepassing zijn een heel stuk sneller kan gebeuren. Regels die vaak gebruikt worden, kunnen bijvoorbeeld in een cache bijgehouden worden.

Nu halen we alle regels op die overeenkomen met subject, object en methode, en dan pas gaan we een voor een van iedere regel de vereisten testen. Als er dan veel regels zijn, kan dit voor veel overhead zorgen zeker als het enkel de laatste regel is die toegang verschaft. Men kan dit verhelpen door bijvoorbeeld eerst de regels te controleren die het meest kans hebben op slagen. De kans op slagen kan natuurlijk van veel factoren afhangen (bv. geschiedenis van de regel, geschiedenis van het object, ...).

7.1.2 Statisch informatie ophalen

Zoals we het nu doen wordt de informatie, nodig voor de toegangscontrole, dynamisch opgehaald in de *InformationRetriever*. Telkens informatie nodig is, over het doelobject of het subject, worden de attributen (zoals die in respectievelijk de interface-connector en de subject-connector voorkomen) ingevuld met de concrete waarden uit het object of subject. Gezien dit ook met behulp van *reflectie* gebeurt, zorgt dit voor een behoorlijke overhead.

Het gebruik van een statische *InformationRetriever* zou het gebruik van reflectie moeten vermijden. Een mogelijke werkwijze is bij het aanmaken van de *InformationRetriever* alle attributen die in de connectoren staan, te mappen op de feitelijke implementaties. Met andere woorden eenmaal reflectie te gebruiken per attribuut, bij de start van het programma. Stel dat we in een connector een attribuut *owner* hebben met waarde *getResponsibleDoctor()*. Dan zou bijvoorbeeld een methode met java-signatuur *Object getAttribute(String attribute, Object wrappee, InterfaceConnector connector)*, de overeenkomstige methode volgens *attribute* en *connector* opzoeken en onmiddellijk uitvoeren op *wrappee*. In *getAttribute* wordt dus enkel nog een “invoke” uitgevoerd en geen reflectie meer, waarna het resultaat wordt teruggegeven. Een andere manier kan zijn om de *InformationRetriever* te integreren in de *AccessWrapper*. Op die manier bevat die al de correcte methodes en moet er bijgevolg noch reflectie noch een “invoke” gebeuren.

Wanneer men een statische *InformationRetriever* gebruikt, dan mag dit uiteraard niet ten koste zijn van het algemene dynamische karakter van de beveiliging.

7.1.3 Uitschakelen beveiliging tijdens beveiliging

Wat als het beveiligingssysteem zelf ook methodes oproept die in principe toegangscontrole vereisen? Die oproepen zouden automatisch moeten toegelaten worden, zonder dat het hele beveiligingsmechanisme (opnieuw) in werking treedt. Om het gevaar een beetje duidelijk te maken, geven we hier een kort voorbeeld.

Stel dat een methode *getFirstName()* op de klasse *Person* beveiligd is. Tijdens de toegangscontrole heeft het beslissingsmechanisme de voornaam van het subject nodig, die enkel kan verkregen worden met die methode *getFirstName()*. Maar omdat die methode beveiligd is, moet het mechanisme opnieuw in werking treden, waarvoor de voornaam van het subject nodig is, ...

We komen dus in een oneindige lus terecht. Om dit te vermijden zouden we dus op een of andere manier de beveiliging moeten kunnen uitschakelen, eenmaal het beslissingsmechanisme in werking treedt. Wanneer de beslissing dan is genomen, kan de beveiliging opnieuw worden ingeschakeld. Eén manier waarop dit kan gebeuren, is door een vlag te zetten per thread eenmaal het beslissingsmechanisme gestart is. Op die manier zal de methode *getFirstName()* uit het voorbeeld toch mogen uitgevoerd worden door het beveiligingsmechanisme zelf (de vlag zal namelijk uitstaan).

7.1.4 Controleren van inconsistentie in regels en connectoren

Momenteel worden de regels en connectoren niet gecontroleerd op inconsistenties. De XML-bestanden worden nu ingelezen en verwerkt zonder supervisie. De controle zou zowel moeten gebeuren op vlak van syntax als van inhoud.

Bij de syntax moet er dan op gelet worden dat er enkel geldige velden worden gebruikt. Inhoudelijk betekent dit dat wat er in de velden staat, effectief moet zijn wat er verwacht wordt. Bij connectoren zou het erop neerkomen dat men voor een connector aan de hand van de overeenkomstige interface alle velden controleert. Als de interface zegt dat er bijvoorbeeld attributen “naam” en “voornaam” zijn, dan moet de connector die deze interface implementeert die twee attributen ook bevatten. Nu worden de interfaces niet gebruikt in onze toepassing. We gebruiken ze enkel als referentie wanneer we handmatig de connectoren aanmaken. Inconsistenties in een regel zouden kunnen inhouden dat de regel onbestaande interfaces of attributen van interfaces gebruikt.

7.1.5 Functionaliteiten beleidstaal uitbreiden

De beleidstaal die wij gebruiken en in hoofdstuk 3 voorstelden is redelijk beperkt. Omdat in een medische toepassing het beleid bijna uitsluitend bestaat uit positieve regels hebben we geen aandacht geschonken aan de integratie van negatieve regels. Er zijn natuurlijk wel toepassingen waar men nood heeft aan negatieve regels.

Het toelaten van negatieve regels zou geen enkele invloed hebben op de interfaces en connectoren. In de regels zou men op één of andere manier moeten duidelijk kunnen maken dat het om een positieve respectievelijk negatieve regel gaat. De grootste verandering zou moeten plaatsvinden in het beslissingsmechanisme zelf dat nu ook moet controleren op inconsistenties in de regels. Het al dan niet verlenen van toegang zal niet meer alleen afhangen van het slagen van één positieve regel.

Een eenvoudigere uitbreiding zou kunnen zijn het integreren van tijd in de regels. Ook hier zouden we eventueel gebruik kunnen maken van een connector die de attributen (datum, uur, ...) uit het systeem haalt. We hebben hier maar kort twee verfijningen aangehaald, het is duidelijk dat onze taal nog verder uitgebreid kan worden wanneer men ze vergelijkt met andere bestaande beleidstalen.

7.2 Bevindingen

Het onderliggend doel in onze thesis was onderzoeken in hoeverre AOP, en meerbepaald JAC zich leende om aan beveiliging te doen. Hier gaan we na of we effectief bereikt hebben wat we wilden door eerst algemeen beveiliging via AOP te bespreken (7.2.1) en dan specifiek AOP met behulp van JAC te bekijken (7.2.2).

7.2.1 Beveiliging via AOP

In de literatuur is al meermaals beschreven dat AOP in grote mate geschikt is om beveiliging als belang af te scheiden van de rest van de applicatie. We bespreken hier kort de voor- en nadelen (in respectievelijk 7.2.1.1 en 7.2.1.2) van het gebruik van AOP.

7.2.1.1 Voordelen

Het grootste voordeel van aspectgeoriënteerd ten opzichte van klassiek objectgericht programmeren is de mogelijkheid een grotere onafhankelijkheid van de onderliggende toepassing te realiseren. Wanneer men beveiliging met gewone objectgerichte principes wil implementeren, zal de beveiligingscode verstrengeld en verspreid zitten in de code van de toepassing. Met behulp van aspecten kan men echter een geavanceerde scheiding van belangen bekomen (zie hoofdstuk 2).

Volledige scheiding is, zoals we in hoofdstuk 1 zagen, quasi onmogelijk. Zo hebben we wel in onze applicatie (zie 5.3.1) enkele wijzigingen moeten doorvoeren in verband met de rollen. We hebben elke rol een `department` gegeven, omdat het handig kan zijn dat men weet op welke afdeling iemand een bepaalde functie (rol dus) uitoefent. Voor de rest is de applicatie wel onafhankelijk van de beveiliging. Een ander probleem betreft wat er moet gebeuren wanneer de toegang geweigerd is. Misschien moeten er bepaalde handelingen die ervoor plaats hebben gevonden, ongedaan worden gemaakt. De beveiliging moet dan dus weet hebben van hoe de applicatie eruit ziet, om die annulering te kunnen doorvoeren.

Voor het vervullen van extra niet-functionele vereisten biedt AOP vaak een eenvoudige oplossing. We merken hier nogmaals op dat dit niet altijd het geval is, bijvoorbeeld wanneer het om complexere vereisten gaat. Zo was het niet triviaal om onze toegangscontrole in een aspect te gieten.

7.2.1.2 Nadelen

Het belangrijkste nadeel bij het gebruik van aspecten is dat er geen formalismes bestaan voor het modelleren van aspecten, en in het bijzonder communicatie tussen aspecten. Als gevolg hiervan hebben we zelf nog moeten zoeken hoe we die communicatie het beste tot stand brachten. Dit hebben we uitgebreid besproken in sectie 5.2.

Daarmee gerelateerd hebben we vastgesteld dat het via AOP wel mogelijk is om de beveiliging op zijn geheel af te scheiden, maar in de beveiliging zelf was het onmogelijk om de aspecten volledig onafhankelijk van elkaar te modelleren. Er in onze oplossing een expliciete afhankelijkheid tussen het aspect dat de authenticatie en de sessies regelt, en het aspect dat verantwoordelijk is voor de toegangscontrole.

7.2.2 AOP via JAC

Wij hebben in de eerste plaats JAC als programmeeromgeving geëxploreerd en meerbepaald hoe we met JAC een beveiliging kunnen bouwen. JAC biedt zelf al een resem van *libraries* aan. De mogelijkheden van JAC zijn zeker niet beperkt, wij hebben enkel maar de mogelijkheden van JAC bekeken specifiek voor beveiliging zonder specifiek bestaande libraries te gebruiken. We bekijken hier meerbepaald de goede punten die we ervoeren met JAC in 7.2.2.1 en de negatieve punten in 7.2.2.2

7.2.2.1 Voordelen

a) Uitbreiding op Java

Java is tegenwoordig een van de meest gebruikte programmeertalen. De meeste programmeurs zijn vertrouwd met Java en bovendien kan men gewone javaklassen zonder probleem gebruiken. Er zijn wel enkele uitzonderingen maar daar heeft JAC dan eigen klassen voor geschreven (bv. `JacVector` vervangt `Vector`).

b) Dynamisch karakter

In tegenstelling tot bijvoorbeeld AspectJ is het mogelijk om dynamisch aspecten in en uit te weven. Hét grote voordeel is dat men dus tijdens de uitvoering van het systeem kan kiezen welke aspecten men wil gebruiken. Zo moet men bij verandering van bijvoorbeeld het login-mechanisme, niet het hele systeem stilleggen maar enkel het gewijzigde aspect invoegen. Dit werd eerder in deze tekst al besproken.

c) Expressiviteit

De syntax voor de configuratiebestanden van JAC (*.acc-bestanden) is heel eenvoudig. Bijgevolg is het definiëren van pointcuts niet ingewikkeld. Al wat moet meegegeven worden, is waar (welke objecten, klassen en methodes) het aspect moet ingrijpen en welke methode uit de wrapper dan moet worden uitgevoerd. Een voorbeeld hiervan kan in 4.1.2 op p. 47 gevonden worden.

Ook het feit dat we niet al te veel problemen ondervonden om het Uitgebreid View-Connectorenmodel in JAC te implementeren duidt wel op enige graad van expressiviteit.

7.2.2.2 Nadelen

a) *Gebrek aan maturiteit*

JAC zit nog steeds in zijn ontwikkelingsfase (pas in 2002 ontwikkeld), waardoor er nog een aantal kinderziektes in te vinden zijn. Zo zitten de klassen `JacHashtable` en `JacVector` niet in het jar-bestand van JAC. Gelukkig was de broncode beschikbaar en konden we zo die twee klassen zelf compileren en aan dat bestand toevoegen.

In de beginfase van de ontwikkeling gebruikten we ook een gebruikersinterface die door JAC zelf gegenereerd werd. Ondermeer omdat die verschillende waarschuwingen (*warnings*) gaf, zijn we overgeschakeld op een zelfgemaakte interface.

De makers van het raamwerk zijn zich echter bewust van de aanwezigheid van deze bugs, en stellen alles in het werk om ze te verhelpen.

b) *JAC en overerving*

We hebben ook problemen ondervonden bij het gebruik van pointcuts op klassen die zich in een hiërarchie bevinden. Wanneer een methode op een subklasse wordt opgeroepen die niet specifiek in die klasse geïmplementeerd is (dus wel in een superklasse), dan zal een pointcut op die methode niet geactiveerd worden.

Om dit wat duidelijker te maken geven we een klein voorbeeldje.

Stel dat we twee klassen hebben: *Person* en *Woman*. *Person* heeft een methode “`getName()`”:

```
public class Person {
    public Person(String name) {
        this.name = name;
    }
    ...
    public String getName() {
        return name;
    }
    ...
    private String name;
}
```

De klasse *Woman* is een subklasse van *Person*, waarin de methode “`getName()`” niet expliciet gedefiniëerd is:

```
public class Woman extends Person {
    public Woman(String name) {
        super(name);
    }
    ...
    public String getGender() {
        return "Female";
    }
    ...
}
```

Stel: er is een pointcut gedefinieerd op de methode `getName()` van de klasse `Person`. Wanneer een gebruiker die methode nu oproept op een object van het type `Woman`, een subklasse van `Person`, zal de pointcut niet geactiveerd worden. Oorzaak hiervan is dat JAC niet overweg kan met dynamische binding, wat betreft het plaatsen van pointcuts. Vandaar dat het dus noodzakelijk was om in elke subklasse elke methode waarop een pointcut geplaatst is, expliciet te definiëren.

c) *toString()*

JAC gebruikt blijkbaar intern de methode `toString()` van een klasse, waardoor ze al wordt opgeroepen, nog voor bepaalde attributen ingevuld zijn. Gevolg is dat er een *NullPointerException* gegooid wordt, waardoor de uitvoering van het programma beëindigd wordt. Een echt sluitende oplossing hebben we hier niet voor gevonden, al wat we konden doen is ervoor zorgen dat die methode geen attributen nodig heeft, die bij een nieuwe instantiatie niet onmiddellijk worden ingevuld.

d) *Geen recursieve wrapping*

Het is niet mogelijk om rond een aspect nog een ander aspect te wrappen. De architectuur van JAC is namelijk zo ontworpen dat klassen vertaald moeten worden op bytecode-niveau om ervoor te zorgen dat er rond hen gewrapped kan worden. Aangezien wrappers en aspect componenten niet door JAC vertaald worden, kan er bijgevolg ook niet rond hen gewrapped worden. We hebben dit al besproken toen we spraken over communicatie tussen aspecten, meerbepaald bij de bespreking van het model waar een aspect rond een ander aspect gewrapped wordt (5.2.1.4).

7.3 Distributie

Zoals we de beveiliging nu geïmplementeerd hebben, wordt alles op dezelfde locatie uitgevoerd. Maar we hebben wel een aanzet gegeven om het gedistribueerd te kunnen doen. Zoals in figuur 6.1 op p. 67 wordt weergegeven, kan het beslissingsmechanisme (dat *AccessDecisionFunction*, *RuleManager* en *RuleMatcher* omvat) op een andere locatie, meerbepaald een server, worden ontplooid.

JAC zelf biedt enkele concepten aan die distributie ondersteunen (7.3.1). Daarna maken we een gedistribueerde applicatie (7.3.2) en bekijken we de schaalbaarheid van onze beveiliging in zo een omgeving (7.3.3).

7.3.1 JAC en distributie

Om distributie te verwezenlijken kan bijvoorbeeld Java RMI gebruikt worden. JAC biedt evenwel zelf een mechanisme aan om bepaalde modules op verschillende locaties te ontplooiën. Concreet gebeurt dit met het *Deployment-aspect* (`org.objectweb.jac.aspects.distribution.DeploymentAC`), dat in JAC voorzien is. Dit aspect bevat een aantal ontplooiingsregels die de programmeur kan gebruiken om zijn toepassing over een aantal containers te ontplooiën. Eenmaal de gedistribueerde ontplooiing een feit is, kunnen er nog bijkomende eigenschappen opgelegd worden. Zo zijn er in JAC ook nog het *Consistentie-aspect* (`org.objectweb.jac.aspects.distribution.ConsistencyAC`) en het *Load-Balancing-aspect* (`org.objectweb.jac.aspects.distribution.LoadBalancingAC`).

7.3.2 Gedistribueerde applicatie

De door ons ontwikkelde beveiliging is voorzien op distributie (zie 6.1). Daar hebben we namelijk al een client-server architectuur. In figuur 7.1 tonen we hoe we de componenten van de beveiliging op verschillende machines kunnen plaatsen.

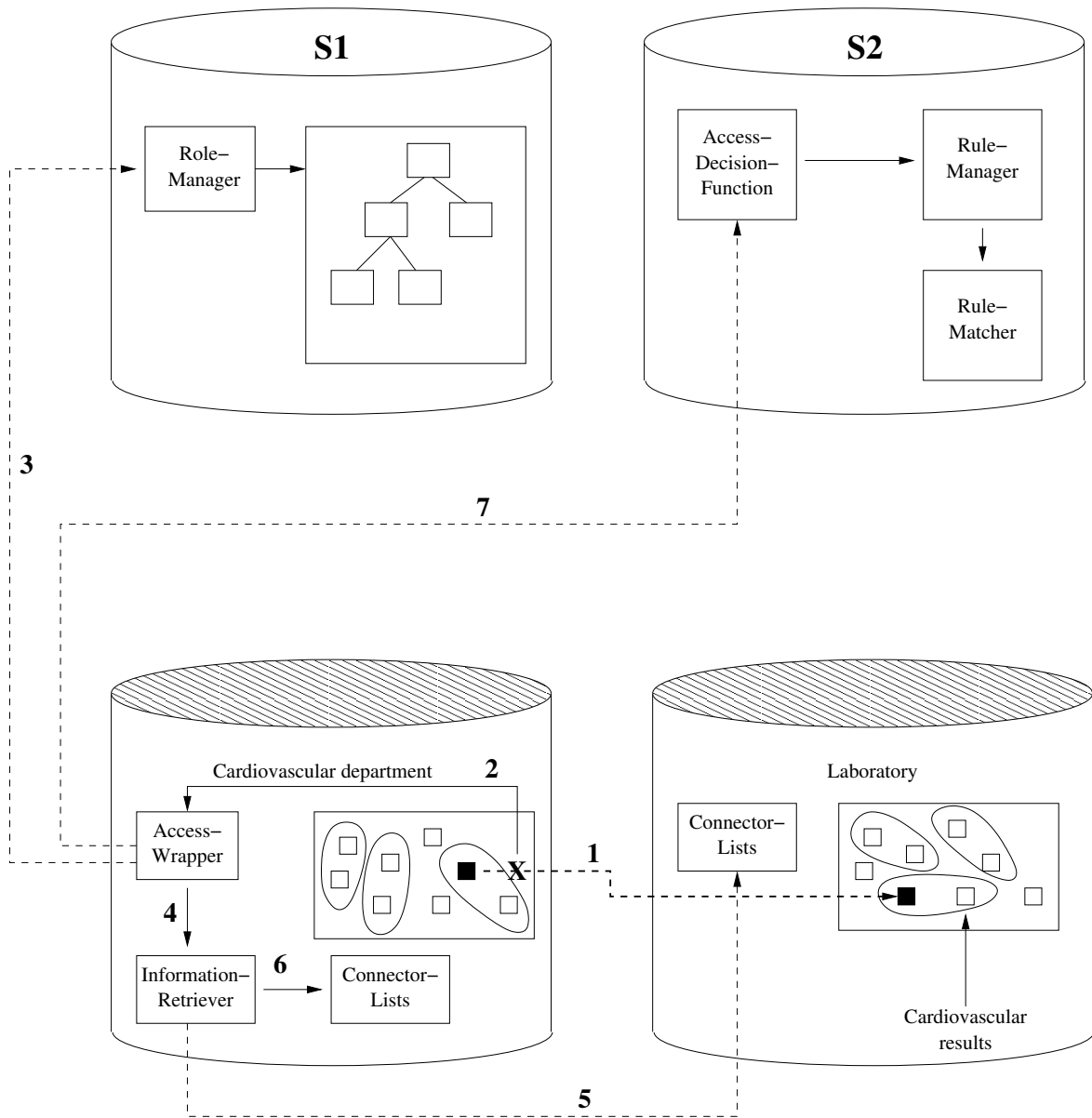
We nemen aan dat een ziekenhuis bestaat uit verschillende afdelingen, elk met hun eigen systeem. Verder zijn er twee servers waar elk systeem mee kan communiceren. Op de eerste server (**S1**) bevindt zich de *RoleManager*, die instaat voor het beheer van de rollen. Op een tweede server (**S2**) staat het beslissingsmechanisme dat door de toegangscontrole wordt gebruikt.

In figuur 7.1 worden twee afdelingen voorgesteld, namelijk het laboratorium en de afdeling Hart- en Vaatziekten (cardiovascular department). Een cardioloog wil nu de resultaten bekijken van een test die door het laboratorium is uitgevoerd (op de figuur aangeduid met 1). Die gegevens bevinden zich op het systeem van het labo. Op het moment dat de specialist de resultaten wil inkijken, springt de beveiliging in (2). Om te kunnen beslissen of de cardioloog toegang krijgt, heeft de beveiliging bepaalde informatie nodig, zowel over het object (5) als over het subject (6). Voor de informatie over het object moet er wel gecommuniceerd worden met het systeem waarop het zich bevindt, in de figuur dus dat van het laboratorium. Naast die informatie moet ook achterhaald worden welke rollen de gebruiker heeft. Die worden opgevraagd aan de *RoleManager*, die zich op de server **S1** bevindt (3). Eenmaal de nodige informatie beschikbaar is, geeft de *AccessWrapper* aan dat het beslissingsmechanisme in werking mag treden (7).

7.3.3 Schaalbaarheid van onze oplossing

Voor elke applicatie is het noodzakelijk om connectoren te definiëren. Aan de interfaces en beleidsregels moet er echter niets veranderd worden, bij ingebruikname van een andere toepassing. Wanneer een toepassing nu uit een groot aantal componenten bestaat, moet men heel wat connectoren invullen. Dat op zich is niet echt een probleem, gezien het voor een stuk kan opgelost worden met behulp van overerving. Er kunnen dan namelijk connectoren voor de topklassen van een hiërarchie gemaakt worden, die ook toepasbaar zijn voor klassen die zich lager in de hiërarchie bevinden.

Zoals we echter zagen in punt b) van 7.2.2.2, geeft JAC juist problemen bij het uitbuiten van die overerving. Gezien connectoren dan op elke klasse moeten gedefinieerd worden, resulteert dit in een enorm aantal connectoren. Hoewel ons model wel degelijk schaalbaar is, zal de implementatie met behulp van JAC dat niet zijn, zolang die problemen niet opgelost raken.



Figuur 7.1: Gedistribueerd model

7.4 Conclusie

We hebben in dit hoofdstuk enkele verfijningen voor onze beveiliging gezien. Zo kan de performantie ervan opgedreven worden door bijvoorbeeld de beleidsregels op een andere manier bij te houden. Een andere verfijning kan erin bestaan de nodige informatie statisch op te halen, zodat er minder reflectie voor nodig is. Daarnaast kan het nuttig zijn om het mogelijk te maken de beveiliging uit te schakelen op het moment dat ze al aan het werk is. We hebben ook opgemerkt dat voor de beleidstaal die wij gebruiken een aantal uitbreidingen mogelijk zijn, zoals bijvoorbeeld het toelaten van negatieve regels. Het moet ook mogelijk zijn om er een compiler voor te schrijven die dan inconsistenties tussen conflicterende regels oplost.

Vervolgens hebben we het gehad over onze ervaringen met aspectgeoriënteerd programmeren en in het bijzonder met behulp van Java Aspect Components. We hebben zowel enkele voor- als nadelen besproken. Tenslotte hebben we besproken dat het mogelijk is om ons programma ook gedistribueerd te ontplooien. Dit hebben we aangetoond met een voorbeeld over hoe die ontplooiing er zou kunnen uitzien.

Besluit

We zijn in deze thesis gestart met het bepalen van wat de vereisten zijn voor medische data. We hebben ondervonden dat het beveiligen van medische gegevens een complex probleem is. Ten eerste zijn er veel gebruikers van de applicatie die vaak totaal verschillende belangen hebben, zoals specialisten, verplegers of gewoon administratief personeel. Ten tweede is er veel informatie die moet beschermd worden tegen zowel externen als internen. We hebben ook gemerkt dat de toegang tot medische gegevens bepaald wordt door een veelheid aan regels. Daarbij komt nog dat die regels kunnen veranderen door bijvoorbeeld een wetswijziging, een fusie of verandering van beleid. De uitdaging bestond erin een beveiliging te ontwerpen die rekening houdt met dit alles. Het kwam er voor ons dus op neer om een flexibele toegangscontrole te ontwikkelen.

Verder vonden we in de literatuur twee oorzaken voor de complexiteit van softwarebeveiliging. Naast veilig coderen is er ook het feit dat de beveiligingslogica verstrengeld en verspreid is in de code van de onderliggende toepassing. Om deze complexiteit aan te pakken, deden we beroep op het principe van het *scheiden van belangen*. Die scheiding kan (grotendeels) bekomen worden door gebruik te maken van *aspectgeoriënteerd programmeren (AOP)*. We tekenden de drie *krijtlijnen* uit voor onze thesis: ten eerste moesten we de verstrengeling en verspreiding van de beveiligingslogica beheeren, daarnaast moesten we de beveiliging kunnen aanpassen aan veranderende omstandigheden en tenslotte moesten we een beveiligingsbeleid op een hoog niveau kunnen realiseren.

Om de laatste krijtlijn uit te werken bestudeerden we eerst twee bestaande talen. Daar zagen we dat Ponder en SPL geen sluitende oplossing bieden voor de binding tussen de regels van het beleid en een onderliggende applicatie. De taal die we zelf ontwikkelden is niet zo geavanceerd als de bestaande talen, maar het belangrijkste is dat we een model, namelijk het Uitgebreid View-connectormodel, ontwikkelden dat onze algemene beleidstaal kan binden met een specifieke applicatie. Later kan de voorgestelde taal eventueel verder uitgebreid worden om dezelfde functionaliteiten te bevatten als de huidige gebruikte systemen, maar dit valt buiten het doel van onze thesis.

Om het probleem van de tweede krijtlijn, de mogelijkheid om de beveiliging te kunnen aanpassen aan veranderende omstandigheden, te kunnen aanpakken, hebben we *JAC (Java Aspect Components)* bestudeerd, een raamwerk voor aspectgeoriënteerd programmeren in Java. Het verschil met andere aspectgeoriënteerde systemen zoals AspectJ, is het dynamische karakter ervan. In JAC kan men aspecten namelijk at runtime in- en uitweven, en zelfs herconfigureren. Dit is een groot voordeel omdat de beveiliging (toegangscontrole, authenticatie, ...) op die manier eenvoudig kan worden aangepast wanneer de situatie dit vereist.

Door onze beveiliging met behulp van AOP te ontwikkelen vonden we zo een oplossing voor de eerste krijtlijn die oplegde dat men met de verstrengeling en verspreiding van de beveiligingslogica overweg moet kunnen. De door ons ontwikkelde beveiliging bestaat uit twee aspecten: een authenticatie- en een toegangscontroleaspect. Om die twee te laten samenwerken zijn we op zoek moeten gaan naar mogelijke manieren om communicatie tussen aspecten te verwezenlijken. Dat was geen sinecure, gezien er momenteel nog geen richtlijnen bestaan over hoe die communicatie gerealiseerd kan worden. Het aspect dat de toegangscontrole regelt, maakt gebruik van het Uitgebreid View-connectormodel dat de beleidstaal bindt met een applicatie. Dit aspect bestaat uit twee grote delen: een client die beveiligde oproepen onderschept en de nodige informatie verzamelt, en een server waar het beslissingsmechanisme zich op bevindt. Door het invoeren van nieuwe gegevensstructuren kunnen we de verzamelde informatie eenvoudiger doorgeven van de client naar de server. Een ander voordeel van die invoering is dat het beslissingsmechanisme op die manier onafhankelijk wordt van de applicatie.

We hebben ook enkele verfijningen voor onze beveiliging voorgesteld. Zo kan de performantie ervan opgedreven worden door bijvoorbeeld de beleidsregels op een andere manier bij te houden. Een andere verfijning kan erin bestaan de nodige informatie statisch op te halen, zodat er minder reflectie voor nodig is. Daarnaast kan het nuttig zijn om het mogelijk te maken de beveiliging uit te schakelen op het moment dat ze al aan het werk is. We hebben ook opgemerkt dat voor de beleidstaal die wij gebruiken een aantal uitbreidingen mogelijk zijn, zoals bijvoorbeeld het toelaten van negatieve regels. Het moet ook mogelijk zijn om er een compiler voor te schrijven die dan inconsistenties tussen conflicterende regels oplost.

Vervolgens hebben we het gehad over onze ervaringen met aspectgeoriënteerd programmeren en in het bijzonder met behulp van Java Aspect Components. We hebben zowel enkele voor- als nadelen besproken. Tenslotte hebben we getoond hoe het mogelijk is om ons programma in een gedistribueerde omgeving te ontplooiën.

In de loop van deze thesis hebben we ondervonden dat aspectgeoriënteerd programmeren een grote verbetering inhoudt tegenover objectgerichte implementaties op het vlak van scheiden van belangen, maar dat er toch nog veel werk aan de winkel is om de kinderziektes van AOP weg te werken. We hebben in dit werk bewezen dat het ontwikkelen van een beveiliging die bijna volledig onafhankelijk is van de onderliggende applicatie wel degelijk mogelijk is. We hopen dan ook dat hier in de toekomst kan op verder gebouwd worden.

Bijlage A

Binding applicatie

A.1 Domeinen

```
<file>
  <domain>
    <name>
      PatientDomain
    </name>
    <typeList>
      <type>
        application.Patient
      </type>
      <type>
        application.record.CardioRecord
      </type>
      <type>
        application.record.PhysioRecord
      </type>
      <type>
        application.record.PsychoRecord
      </type>
    </typeList>
  </domain>
  <domain>
    <name>
      AdministrationDomain
    </name>
    <typeList>
      <type>
        application.record.GeneralRecord
      </type>
      <type>
        application.Accountant
      </type>
    </typeList>
  </domain>
</file>
```

A.2 Interfaces

A.2.1 Access-interfaces

```
<file>
  <access-interface>
    <interface>
      PatientInterface
    </interface>
    <attributes>
      <attribute>
        owner
      </attribute>
      <attribute>
        patient
      </attribute>
      <attribute>
        departmentPatient
      </attribute>
    </attributes>
    <actions>
      <action>
        readAccess
      </action>
    </actions>
  </access-interface>
  <access-interface>
    <interface>
      AdministrationInterface
    </interface>
    <attributes>
      <attribute>
        patient
      </attribute>
      <attribute>
        departmentPatient
      </attribute>
    </attributes>
    <actions>
      <action>
        readAccess
      </action>
    </actions>
  </access-interface>
</file>
```

A.2.2 Subject-interface

```
<file>
  <subject-interface>
    <interface>
      ClinicianInterface
    </interface>
    <attributes>
      <attribute>
        department
      </attribute>
    </attributes>
  </subject-interface>
</file>
```

A.3 View-connectoren

A.3.1 Object

Voorlopig hebben we maar enkel een view-connector voor GeneralRecord en CardioRecord, maar met wat knip en plak werk kunnen voor de andere records zeker de connectoren ook gemaakt worden.

```
<file>
  <view-connector>
    <type>
      application.record.CardioRecord
    </type>
    <access-interface>
      PatientInterface
    </access-interface>
    <attributes>
      <attribute>
        <name>
          owner
        </name>
        <value>
          getResponsibleDoctor()
        </value>
      </attribute>
      <attribute>
        <name>
          patient
        </name>
        <value>
          getPatient()
        </value>
      </attribute>
      <attribute>
        <name>
          departmentPatient
        </name>
        <value>
          getPatient().getPrincipalRole().getDepartment()
        </value>
      </attribute>
    </attributes>
    <actions>
      <action>
        <name>
          readAccess
        </name>
        <value>
          read():void
        </value>
      </action>
      <action>
        <name>
          writeAccess
        </name>
        <value>
          write(java.lang.String):void
        </value>
      </action>
    </actions>
  </view-connector>
```

```

<view-connector>
  <type>
    application.record.GeneralRecord
  </type>
  <access-interface>
    AdministrationInterface
  </access-interface>
  <attributes>
    <attribute>
      <name>
        patient
      </name>
      <value>
        getPatient()
      </value>
    </attribute>
    <attribute>
      <name>
        departmentPatient
      </name>
      <value>
        getPatient().getPrincipalRole().getDepartment()
      </value>
    </attribute>
  </attributes>
  <actions>
    <action>
      <name>
        readAccess
      </name>
      <value>
        read():void
      </value>
    </action>
  </actions>
</view-connector>
</file>

```

A.3.2 Subject

```

<file>
  <view-connector>
    <type>
      application.Person
    </type>
    <subject-interface>
      ClinicianInterface
    </subject-interface>
    <attributes>
      <attribute>
        <name>
          department
        </name>
        <value>
          getPrincipalRole().getDepartment()
        </value>
      </attribute>
    </attributes>
  </view-connector>
</file>

```

Bijlage B

Het programma

B.1 Installatie

JAC kan gevonden worden op <http://jac.objectweb.org/download.html>. Uitleg over welke packages nodig zijn, kan worden gevonden op de download-site.

Standaard ontbreken er in het bestand *jac.jar* enkele klassen die noodzakelijk zijn voor ons programma. Die klassen zijn `org.objectweb.jac.core.utils.JacHashtable` en `org.objectweb.jac.core.utils.JacVector`. De broncode van JAC kan gedownload worden van de site, zodat de gebruiker die klassen zelf kan compileren en toevoegen aan het jar-bestand. Houd hierbij wel rekening met de pad-namen van die twee klassen.

B.2 Werking

De uitvoering van het programma wordt gestart met het volgende commando (vervang *<jac-dir>* en *<programma-dir>* respectievelijk door de map waar JAC en die waar het programma geïnstalleerd zijn):

```
java -Djava.security.policy=<jac-dir>\jac.policy
    -jar <jac-dir>\jac.jar
    -w
    -R <jac-dir>
    -C <programma-dir>\classes
    <programma-dir>\files\thesis.jac
```

In wat volgt wordt een kort voorbeeld uitgewerkt om de werking van het programma te illustreren. Om het voorbeeld zelf uit te voeren, volstaat het om de cursieve regels te bekijken. Eenmaal het programma gestart is, verschijnt er een venster dat de gebruiker vraagt om in te loggen.

*Log in met gebruikersnaam **tim** en paswoord **mat**.*



Na het inloggen wordt het hoofdvenster van het programma getoond. Zoals men kan zien, heeft de gebruiker op dit moment drie opties:

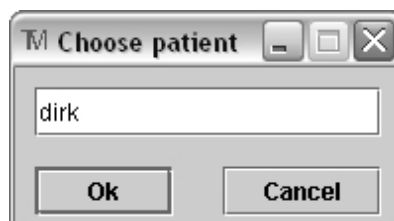


List records: De naam van een patiënt ingeven van wie de gebruiker een record wil lezen of schrijven.

End session: De huidige sessie beëindigen, met als gevolg dat wanneer er vanaf nu “List Records” wordt gekozen, de gebruiker zich opnieuw moet authenticeren.

Quit program: De uitvoering van het programma beëindigen.

*Kies de optie **List records**, geef als naam van de patiënt **dirk** in, en druk op **Ok**.*

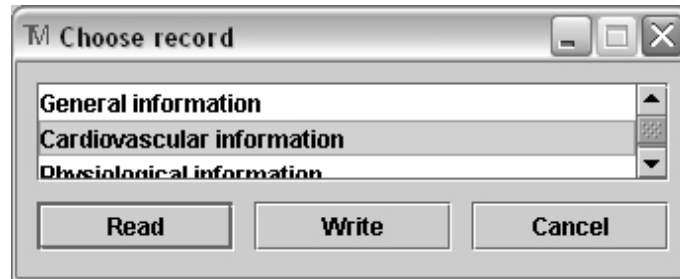


Na het ingeven van de naam van een patiënt, wordt aan de gebruiker de mogelijkheid gegeven om een record van die patiënt te kiezen. Het volstaat om het gewenste record in de lijst aan te duiden en dan op de knop te drukken van de actie die men erop wil uitvoeren:

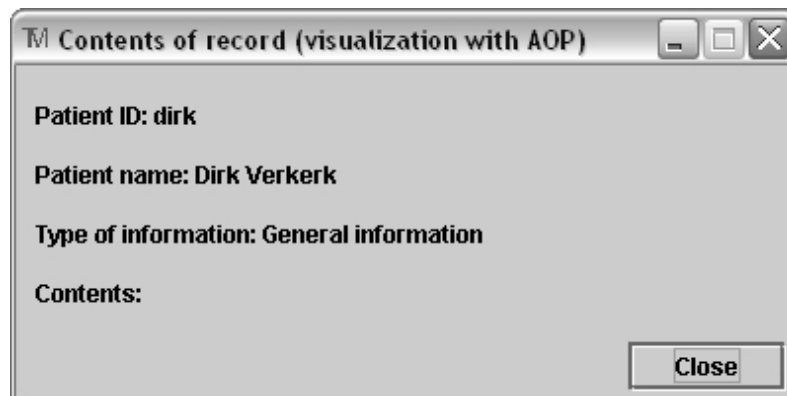
Read: Lezen van de informatie uit het record.

Write: Schrijven van nieuwe informatie in het record. Hier wordt in een nieuw venster gevraagd wat de informatie is die de gebruiker wil schrijven. Daarna wordt die tekst dan in het record geschreven.

*Kies het record **Cardiovascular information** en druk op **Read** om de inhoud ervan te lezen.*



Na het uitvoeren van de gewenste actie, wordt de informatie van het record getoond.



Bibliografie

- [1] Patiënteninformatie in ziekenhuizen slecht beveiligd. *De Standaard*, 1 April 2004.
- [2] Patiëntengegevens zwak beveiligd. *De Standaard*, 1 April 2004.
- [3] Dr. R. J. Anderson. Security in clinical information systems. *Computer Laboratory, University of Cambridge*, Januari 1996.
- [4] General Medical Council. Good medical practice. <http://www.gmc-uk.org/standards/GMP.pdf>, Mei 2001.
- [5] Office of Technology Assessment US Government Printing Office. Protecting privacy in computerized medical information. 1993.
- [6] B. Darley & A. Griew & K. McLoughlin & J. Williams at HMSO. *How to keep a clinical confidence: A Summary of Law and Guidance on Maintaining the Patient's Privacy*. The Stationery Office Books, Oktober 1994.
- [7] Wet tot bescherming van de persoonlijke levenssfeer ten opzichte van de verwerking van persoonsgegevens, gecoördineerde versie. http://www.privacy.fgov.be/normatieve_teksten/NL_8-12-92_levenssfeer.pdf, 2003.
- [8] Verenigd Koninkrijk. Data protection act. 1996.
- [9] L. Jaco & T. Verhanneman & B. De Win & F. Piessens & W. Joosen. Adaptable access control policies for medical information systems. *Departement Computerwetenschappen, K.U.Leuven, Report CW 363*, Augustus 2003.
- [10] B. De Win & F. Piessens & W. Joosen & T. Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. *Workshop on the Application of Engineering Principles to System Security Design*, December 2002.
- [11] R. Pawlak. *La programmation par aspects interactionelle pour la construction d'applications à préoccupations multiples: Deuxième partie: La séparation des préoccupations: état de l'art*. PhD thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Paris, December 2002.
- [12] R. E. Filman & D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Oktober 2000.

-
- [13] N. Damianou & N. Dulay & E. Lupu & M. Sloman. The ponder policy specification language. *Workshop on Policies for Distributed Systems and Networks (Policy2001)*, HP Labs Bristol, Januari 2001.
- [14] C. Ribeiro & A. Zúquete & P. Ferreira & P. Guedes. Spl: An access control language for security policies with complex constraints. *Network and Distributed System Security Symposium (NDSS01)*, San Diego, California, Februari 2001.
- [15] T. Verhanneman & F. Piessens & B. De Win & W. Joosen. View connectors for the integration of domain specific access control. *AOSD 2004 Workshop*, Lancaster, UK, Maart 2004.
- [16] R. Pawlak. *La programmation par aspects interactionelle pour la construction d'applications à préoccupations multiples: Cinquième partie: An aspect-oriented general purpose environment implementation: Java Aspect Components*. PhD thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Paris, December 2002.
- [17] Java Aspect Components website. <http://jac.objectweb.org>.
- [18] E. Gamma & R. Helm & R. Johnson & J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.