Mobile Ubiquitous Computing

Tom Vanliefde

promotor :
Prof. dr. Karin CONINX

co-promotor :
Prof. dr. Kris LUYTEN

Eindverhandeling voorgedragen tot het bekomen van de graad
Master in de informatica human-computer interaction

Universiteit Maastricht

universiteit hasselt

Abstract

Tegenwoordig hebben de meeste mensen een heel scala aan apparaten waarop ze kunnen werken, gaande van een volledige desktop pc tot een gsm of PDA. Er is niet alleen een grote diversiteit aan apparaten, maar ook blijken de mensen vaak veel mobieler dan vroeger. Een zelfde persoon werkt niet alleen meer op het kantoor maar ook onderweg, bij een klant of thuis. Het zou dan ook handig zijn dat de applicaties waarmee zij werken dezelfde mobiliteit bezitten. Dit houd in dat een gebruiker zijn gewenste applicaties overal kan mee nemen, ongeacht het soort apparaat waarop deze terecht kan komen. Bij het verhuizen van een applicatie moet de staat ervan bewaard blijven. Dit zodat na de verhuizing de taak, waar men mee bezig was, verder gezet kan worden. Op deze wijze zal het apparaat waarop gewerkt wordt van minder belang worden. Het uitvoeren van een taak in een bepaalde applicatie zal nu centraal staan. Men kan zich nu meer focussen op het voltooien van een taak in een bepaalde applicatie, terwijl het apparaat waarmee dit gedaan wordt naar de achtergrond verdwijnt. Combineer dit feit met het mobiele karakter en we kunnen spreken van Mobile Ubiquitous Computing.

Om dit te kunnen bereiken zal onderzoek gedaan worden naar Mobiele Code en reeds bestaande toepassingen die hiervan gebruik maken, gaande van het simpele Client-Server paradigma tot de geavanceerde Mobile Agents. Eveneens wordt gekeken naar de veiligheidsaspecten bij het gebruik van Mobiele Code. Omdat een applicatie verplaatsen van het ene apparaat naar een ander meer inhoud dan enkel code verplaatsen, zullen we ook bestuderen wat het effect hiervan kan zijn op de User Interface. Ten slotte onderzoeken we in welke mate het .NET Framework geschikt is voor het ontwikkelen van Mobile Code applicaties en stellen we een zelf ontwikkelde toepassing voor die het mogelijk maakt om een applicatie te verplaatsen van apparaat naar apparaat met behoud van zijn staat.

Woord vooraf

Eerst en vooral wil ik iedereen danken die rechtstreeks en onrechtstreeks hebben bijgedragen tot het tot stand komen van deze thesis. Speciale dank gaat uit naar mijn ouders die het mogelijk maakte dat ik verder kon studeren, mijn vriendin zonder wie ik het niet zou hebben kunnen volhouden en Prof. dr. Kris Luyten voor voor zijn raad en de nodige tips.

Inhoudsopgave

Lijst van Codefragmenten

Lijst van Afbeeldingen

1 Inleiding

1.1 Inleiding

Tegenwoordig hebben de meeste mensen een heel scala aan apparaten waarop ze kunnen werken, gaande van een volledige desktop pc tot een gsm of PDA. Er is niet alleen een grote diversiteit aan apparaten, maar ook blijken de mensen vaak veel mobieler dan vroeger. Een zelfde persoon werkt niet alleen meer op het kantoor maar ook onderweg, bij een klant of thuis.

Op software gebied kan dit enkele problemen geven. De gebruiker dient op elk apparaat dat hij wil gebruiken, een versie te hebben van de nodige software. Daarbij moet bij elke verhuizing, naar een ander apparaat, gezorgd worden dat alle nodige data mee verhuist wordt. Dit opdat men verder zou kunnen werken vanwaar men gestopt was.

Een beter alternatief zou zijn dat de gebruiker simpelweg de gewenste applicatie kan “zenden” naar het nieuwe apparaat. Zodat hij direct verder kan werken zonder zich zorgen te hoeven maken over de aanwezige software en het overbrengen van de nodige data. Op deze manier kan men op verschillende locaties en apparaten blijven werken, zonder expliciet data te moeten overbrengen en de software te installeren.

Zoals men kan merken dient de installatie zo transparant mogelijk te verlopen, zodat de gebruiker zijn software enkel maar hoeft te “verhuizen”. In deze thesis zal er onderzocht worden in welke mate we dit mogelijk kunnen maken en welke technieken hiervoor al bestaan. Na het onderzoek zal getracht worden een Mobiel Applicatie Systeem te ontwikkelen, die de gebruiker kan helpen in het voorgaande scenario.

Het onderzoek van de thesis omvat verschillende delen. In de eerste plaats zal geprobeerd worden een beter beeld te vormen omtrent Mobiele Code (Mobile Code). Dit is een techniek waarbij het verhuizen van code tot werkende applicaties (Running Applications) centraal staat. Er zal gekeken worden welke Mobile Code paradigma's er bestaan, wat de voor- en nadelen hiervan zijn en welke soort toepassingen ervan gebruik maken. Vervolgens zal er onderzocht worden welke veiligheidsproblemen er bestaan bij Mobile Code, wat de gevaren hiervan zijn en welke maatregelen ertegen genomen kunnen worden.

Bij het verhuizen van een applicatie dient er niet enkel rekening gehouden te worden met de code, ook de User Interface is hierbij belangrijk. Een applicatie verhuizen van een desktop pc naar een PDA is niet zo evident als het lijkt. We zullen overlopen welke soort User Interface migraties er bestaan en welke soort mappingen toegepast kunnen worden.

Het onderzoek van het .NET Framework vormt het laatste deel van deze thesis. Hierbij wordt gekeken in welke mate deze geschikt is voor het ontwikkelen van een Mobiele Applicatie Systeem. Vervolgens zullen we de opgedane kennis trachten te gebruiken in de ontwikkeling van dit soort systemen. Tot slot wordt het ontwikkelde systeem vergeleken met een reeds bestaand Mobiele Applicatie Framework.

1.2 Use Cases

Hier proberen zal ik proberen enkele situaties te schetsen welke het nut van Mobiele Applicaties kan aantonen:

Reiziger	
Beschrijving	De reiziger zit momenteel nog te werken op zijn pc. Hij moet om 3 uur in Gent zijn en is van plan de trein van 2 uur te nemen zodat hij mooi op tijd is. Natuurlijk zou de reistijd op de trein verloren tijd zijn indien hij niet verder zou kunnen werken. Voor hij vertrekt zendt hij de applicatie waarin hij aan het werken is naar zijn PDA. Eenmaal in de trein kan hij rustig verder werken. Wanneer de nieuwe locatie bereikt wordt kan de applicatie weer van de PDA naar een desktop pc verzonden worden.
Actoren	Nomadische gebruiker
Conditions	● Elke locatie waarnaar de gebruiker wil verhuizen moet het Mobile Host Systeem bevatten. ● Na het verlaten van een Mobile Host Systeem moet elk spoor van de gebruiker hierop verdwenen zijn of op zijn minst onleesbaar voor derden.
Voordelen	De gewenste applicatie van de reiziger moet niet meer op elk systeem geïnstalleerd zijn, noch moet het expliciet geïnstalleerd worden bij aankomst. De gebruiker moet ook geen extra handelingen uitvoeren voor het verplaatsen van de staat van zijn werk.

Museum	
Beschrijving	Een museum zou verschillende tentoongestelde items kunnen taggen met Mobiele Code. Wanneer een bezoeker met zijn PDA in de buurt komt van deze tags kan hij de code ervan inladen en gebruiken voor het verdere verloop van de interactieve rondleiding in het museum. Wanneer het bezoek voltooid is, kan de gebruiker in de pc-kamer van het museum nog eens alles overlopen door zijn gevonden “items” naar een pc door te sturen.
Actoren	Museum bezoeker (PDA gebruiker)
Condities	● Op elke interactieve locatie dient Mobiele Code aangebracht te worden op zo een manier dat deze leesbaar is voor de PDA. ● De PDA's en pc's dienen het Mobile Host Systeem te bevatten
Voordelen	● Wanneer een bepaalde interactieve locatie verdwijnt moet de software op de vele PDA's niet aangepast worden, vermits de code zich bevind op het verwijderde item. Hetzelfde geldt ook wanneer het museum uitbreidt en er extra locaties bijkomen. ● Wanneer het museum heringericht wordt en dus de interactieve locaties van plaats veranderen moet eveneens de software op de PDA's niet opnieuw vernieuwd worden.

2 Mobiliteit van applicaties

Volgens Luca Cardelli & Andrew D. Gordon [1] kunnen we twee verschillende visies hebben op mobiliteit. Namelijk het uitvoeren van applicaties op mobiele apparaten zoals bijvoorbeeld PDA's en laptops of een applicatie die van het ene apparaat naar het andere kan verhuizen. Deze paper zal het vooral hebben over het tweede geval, ook wel Mobile Code genoemd.

3 Mobiele Code

Tegenwoordig zijn netwerken meer dan normaal. Steeds meer en meer worden deze met elkaar verbonden, wat betekent dat meer computers met elkaar in verbinding komen te staan. Kortom, er is geen plaats op de wereld waar je jezelf niet kunt verbinden met je thuis-netwerk. En niet alleen netwerken groeien, maar de mobiliteit van de gebruikers vergroot voortdurend. De dag van vandaag kan bijna elke gsm Java applicaties uitvoeren. Zeer veel mensen hebben een PDA of Smartphone en laptops verkopen beter dan desktop pc's[1]. Aan de hand van volgende voorbeelden zien we dat Mobiele Code reeds vaak gebruikt wordt zonder dat men dit echt beseft.

- PC's die niet krachtig genoeg zijn om een zware taak uit te voeren, kunnen verbinding maken met een krachtigere computer of groep van computers. Deze laat men dan de zware opdracht uitvoeren. Daarna dient de PC slechts het resultaat te verwerken.
- Websites worden steeds krachtiger. Ze lijken bijna op volwaardig desktop applicaties. Dit met behulp van Java Script en Applets die verkregen worden van de webserver waarop de website gehost is.
- Hulp applicaties kunnen zelfstandig op het netwerk naar informatie op zoek gaan, onderhandelen over prijzen, akkoorden sluiten, ... etc.
- Zakenmensen kunnen onderweg naar een klant informatie ophalen van het kantoor. Alsof dit de normaalste zaak van de wereld is!

Al deze zaken vormen een fenomeen wat Mobile Code wordt genoemd. Mobile Code kan meer precies gedefinieerd worden als “Software modules die worden verkregen van remote systemen, door middel van het versturen over een netwerk, en vervolgens gedownload en uitgevoerd op een lokaal systeem en zonder dat de gebruiker hiervoor extra handelingen voor hoeft uit te voeren. “ [2]

Zoals we merken, staat het transparant verzenden, ontvangen en installeren van Mobile Code hier centraal. Wanneer we naar het oorspronkelijk doel van deze paper kijken blijkt dit een zeer goede techniek om de oplossing voor ons probleem te verwezenlijken.

Mobile code wordt zeer vaak gebruikt. Maar omdat het transparant gebeurd voor de gebruiker, merken we hier vaak niet veel van. Java Scripts op een site zijn hier een vorm van. Sommige virussen kunnen zelfs als Mobile Code beschouwd worden vermits ze zonder dat de gebruiker het door heeft, verzonden, geïnstalleerd en uitgevoerd worden. Een wat uitgebreidere vorm van Mobile Code zien we weer in Grid Computing. We zullen nu de verschillende vormen van Mobiele Code bespreken.

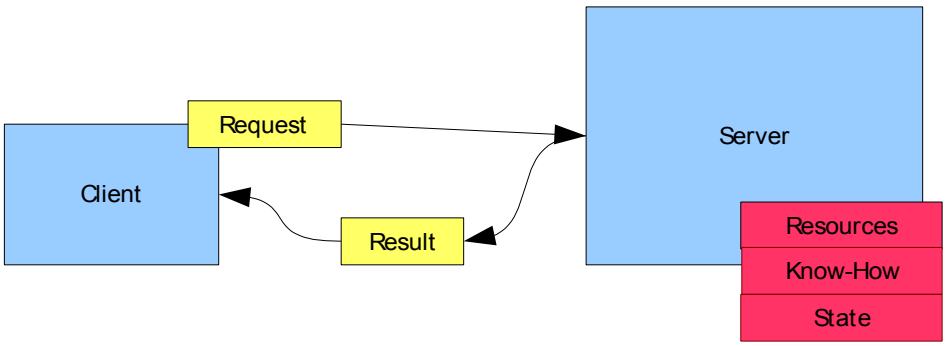
1 http://marketwatch-cnet.com.com/Notebooks+pass+desktops+in+U.S.+retail/2100-1044_3-6033967.html
2 Wikipedia: http://en.wikipedia.org/wiki/Mobile_code

3.1 Mobiele Code Paradigma's

Zoals eerder gezegd bestaan er verschillende soorten Mobile Code. Het is mogelijk deze soorten volgens hun werking op te delen. Deze opdeling kunnen we Mobile Code Paradigma's noemen. We kunnen volgende paradigma's onderscheiden: Client-Server, Remote Evaluation Code On Demand en Mobile Agents. Waarbij Client-Server als de simpelste beschouwd mag worden en het Mobile Agent Paradigma als de meest geavanceerde. We zullen nu dieper ingaan op elke paradigma.

3.1.1 Client-Server

Het Client-Server model [2] is één van de meest voorkomende Mobile Code paradigma's. Er moet opgemerkt worden, dat sommige deze niet tot de Mobile Code familie rekenen. Dit omdat er alleen een aanvraag verzonden wordt en er geen code lokaal uitgevoerd moet worden. Hierdoor wordt er niet voldaan aan de omschrijving van Mobile Code. Toch wil ik het vermelden vermits het toch de basis gevormd heeft voor de verdere ontwikkeling van Mobile Code.
Het Client-Server model houdt in dat een client graag een bepaald gegeven had gekregen, maar totaal niet weet hoe hij hieraan moet komen. Wel kent hij iemand die weet hoe je aan het gegeven kan komen en hiervoor ook de beschikbare resources heeft. De client zendt een aanvraag naar de server die het gevraagde dan zal berekenen en het resultaat terug zendt.
Een simpel voorbeeld is het synchroniseren van een klok. De client-pc wil graag zijn klok gelijk zetten volgens het huidige uur in Frankrijk. Maar de pc zelf weet niet hoe laat het in Frankrijk is. Gelukkig is er een server beschikbaar die op aanvraag het huidige uur van gelijk welk land op de wereld kan geven. De client-pc doet een aanvraag naar deze server voor het uur van Frankrijk en krijgt deze dan toegezonden.



Afbeelding 1: Schema Client-Server

Zoals afgebeeld op het schema blijven de resources (klok), know-how (tijdzones, ...) en state (data & execution state) op de server. Het enige dat zal rondreizen is de aanvraag en het resultaat.
Voorbeelden van technologieën die volgens deze paradigma werken zijn:

- **RMI**

  Voluit Remote Method Iinvocation, zoals al afgeleid kan worden uit de naam houdt het systeem in dat er een methode opgeroepen kan worden van een object die niet lokaal hoeft gedefinieerd te zijn. De methode wordt, na het oproepen ervan, uitgevoerd op de computer van het object die deze aanbiedt. Ten slotte wordt het

resultaat naar de aanvrager verzonden. Java en het .NET Framework bieden bijvoorbeeld elk deze technologie aan, respectievelijk Java RMI[3] en Remoting[4] genoemd.

- **SOAP**

  Is een programmeertaal overschrijdende standaard[5] waarmee een client-server service aangeboden kan worden. De communicatie gebeurd via een soort van XML. Soap omschrijft hoe deze XML opgesteld moet worden. De implementatie ervan ligt bij de ontwikkelaars zelf. Dit wordt vaak gebruikt bij webservices zoals het voorbeeld omtrent het gelijkzetten van de klok met behulp van een remote server.

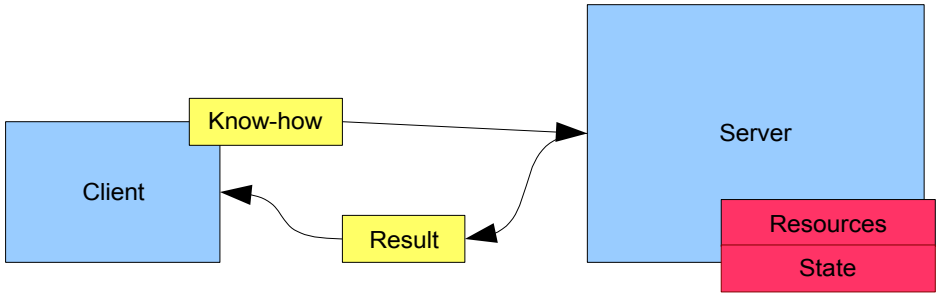
Er zal later terugkomen worden op Remoting en Soap, wanneer we dieper in zullen gaan op het .NET Framework en bij de implementatie van het Mobiele Applicatie Systeem.

3.1.2 Remote Evaluation

Bij Remote Evaluation [3] hebben we het omgekeerde van het client-server model. Hier weet de client wel hoe hij het moet doen, maar kan hij het niet uitvoeren door gebrek aan resources. Resources zijn hier een ruim begrip en kunnen verschillende zaken voorstellen:

- Hardware: printer, nodige rekenkracht, ...
- Softwarematig: zoals de nodige software voor het uitvoeren van scripts
- Data: de nodige data die niet lokaal beschikbaar is.

De client stuurt zijn opdracht onder de vorm van code (de know-how) door naar de server die deze uitvoert en het resultaat terug zendt.



Afbeelding 2: Schema Remote Evaluation

Op het schema is duidelijk zichtbaar dat er nu meer dan alleen een aanvraag doorgezonden wordt. De opdracht bestaat uit code die vertelt hoe de server de opdracht kan uitvoeren. Hier wordt dus effectief code verzonden van de client naar de server toe. Toepassingen die hiervan gebruik maken zijn:

- **Database Servers**

  Het is mogelijk dat men soms bepaalde data wil bekomen uit een verzameling gegevens op een externe server. Dit kan gebeuren door middel van SQL. Bij SQL weet een client wel hoe hij aan zijn gewenste gegevens kan komen alleen heeft hij de nodige resources niet, namelijk de databank waaruit de gegevens gehaald moeten worden. Hij stuurt dus zijn “code” naar de database server die de code

3 RMI: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
4 Remoting: http://msdn2.microsoft.com/en-us/library/ms973857.aspx
5 SOAP: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

uitvoert en een resultaat terug stuurt.

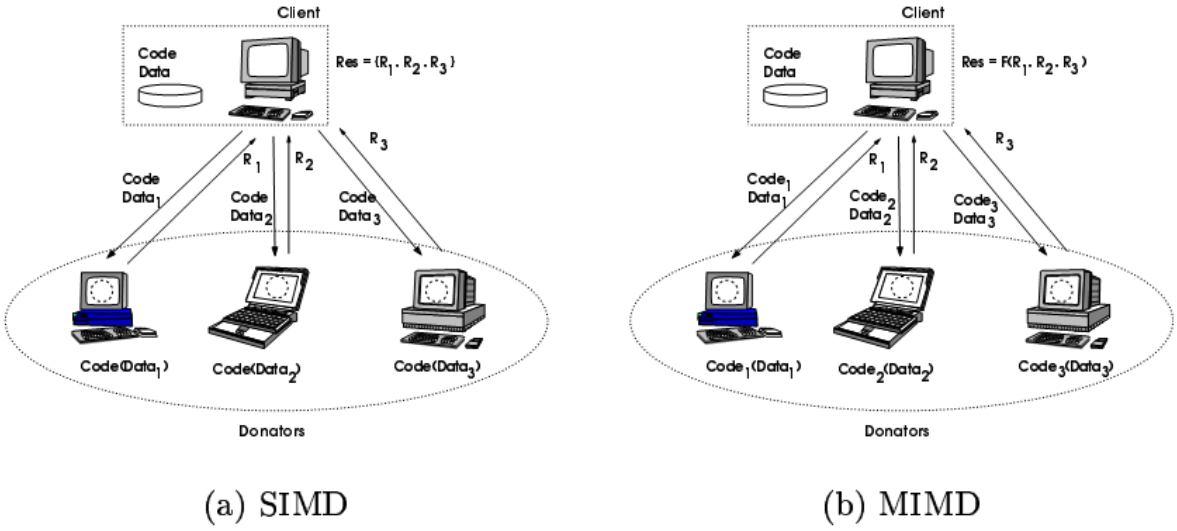
```
SELECT user_name, user_age
FROM tblUsers
WHERE user_Country = 'Belgium
```

Code 1: SQL-code voorbeeld

- **Grid Computing**

  Zoals eerder al vermeld valt gridcomputing onder de noemer van Mobile Code, meer specifiek onder Remote Evaluation. Dit wordt duidelijk wanneer we de werking van Grid Computing wat beter bekijken. Bij Grid Computing zal een client een opdracht sturen onder de vorm van code naar een Computer Grid. Het grid zal dan deze opdracht uitvoeren en vervolgens het resultaat terugzenden. Volgens [4] kunnen we twee manieren van werken bij grid computing onderscheiden:

  1. Er moet verschillende keren een zelfde taak uitgevoerd worden, maar met verschillende set van data. Het is hier dus makkelijk om de taak te verdelen in zoveel delen als dat er datasets zijn. Elke deeltaak wordt dan verzonden naar een node in het grid die deze uitvoert en het resultaat terug zendt.
  2. Een grote taak wordt opgedeeld in verschillende deeltaken. Deze kunnen die parallel met elkaar kunnen lopen. Elke subtaak kan dan uitgevoerd worden op een verschillende node waarna de resultaten weer gecombineerd worden.



(a) SIMD (b) MIMD

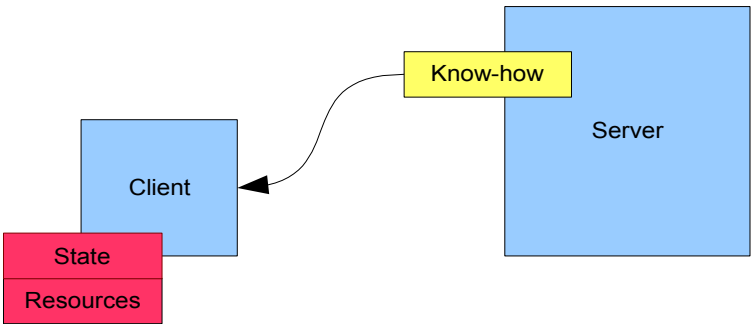
Afbeelding 3: Schema soorten Grid Computing volgens [4]

Aan de hand van het schema is duidelijk te zien dat er zowel code als data naar de verschillende nodes in het grid gezonden wordt. Dit maakt duidelijk maakt dat Grid Computing gebruik maakt van een vorm van Mobiele Code.
Er bestaat nog een speciaal fenomeen bij Grid Computing, soms krijgt een node teveel taken te verwerken terwijl anderen niks te verwerken hebben. Op dat moment kan het grid beslissen om taken van een druk bezette node te verplaatsen naar een werkloze node. Dit wordt load balancing genoemd. Het houdt in dat een taak gepauzeerd, verzonden en ingeladen wordt op een andere node. We zullen later zien dat dit de meest geavanceerde vorm van Mobiele Code is.

3.1.3 Code on Demand

Het kan voorkomen dat men alle resources beschikbaar heeft die nodig zijn om een bepaalde taak uit te voeren, maar men beschikt niet over de nodige kennis. Hoe kan men dan de taak toch nog uitvoeren? Er kan op zoek gegaan worden naar iemand die wel de nodige kennis bezit. Aan deze vraagt men dan of hij de nodige uitleg wilt verschaffen zodat we toch nog de taak zelf kunnen uitvoeren. Dit is wat bij Code on Demand gebeurd.

Bij Code on Demand [5] bezit een computer alle nodige resources gaande van rekenkracht tot de nodige data. Wanneer deze iets wil uitvoeren, waarvan hij niet weet hoe hij dit moet doen, vraagt hij de nodige code bij iemand die het wel weet. Hier zit het grote verschil met Remote Evaluation. Daar zegt de client “Ik weet het niet en ik wil het ook niet weten, stuur mij enkel het resultaat”. Bij Code on Demand zal de know-how onder de vorm van code gestuurd worden. De client zelf zal dit dan uitvoeren om zo aan een resultaat te komen. Dit kan gebruikt worden wanneer de clients in een systeem krachtig genoeg zijn om bepaalde taken uit te voeren en men de server wil ontlasten. In de plaats van de server alle berekeningen te laten doen, zoals bij Remote Evaluation, wordt dit nu op de clients zelf gedaan.



Afbeelding 4: Schema Code on Demand

Code on Demand zit in vele soorten toepassingen verwerkt. Maar de bekendste is toch de webbrowser. Zonder dat we hiervoor expliciet extra handelingen moeten uitvoeren, worden daarop verschillende soorten Mobiele Code ontvangen, geïnstalleerd en uitgevoerd. Voorbeelden van van deze soort paradigma zijn:

- **Javascript**

  Via javascript[6] kunnen we de functionaliteit van een site wat uitbreiden en het gebruiksgemak verhogen. Ajax maakt bijvoorbeeld gebruik van javascript.

- **Java Applets**

  Java applets[7] worden geprogrammeerd met behulp van Java. Bij Java wordt de applicatie gecompileerd naar bytecode die daarna door een Java Virtual Machine uitgevoerd wordt. Omdat we nu de kracht van Java ter beschikking hebben kunnen we echte applicaties in de webbrowsers laten uitvoeren. Het zenden, installeren en uitvoeren van de applets gebeurd volledig transparant.

6 Javascript: http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html
7 Java Applets: http://java.sun.com/applets/

- **ActiveX Controls**

  Is een Microsoft technologie[8] gelijkende op Java Applets waarbij een gecompileerde executable via een website gedownload en uitgevoerd wordt. Ook hier weer gebeurt alles volledig transparant naar de gebruiker toe. Het grote verschil met Java Applets is dat ActiveX Controls enkel op het Windows besturingssysteem werken. Ook hebben ze volledige toegang tot het ganse besturingssysteem en draaien ze niet in een Virtuele Machine met beperkte rechten.

- **Flash**

  Flash[9] was in het begin vooral gericht op het afbeelden van animaties. Maar tegenwoordig bevat flash een programmeertaal genaamd “ActionScript“[10] waarmee volledige applicaties ontworpen kunnen worden. Bij flash wordt er een flash bestand doorgezonden naar de browser die deze dan via een flash plugin kan uitvoeren. De plugin zelf vormt een Virtuele Machine waarin de flash code uitgevoerd kan worden.

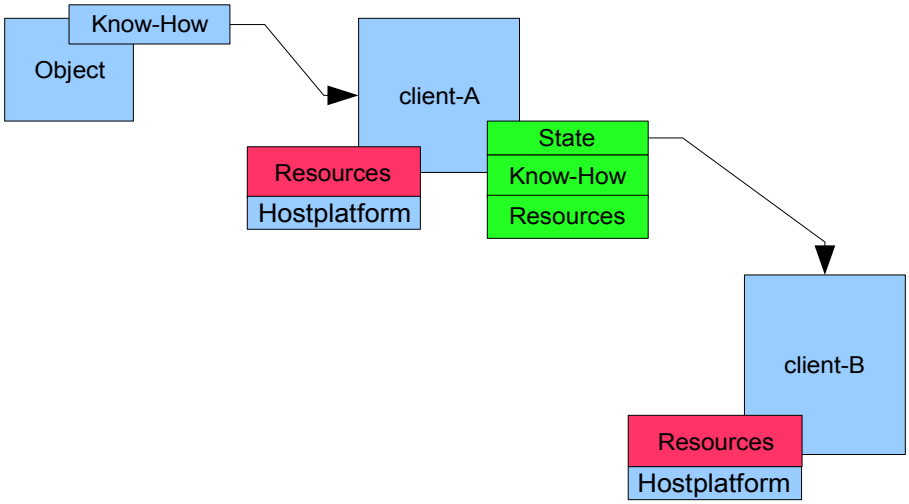
3.1.4 Mobile Agent

Mobile Agents, zoals besproken in [5], zijn een al wat meer geavanceerde vorm van Mobile Code. Een mobile agent bestaat uit een stuk software dat zichzelf kan verplaatsen, van het ene apparaat naar een ander, waarbij hij zijn state bewaart. Dit zodat wanneer hij eenmaal aangekomen is op het nieuwe apparaat, zijn werk verder kan zetten. Een mobile agent kan zelf beslissen of hij zal verhuizen of niet. Er zijn vele redenen waarom een Mobile Agent tot deze actie kan over gaan, enkele voorbeelden hiervan zijn:

- resources die niet beschikbaar zijn op de huidige host
- simpelweg klaar zijn met zijn taak
- ...

Een mobile agent kan zijn werk volledig zelfstandig uitvoeren. Het gaat zelfs zover dat de opdrachtgever kan verdwijnen van het netwerk waarop de mobile agent actief is. Wanneer deze dan later terug online komt, kan de agent de resultaten geven. De opdrachtgever maakt bijvoorbeeld een mobile agent aan op zijn PDA, geeft deze een opdracht en sluit zijn PDA af. Na enkele uren meldt hij zich weer aan op het netwerk vanop zijn desktop computer. Daar krijgt hij onmiddellijk de resultaten van de mobile agent die klaar is met zijn werk.

8 ActiveX: http://msdn2.microsoft.com/en-us/library/Aa751972.aspx
9 Flash: http://en.wikipedia.org/wiki/Adobe_Flash
10 Action Script: http://en.wikipedia.org/wiki/ActionScript



Afbeelding 5: Schema Mobile Agents

Het is duidelijk dat elk apparaat waar de mobile agent naartoe zou willen reizen een hostplatform moet aanbieden waarop de mobile agent zichzelf kan laten uitvoeren. Dit hostplatform moet zo gemaakt zijn dat hij zonder problemen meerdere mobile agents kan hosten, zonder dat deze elkaar kunnen beïnvloeden. Fouten in een bepaalde mobile agent die tot een crash leiden mogen niet de andere mobile agents storen in hun werk.
Er bestaan ontelbaar veel mobile agent systemen, enkele daarvan zijn:

- **Aglets**

  Ontworpen door IBM was één van de eerste Mobile Agent Systemen. Dit is nu beschikbaar als opensource project op SourceForge. In *[6]* en *[7]* vinden we hier meer informatie over. Kort samen gevat is het Aglet systeem geen echte applicatie maar een framework waarmee Mobile Agent Systemen opgebouwd kunnen worden.

- **MagNET**

  In [8] lezen we dat Magnet een Mobile Agent Systeem is, gericht op elektronische handel ontwikkeld met behulp van Aglets. In MagNET maken kopers mobile agents aan en verzenden ze deze naar verschillende handelaren. De mobile agents gaan dan autonoom onderhandelen over de prijs en leveringen. Wanneer ze klaar zijn met hun werk keren ze terug naar de opdrachtgever die dan de beste deals, gevonden door de mobile agents, kan bekijken.

- **Monad**

  Volgens [9] profileert Monad zich niet echt als een Mobile Agent Systeem. Maar het maakt gebruik van deze technologie om zijn doel te verwezenlijken. Het wordt gebruikt voor kleine gemeenschappen die artefacten willen uitwisselen zoals foto's, tekeningen, teksten, ... Gebruikers kunnen inloggen op de gemeenschap vanuit verschillende locaties en in het bezit zijn van meerdere apparaten. Nomad wordt gebruik om bijvoorbeeld te controleren of er al nieuwe versies op het netwerk zijn van bepaalde artefacten. Hiervoor worden er agents aangemaakt die het netwerk afzoeken en dan terugkeren naar de opdrachtgever met het resultaat.

3.2 Categorieën in Mobiele Code

De mobiliteit van Mobiele Code kan,volgens [5], grofweg in twee categorieën verdeeld worden. Wanneer we de criteria wat verstrengen kunnen we zelfs drie categorieën onderscheiden. De opdeling gebeurd volgens drie belangrijke factoren:

- De Code
  Deze kan komen van een scripting taal zoals Perl, het kan byte code zijn van Java of .NET applicatie of machine taal komende van een gecompileerd C++ programma. Deze code vormt de eigenlijke applicatie in zijn initiële staat.

- De Data State
  De staat van alle variabelen.

- De Execution state
  De execution state omvat alle data die bij het proces hoort. Hoe moeilijk of makkelijk men de execution state kan bekomen is afhankelijk van de gebruikte programmeertaal. Bij de ene programmeertaal kan men de execution state makkelijk bekomen, terwijl andere talen hier een probleem mee hebben. Dit is vooral het geval bij bijvoorbeeld .NET en Java. Dit omdat de Virtuele Machine ons afschermt van deze data. Het gaat hier dan vooral over de “call stack” en “instruction pointer”.

Sommige applicaties gebruiken het systeem van Mobile Code om deze na een crash weer hun oorspronkelijke staat te kunnen herstellen. Bijvoorbeeld met behulp van checkpointing[11]. Het spreekt voor zich dat de applicatie niet moet hersteld worden op dezelfde computer, maar dat dit gerust kan gebeuren op een andere. Dit bijvoorbeeld in het geval van een hardware fout waarbij herstel op dezelfde computer onmogelijk is.

Aan de hand van de besproken factoren kunnen we drie categorieën onderscheiden: Very Weak, Weak en Strong. We zullen nu dieper ingaan op elke categorie.

3.2.1 Very Weak

Wanneer er gesproken wordt over de categorieën van Mobile Code, zal men vaak lezen over Weak en Strong Mobility, [5] behandeld beide uitgebreid. Omdat bevonden werd dat de term Weak Mobility een te ruime betekenis heeft, zullen we een nieuwe term introduceren namelijk Very Weak Mobility.

Bij Very Weak Mobility wordt enkel de code verzonden. Deze vormen zijn Code on Demand en Remote Evaluation. De code kan bijvoorbeeld bestaan uit SQL-instructies of Java Byte code.

3.2.2 Weak

Wanneer we de algemeen gebruikte categorieën van Mobiele Code bekijken, houd Weak Mobility in dat enkel de code en eventueel ook de data staat verzonden wordt. Dit kunnen we terugvinden in[5]. Zoals eerder vermeld, leek dit een te ruime omschrijving. Vandaar de extra categorie Very Weak Mobility. Door deze extra categorie, werden de criteria voor Weak Mobility wat verstrengd. Enkel wanneer de code en data staat samen verzonden worden, spreken we van Weak Mobility.

11 Wikipedia: http://en.wikipedia.org/wiki/Application_checkpointing

Weak Mobility houd in dat we zeker moeten zijn dat alle huidige threads met hun werk klaar zijn, voordat we de staat willen vastleggen. De programmeur zorgt voor het opvangen van deze taak. Deze manier is de simpelste en wordt dan ook het meeste toegepast.

Weak mobility werkt vaak volgens een vast patroon zoals beschreven in [10] en is algemeen toepasbaar op verschillende soorten Mobile Code van mobiele applicaties tot mobile agents.

1. De uitvoering wordt gestopt
2. De data staat wordt verzameld, vaak door het te verhuizen proces zelf.
3. De code en de data staat worden naar de bestemming verzonden.
4. Code en data staat worden ingeladen en hersteld.
5. De uitvoering wordt weer gestart.

3.2.3 Strong

Bij strong migration [5] worden de code, data en execution state gemigreerd. Dit houdt in dat een proces echt gepauzeerd en bewaard moet worden om daarna weer verder te kunnen gaan op een ander apparaat. Dit op zo een manier dat het lopende proces hier niks van merkt. Vandaar dat het ook vaak de term Transparant Migration gebruikt wordt bij Strong Mobility.

Strong mobility wordt gebruikt bij grid computing, namelijk wanneer er bijvoorbeeld aan load balancing gedaan wordt. Zoals eerder vermeld moet het proces dan verhuizen van de ene node naar een andere. Eenmaal aangekomen op de nieuwe node moet het proces verder gezet worden alsof deze nooit werd verhuisd.
Strong mobility is niet echt makkelijk te bereiken. Indien het mogelijk was een thread te serializeren en deserializeren, was het probleem opgelost. Spijtig genoeg is dit niet zomaar mogelijk bij het .NET Framework welke gebruikt zal worden bij de implementatie van de voorbeeld-applicatie. Er zal hier later op terug gekomen worden.

Wanneer we zelf toegang zouden hebben tot de program counter en de call stack, kunnen we deze zelf opslaan. Dit op zo een manier dat we het later weer kunnen inlezen en laten uitvoeren. Ook dit is weer niet altijd mogelijk. De Virtuele Machine van .NET en Java laten dit bijvoorbeeld niet toe. Door de virtuele machine zo uit te breiden, zodat er toch voldoende toegang tot de benodigde gegevens verkregen kan worden, kan dit probleem opgelost worden. Een voorbeeld hiervan is Sumatra [11], een uitbreiding van de Java Virutal machine. Het nadeel hiervan is dat elke keer er een nieuwe versie van de Virtuele Machine uitkomt, deze opnieuw moet worden aangepast. Een bijkomend nadeel is dat vanaf nu elke host de aangepaste Virtuele Machine moet draaien, indien we gebruik willen maken van de extra ingebouwde functies. Dit betekend dat het niet evident is om gelijk waar de software, die hiervan gebruikt maakt, uit te voeren. In [12] zien we dat sommige zelfs zover gaan dat ze niet de Virtuele Machine aanpassen, maar hun eigen interpreter schrijven.

[13] stelt een manier voor waarop we de execution state van een applicatie toch, zonder de virtual machine te moeten aanpassen of te rekenen op thread serializatie, kunnen bewaren. Ze werken met preprocessing waarbij de broncode van een applicatie overlopen wordt en er extra code aan toegevoegd wordt. Deze extra code zorgt ervoor dat er checkpoints gegenereerd worden. Op die manier maken ze het mogelijk om de staat van

de applicatie (data en execution state) te bewaren en terug in te laden. Dit lijkt een mooie oplossing maar is zeker niet simpel te realiseren. Het toevoegen van deze instructies is niet alleen moeilijk, maar ze zorgen ook voor extra overhead. De extra instructies betekenen extra executietijd.

Very Weak	Weak	Strong
Enkel de Code wordt verzonden, hierdoor hoeft geen rekening gehouden te worden met de applicatie staat.	De applicatie zelf dient alles goed af te handelen. Deze staat dus in voor het bewaren en laden van de applicatie staat. Er moet gezorgd worden dat alle threads klaar zijn met hun werk, dan pas mag de Mobiele Applicatie verzonden worden.	Processen worden op een transparante manier gestopt. Een proces kan op elk moment gepauzeerd en verzonden worden naar een andere client. Daar, waar het gestopt is, gaat het proces weer verder. Alles gebeurt transparant voor de applicatie/proces die verplaatst gaat worden. Deze hoeft dan ook niks zelf hiervoor te doen

Wanneer Mobile Code toepassingen in .NET of Java geschreven worden, is het realiseren van Strong Mobilty niet makkelijk. Dit doordat de Execution State door de Virtuele Machine van de programmeur afgeschermd wordt. Hiervoor bestaan er verschillende oplossingen, zoals eerder vermeld, maar deze zijn van een zekere complexiteit waardoor ze niet vaak gebruikt worden. Dit is dan ook de reden waarom bijna altijd voor Weak Mobility gekozen wordt (bij Mobile Code systemen geprogrammeerd in .NET of Java). Voor deze reden zal de implementatie, waar we later dieper op zullen ingaan, van het Mobiele Applicatie Systeem dan ook slechts Weak Mobility ondersteunen.

3.3 Resources

Bij Mobile Code moet niet enkel rekening gehouden worden met code, data en execution state. Er zijn ook nog de resources die al dan niet mee verzonden kunnen worden met de Mobile Code. Deze kunnen bestaan uit prenten, teksten, data, ... Er bestaan ook resources die zich bevinden op de host apparaten en dus niet zomaar verzonden kunnen worden. Volgens [14] kunnen we deze resources dan ook opdelen in drie categorieën.

- **Vrij verzendbare resources**

  Kunnen vrij mee verzonden worden met de Mobile Code en zijn staat.

- **Onwijzigbare / Onverplaatsbare resources**

  Zouden eventueel verzonden kunnen worden, maar bevinden zich in een staat die verzending niet toelaat. Voorbeelden hiervan zijn open files, veel te grote files, ...

- **Onverzendbare vaste resources**

  Resources die onmogelijk verzonden kunnen worden. CPU tijd, printers, ...

Wanneer Mobile Code verzonden wordt, moet er dus rekening gehouden worden met welke resources we mee kunnen zenden en welke niet. Het mooiste zou zijn als we de

verzendbare resources samen met de Mobile Code zouden kunnen verpakken. Sommige talen zoals Java met zijn Jar's en .NET met Assemblies ondersteunen dit. Op .NET Assemblies komen we later nog terug, wanneer het .NET Framework uitvoeriger besproken wordt.

4 Veiligheidsaspecten

De laatste jaren is de eis naar meer aandacht voor veiligheid in de IT wereld steeds maar gestegen. Computers zijn niet meer uit het dagelijkse leven te denken en worden dan ook voor enorm veel zaken gebruikt. Waar men vroeger de normale inbrekers had die bedrijfsgeheimen probeerden te stelen uit een kluis in het bedrijf, zijn er nu de IT-criminelen die computersystemen proberen binnen te dringen om hiervan gevoelige informatie te stelen. Een veilige computer is niet alleen belangrijk voor bedrijven maar ook voor de consument die zijn computer steeds meer en meer voor gevoelige zaken gaat gebruiken. Een voorbeeld hiervan is e-banking. Wanneer geldzaken via het internet geregeld worden, is een niet gecorrupteerde computer toch wel een noodzaak.

Beveiliging in de IT sector is een zeer ruim begrip. Voor sommige betekent veiligheid dat men de fysieke toegang tot computers gaat beperken. Anderen maken zich meer zorgen over de integriteit van de data op hun computersysteem. Soms wilt men de toegang tot data beperken tot een select groepje mensen, die hiervoor al dan niet moeten betalen. Sommige bedrijven verhuren hun hardware in de zin van computertijd, netwerkcapaciteit, ... waardoor men wil weten wie een bepaalde opdracht gegeven heeft. Beveiliging houdt dus grofweg in dat men gebruikers wil kunnen identificeren, data afschermen en de integriteit van applicaties en data kan waarborgen.

Door de jaren heen zijn de kosten van malafide code steeds meer gaan stijgen en worden aanvallen steeds frequenter. In 2001 bijvoorbeeld schat men de wereldwijde economische kost van malafide code op zo'n 13,2 miljard dollar[12].

4.1 Gevaren

Alles komt er dus op neer dat een computer geen malafide software mag staan hebben. Dit is nu net het probleem van Mobiele Code. De definitie van Mobiele Code zegt dat er code gedownload, geïnstalleerd en uitgevoerd wordt zonder dat hiervoor extra handelingen dienen vereist voor zijn. Maar niemand kan ons verzekeren dat deze code veilig is. Vermits Mobiele Code ook echt uitgevoerd wordt op de computer waar het naartoe verzonden werd, brengt dit een ernstig veiligheidsrisico met zich mee. Bij Very wak mobility is dit risico zeer laag, vermits hier geen echte code uitgewisseld wordt. Bij weak en strong mobility ligt dit al wat moeilijker. Hierbij zal er wel code uitgewisseld worden.

Bij Mobiele Code systemen kunnen we twee probleem domeinen onderscheiden [15], namelijk de Mobiele Code en de host. De host moet beschermd worden tegen malafide code en omgekeerd dient de Mobiele Code beschermd te worden tegen een malafide host. We zullen eerst bespreken welke gevaren er zijn voor de host.

12 http://www.computereconomics.com/article.cfm?id=133

4.1.1 Gevaren voor de Host

Denial of service

Vermits mobile code inhoud dat vreemde code uitgevoerd wordt kan het gebeuren dat een Mobiele Applicatie alle systeem resources opgebruikt waardoor het hostplatform trager gaat werken of zelfs op zijn knieën kan gaan. Wanneer dit opzettelijk gebeurt, dan spreken we van een Denial of Service. In [16] vinden we dit specifiek probleem terug.

Masquerading

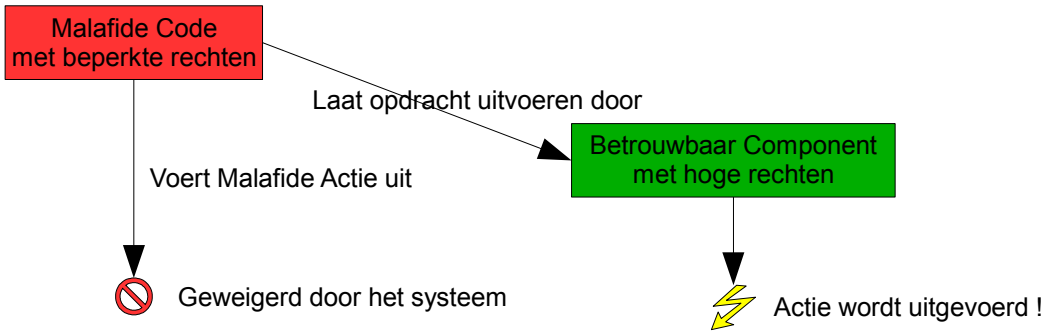
Dit probleem wordt besproken in [16] en [17]. Het gevaar voor de host zit hem hier in code die zich probeert voor te doen als code komende van een vertrouwde partij. Dit kan verschillende gevolgen hebben:

- Er kan geprobeerd worden de acties uit te voeren in naam van iemand anders. We mogen niet vergeten dat bijvoorbeeld Mobile Agents gebruikt worden om automatisch verkoopovereenkomsten te sluiten. Wanneer een Agent werkt onder een valse naam kan deze de originele eigenaar in problemen brengen. De agent zou bijvoorbeeld grote hoeveelheden kunnen inkopen of zaken zeer goedkoop verkopen.

- Meer rechten proberen te krijgen:
  Niet elke Mobiele Applicatie mag evenveel op een hostsysteem. Soms zullen host bepaalde mobiele applicaties meer vertrouwen dan anderen en daarom deze ook meer rechten geven. De mate van vertrouwen kan gegeven worden op basis van de auteur van de code, de bron, ... etc. Wanneer een Mobiele Applicatie werkt onder een valse identiteit kan deze meer rechten krijgen. Dit geeft hem de mogelijkheid om makkelijker over te gaan tot aanvallen zoals disclosure of information of corruption of information

Luring Attacks

Bij een “Luring Attack” probeert malafide code met beperkte uitvoerrechten, code met hoge uitvoerrechten te “overtuigen” bepaalde handelingen op een host uit te voeren. Doordat de code met hoge uitvoerrechten meer toegang tot het systeem van de host heeft, zal deze de opdracht ongehinderd kunnen uitvoeren.



[18] bespreekt dit probleem. Deze type aanval wordt expliciet aangepakt bij het .NET Framework. Wanneer we het zullen hebben over “Stack Walking” zal hier dieper op in gegaan worden.

Disclosure and Corruption of information

Bij Disclosure of Information wordt geprobeerd om toegang te verkrijgen tot afgeschermde informatie. Om dit te bereiken wordt vaak gebruik gemaakt van reeds besproken technieken zoals Masquerading en Luring Attacks. Soms is de aanvaller er niet enkel op uit om data te vergaderen, maar ook om deze aan te passen of te vernietigen. Wanneer dit het geval is, dan spreken we van Corruption of Information. We kunnen een uitgebreide bespreking van beide problemen eveneens terugvinden in [16].

4.1.2 Gevaren voor de Mobiele Applicatie

Alteration

In [19] bespreekt men het probleem van Alteration. Zoals eerder vermeld, kan het gebeuren dat mobile code verhuist van systeem tot systeem. Elk systeem waarop de Mobiele Code huist, heeft in feite volledige toegang tot alle code en data van de Mobiele Applicatie. Een host systeem zou bijvoorbeeld de code van de Mobiele Applicatie kunnen veranderen zodat deze op latere host systemen een aanval uitvoert. De integriteit van de Mobiele Applicatie moet dus gecontroleerd kunnen worden zodat veranderingen worden opgemerkt.

Er moet vermeld worden dat een Mobiele Applicatie aan nog meer gevaren blootgesteld wordt, maar deze zijn vaak gericht op Mobiele Agenten. In [19] vinden we hier meer over. Deze thesis richt zich vooral op het algemeen gebruik van Mobile Code. Hierdoor zullen we er dan ook niet meer verder op in gaan.

4.2 Maatregelen

In dit deel bespreken we hoe we het vernietigen, wijzigen en doorzenden van gegevens tegengaan, hoe we misbruik van de host voorkomen en masquerading onmogelijk kunnen maken. Hiervoor bestaan verschillende technieken. Wanneer we deze technieken toepassen, kunnen we een veilig geheel creëren waarin Mobiele Applicaties zonder al te veel risico's uitgevoerd kunnen worden.

4.2.1 Access control

Hier proberen we de host tegen Mobiele Code te beschermen. Dit kan door deze te beperking in zijn mogelijkheden. Aan de hand van bepaalde eigenschappen van de Mobiele Code kan beslist worden in welke mate we de code beperkingen gaan opleggen. Enkele eigenschappen zijn: wie de auteur was, vanwaar de code komt,... . Hierover wordt er meer uitgelegd wanneer we het over Code Access Security zullen hebben.

4.2.2 Analyseren van de werking

Men kan de werking van Mobiele Code gaan analyseren. Dit om na te gaan of deze verdacht gedrag vertoont. Aan de hand hiervan stoppen we de werking ervan. Hiervoor bestaan er verschillende methoden. Maar allen zijn verre van perfect. Toch vormen ze een extra laag van verdediging voor de host tegen malafide code.

Scannen van de code

De code van een applicatie wordt, vóór uitvoering of compilering, gecontroleerd. Hierbij wordt de code van de applicatie overlopen en nagezien of er enige verdachte opdrachten

in gedefinieerd staan. In [20] wordt een C en C++ code analysator besproken en in [21] zien we hoe men datamining technieken toepast om malafide applicaties te herkennen.

Actief Scannen van werking
Tijdens de uitvoering van de code wordt er gecontroleerd of deze code geen verdachte handelingen wil uitvoeren. Deze zouden namelijk een gevaar kunnen betekenen voor het systeem. Wanneer dit opgemerkt wordt, kan de uitvoering van de actie geweigerd worden. Virusscanners[13] maken gebruik van deze methode voor de ontdekking van nog onbekende virussen.

Geschiedenis gebaseerd
In [22] bespreekt men een geschiedenis gebaseerde methode. Deze methode houdt in dat elke request van een Mobiele Applicatie, tijdens de uitvoering ervan, bijgehouden wordt. Op basis van deze geschiedenis kunnen we beslissen om bepaalde acties al dan niet toe te laten. Bijvoorbeeld: een Mobiele Applicatie kan het recht krijgen tot inlezen van data als deze voordien nog geen socket geopend heeft. Omgekeerd geldt dan dat een socket openen alleen mag als men nog geen data ingelezen heeft. Op deze manier kunnen we een applicatie toch nog bepaalde acties toelaten zonder dat deze misschien vertrouwelijke informatie verspreidt.

Er kan opgemerkt worden dat bij de drie methoden alles berust op aannames, het is mogelijk dat de code iets kwaadaardig wilt uitvoeren, maar het is ook mogelijk dat het hier om een onschuldige applicatie gaat waarbij zijn verdachte handelingen een normale gang van zaken zijn. Wanneer zo'n legitieme applicatie wegens de uitvoering van zo'n verdachte handeling gestopt wordt, spreken we van een “False Positive”. Het grote probleem zit hem in het vinden van de goede balans tussen hoe streng men moet reageren (zodat malafide code geen kans krijgt) en hoe te voorkomen dat legitieme applicaties gestopt worden in hun werking.

4.2.3 Authenticatie & Integriteit

Wanneer we over de afkomst van Mobile Code zeker willen zijn, kunnen we deze ondertekenen. De auteur van een Mobiele Applicatie kan zijn Mobiele Applicatie digitaal onderteken. Hiermee is het mogelijk de afkomst van de code te bepalen en kunnen we eveneens controleren of de code ongewijzigd is gebleven. Hierdoor beschermen we de host tegen masquerading en de agent tegen alteration. Er zal hier nog op teruggekomen worden wanneer het .NET Framework besproken wordt.

4.2.4 Save Execution

- Sandboxing
  Hierbij zorgen we dat de Mobiele Applicatie in een afgesloten ruimte draait. Dit heeft als voordeel dat de applicatie geen externe resources kan aanspreken zonder expliciete toestemming van de host. Ook worden andere processen van de Mobiele Applicatie op de host afgeschermd. De sandbox[14] kan er ook voor zorgen dat een fout in één bepaalde Mobiele Applicatie geen effect heeft op de rest van het systeem. [16] en [17] halen deze techniek aan.

  Bij het onderzoek van het .NET Framework zullen hier verder op in gaan wanneer we Applicatie Domeinen gaan bespreken..

13 http://en.wikipedia.org/wiki/Antivirus_software#Dictionary
14 http://en.wikipedia.org/wiki/Sandbox_(computer_security)

- Proof Carrying Code
  Is een techniek waarbij een host kan controleren of Mobile Code kan voldoen aan een set van voorgedefinieerde regels. Wanneer hieraan voldaan wordt, kan men met zekerheid zeggen dat de applicatie geen gevaar zal opleveren voor het host-systeem. De controle kan gebeuren aan de hand van een attest geleverd door de Mobiele Applicatie. [23] bespreekt algemeen Proof Carring Code en in [24] vinden we terug hoe deze nuttig kan zijn in Mobiele Code Systemen.

4.2.5 Encryptie

Een efficiënte maatregel tegen “Disclosure of Information” is het encrypteren van de informatie. Zo voorkomen we dat, wanneer onbevoegden toch toegang krijgen tot bepaalde data, deze niet zomaar leesbaar is.

Wanneer de communicatie niet afgeluisterd mag worden, kan gebruik gemaakt worden van encryptie. Wanneer de communicatie geëncrypteerd wordt, is de kans er dat een “Earsdropper” iets kan doen met de opgevangen informatie miniem tot niet-bestaand.

Wanneer de staat van een Mobiele Applicatie vertrouwelijke informatie bevat, mag deze ook niet simpelweg opgeslagen worden onder een vorm die voor iedereen leesbaar is. Wanneer hier ook encryptie gebruikt wordt, voorkomt men dat de vertrouwelijke informatie in verkeerde handen terecht kan komen.

5 Mobiele User Interface

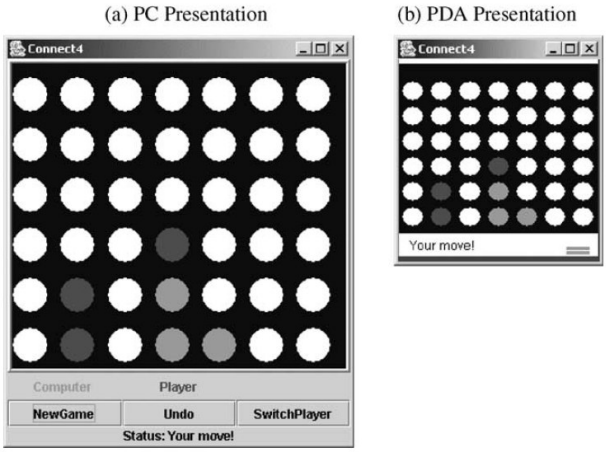
Het verplaatsen van code voor een Mobiele Applicatie is niet het enige waar rekening mee gehouden moet worden. Een applicatie die van apparaat naar apparaat verhuist zal ook rekening moeten houden met zijn User Interface. Niet elk apparaat bied dezelfde in- en uitvoer mogelijkheden. Een PDA heeft bijvoorbeeld een veel kleiner scherm ter beschikking dan een desktop pc. Dit betekend dat, om de applicatie bruikbaar te houden, soms een enkel scherm opgedeeld dient te worden in verschillende kleinere schermen. Het is zelf mogelijk dat er geen beeldscherm aanwezig is en er gebruik moet gemaakt worden van een Voice Interface [25]. Hierbij zal een grafische interface moet gemapt worden naar een spraak gerichte interface.

We zullen nu een overzicht geven van enkele taxonomieën omtrent de soorten mappingen en migraties die er bestaan.

5.1 Migratie Types

Men kan verschillende soorten migraties onderscheiden. We kunnen deze verschillende types classificeren op basis van hoeveel en welk gedeelte van de interface verhuist wordt. In [26], [27], [28] en [29] kunnen we volgende verdeling vinden:

- Totale Migratie
  Hierbij wordt de volledige interface van de applicatie en alle code verhuist naar het nieuwe apparaat. We zullen hier later twee voorbeelden van bespreken, namelijk het eigen ontwikkeld Mobiel Applicatie Systeem en Roam [28].

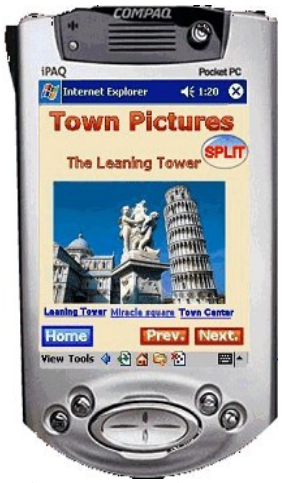


Afbeelding 6: Voorbeeld totale Migratie in Roam

Hierbij is de grootste zorg het aanpassen van de User Interface naargelang het apparaat. Hiervoor zal een mapping moeten plaatsvinden. Welke mappingen mogelijk zijn zullen we later bespreken.

- Gedeeltelijke Migratie
  Ook wel Controle Migratie genoemd. De Interface wordt verdeeld in twee delen. Het ene deel wordt gebruikt voor het besturen van de applicatie (controleren) en het andere voor het visualiseren van bepaalde data (video, landkaart,foto, ...). Zowel het controle als het visualisatie gedeelte kan verhuizen naar het nieuwe apparaat. Dit naargelang op wel apparaat men momenteel werkt. Werkt men op een PDA kan besloten worden het visualisatie gedeelte te verhuizen naar een apparaat met groot scherm. Of omgekeerd, men zit op een pc maar wilt van op een afstand een slideshow controleren, in dit geval kan men het controle gedeelte verhuizen naar de PDA. Op deze wijze kunnen we de pda gebruiken voor het besturen van de slideshow op de pc.

  Het Pebbles systeem [30] is hier een mooi voorbeeld van. In dit systeem wil men de PDA gebruiken als afstandsbediening voor verschillende apparaten. Hier wordt een beschrijving van het controle gedeelte getransporteerd van het origineel apparaat naar de PDA. Op de PDA wordt vervolgens een User Interface gegenereerd waarmee het apparaat bedient kan worden. Op deze wijze kan men dit apparaat bedienen via de PDA, waar men vroeger de User Interface op het apparaat zelf moest gebruiken.

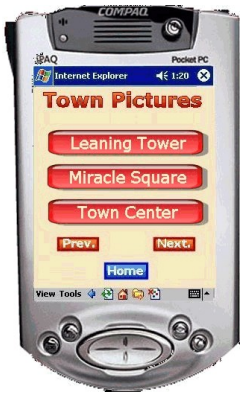


Afbeelding 7: Originele UI

Een ander voorbeeld is het Partial Migration Service komende uit [27]. We zien hier een web-applicatie, waarbij men foto's van monumenten kan bekijken. Natuurlijk zal men op een PDA scherm nooit volledig kunnen profiteren van de hoge resolutie foto's die beschikbaar zijn.

Met behulp Partial Migration Service systeem kan het visualisatie-gedeelte van de User Interface verhuist worden naar een apparaat met een groter scherm. Bijvoorbeeld een desktop-pc.

De visualisatie gedeelte op de pc wordt nu bestuurd door het controle gedeelte op de pda.



Afbeelding 8: Controle gedeelte van de UI

Afbeelding 9: Visualisatie Gedeelte van de UI

- Gemengde Migratie
  Hierbij wordt de User Interface opgedeeld in twee of meerdere delen. Elk deel zal zal zowel een presentatie als controle gedeelte bevatten. Deze manier van werken is eveneens mogelijk bij Roam. Hier zullen we later uitgebreider op terug komen.

Voor het soort Mobiele Applicatie waar vooral op gedoeld wordt in deze thesis, is de Totale Migratie de meest geschikt. Dit omdat we wensen de applicatie te verhuizen met als doel verder te werken op een andere locatie en apparaat. Bij gedeeltelijke en gemengde migratie kan dit soms moeilijk uitvallen, wanneer we ons verwijderen van het originele apparaat.

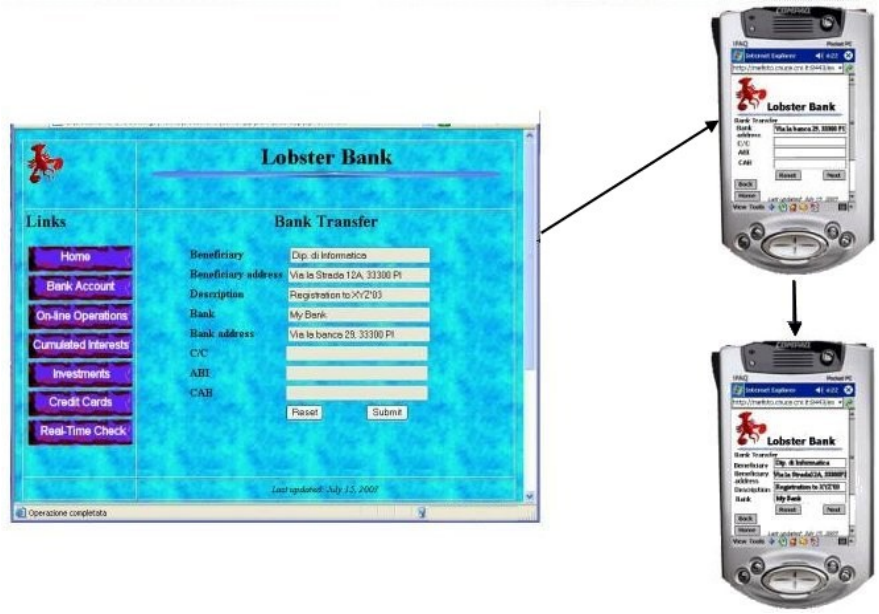
We hebben het nog niet gehad over het feit dat deze user interfaces natuurlijk aangepast kunnen worden naar het apparaat waarop ze zich bevinden. Dit kan gebeuren door de UI te mappen naar een nieuwe UI.

5.2 Mappings

Zoals eerder vermeld kan het gebeuren dat een User Interface opgedeeld wordt in kleinere delen. Dit om het bruikbaar te houden op kleine schermen. Natuurlijk is het omgekeerde ook mogelijk. Wanneer we van een klein scherm naar een groot scherm verhuizen is het absurd hier geen gebruik van te maken. We kunnen dan ook verschillende onderdelen samen nemen tot een geheel. In [26], [27], [28] en [29] worden volgende mappingen besproken:

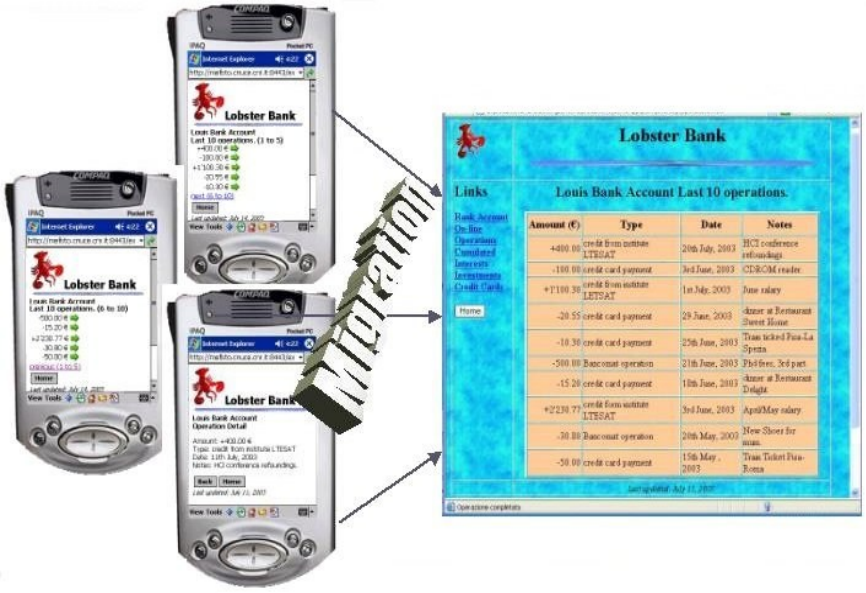
- One to One
  Hierbij worden alle UI-elementen één op één gemapt naar de nieuwe UI. Dit betekend dat er vaak geen veranderen plaats vinden aan de de UI-elementen. Maar het kan voorkomen dat men een bepaalde transformatie toepast zoals een list widget die veranderd wordt in een dropdown box widgit. Dit om plaats te besparen. Het resultaat blijft een gelijkaardige UI, waar men direct de vorige versie op herkent.

- One to Many
  Hierbij wordt een enkel UI-element gemapt naar verschillende andere UI-elementen. Dit kan gaan van een simpele UI-element tot een volledig scherm. In het volgende voorbeeld zien we hoe een enkel invoer-scherm voor het invoeren van gegevens, omgevormd wordt naar een wizard bestaande uit twee schermen.



Afbeelding 10: One to Many, http://giove.cnuce.cnr.it/MIgration/

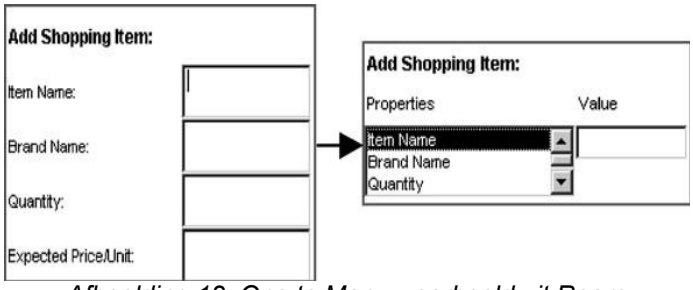
- Many to One
  Zoals eerder vermeld kan het ook gebeuren dat men van een groot scherm naar een klein verhuist. In dat geval zal er vaak een Many to One mapping gebeuren. Dit houd in dat verschillende UI-elementen gecombineerd worden tot een enkele representatie ervan.



Afbeelding 11: Many to One, http://giove.cnuce.cnr.it/MIgration/

In het voorbeeld hierboven zien we dat de informatie, komende van drie kleine schermen, samen genomen wordt tot een enkel groot scherm

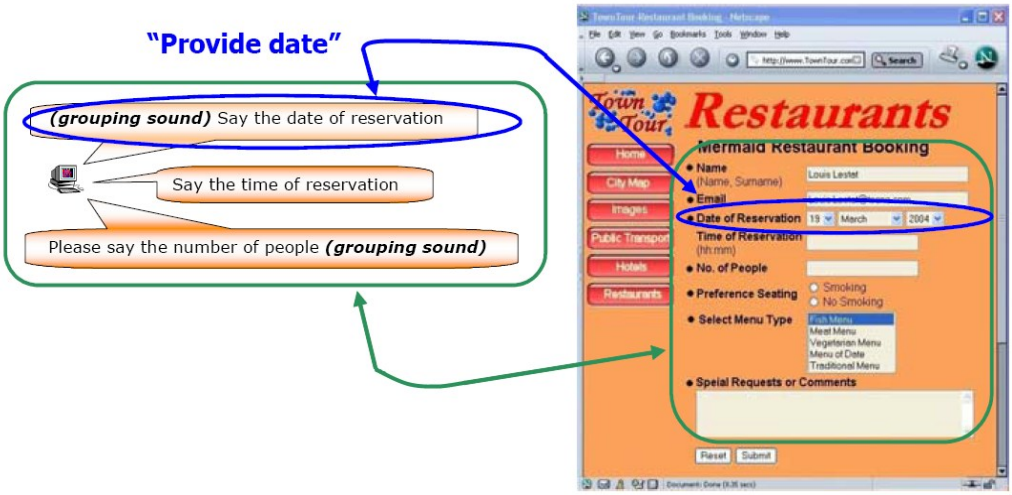
- Many to Many
  Hierbij zullen verschillende UI-elementen gemapt worden naar verschillende andere UI-elementen. Een voorbeeld hiervan zien we in de afbeeldinging hieronder. We zien dat er een verzameling tekstveld (many) gemapt worden naar een Listbox en een tekstveld (many).



Afbeelding 12: One to Many, voorbeeld uit Roam

5.3 Trans- en Unimodale migratie

We zijn er voorlopig van uit gegaan dat wanneer we de applicatie met zijn UI verhuizen naar een nieuw apparaat, de interactie-methoden hetzelfde bleven. Het kan natuurlijk mogelijk zijn dat dit niet zo is. Stel dat we onze bankzaken aan het regelen zijn via pc-banking. Op een gegeven moment moeten we weg en willen we toch nog ons werk voort zetten aan de hand van phone-banking. We zullen nu de invoer van gegevens via gesproken commando's uitvoeren. Hierbij gaan we van een grafische interface naar een spraak-gebaseerde interface. Hierdoor is de volledige manier (Modaliteit) van data-invoer veranderd. Wanneer dit gebeurd spreken we van Trans-Modale migratie. [31] bespreekt een oplossing voor de Trans-Modale migratie van web-applicaties. Hierbij wordt over gegaan van een grafische interface naar een spraak-gebaseerde interface.



Afbeelding 13: Voorbeeld Trans-Modale Migratie

We kunnen deze migratie koppelen met de reeds besproken mappings en migratie typen. We kunnen een een applicatie hebben op de pc, waarbij we het controle gedeelte laten verplaatsen naar een gsm. Het visualisatie gedeelte blijft op de pc. Via spraak commando's (volgende slide, vorige slide, ...) besturen we de visualisatie. Op deze manier hebben we een Trans-Modale gedeeltelijke migratie gedaan.

Hetzelfde geld voor de verschillende mappingen. Het is mogelijk dat een getal in een enkele keer ingegeven moet worden op de grafische interface, maar dat bij de spraak-gebaseerde interface men dit in verschillende stappen moet doen (eerst de duizendtallen opgeven, dan de honderdtallen, ...). Hier zouden we kunnen spreken van een Trans-Modale One to Many mapping.

Wanneer er geen verandering gebeurd spreken we van een Unimodale Migratie[15]. Dit is de situatie waarvan we uit gegaan zijn in de voorbeelden gegeven bij de uitleg van de verschillende soorten migratie en mappings.

6 Het .NET Framework

In dit deel van de thesis zullen we het hebben over het .NET Framework. Er zal nagegaan worden welke features dit framework ons aanbiedt voor de creatie van Mobiele Applicaties en in welke mate het daardoor geschikt hiervoor geschikt is.

Het .NET Framework stelt een programmeur in staat om applicaties te ontwikkelen in een high level taal voor verschillende platformen:

- Desktop-platform: Console & GUI applicaties
- Web-platform: ASP.NET applicaties & XML Web services
- Mobiele-Platform: Compact Framework voor het ontwikkelen van applicaties voor PDA's, smartphones en pocketpc's

Wanneer we kijken naar de doelstelling van de thesis, namelijk de creatie van een Mobile Applicatie welke op verschillende platformen kan werken, lijkt de ondersteuning van deze

15 The Migration Project: http://giove.cnuce.cnr.it/MIgration/unimodal.html

verschillende platformen al een stap in de goede richting. Het belangrijkste voor ons is de compatibiliteit tussen het .NET framework en het .NET Compact Framework. Wanneer deze beide compatibel genoeg zijn, kunnen we een applicatie ontwikkelen die zonder problemen op beide platformen kan worden uitgevoerd.

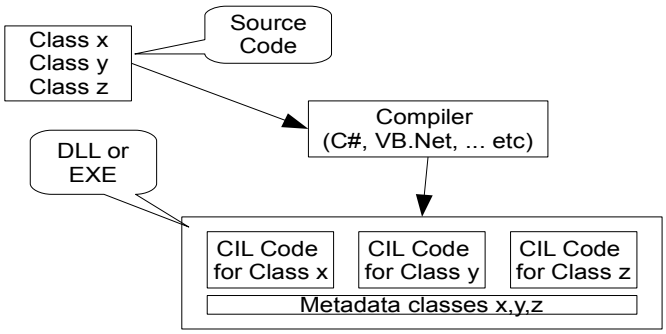
Jammer genoeg, maar zoals verwacht, komen beide niet volledig overeen. De desktop versie van het .NET framework is veel uitgebreider en bevat features die te zwaar zouden zijn om op een PDA, smartphone of pocketpc uit te voeren. Een uitgebreide vergelijking tussen het .NET en .NET Compact framework vinden we in [32]. Wanneer we applicaties ontwikkelen die op beide platformen moeten werken, mogen alleen deze features gebruiken die op beide platformen aanwezig zijn.

We zullen nu het .NET Framework bekijken waarbij we onze aandacht vestigen op de zaken die nodig zijn voor het ontwikkelen van een veilige Mobiel Applicatie Systeem. De aandachtspunten zijn:

- Algemene werking van het .NET Framework.
- Hoe we resources en code gemakkelijk kunnen verpakken met behulp van Assemblies
- Dynamische inladen van code.
- Evidence & Code Acces Security, waarmee we kunnen bepalen in welke mate welke code toegang tot het systeem krijgt
- Serializatie, zowel Binair als XML gebaseerd. Bruikbaar voor het bewaren van de applicatie staat.
- Reflectie waarmee we de eigenschappen van objecten kunnen opvragen en onderzoeken.

Een kleine opmerking: doorheen de thesis zal het vaak gaan over code. Het gaat hier steeds om Managed .NET Code en geen Native Code.

6.1 Algemene Werking



Afbeelding 14: Compilatie in .NET Framework

Met het .NET Framework kan men in verschillende talen programmeren zoals C# of VB.Net. Bij compilatie van de code wordt deze niet rechtstreeks omgezet naar machine taal maar naar een Intermediate Language, meer specifiek de 'Common Intermediate Language' (CIL). Dit wordt dan met de nodige meta informatie in een dll of exe verpakt.

Hieronder wordt een voorbeeld getoond van de conversie van een stuk C# code naar CIL code. Dit voorbeeld kan terug gevonden worden in [33].

```
using System;
class Program
{
   static void Main()
   {
      Console.WriteLine("Hello, World!");
   }
}
```

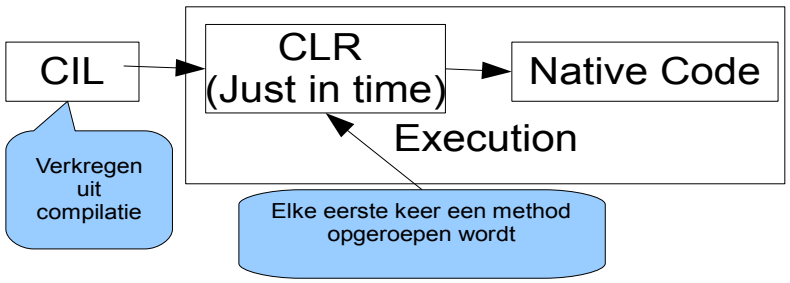
Code 2: C# code

Bovenstaande stukje code wordt na compilatie omgezet naar een CIL versie

```
.assembly HelloWorld {}
.assembly extern mscorlib {}

.class Program extends [mscorlib]System.Object
{

   .method static void Main() cil managed
   {
      .entrypoint
      .maxstack 1
      ldstr "Hello, World!"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
   }
}
```

Code 3: CIL Code

Bovenstaande stuk code is de tekstuele versie van de CIL, deze wordt onder binaire vorm opgeslagen. Bij de uitvoering van de dll of exe wordt deze door de Common Language Centime (CLR) 'Sust In Time' omgezet naar Native Code zodat deze uitgevoerd kan worden door de computer.



Afbeelding 15: Uitvoering van een .NET exe of dll

Een .NET dll of exe wordt ook een Assembler genoemd. Deze bevat niet enkel de gecompileerde code maar ook extra informatie.

6.2 .NET Assembler

Een Assembler[16] is de verpakking van een .NET applicatie of bibliotheek. We zullen nu nagaan wat deze allemaal bevat en in welke mate het voor Mobiele Applicaties gebruikt

16 http://msdn2.microsoft.com/en-us/library/k3677y81(vs.80).aspx

kan worden.

Een .NET Assembler bestaat uit vier delen, namelijk:

- CIL code: Zoals reeds uitgelegd wordt houd het compileren van .NET code in dat deze omgezet wordt naar een Common Intermediaire Language. Deze code zal dan pas bij het uitvoeren van de applicatie omgezet worden naar native code. De Assembly bevat dus een soort broncode van de applicatie.

- Type Metadata: bestaande uit
  - Type Informatie: klassen waarvan overgeërfd geërfd werd en de geïmplementeerde interfaces.
  - Informatie over Class Members

  Deze informatie vormt de hoeksteen van het .NET Framework [34]. Het wordt voor talloze zaken gebruikt waaronder:
  - Serializatie engines gebruiken metadata om te weten hoe ze typevelden moeten serializeren.
  - De garbage collector gebruikt deze informatie om te weten wanneer velden in een object naar andere objecten verwijzen.
  - Compilers kunnen dankzij deze metadata zonder hulpmiddelen naar types verwijzen die gedefinieerd zijn in een andere programmeertaal.
  - De Reflection API in het .NET framework steunt op deze data.
  - Visual Studio gebruikt de metadata voor zijn IntelliSense.
  - ...

- Manifest: ook Assembly Metadata genoemd en beschrijft de Assembly
  Het manifest[17] bevat een hele reeks aan attributen, we zullen hier de belangrijkste opsommen. Voor een volledige lijst kan gekeken worden op de MSDN site van Microsoft[18].
  - Naam, string met de naam van de Assembly
  - Versienummer, een major en minor versie nummer,een revisie en build nummer.
  - Culture, informatie omtrent de cultuur of taal die de Assembly ondersteunt.
  - Strong Name informatie, De publieke sleutel en digitale handtekening die gebruikt kan worden om de integriteit van de Assembly te controleren,later zal hier meer informatie over geven worden.
  - Een lijst van alle bestanden in de Assembly bestaande uit de hash en de naam van het bestand
  - Een lijst met van andere Assemblies waar naar gerefereerd werd door de Assembly
  - ...

  De eerste vier attributen vormen samen de Strong Name die gebruikt kan worden voor de unieke identificatie van een Assembly. Ook hier zullen we op terugkomen.

- Resources: Alle zaken zoals prenten, tekstbestanden, ... etc.

Een Assembly hoeft niet elk onderdeel te bevatten. Het enige verplichte onderdeel is het

17 Manifest: http://msdn2.microsoft.com/en-us/library/zst29sk2.aspx
18 Manifest attributen: http://msdn2.microsoft.com/en-us/library/4w8c1y2s(vs.80).aspx

Manifest, zonder deze zal een Assembly nooit uitgevoerd worden. Het spreekt voor zich dat een Assembly, hoewel mogelijk, met enkel een Manifest niet veel nut heeft. Zoals reeds vermeld is verzenden van code en resources een belangrijk onderdeel bij Mobiele Applicaties. Het feit dat een Assembly een combinatie kan zijn van code en resources zal dus helpen bij het gebruik van het .NET Framework hiervoor.

Een tweede belangrijk aspect, welke nog niet vermeld werd, omtrent Assemblies is dat deze een oplossing kan bieden voor enkele van de reeds aangekaarte veiligheidsproblemen bij Mobile Code. Hier zal nu dieper op in gegaan worden.

6.2.1 .NET Assembly en Veiligheid[19]

Een .NET Assembly geeft ons enkele mogelijkheden om de integriteit en tot op een zekere hoogte de afkomst van een Assembly te garanderen. Dit kan door een Assembly digitaal te ondertekenen. Wanneer dit gedaan wordt krijgen we een Strong Named Assembly. Hiervoor wordt gebruik gemaakt van het publieke sleutel (public key) algoritme[20] en hashing. Eerst wordt een hash gemaakt van de inhoud van de Assembly, daarna wordt de hash code met de private key van de Assembly eigenaar geëncrypteerd. Deze code wordt dan mee verpakt in de Assembly samen met de Public Key van de Assembly eigenaar. De gebruiker van de Assembly kan dan aan de hand van de Public Key de originele hashcode te weten komen. Wanneer de gebruiker ook een hashcode van de Assembly te berekent kan deze met zekerheid weten of de integriteit van de Assembly nog in tact is.

Een Strong Named Assembly biedt ons dus enkele voordelen[21]:

- Doordat bij Strong Named Assemblies de hashcode onderdeel uitmaakt van de naam kan een Assembly altijd uniek geïdentificeerd worden. Niemand kan een andere Assembly aanmaken met dezelfde naam als deze van de oorspronkelijke uitgever. Wanneer men dus weet bij wie de publiek sleutel hoort, kan men redelijk zeker zijn omtrent de afkomst van een Assembly.

- Doordat de hashcode, die ter controle gebruikt wordt voor het verifiëren van de integriteit geëncrypteerd wordt, kan niemand zonder de juist Private sleutel de Assembly aanpassen zonder dat dit onopgemerkt blijft.

Maar bij deze voordelen horen ook enkele nadelen, namelijk:

- Het public key algoritme is veilig wanneer we er van uit gaan dat de ondertekenaar zijn Private Key nooit geconfisqeerd werd. Wanneer dit het geval is, kan de integriteit van de Assembly niet meer gegarandeerd worden. Er is voor de eigenaar van de Assembly ook geen manier om aan te geven dat zijn huidige sleutel niet meer veilig is en dat Assemblies, ondertekend met deze sleutel, niet meer aanvaard mogen worden.

- Een bijkomend probleem is dat we niet volledig zeker kunnen zijn omtrent de identiteit van de maker van de Assembly. Iedereen kan zeggen “ik ben persoon X en dit is mijn sleutel”, maar er is geen enkel bewijs dat deze persoon dan ook echt persoon X is.

19 http://msdn2.microsoft.com/en-us/library/ab4eace3.aspx
20 http://en.wikipedia.org/wiki/Public_key
21 http://msdn2.microsoft.com/en-us/library/wd40t7ad.aspx

Gelukkig zijn deze problemen niet onontkoombaar. Het .NET framework maakt het mogelijk om een Assembly te ondertekenen met behulp van een certificaat. Deze certificaten dienen aangevraagd te worden bij een erkende uitgever. Bij de aanvraag van zo'n certificaat zal de aanvrager zijn identiteit moeten kunnen bewijzen. Op deze wijze zijn we altijd zeker dat we ook de echte identiteit van de maker van de Assembly hebben. Het tweede grote voordeel bij het gebruik van certificaten, is het feit dat een certificaat ongeldig verklaard kan worden. Bij het gebruik van een assembly, ondertekend met behulp van een certificaat, moet elke maal opnieuw gecontroleerd moet worden of dit certificaat nog geldig is. Hierdoor is er nu wel een manier om gebruikers te waarschuwen dat Assemblies, ondertekend met een gecomprimeerde private key, niet meer geldig zijn.

Dit vormt slechts een onderdeel van de verschillende veiligheidsmaatregelen die het .NET Framework aanbied. Andere maatregelen worden later nog uitgebreid uitgelegd.

6.2.2 Conclusie Assemblies

Wanneer we controleren, met de reeds gegeven informatie, in welke mate .NET geschikt zou zijn voor Mobile Code zien we dat deze voorlopig uitstekend scoort. De Assembly zorgt voor een makkelijke manier in het transporteren van code en resources. Wanneer we terugkijken naar de veiligheidsproblemen betreffende Mobile Code zien we dat er al twee grote problemen opgelost werden, namelijk authenticatie en het aantonen van de integriteit. Dit door het gebruik van Strong Named Assemblies.

6.3 Dynamic Code Loading

Sommige vormen van Mobile Code vereisen dat een reeds lopende applicatie, dynamische code kan inladen. Bijvoorbeeld bij het Mobile Agent paradigma zal de host-applicatie reeds gestart zijn en moet deze de Mobile Agents kunnen inladen en uitvoeren. Het zomaar inladen en uitvoeren van code heeft, zoals eerder gezegd, vergaande gevolgen omtrent de veiligheid van een computersysteem. Daarmee is het ook aangewezen dat we toch enige vorm van bescherming kunnen krijgen tegen malafide code zodat deze zo weinig mogelijk kansen krijgt tot het verrichten van kwaadaardige operaties. We moeten de ingeladen code kunnen isoleren van de rest van het systeem en ervoor zorgen dat deze niet zomaar gelijk welke acties kan uitvoeren. Samengevat moeten we bij Mobiele Applicatie systemen de mogelijkheid hebben tot het veilig en dynamische inladen van code. Het .NET framework heeft hiervoor Applicatie Domeinen[22] [23], deze maken het mogelijk om “at-runtime” Assemblies in te laden in een afgescheiden ruimte.

6.3.1 Applicatie Domeinen

In een normale omstandigheden is een proces de kleinst mogelijk eenheid waarin een applicatie geïsoleerd uitgevoerd kan worden. Voor elke applicatie die opgestart wordt, wordt een proces gestart. Een proces wordt aangemaakt op het niveau van het besturingssysteem en houd in dat het systeem een applicatie voorziet van een afgeschermd stuk werkgeheugen en de nodige systeembronnen. Elk proces krijgt een ID en set van een set van veiligheid permissies. Op deze manier wordt getracht applicaties te isoleren zodat ze elkaar niet rechtstreeks kunnen beïnvloeden.

Op zich lijkt een proces een voldoende manier om een geïsoleerde ruimte te creëren. Het

22 Applicatie Domeinen: http://aspalliance.com/951

23 MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconapplicationdomains.asp

nadeel, dat voor enige belemmering zorgt bij gebruik van processen, is dat ze redelijk kostbaar zijn om aan te maken. Wanneer we voor elk in te laden Mobile Applicatie een gans nieuwe proces zouden moeten starten kan dit voor performantie problemen zorgen. Zeker in een omgeving waar vaak Mobiele Applicaties gestart en gestopt dienen te worden.

```
ProcessStartInfo psi = new
ProcessStartInfo();
psi.FileName = "cmd.exe";
Process proc = Process.Start(psi);
```

Code 4: Het aanmaken van een nieuw proces in .NET C#

Een Applicatie Domein heeft ongeveer de zelfde functionaliteit als een proces, alleen draait deze in een proces waarbij een proces meerdere Applicatie Domeinen kan bevatten. Een Applicatie Domein creëren in een reeds bestaand proces is veel lichter dan het creëren van een heel nieuw proces. Het applicatie domein is een extra laag bovenop een proces. Dit vormt een uitstekend hulpmiddel voor de creatie van een Sandbox waarin de code geladen kan worden.

6.3.2 Waarom Applicatie Domeinen gebruiken?

Dat een Applicatie Domein minder zwaar is om te creëren is niet de enigste reden waarom we deze zouden willen gebruiken. Andere redenen zijn:

- Een fout in een Applicatie Domein heeft geen invloed op applicaties in hetzelfde proces.

- Configuratie kan per domein gebeuren en moet niet op proces niveau. Voorbeelden hiervan zijn het instellen van het “Cach Path”, gebruikt voor het bewaren van Shadow kopieën van Assemblies en de “Base Directory”, hierin wordt gekeken bij het zoeken naar in te laden Assemblies.

- Elk Applicatie Domein in eenzelfde proces kan een eigen Security Acces Level krijgen. Code in een applicatie domein kan hiermee beperkt worden in de acties die deze kan uitvoeren. IO operaties kunnen bijvoorbeeld geweigerd worden. Later wordt hier dieper op in gegaan wanneer het onderwerp “Veiligheid in het .NET Framework” aangekaart wordt.

- Code, uitgevoerd in een bepaald Applicatie Domein, kan niet rechtstreeks toegang krijgen tot code en resources van een ander Applicatie Domein. Wanneer informatie, tussen verschillende Applicatie Domeinen uitgewisseld moet worden, dient gebruik gemaakt te worden van Remoting.

- Een Applicatie Domein kan verwijderd worden zonder dat dit andere Applicatie Domeinen in hetzelfde proces beïnvloedt. Het verwijderen van een Applicatie Domein uit het geheugen gaat gepaard met de verwijdering van alle Assemblies geladen in dit domein. Dit is een zeer handig feit. Onder normale omstandigheden is het niet mogelijk om een geladen Assembly te verwijderen uit het geheugen zonder het proces te stoppen. Ze blijven dus ingeladen zolang de applicatie die ze ingeladen had bestaat. Dit is nefast voor het geheugengebruik wanneer we vaak Assemblies moeten inladen die maar voor een beperkte tijd nodig zijn. Een typische voorbeeld hiervan zijn Mobile Agents. Wanneer er voor elk in te laden Assembly een

nieuw Applicatie Domein gecreëerd wordt, kunnen we een Assemblies verwijderen door het Applicatie Domein waarin de Assembly geladen werd te verwijderen.

6.3.3 Conclusie

Applicatie Domeinen vormen een handig hulpmiddel voor het ontwikkelen van Mobiele Applicaties. Hiermee kunnen we Sandboxing toepassen welke we reeds besproken hebben als oplossing voor het beschermen van de host tegen malafide code.

6.3.4 Serializatie

Serializatie geeft de mogelijkheid om de staat van een actief object te bewaren op een opslag medium of te verzenden over een netwerk. Via de opgeslagen data is het daarna mogelijk het object in zijn oorspronkelijke staat te herstellen. Het bewaren van een object staat wordt serializeren genoemd, het herstellen ervan deserializeren.

6.3.5 Gebruik van serializatie

Zoals eerder vermeld wordt serializatie hoofdzakelijk voor twee zaken gebruikt, persistentie en uitwisselen van objecten in gedistribueerde systemen.

In de meeste applicaties is het nodig om objecten voor langere tijd dan de executie te bewaren. Op dat moment kan kan gebruik gemaakt worden van serializatie om deze te bewaren. Hierdoor worden de objecten persistent, dit betekend dat ze op een later tijdstip weer opgevraagd kunnen worden en dus niet verdwijnen met het afsluiten van de applicatie waartoe ze behoren. In [35] wordt het fenomeen van Persistente Objecten uitgebreid bestudeerd.

Buiten het bewaren van gegevens kan serializatie ook gebruikt worden bij het uitwisselen van objecten in een gedistribueerd systeem. Hierbij moeten objecten tussen verschillende applicaties uitgewisseld kunnen worden. Bijvoorbeeld bij RMI worden vaak objecten als parameter meegegeven in de oproep van een Remote Method. Aan de hand van serializatie kunnen deze objecten omgezet worden naar een vorm die dit mogelijk maakt. Naargelang de gebruikte techniek kan dit een binaire of xml-gebaseerde vorm zijn. [36] geeft ons een goed inzicht omtrent dit gebruik van serializatie.

We zullen nu de mogelijkheden van serializatie in het .NET Framework wat uitgebreider bespreken.

6.3.6 Soorten Serializatie in .NET

In .NET[24] zijn er twee vormen van serializatie:

- Binaire Serializatie
  De meest geavanceerde vorm van Serializatie in .NET Framework. Bij de Binaire Serializatie wordt alle data mogelijk nodig voor het herinstantiëren van een object bewaard, hierdoor maakt Binaire Serializatie het mogelijk om volledige staat van een object te bewaren en te herstellen. Deze vorm kan ook als XML document opgeslagen worden met behulp van de SOAPFormatter. Hierover volgt later meer uitleg.

24 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrserializingobjects.asp?frame=true

- XML Serializatie
  Zal in tegenstelling tot Binaire Serializatie niet de volledige staat van een object bewaren, maar slechts de Publieke Velden en Eigenschappen die zowel lees als schrijfbaar zijn. De reden waarom deze manier van serializatie toch gebruikt wordt is dat alle data onder XML vorm opgeslagen wordt en dus zeer makkelijk leesbaar is. Dit maakt het mogelijk om geserialiseerde object te delen tussen verschillende programmeertalen, wat bij Binaire Serializatie niet zo simpel is. Met de SOAPFormatter kan men natuurlijk ook een XML-versie maken van de staat. Maar hierbij moet de SOAP-standaard gevolgd worden, wat bij de simpele XML-Serializatie niet hoeft. Dit bied natuurlijk een grotere flexibiliteit aan qua opstelling van het xml-document.

We zullen nu beide vormen uitvoerig beschrijven.

6.3.7 Binaire & SOAP Serializatie

Zoals eerder vermeld kan via Binaire Serializatie [37] [38] een exacte kloon gecreëerd worden van een object. Binaire Serializatie houdt in dat alle publieke en private velden, met hun Type Identity en Assembly informatie, van een object geconverteerd worden naar een stroom van bytes. Deze stroom van bytes kunnen dan opgeslagen worden in een bestand of verzonden over het netwerk. Door het bijhouden van de Type Identity en Assembly informatie wordt Type Fidelity verzekerd, dit zorgt ervoor dat er gegarandeerd kan worden dat een object steeds naar zijn oorspronkelijk Object Type gedeserialiseerd zal worden. Een nadeel hiervan is dat toegang tot het origineel Object Type moet zijn bij de deserializatie ervan.

Soap Serializatie,niet te verwarren met de soap mogelijkheden van XML Serializatie, werkt op dezelfde manier als Binaire Serializatie maar zal de nodige data niet opslaan onder binaire vorm maar als een XML document

Bij het serializeren wordt gebruik gemaakt van een Formatter, een BinaryFormatter of een SOAPFormatter. Deze zal aan de hand van volgende stappen een object serializeren:

1. De formatter controleert of hij een Surrogate Selector[25] bezit voor het te serializeren object. Deze Surrogate Selector kan de formatter helpen bij het verkrijgen van de nodige data voor de serializatie. Er volgt later nog uitleg omtrent deze Surrogate Selector.

2. Wanneer voor het object geen geschikte Surrogate Selector beschikbaar is wordt er gecontroleerd of het object het Serializable attribuut bezit. Dit attribuut duidt aan dat een klasse geserializeerd kan worden, wanneer dit niet zo is kan de Serializatie niet doorgaan.

   Het .NET Framework voorziet ook in een manier om bepaalde gevens van een object niet te serializaren. Hiervoor plaatst men een NonSerialized attribuut voor het veld waarvan de data niet geserializeerd hoeft te worden.

25 http://msdn2.microsoft.com/en-us/library/system.runtime.serialization.Surrogateselector(VS.80).aspx

```
[Serializable]
public class Object_State
{
        private string name;
        private int obj_value = 0;
        [NonSerialized] private int toreador = 0;

            public Object_State()
            {

            }
 }
```

Code 5: klasse met Serializable en NonSerialized attribuut

3. Als het Serializable attribuut aanwezig is wordt er gecontroleerd of het object de Iserializable Interface implementeert. Hiervoor dient de GetObjectData method geïmplementeerd te worden, deze moet instaan voor het vullen van het SerializationInfo object. Dit SerializationInfo bevat de nodige data voor het herinitialiseren van het geserialiseerd object. Bij de deserializatie zal de speciale, eveneens te implementeren, constructor aangeroepen worden zodat op basis van het SerializationInfo Object de velden van het object de gewenste waarden kunnen krijgen.

```
[Serializable]
public class Object_State : ISerializable
{
   private string name;
   private int obj_value = 0;

    public Object_State()
    {

    }

    protected Object_State(SerializationInfo info, StreamingContext context)
    {
       name = info.GetInt32("n");
       obj_value = info.GetInt32("o");
    }

    public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
    {
       info.AddValue("n", name);
       info.AddValue("o", obj_value);
    }
}
```

Code 6: Object die de ISerializable interface implementeert

4. Wanneer er geen Iserializable Interface geïmplementeerd is, zal de standaard werkwijze gebruikt worden, namelijk het serializeren van alle velden die niet NonSerialized gemarkeerd zijn.

Er moet opgelet worden dat wanneer een object geserializeerd wordt elk veld van dat object eveneens serializeerbaar moet zijn, indien dit niet zo is zal de serializatie falen.

Binaire Serializatie lijkt ons zeer geschikt voor het schrijven van Mobile Applicaties. Dit kan

bewezen worden door het gebruik ervan in Mobile Agent projecten zoals [8] [9]. Het maakt het mogelijk om de volledige staat van een serializeerbaar object te bewaren en er kan flexibel omgegaan worden met welke data wel of niet bewaard dient te worden. Een bijkomend voordeel is dat het resultaat van Binaire Serializatie een compact binair bestand is, wat schijfruimte bespaard en minder netwerk belastend is bij het doorzenden van een object en zijn staat.

Het groot nadeel bij Binaire Serializatie is dat de binaire data enkel geschikt is voor de uitwisseling van objecten tussen .NET applicaties. Een oplossing hiervoor is het gebruik maken van de SOAPformatter wat een XML bestand oplevert die makkelijker leesbaar is en tot interoperabiliteit kan leiden tussen .NET en niet .NET applicaties.

Overzicht Binaire en SOAP Serializatie

- Binary Serialization:
  - voordelen
    - Resultaat veel compacter dan tekstuele vormen zoals XML
    - Beste performantie, er moet geen XML document gegenereerd worden.
    - De volledige Object staat wordt geserializeerd
  - nadelen
    - Binair formaat .NET specifiek en niet menselijk lees- of editeerbaar

- SOAP Serialization
  - voordelen
    - Creëert SOAP berichten.
    - Interoperabiliteit tussen .NET en niet .NET applicaties
  - nadelen
    - Verstaat enkel SOAP, andere XML documenten niet leesbaar

Er is slechts één grote tekortkoming waardoor Binaire of Soap serializatie niet gebruikt kon worden bij de de implementatie van het Mobiele Applicatie Systeem. Het is namelijk niet beschikbaar voor het .NET Compact Framework. Een meer beperkte vorm van Serializatie is wel beschikbaar, namelijk XML Serializatie.

6.3.8 Surrogates & Surrogate Selectors

Het .NET Framework kan bij serializatie gebruik maken van Surrogates en Surrogate Selectors[26]. Een Surrogate is een klasse die de Serializatie van een bepaalde type kan afhandelen.

Een serializatie Surrogate implementeert de ISerializationSurrogate interface. Dit betekend dat de methoden GetObjectData en SetObjectData geïmplementeerd dienen te worden. Deze methoden staan in voor het vullen en uitlezen van het reeds besproken SerializationInfo object.

26 Application Development: Advanced serialization in .NET

```
public class SurrogateVoorbeeld : ISerializationSurrogate
{
    public SurrogateVoorbeeld()
    {
    }

    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        VoorbeeldObject objVoorbeeldObject = (VoorbeeldObject)obj;
        info.AddValue("Data", objVoorbeeldObject.Data);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        VoorbeeldObject objVoorbeeldObject = (VoorbeeldObject)obj;
        objVoorbeeldObject.Data = info.GetValue("Data");
        return (objVoorbeeldObject);
    }
}
```

Code 7: Implementatie ISerializationSurrogate

Een Surrogate kan ervoor zorgen dat een klasse die niet serializeerbaar is toch geserialiseerd kan worden. Natuurlijk hangt de correctheid en volledigheid van de serializatie af van hoe men de ISerializationSurrogate Interface geïmplementeerd heeft.

Er zijn verschillende redenen waarom men via een Surrogate Selector zou willen werken. De belangrijkste reden is dat we soms een type willen serializeren welke niet serializeerbaar is. Soms is het niet mogelijk om een type aan te passen zodat deze wel serializeerbaar is. Dit kan wanneer het type bijvoorbeeld afkomstig is uit een externe bibliotheek. In dat geval kan gebruikt gemaakt worden van een Surrogate. Een andere reden kan zijn dat we extra informatie aan een type willen toevoegen op het moment van serializatie.

Een Surrogate Selectors is een verzameling van Surrogates. Deze kan op aanvraag de juiste Surrogate voor een bepaald type geven. Dit zodat de serializatie ervan juist kan gebeuren. We kunnen een Surrogate Selectors koppelen aan een formatter. Deze formatter zal, zoals eerder al uitgelegd, bij het serializeren van een type controleren of er een Surrogate beschikbaar. Indien zo wordt deze gebruikt voor het serializeren van het object.

```
public void SerializatieVoorbeeld()
{
    //aanmaken van object, Stream, Formatter Surrogate Selector, Surrogate
    VoorbeeldObject vbObject = new VoorbeeldObject();
    MemoryStream objStream = new MemoryStream();
    BinaryFormatter formatter = new BinaryFormatter();
    SurrogateSelector surrogateSelector = new SurrogateSelector();
    SurrogateVoorbeeld surrogate = new SurrogateVoorbeeld();

    //Surrogate toevoegen aan de Surrogate Selector
    surrogateSelector.AddSurrogate(typeof(VoorbeeldObject), new
StreamingContext(StreamingContextStates.All), surrogate);

    //serializatie
    formatter.SurrogateSelector = surrogateSelector;
    formatter.Serialize(objStream, vbObject);
    objStream.Position = 0;

    //Deserializatie
    VoorbeeldObject objTestClassNew = (VoorbeeldObject)formatter.Deserialize(objStream);
}
```

Code 8: Gebruik van Surrogate en Surrogate Selector

6.3.9 XML Serializatie

XML Serializatie[27] zal, net als bij de Binaire variant, de staat van een object bewaren. Dit gebeurt onder de vorm van een XML bestand, wat het lees- en editeerbaar maakt voor en door mensen. Ook het uitwisselen van objecten tussen .NET en niet .NET applicaties wordt hiermee ook mogelijk. Informatie omtrent XML serializatie vinden we in [37] en [39].

XML serializatie kent wel een grotere beperking dan Binaire en SOAP serializatie:

- XML Serializatie zal in tegenstelling tot Binaire Serializatie niet de volledige staat van een object bewaren maar slechts de Publieke Velden en Eigenschappen die zowel lees als schrijfbaar moeten zijn. Hierdoor zullen niet alle types goed geserializeerd kunnen worden. Het type “System.Drawing.Color” is hier een voorbeeld van. Deze bezit niet de nodige publieke Velden en zijn Eigenschappen zijn enkel leesbaar. Hierdoor zal bij het serializeren geen informatie opgeslagen worden.

- Types moeten een default Constructor (parameterloze) bezitten als ze serializeerbaar willen zijn. Bij het deserializeren zal de XmlSerializer eerst een instantie maken van een nieuw object om vervolgens de eigenschappen en velden te vullen met de nodige data. Zonder default constructor kan dit lege object niet aangemaakt worden. Dit is ook de reden waarom properties zowel lees als schrijfbaar moeten zijn. Wanneer deze niet schrijfbaar zijn, kan de nodige data niet terug geplaatst worden.

- Klassen die de IDictionary implementeren kunnen standaard niet geserializeerd worden[28] en Collections[29] afgeleid van “System.Collections.CollectionBase” moeten een een Add methode en een Indexer bezitten willen zij serializeerbaar zijn. Zelfs als ze dit bezitten zal nog niet de volledige klasse volledig bewaard worden, enkel de

27 Soap en XML-serializatie: http://www.codeproject.com/soap/Serialization_Samples.asp
28 http://msmvps.com/blogs/rakeshrajan/archive/2006/01/15/81105.aspx
29 http://msdn2.microsoft.com/en-us/library/ms950721.aspx

Collection zelf. Alle publieke velden en properties van het object worden niet geserializeerd.

Bij de XML serializer wordt wel Type Information bijgehouden maar geen gegevens over de gebruikte Assemblies zodat geen Type Fidelity verzekerd kan worden. Dit betekend dat er geen garantie is dat een object gedeserializeerd wordt naar zijn oorspronkelijk versie van het type. Dit nadeel vormt eveneens een voordeel, een object kan gedeserializeerd worden zonder dat het originele Object Type aanwezig moet zijn. XML-serializatie laat toe dat er velden bijkomen, velden verwijderd worden, types van velden veranderen en veldnamen veranderen. Een object kan gedeserialiseerd worden naar een nieuwere of oudere versie van het originele Object Type. Dit bewijst dat XML serialization zeer flexibel is.

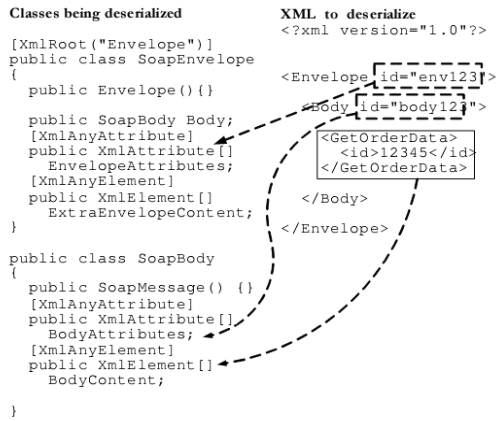
```
public class Voorbeeld{
   public int eenAantal;
}

wordt standaard omgezet naar

<Voorbeeld>
    <eenAantal>120</eenAantal>
</Voorbeeld>
```

Code 9: Voorbeeld XML serializatie

Door gebruik te maken van attributen kunnen we aangeven hoe de XmlSerializer een klasse moet converteren naar XML[30] zoals:

- Aangeven of een veld of eigenschap als een attribuut of een element geconverteerd moet worden
- Opgeven welk XML namespace gebruikt dient te worden.
- Een naam kiezen voor een element wanneer de naam van het veld of eigenschap niet voldoet.

Op deze manier kunnen we zorgen dat het gegenereerde XML bestand voldoet aan een bepaald schema wanneer dit nodig is.



Afbeelding 16: Voorbeeld uit XmlSerializer .NET Tutorial van TopXML

30 Introducing XML Serialization: http://msdn2.microsoft.com/en-us/library/182eeyhh(VS.80).aspx

Overzicht XML-Serializatie

- voordelen
  - Volledig en flexibele controle over resulterend XML document
  - Menselijke lees- en editeerbaar
  - Volledige Object staat wordt geserializeerd
- nadelen
  - Te serializeren Object moet een Default Constructor bezitten.
  - Enkel Public Fields en lees- en editeerbare Properties worden geserializeerd

Ondanks zijn beperkingen kan XML-Serializatie toch goed gebruikt worden wanneer we de staat van objecten op een simpele en menselijk leesbare manier willen opslaan. Een ander groot voordeel is dat XML-Serializatie beschikbaar is op het .NET Compact en Desktop Framework. Dit maakt het een zeer geschikte tool voor de ontwikkeling van het Mobiele Applicatie Systeem waar later meer info over volgt.

6.4 Reflectie

Zoals besproken bij Mobile Code Paradigma's is een essentieel onderdeel van een Mobile Applicaties, zoals bijvoorbeeld een Mobile Agent, het kunnen bewaren van de applicatie staat. Dit moet het mogelijk maken om een applicatie te kunnen starten in dezelfde staat als dat deze was toen hij gestopt werd. Er werd al uitgelegd dat dit gedeeltelijk mogelijk is met behulp van Serializatie, maar hierbij werd ook reeds vermeld dat niet alle types zich zomaar laten serializeren. Wanneer we toch nog de staat van deze types willen bewaren, kunnen we gebruik maken van Reflectie. Reflectie geeft de mogelijkheid tot het dynamisch onderzoeken en gebruiken van klassen. Reflectie steunt op de metadata, die opgeslagen zit in de Assemblies, gecreëerd door het .NET Framwork. Zoals eerder vermeld bevat deze metadata onder andere het type informatie zoals welke methoden en leden het Assembly type bezit. Met behulp van reflectie kan deze data uitgelezen en gebruikt worden.

```
foreach (FieldInfo fieldInfo in currentType.GetFields(BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance | BindingFlags.DeclaredOnly))
{
      Object obj = fieldInfo.GetValue(obj)
}
```

Code 10: Overlopen van de velden in een object zonder deze te kennen

We kunnen, buiten het uitlezen van data, ook met behulp van reflectie methoden aanroepen en data wegschrijven naar de velden van een klasse.

```
FieldInfo fieldinfo = obj.GetType().GetField("fieldnaam", BindingFlags.Public | BindingFlags.NonPublic
| BindingFlags.Instance | BindingFlags.DeclaredOnly);

fieldinfo.SetValue(obj, value);
```

Code 11: Toekennen van een waarden aan een veld door middel van reflectie

Omdat we via reflectie dus alle nodige data kunnen ophalen en wegschrijven van en naar een object instantie, kunnen we dit gebruiken in het uiteindelijke Mobiele Applicatie Systeem om de staat van een applicatie te bewaren en laden, zeker wanneer we dit moeten doen voor types die geen gebruik kunnen maken van de serializatie mogelijkheden in het .NET Framework. Naast het bewaren van de staat geeft het ons ook de mogelijkheid

om waarden toe te kennen aan leden van een object. Hiermee biedt het .NET Framework ons een goed werktuig aan voor de ontwikkeling van Mobiele Applicaties.

7 Veiligheidsmogelijkheden in het .NET Framework

Zoals al eerder aangekaart willen we kunnen voorkomen dat Mobile Code zomaar gelijk welke actie op een systeem kan uitvoeren. Zonder deze restrictie is het afgeraden een Mobiel Applicatie Systeem te bouwen, vermits de veiligheid van het systeem niet meer gegarandeerd kan worden. Het .NET framework bied verschillende manieren[31] aan om zulke restricties mogelijk te maken. Deze restricties worden op twee niveaus bepaald, op het gebruikers-niveau (User-based) en op code niveau (Code Access Security).

7.1 User-based Security

Heel vaak zullen de mogelijke acties die uitgevoerd mogen worden op een computersysteem afhangen van de persoon die momenteel ingelogd is. User-based Security wordt in .NET Role-based Security genoemd, omdat aan een User Account een rol gekoppeld wordt analoog naar de rol van een gebruiker in een bedrijf (manager, bediende, systeem-beheerder). Elke rol zal dan ook de nodige rechten naargelang het doel van de rol.

In het .NET Framework zal een Assembly uitgevoerd worden onder een bepaalde Security Context aan de hand van een bepaalde rol. Het bepalen van de rol gebeurt door de Authentication en het bepalen van de verkregen rechten door Authorization.

7.1.1 Authentication

Bij Authentication[32] zal er gecontroleerd worden onder welke User Account we de code zullen gaan uitvoeren, aan elke User Account is een rol gekoppeld. Het is aan de hand van deze rol dat we later de Security Permissions kunnen bepalen. Het .NET framework biedt verschillende mogelijkheden tot Authentication:

- Windows Authentication
  Dit is de standaard methode gebruikt in het .NET Framework. Er moet hiervoor geen extra code geschreven worden. Dit houdt in dat er gekeken wordt naar de rol van de reeds ingelogde gebruiker op het systeem.

- IIS Authentication
  Authentication via de IIS Web Server

- Passport Authentication
  Authentication gebruik makend van de Microsoft Passport Authentication Server

- Forms Authentication
  Laat ontwikkelaars toe om een gestandaardiseerde manier van inloggen te ontwikkelen voor een applicatie. User Account kunnen applicatie specifiek gemaakt worden.

31 .NET Security Overview: http://msdn2.microsoft.com/en-us/library/aa302369.aspx
32 Authentication: http://msdn2.microsoft.com/en-us/library/syf5yeat.aspx

7.1.2 Authorization

Na de Authentication volgt de Authorization. Dit houdt in dat er gecontroleerd wordt welke rechten de gebruiker heeft aan de hand van zijn rol. Elke soort rol heeft een bepaalde verzameling van rechten. Wanneer we een rol koppelen aan een User Account zal deze de rechten, horende bij de rol, krijgen. Vandaar de naam Role-based Security.

De rol en identiteit van een persoon wordt de Principal genoemd en stelt de Security Context voor waaronder Code uitgevoerd wordt. Aan de hand van deze Principal kan de CLR beslissen welke acties toegelaten zijn en welke niet.

7.1.3 Conclusie

Role-based Security biedt een meer verfijnde vorm van veiligheidsbeleid aan, maar het grote probleem zit in het feit dat de restricties op gebruikers niveau gedefinieerd worden. Dit betekent dat een gebruiker met de rol “Administrator” nog steeds alle acties mag uitvoeren en Mobile Code uitgevoerd onder deze rol nog steeds toegang heeft tot het hele systeem.

Het is dus mogelijk in het .NET Framework om de acties van code tot op een zeker hoogte te beperken met behulp van Authentication en Authorization. Spijtig genoeg bied dit niet de vereiste verfijning die nodig is bij Mobiele Applicatie Systemen. Er zal nu gecontroleerd worden welke mogelijkheden Code Access Security biedt en of deze wel voldoende zijn.

7.2 Code Access Security

Zoals eerder aangekaart lost het gebruik van digitaal ondertekende Assemblies al een deel van de veiligheidsproblemen op. Op deze wijze kunnen we de afkomst van de Assembly te weten komen en beslissen of we deze vertrouwen of niet. Authenticatie en Authorization geven ons de mogelijkheid de acties te beperken die code onder een bepaalde rol kan uitvoeren. Wanneer we enkel deze twee methoden zouden gebruiken zou het toepassen van een veiligheidsbeleid neerkomen op:

- Eigenaar van de ontvangen Assembly gekend en vertrouwd: code alles toelaten binnen de veiligheids-context van de persoon die de code laat uitvoeren.
- Eigenaar van de ontvangen Assembly gekend en niet vertrouwd: code niet uitvoeren
- Eigenaar van de ontvangen Assembly niet gekend en niet vertrouwd: code niet uitvoeren

Men kan al merken dat er weinig subtiliteit zit in deze methode. Liever zouden we meer verfijnder te werk gaan. Soms willen we code afkomstig van een onbetrouwbare bron toch uitvoeren, zij het met de nodige restricties. Ook dienen Assemblies niet altijd even veel rechten te krijgen als de persoon die de assembly uitvoert, ook al komen ze van een betrouwbare bron.

Het .NET Framework kan dit soort verfijning aanbieden. “Code Access Security” [33] [34] of kortweg CAS kan bepaalde permissies geven of weigeren aan code. Het gaat verder dan het kortweg kijken naar de identiteit waaronder de code gedraaid wordt. Bij CAS wordt er bij elke Assembly gekeken over welke code het gaat en waar deze vandaan komt om

33 http://www.codeproject.com/gen/design/CASDesignPatterns.asp?df=100&forumid=45651&select=1652814#xx1652814xx

34 http://msdn2.microsoft.com/en-us/library/930b76w0(VS.80).aspx

vervolgens een gepast veiligheidsbeleid te kiezen.

CAS omvat verschillende elementen, welke we elk zullen uitdiepen:

- Evidence: Identificeren van code
- Permissions: Een toelating tot het uitvoeren van een bepaalde actie of verlening van toegang tot een bepaalde resource
- Permission sets: Verzameling van toelatingen
- Code groups: Groepering van code met welke voldoen aan een specifieke eisen
- Policy: Een hiërarchische structuur van Security Policies

7.2.1 Evidence, het identificeren van Code

Evidence[35] is een verzameling van eigenschappen van een Assembly. Wanneer code uitgevoerd moet worden door de CLR zal deze de Evidence van een Assembly controleren. Hierbij wordt gekeken naar:

- Wie schreef de code
- Van waar komt de code (internet, intranet, ...)
- Van waaruit wordt het uitgevoerd

Met deze informatie kan dan beslist worden welke acties toegelaten zijn en welke resources in welke mate beschikbaar gesteld mogen worden. Het .NET framework bezit een reeks van standaard eigenschappen die gebruikt kunnen worden als Evidence, maar het is ook mogelijk deze lijst uit te breiden met zelf gedefinieerde eigenschappen:

- Application Directorie
  Van waaruit wordt de Assembly uitgevoerd.

- Strong Name
  Werd reeds vermeld, bestaat uit de naam, versie nummer, cultuur informatie, de public key en een hash van de Assembly inhoud. Een Strong Named Assembly kan steeds uniek geïdentificeerd worden.

- Publisher
  Gebruik makend van een certificaat kan de uitgever van de code gecontroleerd worden op een wijze die meer zekerheid bied dan enkel maar Strong Named Assemblies. Via een certificaat kunnen we elke keer opnieuw controleren wie de Publisher van een Assembly is en of de gebruikte sleutel nog geldig is.

- URL
  Is de URL waarvan de Assembly gehaald werd. Dit kan bijvoorbeeld http://pctom/Assemblies/project/ zijn maar ook file://c:/Assemblies/project/ naargelang deze locatie kunnen er permissies gegeven of geweigerd worden.

- Site
  Bevat de naam van de site waarvan de Assembly ingeladen werd. Deze wordt bekomen uit de URL. Het principe is hetzelfde als bij de URL, op basis van de naam kan er beslist worden in welke mate de code vertrouwd kan worden.

35 http://msdn2.microsoft.com/en-us/library/7y5x1hcd(vs.71).aspx

- Zone
  Duid de zone aan waarvan de Assembly gehaald werd. Het .NET framework heeft enkele standaard zones bijvoorbeeld My_Computer_Zone of Internet. Deze zones zijn een vagere omschrijving dan de Site en URL om te weten te komen waar de Assembly vandaan komt.

Wanneer er voldoende Evidence verzameld werd kan er overgegaan worden tot het geven van de nodige rechten voor het uitvoeren van acties en benaderen van resources.

7.2.2 Permissies

Zij vormen de basis van het hele Code Access Security principe. Permissies[36] (Permissions) zijn, zoals de naam het al aangeeft, toelatingen. De CLR zal alleen code toelaten acties uit te voeren wanneer deze de nodige Permissies bezit.

Er bestaan voor het .NET Framework een hele reeks gedefinieerde Permissies zoals

- PrintingPermission: In welke mate het printen toegestaan is
- FileIOPermission: In welke mate toegang tot het bestandssysteem verkregen wordt
- UIPermission: Toelating tot het tonen van een UI en gebruik maken van het clipboard
- ...

7.2.3 Permission Sets

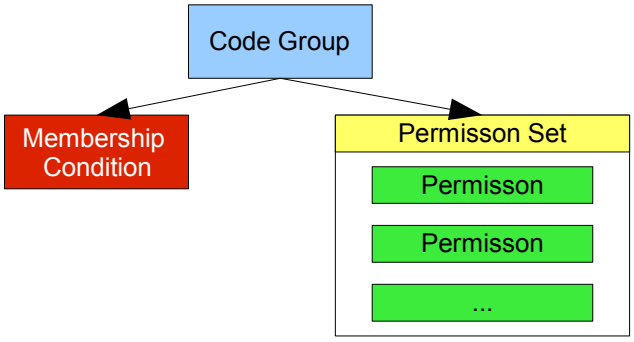
Permissies vormden de basis steen voor het bouwen van een veiligheidsbeleid. Maar het werken met individuele permissies kan omslachtig zijn. Hiervoor bestaan de zogenaamde Permissie Sets. Dit zijn een verzameling van Permissies. Het .NET Framework bezit al een reeks voorgemaakte sets en het is ook mogelijk om zelf een eigen Permissie Set aan te maken. Enkele voorgemaakte Permissie Sets zijn:

- FullTrust: Volledige toegang tot alle resources
- Execution: De code mag uitgevoerd worden, maar krijgt geen toegang tot beschermde resources
- Nothing: Geen toelatingen, code mag niet uitgevoerd worden

7.2.4 Code Groep

Een Code Groep (Code Group) bestaat uit een Permissie Set en een Lidmaatschap Conditie (Membership Condition). De Lidmaatschap Conditie zijn de voorwaarden waaraan de Evidence van code moet voldoen om tot deze Code Groep te behoren. Wanneer code tot een Code Groep behoord krijgt deze de bijhorende Permissie Set.

36 http://www.codeproject.com/dotnet/UB_CAS_NET.asp#Declarative%20vs%20Imperative

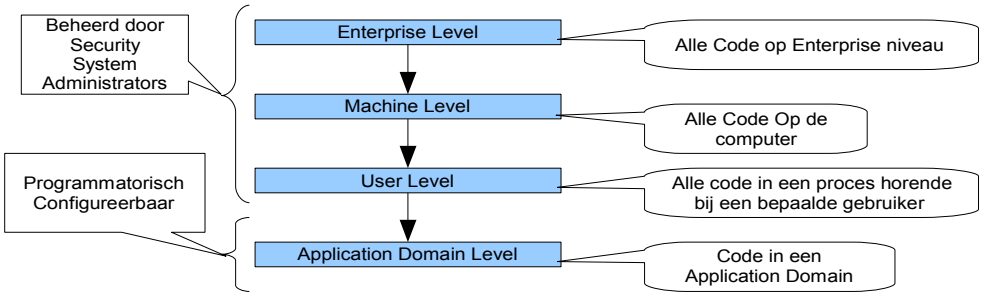


Afbeelding 17: Inhoud van een Code Group

Code Groepen kunnen hiërarchische geordend worden. Een boomstructuur van Code Groepen wordt een Veiligheids Beleid (Security Policy) genoemd.

7.2.5 Veiligheids Beleid

Een Veiligheids Beleid zorgt voor de mapping tussen een Assembly en een Permissie Set op basis van de Evidence van de Assembly. Het .NET Framework bezit vier verschillende Policy Levels:

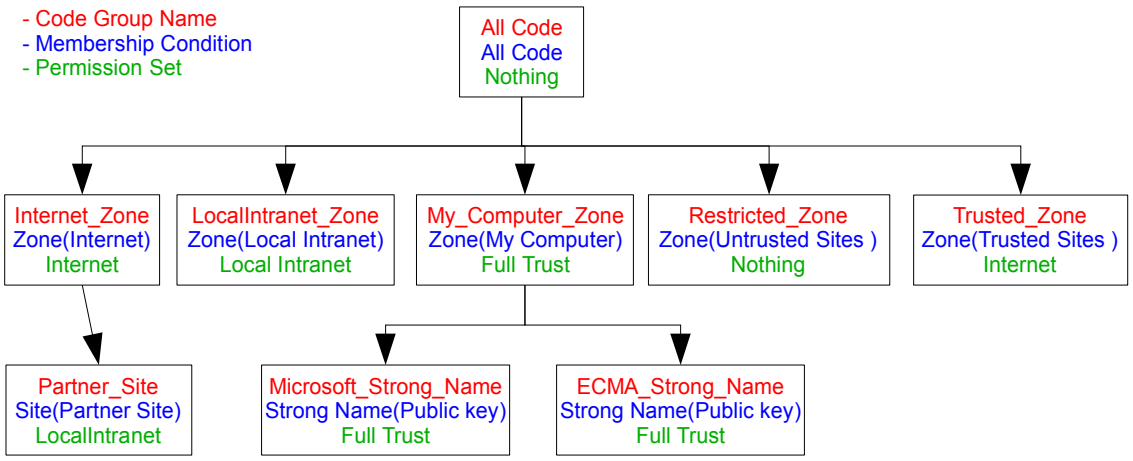


Afbeelding 18: Hiërarchische Structuur van Security Policies

7.2.6 Toekennen van Permissies

Op basis van een enkel Beleid

De basis van een Veiligheids Beleid is een boomstructuur bestaande uit Code Groups. Elke Code Group bevat zoals eerder vermeld een reeks van Permissies, een Permissie Set genoemd en een voorwaarde waaraan voldaan moet worden om tot de Code Groep te behoren.

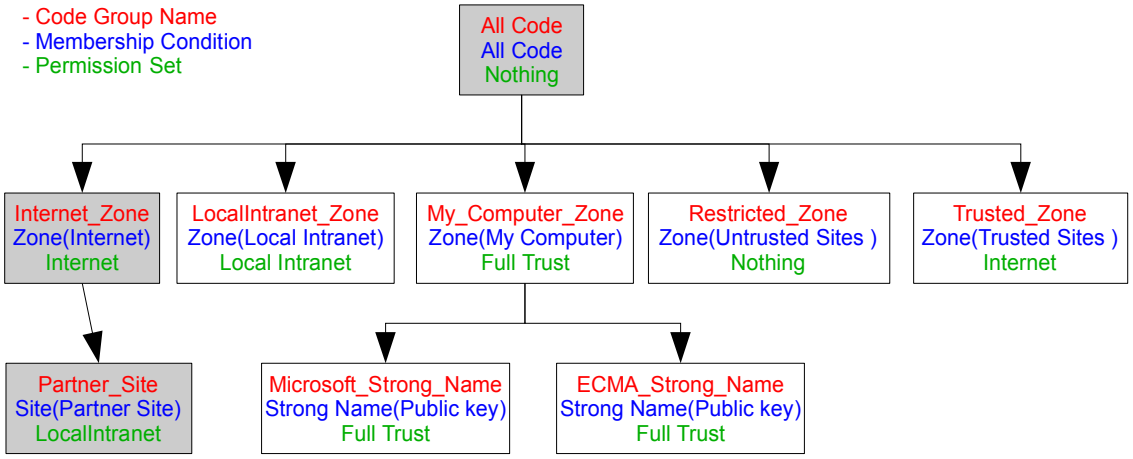


Afbeelding 19: Voorbeeld Machine Policy Level Code Group Hierarchy (http://www.15seconds.com/Issue/040121.htm)

Via deze boomstructuur kan er beslist worden welke permissies gegeven mogen worden. Bij het doorlopen van de boom wordt elke keer gekeken door de CLR tot welke Code Group code kan behoren. Elke Code Group zal zijn set van Permissies geven. Code kan tot verschillende Code Groups behoren. In dat geval wordt er unie gemaakt van de verschillende verkregen Permission Sets.

De CLR zal de boomstructuur doorlopen volgens volgende regels:

- Er wordt begonnen bij de root van de tree
- Wanneer de Membership Condition niet voldaan werd van een Parent Node wordt er niet meer verder gekeken naar zijn Child Nodes.
- Nodes kunnen attributen hebben die het berekenen van de rechten beïnvloeden. Zoals bijvoorbeeld “Exclusive”, in dit geval zal er geen unie gemaakt worden van de verschillende Permissie Sets enkel de Permissie Set van de Exclusive node zal gebruikt worden.

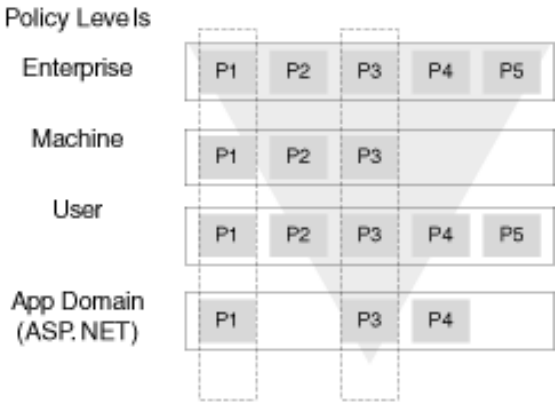


Afbeelding 20: Doorlopen Machine Policy Level Code Group Hierarchy
Verkregen Permissie Set = Nothing + Internet + LocalIntranet

Rekening houdende met de verschillende beleiden

Het is nu duidelijk hoe de Permissies van een enkele Beleid berekend worden. Maar er zijn

meerdere beleiden die tegelijk gebruikt worden. De uiteindelijke Permissie Set die toegekend zal worden, wordt berekend uit de combinatie van de verkregen Permissie Sets van elk Beleids Niveau.
Om een veilig systeem te kunnen garanderen wordt hier niet zomaar een unie van alle permissies gemaakt, maar maakt de CLR een doorsnede van alle Permissie Sets. Enkel de Permissies die aanwezig zijn in elk Beleids Niveau Permissie Set worden uiteindelijk toegekend aan de Code.



Afbeelding 21: Berekenen van Permissies door CLR (MSDN)

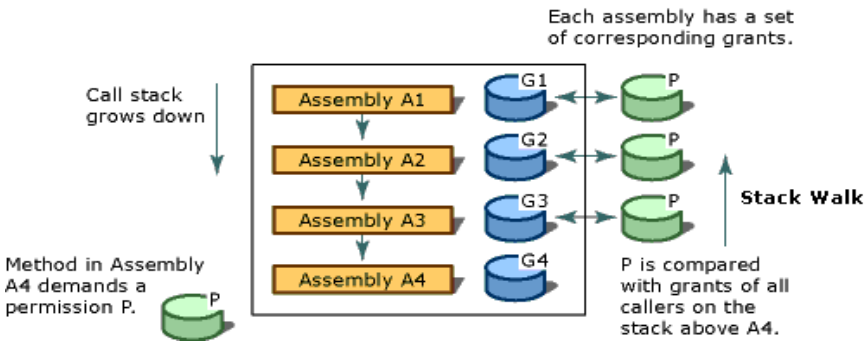
We zullen later bij, de implementatie van het systeem, nog terug op komen Code Access Security, vermits we hiervan gebruik zullen maken voor het beperken van de rechten van Mobiele Applicaties.

Het is nu duidelijk hoe permissies worden toegekend en berekend maar nog niet hoe en wanneer deze worden gecontroleerd.

7.2.7 Controleren van de permissies, Stack Walk

Wanneer code een bepaalde actie uitvoert of een resource benadert wordt er door de CLR gecontroleerd of deze actie op dat moment wel mag uitgevoerd worden door de code. Dit wordt gedaan door een zogenaamde “Stack Walk” [37] [38] uit te voeren. Hierbij gaat de CLR de Call stack aflopen tot men de oorspronkelijk opdrachtgever van de actie bereikt. Bij elke Caller die zich op de stack bevindt, wordt gecontroleerd of deze de nodige permissies heeft voor het uitvoeren van de gevraagde opdracht. Wanneer één van de Callers op de StackWalk niet de nodige permissies bezit zal er een “Security Exception” optreden en wordt de opdracht niet uitgevoerd.

37 http://msdn2.microsoft.com/en-us/library/system.security.istackwalk.aspx
38 http://msdn2.microsoft.com/en-us/library/c5tk9z76(vs.71).aspx



Afbeelding 22: Overzicht Stack Walk (bron: msdn)

Hiermee tracht men “Luring Attacks” te voorkomen. Deze vorm van aanval werd al eerder besproken. Dit houdt in dat onbetrouwbare code probeert een opdracht uit te voeren waarvoor hij geen permissies heeft via een component met meer rechten. Indien er geen Stack Walk zou plaats vinden zou de actie van de onbetrouwbare code zonder problemen uitgevoerd worden via het betrouwbare component vermits deze hiervoor genoeg rechten bezit.

7.3 Encryptie

Als laatste wil ik nog vermelden dat het .NET Framework het mogelijk maakt om een Byte Stream te encrypteren met behulp van de CryptoStream klasse[39] [40]. Dit maakt het mogelijk om afluisteren of “earsdropping” tegen te gaan en zo veilige communicatie mogelijk te maken. Het kan eveneens gebruikt worden om bestanden te encrypteren zodat derden deze niet meer kunnen lezen. Dit kan nodig zijn wanneer de staat van een applicatie gevoelige informatie bezit en bewaard dient te worden op een publiek toegankelijke computer. We kunnen besluiten dat .NET eveneens de mogelijkheden bied om de reeds besproken veiligheidsproblemen omtrent Eavesdropping en Information Disclosure te beperken.

De Cryptostream bied standaard volgende encryptie-methoden aan:

- RSA
- DES
- DSA
- Rijndael
- TripleDES

39 http://www.devtips.net/Artikel.aspx?id=90
40 http://www.codeproject.com/csharp/using_cryptostream.asp

```
FileStream stream = new
FileStream("C:\\test.txt",FileMode.OpenOrCreate,FileAccess.Write);

DESCryptoServiceProvider cryptic = new DESCryptoServiceProvider();

cryptic.Key = ASCIIEncoding.ASCII.GetBytes("BCDEFGH");
cryptic.IV = ASCIIEncoding.ASCII.GetBytes("ABCDEFGH");

CryptoStream crStream = new CryptoStream(stream,
  cryptic.CreateEncryptor(),CryptoStreamMode.Write);


byte[] data = ASCIIEncoding.ASCII.GetBytes("Hello World!");

crStream.Write(data,0,data.Length);

crStream.Close();
stream.Close();
```

Code 12: Encrypteren, voorbeeld uit "The Code Project"

```
FileStream stream = new FileStream("C:\\test.txt",
                        FileMode.Open,FileAccess.Read);

DESCryptoServiceProvider cryptic = new DESCryptoServiceProvider();

cryptic.Key = ASCIIEncoding.ASCII.GetBytes("ABCDEFGH");
cryptic.IV = ASCIIEncoding.ASCII.GetBytes("ABCDEFGH");

CryptoStream crStream = new CryptoStream(stream,
   cryptic.CreateDecryptor(),CryptoStreamMode.Read);

StreamReader reader = new StreamReader(crStream);

string data = reader.ReadToEnd();

reader.Close();
stream.Close();
```

Code 13: Decrypteren, voorbeeld uit "The Code Project"

7.4 Conclusie

Het .NET Framework biedt ons dankzij features als Dynamische laden van code, CAS, Role Based security, Reflection, Serializatie, ... een breed scala aan hulpmiddelen aan voor het ontwikkelen van Mobiele Applicaties. Hierdoor kunnen we gerust zeggen dat het .NET Framework zeker geschikt is voor de ontwikkeling van Mobiele Code Systemen.

8 Implementatie

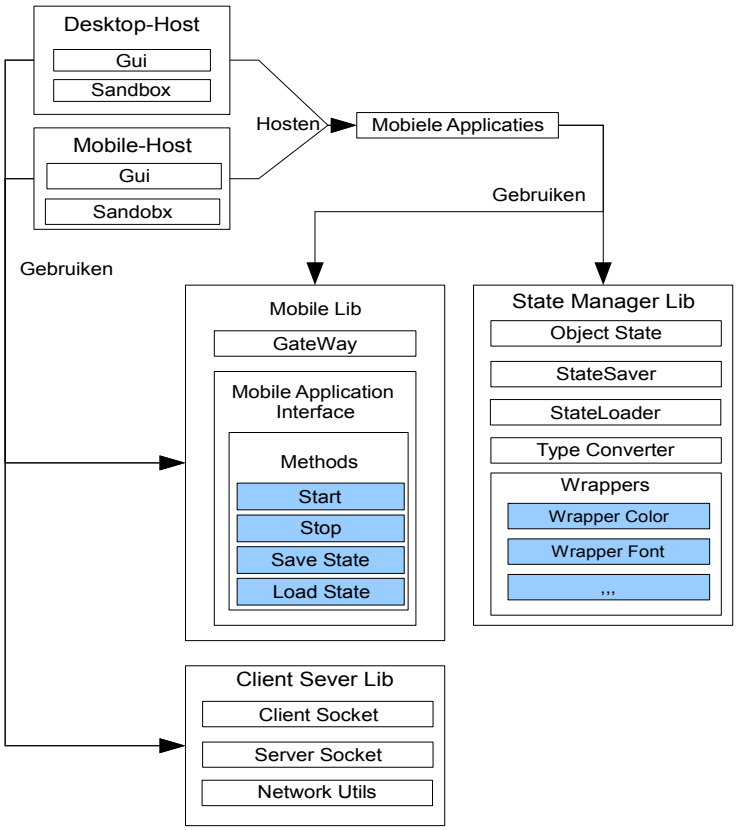
Omdat het belangrijkste doel, die we willen bereiken, het bewaren van de applicatie staat is, werd daardoor hier vooral op gefocust bij de ontwikkeling van het Mobiele Applicatie Systeem. Voordat de finale versie van het Mobiele Applicatie Systeem ontstond, werden er eerst enkele test-versies ontwikkeld om een beter zicht te krijgen van de mogelijkheden op het .NET Framework. Het finale-systeem maakt het mogelijk om een applicatie zijn staat te bewaren, waarna deze samen met de applicatie zelf verzonden kan worden naar een

nieuwe host. Het host-systeem kan uitgevoerd worden op een pda of computer.

8.1 Architectuur

Voordat we van start gaan met het bespreken van de verschillende versies van het Mobiele Applicatie Systeem geven we eerst een beeld van de architectuur van de finale versie van het doel systeem. Dit om makkelijker te kunnen volgen wanneer we de tot stand koming hiervan bespreken.

De uiteindelijke versie van het Mobiele Applicatie Systeem diende uitvoerbaar te zijn op PDA's, smartphones, desktop pc's, ... kortom alle mogelijke apparaten waarop het .NET en .NET Compact Framework kan draaien. Hierdoor moesten er twee host-systemen ontwikkeld worden, één Compact Framework en één .NET Framework versie. Buiten de host-systemen werden er ook verschillende bibliotheken ontwikkeld. De belangrijkste is de “State Manager” bibliotheek. Deze staat in voor het bewaren en laden van de applicatie staat. Daarnaast is er ook een bibliotheek die het netwerk gedeelte van het systeem afhandelt, genaamd “ClientServerLib” en een bibliotheek met als naam “Movable” welke een Mobiele Applicatie interface en GateWay bevat. We zullen later op elk onderdeel dieper ingaan.



Afbeelding 23: Overzicht Architectuur

Na dit overzicht van de uiteindelijke architectuur van het Mobiele Applicatie Systeem zullen we nu overlopen hoe we hiertoe gekomen zijn, welke problemen we hierbij tegen kwamen en voor welke oplossingen gekozen werd.

8.2 Delegeren verantwoordelijkheid

In de eerste versie van het Mobiele Applicatie Systeem lag de verantwoordelijkheid, voor het bewaren en laden van de applicatie staat, volledig bij de programmeur van de Mobiele Applicatie. Hierdoor werd het het complexe probleem vermeden waarbij de staat van een onbekende applicatie via een generieke methode moest kunnen worden bewaard en ingeladen.

Voor het bewaren van de staat moesten de Mobiele Applicaties de methode “SaveState” uit de “Mobile Appliction” Interface implementeren. Dit betekende dat de nodige code geschreven moet worden voor het bewaren en laden van de staat van de Mobiele Applicatie. Hoe en onder welke vorm een mobiele applicatie deze staat bewaarde, was van geen belang, zolang hij hiertoe in staat was. Het Host Systeem melde aan de Mobiele Applicatie dat deze zijn staat moest opslaan via het oproepen van de “SaveState” methode. Nadat de Mobiele Applicatie zijn staat bewaard had konden beide doorgezonden worden naar het volgende Host Systeem. Het laden van de staat gebeurde op gelijkaardige manier, hiervoor moesten de Mobiele Applicaties de methode “LoadState” implementeren. In de latere versies moeten deze methodes nog steeds geïmplementeerd worden, maar zal de programmeur een bibliotheek tot zijn beschikking hebben waardoor dit simpeler wordt. We zullen hier later dieper op in gaan.

Door op deze manier te werken kan de programmeur van de Mobiele Applicatie zelf bepalen welke gegevens dienen bewaard te worden om de staat-herstelling mogelijk te maken. Wanneer het bijvoorbeeld enkel nodig was om de staat van een enkele variabele bij te houden, kon de programmeur ervoor kiezen enkel deze te bewaren. Deze manier van werken zal bijna altijd de beste performantie geven, vermits er zelf beslist kan worden wat er bewaard moet worden en wat niet. Hierdoor vermijdt men de verwerking van nutteloze gegevens wat later optreedt bij de meer generieke oplossing. Het resultaat van de “SaveState” Methode kan dan ook een zeer klein bestand bestand opleveren. Dit komt de verwerking en verzending ervan ten goede komt.

Elke methode van werken heeft ook zijn nadelen en deze methode is hier geen uitzondering op:

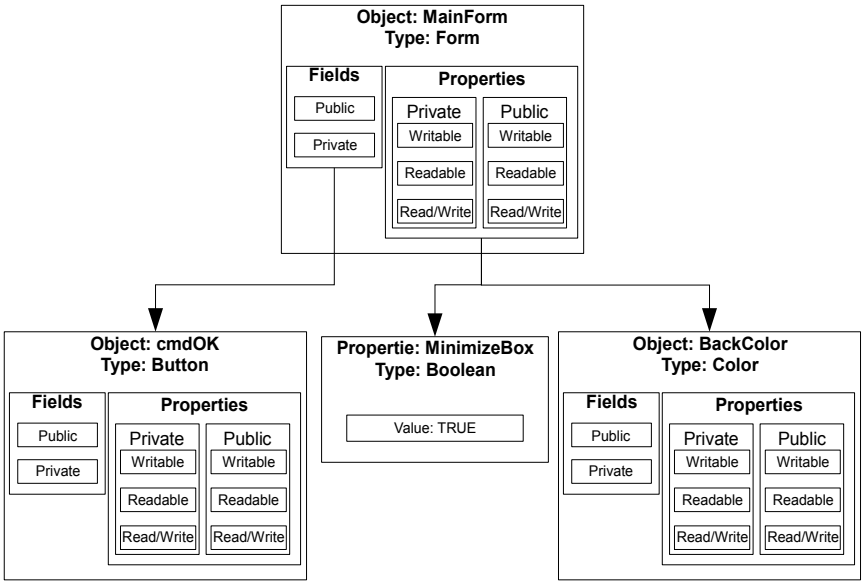
- Complexiteit Mobiele Applicatie
  Eén van de nadelen is dat Mobile Applicaties complexer worden. Ze moeten niet alleen de code bevatten voor het uitvoeren van hun taken, maar ook nog eens de code die hun de mogelijkheid moet geven tot het bewaren en laden van de staat. Elke bijkomstige code vergroot de kans op fouten, hoe minder code de Mobiele Applicatie moet bevatten hoe beter.

- Geen consistentie bij opslaan van de staat
  Doordat de programmeur van de de Mobiele Applicatie zelf verantwoordelijk is voor het bewaren van zijn staat moet deze zelf het wegschrijven en inlezen hiervan implementeren. Dit betekent dat elke applicatie zijn staat op zijn eigen manier zal opslaan.

Het zou veel interessanter zijn moesten de ontwikkelaars van mobiele applicaties zich geen zorgen hoeven te maken over het bewaren en laden van de staat. Een generieke en uniforme bibliotheek die instaat voor het het bewaren en laden van de staat leek een veel beter idee.

8.3 Ontwikkelen van een generieke methode

De eerste stap tot het realiseren van een generieke methode voor het bewaren van een applicatie staat, is het bewaren van de staat van een object. Zoals eerder besproken bezit het .NET Framework hier de mogelijkheid toe door middel van Serializatie. Het probleem dat deze methode met zich mee brengt, is dat zowel de Binaire, Soap als XML Serializer niet zomaar om het even welk object type kan serializeren. Toch zou het ook mogelijk moeten zijn om de staat van niet-serializeerbare objecten te kunnen bijhouden.



Afbeelding 24: overzicht inhoud probleem types

Zoals afgebeeld op bovenstaande afbeelding, bestaat de staat die we willen bewaren van een object uit velden (fields) en eigenschappen (properties). We kunnen opmerken dat de velden en eigenschappen privaat (private) of publiek (public) kunnen zijn, we hebben al besproken dat dit van belang kan zijn bij de XML-serializatie van het .NET Framework. Daar komt nog bij dat de we bij eigenschappen een onderscheid kunnen maken tussen de schrijfbare (writable) en leesbare (readable) en ook degene die zowel lees- als schrijfbaar zijn. Ook dit is van belang bij de XML-serializatie zoals eerder besproken.

De waarden van deze velden en eigenschappen kunnen primitieve types zijn zoals Integers die geen serializatie problemen opleveren. Maar het is ook mogelijk dat een lid (veld of eigenschap) bestaat uit een complex type die zelf ook weer uit meerdere onderdelen bestaat. Zoals eerder vermeld moeten alle velden en eigenschappen van het object dat men wil gaan serializeren ook serializeerbaar zijn. Zo niet zal de serializatie niet lukken.

Rekening houdende met deze feiten werd een test systeem ontwikkeld waarbij we zelf een XML-Serializatie algoritme schreven en een systeem waarbij gebruik gemaakt werd van Binaire Serializatie. Na deze twee versies was er voldoende informatie verzameld om een

finale versie te creëren die zowel op het .NET Compact Framework als op de normale versie van het .NET Framework kon werken. Hiervoor werd gebruik gemaakt van de standaard xml-serializer.

We zullen nu de eerste methode bespreken, namelijk de zelf geschreven XML-Serialization

8.3.1 Eigen XML-serialization

Hoewel het .NET Framework een xml-serializer bezit, is er in de de eerste instantie toch voor gekozen om zelf een versie te schrijven. Dit omdat met de standaard XML-serializatie niet alle complexe types geserializeerd kunnen worden. Voorbeelden hiervan zijn het type “Forms” die simpelweg geweigerd wordt door de xml-serializer of het type “Color”die wel geserializeerd wordt, maar zijn staat verliest. Met behulp van Reflectie en het XmlDoucment type van het .NET Framework probeerden we hier een oplossing voor te vinden.

Het XmlDocument maakt het mogelijk om XML-documenten op te bouwen in het geheugen en later weg te schrijven via een XmlWriter. Doordat het xml-document opgebouwd wordt in het geheugen kunnen we makkelijk gebruik maken van recursie wat het algoritme voor het doorlopen van objecten eenvoudiger maakt. Moest het XML-bestand lijn per lijn opgebouwd en weggeschreven worden zou dit voor veel meer problemen gezorgd hebben. Via reflectie worden de velden en eigenschappen van het te serializeren object overlopen om de nodige data te bekomen. Wanneer alle leden van het object overlopen zijn, kunnen we de bekomen XML-structuur naar een bestand wegschrijven.

Het overlopen van de objecten verloopt in volgende stappen:

1. De naam en het type van het object wordt opgezocht via reflectie en weggeschreven als XML-attribuut voor de XML-representatie van het object.

2. Vervolgens wordt onderzocht door middel van een TypeConvertor of het type van het object converteerbaar is naar een string.

3. Indien het object niet converteerbaar is door de TypeConvertor, doorlopen we alle publieke en private velden van het object door middel van Reflectie. Elk gevonden veld wordt eveneens gecontroleerd of het converteerbaar is. Indien dit zo is kan men de waarde ervan rechtstreeks wegschrijven als textnode in het xml-document

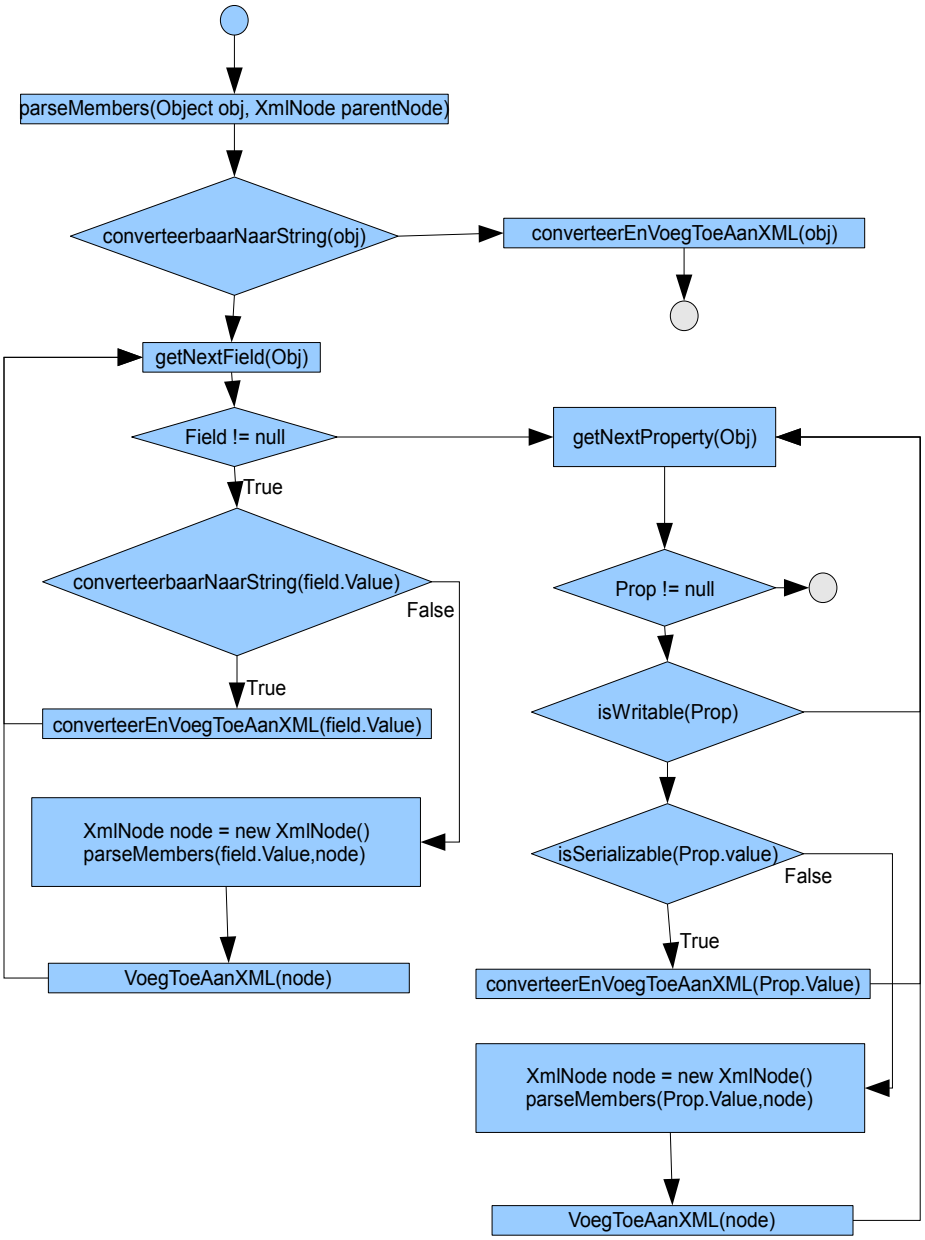
4. Wanneer een veld niet converteerbaar is, doorlopen we recursief de vorige stappen opnieuw, totdat het veld en al zijn velden en eigenschappen eveneens geconverteerd kunnen worden.

5. Na het onderzoeken van de velden doen we hetzelfde voor de eigenschappen. Bij het overlopen van de eigenschappen wordt nog een extra controle uitgevoerd, namelijk of de eigenschap schrijfbaar is. Wanneer dit niet zo is, kunnen we hiervan later de staat niet herstellen. Het heeft dus ook geen zin om deze data te bewaren.

Het gebruik van de TypeConvertor bespaarde veel tijd. Deze maakte het mogelijk om converteerbare objecten onmiddellijk weg te schrijven als een tekst-node in het xml-document. De TypeConvertor biedt eveneens een functie aan om types te instantiëren op

basis van tekstuele data. Dit zorgde ervoor dat, bij het inladen van de staat, deze op een eenvoudige manier terug geconverteerd kon worden naar zijn oorspronkelijk vorm. Zonder TypeConvertor zou, van elk object die geserializeerd moet worden, alle leden volledig doorlopen moeten worden. Dit kan een tijdsintensieve operatie kan zijn. Spijtig genoeg is deze functie niet beschikbaar op het .NET Compact Framework. Hierdoor zullen we in de finale versie van het Mobiele Applicatie Systeem een andere methode gebruiken.

We merken ook dat deze generieke methode waarschijnlijk vaak informatie zal opslaan, die niet hoefde bewaard te worden voor de het herstellen van de Applicatie Staat. Dit is het nadeel die een generieke methode met zich mee kan brengen tegenover een gespecialiseerde oplossing.



Afbeelding 25: Vereenvoudigde flow voor het vullen van een Object_State object

Het resultaat van de zelfgeschreven XML-Serialisator is een xml-document waarbij alle

publieke en private velden beschreven staan. Van de eigenschappen worden alleen deze beschreven die zowel lees- als schrijfbaar zijn, voor de eerder genoemde rede. Deze manier van werken zorgt ervoor dat de staat van een niet serializeerbaar object toch nog bewaard kan worden, hetzij niet helemaal.

Afbeelding 26: Resultaat custom xml-serialisatie

Voor het laden van de staat werd het XML-bestand overlopen waarbij de opgeslagen data weer werd toegekend aan een nieuwe instantie van een object. Dit werd gedaan met behulp van reflectie.

In het XML-Bestand werden de namen van de velden en eigenschappen van een object opgehaald. Bij elke gevonden naam werd dit lid opgezocht in de instantie van het object met behulp van reflectie. Daarna konden de juiste waarden, afgeleid uit de informatie van het XML-bestand, toegekend worden.

```
FieldInfo fieldinfo = obj.GetType().GetField(FieldName, BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.DeclaredOnly);

                if (fieldinfo != null)
                {
                    fieldinfo.SetValue(xmlValue, childstate.Value);
                }
```

Code 14: Het ophalen en instellen van een veld in een object

Door deze methode in een bibliotheek te steken, kunnen programmeurs van Mobiele

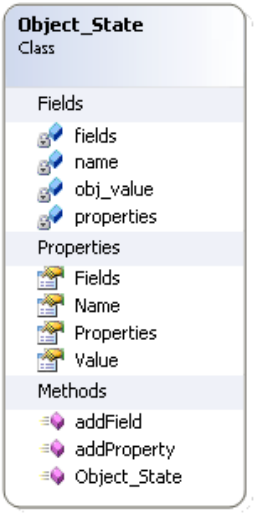
Applicaties hiervan gebruiken maken zodat ze zelf geen code meer hoeven te schrijven voor het bewaren en laden van de staat.

8.3.2 Binaire Serializatie

Hoewel binaire serializatie niet mogelijk is op het .NET Compact Framework en we deze techniek niet in het uiteindelijk Mobiele Applicatie Systeem zullen kunnen gebruiken, is er toch voor gekozen een test-versie te ontwikkelen die hiervan gebruik maakt. Dit om een beter beeld te krijgen van Binaire Serializatie in de praktijk.

Binaire Serializatie kan veel serializeren maar nog steeds niet alles. Hiervoor moest een oplossing gevonden worden, indien we willen op een deftige manier de staat van een Mobiele Applicatie kunnen bijhouden. Bij de eigen xml-serializer konden we alle eigenschappen en velden van een probleem-type via reflectie overlopen en de nodige data rechtstreeks naar een XML-structuur in het geheugen wegschrijven. Bij Binaire serializatie ligt dit iets moeilijker. Hier is het niet zomaar mogelijk om zelf alle data lijntje per lijntje weg te schrijven in een binair bestand of structuur in het geheugen.

Om dit probleem te omzeilen, creëerden we een nieuw type die we als “buffer” gebruiken zodat we toch via reflectie een object lid per lid konden overlopen en de waarden ervan opslaan. De nodige data werd bewaard in een instantie van dit nieuwe type die we zo opstelden dat deze wel serializeerbaar was. Het type noemden we een Object_State.



Afbeelding 27: Object_State

We zien in het bovenstaande overzicht van het Object_State dat deze bestaat uit verschillende velden die overeen komen met de gegevens die bewaard dienen te worden bij de serializatie van een object. Het opvullen van de serializatie-buffer gebeurt gelijkaardig als bij de zelf geschreven xml-serializer, zij het met enkele kleine verschillen:

1. De naam van het object wordt opgezocht via reflectie en weggeschreven naar het veld “name” in het Object_State object.

2. Daarna wordt er gecontroleerd of het object serializeerbaar is door het controleren van het “IsSerializable” eigenschap. Indien dit zo is kan het object weggeschreven

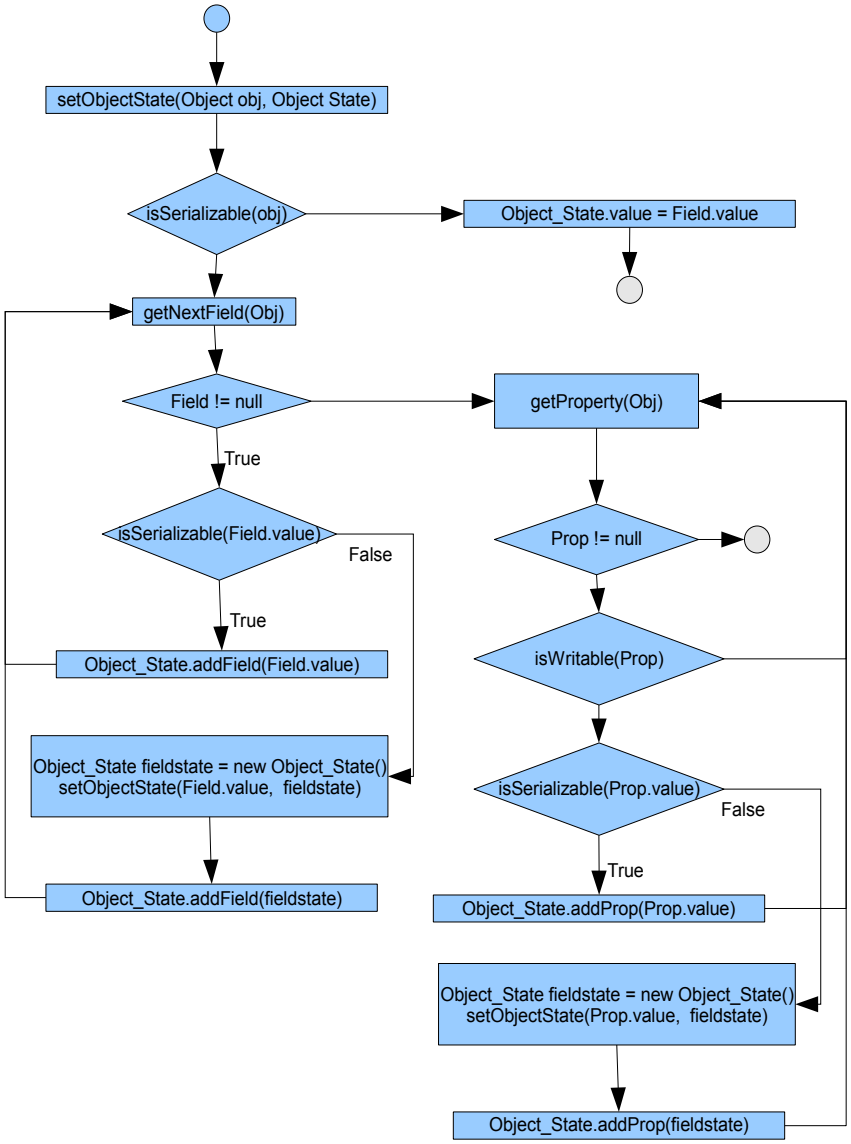
worden naar het veld “obj_value” in het Object_State object. Wanneer dit zo gebeurt, is verder onderzoek niet meer nodig.

3. Indien het object niet serializeerbaar is, doorlopen we alle publieke en private velden van het object door middel van Reflectie. Elk gevonden veld wordt eveneens op de mogelijkheid van serializatie gecontroleerd. Indien het serializeerbaar is, kan men de waarde ervan, in het lijstje “fields” van het Object_State object, rechtstreeks wegschrijven.

4. Wanneer een veld niet serializeerbaar is, doorlopen we recursief de vorige stappen opnieuw totdat het veld en al zijn leden eveneens geserializeerd kunnen worden. Het resultaat, een Object_State object, van deze procedure wordt dan weggeschreven in het lijstje “fields” van het Object_State object horende bij het te serializeren object.

5. Na het onderzoek van de velden, doen we hetzelfde voor de eigenschapen. Ook hier wordt de extra controle op schrijfbaarheid uitgevoerd. Dit voor dezelfde reden die bij de bespreking van de vorige versie van het systeem gegeven werd.

Van zodra al deze stappen doorlopen werden, hebben we een compleet Object_State object die we kunnen doorgeven aan de Binaire Serializer om deze weg te laten schrijven naar een binaire bestand. Op deze manier kunnen we toch nog complexe typen serializeren, die normaal geweigerd zouden worden door de Binaire serializatie van het .NET Framework. Een ander bijkomend voordeel is dat we ons niet meer bezig hoeven te houden onder welke vorm we de data zullen wegschrijven. Bij de zelf geschreven XML-serializatie was het nodig zelf te beslissen onder welke vorm we de data gingen bewaren.



Afbeelding 28: Vereenvoudigde flow voor het vullen van een Object_State object

Bij verschillende testen konden we bevestigen wat het onderzoek van Binaire Serializatie ons al had geleerd. Namelijk dat deze vorm als resultaat een compacter bestand opleverde dan de xml-versie voor dezelfde data. Ook zorgde de functie “IsSerializable” ervoor dat vele objecten niet volledige doorlopen moesten worden. Deze konden direct bewaard worden in het Object_State object. Een nadeel die we ondervonden was dat we het binaire bestand niet konden gebruiken ter analyse wanneer er iets fout gelopen was. Bij de XML-versie hadden we hiermee geen probleem. We konden makkelijk ontdekken waar een bepaald probleem zich voor deed, door het bestand te lezen. Ook dit lag in de lijn van de verwachte nadelen bij Binaire Serializatie.

8.3.3 .NET XML serializatie

Het schrijven van een eigen xml-serializer bleek wat omslachtig voor het gebruik in het uiteindelijke systeem. Dit omdat we geen gebruik konden maken van de ConvertTo functie, daar deze niet beschikbaar is voor het .NET Compact Framework. Hierdoor konden enkel

primitieve types makkelijk geconverteerd worden naar een string ter gebruik in het xml-document. De tweede mogelijkheid, Binaire Serializatie was eveneens niet mogelijk, omdat deze eveneens niet beschikbaar is voor het .NET Compact Framework. Daarom kozen we ervoor om gebruik te maken van de XLM-serializatie van het .NET Framework. Deze is gelukkig wel beschikbaar op de .NET Compact versie.

Het grote voordeel bij gebruik van XML-Serializatie was dat we zelf niet meer hoefden code te schrijven voor het genereren en lezen van een xml-document. Hoewel het gebruik van XML-Serializatie enig gemak met zich mee bracht zijn er toch ook enkele problemen. Niet alle types werden even goed geserializeerd. Dit omdat de XML-serializatie enkel publieke velden en geschikte eigenschappen kan verwerken. Hierdoor zal een type, bijvoorbeeld als “System.Drawing.Color” wel geserializeerd worden door de XML-serializatie, maar zal het resultaat geen data bevatten. Hierdoor kon er niet achterhaalt worden welk kleur het object voorstelde. Het type Color bezit namelijk enkel private velden en “read-only” eigenschappen. Hiervoor moest een oplossing gevonden worden.

Wanneer we nu een type willen XML-serializeren zorgen we ervoor dat deze eerst naar een voor de XML-serializatie handelbaar type geconverteerd wordt. Hiervoor wordt gekeken naar het type van het te converteren object. Daarna wordt gekeken of er een wrapper voor bestaat. Dit is een klasse die dezelfde data als het object bevat, maar onder een vorm die de XML-serialitie beter kan serializeren.

```
public class Wrapper_Color
{
    private Color original_color;

    public Wrapper_Color ()
    {
        original_color = Color.Black;
    }

    public Wrapper_Color(Color kleur)
    {
        original_color = kleur;
    }

    [XmlAttribute("Argb")]
    public int Argb
    {
        get { return original_color.ToArgb();}
        set { original_color = Color.FromArgb(value); }
    }

    [XmlIgnore]
    public Color Original_color
    {
        get { return original_color; }
        set { original_color = value; }
    }
}
```

Code 15: Voorbeeld van Wrapper Class

De wrapper zal, zoals aangeraden op MSDN[41], ervoor zorgen dat de nodige data beschikbaar gemaakt wordt voor de xml-serializer. Dit zal in het geval van het type

41 http://msdn2.microsoft.com/en-us/library/ms950721.aspx

“System.Drawing.Color” betekenen dat er een eigenschap aangemaakt wordt waarmee men een kleur kan ophalen en ingeven. Zoals eerder vermeld, was dit voordien niet mogelijk doordat alle velden privaat waren en de eigenschappen alleen konden uitgelezen worden.

```
public static object convertToXMLserializable(Object obj)
{
    Object result = obj;

    switch (obj.GetType().ToString())
    {
       case "System.Drawing.Color":
             result = new Wrapper_Color((Color)obj);
          break;

       case "System.Drawing.Font":
          result = new Wrapper_Font((Font)obj);
          break;


       ,,,,,,,,,,,,,,,,,,,,

    }
    return result;
 }
```

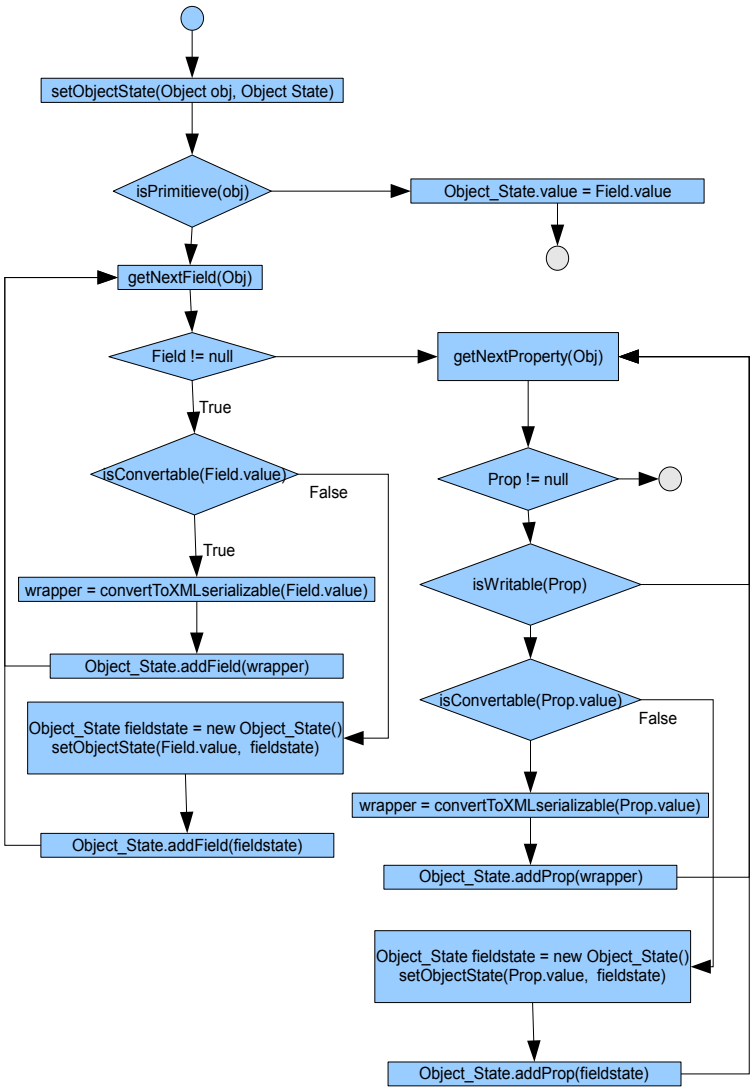
Code 16: Convertor

Natuurlijk is het onmogelijk om elk probleem-type te anticiperen. We kunnen wel wrappers schrijven voor de meest voorkomende types, maar de op maat gemaakte types kunnen we onmogelijk kennen. Toch willen we van deze eveneens de staat kunnen opslaan. Daarbij zijn er nog types die gewoonweg niet geserializeerd kunnen worden door de xml-serializer, zoals bijvoorbeeld het type “System.Windows.Forms”. Voor deze beide problemen gaan we op dezelfde manier te werk als bij de binaire serializatie vermeld werd. Het object wordt met behulp van Reflectie volledig doorlopen. Van elke lid wordt gecontroleerd of deze converteerbaar is naar een type die de xml-serializer correct kan serializeren. Wanneer dit zo is, kan de waarde van het lid onmiddellijk in het Object_State object bewaard worden. Indien dit niet zo is, dan wordt het lid verder, door middel van Reflectie, ontleed.

De wrappers kunnen dus in twee gevallen voor een verbetering zorgen:

1. Het object is niet XML-serializeerbaar en zou moeten volledig doorlopen worden:
   Door een geschikte wrapper te schrijven, kan men nu onmiddellijk alle nodige data beschikbaar stellen aan de xml-serializer, zonder dat het object door ons algoritme volledig doorlopen moet worden.

2. Het object is XML-serializeerbaar maar bezit geen geschikte velden en eigenschappen:
   De xml-serializer zou, wanneer er geen wrapper gemaakt werd voor het betreffende type, de data niet correct kunnen verwerken. Dit heeft het verlies van de staat van het object als gevolg. De Wrapper zorgt ervoor dat de nodige velden en eigenschappen beschikbaar gemaakt worden voor de xml-serializer zodat deze zijn werk correct kan uitvoeren.



Afbeelding 29: Vereenvoudigde flow voor het vullen van een Object_State object geschikt voor XML-serializatie

8.3.4 Behoud van de Object Graaf

Er is nog een probleem waar we het niet over gehad hebben. Wanneer we een object serializeren wordt hiervan een boomstructuur opgemaakt, maar de structuur van het werkelijke object is een graaf. Hierbij kunnen verschillende objecten verwijzen naar eenzelfde object of verwijst een object naar zichzelf. We hebben op dat moment te maken met dubbele en circulaire referenties. Deze dienen behouden te worden indien we een object correct willen serializeren en deserializeren. Wanneer we hier geen rekening mee houden kan het algoritme voor het serializeren in een oneindige lus komen te zitten. Er moet gezorgd worden dat dubbele en circulaire referenties worden herkend en dat deze op een correcte manier afgehandeld worden.

De oplossing, waarvoor gekozen werd bij onze serializator, is het bijhouden van een lijst van alle objecten die reeds onderzocht werden. Elke maal een nieuw object onderzocht

dient te worden, zal gecontroleerd worden of deze niet verwijst naar een object die reeds voorkomt in deze lijst. Wanneer dit het geval is, doorlopen we het object niet meer verder. Op deze manier voorkomen we een oneindige lus.

Hiermee lossen we het oneindig lus probleem op. Toch zijn nog niet alle problemen van de baan. We verliezen nog informatie, namelijk de link tussen het oorspronkelijk object en alle referenties ervan. Dit wordt opgelost door middel van een speciale wrapper. Wanneer een dubbele referentie ontdekt wordt, creëren we een referentie wrapper. Deze wrapper wordt dan als waarde, voor het betreffende object, opgeslagen in het Object_State object. De wrapper bevat de fully qualified name van het object waarnaar verwezen wordt. Hierdoor kunnen we het object ondubbelzinnig refereren.

In het voorbeeld hieronder zien we een mooi voorbeeld hiervan. De eigenschap DesktopBounds, van het form type, bestaat uit verschillende onderdelen, waaronder de locatie. Een form type heeft daarnaast nog een andere eigenschappen, zoals bijvoorbeeld DesktopLocation. Deze stelt hetzelfde voor als de Locatie bij DesktopBounds eigenschap. De serializator herkent dat deze informatie reeds opgeslagen werd en reageert hier ook op. Het object wordt niet meer volledig doorlopen en er wordt een referentie als waarde weggeschreven. Deze referentie is, zoals eerder vermeld, de fully qualified name van het object waar we naartoe willen verwijzen.

```
− <Property Name="DesktopBounds">
    <Value xsi:nil="true"/>
  − <Properties>
    + <Property Name="Location"></Property>
    + <Property Name="Size"></Property>
    + <Property Name="X"></Property>
    + <Property Name="Y"></Property>
    + <Property Name="Width"></Property>
    + <Property Name="Height"></Property>
    </Properties>
  + <Fields></Fields>
  </Property>
− <Property Name="DesktopLocation">
    <Value xsi:type="Wrapper_Reference" Reference="Init.DesktopBounds.Location"/>
    <Properties/>
    <Fields/>
  </Property>
```

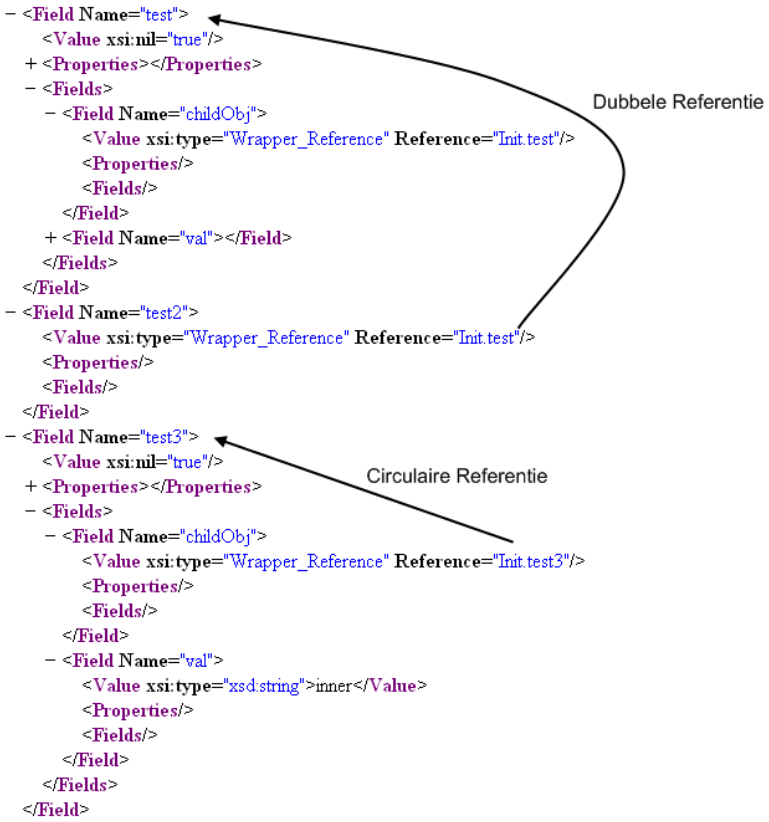
Afbeelding 30: Resultaat bij serializeren van een Form

Doordat we de referentie nu bijhouden verliezen we geen informatie wanneer er dubbele en circulaire referenties voorkomen in de object graaf. Bij het herstellen van de objectstaat zullen met behulp van reflectie het juiste object ophalen en linken wanneer we een referentie-wrapper tegen komen.

```
TestObject test;
TestObject test2;
TestObject test3;

test = new TestObject("testing");
test2 = test;
test.ChildObj = test2;

test3 = new TestObject("inner");
test3.ChildObj = test3;
```

Code 17: Aanmaken van dubbele en circulaire referentie

Wanneer we de objecten hierboven laten serializeren krijgen we volgende resultaat:




```
- <Field Name="test">
    <Value xsi:nil="true"/>
  + <Properties></Properties>
  - <Fields>
    - <Field Name="childObj">
        <Value xsi:type="Wrapper_Reference" Reference="Init.test"/>
        <Properties/>
        <Fields/>
      </Field>
    + <Field Name="val"></Field>
    </Fields>
  </Field>
- <Field Name="test2">
    <Value xsi:type="Wrapper_Reference" Reference="Init.test"/>
    <Properties/>
    <Fields/>
  </Field>
- <Field Name="test3">
    <Value xsi:nil="true"/>
  + <Properties></Properties>
  - <Fields>
    - <Field Name="childObj">
        <Value xsi:type="Wrapper_Reference" Reference="Init.test3"/>
        <Properties/>
        <Fields/>
      </Field>
    - <Field Name="val">
        <Value xsi:type="xsd:string">inner</Value>
        <Properties/>
        <Fields/>
      </Field>
    </Fields>
  </Field>
```

Afbeelding 31: XML resultaat van dubbele en circulaire referenties

We merken dat het object “test3” naarzichzelf verwijst en object “test2” naar het object “test”. De XML is zo opgesteld dat makkelijk leesbaar is voor een mens. We kunnen direct herkennen wanneer we met een referentie te maken hebben en we zien eveneens naar welk object gerefereerd wordt. Door gebruik te maken van een fully qualified name kunnen we zelfs makkelijk zelf een referentie aanpassen of toevoegen.

8.3.5 Conclusie

Door de ontwikkeling van de zelfgeschreven xml-serializator en door het gebruik van de Binaire Serializator deden we ervaringen op, die ten goede kwamen bij het ontwikkelen van de uiteindelijke versie van het Mobiele Applicatie Systeem. Grote hulp hierbij was het gebruik van het Object_State object (uit de Binaire Serializatie versie) en het algoritme voor het overlopen van een object (gebruikt in de eigen xml-serializer).

De uiteindelijke eigen serializator is robuust en zal altijd proberen een object zo goed mogelijk te serializeren en deserializeren. Ook al is het te serializeren object complex en is er geen wrapper voorhanden. Het serializeren gebeurd volledig transparant en vereist geen extra attributen zoals bij de standaard .NET serializatie soms wel het geval is. Het kan ook overweg met dubbele en circulaire referenties, waardoor de Object Graaf kan behouden worden na serializatie.

Een nadeel is wel de snelheid. Door het vele gebruik van reflectie is de snelheid verre van optimaal. Het bewaren van de Object Graaf zou eventueel performanter gemaakt kunnen worden door het gebruik van Reflectie te vermijden. Een gelijkaardig serializatie-project welke hierin wel slaagt is Xstream.NET.

8.4 Xstream.NET

Xstream.NET is een xml-serializer, die grote gelijkenissen heeft met onze manier van werken bij het bewaren van de applicatiestaat. Het kan, zoals bij onze Serializer, objecten serializeren welke normaal niet serializeerbaar zijn. Xstream.NET is een port van Xstream, welke in Java ontwikkeld werd. Omdat onze Serializer in .NET geschreven is, ligt het voor de hand dat we deze vergelijken met een andere .NET implementatie. Dit opdat de verschillen of gelijkenissen makkelijk herkenbaar zijn.

8.4.1 Werking

Xstream.NET stelt zelf zijn xml-document op en laat deze niet generen door de .NET xml-serializator. Er wordt dus gewerkt zoals we in de eerste versie van ons systeem geprobeerd hebben. De xml wordt zelf opgebouwd naarmate men een object doorloopt. In Xstream.NET wordt enkel gekeken naar de velden van een object en niet de eigenschappen.

Het overlopen van de objecten verloopt in volgende stappen:

1. Er wordt gecontroleerd of er een convertor bestaat voor het specifieke type van het object. Een convertor kan van een gekend type rechtstreeks de xml-structuur generen. Op deze wijze moet er geen gebruik gemaakt worden van reflectie en hoeft het object niet meer verder overlopen te worden.

2. Indien er geen convertor voor het type bestaat worden alle publieke en private velden van het object door middel van Reflectie doorlopen. Voor elk gevonden veld wordt eveneens gecontroleerd of er een convertor beschikbaar voor is. Indien dit zo is kan hier ook direct de xml-structuur gegenereerd worden door deze convertor.

3. Wanneer er voor het type van een veld geen convertor beschikbaar is, doorlopen we recursief de vorige stappen opnieuw totdat voor het veld en al zijn velden eveneens een convertor gevonden wordt.

4. Het object is volledig doorlopen en kan weggeschreven worden naar een xml-bestand.

8.4.2 Vergelijking

Door gebruik te maken van convertors kan overmatig gebruik van reflectie voorkomen worden. Enkel wanneer er geen convertor beschikbaar is gaat men over tot het overlopen

van het object met behulp van reflectie. Wanneer voldoende convertors beschikbaar zijn, zal het gebruik hiervan beperkt blijven. Reflectie is een krachtige techniek, maar eveneens een zeer trage. Het vermijden van het gebruik ervan komt de snelheid van de serializatie zeker ten goede.

```
public class DoubleConverter : IConverter
{
    private static readonly Type __type = typeof( double );

    public void Register( MarshalContext context )
    {
        context.RegisterConverter( __type, this );
        context.Alias( "double", __type );
    }

    public void ToXml( object value, FieldInfo field, XmlTextWriter xml, MarshalContext context )
    {
        context.WriteStartTag( __type, field, xml );
        xml.WriteString( ( (double) value ).ToString( "r" ) );
        context.WriteEndTag( __type, field, xml );
    }

    public object FromXml( object parent, FieldInfo field, Type type, XmlNode xml, MarshalContext context )
    {
        return double.Parse( xml.InnerText );
    }
}
```

Code 18: Xstream.NET: Convertor

Een ander belangrijk feit is dat er rechtstreeks weggeschreven wordt naar een xml-structuur in het geheugen. In ons systeem wordt, zoals eerder uitgelegd, gewerkt met een "buffer". Deze buffer zal, na het overlopen van het te serializeren object, door de standaard .NET xml-serializator geserializeerd worden. De xml-serializator maakt eveneens gebruik van Reflectie om de nodige data te bekomen. Dit betekend dat er bij ons systeem twee maal zwaar gebruik gemaakt wordt van Reflectie. Eerste maal voor het opvullen van de buffer en de tweede maal wanneer de xml-serializator de buffer serializeert. Zoals eerder aangekaart is Reflectie een trage techniek. Hierdoor zal de snelheid van ons systeem eronder lijden. Xstream heeft van dit probleem geen zo'n last.

We hebben het al verschillende malen gehad over de convertors, gebruikt in Xstream.NET. Deze zorgen voor een snelheidswinst maar vergen wel veel programmeerwerk. Zelfs al gebruikt men geen uitgebreide collectie van convertors, toch zal men nog steeds voor elk basis type één moeten schrijven. Wordt dit niet gedaan zal de serializatie van een object niet goed lukken. In ons systeem hoeven we ons hier geen zorgen om maken. De xml-serializator zal alle basis typen en zelfs meer voor ons serializeren en voor probleem gevallen kunnen we een wrapper schrijven. Deze manier van werken spaart ons wat werk uit, maar is natuurlijk niet erg performant.

Het afhandelen van dubbele en circulaire referenties worden in Xstream.NET bijna gelijkaardig opgevangen. Alleen hier ook weer op een meer performantere manier. In ons systeem wordt voor het refereren van een object, de fully qualified name gebruikt. Bij het herstellen van de object graaf zal op basis van deze naam, via Reflectie, het nodige object opgehaald worden. Ook hier wordt weer gebruik gemaakt van Reflectie, wat de snelheid niet ten goede komt. Bij Xstream.NET wordt er bij het opbouwen van het xml-bestand een

object-lijst opgebouwd. Wanneer er verwezen moet worden naar een object die reeds onderzocht werd, zal de index van het object in de lijst gebruikt worden als referentie. Bij het herstellen van de object graaf, bouwt Xstream.NET deze zelfde lijst weer op. Wanneer er een referentie naar een object gevonden wordt in het xml-bestand, hoeft niet via reflectie naar het juist object gezocht worden, maar kan deze direct opgehaald worden aan de hand van zijn index uit de object-lijst. Snelheid is hier weer het voordeel. Toch moet gezegd dat de methode gebruikt in ons systeem zeker niet nutteloos is. Het resultaat is veel makkelijker leesbaar voor een mens, dan de aanduiding van een index. Men kan nu zeer makkelijk lezen naar welk object een ander object refereert. Ook is het aanpassen van een referentie in het xml-bestand zelf op deze manier zeer simpel.

8.4.3 Conclusie

Wanneer we naar de mogelijkheden kijken van Xstream.NET en onze serializator, kunnen we concluderen dat beide redelijk gelijkaardig zijn. Met het verschil dat Xstream.NET een veel performantere oplossing biedt. Bij ons systeem wordt de de performantie genekt door het overmatig gebruikt van Reflectie. We kunnen ook zeggen dat Xstream.NET door het zelf opstellen van zijn xml-document, flexibeler is dan onze serializator. Dit doordat men bij Xstream.NET geen rekening hoeft te houden met de beperkingen en eisen van de .NET xml-serializator. Daartegen over staat dat bij Xstream.NET veel werk gestoken moet worden in het ontwikkelen van de convertors, maar dit is zeker het overwegen waard wanneer men een performante flexibele serializator wil creëren.

8.5 Communicatie

Het is nu mogelijk de staat van een applicatie te bewaren. Maar hiermee is nog niet alles wat nodig is voor het Mobiele Applicatie Systeem, geïmplementeerd. Wanneer we terugblikken naar het begin van deze thesis, meer bepaald naar beide Use Cases, blijkt dat het bewaren van de staat niet de enigste vereiste was. Het moet ook mogelijk zijn om een een mobile applicatie en zijn staat naar een nieuwe host te zenden.

Een methode, die hiervoor zou kunnen gebruikt worden, is Remoting. Maar ook hier doet zich hetzelfde probleem voor, namelijk dat Compact versie van het .NET framework hiervoor geen ondersteuning biedt. Daardoor maken we gebruik van Sockets. Zowel het .NET Framework als het .NET Compact Framework bieden de mogelijkheid aan om netwerk applicaties met deze techniek te ontwikkelen. Dit stelde ons in staat om een “Peer to Peer” systeem te ontwikkelen. Hierbij kan, zowel de desktop versie als de PDA versie, een Mobiele Applicatie en zijn staat verzenden en ontvangen.

Zoals eerder besproken, kunnen Assemblies alle nodige code en resources van een applicatie bevatten. We zullen dan ook gebruik maken van deze Assemblies om Mobiele Applicaties te verzenden.

We lezen de gepaste Assembly in via een FileReader. Deze zetten we om naar een Byte Array, zodat ze verzonden kan worden over een socket-connectie, opgezet tussen de originele en toekomstige host.

```
FileStream s1 = new FileStream(asmName + ".dll", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(s1);
byte[] bytesRead = br.ReadBytes((int)s1.Length);
```

Code 19: Converteren van een File naar een Byte Array

Bij aankomst wordt de Byte Array naar een bestand weggeschreven. Deze kan dan later, op het gewenste tijdstip, ingeladen worden. Maar hier doet zich een probleem voor. Een Assembly kan namelijk niet ingeladen worden als de bestandsnaam ervan niet overeenkomt met de naam die in het manifest opgegeven is. Bij het verzenden van het Assembly-bestand, onder de vorm van een Byte Array, verliezen we deze informatie. Het is dus noodzakelijk dat we eveneens de naam apart verzenden, zodat de Assembly onder de juiste naam weggeschreven kan worden.

Het apart verzenden van de naam kan voorkomen worden, wanneer we de Assembly op de oude host inladen en het verkregen Assembly Object naar een Byte Array converteren. Deze Byte Array zenden we dan naar de nieuwe Mobiele Applicatie Host, die deze weer converteert naar een Assembly Object. Met dit Assembly object kunnen we via reflectie toch nog de naam te weten komen. Ook hier doet zich opnieuw het probleem voor dat de Compact versie van het Framework te beperkt is en deze methode van werken niet ondersteunt.

```
private byte[] ObjectToByteArray(Object obj)
{
    if (obj == null)
        return null;
    BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    bf.Serialize(ms, obj);
    return ms.ToArray();
}

private Object ByteArrayToObject(byte[] arrBytes)
{
    MemoryStream memStream = new MemoryStream();
    BinaryFormatter binForm = new BinaryFormatter();
    memStream.Write(arrBytes, 0, arrBytes.Length);
    memStream.Seek(0, SeekOrigin.Begin);
    Object obj = (Object)binForm.Deserialize(memStream);
    return obj;
}
```

Code 20: Converteren van objecten en Byte Array's

In het bovenstaande voorbeeld zien we dat voor het converteren van een object naar een Byte Array, een BinaryFormatter vereist is. Helaas is dit niet aanwezig op het .NET Compact Framework, waardoor we hier geen gebruik van kunnen maken.

Voor het verzenden van staat, was eveneens het oorspronkelijk idee dat het Object_State object rechtstreeks verzonden zou worden naar de nieuwe host door deze te converteren naar een Byte Array. Maar zoals we al reeds weten, zal dit niet lukken voor het .NET Compact Framework. Om de staat te verzenden, moet men een omweg maken. Dit doen we door het Object_State object eerst weg te schrijven naar een XML-bestand, dat we wel kunnen converteren naar een Byte Array. De ontvangende host dient de Byte Array opnieuw weg te schrijven naar een XML bestand, dat later voor het herstellen van de staat gebruikt kan worden.

We kunnen nu de code, resources en staat van een Mobiele Applicatie verzenden. Maar dit moet nog op een gestructureerde manier gebeuren. Hier werd een simpel protocol voor ontwikkeld:

1. De host die een Mobiele Applicatie wil verzenden, neemt contact op met de nieuwe host en maakt een verbinding.

```
IPAddress ipAddress = IPAddress.Parse(address);
IPEndPoint endPoint = new IPEndPoint(ipAddress, port);
Socket _socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
_socket.BeginConnect(endPoint, _connectCallback, null);
```

Code 21: Maken van verbinding met een server via TCP Sockets

2. Wanneer de verbinding tot stand gekomen is, zendt de oude host de naam van de Assembly eerst en wacht op reactie van de nieuwe host.

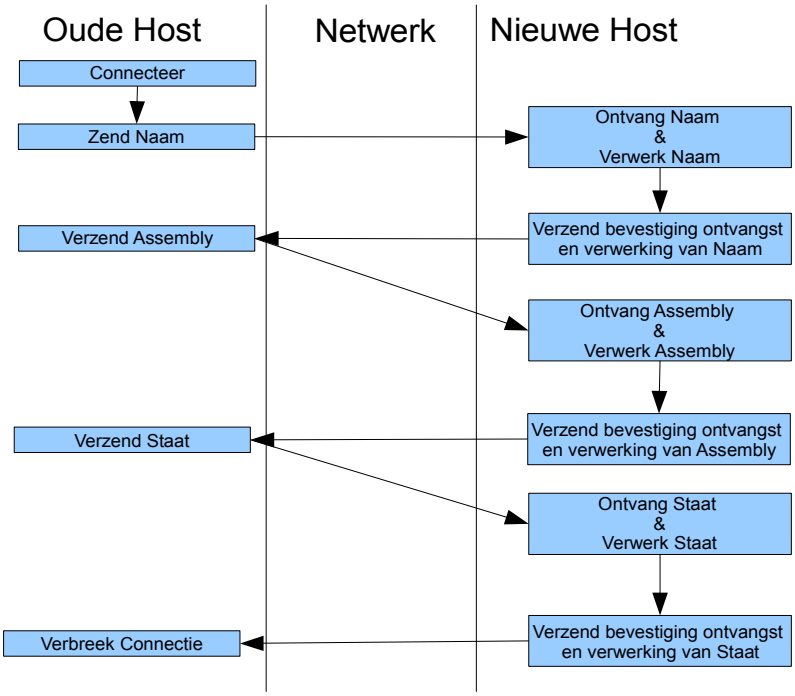
```
Encoding unicode = Encoding.Unicode;
byte[] unicodeBytes = unicode.GetBytes(asmName);
```

Code 22: String omzetten naar een Byte Array

3. Wanneer de bevestiging, namelijk dat de naam verwerkt is, ontvangen wordt, zendt de oude host de Assembly naar de nieuwe host. Hierna wordt opnieuw op de volgende bevestiging gewacht.

4. Wanneer de bevestiging, omtrent de Assembly, ontvangen wordt door de oude host, kan deze het laatste onderdeel verzenden, namelijk de staat. Na het verzenden hiervan, wordt weer op bevestiging gewacht.

5. Wanneer de bevestiging omtrent de staat ontvangen werd, is het verzenden van alle nodige data voltooid. De connectie kan nu afgebroken worden.



Afbeelding 32: Overzicht werking protocol

Er rest ons nog één punt die behandeld moet worden, namelijk de veiligheidsmaatregelen.

8.6 Veiligheid

We hebben al eerder besproken dat het .NET Framework verschillende maatregelen, voor het verhogen van de veiligheid van Mobiele Code Systemen, aanbiedt. Voor onze applicatie hebben we gebruik gemaakt van Applicatie Domeinen en CAS (Code Access Security).

We zijn nu op een punt gekomen waarbij de implementatie van het host systeem op het Compact Framework niet meer dezelfde mogelijkheden als de desktop versie kan aanbieden. Dit komt omdat het .NET Compact Framework geen CAS[42] ondersteuning heeft. Alle code op het Compact Framework krijgt het volledig vertrouwen (Full Trust) van de CLR. Hierdoor werd er enkel gekeken naar het desktop systeem voor de implementatie van de Applicatie Domeinen en CAS. Er moet opgemerkt worden dat dit geldt voor alle versies van het .NET Compact Framework tot en met versie 2. Microsoft beweert dat latere versies wel elementaire ondersteuning voor CAS zullen aanbieden.

8.6.1 Gebruik van Applicatie Domein

Om gebruik te kunnen maken van Code Access Security zonder dat de host applicatie hiervoor moeten worden beperkt in zijn mogelijkheden, moeten we ervoor zorgen dat alle Mobiele Applicaties in een apart Applicatie Domein worden uitgevoerd. Zoals eerder besproken, zorgen Applicatie Domeinen voor een veilige scheiding tussen verschillende Applicaties binnen een zelfde proces. Hoewel het .NET Compact Framework het gebruik van Applicatie Domeinen ondersteunt, is deze ondersteuning voor het gebruik in ons

42 http://msdn2.microsoft.com/en-us/library/13s3wxyw(vs.80).aspx

systeem, te beperkt.

Hoewel het gebruik van een Applicatie Domein op het eerste zicht relatief simpel lijkt, blijkt het toch niet zo evident te zijn. Wanneer we willen voorkomen dat de Assembly van een Mobiele Applicatie ingeladen wordt in het Applicatie Domein van de host, moeten we op enkele zaken letten.

Wanneer data uitgewisseld wordt tussen verschillende Applicatie Domeinen, kunnen we drie situaties onderscheiden:

1. De data is Binaire Serializeerbaar
   Er wordt een kopie gemaakt van de data via Binaire Serializatie in het ene Applicatie Domein en verzonden naar het ander Applicatie Domein. Bij ontvangst zal de data gedeserialiseerd moeten worden en hier ligt juist het probleem. Voor het deserialiseren van de data, is de exacte Assembly vereist waarin het type van de verkregen data gedefinieerd is. Hierdoor zal de host automatisch proberen de gebruikte Assembly in te laden. Wanneer er niet opgelet wordt, betekend dit dat het verzenden van een type, afkomstig uit een Assembly van het ene domein, ervoor zorgt dat de ontvanger hiervan deze Assembly inlaadt. Dit willen we ten alle kosten vermijden in het Mobiele Applicatie Systeem.

2. De data is “*remotable”*
   *Dit houdt in d*at de klasse van de data afgeleid is van “System.MarshalByRefObject”, dit betekent in dat er geen er kopie van de data gemaakt wordt, maar een referentie ernaar. Hierdoor moet de data zelf niet verzonden worden. Hoewel er geen data verzonden hoeft te worden, zal de ontvanger nog steeds het gerefereerde type moeten kennen. Daarom zal nog steeds de nodige Assembly ingeladen worden, indien dit nog niet zo was. Hetzelfde geldt hier ook, dit willen we ten alle tijden vermijden bij het gebruik ervan in het Mobiele Applicatie Systeem.

3. De data is noch serializeerbaar, noch remotable
   De data zal tussen de twee domeinen niet uitgewisseld kunnen worden.

```
Assembly Suspect = appDomain.Load(MobileAppName);
// Use Assembly
```

Code 23: Foutief

Wanneer we bovenstaand voorbeeld gebruiken, kunnen we in de problemen komen. De Load-functie zal de Assembly laden in het Applicatie Domein die de functie opriep. De reden waarom het hier zal mis lopen is dat het resultaat van de Load-functie een serializeerbaar type is, afkomstig uit het andere domein. Dit betekent dat het verkregen Assembly Type bij ontvangst gedeserializeerd zal moeten worden. Zoals eerder uitgelegd, zal hierdoor de host eveneens diezelfde Assembly moeten inladen. Hierdoor zal deze in het Applicatie Domein van de host geladen blijven, tot deze zelf afgesloten wordt.

Voor het Mobiele Applicatie Systeem is het essentieel dat bij het laden en gebruiken van een Mobiele Applicatie, dat deze niet geladen wordt in het Applicatie Domein van de host zelf. De oplossing hiervoor is het creëren van een soort proxy. Deze zal instaan voor het laden en uitvoeren van de Mobiele Applicatie in een tijdelijk Applicatie Domein. Via deze

proxy kunnen we communiceren met de Mobiele Applicatie vanuit het Host Applicatie Domein, zonder dat deze “besmet” raakt met de Assembly van de Mobiele Applicatie.

De eerste stap hiervoor is het ontwikkelen van de Proxy Assembly. Deze zal dan geladen worden in het tijdelijke Applicatie Domein van de host, dat voor het hosten van een Mobiele Applicatie gecreëerd werd. De Proxy Assembly zal bij het inladen in het tijdelijke Applicatie Domein ook geladen worden in de host zijn Applicatie Domein. Dit om de reeds eerder genoemde reden. Maar in dit geval is dit niet erg. Voor elke Mobiele Applicatie die ingeladen dient te worden, zal de Proxy toch weer moeten geladen worden. Deze mag dus standaard mee geladen worden bij het starten van het host-systeem. De Proxy klasse moet van “System.MarshalByRefObject” afgeleid worden, zodat we een referentie kunnen krijgen naar de instantie van het proxy object in een andere Applicatie Domein. Dit voorkomt dat het volledig object doorgegeven wordt, wat als gevolg heeft dat de Mobiele Applicatie, geladen in de proxy, toch weer terecht komt in het applicatie domein van de host.

Voor het verkrijgen van de referentie van de Proxy instantie, in het tijdelijke applicatie domein, gebruiken we de functie “CreateInstanceAndUnwrap”. Deze functie zorgt ervoor dat een instantie van het gewenste type uit een Assembly aangemaakt wordt, binnen het applicatie domein waarin de Assembly geladen werd. Als resultaat geeft de functie een referentie terug naar de instantie binnen het gebruikte applicatie domein.

```
Proxy proxy = (Proxy)appDom.CreateInstanceAndUnwrap("Movable",
typeof(Proxy).FullName);
```

Code 24: Gebruik de "CreateInstanceAndUnwrap" functie

We kunnen deze referentie nu gebruiken in het host applicatie domein om functies, gedefinieerd in het Proxy Type, aan te roepen. De Proxy implementeert eveneens de Mobile_Applicatie Interface waarover we het al gehad hebben, plus de methode “LoadAssembly” met als parameter de naam van de in te laden Assembly. Enkel de naam moet mee gegeven worden, omdat bij de creatie van het tijdelijke applicatie domein, al aangegeven werd waar gezocht moet worden naar in te laden Assemblies. Hiervoor geven we een pad op in het “ApplicationBase” eigenschap.

```
public void loadAssembly(string assemblyName)
{
   Assembly mobileApp = Assembly.Load(assemblyName);

   bool gevonden = false;
   foreach (Type t in mobileApp.GetTypes())
   {
      foreach (Type iface in t.GetInterfaces())
      {
         if (iface.Equals(typeof(Movable_Application)))
         {
            plugin = (Movable_Application)Activator.CreateInstance(t);
            gevonden = true;
         }
      }
   }

   if (!gevonden)
      throw new Exception("Not a Movable Application");
}
```

Code 25: "loadAssembly" mehtode

Zoals we kunnen zien in het stukje code hierboven, zal deze functie ervoor zorgen dat een Assembly geladen wordt. We merken op dat in de Proxy zelf geen gebruik gemaakt wordt van applicatie domeinen. Dit hoeft ook niet meer, de Proxy zit al in een applicatie domein waardoor de code van het Host Applicatie Domein gescheiden blijft.

Na het laden van de Assembly maken we een instantie aan van de klasse in deze, die de Mobile_Applicatie interface implementeert. Dit zodat we de Mobiele Applicatie kunnen besturen. We weten, als programmeur van de host applicatie, bijna niks van de Mobiele Applicaties. Er is enkel geweten dat deze een klasse moet bezitten die de Mobiele Applicatie interface implementeert. Daarom moeten we de juiste klasse zien te vinden in de Assembly van de Mobiele Applicatie. Dit doen we door alle types hierin te overlopen, totdat we deze vinden die de Interface implementeert. Hiervan maken we een instantie, plugin genaamd.

```
foreach (Type t in mobileApp.GetTypes())
        {
           foreach (Type iface in t.GetInterfaces())
           {
              if (iface.Equals(typeof(Movable_Application)))
              {
                 plugin = (Movable_Application)Activator.CreateInstance(t);
                 gevonden = true;
              }
           }
        }
```

Code 26: Op zoek naar de juiste klasse

Wanneer de juiste klasse gevonden werd en hiervan een instantie werd gemaakt, kunnen we de Mobiele Applicatie bevelen geven. We kunnen hem starten, stoppen of opdragen zijn staat te laden of bewaren.

```
public void start()
{
   plugin.start();
}

public void stop()
{
   plugin.stop();
}

public void SaveState()
{
   plugin.SaveState();
}

public void SaveState(string name)
{
   plugin.SaveState(name);
}

public void loadState()
{
   plugin.loadState();
}

public void loadstate(string path)
{
   plugin.loadstate(path);
}
```

Code 27: Implementatie van de "Mobile_Application" interface in de proxy klasse

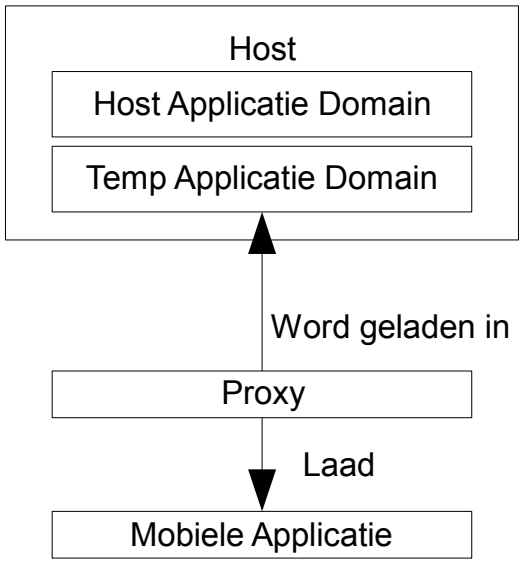
In het code voorbeeld 27 zien we de implementatie van de Mobile_Application interface in de Proxy klasse. Hieruit blijkt dat de Proxy, zoals zijn naam het al aangeeft, alle opdrachten rechtstreeks doorgeeft aan de geïnstantieerde Mobiele Applicatie. In het Host Applicatie Domein roepen we de Proxy nu aan. We geven hem de opdrachten, die door de Mobiele Applicatie uitgevoerd moeten worden.

Door op deze manier te werken, voorkomen we dat de Assembly van een Mobiele Applicatie geladen wordt in het Applicatie Domein van de host.

```
proxy.stop();
AppDomain.Unload(tempApp);
```

Code 28: stoppen en verwijderen van een Mobiele Applicatie

Wanneer de Mobiele Applicatie niet meer gebruikt hoeft te worden, kunnen we het tijdelijke Applicatie Domein verwijderen. Wanneer we een Applicatie Domein verwijderen, worden alle Assemblies, geladen in dit domein, eveneens verwijderd. Hierdoor zal de Mobiele Applicatie uit het geheugen verwijderen worden.



Afbeelding 33: Overzicht laden van Mobiele Applicatie

8.6.2 Gebruik van Code Access Security

We hebben nu een goed werkend systeem voor het gebruik van Applicatie Domeinen. Hierdoor is het niet meer zo moeilijk om code, in een tijdelijk Applicatie Domein, te beperken in zijn acties. We hebben al besproken wat CAS inhoudt, maar nog niet hoe we deze in onze implementatie gebruiken. We moeten hiervoor, bij de creatie van een Applicatie Domein, een “Permission Set” als parameter meegeven. In deze “Permission Set” staan alle acties die de code in het Applicatie Domein mag en niet mag uitvoeren.

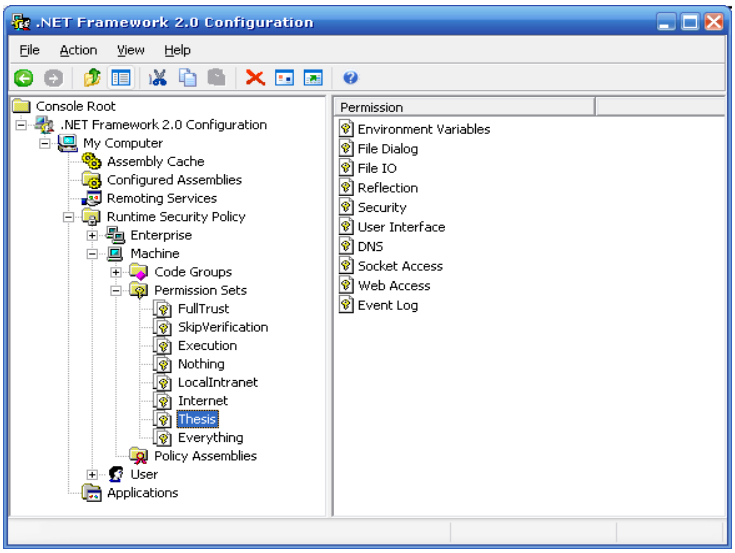
Permissies sets kunnen in de code gedefinieerd worden, maar dit kan heel wat lijnen code vergen. Daarom gebruiken we, in onze implementatie, voorgemaakte Permissie Set die reeds op het Systeem staat.

```
//Niks toelaten
PermissionSet set = new PermissionSet(PermissionState.None);
//Permisies aanmaken
SecurityPermission p1 = new SecurityPermission(SecurityPermissionFlag.Execution);
FileIOPermission p2 = new FileIOPermission(FileIOPermissionAccess.Read,@"C:\");
//Permisies toevoegen aan de set
set.AddPermission(p1);
set.AddPermission(p2);
```

Code 29: Voorbeeld aanmaken permissie set "at runtime"

In het bovenstaande voorbeeld maken we een permissies set aan. Wanneer deze aan een Applicatie Domein gegeven wordt, zal alle code (die hierin uitgevoerd wordt) enkel leesrecht hebben tot de C-schijf.

Voor het Mobiele Applicatie Systeem werd een Permissies Set aangemaakt, met behulp van de .NET framework configuratie console, en toegevoegd aan het systeem.



Afbeelding 34: .NET Configuratie Console

Op deze manier kunnen we de Permissie Set makkelijk aanpassen, zonder dat de Mobiele Applicatie Host opnieuw gecompileerd moet worden. Dit zou het geval zijn indien de Permissie Set in de code aangemaakt werd.

8.6.3 Conclusie

Er wordt rekening gehouden met eventuele veiligheidsrisico's in ons Mobiel Applicatie Systeem, maar zeker niet met alles. We beschermen de host momenteel zoveel mogelijk tegen kwaadaardige Mobiele Applicaties. Dit door de acties, die de Mobiele Applicaties mogen uitvoeren, te beperken. De mobiele applicaties zelf blijven kwetsbaar voor malafide hosts. Ook wordt geen gebruik gemaakt van encryptie, wat het afluisteren van Hosts en Agents makkelijk maakt. De staat wordt eveneens ongeëncrypteerd opgeslagen, wat het systeem ongeschikt maakt voor het gebruiken van informatie gevoelige applicaties op openbare computers. Er zijn reeds maatregelen omtrent deze problemen bespreken en we hebben gemerkt dat het geen probleem is deze te implementeren met behulp van het .NET Framework. Door gebruik te maken van encryptie en digitaal ondertekende Agents is het mogelijk ons systeem uit te breiden zodat het een veiliger geheeld vormt.

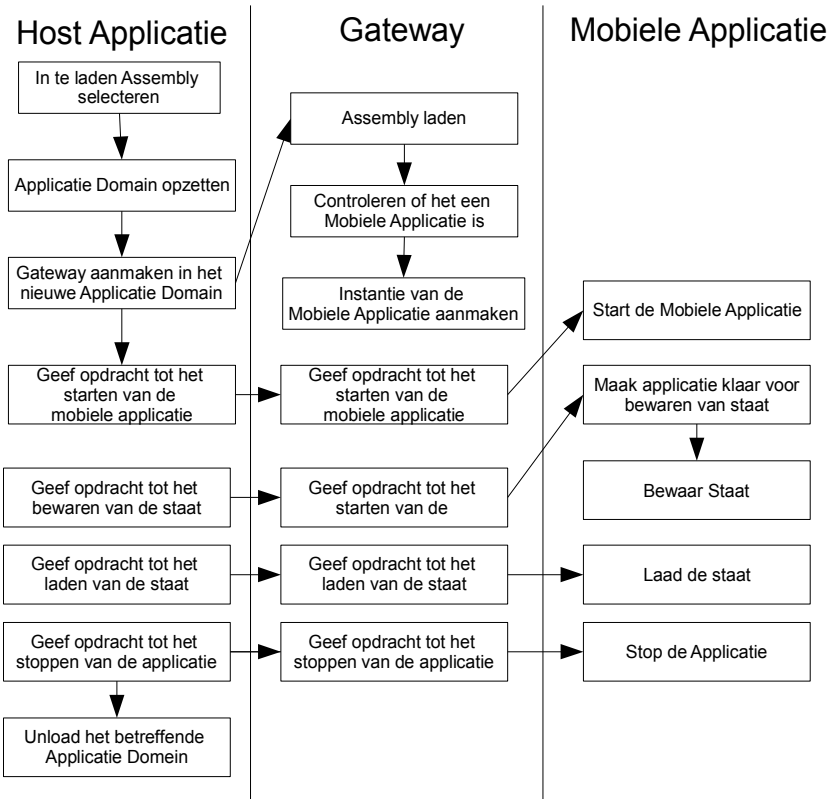
8.7 Overzicht werking

In het volgende overzicht zien we hoe een Mobiele Applicatie geladen wordt op een host. Dit begint met het kiezen van de Assembly die de Mobiele Applicatie bevat. Hierna wordt op de host een Applicatie Domain aangemaakt. Vervolgens wordt in het nieuwe Applicatie Domein een Gateway aangemaakt. Deze staat, zoals eerder uitgelegd, in als proxy tussen de host en de Mobiele Applicatie. De Gateway zal op het moment van inladen controleren of de opgegeven Assembly Effectief een Mobiele Applictie is. Wanneer dit niet zo is, zal het laden van de Assembly gestopt worden en wordt er een foutmelding gegeven.

Eenmaal het gelukt is om de Mobiele Applicatie te laden kan nu, via de gateway, aan deze opdrachten gegeven worden. Deze opdrachten kunnen zijn:

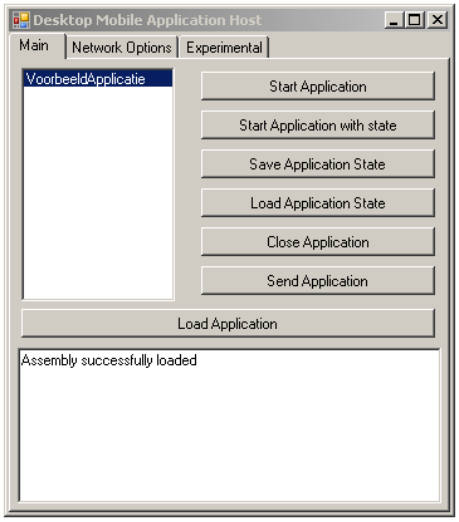
- Starten van de Applicatie
- Bewaren van de Applicatie staat
- Het herstellen van de Applicatie Staat

- Stoppen van de Applicatie



Afbeelding 35: Overzicht werking: starten, bewaren, laden en stoppen

We zullen nu een reëel voorbeeld doorlopen hoe dit alles in zijn werk gaat.



Afbeelding 36: Desktop Host met beschikbare Mobiele Applicatie

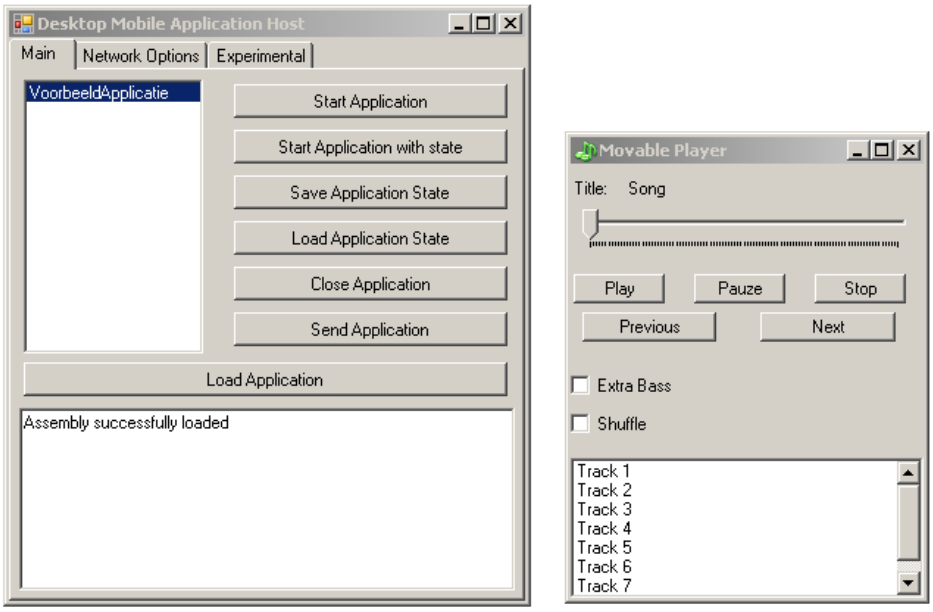
We beginnen met het starten van de host-applicatie op een desktop pc. Deze pc is verbonden met het netwerk via een netwerkkabel.

Eenmaal de host gestart is, laden we een Mobiele Applicatie in. We hebben deze simpelweg de naam “VoorbeeldApplicatie” gegeven.

Zoals we kunnen zien op het info gedeelte van de host-applicatie, is de mobiele applicatie succesvol geladen. Dit betekend dat er een gateway aangemaakt werd in een Applicatie Domain. De gateway in het aparte Applicatie Domain heeft vervolgens de opgegeven Mobiele Applicatie ingeladen.

We kunnen nu de Mobiele Applicatie Starten. Dit doen

door in de host-applicatie op “Start Application” knop te klikken. Dit commando wordt doorgegeven naar de gateway in het Applicatie Domein van de geselecteerde Mobiele Applicatie. De gateway zal vervolgens de Mobiele Applicatie de opdracht geven zichzelf te starten.



Afbeelding 37: Desktop Host met Actieve Mobiele Applicatie

We merken op dat het hier om een muziekspeler gaat. Men kan een liedje selecteren en deze laten afspelen. Er moet wel vermeld dat het hier om een simpel voorbeeld gaat. De applicatie werkt volledig, buiten het feit dat er geen echte liedjes afgespeeld worden. De vooruitgang bij het afspelen is zichtbaar, de titel van het actieve liedje wordt aangegeven, op het einde van het liedje zal overgegaan worden naar de volgende en shuffle zorgt ervoor dat de liedjes in willekeurig volgorde afgespeeld worden. We hebben de applicatie geprogrammeerd zonder rekening te houden dat het uiteindelijk een Mobiele Applicatie zou worden. De enige toevoeging die we deden om er een van te maken, was het implementeren van de Mobiele Applicatie interface.

Afbeelding 38: Actieve Mobiele Applicatie

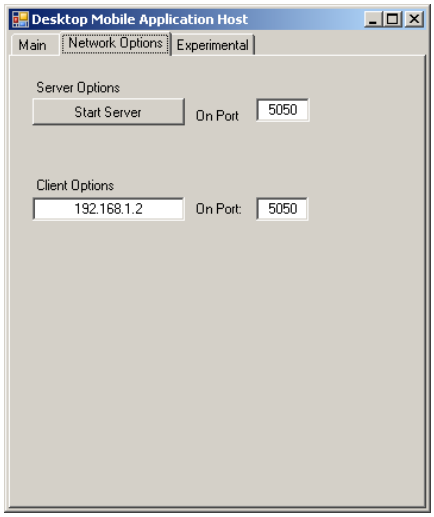
We kunnen nu werken met de mobiele applicatie, alsof het om een normale applicatie gaat. We selecteren een liedje en laten deze “afspelen”.

Op een gegeven moment beslissen we dat we verder willen luisteren op onze PDA. We gaan nu over tot het verhuizen van de applicatie van de desktop pc naar een PDA. De PDA is verbonden met ons netwerk via wifi.

Hiervoor dienen we eerst, bij de netwerk opties van de huidige host, het IP-adres opgeven van de nieuwe host.

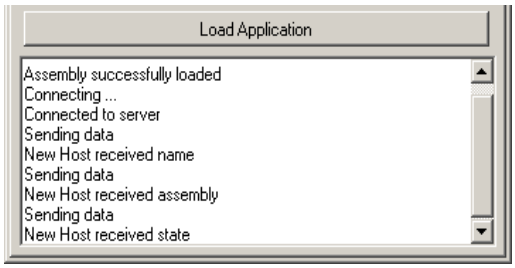
Hierna kunnen we het commando geven voor het verhuizen van de Mobiele Applicatie. Dit doen we via de host, deze zal dan via de gateway de Mobiele Applicatie de opdracht geven zijn staat te bewaren. Wanneer dit gebeurd is kan over gegaan worden

tot het echt verzenden van de Applicatie.



Afbeelding 39: Netwerk Opties

Er wordt een verbinding aangemaakt met de nieuwe host met behulp van TCP Sockets. We hebben al eerder uitgelegd hoe dit in zijn werk gaat. We kunnen op de oude host zien hoe de verzending verloopt.



Afbeelding 40: Oude Host, verloop zending

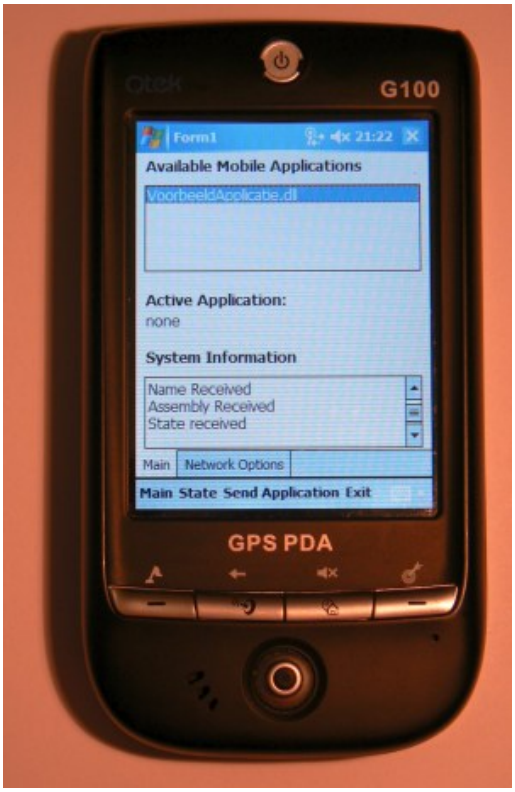
Afbeelding 41: Mogelijkheden bij starten Mobiele Applicatie

Op de nieuwe host krijgt men eveneens te zien hoe het verzenden verloopt. We zien dat de host achtereenvolgens de naam, Assembly en staat van de Mobiele Applicatie ontvangt. Hierna krijgt men de Mobiele Applicatie te zien in de lijst van beschikbare applicaties. Deze kan nu gestart worden indien gewenst.

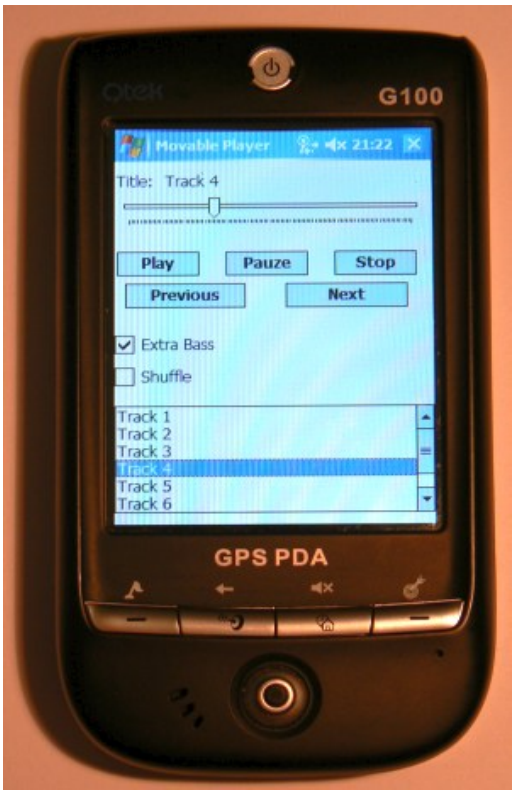
We zullen nu de ontvangen Mobiele Applicatie starten. Hierbij hebben we keuze uit “starten zonder vorige staat” en “starten met vorige staat”. We kiezen natuurlijk voor het starten met herstel van de staat, vermits we willen verder luisteren van waar we gestopt waren op de pc.

We zien dat op de PDA de Mobiele Applicatie gestart wordt met de staat waarmee hij verzonden werd op de desktop pc. Het liedje speelt nu verder af zoals het zou gedaan hebben op de vorige host.

Wanneer gewenst kunnen we de Mobiele Applicatie opnieuw zenden naar een nieuwe host. Dit kan weer een PDA zijn of een Desktop pc zijn.



Afbeelding 43: Nieuwe Host



Afbeelding 42: Mobiele Applicatie, geladen met behoud van staat

8.8 Conclusie

Na het implementeren van het Mobiele Applicatie Systeem kunnen we bevestigen wat bij het onderzoek van het .NET Framework al ondervonden werd. Het .NET Framework bied al de nodige features aan voor het creëren van mobiele applicaties. Dit houd in dat het is mogelijk om op een veilige manier code van onbetrouwbare bronnen dynamisch in te laden. Welke we na gebruik weer uit het geheugen kunnen verwijderen. Dit laatste is zeker nodig wanneer een host systeem maanden na elkaar operationeel is en constant Mobiele Applicaties host. Code en Resources kunnen verpakt worden in een handige container, de zogenaamde Assemblies. Het .NET Framework voorziet eveneens in mogelijkheden om netwerk-applicaties mogelijk te maken. Bij de ontwikkeling van de Mobiele Applicatie Host werd enige moeite ondervonden in het evenaren van de mogelijkheden van de Desktop versie. Dit is natuurlijk niet meer dan normaal, vermits de Compact versie van het .NET Framework beperkter is qua functionaliteit tegenover de Desktop versie. Toch zijn we er redelijk in geslaagd om een evenwaardige systeem te bouwen, op de veiligheid na. Dit maakt de PDA minder geschikt voor gebruik in Mobiele Applicatie Systemen, wanneer van geavanceerde veiligheidstechnieken zoals CAS gebruik gemaakt moet worden.

Het geïmplementeerde Mobiele Applicatie Systeem is verre van perfect. Waar bij deze implementatie niet op gelet werd, is de migratie van de UI. De UI van de applicatie die we in ons Mobiele Applicatie Systeem willen gebruiken moet zo gemaakt zijn dat deze bruikbaar is op de een PDA of smartphone. Dit omdat de UI één op één overgezet wordt bij het migreren van de applicatie.

Een Mobiele Applicatie die uitgevoerd dient te worden op zowel het desktop platform, als het pda platform, moet rekening houden met de beperkte mogelijkheden van dit laatste platform. Dit betekend dat de applicatie nooit volledig gebruik zal kunnen maken van de mogelijkheden die het desktop platform biedt.

Ook moet er rekening gehouden worden dat, zoals eerder aangekaart, het hier om weak mobility gaat. Niet de volledige applicatie staat wordt bewaard en terug geladen. Er kan dus niet verzekerd worden dat de applicatie volledig hersteld wordt in zijn vorige staat. Er wordt wel geprobeerd dit zo goed mogelijk te benaderen.

Een positief gevolg van deze gebreken, is het feit dat het ontwikkelen van een Mobiele Applicatie voor het systeem relatief simpel is. Men kan een simpele applicatie ontwikkelen zonder dat aan enige regels voldaan moet worden. De enigste vereist is dat de Mobiele Applicatie Interface geïmplementeerd wordt.

We zullen nu een gelijkaardige project bespreken.

9 Roam

Roam [28] is een geavanceerd Mobiel Applicatie Framework ontwikkeld in Java. Het biedt de mogelijkheid om een applicatie te ontwikkelen die van apparaat naar apparaat kan verhuizen, waarbij rekening gehouden wordt met de mogelijkheden en beperkingen van elk apparaat. Dit houdt in dat er gelet zal worden op de Java Virtual Machine versie, de grootte van het beeldscherm en de invoer mogelijkheden. Hierdoor kan een applicatie de volledige mogelijkheden benutten op elk platform. Op een mobiel apparaat met een beperktere Java versie zullen geen functies gebruikt worden die niet ondersteunt worden door de

aanwezige Java Virtual Machine.

We zullen nu bespreken hoe dit in zijn werk gaat, wat de voordelen en nadelen van dit systeem zijn tegenover ons Mobiel Applicatie Systeem.

9.1 Adaptatie Mechanisme

Om het mogelijk te maken dat Roam Applicaties aangepast kunnen worden naargelang het platform waarop ze draaien, maakt Roam gebruik van enkele technieken.

- **Dynamic Instantiation**

  Een applicatie wordt verdeeld in verschillende componenten, welke dezelfde functie hebben. Het verschil tussen de componenten zit hem in het feit dat elke component voor een specifiek platform bedoelt is. In Roam noemt zo'n component een M-DD (Multiple Device-Dependent). Wanneer een applicatie gestart of gemigreerd wordt, zal er gekeken worden welke set van M-DD's het best passen voor het platform waarop ze uitgevoerd zullen worden. Op deze manier kan steeds maximaal gebruik gemaakt worden van het host-platform.

- **Offloading Computation**

  Roam maakt het mogelijk dat applicaties gebruik maken van gedistribueerde computer bronnen. Dit maakt het mogelijk om bepaalde componenten, van een Mobiele Applicatie, uit te laten voeren op een ander systeem. Dit kan gebeuren wanneer de Mobiele Applicatie een functie moet uitvoeren, welke niet geschikt is voor het huidige host-platform. Dit kan gaan van te beperkte Virtual Machine tot te beperkte hardware (cpu, geheugen, ...). Wanneer een Roam applicatie een component bezit welke niet geschikt is voor het huidige platform, zal gezocht worden naar een server die in staat is deze wel uit te voeren. Wanneer deze gevonden wordt zal het betreffende component naar deze server verhuist worden. Een component die enkel uitgevoerd kan worden op één bepaald platform, wordt S-DD (Single Device-Dependent) genoemd.

- **Transformation**

  Een Roam applicatie kan ook Device-Independent Componenten (S-DI) bezitten. Deze kunnen tijdens de uitvoering van de applicatie getransformeerd worden, naargelang de capaciteiten van het platform waarop ze uitgevoerd worden. Roam is voorzien met een device-independent GUI toolkit. Dit maakt het mogelijk voor ontwikkelaars om een device-independent UI te maken. Roam neemt de transformatie, tot een geschikte versie voor een bepaald apparaat, voor zijn rekening.

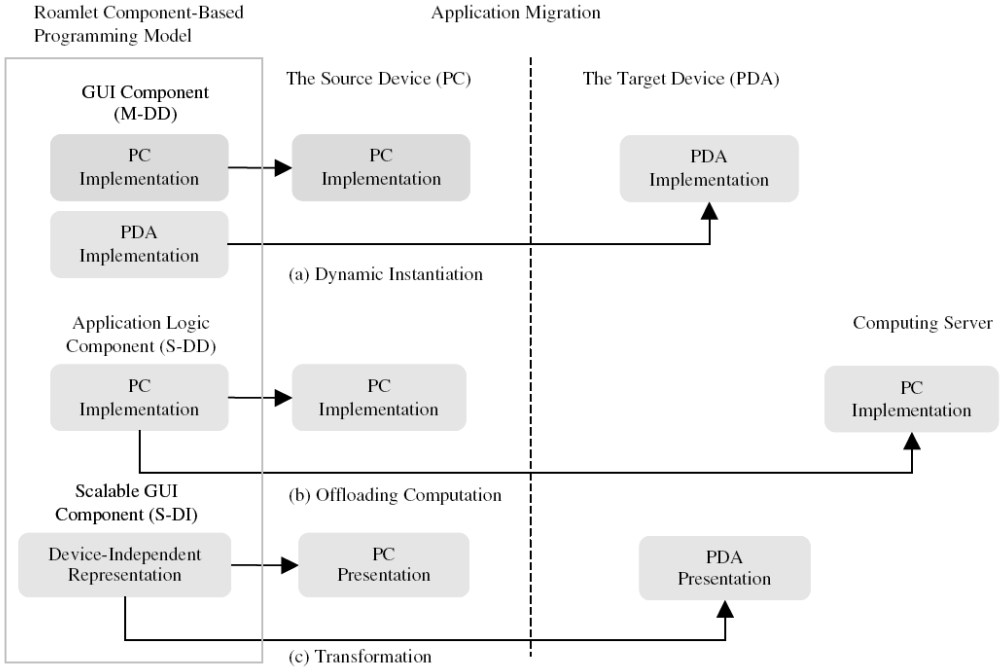
Bij Roam dient de programmeur van mobiele applicaties rekening te houden met deze drie technieken. Hij moet zijn applicatie opdelen in verschillende componenten, naargelang hun functie en eisen. Een UI wordt bijvoorbeeld het best als S-DI geïmplementeerd met behulp van de voorziene toolkit. Hierbij moet dan weer rekening gehouden worden dat de automatische transformatie niet in alle gevallen een hoogwaardige UI kan leveren. In deze gevallen moet men overstappen naar M-DD's en voor elk gewenst platform een versie

implementeren. De logica van de applicatie kan dan weer als S-DD geïmplementeerd worden. Dit maakt het verhuizen ervan mogelijk, zodat gedistribueerd computing mogelijk is wanneer nodig.

9.2 Architectuur

Het Roam systeem bestaat uit drie belangrijke onderdelen:

- Roamlets
  Een Roamlet kan migreren tussen twee verbonden apparaten die het Roam systeem hebben draaien. Roamlets zijn component gebaseerd. Deze componenten zijn de reeds besproken M-DD's, S-DD's en S-DI's. Roamlets zorgen ervoor dat zowel volledige, gedeeltelijke en gemengde migratie van de User interface mogelijk is.



Afbeelding 44: Overzicht Roamlet

- Roam Agents
  Elk apparaat moet een agent geïnstalleerd hebben, wil deze applicaties kunnen ontvangen en verzenden. De Agents kunnen met elkaar communiceren. Wanneer een Roamlet wilt verhuizen, maken deze Agents afspreken met elkaar zodat de Roamlet correct verplaats kan worden.

- HTTP server
  Via deze HTTP servers kunnen de host systemen de code bekomen van de Roamlets die naar hun willen verhuizen.

9.3 Migratie Flow

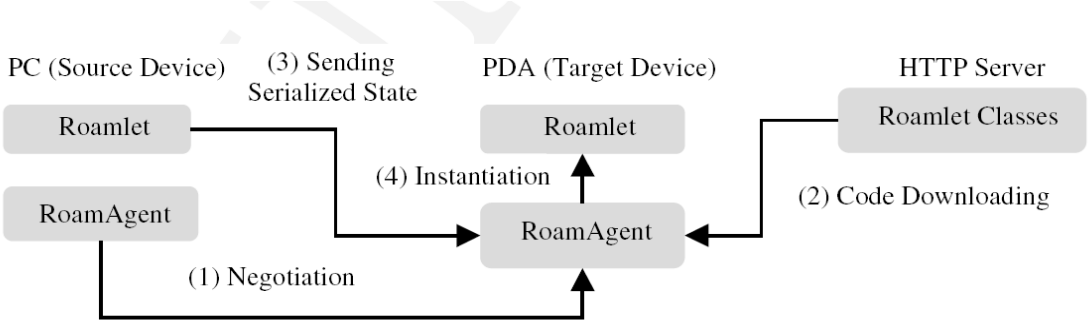
Elk onderdeel van Roam bezit zijn specifieke taak bij het migreren van een Roamlet. De migratie gebeurd volgens volgende stappen:

1. De Roam Agent op het bron apparaat onderhandelt met de Agent op het toekomstig apparaat. Hierbij worden de systeem eisen van de Roamlet, de systeem specificaties van het toekomstig apparaat en de URL (waar de Roamlet code afgehaald kan worden) uitgewisseld.

2. De Roam agent op het toekomstig apparaat weet nu welke systeemeisen vereist zijn voor de verschillende componenten van de Roamlet. Hij kan nu de componenten, welke geïnstantieerd zullen worden op het systeem, afhalen van de HTTP server met behulp van de reeds ontvangen URL. Bij een D-DI zal er nog een transformatie moeten gebeuren naar het huidig platform. Dit op basis van de voorziene transformatie-regels.

3. De Roamlet op het bron apparaat serializeert zijn staat en zendt het naar de Agent op het toekomstig apparaat. Bij ontvangst van de staat moet rekening gehouden worden dat de staat afkomstig kan zijn van een M-DD. Dit betekent dat wanneer het platform verandert de staat niet meer overeen zal komen. Hierdoor kan het gebeuren dat er nog een transformatie van de staat gebeurd, zodat deze wel overeen komt met het nieuwe platform.

4. De Agent op het nieuwe apparaat instantieert de Roamlet. Op dit moment kan er ook gezocht worden naar een server. Hierop kunnen de componenten uitgevoerd worden die niet of niet goed op het nieuwe systeem werken (offloading).



Afbeelding 45: Roam: Migratie Flow

9.4 Veiligheid

Zoals een goed Mobiel Applicatie Systeem beaamd, werd ook bij Roam rekening gehouden met de veiligheid. Er wordt vanuit gegaan dat de Roam Agents, geïnstalleerd op de verschillende apparaten, te vertrouwen zijn. Het Roam veiligheidssysteem richt zich vooral op integriteit, privacy en authenticatie. Hiervoor wordt de executiestaat van de mobiele applicatie wordt geëncrypteerd. Zodat alleen de gerechtigde ontvanger deze kan decrypteren. Hiermee is hij ook direct zeker dat de staat van hem afkomstig is.

Dit gaat als volgt in zijn werk:

1. Wanneer een applicatie gemigreerd dient te worden, zal de Roam Agent vragen om een wachtwoord te kiezen. Dit wachtwoord is een one-time wachtwoord dat enkel geldig is voor één bepaalde applicatie migratie.

2. Hierna zal de Agent een uniek nummer (uniek voor de Agent) genereren. Op basis van dit nummer en het gekozen wachtwoord van de gebruiker wordt de staat geëncrypteerd.

3. Vervolgens wordt de geëncrypteerde staat verzonden, samen met het unieke nummer, naar de volgende host.

4. Eenmaal aangekomen op de nieuwe host, zal deze de gebruiker vragen om zijn wachtwoord. Dit om de staat te kunnen decrypeteren.

Door deze werkwijze weet de ontvanger zeker, indien de host betrouwbaar is, dat de staat onveranderd verzonden werd. Bij ontvangst is men eveneens zeker dat de staat effectief de zijn verzonden staat is. Indien dit niet zo is, zal het decrypteren niet lukken. Het spijtige aan dit systeem is dat de gebruiker hier lastig gevallen wordt met een vraag bij de verhuis. Dit zorgt ervoor dat het verhuizen van de applicatie niet meer volledig transparant verloopt wat toch een afbreuk vormt van het Mobile Ubiquitous Computing.

9.5 Conclusie

Wanneer we Roam vergelijken met ons ontwikkeld systeem, merken we dat Roam een oplossing biedt voor enkele van onze problemen. Het is mogelijk om optimaal gebruik te maken van het aangeboden platform waarop een mobiele applicatie uitgevoerd zal worden. Er kan rekening gehouden worden met de systeem specificaties zoals de versie van de Java Virtual Machine en schermgrootte. Roam ondersteunt zelfs alle typen van User Interface migratie, zoals besproken in hoofdstuk 5. Het systeem, die hiervoor in staat, vergt wel extra moeite van de ontwikkelaars van Mobiele Applicaties. Deze moeten ervoor kiezen om ofwel voor elk platform een versie te ontwikkelen, ofwel om te werken in de platform onafhankelijke toolkit.

De ontwikkelaars moeten ook extra moeite steken in het opdelen van de applicatie. Deze moet namelijk uit meerdere componenten bestaan, welke al eerder aangekaart werden. Dit zodat er optimaal gebruik gemaakt kan worden van de mogelijkheden van het Roam Systeem. Dit betekend extra denk- en codeerwerk. Ontwikkelaars ondervonden dat bijvoorbeeld een schaakspel, welke origineel 7900 lijnen code bevatte, na conversie naar Roam 9000 lijnen code bevatte.

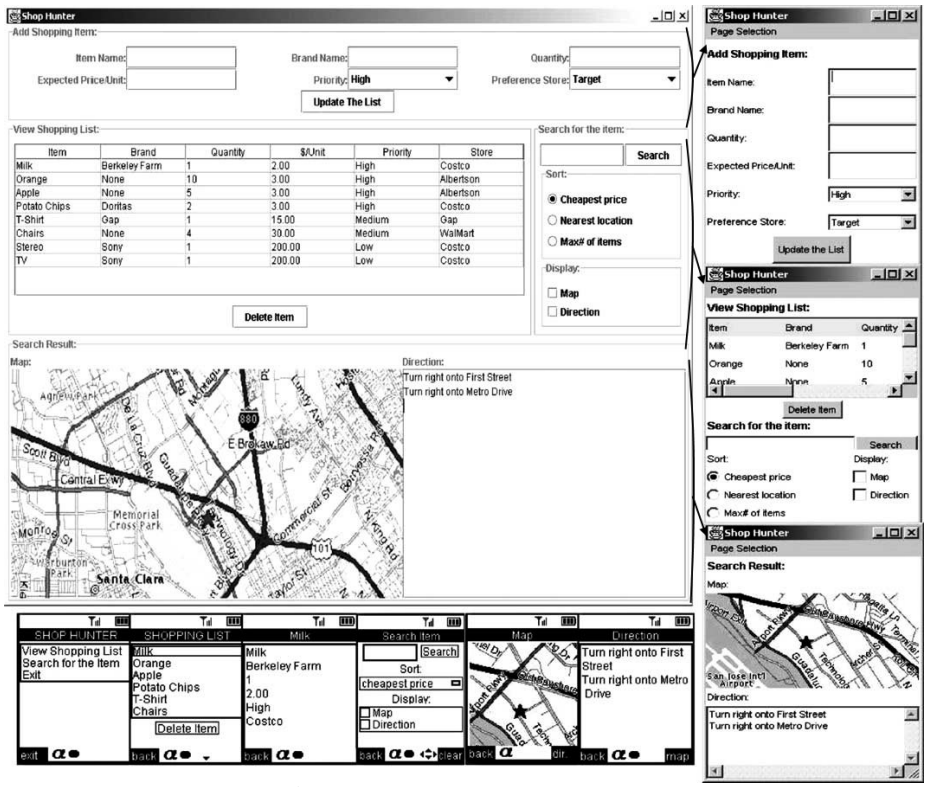
Om de automatische transformatie van een platform onafhankelijke User Interface mogelijk te maken, moeten de ontwikkelaars werken met de hiervoor voorziene toolkit. Dit betekent dat de programmeurs geen gebruik kunnen maken van een voor hun vertrouwde toolkit, wat als als nadelig beschouwd kan worden. Ons systeem houdt geen rekening met de User Interface. Hierdoor hebben we dit probleem niet, het gevolg is wel dat we deze zeer nuttige functie moeten missen.

Het overzetten van de staat is ook niet altijd even simpel bij Roam. Dit omdat door gebruik te maken van M-DD's de staat, na migratie, niet altijd rechtstreeks gelinkt kan worden met de nieuwe instantie. Hierdoor moet deze bewaard worden in een device-independent manier, dit kost natuurlijk ook weer extra moeite. S-DD's en S-DI's hebben hier geen last van. Omdat ons systeem steeds met dezelfde componenten werkt, hebben we hier geen last van.

Wanneer we het veiligheidssysteem van Roam vergelijken met ons systeem, merken we dat het verschil vooral zit in wat beveiligd wordt. In Roam wordt de staat van de applicatie beschermd door deze te encrypteren. Op deze wijze kan deze niet door derden gelezen of aangepast worden. In ons systeem beschermen we vooral de host, door gebruik te maken van Applicatie Domeinen. Deze beperken de mogelijkheden van een malafide applicatie, zodat deze niet teveel schade kan aanrichten.

Ondanks zijn extra vereisten, biedt Roam een zeer mooi systeem aan. Het maakt het mogelijk om een echte mobiele applicatie te ontwikkelen, waarbij volledig rekening gehouden wordt met de apparaten waarop ze uitgevoerd worden. Dit maakt het mogelijk om op elk apparaat optimaal gebruikt te maken van de aanwezige voorzieningen. Dit in tegenstelling tot ons systeem waarbij we eisen dat de Mobiele Applicatie volledig werkt op het zwakste apparaat waarop deze uitgevoerd kan worden. Hierdoor wordt op een krachtiger platform geen gebruik gemaakt van de extra mogelijkheden.

Tot slot geven we nog een voorbeeld van de User Interface van een Roam applicatie. We zien hier hoe een desktop applicatie, bestaande uit een enkel scherm, omgezet wordt naar een applicatie bestaande uit meerdere schermen. De functionaliteit van het systeem blijft bewaard en men kan toch nog op beide platformen comfortabel werken.



Afbeelding 46: Roam: Pc -> PDA

10 Conclusie

We zien dat de mens steeds mobieler wordt, maar spijtig genoeg bezitten onze applicaties nog steeds deze mobiliteit niet. Dit komt door de vele moeilijkheden die overwonnen dienen te worden bij het creëren ervan, namelijk de taak continuïteit, de veiligheid en het behouden van de Usability.

We hebben gezien dat het bewaren van de staat van een applicatie vaak verre van evident is. Toch mag dit niet verwaarloos worden. Het stel een gebruiker in staat om zijn applicatie te verhuizen zonder zich bezig te houden met het verhuizen van de nodige data. Dit zorgt dat het wisselen van apparaten bij Mobiele Applicaties zo transparant mogelijk verloopt. De gebruiker kan daardoor zich blijven focussen op zijn taak en niet zo zeer op de verhuis zelf. Zonder het behoud van deze staat, zal dus minder rap gesproken kunnen worden van Mobile Ubiquitous Computing. In de thesis hebben we dan ook een manier besproken waarmee we dit probleem kunnen oplossen. Dit door middel van de ontwikkelde serializator.

Het zomaar verhuizen van applicaties van het ene apparaat naar een andere is natuurlijk niet zonder gevaar. Er moet rekening gehouden worden met de gevaren voor zowel het host-systeem als de Mobiele Applicatie zelf. Dit op zich is niet zo nodig om een gevoel van Mobile Ubiquitous Computing te geven, maar mag zeker niet verwaarloost worden. Een malafide Mobiele Applicatie op een onbeschermde host, kan zeer veel schade aanrichten. We zien elke dag weer hoe gevaarlijk het internet geworden is en welke problemen Mobiele Code daar veroorzaakt. Het is dan ook onzinnig om een Mobiel Applicatie systeem te ontwikkelen zonder hier rekening mee te houden. Gelukkig zijn veel mogelijkheden voor handen om dit te realiseren. Wanneer een applicatie bijvoorbeeld met behulp het .NET Framework geprogrammeerd wordt, is er een breed scala van mogelijkheden tot beveiligen beschikbaar. Wanneer die goed gebruikt worden zal de gebruiker hier zelf niks van merken, tot op het moment er daadwerkelijk moet opgetreden worden.

Bij het verhuizen van een applicatie van een apparaat naar een ander is het veilig overbrengen van code en behouden van de staat niet het enigste waar rekening mee gehouden moet worden. Wanneer bijvoorbeeld simpelweg een User Interface gericht op grote schermen overgebracht wordt naar een klein scherm, treden hier ernstige usablity problemen op. Tekst, knoppen, ... worden te klein. Ook kunnen de invoermogelijkheden kunnen veranderen wanneer men bijvoorbeeld niet meer beschikt over een volledig toetsenbord en muis. Een Mobiele Applicatie dient dus rekening te houden met het soort apparaat waarop hij zich bevindt. Er zijn verschillende methoden hiervoor. Een mooie oplossing is het abstract beschrijven van de interface, zodat deze dynamische opgebouwd kan worden. Hierbij rekening houdende met de specificaties van het apparaat waar de applicatie zich op bevindt. Dit betekend dat programmeurs op voorhand hier rekening mee zullen moeten houden en goed nadenken over de opdeling van hun User Interface. Spijtig genoeg is het automatische opbouwen of generen van een interface geen simpele zaak. Zeker niet wanneer Usablity belangrijk is. Een simpeler oplossing is voor elke applicatie verschillende User Interfaces beschikbaar te stellen. Eén voor elk apparaat waarop de applicatie terecht kan komen. Welke oplossing ook gekozen wordt, het blijft een feit dat dit extra inspanning vraagt van de programmeurs en niet altijd even simpel is.

Het is duidelijk dat het creëren van een mobiele applicatie niet zomaar even gebeurd. Toch hebben we gezien dat de huidige technologieën krachtig genoeg zijn om dit mogelijk te

maken. Het vergt natuurlijk wel meer moeite en misschien wringt hem hier het schoentje.

Bibliografie

1: Luca Cardelli & Andrew D. Gordon, Mobile Ambients, 1998

2: Schussel George, Client/Server Past, Present, and Future, 1995

3: James W. Stamos & David K. Gifford, Remote Evaluation, ACM Transactions on Programming Languages and Systems, Vol.12, No. 14., 1990

4: Walter Binder, Giovanna Di Marzo Serugendo, Jarle Hulaas, Towards a Secure and Efficient Model for Grid Computing using Mobile Code, 2002

5: Alfonso Fuggetta, Gian Pietro Picco & Giovanni Vigna, Understanding Code Mobility, 1998

6: Chong Xu, Dongbin Tao, Building Distributed Application with Aglet

7: P.E.Clements, T.Papaioannou, J.Edwards, Aglets: Enabling the Virtual Enterprise, 1997

8: P. Dasgupta, N. Narasimhan, L. E. Moser & P. M. Melliar-Smith, MAgNET: Mobile Agents for Networked Electronic Trading

9: JITEN RAMA & , Towards a mobile agent framework for Nomad using .NET, 2004

10: Giacomo Cabri, Letizia Leonardi & Franco Zambonelli, Weak and Strong Mobility in Mobile Agent Applications

11: M. Ranganathan, Anurag Acharya, Shamik D. Sharma & Joel Saltz, Network-aware Mobile Programs, 1997

12: Holger Peine & Torsten Stolpmann, The Architecture of the Ara Platform for Mobile Agents , 1997

13: Stefan Fünfrocken, Transparent Migration of Java-based Mobile Agents Capturing and Reestablishing the State of Java Programs, 1998

14: Thomas Ledoux & Noury M.N.Bouraqadi-Saâdani, Adaptability in Mobile Agent Systems using Reflection, 2000

15: Jonathan Moore, Mobile Code Security Techniques, 1998

16: Wayne Jansen & Tom Karygiannis, Mobile Agent Security, 2000

17: Anthony H. W. Chan & Michael R. Lyu, The mobile code paradigm and its security issues

18: Dan S. Wallach, Dirk Balfanz, Drew Dean & Edward W. Felten, Extensible Security Architectures for Java, 1997

19: Joy Algesheimer, Christian Cachin, Jan Camenisch & Gunter Karjoth, Cryptographic Security for Mobile Code

20: John Viega, J.T. Bloch, Yoshi Kohno, Gary McGraw, A Static Vulnerability Scanner for C and C++ Code, 2000

21: Erez Zadok, Matthew G. Schultz and Eleazar Eskin, Data Mining Methods for Detection of New Malicious Executables,

22: Guy Edjlali, Anurag Acharya & Vipin Chaudhary, History-based Access Control for Mobile Code, 1998

23: George C. Necula & Peter Lee, Proof-Carrying Code, 1996

24: Peter Lee & George Necula, Research on Proof-Carrying Code for Mobile-Code

Security , 1997

25: Håkan Melin, ATLAS: A generic software platform for speechtechnology based applications, 2001

26: Renata Bandellon & Fabio Paternò, Migratory User Interfaces Able to Adapt to Various Interaction Platforms, 2003

27: Renata Bandelloni & Fabio Paternò, Flexible Interface Migration, 2004

28: Hao-hua Chu, Henry Song, Candy Wong, Shoji Kurakake, Masaji Katagiri, Roam, a seamless application framework, 2002

29: Krishna A. Bharat & Luca Cardelli, Migratory Applications, 1995

30: Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld & Mathilde Pignol, Generating Remote Control Interfaces forComplex Appliances, 2002

31: Renata Bandelloni, Silvia Bert &, Fabio Paternò, Mixed-Initiative, Trans-Modal Interface Migration, 2004

32: Nickolas Landry, Mobile CoDe.NET: Exploring the .NET Compact Framework (cont'd), 2003

33: Joe Duffy, Professional .NET Framework 2.0

34: Jeffrey Richter, Metadata zijn de hoekstenen van het .NET Framework, 2002

35: M. van Elswijk, Persistente Objecten

36: Jim Farley, Java Distributed Computing, 1998

37: Jim Mischel, .NET Reference Guide

38: Piet Obermeyer and Jonathan Hawkins , Object Serialization in the .NET Framework,

39: Zach Smith , Implement XML serialization in the .NET Framework

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Mobile Ubiquitous Computing
Richting: **Master in de informatica** Jaar: **2007**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

Tom Vanliefde

Datum: **23.05.2007**