Distributed Java/Smalltalk framework

een framework ter ondersteuning van gedistribueerde Java/Smalltalk applicaties

Faber, R.D.H.J.

Award date:
1997

Technische Universiteit Eindhoven

Faculteit Elektrotechniek
Vakgroep Digitale Informatie Systemen (ICS/EB)

DISTRIBUTED JAVA/SMALLTALK FRAMEWORK

"EEN FRAMEWORK TER ONDERSTEUNING VAN GEDISTRIBUEERDE JAVA/SMALLTALK APPLICATIES"

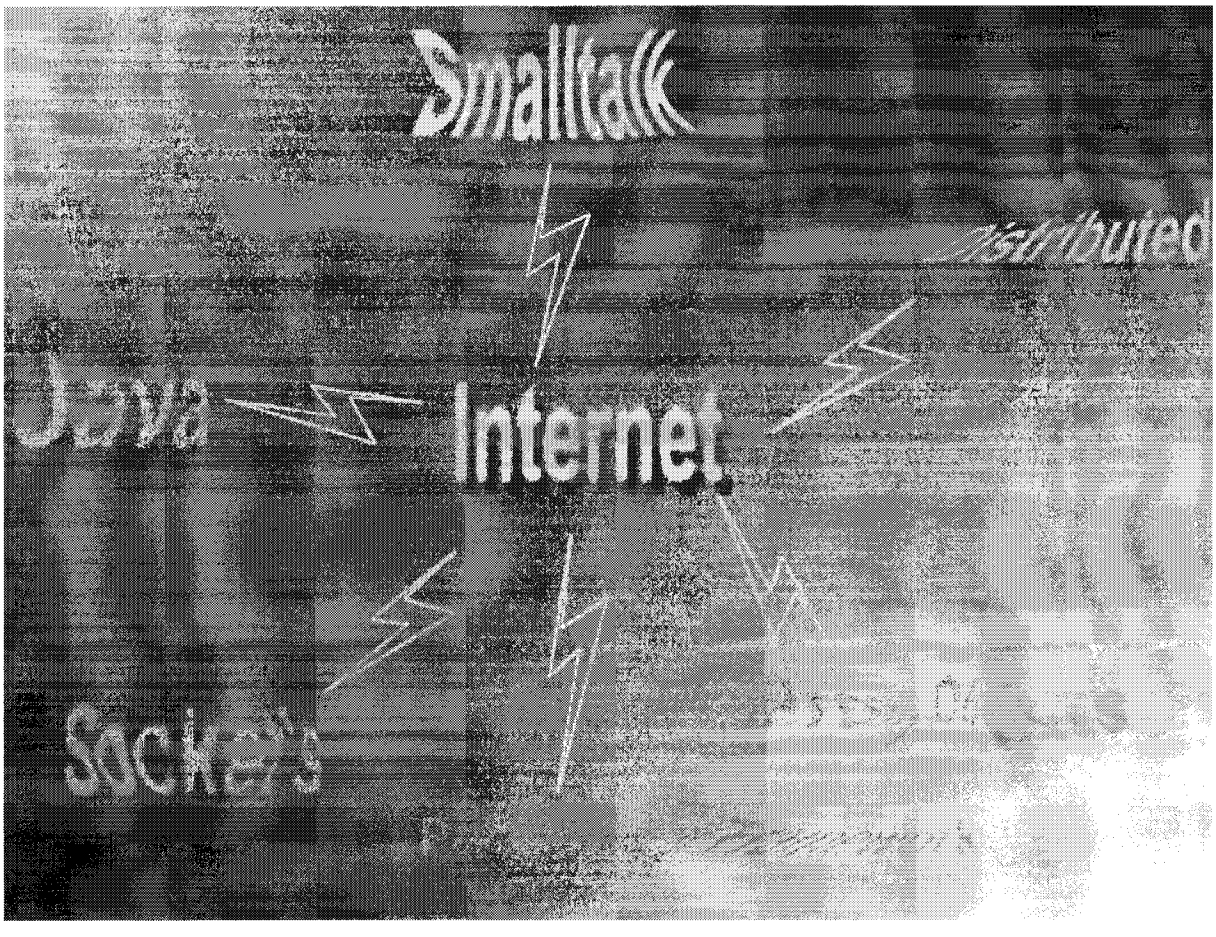
Door R.D.H.J. Faber

Auteur: R.D.H.J. Faber (idnr.: 347761).
Begeleiders: Ir. M.J.M. v. Weert, Ir. S. v.d. Kuilen
Professor: Ir. M.P.J. Stevens.

Datum: 10 April 1997.

Distributed Java/Smalltalk Framework

" Een framework ter ondersteuning van gedistribueerde Java/Smalltalk applicaties "



door: R.D.H.J. Faber

datum : 04-04-97

Distributed Java/Smalltalk Framework

" Een framework ter ondersteuning van gedistribueerde Java/Smalltalk applicaties "

geschreven in opdracht van:

ELC Object Technology B.V.

ter afsluiting van een studie:

Informatietechniek

aan de:

Technische Universiteit Eindhoven

door: R.D.H.J. Faber

datum : 04-04-97

Voorwoord

Het verslag dat hier voor u ligt is het eindresultaat van een traject van LTS, MTS, HTS om uiteindelijk aan de Technische Universiteit Eindhoven af te studeren in de richting Informatietechniek. Alle stappen in dit traject, inclusief de eerste, zijn voor mij altijd logisch geweest. Al sinds mijn jeugd ging mijn interesse uit naar techniek, maar dat deze interesse zou leiden tot een ingenieurs titel had niemand gedacht.

Binnen ELC Object Technology heb ik niet één begeleider gehad, maar heb ik 'gebruik kunnen maken' van een heel team van professionals die mij altijd met raad en daad hebben bijgestaan. Tijdens mijn afstudeerperiode heb ik regelmatig mijn bevindingen op vergaderingen gepresenteerd. Hierbij was ik steeds verbaasd over de feed-back die ik kreeg van mijn collega's. Bij deze wil ik hen dan ook bedanken voor hun steun. Tevens wil ik hierbij R. van Weert van de Technische Universiteit Eindhoven bedanken voor zijn begeleiding.

Dit afstudeerwerk wil ik opdragen aan mijn moeder, die dit helaas niet meer mee mag maken.

R.D.H.J. Faber

Capelle a/d IJssel, maart 1997

Samenvatting

Gedistribueerde systemen bestaan al sinds er computernetwerken zijn. 'Gedistribueerde systemen' is een breed begrip. De computers in zo'n gedistribueerd systeem kunnen gebruik maken van elkaars diensten. Met een gedistribueerd systeem in object georiënteerde programmeertalen wordt bedoeld dat de objecten van de ene computer gebruik kunnen maken van de objecten op een andere computer. Zowel Java als Smalltalk zijn object georiënteerde programmeertalen. Een Java applicatie kan worden toegevoegd aan een internet pagina en is verplaatsbaar over een computernetwerk. Een gedistribueerde Java/Smalltalk applicatie is een applicatie waarbij de Java objecten gebruik maken van Smalltalk objecten, die zich op een fysiek andere lokatie bevinden.

Het "Distributed Java/Smalltalk framework" bestaat uit componenten die het eenvoudig maken om zo'n gedistribueerde Java/Smalltalk applicatie te ontwikkelen. Hiervoor zijn drie componenten ontwikkeld. De eerste component is een **Smalltalkobjectserver** die aanvragen van diensten verwerkt. De tweede component is de **Javaproxycommunicator**, die ervoor zorgt dat de aanvragen van diensten bij de server terecht komen en de resultaten hiervan verwerkt worden. De laatste component is een **Javaproxygenerator** die proxyobjecten genereert van de Smalltalk objecten waarvan de diensten gebruikt moeten gaan worden. Door de proxyobjecten is het mogelijk op een gestructureerde manier van diensten van objecten op een andere computer gebruik te maken.

Met behulp van dit distributed Java/Smalltalk framework is het eenvoudig om gedistribueerde Java/Smalltalk applicaties te ontwikkelen. De ontwikkelaar kan zich concentreren op het ontwikkelen van de functionaliteit van de gedistribueerde applicatie. Hierbij hoeft hij zich geen zorgen te maken over de manier waarop de communicatie tot stand komt tussen enerzijds het deelsysteem in Java en anderzijds het deelsysteem in Smalltalk. Het distributed Java/Smalltalk framework zal voornamelijk toegepast worden in omgevingen waar al Smalltalk applicaties aanwezig zijn. Met dit framework kunnen bepaalde deelsystemen op eenvoudige wijze beschikbaar worden gemaakt voor het internet. De huidige infrastructuur van het internet stelt echter wel zijn beperkingen aan de complexiteit van de gedistribueerde applicaties. Dit heeft twee redenen. Enerzijds zal een complexe applicatie groter van omvang zijn en zal het daarom langer duren voordat de Java applicatie via het internet is getransporteerd. Anderzijds zal een complexe applicatie meer communiceren met de Smalltalk server en zal daardoor ook trager worden.

Inhoudsopgave

Hoofdstuk 1 Inleiding

*M*et de komst van computernetwerken is een nieuw fenomeen geboren: het gedistribueerde systeem. In een gedistribueerd systeem kunnen computers gebruik maken van diensten die een andere computer binnen een computernetwerk, te bieden heeft. Tevens is met de komst van het World Wide Web, het fenomeen Java [Lemay 1996, Hoff 1996, Manger 1996] geboren. Java is een object georiënteerde programmeertaal, die is afgeleid van C++. Het voordeel van deze taal is dat deze in gecompileerde vorm platformonafhankelijk is, waardoor de mogelijkheid ontstaat om Java programma's op allerlei platformen uit te voeren. Kleine Java programma's (**applets** genaamd) kunnen toegevoegd worden aan internet pagina's. Hierdoor kunnen de internet pagina's een zekere mate van interactiviteit verkrijgen, die zonder Java applets beperkt is.

Smalltalk [IBM 1995d,e] is een object georiënteerde programmeertaal waarmee bij ELC Object Technology veel applicaties en frameworks ontwikkeld worden. Hierbij wordt gebruik gemaakt van VisualAge for Smalltalk [IBM 1995a,b,c] van IBM. Eén van de in VisualAge for Smalltalk ontwikkelde frameworks is het zogenaamde "persistency framework". Het persistency framework [ELC 1995, ELC 1996] voorziet in alle database acties bij het opslaan, bewaken van de consistentie en bescherming van de objecten, kortom het persistent maken van objecten. Om aan de huidige en toekomstige wensen van de klanten te kunnen voldoen is ELC Object Technology geïnteresseerd in het ontwikkelen van gedistribueerde Java/Smalltalk applicaties en heeft daarom de volgende opdracht geformuleerd:

"Ontwikkel een applicatie ter ondersteuning van gedistribueerde Java/Smalltalk applicaties, die aansluit bij het bestaande persistency framework."

Deze ondersteunende applicatie (in het vervolg **distributed Java/Smalltalk framework** of kortweg **framework** genoemd) moet de ontwikkelaar voorzien in de nodige componenten om gedistribueerde Java/Smalltalk applicaties op te zetten.

Hoofdstuk 2 bespreekt op welke manier gedistribueerde applicaties op een gestructureerde manier opgezet kunnen worden en hoe de opbouw van zo'n gedistribueerd Java/Smalltalk systeem eruitziet. Dit hoofdstuk bespreekt ook het communicatiemodel dat gebruikt wordt voor de client/server communicatie. Hoofdstuk 3 beschrijft de verschillende componenten waaruit het framework bestaat. De manier waarop het framework gebruikt kan worden, wordt besproken in hoofdstuk 4. Tevens beschrijft dit hoofdstuk de manier waarop het framework gebruikt kan worden in combinatie met het persistency framework. In hoofdstuk 5 worden conclusies getrokken met betrekking tot het ontwikkelde framework.

Hoofdstuk 2 Gedistribueerde Java/Smalltalk applicaties

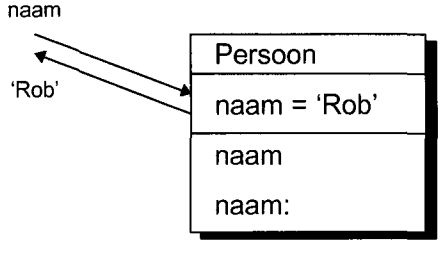
Gedistribueerde applicaties die ontwikkeld worden in object georiënteerde programmeertalen kunnen op een gestructureerde wijze opgezet worden. Dit wordt bereikt door ervoor te zorgen dat er geen onderscheid gemaakt kan worden tussen 'lokale diensten' en 'gedistribueerde diensten'. Op welke manier tot deze gestructureerde opzet is gekomen, staat beschreven in paragraaf 2.1. Hierna wordt in paragraaf 2.2 toegelicht welke specifieke eigenschappen gedistribueerde Java/Smalltalk applicaties hebben. Paragraaf 2.3 bespreekt de opbouw van gedistribueerde applicaties zoals deze ontwikkeld gaan worden. De manier waarop er tussen de systemen gecommuniceerd wordt, is onderwerp van paragraaf 2.4.

§2.1 Gedistribueerde applicaties in object georiënteerde programmeertalen

Bij object georiënteerde programmeertalen zijn diensten altijd aanwezig in bepaalde klassen. Klassen leggen de gemeenschappelijke structuur van eigenschappen en diensten van bepaalde objecten vast. Over het verloop van object georiënteerde applicaties schrijft Bennett het volgende:

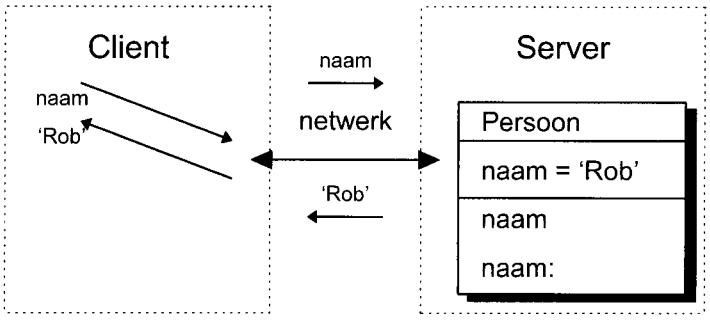
"The sole paradigm for object interaction in Smalltalk is message passing. A sender sends a message to a receiver to cause the receiver to perform some action". [Bennett 1987]

In bovenstaand citaat kan *"Smalltalk"* gelezen worden als "Object oriented programming languages". Dit principe van het sturen van berichten naar objecten is geschetst in Figuur 1.



Figuur 1 Sturen van berichten

Afhankelijk van het resultaat van deze methoden worden weer andere methoden aangeroepen. Het is ook mogelijk om deze berichten via een computernetwerk naar een object te sturen. Het resultaat van deze methoden wordt dan weer via dit computernetwerk teruggestuurd(Figuur 2).

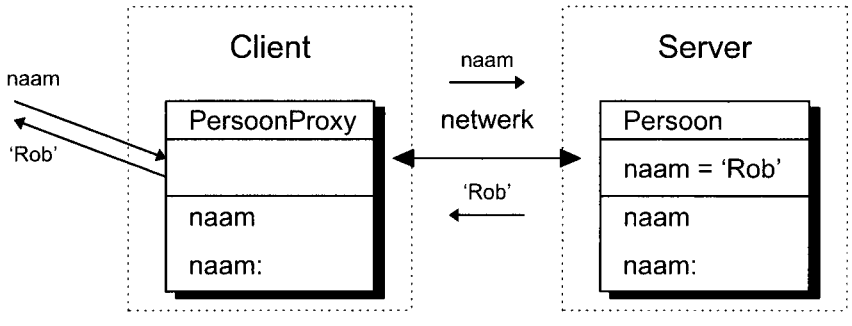


Figuur 2 Sturen van berichten via een netwerk

Om het aanroepen van methoden via een computernetwerk zo gestructureerd mogelijk te laten verlopen, worden aan de client zijde van een gedistribueerde applicatie zogenaamde **proxyobjecten** opgenomen. Het gebruik van proxyobjecten is niet nieuw, Bennet schrijft hierover:

"A proxy object represents a remote object to all objects in the local address space. There is one proxyObject per host per remote object referenced by that host. In this way, proxyObjects cause a remote object's message interface to appear to local objects as if the remote object were locally present" [Bennett 1987]

Hieruit blijkt dat door gebruik te maken van proxyobjecten, het lijkt alsof de objecten lokaal aanwezig zijn(Figuur 3).



Figuur 3 Sturen van berichten via een netwerk m.b.v. proxyobjecten

In Figuur 3 valt op dat het proxyobject PersoonProxy geen attributen heeft. De attributen worden benaderd via de **getters** en **setters**. Getters zijn methoden waarmee de waarden van attributen opgevraagd kunnen worden. Setters zijn methoden waarmee de waarden van attributen gezet kunnen worden. Deze getters en setters zijn wel opgenomen in de proxyobjecten. Dit heeft als nadeel dat elke keer als de waarde van een attribuut opgevraagd wordt, dit via het computernetwerk gebeurd. Een andere implementatie zou wel attributen kunnen opnemen in de proxyobjecten. Dit heeft als voordeel dat niet elke keer gecommuniceerd wordt met de objecten op de server. De consequentie hiervan is dat de waarde van de attributen aan de client zijde niet altijd up-to-date zijn en er dus een refresh mechanisme, geïnitieerd door de client of server, ingebouwd moet worden. De client kan een refresh om een bepaalde tijd uitvoeren. Het nadeel hiervan is dat er nooit met zekerheid gezegd kan worden of de attribuutwaarden van objecten aan de server zijde gelijk zijn aan de attribuutwaarden van ob-

jecten aan de client zijde. Het refresh mechanisme kan ook geïnitieerd worden door de server. Als een attribuutwaarde aan de server zijde verandert, zal de server een soort broadcast uitvoeren naar alle clients die het betreffende object gebruiken. In dit geval moet het wel mogelijk zijn om aan de client zijde alle objecten die op een willekeurig moment in een applicatie voorkomen, te traceren. Omdat dit laatste in Java niet zonder meer mogelijk is, is gekozen voor een implementatie zonder attributen in de proxyobjecten.

§2.2 Specifieke eigenschappen van gedistribueerde Java/Smalltalk applicaties

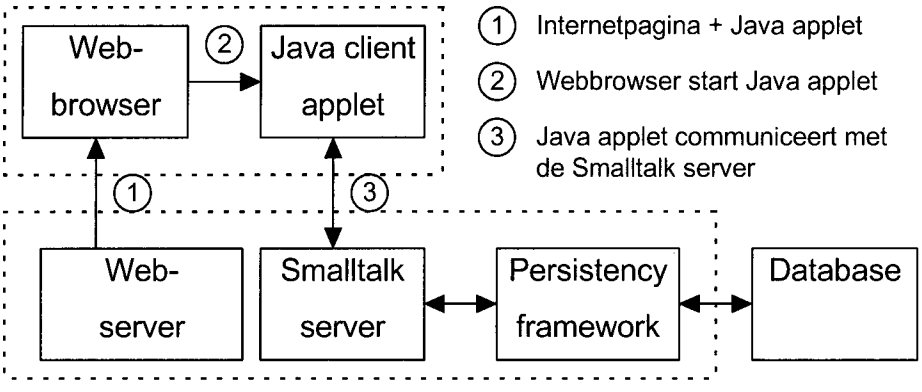
In het geval van gedistribueerde Java/Smalltalk applicaties moet rekening gehouden worden met de specifieke eigenschappen van de twee programmeertalen. Met betrekking tot de programmeertaal Java moet onderscheid gemaakt worden tussen **applications** en **applets**. Applets zijn programma's die opgenomen kunnen worden in internet pagina's. Applications zijn normale programma's zoals deze met elke andere taal ontwikkeld worden. Er zijn enkele belangrijke verschillen tussen applications en applets. Java applets moeten aan strenge veiligheidseisen voldoen. Deze veiligheidseisen kennen Java applications niet. ELC Object Technology is in eerste instantie geïnteresseerd in de internet toepassingen van Java, dus het gebruik van Java applets. De belangrijkste beperkingen die Java applets hebben zijn:

- een Java applet mag alleen communiceren met de computer waarvan deze in eerste instantie afkomstig is,
- een Java applet mag geen gebruik maken van op de client aanwezige opslagmogelijkheden.

Dit heeft een aantal consequenties voor gedistribueerde Java/Smalltalk applicaties. Eén van de consequenties is dat de Smalltalk server op dezelfde computer moet worden opgenomen als de webserver. Een tweede consequentie is dat de gedistribueerde Java/Smalltalk applicaties die ontwikkeld gaan worden beperkt zijn in functionaliteit doordat er geen gebruik gemaakt mag worden van de lokaal aanwezige opslagmogelijkheden.
Doordat de Java applet geen gebruik mag maken van de harde schijf op de client en maar met één enkel ander systeem mag communiceren is deze ongeschikt om als server te fungeren. Bij de hier beschouwde gedistribueerde Java/Smalltalk applicaties zal de computer waarop het Java gedeelte uitgevoerd wordt altijd als client fungeren. Hierdoor zal het gebruik maken van diensten altijd één kant op gericht zijn, namelijk de Java client zal gebruik maken van diensten die de Smalltalk server te bieden heeft.

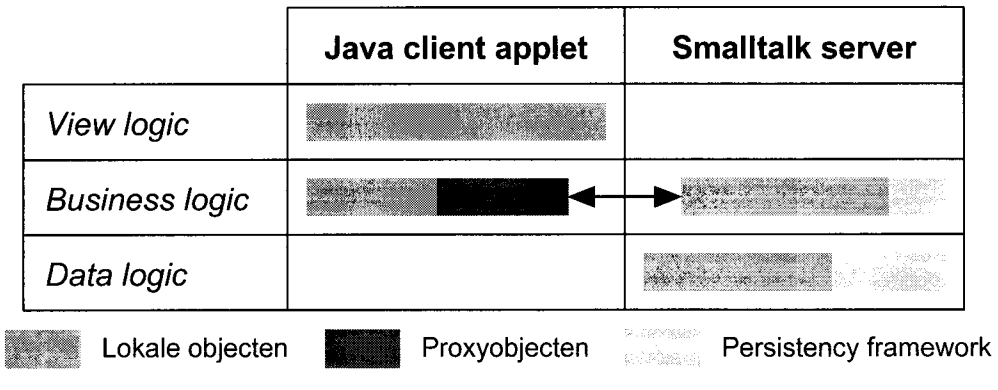
§2.3 Opbouw van een gedistribueerde Java/Smalltalk applicatie

Een gedistribueerd Java/Smalltalk systeem dat gebruik maakt van het persistency framework wordt opgebouwd volgens Figuur 4. Een webserver voorziet de webbrowser in de gevraagde internetpagina's. Als er in de gevraagde internetpagina's een Java applet is opgenomen, zal een Java-enabled webbrowser deze opstarten.



Figuur 4 Opbouw van een gedistribueerd Java/Smalltalk systeem

In deze Java applet is de benodigde client software opgenomen die een verbinding tussen de Java applet en de Smalltalk server mogelijk maakt. De Smalltalk server maakt gebruik van het persistency framework om een koppeling te leggen met een database. Een gedistribueerde Java/Smalltalk applicatie is het geheel van Java client applet, Smalltalk server en eventueel persistency framework. De opbouw van een gedistribueerde applicatie gebeurt conform een drie lagen model. Dit drie lagen model beschrijft een applicatie in drie lagen. In de eerste laag is alle logica opgenomen die voorziet in de interactie met de gebruiker. Deze laag wordt hierom ook de "**View logic**" genoemd. De tweede laag beschrijft de kern van een applicatie waarin alle business objecten zijn opgenomen. Deze laag wordt vaak "**Business logic**" of "**Domain logic**" genoemd. Alle logica die beschrijft op welke manier de business objecten opgeslagen en opgehaald kunnen worden, zijn opgenomen in de laag die "**Data logic**" wordt genoemd. Figuur 5 laat zien op welke manier bij een gedistribueerde Java/Smalltalk ap-

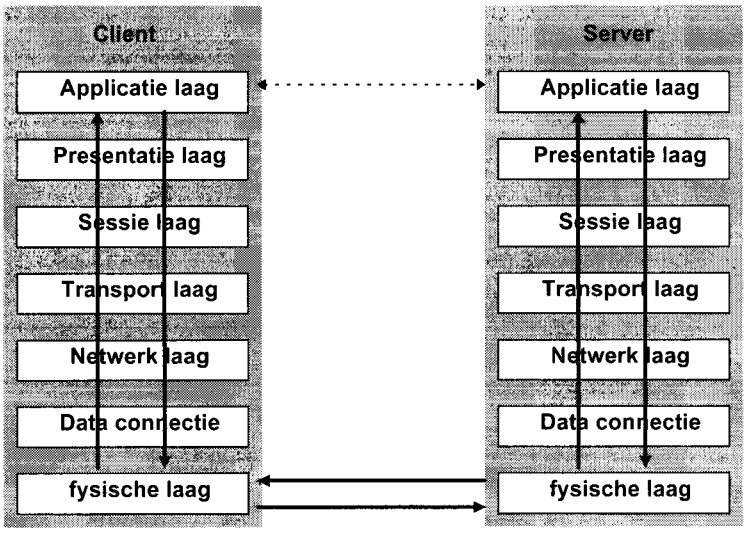


Figuur 5 Opbouw van gedistribueerde Java/Smalltalk applicaties

plicatie, de verschillende lagen verdeeld zijn over de Java client applet en de Smalltalk server. Hierbij moet in acht worden genomen dat de verschillende objecten die nodig zijn om de verbinding te leggen tussen de twee systemen, als mede de objecten die nodig zijn om de communicatie op een gestructureerde manier tot stand te laten komen, aan beide zijde opgenomen zijn in de business logic, en wel als onderdeel van de lokale objecten.

§2.4 De client/server communicatie

De communicatie die plaatsvindt tussen de client en de server, maakt gebruik van het TCP/IP protocol. Dit protocol definieert socket's als zijnde de connectie tussen de client en server. Het is mogelijk om meerdere connecties tegelijkertijd tot stand te laten komen op een enkele computer. Hiertoe zijn poorten geïntroduceerd. Bij elke socket hoort een poort met een bepaald nummer. Sommige poorten worden gebruikt voor bijvoorbeeld het Hypertext Transfer Protocol (HTTP, poort 80) of het File Transfer Protocol (FTP, poort 21). Dit TCP/IP protocol is gekozen omdat dit het enige protocol is dat standaard door Java ondersteund wordt en omdat dit het standaard internet protocol is. Wanneer er een socketverbinding tot stand is gekomen tussen de client en de server is het mogelijk om rechtstreeks met karakterreeksen te communiceren.



Figuur 6 Client/server communicatie aan de hand van het OSI-model

Een veel gebruikt manier om client/server systemen te modelleren is volgens het OSI-model. Het OSI-model geeft een goed inzicht in de manier waarop communicatie tussen de client en server plaatsvind[OSI resources]. Door gebruik te maken van socket's waarover gecommuniceerd wordt volgens het TCP/IP protocol, zijn de onderste vier lagen van het OSI-model al geïmplementeerd. In de applicatielaag worden aan de client zijde de proxyobjecten opgenomen en aan de server zijde de 'echte' objecten. De Smalltalkobjectserver en Javaproxycommunicator dienen nu beide laag vijf en zes van het OSI-model te implementeren. Hiervoor dient een communicatiemodel gedefinieerd te worden, die de te versturen informatie representeert.

Dit communicatiemodel beschrijft de manier waarop de berichten vertaald kunnen worden naar communicatiestrings. In deze berichten kunnen argumenten voorkomen. Deze argumenten moeten ook vertaald worden naar communicatiestrings. Voor het resultaat van dit bericht geldt ook dat deze vertaald moet worden naar een communicatiestring, zodat deze teruggestuurd kan worden naar de client.

Bij het sturen van berichten over een computernetwerk, moet het wel duidelijk zijn naar welk object dit bericht gestuurd moet gaan worden. Hiervoor is elk object dat gedistribueerd wordt, voorzien van een **object identifier**. Object identifiers worden ook door Bennett en Decouchant gebruikt, maar worden daar object-oriented pointers (OOP) genoemd [Bennett 1987, Decouchant 1986]. Deze object identifier dient uniek te zijn binnen de bestaande object identifiers van een bepaalde klasse. Nu is het mogelijk een directe vertaling te maken, van methode aanroepen (berichten) en resultaten van methoden naar een string, die over een netwerk verzonden kan worden.

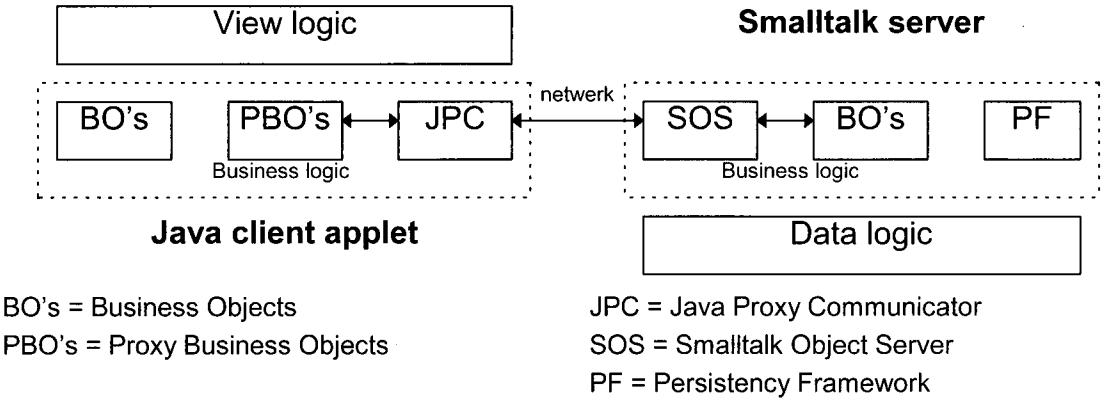
De objecten zoals deze in Java en Smalltalk voorkomen, worden opgedeeld in zogenaamde **primitieve objecten** en **complexe objecten**. Primitieve objecten zijn objecten van het type String, Integer, Float, Boolean e.d.. Complexe objecten zijn objecten waarvan in de andere omgeving geen equivalent bestaat. Complexe objecten zijn bijvoorbeeld de gedistribueerde Smalltalk objecten zoals Persoon, Rekening, Pas e.d. In bijlage 1 zijn drie tabellen opgenomen waarin alle soorten vertalingen van klasse- en instantiemethoden, primitieve en complexe objecten naar **communicatiestrings** voorkomen. De client vertaalt conform dit communicatiemodel de berichten met eventuele argumenten, zodat deze naar de server gestuurd kunnen worden. De server voert een inverse bewerking uit op de ontvangen communicatiestring. Deze inverse bewerking bepaald uit een gegeven communicatiestring de bijbehorende methode aanroep, de opdracht. Hierna kan de server deze opdracht uitvoeren. Het resultaat van deze opdracht wordt dan weer vertaald naar zo'n communicatiestring en teruggestuurd naar de client, die dan op zijn beurt een inverse bewerking uitvoert op de ontvangen communicatiestring.

Voor de client/server communicatie die tot stand moet komen kan ook gebruik worden gemaakt van bestaande Object Request Brokers(ORB's) die CORBA-Compliant zijn[OMG]. Om twee redenen is hier echter van afgezien:

1. Door de hoeveelheid overhead die optreed bij het gebruik van CORBA zal de performance van de gedistribueerde applicaties slechter worden.
2. CORBA-Compliance wil (nog) niet altijd zeggen dat de ORB's van verschillende programmeertalen en platformen ook daadwerkelijk op elkaar aansluiten.

Hoofdstuk 3 Componenten van het framework

Nu de algemene structuur van gedistribueerde Java/Smalltalk applicaties bekend is, kunnen de verschillende componenten zoals deze in het framework voorkomen, onderscheiden worden. Zoals uit het voorgaande hoofdstuk blijkt, bestaat het gedistribueerde systeem uit een Java client applet en een Smalltalk server. De Java client applet bestaat buiten de normale business objecten, ook uit proxy-objecten die automatisch uit de Smalltalk business objecten gegenereerd worden. Hiervoor is een Javaproxygenerator ontwikkeld. In Figuur 7 is de business logica van zowel de Java client applet als de Smalltalk server, opgedeeld in verschillende onderdelen. Zo bestaat de business logica van de Java client applet uit lokale business objecten, proxy business objecten en ondersteunende logica die in het vervolg de Javaproxycommunicator zal worden genoemd.



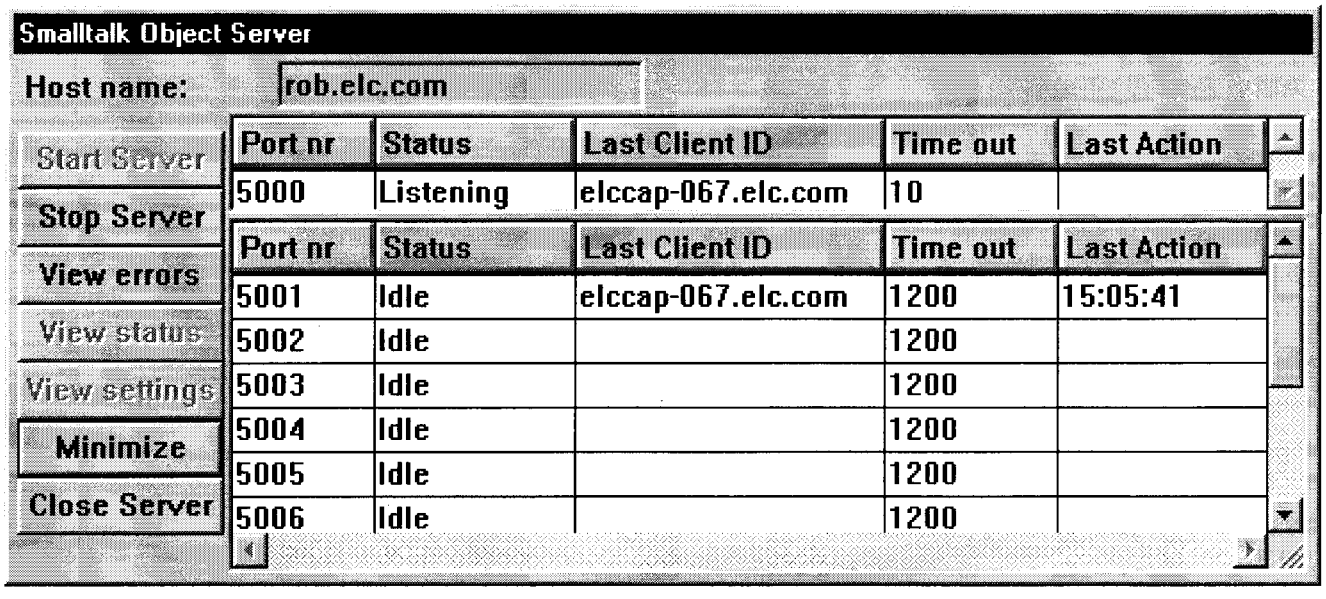
Figuur 7 Opdeling van de business logica

In de business logica aan de Smalltalk zijde kunnen we een Smalltalkobjectserver onderscheiden en de originele business objecten die gedistribueerd zijn naar de Java client applet. In paragraaf 3.1 worden de componenten van de Smalltalkobjectserver besproken en in paragraaf 3.2 de componenten van de Javaproxycommunicator. In paragraaf 3.3 wordt ingegaan op de Javaproxygenerator. Paragraaf 3.4 bespreekt enkele nevenprodukten die ontstaan zijn tijdens het ontwikkelen van het framework.

§3.1 De Smalltalkobjectserver

De Smalltalkobjectserver is in staat om middels het communicatiemodel (paragraaf 2.4) met meerdere clients tegelijkertijd te communiceren. Hiervoor beschikt de server over één algemeen kanaal en meerdere subkanalen. Elk kanaal heeft een nummer dat overeenkomt met het poortnummer van de bijbehorende socket. Een Java client applet die wil communiceren met één van de subkanalen van de server zal dit eerst, via het algemene kanaal, aanvragen. Als er een subkanaal vrij is, zal de server aan de client melden dat hij via dit subkanaal kan gaan communiceren. Voor elk kanaal wordt een

aparte server gestart, er wordt onderscheid gemaakt tussen de zogenaamde **mainportserver** en de **subportserver**. De mainportserver realiseert het algemene aanvraagkanaal van de Smalltalkobjectserver, terwijl de feitelijke communicatie plaatsvindt met behulp van de subportservers. De gebruiker van de Smalltalkobjectserver kan zelf instellen welke poortnummers gebruikt worden voor de verschillende poortservers. Tijdens het gebruik kan van elk willekeurig kanaal de status opgevraagd en gecontroleerd worden. Alle berichten die over het netwerk verstuurd of ontvangen worden kunnen gecomprimeerd/gedecomprimeerd en gecodeerd/gedecodeerd worden. Elke subportserver is uitgerust met een **command translater** welke de communicatiestrings van de client vertaalt, uitvoert, en het resultaat weer vertaalt naar een communicatiestring.



Figuur 8 De Smalltalkobjectserver

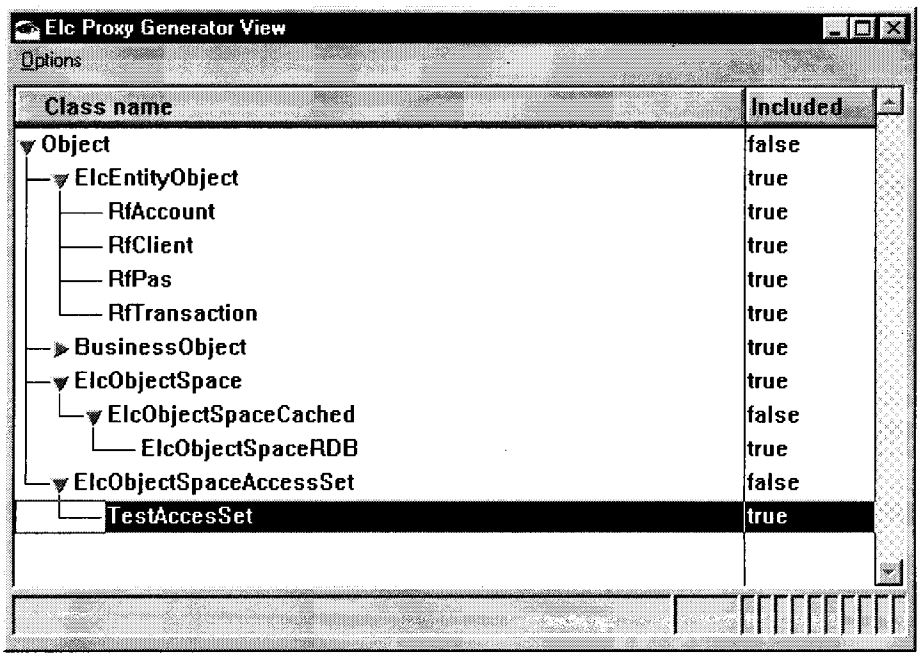
In bijlage 2 staat beschreven op welke manier tot het ontwerp van de Smalltalkobjectserver is gekomen. Figuur 8 laat zien hoe de user interface van de Smalltalkobjectserver eruit ziet. Hierin wordt duidelijk hoe de user interface de toestanden van de mainportserver(boven) en subportservers(onder) weergeeft.

§3.2 De Javaproxycommunicator

Met de Javaproxycommunicator wordt het geheel van klassen bedoeld dat ervoor zorgt dat de Java client applet kan communiceren met de Smalltalkobjectserver. Hiervoor is de Javaproxycommunicator uitgerust met een communicatiemanager die verantwoordelijk is voor het beheren van een communicatiesessie met de server. In deze communicatiemanager zijn de methoden opgenomen om een sessie te openen, te sluiten, data te verzenden en te ontvangen. Ook de Javaproxycommunicator is uitgerust met de klassen om de verstuurde of op te sturen strings te (de)comprimeren of te (de)coderen. De gehele gedistribueerde klasseboom wordt automatisch gekoppeld aan de Javaproxycommunicator. Op welke manier tot het ontwerp van de Javaproxycommunicator is gekomen, staat beschreven in bijlage 3.

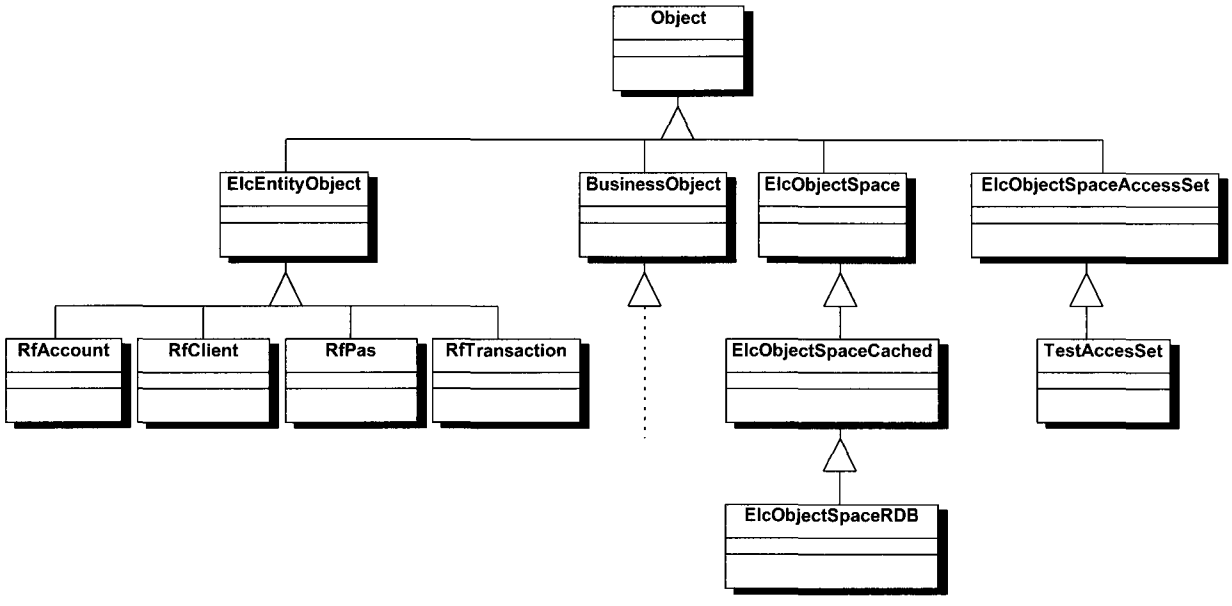
§3.3 De Javaproxygenerator

De Javaproxygenerator wordt tijdens het ontwikkelen van een gedistribueerde Java/Smalltalk applicatie, gebruikt om de benodigde proxyobjecten te genereren uit Smalltalk objecten. Deze Javaproxygenerator stelt de ontwikkelaar in staat om per klasse aan te geven welke methoden gebruikt moeten gaan worden in Java en welk gedrag geërfd moet worden van zijn superklasse. Met de Javaproxygenerator kan een gedistribueerde klasseboom gegenereerd worden. Figuur 9 laat de user interface zien van de Javaproxygenerator.



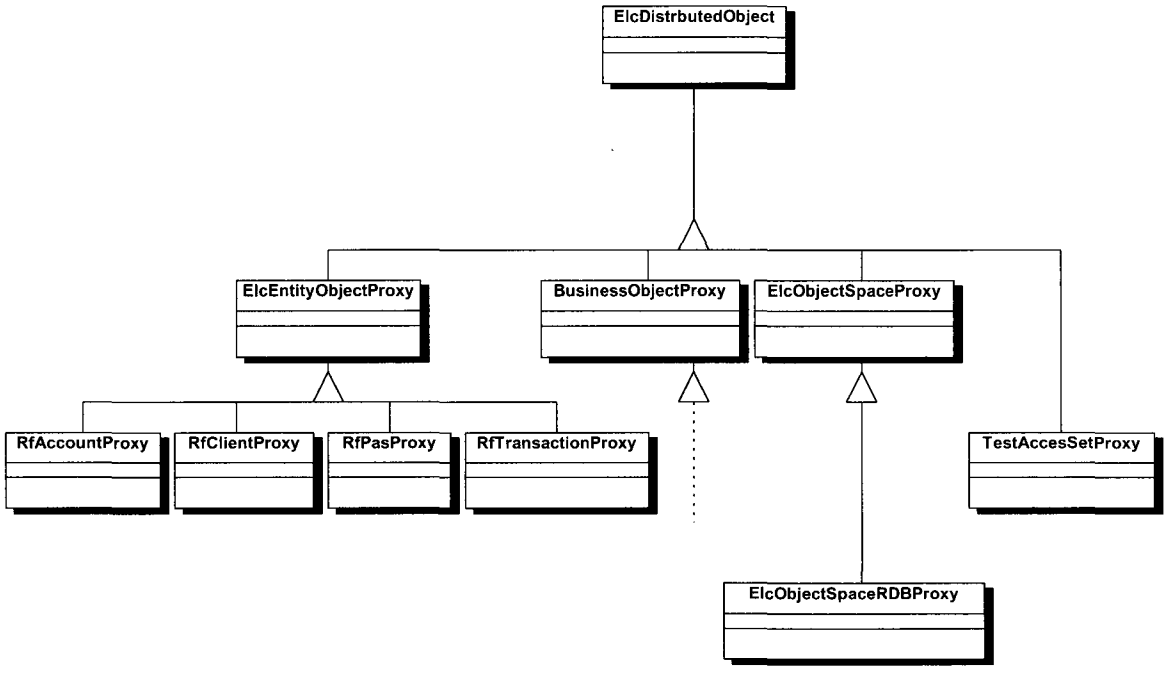
Figuur 9 De Javaproxygenerator

Bij de verschillende klassen moet aangegeven worden of deze in Java al dan niet gebruikt gaan worden. Op deze manier is het bijvoorbeeld mogelijk om niet het gedrag van een superklasse te erven, maar wel weer van de superklasse van de superklasse. De te distribueren klasseboom kan opgeslagen worden, zodat deze niet telkens opnieuw opgebouwd dient te worden. Met één druk op de knop wordt de gedistribueerde Java klasseboom gegenereerd. De volledige beschrijving van het ontstaan van het ontwerp van de Javaproxygenerator is opgenomen in bijlage 4. Om een illustratie te geven van hoe zo'n gedistribueerde klasseboom er uit ziet, beschouwen we Figuur 10 en Figuur 11. Figuur 10 laat een voorbeeld zien van een klasseboom die met de Javaproxygenerator opgebouwd zou kunnen worden. Deze komt toevalligerwijs overeen met de klasseboom zoals deze in Figuur 9 met behulp van de Javaproxygenerator is opgebouwd. In Figuur 10 zijn de methoden van de verschillende klassen niet opgenomen. Van deze klasseboom wordt door middel van de Javaproxygenerator automatisch de gedistribueerde klasseboom gegenereerd die in Java gebruikt kan worden. Deze klasseboom staat afgebeeld in Figuur 11.



Figuur 10 Een klasseboom in Smalltalk

In deze gedistribueerde klasseboom wordt het ook duidelijk hoe het principe van het erven van een superklasse van een superklasse precies gerealiseerd wordt. Figuur 9 laat zien dat de klasse ElcOb-



Figuur 11 De gedistribueerde klasseboom

jectSpaceCached niet toegevoegd dient te worden in de gedistribueerde klasseboom. Om toch de hiërarchie van de klasseboom consistent te houden wordt, ervoor gezorgd dat de klasse ElcObject-SpaceRDBProxy direct erft van klasse ElcObjectSpaceProxy.

De gehele klasseboom wordt aan de Javaproxycommunicator gekoppeld, door de klassen die het hoogst in de boom staan te laten erven van 'ElcDistributedObject'. Deze klasse is onderdeel van de Javaproxycommunicator. In dit voorbeeld zijn dit de klassen 'ElcEntityObject', 'BusinessObjectProxy', 'ElcObjectSpaceProxy', en 'TestAccesSetProxy'. De gedistribueerde klasseboom is nu gereed om in Java gebruikt te worden.

§3.4 Ontwikkelde nevenprodukten

Naast de produkten die bij het distributed Java/Smalltalk framework horen, zijn nog andere produkten ontstaan die het vermelden waard zijn. De ontwikkelde producten zijn de Smalltalkproxygenerator, de Smalltalkproxycommunicator en de distributed Java applet.

§3.4.1 Smalltalkproxygenerator

De Smalltalkproxygenerator heeft dezelfde functie als de hiervoor besproken Javaproxygenerator. Met behulp van de Smalltalkproxygenerator kan een klasseboom opgebouwd worden die in één keer de Smalltalk proxy klasseboom genereert.

§3.4.2 Smalltalkproxycommunicator

De Smalltalkproxycommunicator heeft dezelfde functie voor de Smalltalk proxy klasseboom als de Javaproxycommunicator voor de Java proxy klasseboom. De Smalltalkproxycommunicator zorgt voor de koppeling van de Smalltalk proxyobjecten met de objecten die via de Smalltalkobjectserver benaderd kunnen worden.

§3.4.3 Java persistency applet

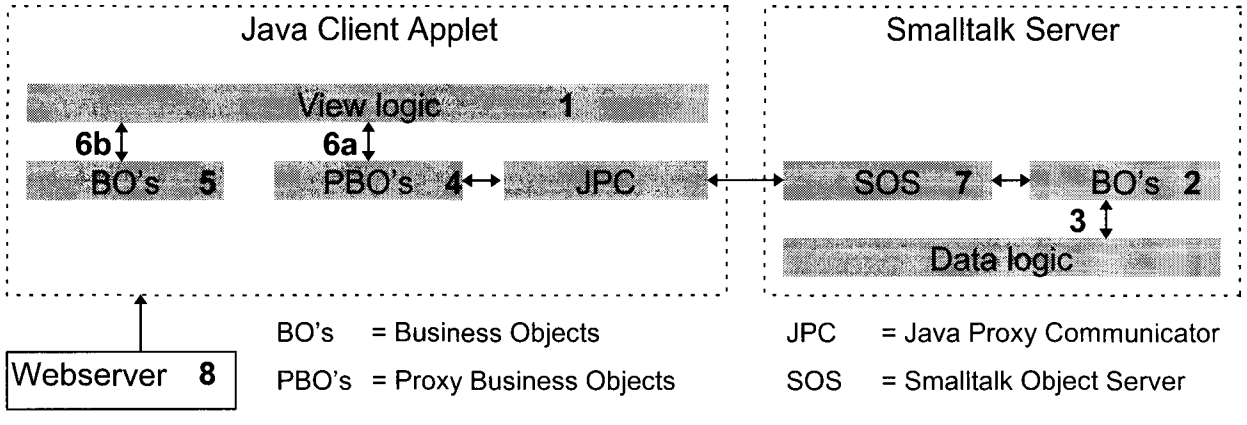
Als er in Java gebruik gemaakt gaat worden van het persistency framework dient een gebruiker zich aan te melden aan de database. Om ervoor te zorgen dat de ontwikkelaar niet telkens opnieuw de code hoeft op te nemen om een gebruiker aan te melden aan de database, is er een PersistencyApplet ontwikkeld die hierin voorziet. Wil de ontwikkelaar een Java applet ontwikkelen die gebruik maakt van het persistency framework, dan dient deze applet een subklasse te zijn van 'ElcPersistencyApplet'. Wanneer dan deze applet gestart wordt, vindt de validatie van een gebruiker automatisch plaats.

Hoofdstuk 4 Gebruik van het framework

Nu het ontwerp van de verschillende componenten van het framework bekend is, kan er besproken worden op welke manier een gedistribueerde Java/Smalltalk applicatie opgezet moet worden. Hiertoe wordt eerst besproken op welke manier gedistribueerde applicaties opgezet moeten worden die zonder het persistency framework werken. Hierna wordt besproken op welke manier het persistency framework gekoppeld kan worden aan het distributed Java/Smalltalk framework.

§4.1 Het opzetten van een gedistribueerde Java/Smalltalk applicatie

Figuur 12 laat zien in welke volgorde een gedistribueerde Java/Smalltalk applicatie opgezet moet



Figuur 12 De stappen bij het opzetten van een gedistribueerde Java/Smalltalk applicatie

worden. Hieronder zullen de opeenvolgende stappen besproken worden.

Stap 1 Ontwikkeling van Java applet user interface

Als eerste wordt de user interface ontwikkeld van de Java applet. Hiervoor zijn verschillende ontwikkel omgevingen beschikbaar zoals: Symantec's Cafe, Symantec's Visual Cafe, VisualAge for Java en Microsoft's Visual J++. Aan de hand van de verschillende schermen kunnen de benodigde business objecten bepaald worden.

Stap 2 Ontwerpen en implementeren van de business objecten in Smalltalk

De volgende stap is het bepalen van de business logica die gedistribueerd dient te gaan worden. Dit zullen voornamelijk objecten zijn die aan de Smalltalk zijde persistent kunnen worden gemaakt. Maar ook andere objecten met een speciale functie kunnen ontworpen en gedistribueerd worden. Hierbij zou bijvoorbeeld gedacht kunnen worden aan een object dat een complexe berekening uitvoert die

voor de Java applet een grote belasting zou betekenen, wanneer deze in Java zelf geïmplementeerd zou zijn.

Stap 3 Realiseer de koppeling tussen de business objecten en de data logic
Nu de business objecten gerealiseerd zijn, is het mogelijk om de objecten die persistent gemaakt dienen te worden, te koppelen aan een bepaalde database. Vaak worden de business objecten ook ontworpen aan de hand van een bestaande database.

Stap 4 Genereren van de gedistribueerde objecten
De volgende stap is het opbouwen van de te distribueren klasseboom met behulp van de Javaproxy-generator. Per klasse wordt aangegeven welke instantie– en klassemethoden gedistribueerd dienen te worden. Deze klasseboom kan opgeslagen worden voor hergebruik. Als later blijkt dat de business objecten nog functionele uitbreiding nodig hebben, kunnen de betreffende methoden altijd toegevoegd worden aan de te distribueren methoden. Nadat de klasseboom is opgebouwd, wordt met een druk op de knop de gedistribueerde Java klasseboom gegenereerd.

Stap 5 Het ontwikkelen van de lokale business objecten in Java
Waarschijnlijk zullen niet alle in Java voorkomende business objecten gedistribueerde proxyobjecten zijn, maar zullen er ook 'lokale' objecten geïmplementeerd dienen te worden. Dit gebeurt in stap 5.

Stap 6 Aanbrengen van de koppeling tussen de view logica en business logica in Java
De belangrijkste en laatste stap in het ontwikkelproces is het koppelen van de in Java ontwikkelde schermen aan de business logica die bestaat uit lokale en gedistribueerde objecten. Met behulp van de binnenkort te verschijnen VisualAge for Java kan dit grafisch gebeuren, maar deze koppeling kan ook tot stand komen door codering van het gedrag van de verschillende schermen.

Stap 7 & 8 Het in bedrijf brengen van de gedistribueerde applicatie
Nadat alle hierboven beschreven stappen uitgevoerd zijn, volgt het opstarten van het systeem. De Web server en Smalltalkobjectserver zijn op dezelfde computer gelokaliseerd. De Smalltalkobjectserver wordt als eerste gestart. De Webserver, die een link naar de pagina met de Java applet bevat, kan hierna gestart worden en het systeem is klaar voor gebruik.

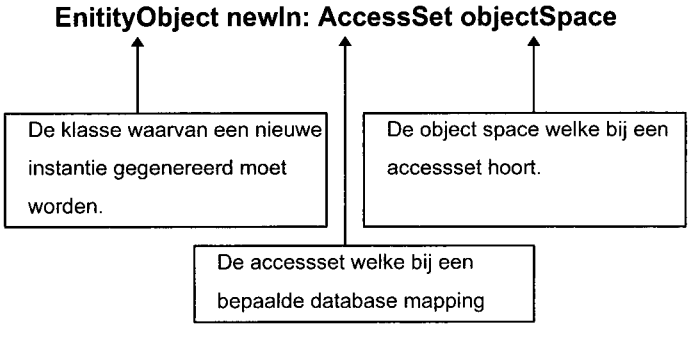
§4.2 Persistency framework ondersteuning

Het persistency framework van ELC Object Technology voorziet in het persistent maken van business objecten. Het persistency framework ondersteunt verschillende soorten databases. De objecten die middels dit framework persistent gemaakt kunnen worden, worden **entiteit objecten** genoemd. Deze entiteit objecten worden 'gemapped' op de kolommen en rijen van een database. Deze mapping is

opgenomen in een **access set**. Hier wordt dus de koppeling beschreven tussen de business objecten en de rijen en kolommen van verschillende tabellen van de database. Elke door de gebruiker aangemaakte access set bezit een zogenaamde **objectspace** waar alle objecten, die opgehaald of naar de database geschreven worden, zich bevinden. Het persistency framework voorziet ook in een transactie ondersteuning. Alle wijzigingen die aangebracht worden aan de entiteit objecten en het creëren van nieuwe instanties van deze entiteit objecten vinden plaats in zo'n transactie. De methoden die voor deze transactie ondersteuning zorgen, zijn opgenomen in de klasse ObjectSpace. Voor een uitvoerige beschrijving van het persistency framework zie [ELC 1995] en [ELC 1996].

Als er vanuit Java gebruik gemaakt moet gaan worden van het persistency framework dan hebben we te maken met drie verschillende klassen. Als eerste de **entiteit objecten** die gedistribueerd dienen te gaan worden. Vervolgens de **access set** waarin de entiteit objecten gemapped worden op de database. Als laatste de **objectspace** die gebruikt wordt in de access set. Om dit te verduidelijken eerst een voorbeeld.

Als er een nieuwe instantie van een entiteit object gemaakt moet worden, dan moet hierbij aangegeven worden in welke objectspace deze moet worden toegevoegd. Figuur 13 laat zien hoe dit in Smalltalk gebeurt.



Figuur 13 Instantieren van een entiteit object in Smalltalk

De nieuwe instantie van de klasse wordt dus aangemaakt in een objectspace, en wel de objectspace die bij een bepaalde access set hoort. Als we in Java hetzelfde zouden willen doen moeten er proxyobjecten zijn van de entiteit objecten, van de access set en van de objectspace. Figuur 9 laat een voorbeeld zien van een klasseboom die gedistribueerd wordt. In de boom zijn de subklassen van ElcEntityObject de entiteit objecten, dit zijn dus RfAccount, RfClient, RfPas en RfTransaction. De access set die gedistribueerd moet worden, is een subklasse van ElcObjectSpaceAccessSet, in dit geval TestAccesSet. Afhankelijk van het type van de database maakt de access set gebruik van een bepaald type objectspace. Deze verschillende soorten objectspaces zijn subklassen van ElcObjectSpace. In dit geval wordt ElcObjectSpaceRDB gebruikt. RDB staat voor **R**elationele **D**ata**B**ase.

Van deze klassen worden nu de proxyobjecten: RfClientProxy, RfAccountProxy, RfPasProxy, RfTransactionProxy, TestAccesSetProxy en ElcObjectSpaceRDBProxy gegenereerd. Hieronder volgt een stukje Java code om te laten zien hoe dit nu gebruikt kan worden(Figuur 14).

```
ElcObjectSpaceRDBProxy objectSpace;
RfClientProxy klant;

objectSpace= TestAccesSetProxy.objectSpace();
klant= RfClientProxy.newIn(objectSpace);
```

Figuur 14 Instantiëren van een klasse in Java

In de eerste twee regels worden de variabelen *objectSpace* en *klant* gedeclareerd. Hierna wordt de variabele *objectSpace* gelijk gemaakt aan de objectspace horende bij TestAccesSetProxy. Als de objectspace bekend is, kan een nieuwe *RfClientProxy* aangemaakt worden. Dit gebeurt in de laatste regel. Met behulp van deze proxies van access sets en objectspaces is het mogelijk om gebruik te maken van het persistency framework.

Alle methoden die voor de transactie ondersteuning zorgen in het persistency framework zijn opgenomen in de klasse ElcObjectSpace. Om in Java gebruik te maken van deze transactie ondersteuning moeten de bijbehorende methoden opgenomen worden in de te distribueren klasseboom.

§4.3 Specifieke eigenschappen van het framework

Bij het ontwikkelen van het framework is getracht de overeenkomsten tussen lokale en gedistribueerde objecten zover mogelijk door te voeren, zodat er in Java geen onderscheid gemaakt kan worden tussen deze objecten. Er zijn echter een aantal eigenschappen van Java die beperkingen hebben opgelegd aan het framework.

§4.3.1 Een klasse als argument

In de Javaproxycommunicator worden standaard **perform** methoden gebruikt om de berichten door te sturen naar Smalltalk(bijlage 3.2). Alle argumenten van perform methoden en het resultaat van deze methoden zijn van het type **Object**. Dit wil zeggen dat alle argumenten die in de perform methoden voor kunnen komen, een subklasse zijn van de Object klasse. Omdat in Java de klasse **Class** geen subklasse is van de klasse Object, is het bijvoorbeeld niet mogelijk om een klasse als argument mee te sturen. Dit is opgelost door een nieuwe klasse te introduceren onder de naam: 'ElcClassArgument'. Deze ElcClassArgument klasse heeft één attribuut en dat is de naam van de klasse die als argument meegestuurd moet worden. Er zou bijvoorbeeld een Smalltalk methode kunnen zijn:

```
collection := RfClient returnAllInObjectSpaceOf: TestAccesSet.
```

die alle objecten van het type RfClient in een bepaalde objectspace van een access set als collectie terug geeft. Hierin is TestAccesSet een klasse die als argument wordt meegestuurd. In Java zou dit de volgende code opleveren:

```
collection = RfClientProxy.returnAllInObjectSpaceOf(TestAccesSetProxy);
```

Maar omdat TestAccesSetProxy een klasse is, kan deze niet als argument worden gebruikt. Hiervoor de volgende oplossing:

```
ElcClassArgument argument = new ElcClassArgument();
argument.name("TestAcesSet");
collection = RfClientProxy.returnAllInObjectSpaceOf(argument);
```

Voor een klasse als argument is een aparte code opgenomen in het communicatiemodel, zodat dit op een juiste wijze vertaald kan worden naar een communicatiestring. Bij de naam van de klasse moet de postfix 'Proxy' weggelaten worden.

§4.3.2 Het starten en afsluiten van een communicatiesessie

De communicatiesessie met de server wordt automatisch gestart als ervoor de eerste keer een bericht wordt gestuurd naar een proxyobject. Het afsluiten van de communicatiesessie kan niet automatisch gebeuren omdat er nergens in een applicatie bekend is of een bepaalde actie de laatste is. Om een communicatiesessie af te sluiten moet een methode aangeroepen worden van *DistributedObject* welke *endAll* heet. Deze kan het beste in de destructor methode van een applet geplaatst worden. De destructor methode wordt bij het afsluiten van een Java applet uitgevoerd. Wanneer de gebruiker de Java applet afsluit, wordt automatisch de communicatiesessie met de server afgesloten. De methode *endAll* is een klasse methode die naar een willekeurig gedistribueerd object gestuurd kan worden.

§4.3.3 Noodzakelijke type-casting

Elke methode die opgenomen is in een gedistribueerd object, geeft een object terug van het type **Object**. Stel dat het resultaat van een methode een String is, dan zal deze toch geretourneerd worden als Object. Om het resultaat toch verder als string te gebruiken, is er expliciete **type-casting** nodig. Hiertoe eerst het volgende voorbeeld:

```
ClientProxy klant;
String naam;
klant = ClientProxy.newObject();
klant.name("Rob");
naam=klant.name();
```

Dit stukje code zal een foutmelding opleveren tijdens het compileren. Dit komt omdat ClientProxy een gedistribueerd object is en al zijn methoden als resultaat een object hebben van het type Object. Hier wordt dus geprobeerd om een object van het type Object toe te kennen aan een String, dit gaat niet. Dit geldt ook voor het toekennen van de klant aan een nieuwe ClientProxy. Dit stukje code zou als volgt moeten zijn:

ClientProxy klant;
String naam;
klant = ***(ClientProxy)****ClientProxy.newObject();*
klant.name("Rob");
*naam=****(String)****klant.name();*

Hier is dus een expliciete type-casting gebruikt om het toekennen van een Object aan een String en van een Object aan een ClientProxy te valideren. Bij het gebruik van gedistribueerde objecten zal deze type-casting dus continu door de applicatie heen, gebruikt moeten worden. Omdat Smalltalk een ongetypeerde taal is, kan dit wel gevaarlijke situaties opleveren. Het zou namelijk voor kunnen komen dat het verwachte resultaat van een methode een string is, terwijl er een integer terug wordt gegeven. Als hier dan een type-casting plaatsvindt naar een string, levert dit een foutmelding op. Het is aan de ontwikkelaar om ervoor te zorgen dat de type-casting ook altijd mogelijk is.

§4.3.4 Creëren van een nieuwe instantie van een proxyobject

In object georiënteerde programmeertalen wordt het verloop van het programma bepaald door het sturen van berichten. In Java zijn hier een paar uitzonderingen op. Het creëren van een nieuwe instantie van een klasse in Java gebeurt niet door het bericht ***new*** er naar toe te sturen, maar door het voor de klasse te plaatsen, bijvoorbeeld:

klant = new ClientProxy();

De methode new bestaat dan ook niet echt. Dit levert een probleem op bij het creëren van nieuwe instanties van proxyobjecten. Als er in Java een nieuwe instantie gecreëerd wordt van een proxy klasse, dan moet in Smalltalk een nieuwe instantie van het 'echte' object gecreëerd worden. Omdat ***new*** niet als methode is geïmplementeerd, is het niet mogelijk deze opnieuw te implementeren. Dit is opgelost door een klasse methode te implementeren die ***newObject*** heet. Deze methode wordt gebruikt om nieuwe instanties van proxyobjecten te creëren, dus:

klant = ClientProxy.newObject();

Deze methode zorgt ervoor dat er in Smalltalk een nieuwe instantie gecreëerd wordt van de klasse **Client**.

Hoofdstuk 5 Conclusies

Het distributed Java/Smalltalk framework bestaat uit een Smalltalkobjectserver, een Javaproxycommunicator en een Javaproxygenerator. Met behulp van de Javaproxygenerator kunnen tijdens de ontwikkeling op eenvoudige wijze proxyobjecten gegenereerd worden van Smalltalk objecten. Deze proxyobjecten worden automatisch gekoppeld aan de Javaproxycommunicator. De Javaproxycommunicator regelt de communicatiesessie en de feitelijke communicatie met de Smalltalkobjectserver. De Smalltalkobjectserver verwerkt commando's die van meerdere clients afkomstig kunnen zijn.

Met behulp van het framework is het eenvoudig gedistribueerde Java/Smalltalk applicaties te ontwikkelen. Dit geldt zeker voor Smalltalk ontwikkelaars, waar dit framework voor bedoeld is. Met de binnenkort te verschijnen ontwikkelomgevingen van Java wordt het mogelijk om de gedistribueerde objecten grafisch aan de user interface te koppelen zodat codering in Java tot een minimum beperkt kan worden. Een groot gedeelte van de functionaliteit kan in Smalltalk ontwikkeld en getest worden alvorens deze functionaliteit te distribueren naar Java.

Het framework is vooral geschikt om deelsystemen van een bestaande Smalltalk omgeving, beschikbaar te maken via het internet. De performance van gedistribueerde Java/Smalltalk applicaties is voornamelijk afhankelijk van de hoeveelheid informatie die via het netwerk getransporteerd wordt en de snelheid van de verbinding op dat moment. Bij intranet toepassingen hoeft hier bijna geen rekening mee gehouden te worden, maar de performance van internet toepassingen is gebaat bij gedoseerd gebruik van de hoeveelheid te versturen informatie. Het comprimeren van lange berichten kan ook bijdragen aan de performance. Om de berichten die over het netwerk verstuurd worden te beveiligen, kunnen deze van een codering worden voorzien. De compressie– en coderingsmethodieken kunnen op eenvoudige wijze uitgebreid worden.

Een nadeel van de huidige implementatie is dat de Smalltalkobjectserver in een enkel operating systeem proces draait. Dit proces krijgt een bepaald percentage van de totale processortijd toegewezen. Omdat de server gebruik maakt van subprocessen voor elke poort, zullen deze weer een fractie van dit percentage aan processortijd toegewezen krijgen. Hierdoor zal de performance van de server bij meerdere gebruikers snel afnemen. Dit is op te lossen door ervoor te zorgen dat voor elke poort een apart operating systeem proces wordt gestart, dat geen subproces is van de Smalltalkobjectserver. Dit is echter nog niet geïmplementeerd.

Met het **distributed Java/Smalltalk framework** is ELC Object Technology in staat nieuwe diensten aan te bieden aan haar klanten. Met de verbetering van de infrastructuur van het internet zal het mogelijk worden steeds meer complexe applicaties via het internet aan te bieden.

Bijlagen

Bijlage 1 Het communicatiemodel

Alle methoden die aangeroepen worden, de argumenten die hierin voorkomen en de resultaten van de methoden moeten vertaald kunnen worden naar communicatiestrings. Deze communicatiestrings kunnen verstuurd worden van de client naar de server en andersom. Hiervoor is een model nodig, zodat aan beide zijden van het client/server systeem de nodige vertaalslagen gemaakt kunnen worden. Er kunnen drie verschillende soorten conversies onderscheiden worden. Als eerste de conversies van de objecten naar communicatiestrings. Het tweede soort betreft de conversie van instantiemethoden naar communicatiestrings. Als laatste onderscheiden we de conversie van klassemethoden naar communicatiestrings.

1.1 Conversies van objecten

In Tabel 1 zijn de conversies opgenomen van objecten naar communicatiestrings. Deze conversies zijn inverteerbaar. Deze communicatiestring representaties van objecten worden gebruikt om resultaten van methoden te representeren, maar ook om de argumenten zoals deze in de methoden kunnen voorkomen, te representeren.

Tabel 1 Conversies van primitieve en complexe objecten

Object type	Code	Voorbeeld	Communicatiestring
Undefined Object	0	nil / null	"0"
String	1 (string)	'Boslaan 3'	"1 (Boslaan 3)"
Integer	2 integer	10	"2 10"
Boolean	3 boolean	true	"3 true"
Float	4 float	1.23456	"4 1.23456"
Character	7 character	'd'	"7 d"
Klasse	253 ClassName	Client	"253 Client"
Collection of objects	254 length (objects)	(Client(1),true,1.23)	"254 3 (255 Client(1) 3 true 4 1.23)"
Complex Object	255 ClassName(oid)	Client met oid=101	"255 Client(101)"

Deze tabel kan naar wens uitgebreid worden met objecten waarvan blijkt dat er een speciaal soort representatie voor nodig is. Als bijvoorbeeld blijkt dat de representatie van een Float object niet toereikend is, kan hiervoor bijvoorbeeld een object Fraction geïntroduceerd worden dat dit reële getal voorstelt. Elk reëel getal is namelijk voor te stellen als een breuk.

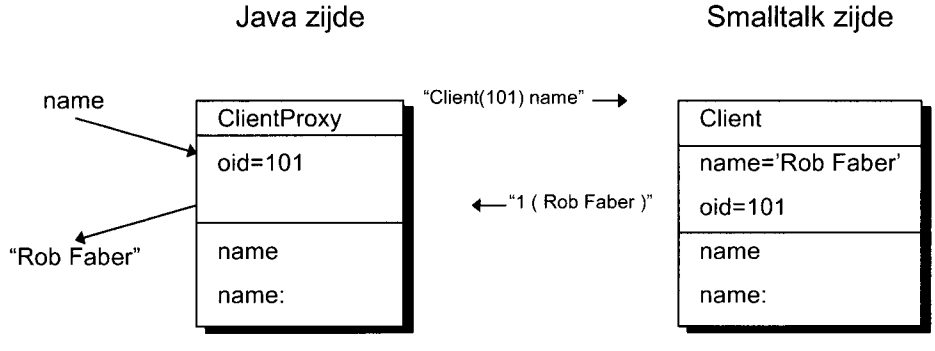
1.2 Conversies van instantie- en klassemethoden

Er zijn twee soorten methoden, te weten **instantiemethoden** en **klassemethoden**. Het verschil in de representatie van instantie- en klassemethoden zit in de object identifier. Een object identifier van een object is een uniek nummer waaronder een bepaald object bekend is bij de Smalltalkobjectserver. Bij een klassemethode bestaat er nog geen instantie van de klasse, geen object, dus is er ook nog geen object identifier. In Tabel 2 zijn de conversies opgenomen van de instantiemethoden naar communicatiestrings.

Tabel 2 Instantiemethode conversies

Args	Code	Voorbeeld	Communicatiestring
0	ClassName(oid) selector	klant name	"Client(101) name"
1	ClassName(oid) selector1: arg1	klant name: 'Rob Faber'	"Client(101) name: 1 (Rob Faber)"
2	ClassName(oid) selector1: selector2: arg1 arg2	klant name: 'Rob Faber' address: 'boslaan 1'	"Client(101) name:address: 1 (Rob Faber) 1 (boslaan 1)"
3	ClassName(oid) selector1: selector2:selector3: arg1 arg2 arg3	klant name: 'Rob Faber' address: 'boslaan1' city: 'Eindhoven'	"Client(101) name:address:city: 1 (Rob Faber) 1 (boslaan 1) 1 (Eindhoven)"
>3	ClassName(oid) selector: (collection of arguments)	klant vullAll: ('Rob Faber', 'boslaan 1','Eindhoven','5611GH')	"Client(101) vulAll: 254 4 (1 (Rob Faber) 1 (Boslaan 1) 1 (Eindhoven) 1 (5611GH))"
Bij alle voorbeelden moet in acht worden genomen dat klant een instantie is van Client met een object identifier(oid=101)			

Ter verduidelijking van bovenstaande conversies is in Figuur 15 een voorbeeld van deze conversies opgenomen. Aan de Java zijde wordt het bericht *name* gestuurd naar een instantie van de klasse



Figuur 15 Instantie methode conversie

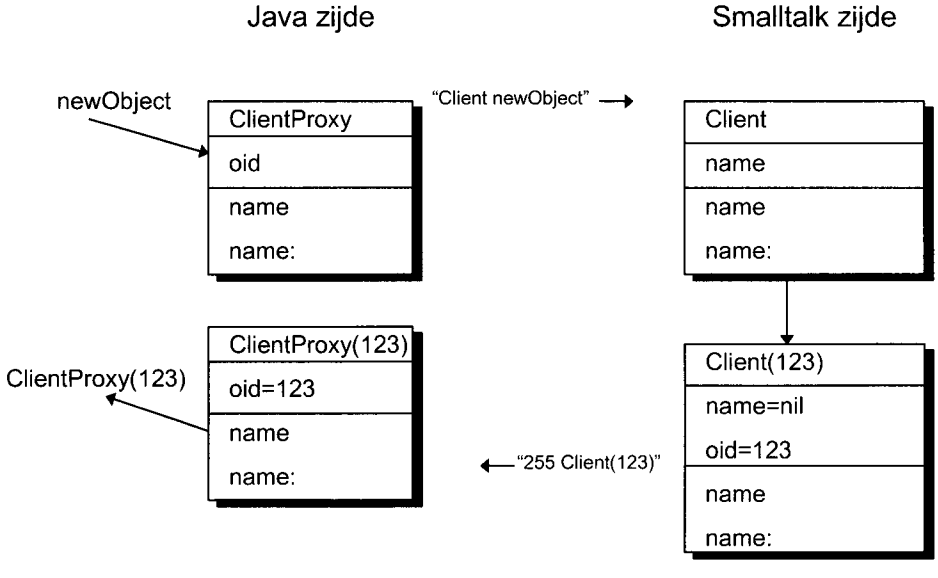
ClientProxy. Deze instantie van de klasse ClientProxy heeft een object identifier gelijk aan de instantie van de klasse Client die in Smalltalk aanwezig is. Het bericht *name* kan nu vertaald worden naar de communicatiestring "Client(101) name" die in Smalltalk naar het object gestuurd wordt met de oid=101 en klasse=Client. Het resultaat "Rob Faber" wordt wederom vertaald naar een communicatiestring, "1 (Rob Faber)". Aan de Java zijde wordt deze communicatiestring weer vertaald naar een string en de methode aanroep heeft als resultaat "Rob Faber".

Een soortgelijke conversie tabel is ook op te stellen voor de klassemethoden. Deze conversie tabel staat afgebeeld in Tabel 3.

Tabel 3 Klassemethode conversies

Args	Code	Voorbeeld	Communicatiestring
0	ClassName selector	Client new	"Client new"
1	ClassName selector1: arg1	Client withName: 'Rob Faber'	"Client withName: 1 (Rob Faber)"
2	ClassName selector1: selector2: arg1 arg2	Client withName: ' Rob Faber ' andAddress: 'boslaan 1'	"Client withName:andAddress: 1 (Rob Faber) 1 (boslaan 1)"
3	ClassName selector1: selector2: selector3: arg1 arg2 arg3	Client withName: 'Rob Faber' andAddress: 'boslaan1' andZip: '2522VA'	"Client withName:andAddress:andZip: 1 (Rob Faber) 1 (Boslaan 1) 1 (2522VA) "
>3	ClassName selector: (collection of arguments)	Client met:: ('Rob Faber' 'boslaan 1' '2522VA' 'Eindhoven')	"Client met:: 254 4 (1 (Rob Faber) 1 (boslaan 1) 1 (2522VA) 1 (Eindhoven))

Deze tabel wordt verduidelijkt aan de hand van het voorbeeld in Figuur 16. In Java wordt de klasse methode *newObject* aangeroepen van de klasse ClientProxy. Dit bericht en de klasse waar dit bericht naar toe wordt gestuurd, wordt conform Tabel 3 vertaald naar de communicatiestring *"Client newObject"*. De Smalltalkobjectserver zal dit bericht doorsturen naar de klasse Client en het resultaat, nl. het creëren van een nieuwe instantie van de klasse Client, wordt als resultaat teruggegeven. Deze instantie krijgt een oid(=123) die dient om de instantie te identificeren. Dit resultaat wordt vertaald naar de communicatiestring "255 Client(123)", waarna er aan de Java zijde een nieuwe instantie gecreëerd kan worden van de klasse ClientProxy. De object identifier van deze klasse wordt dan tevens op 123 gezet. Deze nieuwe instantie wordt als resultaat van de methode *newObject* teruggegeven.



Figuur 16 Klasse methode conversie

Het resultaat van een klassemethode kan zijn dat er een nieuwe instantie van een bepaalde klasse gecreëerd wordt. Als het resultaat van een methode een complex object is, dan wordt deze omgezet naar een proxyobject in Java. De proxyobjecten moeten dan natuurlijk wel bekend zijn in Java. Resumerend, alle complexe klassen die als resultaat voorkomen bij methode aanroepen van een bepaalde klasse, dienen ook gedistribueerd te worden. De primitieve objecten zijn natuurlijk als normale klassen bekend. Het resultaat van een klassemethode kan ook zijn dat deze zichzelf als resultaat heeft, in dat geval moet er natuurlijk geen nieuwe instantie gecreëerd worden. Alleen als de object identifier van een bepaald object afwijkt van de object identifier van het huidige object, wordt er een nieuwe instantie gecreëerd.

Bijlage 2 Ontwerp van de Smalltalkobjectserver

2.1 Analyse

2.1.1 Probleembeschrijving

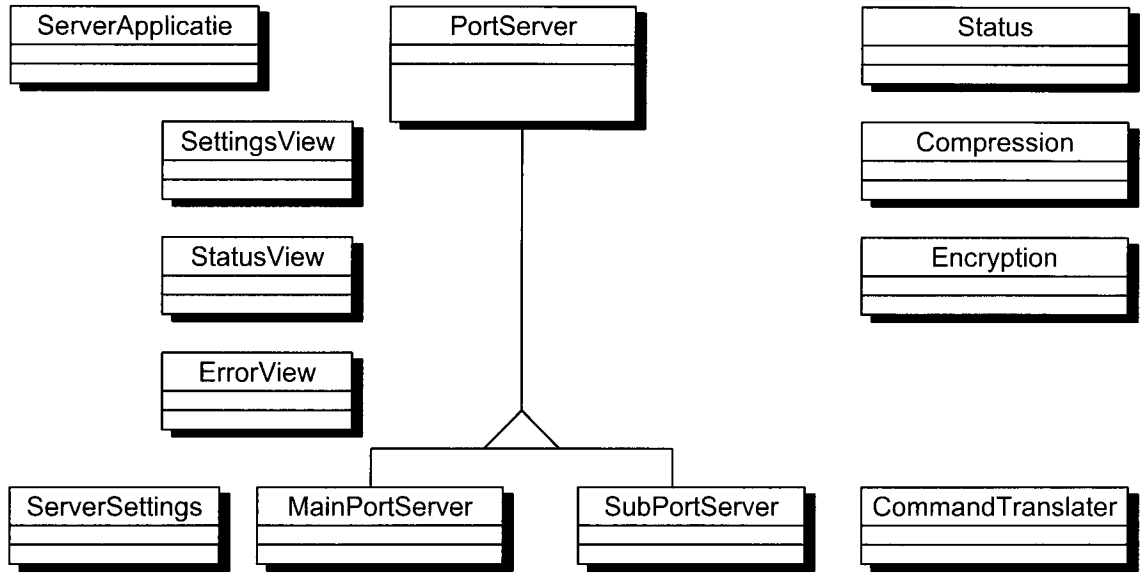
De Smalltalkobjectserver applicatie moet een multi-user server worden. Het is mogelijk om meerdere verbindingen te maken met een enkele host. Hiertoe worden poortnummers gedefinieerd. Als er met behulp van het TCP/IP protocol een connectie (socket verbinding) gemaakt moet worden dient dit poortnummer gespecificeerd te zijn. Deze poortnummers moeten dynamisch worden toegewezen aan de clients die een connectie met de Smalltalkobjectserver willen opzetten. Hiervoor dient een **mainportserver** geïmplementeerd te worden en moet deze gebruik kunnen maken van meerdere **subportservers**. De mainportserver heeft als enige taak om de aanvraag voor communicatie van een client te verwerken. De mainportserver 'kijkt' welke subport vrij is en verwijst de client door naar deze vrije poort. Voor de vrije poort wordt dan een subportserver opgestart, waarmee de rest van de communicatie gaat plaatsvinden. Op deze manier worden de zogenaamde subpoorten dynamisch toegewezen. De Smalltalkobjectserver kent instellingen die verandert moeten kunnen worden. Zo moeten de kanaalnummers ingesteld kunnen worden van de main– en subportservers en moet de hostnaam veranderd kunnen worden. Van elk kanaal moet een status bijgehouden worden. Een kanaal kent verschillende toestanden, te weten:

- het kanaal is vrij,
- het kanaal is bezig met het uitvoeren van een commando,
- het kanaal kent een fout of,
- het kanaal is aan het luisteren in afwachting van een commando.

Verder moeten er mogelijkheden geïmplementeerd worden om de data die verstuurd wordt, te comprimeren en/of van een codering te voorzien. De data die ontvangen wordt, moet kunnen worden gedecodeerd en gedecomprimeerd. Verder moet elke subportserver uitgerust zijn met een commando vertaler, die de binnenkomende commando's vertaalt naar methode aanroepen in Smalltalk. Deze vertaler moet ook het resultaat van de commando's weer kunnen vertalen in een communicatiestring. Als het resultaat van een methode een complex object is, moet dit object opgeslagen worden in een cache, zodat de server bij volgende verwijzingen naar dit object, deze ook kan achterhalen.

2.1.2 Object model

Uit de probleembeschrijving kunnen de volgende klassen gedestilleerd worden: Server applicatie, MainPortServer, SubPortServer, Status, Compression, Encryption, CommandTranslater en ServerSettings. Omdat de mainportserver en subportserver overeenkomstige eigenschappen hebben, zal een superklasse (PortServer) van MainPortServer en SubPortServer geïntroduceerd worden.



Figuur 17 De klassen van de Smalltalkobjectserver

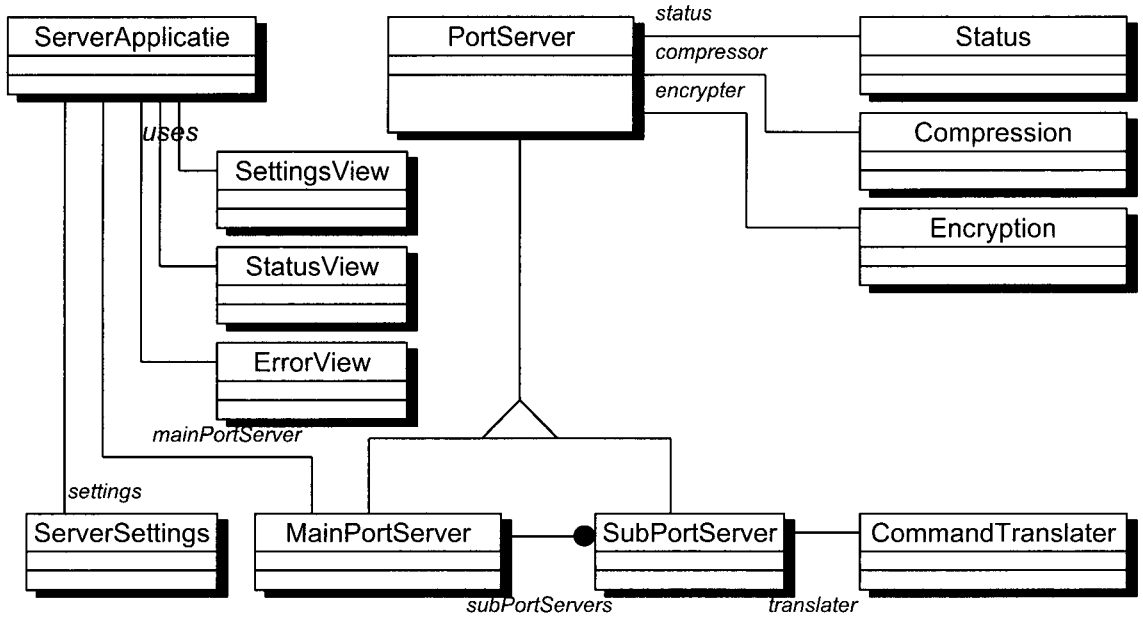
In Tabel 4 is de data dictionary opgenomen van de klassen uit Figuur 17. Hierin worden alle klassen en hun functies besproken.

Tabel 4 Data dictionary van de Smalltalkobjectserver

Object klasse	Beschrijving
ServerApplicatie	De Server applicatie is de klasse waarin alle interactie met de gebruiker plaatsvindt. Dit omvat dus het instellen van de Smalltalkobjectserver, het starten en stoppen van de mainportserver en het bekijken van de status van een server.
ServerSettings	De klasse ServerSettings bevat alle instellingen van de Smalltalkobjectserver.
PortServer	De PortServer klasse voorziet in alle methoden die nodig zijn om een communicatiesessie te starten met een client. De poort server is verder verantwoordelijk voor het bijhouden van de status van een poort.
MainPortServer	De mainportserver heeft als enige taak om aanvragen van een client te verwerken. Hiervoor heeft de mainportserver een collectie van subportservers ter beschikking, welke hij kan initialiseren en starten.
SubPortServer	De voornaamste taak van de subportserver is het ontvangen van de commando's, deze te vertalen naar een Smalltalk bericht, dit bericht uit te voeren en het resultaat te vertalen en terug te sturen.
Status	In de Status klasse zijn alle gegevens opgenomen die informatie geven over de toestand van een bepaald kanaal.
Compression	De Compression klasse bevat methoden om een string te comprimeren en te decomprimeren.
Encryption	De Encryption klasse bevat methoden om een string te coderen en te decoderen.

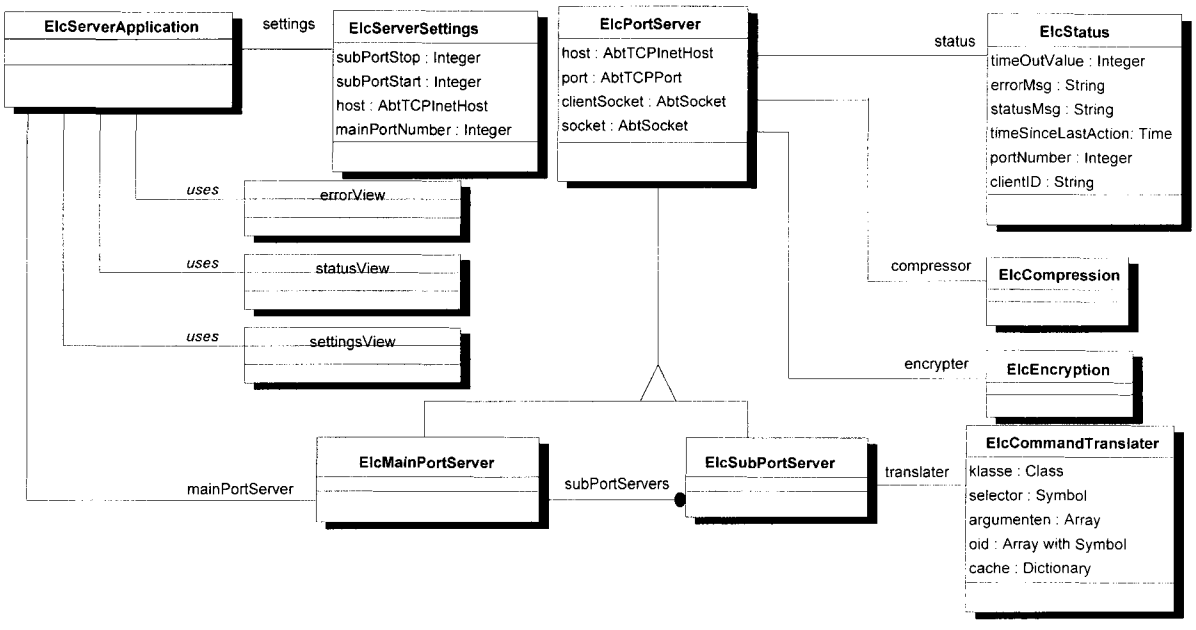
CommandTranslater	De CommandTranslater voorziet in alle vertaalslagen die de subportserver moet maken.
StatusView	De StatusView klasse is een scherm waarin de status van een bepaalde poort weergegeven kan worden.
SettingsView	De SettingsView klasse is een scherm waarin de settings van de server aangepast kunnen worden.
ErrorView	De ErrorView klasse is een scherm waarin alle fouten die opgetreden zijn bij de verschillende poorten weergegeven worden.

Met behulp van de Data dictionary kunnen de verschillende associaties bepaald worden (Zie Figuur 18). De server applicatie heeft een associatie met ServerSettings klasse en met de MainPortServer klasse.



Figuur 18 Associaties in de Smalltalkobjectserver

De mainportserver beschikt over een aantal subportservers. Elke PortServer bezit een bepaalde status en maakt gebruik van de Compression en Encryption klasse om berichten te voorzien van een codering of compressie. Als laatste is elke subportserver uitgerust met een CommandTranslater, zodat alle vertaalslagen en executies van de commando's plaats kan vinden. Nu de associaties tussen de verschillende klassen bekend zijn kunnen de attributen van de klassen aangebracht worden.



Figuur 19 Attributen van de Smalltalkobjectserver klassen

In Figuur 19 zijn voor alle klassen de attributen aangegeven. Alle klassen zijn hier tevens voorzien van een 'Elc' prefix, die bij ELC Object Technology gebruikt wordt.

2.1.3 Dynamisch model

Het dynamische model zoals het hier beschreven wordt, bestaat uit enkele scenario's die doorlopen worden in de mainportserver. Hierna worden enkele niet triviale eventtraces besproken. Als laatste worden de belangrijkste toestandsdiagrammen besproken.

Scenario 1 Openen van een connectie met de server.

- De gebruiker van de Smalltalkobjectserver start de Smalltalkobjectserver.
- De server applicatie start de mainportserver en gaat luisteren naar de mainport.
- De mainport server initialiseert de subportservers.
- De client applicatie opent een connectie met de mainportserver.
- De client applicatie verstuurt "requestCommunication" naar de mainport.
- Als er een subport vrij is wordt een vrij poortnummer terug gegeven d.m.v.: "grantedAtPort: XXXX".
- De client applicatie opent een connectie met de aangegeven poort.
- De client/server communicatie vindt plaats(zie Scenario 3).
- De client applicatie verstuurt "endConnection" naar de server.
- De subportserver wordt vrijgegeven.

Scenario 2 Stoppen van de server

- De gebruiker van de Smalltalkobjectserver drukt op de stop server knop.
- De server applicatie stopt de mainportserver.
- De mainportserver stopt de subportservers.

Scenario 3 Verwerken van commando's

- De subportserver ontvangt een string van de client.
- De subportserver decomprimeert de string.
- de string wordt gedecodeerd.
- De gedecodeerde string wordt vertaald naar een commando.
- Het commando wordt uitgevoerd.
- Het resultaat van het commando wordt vertaald naar een string.
- De string wordt gecodeerd.
- De gecodeerde string wordt gecomprimeerd.
- De gecomprimeerde string wordt naar de client applicatie gestuurd.

Scenario 4 Veranderen van de settings

- De gebruiker van de Smalltalkobjectserver drukt op de settings knop.
- De server applicatie opent een scherm met de huidige settings.
- De gebruiker past de settings aan, bijvoorbeeld wordt de host naam veranderd.
- De gebruiker van de Smalltalkobjectserver drukt op de OK knop.
- De gespecificeerde host wordt gecontroleerd.
- De nieuwe settings worden overgenomen.
- Het scherm wordt gesloten.

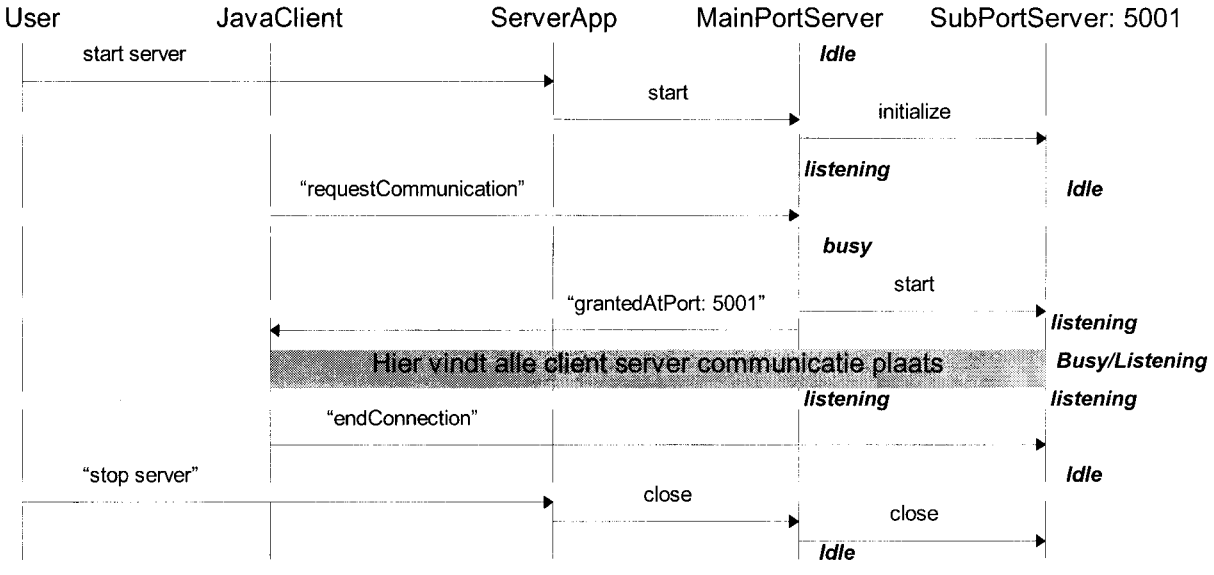
Scenario 5 Bekijken van de status van een poort, veranderen van de time out value

- De gebruiker van de Smalltalkobjectserver selecteert een poort
- De gebruiker van de Smalltalkobjectserver drukt op de status knop
- Er wordt een scherm geopend met de status van de geselecteerde poort
- Dezelfde gebruiker verandert de time out value van de poort
- Dan drukt de gebruiker op OK
- De nieuwe time out value worden overgenomen.
- Het scherm wordt gesloten.

Scenario 6 Bekijken van de opgetreden fouten

- De gebruiker van de Smalltalkobjectserver drukt op de 'view errors' knop.
- De server applicatie opent een scherm met daarin een lijst van alle poorten en hun eventuele fouten.
- Dan drukt de gebruiker op de OK knop.
- Het scherm wordt gesloten.

De twee belangrijkste eventtraces zijn die van het opzetten van de connectie tussen een client en server en die van het verwerken van de commando's. In Figuur 20 wordt de eventtrace weergegeven van het opzetten van een connectie.



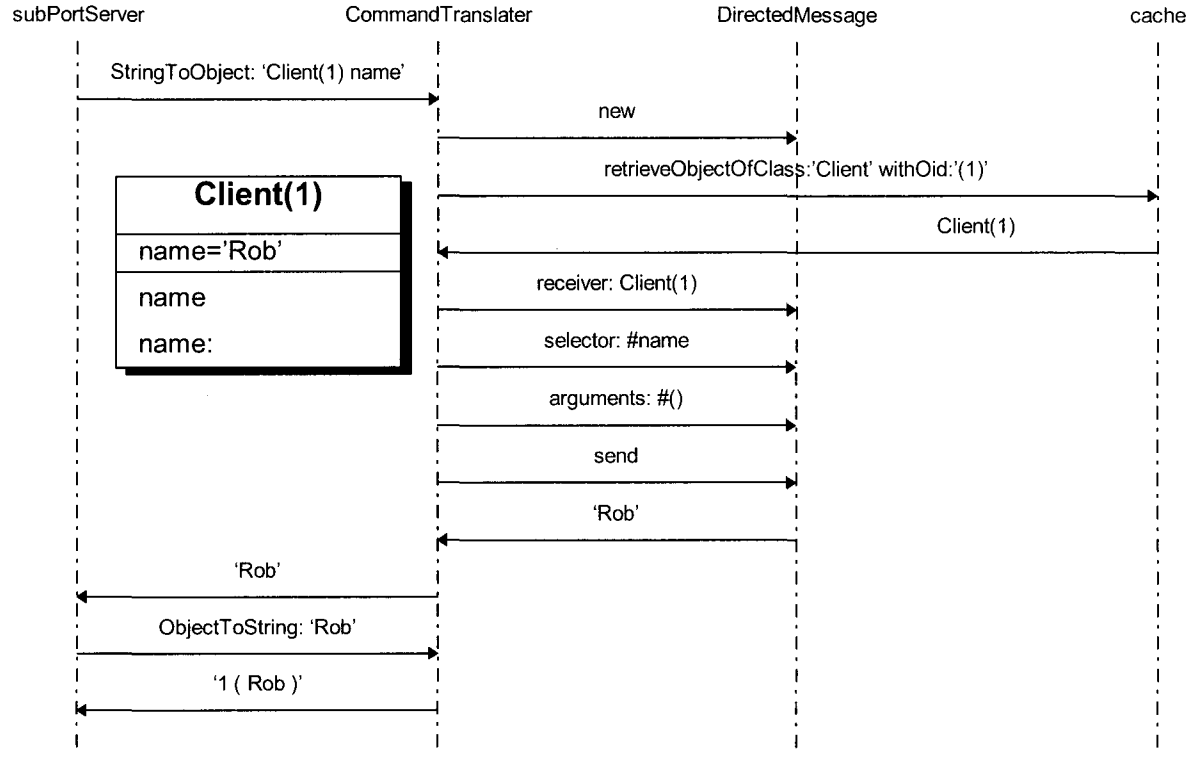
Figuur 20 Eventtrace voor het opzetten van een client/server connectie

Eventtrace voor het opzetten van een client/server connectie

Als eerste start een gebruiker aan de server kant de Smalltalkobjectserver. Door deze start aktie wordt de mainportserver gestart en de subportservers geïnitialiseerd. Vanaf dit moment luistert de mainportserver naar een bepaald kanaal. Als een client nu de string 'requestCommunication' naar deze poort stuurt, dan wordt er eerst een subportserver gestart en dit bevestigd naar de client met het poortnummer waarmee hij mag gaan communiceren. Dit gebeurt door middel van de string 'grantedAtPort: 5001'. De Javaproxycommunicator kan nu verbinding maken met deze toegewezen poort. Als deze connectie tot stand is gekomen, kunnen de proxyobjecten communiceren met de lokale Smalltalk objecten. Op het moment dat de client de string 'endConnection' stuurt, zal de subportserver gesloten worden. De termen 'Idle', 'Listening' en 'Busy' refereren naar de verschillende toestanden waarin de mainportserver en subportserver zich bevinden. Deze toestanden worden in Figuur 22 en Figuur 23 verder uitgediept.

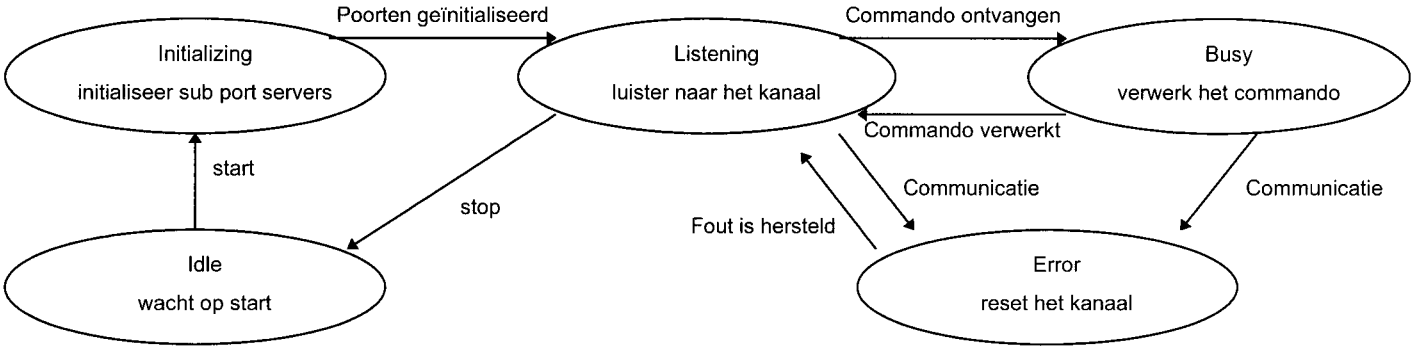
Eventtrace voor het verwerken van een commando

Stel dat de client een ClientProxy met oid=1 heeft, waarvan de methode *name* wordt aangeroepen. Hiervoor stuurt de client de string "Client(1) name" naar de server. De server moet deze string nu verwerken. Dit gebeurt door het bericht stringToObject te sturen naar de CommandTranslater. Deze maakt een nieuwe instantie van DirectedMessage. De klasse DirectedMessage wordt in Smalltalk gebruikt om berichten naar bepaalde objecten of klassen te sturen. Vervolgens wordt de eerste substring van de string verwerkt en wordt bepaald dat de receiver object Client(1) is, dus een instantie van de klasse Client met object identifier gelijk aan 1. Omdat de client refereert aan een object identifier moet deze al een keer naar de client gestuurd zijn. Dit betekent dat dit object in de cache opgeslagen is en hier dus ook weer opgehaald kan worden. De selector van dit bericht is *name* en er zijn geen argumenten. Door nu **send** te sturen naar deze instantie van DirectedMessage wordt het bericht uitgevoerd. Het resultaat van dit bericht wordt teruggegeven aan de subportserver. Vervolgens wordt dit object, namelijk de string met waarde 'Rob', vertaald naar een string die naar de client teruggestuurd kan worden. Dit gebeurt door objectToString te sturen naar CommandTranslater. Deze vertaalt de string 'Rob' naar de communicatiestring '1 (Rob)'. Deze communicatiestring wordt vervolgens



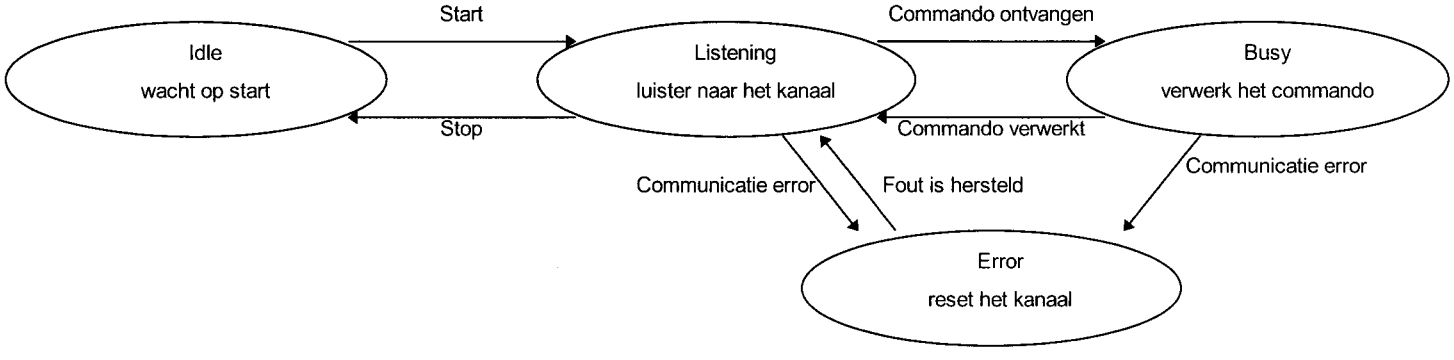
Figuur 21 Eventtrace voor het verwerken van commando's

naar de client teruggestuurd. In de mainportserver en de subportservers komen verschillende toestanden voor. Figuur 22 geeft de toestanden weer van de mainportserver en Figuur 23 geeft de toestanden weer van een subportserver.



Figuur 22 Toestanden van de mainportserver

De mainportserver begint in de 'Idle' toestand en wacht tot de gebruiker op start drukt. Als de gebruiker op start drukt, wordt de mainportserver gestart en worden de subportservers geïnitialiseerd. Als de mainportserver gestart is, begint deze met luisteren. Wordt er een commando ontvangen dan komt de mainportserver in de 'Busy' toestand. Het commando wordt verwerkt en wanneer het commando verwerkt is, komt de server terug in de 'Listening' toestand. De mainportserver zal steeds tussen de 'Listening' en 'Busy' toestand heen en weer gaan totdat de gebruiker de server stopt. Als er iets fout gaat in de communicatie, komt de server in de 'Error' toestand. Als het mogelijk is, wordt deze fout hersteld en komt de server terug in de 'Listening' toestand.

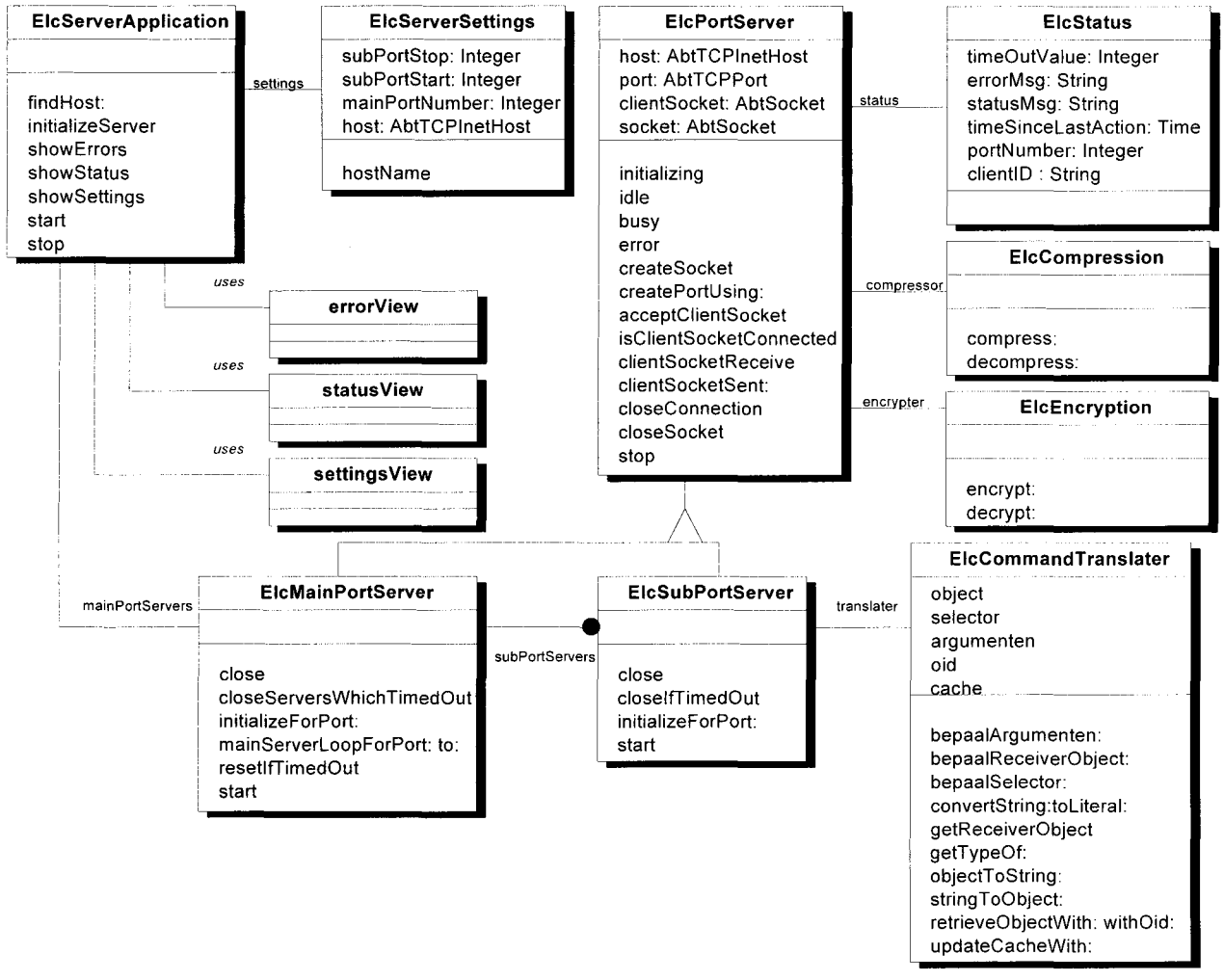


Figuur 23 De toestanden van de subportservers

De toestanden van een subportserver staan afgebeeld in Figuur 23. De normale toestand van de subportserver is de 'Idle' toestand. Als deze poort wordt toegewezen aan een client, wordt de subportserver gestart door de mainportserver. Vanaf dat moment staat de subportserver in de 'Listening' toestand. Wordt er nu een commando ontvangen dan komt de server in de 'Busy' toestand en wordt het commando verwerkt. Als het commando verwerkt is, komt de server terug in de 'Listening' toestand. Ook hier geldt dat als er een fout optreedt in de communicatie, de server in de 'Error' toestand komt. Als deze fout hersteld kan worden, komt de server terug in de 'Listening' toestand.

2.2 Object ontwerp

De laatste stap is het toevoegen van methoden aan elke klasse. De meeste methoden komen direct voort uit de scenario's, eventtraces en toestandsdiagrammen. De functies van de verschillende methoden zullen per klasse besproken worden.



Figuur 24 Object model van de Smalltalkobjectserver

In de onderstaande tabellen staat per klasse beschreven wat de functie is van de verschillende methoden.

Tabel 5 Methoden van de klasse ServerApplication

ElcServerApplication	
initializeServer	initialiseer de server door het instellen van de standaard instellingen.
showSettings	open een scherm met de huidige settings, deze kunnen in dit scherm veranderd worden.
showStatus	open een scherm met de status van een poort, hier kan de time out value van een bepaalde poort veranderd worden.
showErrors	open een scherm waarin voor alle poorten de fouten staan weergegeven.

start	start de server, door de subportservers aan te maken en de mainportserver te starten.
stop	stop de mainportserver en subportservers.
findHost:	Probeer een host aan te maken met de naam die als argument wordt meegegeven.

De methode findHost: wordt bij het starten van de server applicatie uitgevoerd met als argument 'local'. Hier wordt dan getracht om de server te starten voor een lokale host. Bestaat de in het argument gespecificeerde host niet, dan wordt om een nieuwe host naam gevraagd. Als er geen netwerk aansluiting aanwezig is, kan de applicatie niet gestart worden, doordat geen enkele host aangemaakt kan worden.

Tabel 6 Methoden van de ServerSettings

ElcServerSettings	
hostName	geef de naam terug van de huidige host.

De enige methoden, buiten de getters en setters, van de klasse ServerSettings is hostName. Om consequent te blijven, staat deze toch vermeld in een tabel.

Tabel 7 Methoden van de klasse PortServer

ElcPortServer	
initializing	De status van de server wordt: 'Initializing'
error	De status van de server wordt: 'Error'
busy	De status van de server wordt: 'Busy'
idle	De status van de server wordt: 'Idle'
listening	De status van de server wordt: 'Listening'
createPortUsing:	Creëer een poort voor de huidige host. Het poortnummer wordt als argument meegegeven.
createSocket	Creëer een socket voor de huidige poort.
acceptClientSocket	Wacht totdat een client zich verbindt met de poort.
isClientSocketConnected	Geef aan of een client verbonden is met de poort.
clientSocketReceive	Ontvang een string van de verbonden client.
clientSocketSent:	Zend een string naar de verbonden client.
closeConnection	Sluit de connectie met de client.
closeSocket	Sluit de socket van de poort.
stop	Roep achtereenvolgens closeConnection en closeSocket aan.

Tabel 8 Methoden van de klasse MainPortServer

ElcMainPortServer	
close	Sluit de connectie met de verbonden client
closeServersWhichTimedOut	Sluit de subportservers die de time out value bereikt hebben.
initializeForPort:	Initialiseer de mainportserver voor de poort die als argument wordt meegegeven.
mainServerLoopForPort: to:	Verwerk alle aanvragen voor de poorten met de poortnummers argument 1 tot argument2. Blijf dit altijd doen.
resetIfTimedOut	Reset de mainportserver als deze zijn time out waarde bereikt heeft.
start	Start de mainportserver en initialiseer de subportservers.

De methode *closeServersWhichTimedOut* wordt in een apart proces om de tien seconden aangeroepen. Dit geldt ook voor de *resetIfTimedOut* methode.

Tabel 9 Methoden van de klasse SubPortServer

ElcSubPortServer	
close	Maakt eerst de cache schoon en sluit daarna de connectie en de socket.
closeIfTimedOut	Sluit deze server als zijn time out value bereikt is.
initializeForPort:	Initialiseer de server voor een specifieke poort.
start	Start de subportserver lus waarin de commando's ontvangen, verwerkt en geretourneerd worden.

De methode *closeIfTimedOut* wordt aangeroepen door de methode *closeServersWhichTimedOut* van de mainPortServer klasse.

Tabel 10 Methoden van de klasse Compression

ElcCompression	
compress:	Retourneer de gecomprimeerde versie van de string in het argument
decompress:	Retourneer de gedecomprimeerde versie van de string in het argument

De methoden compress en decompress retourneren op dit moment de string zelf zonder deze te comprimeren of decomprimeren. Wanneer het comprimeren en decomprimeren van communicatiestrings gewenst is, is het alleen nodig deze twee methoden aan te passen.

Tabel 11 Methoden van de klasse Encryption

ElcEncryption	
encrypt	Retourneer de gecodeerde versie van de string in het argument
decrypt	Retourneer de gedecodeerde versie van de string in het argument

Ook voor de encrypt en decrypt methode geldt dat deze op dit moment de string zelf terug geven, zonder deze dus te coderen of te decoderen. Ook hier geldt dat als dit toch gewenst is deze twee methoden aangepast dienen te worden.

Tabel 12 Methoden van de klasse CommandTranslater

ElcCommandTranslater	
bepaalArgumenten:	Bepaal de argumenten in de ontvangen string.
bepaalReceiverObject:	Bepaal het object waarnaar het bericht gestuurd gaat worden.
bepaalSelector:	Bepaal de selector in de ontvangen string.
getReceiverObject	Haal het receiver object op uit de cache.
objectToString:	Deze methode vertaalt een object naar een communicatiestring.
stringToObject:	Vertaal de ontvangen string naar een commando, voer deze uit en geef het resultaat object terug.
convertString: toLiteral:	Vertaal een stringwaarde naar een literal die in het tweede argument gespecificeerd wordt.
updateCacheWith:	Het object in het argument wordt toegevoegd of verfrist in de cache.
retrieveObjectWith: withOid:	Haal het object met de gespecificeerde object identifier op uit de cache.
getTypeOf:	Bepaal het type dat hoort bij een bepaalde waarde in de string.

Elke subportserver beschikt over een command translater en elke command translater heeft een cache. Deze cache is aan het begin van elke communicatiesessie leeg. Aan het einde van elke communicatiesessie wordt deze cache leeg gemaakt. In de cache worden alle complexe objecten die het resultaat kunnen zijn van instantie– of klassemethoden, opgeslagen. Hierdoor is de Smalltalkobjectserver in staat om alle referenties naar deze complexe objecten, op de juiste manier te verwerken. Om dit alles te verduidelijken het volgende voorbeeld:
Beschouw onderstaande stukje Java code.

ClientProxy klant = ClientProxy.newObject();

Dit heeft als resultaat dat de string "Client new" naar de Smalltalkobjectserver gestuurd wordt. Deze voert de methode **new** uit en voegt het resultaat toe aan de cache. Hier wordt dan ook de object identifier toegevoegd aan het object, laten we zeggen oid=101. Het resultaat wordt vertaald naar de string "255 Client(101)", waarna er in Java een nieuwe instantie van de klasse ClientProxy wordt aangemaakt. De object identifier van deze nieuwe instantie wordt gelijk gemaakt aan 101. Vervolgens wordt het volgende stukje Java code uitgevoerd:

klant.name("Rob");

dit heeft als resultaat dat de string "Client(101) name: 1 (Rob)" wordt gestuurd naar de Smalltalkobjectserver. De Smalltalkobjectserver haalt het object van de klasse Client met de gespecificeerde

object identifier op uit de cache en stuurt het bericht **name: 'Rob'** naar dit object. Dus met behulp van de cache en de object identifier zijn alle complexe objecten in Smalltalk te lokaliseren.

Bijlage 3 Ontwerp van de Javaproxycommunicator

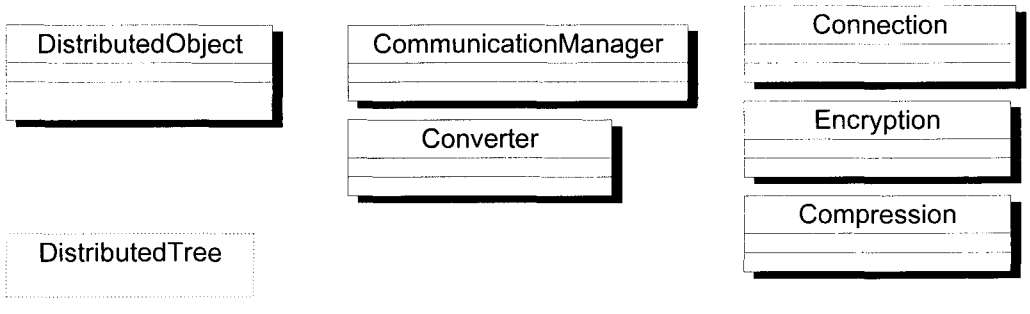
3.1 Analyse

3.1.1 Probleembeschrijving

De Javaproxycommunicator heeft als taak om de methode aanroepen van gedistribueerde proxyobjecten te verwerken. Hiervoor dient een communicatiemanager geïmplementeerd te worden die verbinding kan maken met de server. De argumenten van een methode aanroep moeten vertaald kunnen worden, hiervoor is een vertaler nodig. Deze vertaler moet ook het resultaat van een methode aanroep kunnen vertalen naar een object. De strings die opgestuurd worden, moeten gecomprimeerd en gecodeerd kunnen worden. De strings die ontvangen worden, moeten gedecomprimeerd en gedecodeerd kunnen worden.

3.1.2 Object model

Uit de probleembeschrijving volgen de klassen van de Javaproxycommunicator (zie Figuur 25). Omdat de lokale objecten subklassen zijn van Object, wordt hier een DistributedObject geïntroduceerd dat als superklasse dient voor alle gedistribueerde objecten, oftewel de superklasse voor de gedistribueerde boom.



Figuur 25 klassen van de Javaproxycommunicator

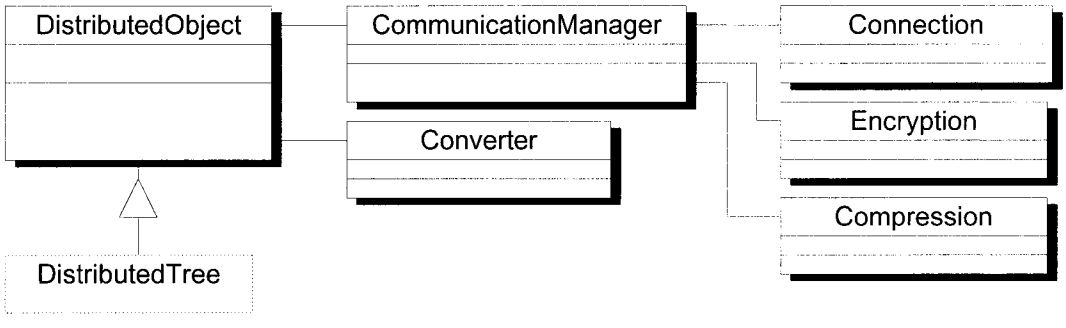
De data dictionary in Tabel 13 beschrijft de verschillende klassen en hun functies.

Tabel 13 Data dictionary van de Javaproxycommunicator

Object klasse	Beschrijving
DistributedObject	Deze klasse dient als superklasse voor alle gedistribueerde objecten. DistributedObject maakt gebruik van de CommunicationManager om te communiceren met de server. De Converter wordt gebruikt om de argumenten van een methode aanroep en het resultaat van een methode aanroep te vertalen.

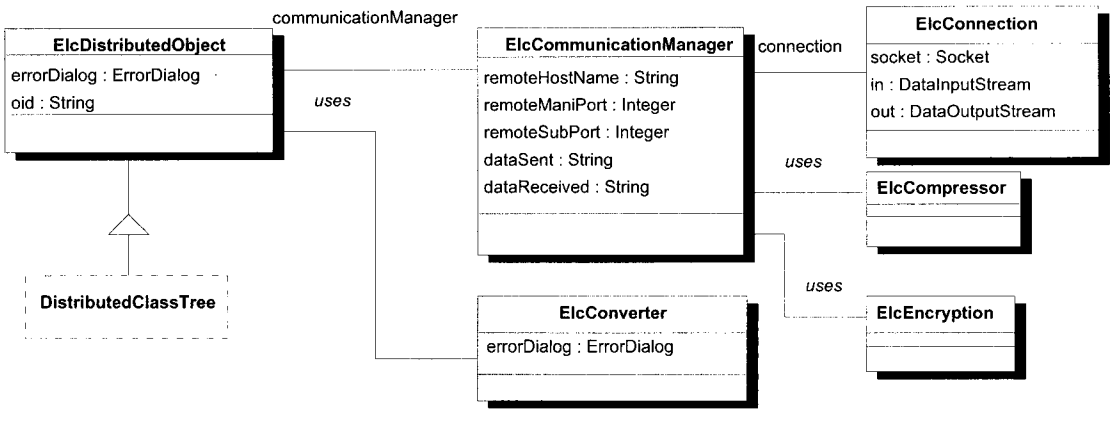
CommunicationManager	De Communication manager beheert de communicatiesessie met de server en maakt hierbij gebruik van de Connection klasse die in de connectie met de server voorziet. De Compression en Encryption klasse worden gebruikt om de strings te comprimeren/decomprimeren en te coderen/decoderen.
Connection	De Connectie klasse voorziet in de connectie met de server.
Converter	De Converter vertaalt alle argumenten naar strings en strings naar objecten.
Compression	De Compression klasse wordt gebruikt om strings te comprimeren en decomprimeren.
Encryption	De Encryption klasse wordt gebruikt om strings te coderen en decoderen.

Uit de data dictionary van de Javaproxycommunicator zijn de verschillende associaties te herleiden. Deze associaties zijn aangebracht in Figuur 26. DistributedObject dient als superklasse voor de gedistribueerde klasseboom. DistributedObject maakt gebruik van de Converter om de benodigde vertaalslagen te maken en gebruikt de communicatiemanager om de commando's te versturen naar de server. Hiervoor gebruikt de communicatiemanager een connectie. Voor het comprimeren /decomprimeren gebruikt de communicatiemanager de Compression klasse. Voor het coderen/decoderen van een string gebruikt de communicatiemanager de Encryption klasse.



Figuur 26 Associaties in de Javaproxycommunicator

Nu de associaties van de Javaproxycommunicator bekend zijn, kunnen de attributen van de verschil-



Figuur 27 Attributen van de Javaproxycommunicator klassen

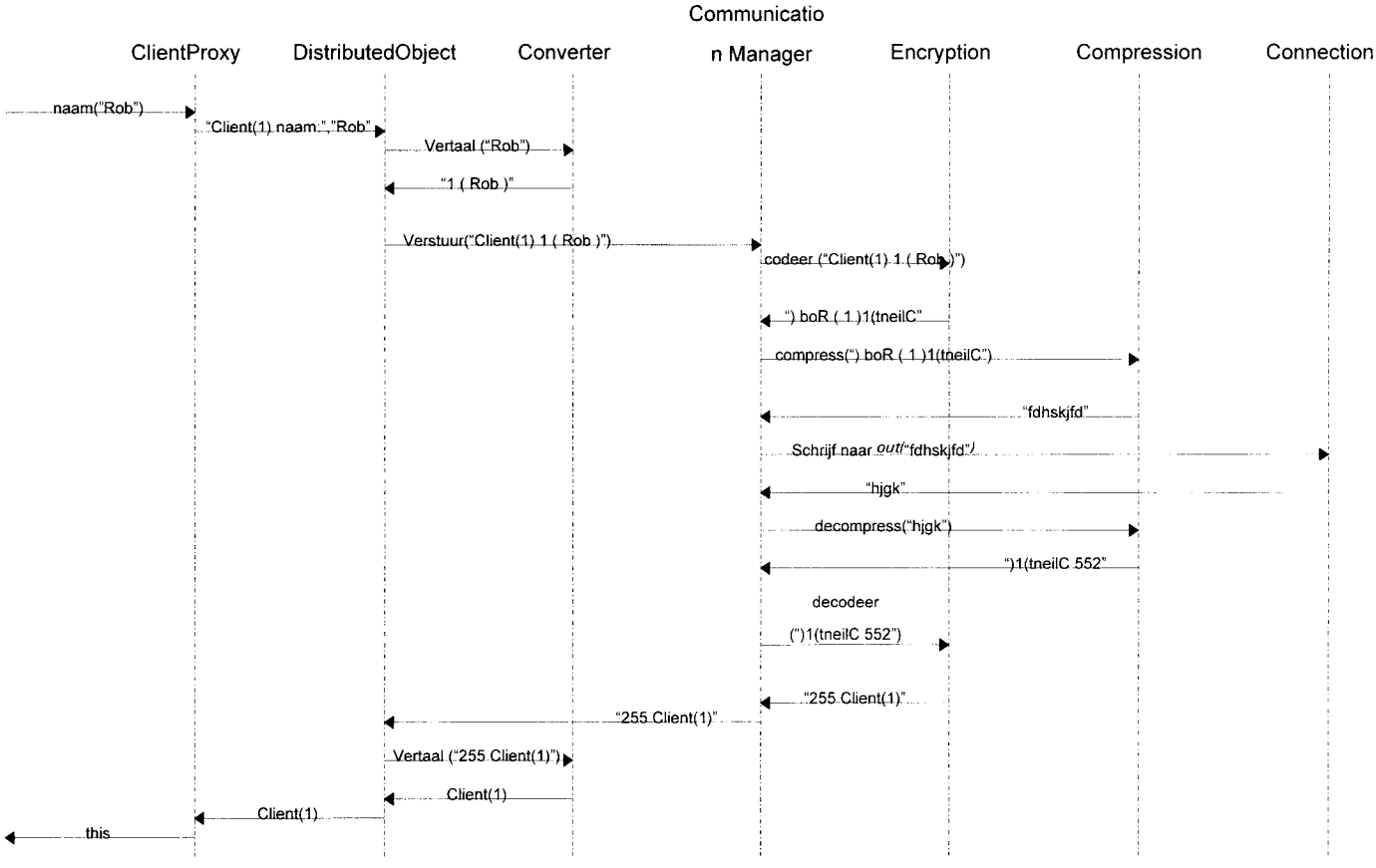
lende klassen aangebracht worden. Het object model met alle attributen staat weergegeven in Figuur 27. Hier is ook weer de prefix 'Elc' toegevoegd aan alle object klassen.
De attributen van DistributedObject zijn *errorDialog* en *oid*. Het attribuut *errorDialog* is een scherm dat geopend wordt als er een fout opgetreden is. Het attribuut *oid* is de unieke object identifier die elk gedistribueerd object identificeert. Het attribuut *remoteHostName* van de communicationManager identificeert de naam van de host waarmee de connectie gemaakt moet worden. Het attribuut *remoteMainPort* is het nummer van het algemene aanvraag kanaal van de server. Het attribuut *remoteSubPort* bevat het kanaalnummer van de subport die toegewezen wordt. De attributen *dataSent* en *dataReceived* bevatten respectievelijk de te verzenden string en de te ontvangen string. Ook de Converter klasse maakt gebruik van een *errorDialog* als er een fout optreedt. De Connectie klasse bevat drie attributen te weten: *in*, *out* en *socket*. De attributen *in* en *out* bevatten de binnenkomende en uitgaande datastroom. De *socket* bevat de *socket* connectie met de server. De datastromen *in* en *out* worden gekoppeld aan deze *socket*.

2.1.3 Dynamisch model

De enige taak van de Javaproxycommunicator is het doorsturen van de berichten die gestuurd worden naar de gedistribueerde objecten. Om deze reden is er ook maar één niet triviaal scenario dat besproken wordt. Van dit scenario staat in Figuur 28 de eventtrace weergegeven.

Scenario 1 Uitvoeren van een methode

- In Java wordt een bericht gestuurd naar een object uit de gedistribueerde klasseboom.
- De implementatie van dit bericht roept een **perform** methode van DistributedObject aan.
- Deze **perform** methode vertaalt de aanroep naar een string.
- Ook vertaalt deze **perform** methode, met behulp van de Converter, de argumenten naar strings.
- Dan wordt de string naar de communicatiemanager gestuurd.
- Deze codeert en comprimeert de string.
- De gecodeerde en gecomprimeerde string wordt via de connectie naar de server gestuurd.
- De communicatiemanager wacht op het resultaat.
- Het resultaat komt binnen en wordt gedecomprimeerd en gedecodeerd.
- de gedecodeerde en gedecomprimeerde string wordt terug gegeven aan DistributedObject
- Deze vertaalt de string naar een Object dat als resultaat waarde van de methode wordt teruggegeven.



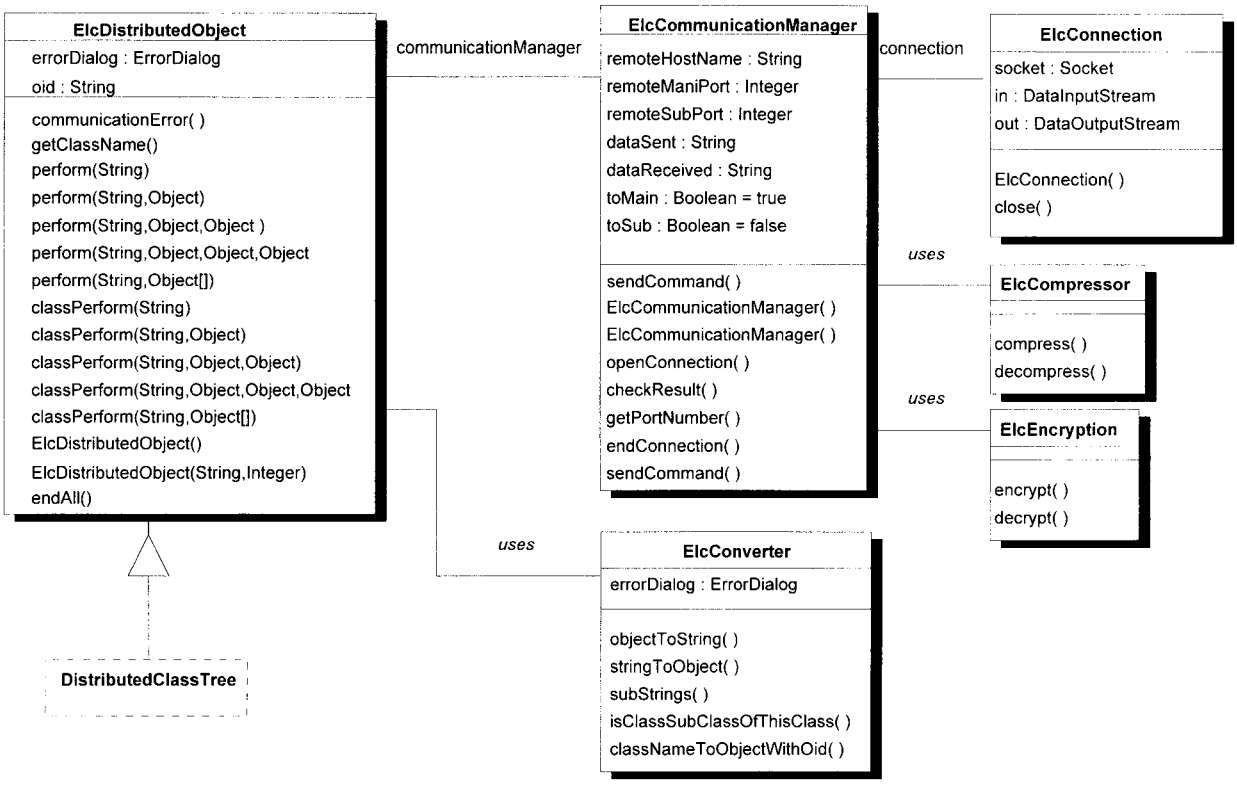
Figuur 28 Eventtrace voor een methode aanroep

In Java bestaat een proxyobject van de Smalltalk klasse Client. Deze klasse heet ClientProxy. De klasse Client kent een methode *naam*: die de naam van een Client zet. Naar een instantie van Client-Proxy (met oid=1) in Java wordt naam("Rob") gestuurd. De implementatie van deze methode voert een *perform*("Client(1) naam:", "Rob") uit. Deze methode is geïmplementeerd in de superklasse DistributedObject. De methode *perform* laat de Converter klasse eerst het argument "Rob" vertalen naar een communicatiestring. Dan wordt het commando, tezamen met het argument, als communicatiestring doorgestuurd naar de communicatiemanager. Deze codeert en comprimeert de string en verstuurt deze naar de server. Het resultaat wordt gedecomprimeerd en gedecodeerd. DistributedObject vertaalt deze string vervolgens weer naar een object dat als resultaat van de in eerste instantie gestuurde methode *naam*("Rob") wordt teruggegeven.

Op deze manier worden alle methoden verwerkt en kent DistributedObject vijf verschillende *perform* methoden nl. voor 0, 1, 2, 3 en meer dan drie argumenten. Ook kent Distributed object op dezelfde manier vijf verschillende *classPerform* methoden die, zoals de naam al suggereert, klasse methoden uitvoeren.

3.2 Object ontwerp

In Figuur 29 zijn de methoden van de klassen toegevoegd.



Figuur 29 Object model van de Javaproxycommunicator

In de onderstaande tabellen zijn van alle klassen de methoden opgenomen en wordt hun functie uitgelegd.

Tabel 14 Methoden van de klasse DistributedObject

ElcDistributedObject	
communicationError	Als er een fout opgetreden is, wordt deze methode aangeroepen. Er wordt dan een scherm geopend met de boodschap dat er een fout is opgetreden in de communicatie.
getClassname	De Java proxy klassen hebben een prefix 'elc.proxyObjects' en een postfix 'Proxy' die verwijderd moet worden.
endAll	Beëindig de communicatiesessie met de server.
perform:	Er zijn vijf verschillende perform methoden. Er zijn methoden voor 0, 1, 2, 3 en meer dan drie argumenten. Deze methoden verwerken de instantie methode van de gedistribueerde klasseboom.

classPerform:	Er zijn vijf verschillende classPerform methoden. Er zijn methoden voor 0, 1, 2, 3 en meer dan drie argumenten. Deze methoden verwerken de klasse methode van de gedistribueerde klasseboom.

Alle gedistribueerde klassen worden als subklassen opgenomen van ElcDistributedObject. In de gedistribueerde klasseboom blijft wel de klasse hiërarchie gehandhaafd.

Tabel 15 Methoden van de klasse Converter

ElcConverter	
objectToString:	Het object in het argument wordt vertaald naar een communicatie-string.
stringToObject:	De string in het argument wordt vertaald naar het object.
subStrings:	Retourneer een collectie van substrings van de string in het argument.
isClassSubClassOfThisClass:	Retourneer true als de klasse in het argument een subklasse is van ElcDistributedObject.
classNameToObjectWithOid:	Genereer een instantie van een bepaalde gedistribueerde klasse met een bepaalde object identifier.

De eerste twee methoden zijn de belangrijkste van de Converter klasse. De andere drie methoden worden ter ondersteuning gebruikt.

Tabel 16 Methoden van de CommunicationManager

ElcCommunicationManager	
openConnection	Open een connectie met de mainportserver, vraag een communicatie-sessie aan en maak een connectie met de toegewezen subport.
checkResult	Controleer of het resultaat van de server op de aanvraag klopt.
endConnection	Sluit de connectie met de subportserver.
getPortNumber	Destilleer het subport nummer uit de string.
sendCommand	Zend het commando in het attribuut dataSent en retourneer de ontvangen string.
sendCommand:	Zend het gespecificeerde commando en retourneer de ontvangen string.
ElcCommunicationManager	De constructor van deze klasse initialiseert de settings van de communicatiemanager.

Tabel 17 Methoden van de klasse Connection

ElcConnection	
ElcConnection	Open een connectie met de server en koppel de *in* en *out* variabelen aan de socket.
close	Sluit de connectie.

Tabel 18 Methoden van de klasse Compression

ElcCompressor	
compress:	Comprimeer de string in het argument.
decompress:	Decomprimeer de string in het argument.

De methoden compress en decompress zijn, net als in de Smalltalkobjectserver, niet geïmplementeerd. Voor deze methoden geldt ook weer dat als compressie en decompressie toch gewenst zijn, alleen deze methoden geïmplementeerd dienen te worden.

Tabel 19 Methoden van de klasse Encryption

ElcEncryption	
encrypt	Codeer de string in het argument.
decrypt	Decodeer de string in het argument.

De methoden encrypt en decrypt zijn ook hier weer niet geïmplementeerd, maar kunnen eenvoudig aangepast worden als codering en decodering toch gewenst is.

Bijlage 4 Ontwerp van de Javaproxygenerator

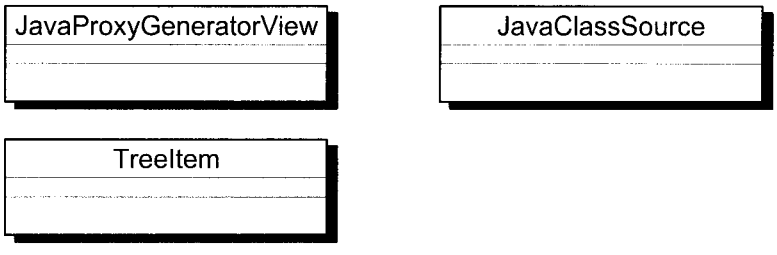
4.1 Analyse

4.1.1 Probleembeschrijving

Van bestaande Smalltalk klassen moeten Java proxy klassen gegenereerd worden. In Smalltalk moet een te distribueren klasseboom opgebouwd kunnen worden en per klasse moet aangegeven kunnen worden welke methoden in Java gebruikt moeten kunnen worden en welke niet. Deze klasseboom moet opgeslagen kunnen worden, zodat bij een volgende keer een bestaande boom geladen kan worden. Per klasse moet aangegeven kunnen worden of deze wel of niet gebruikt gaat worden in Java. Op deze manier zou het mogelijk moeten worden dat het gedrag van een superklasse niet gedistribueerd wordt, maar het gedrag van de superklasse van de superklasse wel. Met een druk op de knop moet de gedistribueerde Java klasseboom gegenereerd en opgeslagen worden.

4.1.2 Object model

De klassen die onderscheiden kunnen worden, zijn de JavaProxyGeneratorView, JavaClassSource en TreeItem (zie Figuur 31).



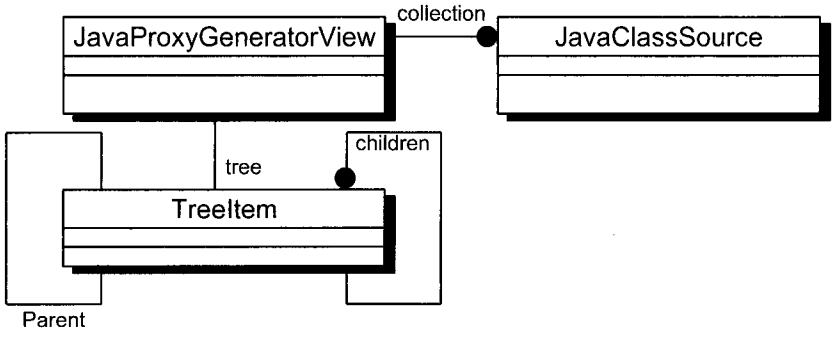
Figuur 30 klassen van de Javaproxygenerator

Tabel 20 bevat de data dictionary van de Javaproxygenerator. Uit deze data dictionary is het weer mogelijk om de associaties van de Javaproxygenerator te bepalen. Deze zijn opgenomen in Figuur 31.

Tabel 20 Data dictionary van de Javaproxygenerator

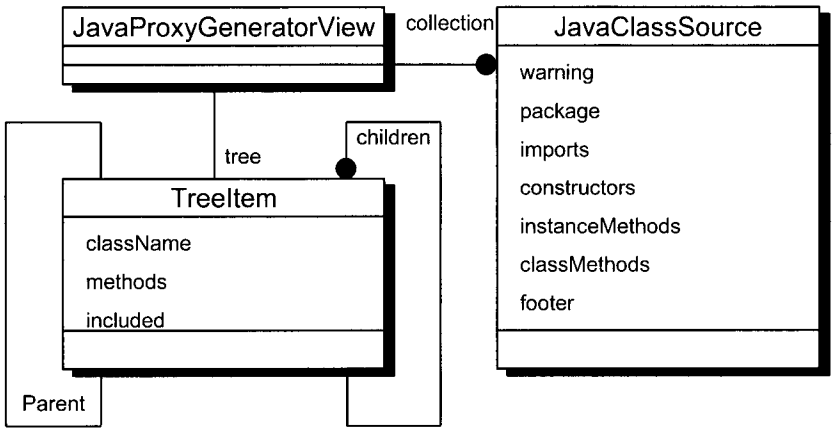
Objectklasse	Beschrijving
JavaProxyGeneratorView	Deze klasse bevat de user interface van de Javaproxygenerator. In deze user interface kan een klasseboom opgebouwd worden. De klasseboom begint initieel met de klasse Object in de boom. De user interface bevat menu opties om een boom te laden of op te slaan en een optie om de gedistribueerde Java klasseboom op te bouwen.

TreeItem	De klasse TreeItem bevat alle klassen in de klasseboom. Initieel bevat deze de klasse Object als root Item. Elke TreeItem heeft een parent (superklasse) en kan meerdere children (subklassen) hebben.
JavaClassSource	De JavaClassSource genereert per klasse uit de te distribueren boom de Java source klasse. Deze wordt automatisch opgeslagen.



Figuur 31 Associaties in de Javaproxygenerator

De JavaProxyGeneratorView bevat een 'tree'. In deze tree is initieel een rootItem 'Object' opgenomen. Deze rootItem heeft geen parent en er kunnen meerdere children (subklassen) opgenomen worden. Met een druk op de knop wordt een collectie van JavaClassSource aangemaakt. In Figuur 32 zijn de attributen van de verschillende object klassen van de Javaproxygenerator aangebracht.



Figuur 32 Attributen in de Javaproxygenerator

De klasse TreeItem bevat de attributen *className* die de naam van de klasse representeert, *methods* dat de te distribueren methoden van de klasse bevat, *included* dat aangeeft of deze klasse wel of niet gedistribueerd dient te worden. De klasse JavaClassSource bevat de attributen, *warning*, *package*, *imports*, *constructors*, *instanceMethods*, *classMethods*, *footer*. Het attribuut *package* geeft de package aan waarin deze Java klasse opgenomen wordt. Deze is standaard 'elc.proxyObjects;'. Het attribuut *imports* geeft aan welke klassen er geïmporteerd dienen te worden, standaard is dit er maar één nl. 'elc.util.ElcDistributedObject;'. Het attribuut *constructors* bevat de constructors van de gedistribueerde

klassen, op dit moment worden deze niet gebruikt. Het attribuut *instanceMethods* bevat de source code van de instantie methoden van deze klasse. Het attribuut *classMethods* bevat de source code van de klassemethoden van deze klasse. Het attribuut *footer* bevat de afsluiting van de header, dit is '}'.

4.1.3 Dynamisch model

Scenario 1 Toevoegen van een subklasse.

- De gebruiker selecteert een klasse uit de boom.
- De gebruiker drukt 'add subclass' in het popup-menu.
- De subklassen van de geselecteerde klasse worden in een selectievenster getoond.
- De gebruiker selecteert één of meerdere klasse(n).
- Het selectievenster wordt gesloten.
- De geselecteerde klassen worden toegevoegd aan de boom.

Scenario 2 Verwijderen van een subklasse.

- De gebruiker selecteert een klasse uit de boom.
- De gebruiker drukt op 'delete subclass' in het popup-menu.
- De aanwezige subklassen van de geselecteerde klasse worden in een selectievenster getoond.
- De gebruiker selecteert één of meerdere subklasse(n).
- Het selectievenster wordt gesloten.
- De geselecteerde subklassen (en hun subklassen !) worden uit de boom verwijderd.

Scenario 3 Wijzigen van de methoden.

- De gebruiker selecteert een klasse uit de boom.
- De gebruiker drukt op 'edit methods' in het popup-menu.
- Er wordt een selectievenster geopend, waarin de methode van de klasse weergegeven worden.
- De gebruiker selecteert de te distribueren methoden.
- Het selectievenster wordt gesloten.
- De geselecteerde methoden worden toegevoegd aan de geselecteerde klasse.

Scenario 4 Include een klasse.

- De gebruiker selecteert een klasse uit de boom.
- De gebruiker drukt op 'include' in het popup-menu.
- Het attribuut *included* wordt op true gezet.
- De boom wordt bijgewerkt.

Scenario 5 Exclude een klasse.

- De gebruiker selecteert een klasse uit de boom.
- De gebruiker drukt op 'exclude' in het popup-menu.

- Het attribuut *included* wordt op false gezet.
- De boom wordt bijgewerkt.

Scenario 6 Bewaren van een klasseboom.

- De gebruiker selecteert 'save class tree' in het menu.
- Er wordt een venster geopend, waarin de gebruiker een bestandsnaam kan selecteren.
- De volgende attributen worden per klasse opgeslagen: *className*, *included* en *methods*.

Scenario 7 Laden van een klasseboom.

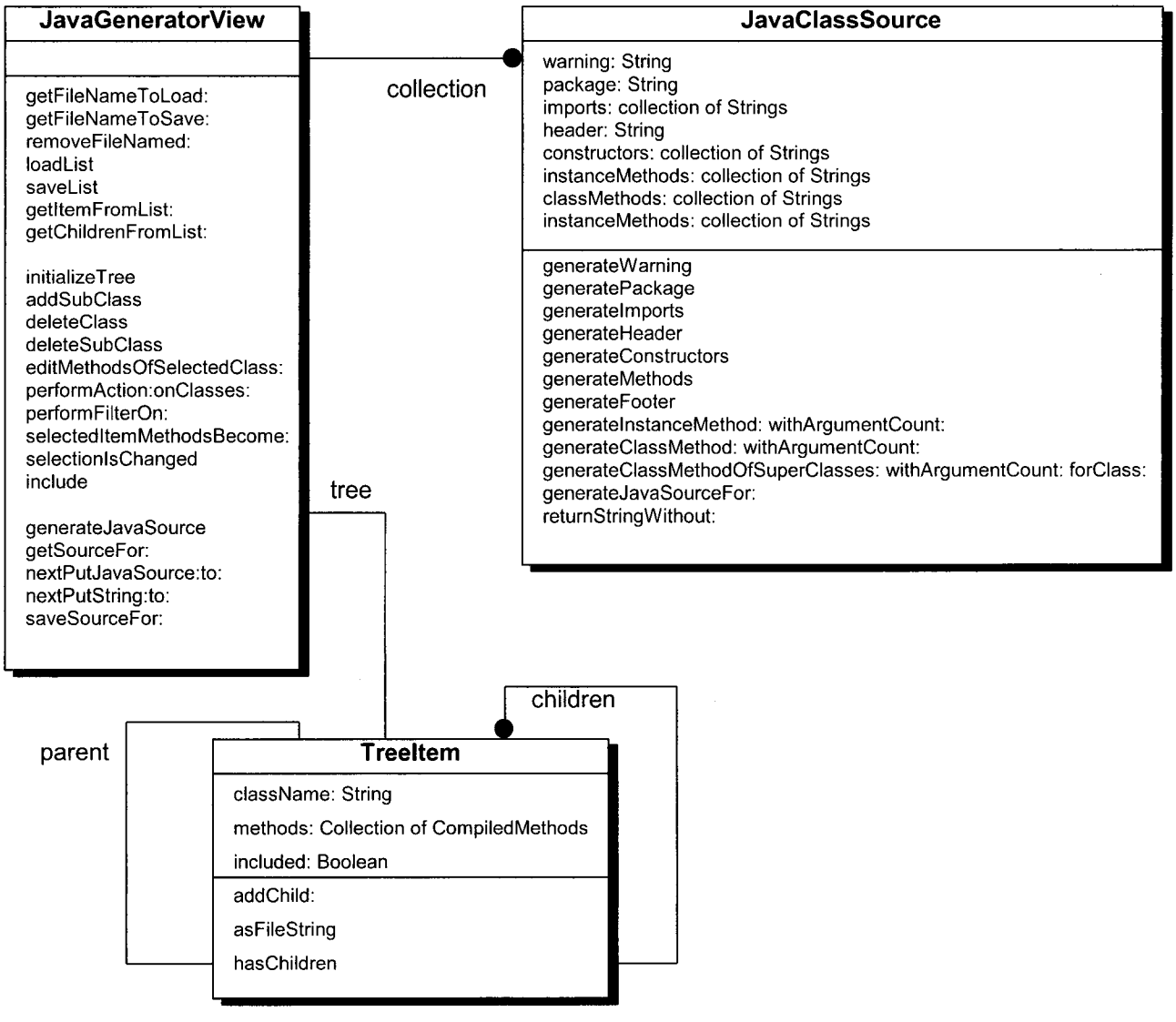
- De gebruiker selecteert 'load class tree' in het menu.
- Er wordt een venster geopend, waarin de gebruiker een bestand kan selecteren.
- Dit bestand wordt ingelezen.
- De klasseboom wordt opgebouwd voor de klassen in het bestand.
- De boom wordt naar het scherm geschreven.

Scenario 8 Genereren van de Java klasseboom.

- De gebruiker selecteert 'generate Java' in het menu.
- Van elke klasse uit de boom:
  - wordt eerst een waarschuwingsstring gegeneerd, die vermeld dat de klasse gegenereerd is.
  - wordt vervolgens de **package** statement gegenereerd.
  - dan volgt de **import** statement.
  - hierna wordt de klasse **header** gegenereerd.
  - vervolgens worden de **instantie methoden** en **klasse methoden** gegenereerd.
  - als laatste wordt de **footer** gegenereerd.
- Alle Java source klassen worden opgeslagen op de harde schijf.

4.2 Object ontwerp

Figuur 33 laat het volledige object model zien van de Javaproxygenerator. In de onderstaande tabellen wordt per klasse de functie van de verschillende methoden besproken.



Figuur 33 Object model van de Javaproxygenerator

Tabel 21 Methoden van de applicatie JavaGeneratorView

JavaGeneratorView	
getFileNameToLoad:	Open een scherm voor het openen van een bestand voor een bepaald masker.
GetFileNameToSave:	Open een scherm voor het opslaan van een bestand met een be-

	paald masker.
RemoveFileNamed:	Verwijder het gespecificeerde bestand.
LoadList	Laad een lijst met daarin een klasseboom.
saveList	Bewaar een lijst met daarin de klasseboom.
getItemFromList:	Destilleer een klasse uit de lijst.
GetChildrenFromList:	Destilleer de kinderen uit de lijst van een bepaalde klasse.
InitializeTree	Initialiseer de klasseboom. Hierin wordt standaard de klasse Object opgenomen.
AddSubClass	Open een scherm waarin subklassen van de geselecteerde klasse kunnen worden geselecteerd en voeg deze toe aan de boom.
DeleteClass	Verwijder de geselecteerde klasse.
DeleteSubClass	Open een scherm, waarin de in de boom opgenomen subklassen van de geselecteerde klasse zijn opgenomen. In dit scherm kunnen de te verwijderen klassen geselecteerd worden.
editMethodsOfSelectedClass:	Wijzig de te distribueren methoden van een bepaalde klasse.
performAction: onClasses:	Deze methode wordt aangeroepen na het sluiten van één van bovenstaande schermen. Deze methode voert de actie uit.
performFilterOn:	Filter bepaalde klassen uit een lijst die voor distributie niet interessant zijn.
selectedItemMethodsBecome:	De methoden van de geselecteerde klasse wordt gelijk gemaakt aan het argument dat een collectie van methoden is.
selectionIsChanged	Deze methode wordt aangeroepen, als de selectie in de boom verandert. Hier worden de pop-menu acties aangepast.
include	Inverteer het attribuut included van het geselecteerde item.
generateJavaSource	Genereer de source code van de klasseboom.
getSourceFor:	Retourneer de source code voor een bepaalde tree item.
nextPutJavaSource: to:	Voeg de Java source code toe aan een bepaalde stream.
nextPutString: to:	Voeg een string toe aan een bepaalde stream.
savaSourceFor:	Bewaar de Java source voor een bepaalde source klasse

Tabel 22 Methoden van de klasse TreeItem

TreeItem	
addChild:	Voeg het argument toe aan de lijst van kinderen van dit item.
asFileString	Retourneer dit item als string, die alle informatie bevat en opgeslagen kan worden in een bestand.
hasChildren	Retourneer true als er kinderen zijn en false als er geen kinderen zijn.

Tabel 23 Methoden van de klasse JavaClassSource

javaClassSource	
generateWarning	Genereer de warning string.
generatePackage	Genereer de package statement.
generateImports	Genereer de import statements.
generateHeader	Genereer de header van de klasse.
generateConstructors	Genereer de constructors van de klasse.
generateMethods	Genereer alle methoden van de klasse.
generateFooter	Genereer de footer van de klasse.
generateInstanceMethod: withArgumentCount:	Genereer de instantie methode van de klasse met een gespecificeerd aantal argumenten.
generateClassMethod: withArgumentCount:	Genereer de klasse methode van de klasse met een gespecificeerd aantal argumenten.
GenerateClassMethodOfSuperClasses:withArgumentCount: forClass:	Genereer de klasse methode van alle superklassen van de klasse. Deze worden opgenomen in de huidige klasse.
generateJavaSourceFor:	Genereer de Java source voor een bepaalde tree item.
returnStringWithout:	Retourneer de string zonder de substrings in het argument.

De gedistribueerde klasseboom die met behulp van de Javaproxygenerator gegenereerd kan worden behoudt de hiërarchie, zoals deze in Smalltalk voorkomt. Hierdoor blijven de eigenschappen van alle klassen zoals inheritance, polymorfisme en encapsulatie gehandhaafd.

Lijst van figuren

Lijst van tabellen

Literatuurlijst

[Rumbaugh 1991]

Rumbaugh, J., Object-Oriented Modeling and Design, Englewood Cliffs, New Jersey, 1991, ISBN: 0-13-63005405

[Derr 1995]

Derr, K.W., Applying OMT; A practical step-by-step guide to using the Object Modeling Technique, Idaho Falls, Idaho, 1995, ISBN: 1-884842-10-0

[IBM 1995a]

IBM Corporation, VisualAge for Smalltalk: User's Reference version 3 release 0, 3e druk, New York ,1995

[IBM 1995b]

IBM Corporation, VisualAge for Smalltalk: Programmer's reference guide to building parts for fun and profit: version 3 release 0, 2e druk, New York, 1995

[IBM 1995c]

IBM Corporation, VisualAge for Smalltalk: VisualAge for Smalltalk: Getting started: version 3 release 0, 3e druk, New York, 1995

[IBM 1995d]

IBM Corporation, IBM Smalltalk: Programmer's reference version 3 release 0, 3e druk, New York, 1995

[IBM 1995e]

IBM Corporation, Introduction to Object Oriented Programming with IBM Smalltalk version 3 release 0, 3e druk, New York, 1995

[ELC 1995]

ELC Object Technology, ELC persistancy framework technical report, Capelle a/d IJssel, 1995.

[ELC 1996]

ELC Object Technology, ELC persistancy framework user manual, Capelle a/d IJssel, 1996.

[Lemay 1996]

Lemay Laura & Perkins Charles L., Teach Yourself Java in 21 days, 1e druk, Sams net, 1996, ISBN 1-575-21030-4

[Hoff 1996]

Hoff van, Arthur & Shaio, Sami & Starbuck, Orco, Hooked on Java; *creating hot web sites with Java applets*, Amsterdam, Addison Wesley, 1996, ISBN 0-201-48837-x (book) , ISBN 0-201-85274-8 (cd-rom)

[Manger 1996]

Manger, Jason J., Essential Java; *Developing interactive applications for the World Wide Web*, London, McGraw-Hill, 1996, ISBN 0-07-709292-9

[Bennet 1987]

Bennett, J.K.

"The design and implementation of Distributed Smalltalk"

ACM Sigplan Notices, 1987, Vol 22, Iss 12, pp 318-330

[Decouchant 1986]

Decouchant, D.

"Design of a Distributed object manager for the Smalltalk-80 system"

ACM Sigplan Notices, 1986, Vol 21, Iss 11, pp 444-452

[OSI resources]

http://www.cstp.umkc.edu/personal/kkev/open-systems/open-system.html

http://www.nodec.com/tech/iso.htm

http://www.uwsg.indiana.edu/usail/network/nfs/network_layers.html

http://www.att.com/interspan/overview/att00417.html

http://www.citybeach.wa.edu.au/lessons/computing12/lan/iso.html

[OMG]

http://www.omg.org/

http://corbanet.dstc.edu.au/

Index