Taking a look inside the Java Virtual Machine

Jaarproject 2004 – 2005

Yet Another Java Profiler

http://yajp.sourceforge.net

Svetozar Misljencevic
Dusty Lefevre

Werkwijze

We hebben de ontwikkeling van de applicatie opgesplitst in twee grote delen. Voor het eerste deel wilden we een programma opleveren dat enkel selectief kon tracen. Het tweede deel zou dan verder bouwen op het eerste en de functionaliteit toevoegen om (real-time) de interne staat van de Virtual Machine weer te geven. Voor elke fase werd steeds de design in grote lijnen vastgelegd (usecases en basisklassen voor de verschillende sub-systemen). Dit hebben we gedaan om een consistentere design te bekomen, in de hoop achter af minder re-engineer werk te hebben. Voor elke functionaliteit hebben we dan de design in meer detail uitgewerkt, en begonnen met de implementatie. Na de implementatie werd steeds de design terug aangepast omdat de implementatie toch dikwijls durfde afwijken van het originele idee (vooral bij callbacks).

Ruwe design en keuze van libraries

Om toegang te krijgen tot het binnenste van de Java Virutal Machine kan je een zogenaamde *agent* gebruiken. Dit is een dynamische bibliotheek die bij het opstarten geladen wordt. Die agent kan dan oa. callbacks doorgeven aan de JVM die uitgevoerd worden bij bepaalde events, aan bytecode instrumentation doen, acties zoals garbage collection forceren, gegevens over de JVM of het lopende programma opvragen... Dit kan door gebruik te maken van de *jvmti* interface (opgelegd) en *jni*. De gebruiker moest ook in staat zijn om op een menselijke manier zo'n sessie te starten, en achteraf ook een trace te bekijken.

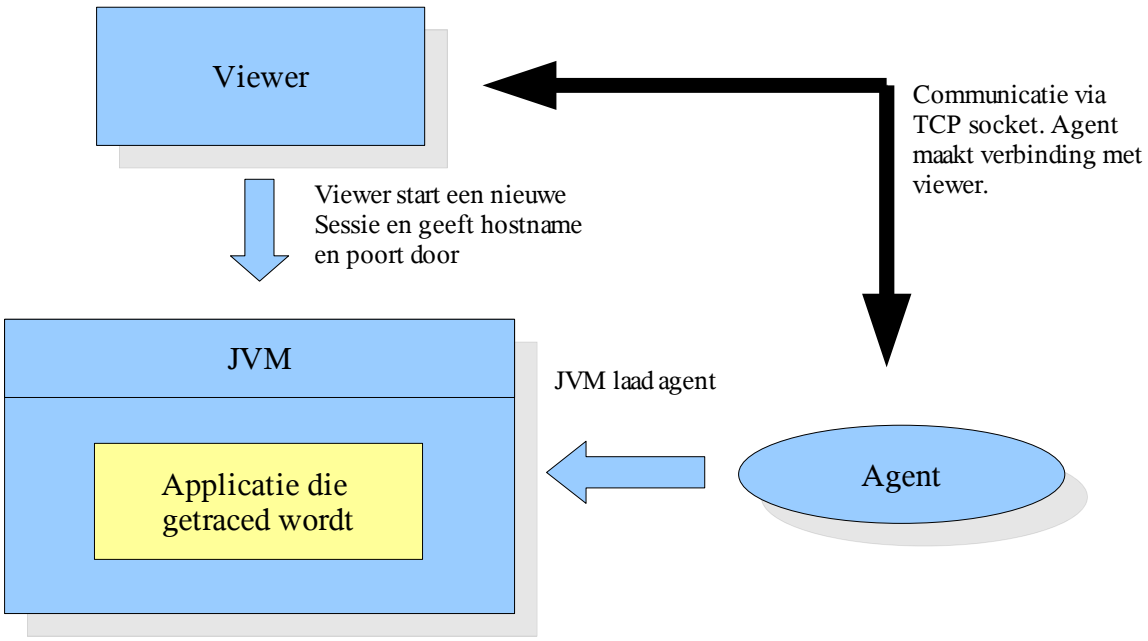
Om dit alles tot een goed einde te brengen moet de *viewer* (het gebruikersprogramma) in staat zijn om met de agent te communiceren (om onder andere de configuratie door te geven). Dit konden we op verschillende manieren doen: d.m.v. parameters, pipes (standard out/in), shared memory of sockets.
Alles via parameters (te tracen klassen, log opties, ...) zou de opstartstring enorm lang maken. Bij windows is deze lengte beperkt, wat voor problemen kan zorgen bij het tracen van heel grote programma's.
Pipes kunnen al door de applicatie die we willen tracen gebruikt worden. We kunne ze wel bij het opstarten gebruiken om een configuratie door te sturen, maar achteraf worden ze onbruikbaar. Dit zou dus een probleem vormen voor de tweede fase omdat we dan gegevens (at run-time) aan de agent moeten kunnen opvragen.
Shared memory hebben we bekeken, maar dit was vrij moeilijk te implementeren via java. Bovendien zouden we een tamelijk complex locking mechanisme nodig hebben.
Sockets (over localhost) vonden we een goede oplossing voor dit probleem. Ze worden ook dikwijls in soortgelijke programma's gebruikt. We konden hiervoor zowel het UDP als het TCP protocol gebruiken. UDP is beduidend sneller als TCP en testen heeft uitgewezen dat als de localhost gebruikt wordt dat pakketten in volgorde en zonder verlies aankomen. TCP brengt echter een gesloten sessie tot stand tussen de twee kanten, en is dus een makkelijkere oplossing. Bovendien is de hoeveelheid data die we moeten versturen tussen viewer en agent nu ook weer niet zo groot, dus een TCP socket zou moeten volstaan.
Onze oplossing: de viewer start voor het starten van de JVM een server op. Bij het opstarten van de JVM worden hostname en poort waarmee de agent moet connecteren als parameters doorgegeven. De agent maakt een verbinding met de server, de viewer stuurt hierop de configuratie door (als xml). Het communicatiekanaal wordt gedurende heel de verbinding opengehouden, dit konden we later gebruiken om de interne staat van de JVM met op te vragen.

Het is evident dat de opslag van gegevens niet in de viewer kon gebeuren (we zouden immers alles via de socket moeten doorgeven). De agent slaat dus alle events in een log op in xml. Omdat performantie hier belangrijk is, gebruiken we gebufferd schrijven. Om de grootte van het bestand en ook vooral de disk overhead te beperken comprimeren we het bestand (on-the-fly) d.m.v. *zlib*. Op xml heeft dit een prachtig effect.
Om een voorbeeld te geven: een klein logbestand van een 73.000 tal events, met als compressiegraad 3 is ongeveer 650 kb groot, ongecomprimeerd ongeveer 8,5 Mb groot (een goede 13 keer groter).

Een ander mooi voorbeeld: een trace (±4,5 miljoen events) van het java spel Tyrant (http://sourceforge.net/projects/tyrant), met compressiegraad 1 geeft een logfile van 24,5 Mb, ongecomprimeerd 539 Mb.

We hebben xml als opslagformaat gekozen omdat hiervoor goede tools bestaan. Ook voor de communicatie tussen agent en viewer gebruiken we xml. Door middel van een sax parser konden we makkelijk een events driven systeem implementeren voor het opvragen van de interne staat van de JVM.

Dit gaf ons een systeem dat schematisch als volgt voor te stellen is:



Omdat we wilden dat de applicatie cross-platform zou zijn, moesten we de libraries van de agent zorgvuldig kiezen. Voor de viewer was dit niet zo'n probleem aangezien java hier voor ontworpen is. Het enige probleem dat we daar zijn tegengekomen was bij de generatie van de opstartcommando's, die op verschillende platformen al eens durven verschillen. Voor de ontwikkeling gebruikten we vooral linux, maar uiteindelijk moest het programma ook op windows kunnen draaien. Een aantal onderdelen van de agent waren fundamenteel anders tussen de verschillende platformen: multithreading, sockets en timing.
Multithreading: Multithreading zal nodig zijn voor interactie tussen de viewer en de agent. Onder linux wordt meestal gebruik gemaakt van de pthreads library. Deze library is niet standaard onder windows beschikbaar, en vereist de installatie van een extra bibliotheek. Het was ook niet zeker of de interface wel volledig dezelfde zou zijn.
Sockets: Fundamenteel voor de communicatie tussen tussen viewer en agent. Onder linux worden BSD Sockets gebruikt en onder windows WinSock.
Timing: Als we de tijd waarop een event voorviel willen vastleggen, zullen we tijdmetingen moeten doen. Ook deze functies verschillen van platform tot platform.
Al deze functionaliteiten maken deel uit van de Apache Portable Runtime library (http://apr.apache.org), dewelke zowel voor windows als linux (*nix) beschikbaar is.

Als xml parser library hebben we Xerces gebruikt. Dit zowel voor de viewer (xerces-j) als voor de agent (xerces-c). We hebben altijd gebruik gemaakt van de event georiënteerde SAX parser. Voor de communicatie tussen agent en viewer was deze keuze voor ons duidelijk. De viewer stuurt immers requests waar agent gewoon op antwoord. Maar we hebben ook een SAX parser gebruikt voor het inlezen van onze logfile. Dit omdat de DOM parser (wat een duidelijkere keuze is voor

documenten) een volledig bestand zal proberen in te lezen. Dit is niet wenselijk omdat de logfiles enorm groot kunnen worden. Daarom ook onze keuze voor een SAX parser, die bursts xml zal aankrijgen.

De reden dat we zlib als compressie library hebben gekozen is de volgende: zlib is voor een groot scala aan platformen beschikbaar, beschikt over een eenvoudige interface, kan on-the-fly comprimeren (wat voor ons wel noodzakelijk is), en is ook onder java op een transparante manier (streams) beschikbaar. Bovendien is zlib ook zeer snel, de overhead bij compressie is heel klein en er wordt heel wat disk overhead weggenomen door de compressie.

We hebben ook een helpsysteem ingebouwd in de viewer. Hiervoor hebben we HelpGUI (http://helpgui.sourceforge.net) gebruikt. Het laat ons toe de pagina's in eenvoudige html te schrijven en een indextabel in xml op te geven. Reden van onze keuze: simpel en proper!

Detecteren van events

Er zijn twee methoden om events (method calls, return calls, thread start/stop events, ...) te detecteren. Er is een systeem dat voor elke JVM werk, nl. bytecode instrumentation en een systeem dat niet noodzakelijk voor elke JVM (aangezien het niet verplicht te implementeren is volgens de jvmti specificaties), nl. callbacks. Bytecode instrumentation heeft als voordeel dat het snel is. Het is echter ook heel complex, en na lang zoeken hebben we geen vrij te gebruiken library gevonden. We hebben daarom besloten dat dit een veel te riskante methode was, aangezien we onmogelijk konden inschatten of we deze methode ook met succes konden implementeren. Prototypes maken, om de haalbaarheid te bepalen, had equivalent geweest met bijna een volledige agent implementeren. Callbacks zijn trager, dit omdat er meer callbacks gemaakt worden dan eigenlijk nodig zijn (alle methodes/ tegenover enkel de methodes die we willen tracen). Ze geven enkel id's terug voor methoden, klassen en threads. De signaturen van methodes, klassen, ... moeten opgevraagd worden door middel van jni calls. Deze calls introduceren een significante overhead. Deze overhead proberen we te reduceren door middel van caching.
De cache werkt als volgt: we hebben de signaturen van alle klassen die we willen loggen in een Configuratie object staan. Wanneer een klasse geladen wordt door de JVM volgt er een callback. Hier vragen we alle methoden op van de klasse, en deze geven we door aan de cache. In de cache wordt er bepaald of het hier over een klasse gaat die we willen loggen, en indien dit zo is vragen we alle signaturen op en slaan deze op in de cache. Er worden ook de nodige globale referenties aangemaakt, om de klasse in het geheugen te houden en te garanderen dan methode id's niet veranderen.
De callbacks van de ander events kunnen hierna aan de hand van bijvoorbeed een methode id een Method object opvragen dat al de nodige informatie bevat, zonder dat er ook maar één jni call nodig is. Als een klasse niet gelogd moet worden geeft de cache een null referentie terug en wordt niets gelogd.

Het enige probleem dat we origineel met deze aanpak hadden was dat de id's de neiging hadden om te veranderen. Dit hebben we opgelost door globale referenties (die geldig blijven buiten de scope van de callback) aan te maken. De cache maakt gebruik van een map om efficient het juiste object te kunnen localiseren.

De c interface van APR

De interface van APR is geïmplementeerd in c. Dit maakte het lastig om deze functies in c++ objecten te gebruiken en de boel tegelijk overzichtelijk te houden. Hiervoor hebben we diverse wrappers voorzien. De belangrijkste is de MManager klasse die de geheugen pools beheerd. Verder is er een Clock wrapper voor de tijdsmetingen, een Mutex en Conditional klasse, en een wrapper die een Thread klasse voorziet. Via inheritance en implementatie van een puur virtuele run methode kan op deze manier makkelijk met multithreading worden omgegaan.

Gebruikte patterns in de agent

Het meest gebruikte pattern is de Singleton. Dit heeft onze code “mogelijk” gemaakt. Verder worden de binnenkomende xml tags door een Chain of responsibility afgehandeld. De tags worden over een ketting van handlernodes doorgegeven tot een node hem afhandelt. Wordt de xml tag niet herkent dan valt hij van de ketting af en wordt hij genegeerd. Zo'n chain brengt wel de nodige overhead met zich mee, daarom worden command tags (dewelke instaan voor de requests voor real-time informatie) als eerste in de ketting afgehandeld.

Realtime informatie opvragen

Dit wordt gedaan door een periodieke request van de viewer. Deze activeert code (die door de agent bij een commando centrum registreert wordt als callbacks), die via de xml handlers geactiveerd wordt. In het geval van de threads, stack informatie of stacktraces worden jni calls of gecachte informatie gebruikt. Bij de heap wordt er iets anders tewerk gegaan. Wanneer een klasse geladen wordt zal deze een tag krijgen, in ons geval een referentie naar een object dat de instanties van deze klasse accumuleert. Wanneer de heap opgevraagd wordt zal er over de heap geïtereerd worden. De referentie wordt dan doorgegeven als tags aan een instantie. We itereren enkel over instanties die nog geen tag hebben. Hierdoor wordt de overhead beperkt. Desondanks deze methode kan de overhead toch nog enorm worden bij programma's die veel objecten aanmaken. Het taggen kan dan ook optioneel uitgeschakeld worden in de viewer.

Problemen bij het implementeren van de agent

Met de jvmti interface hebben we één groot probleem tegengekomen. Het taggen van klassen en objecten deed de JVM op onbekende wijze (at random) crashen. De stackdump die de JVM dan genereerde duide op een segmentation fault in libc.so. We hebben nooit gevonden wat deze crash veroorzaakte en hebben het probleem “omzeild” door het taggen in een aparte jvmti environment te laten gebeuren. Hierna hebben we geen problemen meer gehad met de tags.
APR heeft ons de nodige kopzorgen bezorgt. Blijkbaar is de platform onafhankelijk bibliotheek niet zo onafhankelijk. Een groot deel van de implementatie hebben we eerst onder linux gedaan. Toen we begonnen te testen onder windows bleken sommige socket opties, die we gebruikten onder linux niet beschikbaar (en zelfs niet geïmplementeerd) onder de windows versie. De documentatie van APR is soms nogal beperkt. Een misinterpretatie van de multithreading functionaliteit heeft tegen het einde van het project voor serieuze kopzorgen gezorgd. Deze bug zorgde soms voor random segfaults. We hebben ze gevonden omdat er een specifieke fout optrad, maar wel enkel onder windows, onder linux liep het onstabiel, maar zonder foutmeldingen.

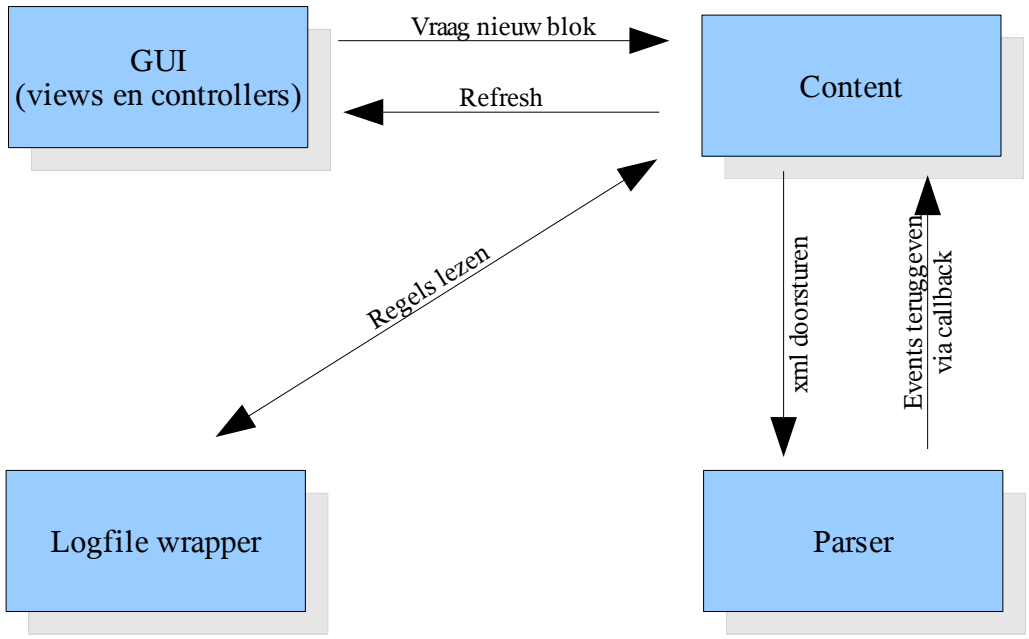
Een programma tracen (viewer)

Dit hebben we geimplementeerd door een singleton Session object. Wanneer een sessie opgestart wordt zal er een Configuration object doorgegeven worden. Hierna is het mogelijk een nieuwe sessie te starten. Het Configuration object bevat onderandere de te loggen klassen, filters, ... Er is ook een methode voorien om deze configuratie uit te lezen als een xml stream. Het path van de JVM wordt in een globaal SysConfiguration object gewaard.
Juist voordat de nieuwe JVM met de agent opgestart wordt zal er een server (in een aparte thread) geactiveerd worden die zal instaan voor de viewer – agent communicatie. Nadat er een verbinding tot stand is gebracht wordt de gui via een callback op de hoogte gebracht, en koppelt deze een Terminal venster aan de sessie. De sessie brengt dit terminalvenster via callbacks op de hoogte van nieuwe uitvoer op stdout en stderr. Invoer in het terminalvenster wordt geforward naar de sessie.

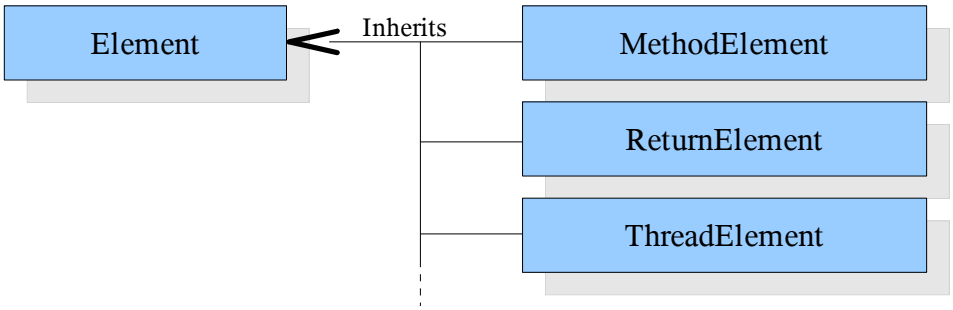
Een log bekijken

Voor het weergeven van de logs stelde zich volgend probleem: een log wordt groot, in een eerder voorbeeld leverde een korte trace ongeveer 4,5 miljoen gelogde events op. Het is niet realistisch om dit allemaal in het werkgeheugen te laden. Daarom konden we de DOM parser al ni niet gebruiken. Wat wij hebben gedaan om dit probleem op te lossen is onze applicatie steeds maar een blok uit de logfile laten inlezen (bv. 1000 events). Wanneer de gebruiker een ander deel van de logfile wilt

bekijken kan hij dan een “border” vooraan of achteraan het venster bijladen. Dit aantal events wordt dan aan de ene zijde toegevoegd, terwijl er aan de andere kant events worden verwijderd. Hierdoor moesten we makkelijk het aantal events in ons logbestand kunnen tellen, zonder dat er al xml geëvalueerd zou worden. Hiervoor laten we de agent alle events op een nieuwe lijn wegschrijven. In de viewer konden we dan een LineNumberReader van java gebruiken. De xml tags worden dan via een stream doorgestuurd naar de parser. Deze zal via callbacks een Content object aanpassen. Wanneer alle lijnen doorgegeven zijn zal de parser een <refresh_event/> tag aankrijgen. Hierna brengt de parser de content op de hoogte dat de GUI aangepast mag worden. De Content doet dit via geregistreerde callbacks.



Het Content object houd de verschillende events bij als abstracte elementen (Element klasse). De implementaties van deze abstracte Element klassen implementeren eventspecifiek gedrag.



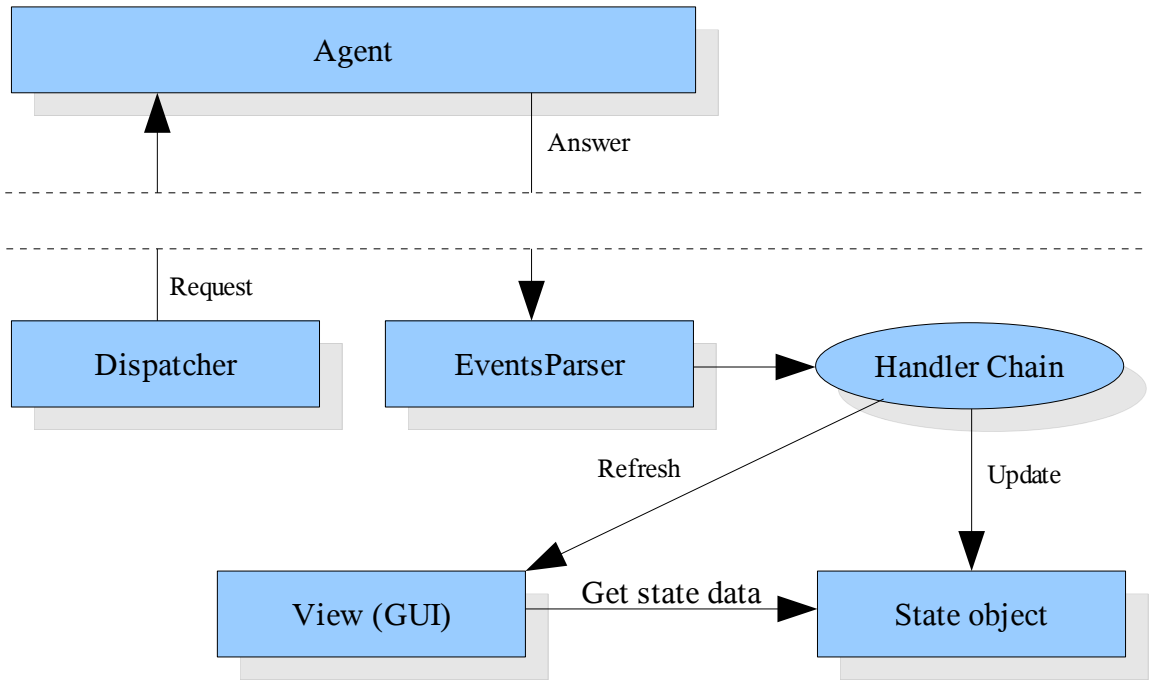
Het Content object wordt ook gebruikt om de codecoverage te genereren. Dit hebben we vooral gedaan om niet teveel code te krijgen die eigenlijk dezelfde taak uitvoert op een andere schaal. Het CodeCoverage object zal hiervoor een nieuw Content object aanmaken (zodat de huidige weergave ongemoeid blijft) en de bestaande functionaliteit gebruiken om achtereenvolgens blokken te laden en deze te accumuleren.

Filters

We hebben geprobeerd om onze filters op een zo'n dynamisch mogelijke manier te implementeren zodat veranderingen makkelijk zouden blijven. Hiervoor hebben we een Filter object gespecificeerd. Dit object is een container dat regels bevat van de vorm: sleutel, operatie (equals, differs, matches), en een waarde. Deze filter wordt door de GUI ingevuld en doorgegeven aan het Content object. Bij het laden van een venster wordt dit filterobject doorgegeven aan elk Element dat zichzelf tegen deze filter verifieert.

Realtime de staat van de JVM weergeven

Om de agent deze gegevens te laten doorsturen moet de viewer op regelmatige tijdstippen om nieuwe gegevens vragen. Dit wordt gedaan via de Dispatcher. De Dispatcher is een actieve singleton die (op voorwaarde dat er de corresponderende views ingeschakeld zijn) periodisch commando's (in xml formaat) naar de agent stuurt. De binnenkomende xml wordt wederom door xerces geparsed, en de tags geanalyseerd met een Chain of responsibility pattern. Hier kan de GUI laag echter stukken callback code registreren bij de nodes in de ketting. Deze code wordt bij het volledig afhandelen van een antwoord (van de agent) uitgevoerd en staat in voor het updaten van de weergave. De gegevens die door de agent worden doorgezonden worden opgeslagen in State objecten (singletons) waar zowel de handlers als de GUI makkelijk bij kunnen.



Gebruikte patterns in de viewer

We hebben vooral veel gebruik gemaakt van Singletons. Voor het parsen van de xml tags wordt steeds een Chain of responsibility gebruikt. De GUI werkt volgens het View/Controller/Model principe. Een view registeerd een callback (listener) bij de datamodellen (Content, realtime package), controllers (load next block, ...) kunnen gebruikt worden om het model te manipuleren, en wanneer er een update nodig is zal de view hiervan op de hoogte worden gebracht via de callback.

De grote YaJP klasse

We maken gebruik van een desktop pane waarin de verschillende views geladen worden. Omdat we ons niet wilden vastwerken in dit principe zijn alle views als JPannels geïmplementeerd. Hierdoor kunnen we makkelijk de samenstelling van de GUI (bv. met tabs) wijzigen. Daarom hebben we de code voor het aanmaken van interne vensters in de YaJP klasse (onze opstartklasse) ondergebracht. Dit samen met de code voor de menu's, de toolbar, en diverse listeners leverde ons een draak van een klasse op. Na een tijd werd werken aan deze klasse moeilijk en onoverzichtelijk. We hebben ze daarom uiteen getrokken in drie kleinere klassen waar de ene de andere uitbreidt door middel van inheritance. Het resultaat: 3 klassen. De eerste levende een abstracte structuur die oa. een Content variabele bijhoud, een methode voor window placement, en protected methodes voor de subklassen. De eerstvolgende subklasse zorgt voor het weergeven van de views, voorziet een protected interface om deze makkelijk in en uit te schakelen en methodes om te controleren wat wordt weergegeven. De laatste subklasse (de YaJP klasse zelf) voegt menu's en de toolbar toe, implementeert de corresponderende listeners en bepaald het uiteindelijke externe gedrag.

Het totaal aan code is hiermee wel groter geworden, maar het heeft ons wel veel meer overzicht opgeleverd. Ook latere uitbreidingen waren makkelijker met deze structuur.

Teststrategie

We hebben hoofdzakelijk gewerkt met unit tests, dit zowel voor de viewer als voor de agent. Voor de viewer hebben we ook (in beperkte mate) contracten (asserties) gebruikt. Dit vooral om inconsistent gedrag te detecteren. Bijvoorbeeld de Sessie die tweemaal gestart wordt, views die geactiveerd worden terwijl ze eigenlijk al actief waren, ... Voor de unit test voor de viewer hebben we JUnit gebruikt, bij de agent cppUnit.
Voor de viewer hebben we praktisch elke klasse individueel getest. De probleemgevallen hier waren de Content en de Session klasse. Deze lieten zich moeilijk testen en zijn eerder met de integratie geverifieerd. Wanneer elke klasse voldoende getest was hebben we deze ook op de integratie (tot zijn subsysteem althans) getest. Om een voorbeeld te geven: bij een Chain of Responsibility werden eerst de individuele handlers getest, daarna de werking van de handlers in de chain, en hierna nog eens de werking van de chain in coöperatie met de parser. Deze testen waren meestal vrij eenvoudig, maar vereiste soms wel het verbreken van Singletons, om hun instanties te resetten en/of te vervangen door stubs. Voor de viewer levert dit ons 80 tests.
Bij de agent werden ook de meeste onderdelen getest, zowel individueel als op integratie. Dit was echter moeilijk voor de threads. Deze zijn nu eenmaal moeilijk te verifiëren. We hebben ze wel kort uitgetest, maar deze code hebben we nooit tot een echte test gemaakt. Bovendien bleek ze toen goed te werken, maar zoals al eerder vermeld zijn we met de Thread implementatie toch een serieuze bug tegengekomen die heel onvoorspelbaar leek. Een vergeten macro veroorzaakte bij langere executie random crashes.
Omdat we veel hebben moeten experimenteren met onze agent hebben we een echte regression test voor de agent pas helemaal tegen het einde toegevoegd. Deze is in java geschreven en maakt een trace van een kort programma en verifieert het bekomen log tegen een voorbeeld. Hierbij worden gelogde tijden en object id's uitgesloten (nodig, want dit is nooit hetzelfde). Verder wordt ook getest of voor een klein voorbeeldprogramma, de agent een geldige staat teruggeeft voor heap, stack en threads. Dit is mogelijk door het testprogramma te laten blokkeren en daarna een request naar de agent te sturen.

De installatie

Elke release hebben we verpakt in een installer-jar. Deze jar genereren we IzPack (http://www.izforge.com/izpack). De jar bevat alle nodige java libraries, de viewer, en agents voor windows, linux, solaris 9 (x86) en solaris 9 (sparc). De solaris agents hebben we niet kunnen testen bij gebrek aan aan solaris 9 systeem, maar zouden in principe moeten werken. Ook de broncode kan geïnstalleerd worden via deze jar. Het is dus altijd mogelijk de agent zelf te compileren.
Het is wel noodig om xerces-c, apr, en voor windows zlib te installeren. Er zijn dll's beschikbaar voor windows via de website. Onder linux en solaris moeten deze zelf geïnstalleerd worden. Bovendien moeten deze ook door de agent gevonden kunnen worden.

Documentatie

Alle documentatie van het project kan teruggevonden worden op de website van het project (http://yajp.sourceforge.net). De verschillende planningen, taken, usecases, uml diagrammen, ...
We hebben klassediagrammen voor de viewer en de agent, Usecase diagrammen voor de twee fases van het project, een state diagram voor een trace, en collaboratiediagrammen voor de cache en de Chain of Responsibility in de agent. De laatste eerder om toen het concept aan te tonen, want we hebben geen soortgelijke diagrammen gemaakt voor de andere toepassingen van dit pattern.
De syntax van de xml voor het logbestand, en voor communicatie tussen viewer en agent is vastgelegd met DTD's.
We hebben ook een eenvoudige handleiding gemaakt. Deze zit ingebouwd in de applicatie als een help systeem. Hierin worden heel kort de verschillende functionaliteiten besproken. Alle pagina's zijn in HTML geschreven.

Nabeschouwing op de planning

We hebben uiteraard regelmatig aan onze planning moeten sleutelen. De eerste grote vertraging liepen we al heel vroeg op met het implementeren van de trace sessies. In de viewer was deze vertraging nog onbeduidend, maar in de agent tamelijk groot. We hebben de complexiteit die met dit systeem gemoeid was hier wat onderschat. De geplande tijd hiervoor was 2 weken, dit is echter serieus uitgelopen. Het grootste probleem was het implementeren van de cache. We veronderstelden dat dit redelijk makkelijk zou zijn. We begrepen toen echter de noodzaak van global references nog niet, hierdoor veranderde de id's van methoden en klassen dikwijls. We hebben ook enkele malen onderdelen van onze design moeten herzien. Vooral omdat bepaalde functionaliteiten van de JVMTI niet deden wat we eerst hadden geanticipeerd. De bufferperiodes waren niet voldoende om alle vertragingen op te vangen. Na fase 1 hebben we tijd, gereserveerd voor de design, ingekort zodat we de implementatie wat verder konden afwerken.
Een usecase hebben we niet geïmplementeerd, nl. het automatische stoppen wanneer een log te groot wordt. De grootte van een bestand is moeilijk te bepalen in de agent vanwege de compressie. Het stond opgegeven als “nice to have”. We hadden echter geen tijd meer om dit er nog mee in te stoppen en is dus van de planning gevallen.
Het debuggen is ook langer uitgedraaid dan we geanticipeerd hebben (tegen het einde dan toch). We hebben veel tijd verloren met de bug in de Thread klasse van de agent. Deze bug manifesteerde zichzelf in diverse soorten random crashes. Na lang zoeken zijn we er uiteindelijk toch op uitgekomen, en een drietal bugs, die we als kritiek beschouwden, verdwenen mee.

Conclusie

Met nieuwe en experimentele technologie werken is niet altijd even makkelijk. De documentatie van de JVMTI interface is wel goed, maar op sommige puntjes komt hij soms toch wel wat tekort waardoor misinterpretatie wel eens voorvalt. Eigenlijk is een basiskennis van JNI toch vereist. Het concept van globale en lokale referenties wordt in de JVMTI documentatie maar kort aangehaald, in de JNI documentatie staat dit gedetailleerd uitgelegd. Ook is er geen grote gemeenschap op het internet aanwezig. Voor basisproblemen is er meestal wel hulp te vinden, maar wanneer het echt misloopt blijft een vraag op usenet dikwijls onbeantwoord. Buffers zijn belangrijk, en deze hebben we toch iets te krap ingeschat. Zeker de andere projecten namen op het einde veel van onze tijd in. We hadden nog graag de Content overgezet op XPath, omdat dit eigenlijk een betere oplossing is dan de SAX parser. Dit is wegens tijdgebrek niet meer gelukt.
Over het algemeen (buiten de Content klasse, dit toch nog wat gerefactor kan gebruiken) zijn we tevreden over het resultaat.