"Entwicklung eines XML-Testwerkzeuges"

Studienarbeit von Marcus Wolschon <marcus.wolschon@calcucare.com>

2004-03-01

XML-basierende Sprachen haben sich auf breiter Basis für die Beschreibung von Kommunikationsprotokollen und Datenstrukturen durchgesetzt. Vor allem gilt dies für sogenannte "web-services", also Programme, deren Kommunikation statt auf RMI, Corba oder DCOM auf das verbreitete http-Protokoll und darüber ein auf XML basierendes Protokoll wie SOAP aufsetzen, finden zunehmend mehr Zuspruch. Entsprechend werden vermehrt auch Werkzeuge benötigt um diese zunehmend komplizierteren Dienste sowohl auf Fehler hin als auch auf ihre Leistungsfähigkeit hin zu testen. Besonders auffällig wird dieser Mangel bei Einsatz von Rapid-Prototyping Entwicklungsmethoden, in denen automatische Tests während der gesamten Entwicklungszeit die Qualität und Funktionsfähigkeit aller Komponenten sicherstellen sollen. In dieser Arbeit wird nun ein bestehendes einfaches Werkzeug für Lasttests, welches lediglich nach erheblicher Einarbeitungszeit und mit einem ebensolchen Pflege-Aufwand von einem Entwickler verwendet werden konnte, um lediglich die Leistungsfähigkeit eines speziellen Gesamtsystems zu überprüfen, durch erhebliche Umbauten zu einem einfach verwendbaren Werkzeug sowohl für die immer wichtigeren Last- wie auch Funktions- und Regressionstests erweitert, mit welchem alle Arten von Web-Services ohne großen Aufwand getestet werden können.

Zunächst möchte ich in Kapitel 1 einige Anregungen geben, zur Lösung welcher Probleme das hier vorgestellte Werkzeug erstellt und erweitert wurde. Danach bespreche ich in Kapitel 2 kurz die Ausgangslage vor Beginn der Arbeit und gehe dann in den Kapiteln 3 bis 6 auf die Details der gewählten Lösung ein. Von der Wahl der beiden Testverfahren, welche den Kern des neuen Werkzeugs bilden, nach außen hin bis zur Präsentation für den User mittels des Testsatz-Editors, welcher den größten Teil der Arbeit beinhaltet.

Inhaltsverzeichnis

1 Motivation

Im Folgenden möchte ich zunächst einige Beispiele geben, welche mögliche Einsatz-Szenarien des im Rahmen dieser Arbeit erstellten Werkzeuges illustrierten sollen.

1.1 Der Mail-Server

Ein großer Mobilfunk-Anbieter bietet seinen Kunden auch den Zugriff auf Email sowohl über (X-)HTML als auch über mobile Geräte mittels WML an.

Bisher sind der Firma nie Probleme mit diesem Dienst zu Ohren gekommen, obwohl tausende von Kunden das System benutzen. Nun möchte die Firma ihren Kunden einen Pauschal-Tarif für den mobilen Zugriff auf ihre Email anbieten, erwartet also einen steigenden Bedarf und verlagert vorausschauend das WML-System auf einen schnelleren Server.

Es wurden Tests mit Shell-Skripten durchgeführt, welche schnell hintereinander die Mail-Übersicht abrufen und meinte, das System sollte der Last gewachsen sein.

Wenige Tage später kam der erwartete Ansturm und die Zugriffszeiten gingen in inakzeptable Höhen.

Was ist passiert?

Im Datenbank-Design wurde eine Tabelle für die Emails benutzt, welche Kunden-Nummer, Betreff und ID-Nummer der Mail enthielt. Für jeden schreibenden Zugriff hatte der unzureichende Datenbank-Treiber einen Schreib-Lese-Spinlock auf die gesamte Tabelle gesetzt, welcher bis zum Abschluss der schreibenden Aktion die gesamte Tabelle für lesende und schreibende Zugriffe gesperrt hatte.

Das Löschen einer Email ist nun ein schreibender Vorgang auf dieser Tabelle und durch das gleichzeitige Löschen der unerwünschten Email vieler Kunden mussten neben diesen auch alle weiteren, welche ihre Mails nur lesen wollten, warten und die mittlere Reaktionszeit stieg. Da der Fehler im Datenbank-Treiber lag, stiegen auch unerwartet die Antwortzeiten auf den kaum beanspruchten HTML-Server und die Firma verlor beträchtliche Kunden-Stämme und Ansehen an die Konkurrenz, obwohl doch der neue Server unter viel stärkerer als der eingetretenen Last getestet worden war.

Warum wurde der Fehler nicht gefunden?

Getestet wurde die einzige Aktion, welche den meisten Datenverkehr und damit (nach Meinung der Entwickler) die größte Belastung auf dem Server schaffen sollte. Nun war das Löschen einer Email aber eine Aktion, welche nahezu vernachlässigbaren Datenverkehr erzeugte, und wurde folglich nicht getestet. Weiterhin wurden nur viele Anfragen sehr schnell hintereinander in einer Schleife getestet, nicht gleichzeitig. Selbst wenn man die Löschen-Operation so getestet hätte, wäre das Spinlock immer nur von nur einem Prozess gleichzeitig beansprucht worden und somit nie ein Konflikt aufgetreten.

Wie hätte man besser testen können?

Mit dem hier entwickelten Werkzeug hätte man aus dem Entwicklungsprozess stammende Aufzeichnungen sowohl von einzelnen Aktionen als auch von

kompletten realen Sitzungen von Menschen, welche den Integrationstest des Systems durchführten, heranziehen können und diese von vielen Rechnern aus gleichzeitig mit mehreren simulierten Usern mit realistischen Verzögerungszeiten gegen den neuen Server laufen lassen können.

Zum einen hätte man so schnell und früh die unentdeckten Abhängigkeiten zwischen mehreren schreibenden Zugriffen entdeckt als auch die_ unentdeckt gebliebene_ ineffiziente Daten-Haltung von offenen Sitzungen. Denn durch die realistischen und zufälligen Verzögerungen zwischen den Aktionen jedes einzelnen Users hätte der Server die Kontext-Daten ihrer Sitzungen vorhalten müssen statt wie im durchgeführten einfachen Test immer nur eine Sitzung für kurze Zeit zu halten.

1.2 Das modulare System

Ein größeres Datenerfassungssystem soll entwickelt werden. Da das System sehr stark skalieren soll und viele zusätzliche Auswertungen und Archivierungen von Daten an verschiedenen Orten eingeplant sind, entscheidet man sich für ein modernes verteiltes Komponenten-Modell. Als Kommunikationsmittel entscheidet man sich aufgrund seiner Popularität und in Anbetracht der Tatsache, dass etliche der Rechner in weit entfernten Netzen hinter unterschiedlich konfigurierten Firewalls stehen, für das XML-basierende SOAP über HTTP. [SOAP]

Man einigt sich auf eine Reihe einfacher Protokolle und verschiedene Gruppen beginnen kleine Module zu entwickeln, welche mit diesen Protokollen kommunizieren.

Nun steht jede der Gruppen vor dem Problem dass ihre halb fertiggestellten Module getestet werden müssten, aber die Module mit welchen sie kommunizieren sollen, noch nicht einsatzbereit sind. Einige der Gruppen entwickeln unit-tests, welche die vom Parser erzeugten Funktionsaufrufe ausführen, und stellen später Fehler in ihren Parsern für das gemeinsame Protokoll fest. Um die Tests auf dem aktuellen Stand zu halten, ist ein erheblicher Aufwand nötig, da jedes mal Kontext-Daten einer echten Verbindung für den unabhängigen Test einzelner Funktionen angelegt und vernichtet werden müssen. Am Ende stellt sich heraus, dass einige der Tests fehlerhaft waren und in der Eile, nach Anpassen eines Tests wieder am eigentlichen Modul voranzukommen, einige Module unter fehlerhaften Annahmen über den Zustand einer Sitzung zu Zeiten von Funktionsaufrufen entwickelt wurden.

Eine andere Gruppe erkennt, dass sie realistische Kommunikationsvorgänge simulieren muss, und steckt viel Arbeit in die Entwicklung von Testservern, welche die anderen Module nachempfinden. Zwar konnte_ durch viele Absprachen, einen durchdachten Entwicklungsprozess und exakte Spezifikation des Protokolls und aller späteren Änderungen daran_ verhindert werden, dass die Testserver sich von den späteren Modulen signifikant im Verhalten unterschieden, aber das Team kommt durch den enormen Aufwand in Verzug und gefährdet schließlich die Markteinführung des Produktes.

Wie hätte man besser testen können?

Zu Beginn der Entwicklung wurden das Protokoll und etliche beispielhafte Use-Cases bereits exakt spezifiziert. Hier hätte man das Schema für gültige Nachrichten jedes Nachrichten-Typs des Protokolls sowie die Beispiel-Fälle benutzen können um damit die einzelnen Module durch das hier entwickelte Pro-

gramm zu benutzen. Unit-Tests wären nur für die tatsächlichen kleinen units und nicht für das gesamte, in seinem Verhalten bereits sehr komplexe, Modul nötig gewesen. Neu entwickelte Testfälle und ausgefeiltere Schemata für Anfragen und Antworten hätte man zwischen den Teams ausgetauscht und der Aufwand für die Aktualisierung der Tests bestünde lediglich im Anpassen der Testfälle in einem hierfür gemachten Editor, welcher den User weitest möglich unterstützt und offensichtliche Fehler schnell aufzeigt. Die parallele Entwicklung der eigenen spezialisierten Test-Programme (mit dazugehörigem Debugging und der Spezifikation der Tests) wäre auf das tatsächlich sinnvolle Minimum reduziert worden. Der Gesamtaufwand und die Fehlerwahrscheinlichkeit im Schreiben der Tests wären ebenfalls wesentlich verringert worden. Außerdem ständen am Ende jedes Entwicklungszyklus bereits vollständige automatisierte Test-Szenarien für Lasttests und menschenlesbare Kommunikations-Beispiele zur Dokumentation und der Schulung neuer Entwickler für das System bereit. Weiterhin wäre der Einarbeitungsaufwand zur Verwendung der vielen kleinen und schnell aber teilweise fehlerhaft geschriebenen Testwerkzeuge stark reduziert, da die wenigen generellen Werkzeuge bereits eine eigene Dokumentation und Einführungen für neue Nutzer mitbringen.

1.3 Die automatisierte Entwicklung

Um schnell und mit wenig unnötigem Aufwand Programme zu entwickeln und zu ändern und dabei gleichzeitig die Qualität des erzeugten Codes zu steigern, setzt man heutzutage auf maximale Automation aller Schritte im Build-Prozess, welche nicht unbedingt von Hand ausgeführt werden müssen.

Hier hat sich Apache Ant[Ant] als Quasi-Standard in der Java-Entwicklung durchgesetzt, da es sehr flexibel in der Integration neuer Aufgaben und durch eine sehr saubere und übersichtliche XML-Datei zu konfigurieren ist. Selbstverständlich wurde das hier entwickelte Test-Werkzeug auch in einen ant-task integriert, so dass das folgende Szenario möglich wird:

Der Entwickler ändert nur noch den Code und ruft ant auf.

Ant checked den aktuellen Stand in den eigenen Zweig der Versionsverwaltung ein und beginnt mit der Übersetzung. Da im Code EJBs verwendet werden, lässt man in einem Ant-Task die fehlerbehaftete Arbeit des Generierens der 4 Interfaces jedes Beans, der von den Servern der verschiedenen Entwickler verwendeten XML-Deskriptoren und spezieller Schlüssel-Klassen automatisch erledigen. Nun laufen bereits die automatischen jUnit-Tests durch, während der Code gleichzeitig bereits für das Deployment auf den EJB-Server gepackt wird. Da der Server noch nicht läuft, wird er automatisch gestartet und der Code deployed. Nun laufen bereits mit einem einzelnen Testserver die Tests unseres Werkzeugs gegen den lokalen Server um nach den unit-tests auch die weitere unbeeinträchtigte Funktion des Gesamtsystems nach den durchgeführten Änderungen sicherzustellen.

Während der Mittagspause und jeden Abend werden wahlweise über den eigenen Scheduler des bisherigen Lasttest-Werkzeuges oder über einen cronjob, welcher einen ant-task startet, automatisch Lasttests gegen die Entwickler-Server gestartet. Dabei wird Ant's ssh/Telnet -Task verwendet um die Clients auf mehreren Maschinen zu starten. Die Jini-Gruppen der Clients und des Masters sind so gewählt, dass sich die verschiedenen (teilweise gleichzeitigen) Tests dabei nicht in's [ins] Gehege kommen und die Ergebnisse verfälschen, aber auch

nicht eine übertriebene Anzahl von Maschinen nötig ist. Zusätzlich werden jede Nacht automatisch von Ant die für die Integration in der Versionsverwaltung eingecheckten Code-Teile ausgechecked und sämtliche Tests automatisch gegen ein automatisch compiliertes und aufgesetztes Integrations-Testsystem ausgeführt, so dass am Morgen der Test-Ingenieur lediglich die fehlerhaften Änderungen an die Entwickler zurückweisen muss und nur noch wenige manuelle Tests durchführt, womit er nur noch halbtags durch diese Aufgabe belastet ist.

Da die Tests bereits vor dem Code von einem anderen Entwickler definiert wurden und auf den Entwicklungs-Servern ein ausgiebiges Logging definiert wurde, wobei die Tests bei einem Fehlschlag nicht nur die Fehlermeldung, sondern auch die für die Erforschung der Ursache des jeweiligen Fehlers nützlichen Status-Informationen ausgeben, ist der Entwickler selber fast nur noch mit dem Coden, der Dokumentation seiner Arbeit und der Spezifikation für seine Kollegen beschäftigt. Hierbei ist er durch die kurzen Pausen während des Ant-Durchlaufes auch dazu gezwungen, währenddessen tatsächlich an der Dokumentation zu arbeiten. Genauso wie ein sinnvolles Logging und damit auch eine gute Fehlerbehandlung im Programm erzwungen wird, was dem späteren Produkt natürlich zugute kommt.

2 Ausgangslage

2.1 Werkzeuge

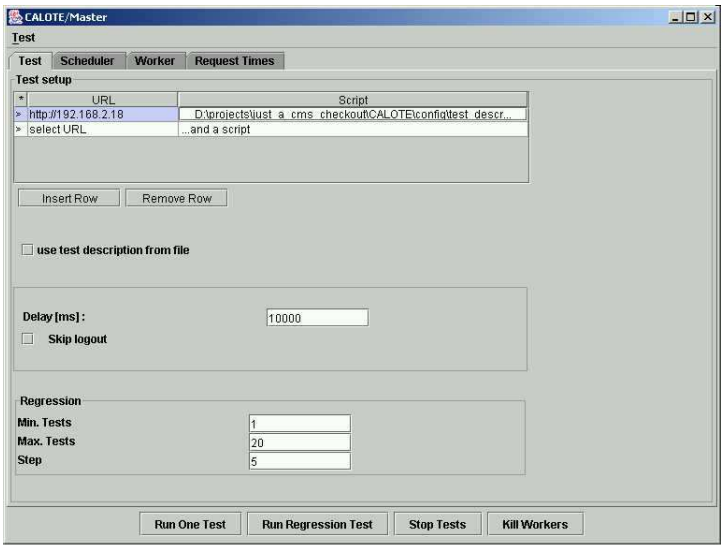

Abbildung 1: Bildschirmansicht des bisherigen Lasttest-werkzeuges

Zu Beginn der Arbeit existierte bereits ein reines Lasttest-Werkzeug namens Calote. (siehe Abbildung 1)

Es war fähig, von jedem seiner Clients aus eine Reihe von Prozessen mit zufälligen Verzögerungszeiten auszuführen und die Ergebnisse jeweils lokal auf

den Client-Rechnern zu loggen, aber dem zentralen Master eine Übersicht über mittlere Verzögerungszeiten und Anzahlen entdeckter Fehler zu melden.

Die Anfragen konnten Variablen enthalten, welche durch eine sehr simple, eigene, XPath-ähnliche Sprache aus bereits eingegangenen Antworten, globalen Variablen und speziellen Variablen mit (deterministischen, zufälligen oder speziellen fest-einkompilierten) Werten versehen werden konnten.

Die einzigen entdeckten Fehler waren grobe Kommunikationsfehler, wie Timeouts und Verbindungsabbrüche, sowie die Unfähigkeit, eine definierte Variable mit ihrem Wert zu belegen (da z.B. die referenzierte Antwort eine Fehlermeldung war).

Calote war darauf ausgelegt, ausschließlich mit einem einzelnen, speziellen Server zusammenzuarbeiten und bot dafür spezielle Unterstützung für Login und Logout vor und nach jedem Test-Prozess mit fest einkompilierten Adressen und Protokollen an.

2.2 Testsatz-Entwurf

Test-Sätze wurden von einem eigenen Mitarbeiter aus Mitschriften der Kommunikation zwischen dem regulären Client und dem Server von Hand in einem Text-Editor mit Hilfe einiger Batch-Dateien erstellt. Dies war in dieser Form mühsam zu pflegen wie auch fehleranfällig.

2.3 Einsatz

Calote suchte sich über Jini im gesamten lokalen Netz sämtliche Clients, was den gleichzeitigen Einsatz bei mehreren Entwicklern natürlich verhinderte. Zwar bot Calote eine Timer-Funktion zum automatischen Starten der Tests, aber diese wurde nicht verwendet, da der Zeitpunkt, an welchem die Testserver (nach dem automatischem Compilieren und den jUnit-Tests) bereit für einen Lasttest waren, nicht vorher als feste Zeit definiert werden konnte. (Im Gegensatz zur Verwendung eines Ant-Tasks, wo keine feste Zeitangabe dafür nötig ist, da der Test einfach als der nächste auszuführende Schritt nach den jUnit-Tests definiert werden kann.)

2.4 Anforderungen

Im Zuge dieser Studienarbeit sollte ein Test-Werkzeug entwickelt werden, welches sich mit dem bestehenden Calote integriert und zusätzlich zu den Lasttests auch Tests der Antworten des zu untersuchenden Servers auf Korrektheit erlaubt. Weiterhin war es natürlich wünschenswert, die bestehenden Test-Definitionen ohne größere Änderungen weiterzuverwenden und nur einen Satz an Test-Anfragen für den Lasttest und die neuen Tests zu pflegen. Ferner sollte das neue Werkzeug möglichst intuitiv zu bedienen sein und die Fehleranfälligkeit der bestehenden Art der Testsatz-Definition verringern. Eine Integration in den automatischen Build-Prozess und/oder den Lasttest wurde ausdrücklich gewünscht um den zusätzlichen Arbeitsaufwand für die Tests zu minimieren. Weiterhin sollte das Werkzeug nicht mehr nur zum Testen eines einzelnen spezialisierten Dienstes verwendbar sein.

2.5 Durchgeführte Änderungen

Um diesen Anforderungen zu genügen wurden im Rahmen dieser Studienarbeit folgende Änderungen durchgeführt:

Das bestehende Lasttest-Werkzeug Calote wurde in seinem XML-Parsing Code, welcher die Antworten des Servers interpretiert, dahingehend erweitert, dass die Antworten des zu testenden Servers gegen ein XML-Schema und/oder einen Schematron-Regelsatz getestet werden können. Hierbei wurde besonderer Wert darauf gelegt, dass das Zeit-Verhalten bei bestehenden Testsätzen sich nicht messbar ändert (obwohl u.a. ein anderer XML-Parser eingesetzt wird) und bestehende Testsätze, welche um die neuen Test-Arten maßvoll erweitert wurden, möglichst nahe am Zeitverhalten des ursprünglichen Tests liegen um mit bestehenden archivierten Antwortszeit-Werten für die Test-Subjekte vergleichbar zu bleiben.

Weiterhin wurden das Dateiformat und die internen Datenstrukturen und (Netzwerk-)Protokolle von Calote dahingehend erweitert, dass es mit minimalen Änderungen die neuen, optionalen Tests beschreiben kann. Für die Protokolle zwischen dem Calote-Master und seinen Workern konnte dies leider nicht kompaktibel gestaltet werden, um bestehende Worker mit einem neuen Master und umgekehrt zu benutzen. Dies stellt jedoch im Einsatz keine Schwierigkeit dar und war auch nicht gefordert.

Außerdem wurden die internen Mechanismen zur Verteilung der Tests auf die Worker (welche diese dann in einem verteilten Lasttest ausführen und an den sog. Master ihre Messwerte und Fehleranzahlen melden) erweitert um mehreren Nutzern die kollisionsfreie, gleichzeitige Verwendung von Calote zu ermöglichen, ohne das Zeitverhalten bestehender Tests zu verändern.

Zusätzlich wurde ein komplett neuer Testsatz-Editor geschrieben, der für reine Konformitäts-Tests (also keine verteilten Lasttests) als bequem zu bedienende Oberfläche dient und dem Entwickler sowohl bei der Pflege der Testsätze als auch bei der Einkreisung gefundener Fehler behilflich sein soll, da hierfür bisher keinerlei Unterstützung existierte.

Außerdem dient dessen Online-Hilfe zusätzlich als Einführung für neue Benutzer dieser Tests.

Weiterhin wurde ein ant-task geschrieben, mit dem sich solch ein Test auch automatisiert nach jedem automatischen deploy durchführen läßt.

Letzendlich wurde auch die nicht mehr aktuelle Dokumentation von Calote überarbeitet und auf den neuesten Stand gebracht.

3 Testverfahren

Um den Versuch des Beweises der Fehlerhaftigkeit des zu testenden Subjekts zu führen, wird nach jeder gesendeten Anfrage die Antwort gegen maximal die zwei folgenden Regelsätze getestet. Die Anfragen werden nicht getestet, da auch explizit fehlerhafte Anfragen im Laufe eines Tests gestellt werden sollen.

Der erste mögliche Regelsatz ist ein XML-Schema in Version 1.0, welches den Quasi-Standard für das Testen von XML-Dokumenten auf Konformität zu einem Regelsatz darstellt.

Die zweite Möglichkeit ist ein Schematron-Regelsatz. Dieser ist wesentlich einfacher zu erstellen und lehnt sich stark an die existierenden unit-tests an.

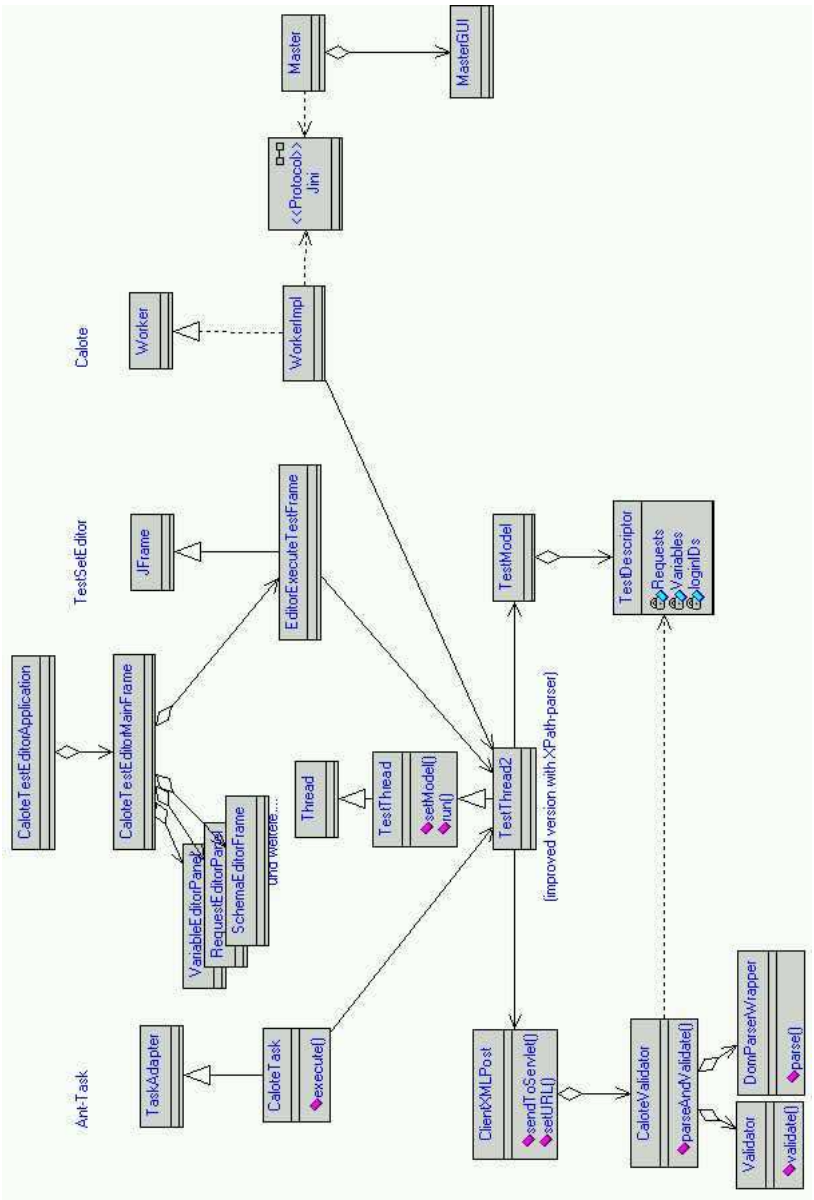

Abbildung 2: Aufbau der Programme in Form eines Klassendiagramms der wichtigsten Klassen

Beide Regelsätze können auch gleichzeitig in einer Anfragedefiniton auftauchen oder weggelassen werden. Im letzten Fall wird dann kein Test der Antwort auf eine Konformität der Antwort durchgeführt, womit das Datenformat kompatibel zu den existierenden Test-Definitionen bleibt.

3.1 Schema-Konformität

Das neue Werkzeug(bestehend aus einem erweiterten Calote, dem Testsatz-Editor und dem ant-Task) erlaubt es, die vom Server gelieferte Antwort gegen ein XML-Schema Version 1.0 [XML-Schema] auf Konformität zu diesem Schema zu testen, da sich dieses mittlerweile als Standard für XML-Dokumente etabliert hat. Hier wird erwartet, dass üblicherweise entweder das Schema für beliebige wohlgeformte Antworten des zu testenden Systems für eine Anfrage verwendet wird oder eine verschärfte Version dieses Schemas, welche lediglich die auf diese eine Anfrage erwarteten Antworten zulässt. Zur Überprüfung der Schema-Konformität wird der eingebaute XML-Schema-Parser aus dem Xerces-J Version-2 XML-Parser verwendet, da dieser aus gesammelten Erfahrungen früherer Projekte bereits einen ausreichenden Entwicklungsstand für ein Produktiv-System gezeigt hat, frei verwendbar ist und in den meisten anderen Projektes des Hauses als XML-Parser eingesetzt wird.

Hierfür war es nötig, den vorher in Calote verwendeten Sun-XML-Parser mit seinen zugehörigen DOM-Klassen und Document-Buildern auszutauschen. Um maximale Flexibilität zu erreichen, wurde, vor allem mit Hilfe der Patterns Faccade und Adaptor, ein Wrapper eingebaut, welcher den nachträglichen Austausch des XML-Parsers mit seinen zugehörigen Dokument-Klassen und Mechanismen erlaubt, indem alle Zugriffe, welche einen konkreten Klassen-Namen benötigen gegen eine zum Wrapper gehörige Klasse stattfinden, welche den (oder die) zugrunde liegenden Parser vor dem restlichen Programm versteckt. Sowohl Xerces als auch der Sun-Parser können so wahlweise verwendet werden (falls sich nachträglich Probleme wie z.B. ein anderes Zeit-Verhalten mit dem Xerces-Parser zeigen), wobei allerdings die Implementation mit dem Sun-Parser keinerlei XML-Schema überprüfen kann. So ist sichergestellt, dass im Falle von auftretenden Problemen die alten Lasttests ohne Schema-Überprüfungen weiterhin ausgeführt und gesammelte Antwortzeiten alter Versionen des Subjekts weiterhin mit aktuellen Versionen verglichen werden können.

Beispiel für ein solches XML-Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ele-
mentFormDefault="qualified">
  <xs:element name="AFClientResponse">
   <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="Service"/>
    </xs:sequence>
   <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="Comment" type="xs:string"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="AFFailure">
   <xs:complexType>
```

```
 <xs:attribute name="message" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
<xs:element name="Service">
 <xs:complexType>
  <xs:choice>
   <xs:element ref="AFList"/>
  <xs:element ref="AFSingle" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="AFTable"/>
   <xs:element ref="AFTree"/>
   <xs:element ref="AFSuccess"/>
   <xs:element ref="AFFailure"/>
  </xs:choice>
  <xs:attribute name="Access" type="xs:string"/>
 <xs:attribute name="Name" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
<xs:element name="Value" type="xs:string"/>
...
 </xs:schema>
```

Hier wird eine Antwort vom Typ AFClientResponse mit dem geforderten Attribut Name und dem optionalen Attribut Comment definiert. Enthalten kann diese Antwort 0-∞ Service-Tags, welche wiederum einen von 6 verschiedenen Tags enthalten dürfen.

Mit dem Schema wird versucht, die gesamte Sprache aller erlaubten Antworten zu definieren. Leider sind solche Schemata in der Praxis schwer vollständig und korrekt zu schreiben und ebenso schwer zu lesen.

3.2 Nachbedingungen

Da das Erstellen und Pflegen eines XML-Schemas mit seinen komplexen Datentypen recht kompliziert werden kann, wurde zusätzlich die Möglichkeit des Testens einzelner Eigenschaften der Antwort eingebaut. Nach einer Literatur-Recherche wurde aus den hierfür existierenden Systemen Schematron in der aktuellen Version 1.5 ausgewählt. Ein eigenes Regelsatz-System wurde nicht in Betracht gezogen, da Schematron bereits ausreichende Möglichkeiten bot und als vorgeschlagener ISO-Standard

"ISO/IEC 19757 - DSDL Document Schema Definition Language -Part 3: Rule-based validation - Schematron" (siehe [ISO])

eine eigene Insel-Lösung durch den geringeren Wartungs- und Schulungsaufwand als nicht zu rechtfertigenden Aufwand klassifizierte. Die Beschreibung eines Tests ist hinreichend intuitiv und bereits durch Tutorials ausreichend dokumentiert, um innerhalb weniger Minuten von einem mit XML und XPath vertrautem User verwendet zu werden.

Für eine bessere Integration in das neue Test-Werkzeug wurde der Versuch unternommen, die bestehenden Variablen des Calote-Systems als XPath-Variablen verwendbar zu machen, jedoch aufgrund der Komplexität und unzureichenden Dokumentation des Variablen-APIs von Xerces zugunsten späterer Versionen aufgegeben.

Aufbau eines Schematron-Regelsatzes: Ein Schematron-Regelsatz ist relativ einfach aufgebaut, weshalb er hier auch als Beschreibung für die Nachbedingungen gewählt wurde. Ein einfaches Beispiel sieht folgendermaßen aus:

```
<SCHEMA>
 <PATTERN NAME = "ANGESTELLTE">
  <RULE CONTEXT = "EMPLOYEE">
  <ASSERT TEST = "DOCUMENT('ANGESTELLTE.XML')//EMPLOYEE[@ID =
CURRENT()/@ID]">DER ZURÜCKGELIEFERTE ANGESTELLTE EXISTIERT NICHT!</ASSERT>
  </RULE>
  <RULE CONTEXT = "EMPLOYEE[WORK!='NONE']">
   <ASSERT TEST = "DOCUMENT('URLAUBER.XML')//EMPLOYEE[@ID =
CURRENT()/@ID]">DER ZURÜCKGELIEFERTE ANGESTELLTE IST MOMENTAN
IN URLAUB!</ASSERT>
  </RULE>
 </PATTERN>
...WEITERE PATTERN
</SCHEMA>
```

Dieses einfache Beispiel-Schema prüft alle <employee>-Elemente des aktuell zu testenden Dokumentes auf 2 Eigenschaften:

Jeder Angestellte muß mit seinem id-Attribut in der externen Angestelltenliste "angestellte.xml" vertreten sein.

Jeder der getesteten Angestellten mit einer Aufgabe darf nicht in der Liste der momentanen Urlauber "urlauber.xml" auftauchen.

current() bezieht sich dabei auf das momentan getestete Tag (Welches über context="employee" auf alle employee-Elemente des Dokumentes an beliebiger Position darin eingegrenzt wurde, wobei für den zweiten Test das work-Attribut nicht den Wert "none" haben darf(alle anderen werden nicht getestet).)

Pattern haben einen Namen, welcher in erzeugten Fehlermeldungen auftaucht. Jede Regel passt auf die unter "context" über einen XPath-Ausdruck (siehe [XPath]) angegebenen Elemente und diese werden nacheinander als Kontext für die in "test" angegebenen booleschen XPath-Ausdrücke verwendet. Ist einer der Ausdrücke falsch, so wird die entsprechende Fehlermeldung zusammen mit einer Positionsangabe, wo in der untersuchten Antwort der Fehler auftrat, gemeldet. Falls der Test direkt im Test-Editor ausgeführt wurde, wird dem User auch der den Fehler-Block umgebende Auszug aus der Antwort zur einfachen Einsicht präsentiert.

Die zusätzlich gebotenen Möglichkeiten von Phasen und Reports werden nicht genutzt, da sie für den gegebenen Einsatz-Zweck nicht notwendig sind, genauso wie die optionalen "subject" und "icon" -Attribute.

4 Datenformat

Im folgenden möchte ich das Datenformat von Calote erklären, da dies zum Verständnis der Arbeitsweise sowohl des Testsatz-Editors als auch der Erweiterungen in Calote selbst notwendig erscheint. Die minimal und zur Vorgängerversion kompaktibel gehaltenen Änderungen am bestehenden Format sind zur besseren Übersicht kursiv und in den Beispielen in Magenta hervorgehoben.

4.1 Anfragen

Das wichtigste Element einer Test-Beschreibung sind die Anfragen selber. Diese liegen in jeweils eigenen XML-Dokumenten vor und können so in mehreren Tests separat verwendet werden. Bis auf enthaltene Marken der Form “$$varname$”, welche jeweils durch den Wert von Variablen mit dem Namen “varname” ersetzt werden sollen, entsprechen sie einem direkten Mitschnitt der Kommunikation zwischen einem Client und einem Server, was die Wartung dieser Datensätze recht einfach hält.

4.2 Prozesse

Ein Prozess ist eine Serie von zusammenhängenden Anfragen. Ein Beispiel sieht folgendermaßen aus:

```
<?xml version="1.0" ?>
<!DOCTYPE process SYSTEM "process_description.dtd">
<process id="Admission" file="test_process_Admission.xml">
 <system namespace="one">
  <sysvar id="rnd_1" type="float" min="1.2" max="3.5" value=""/>
  <sysvar id="rnd_2" type="int" min="100000" max="999999" value=""/>
  <sysvar id="const_1" type="const" min="" max="" value="Vorname_Patient"/>
  <sysvar id="thread_id" type="t_id" min="" max="" value=""/>
 </system>
 <request id="req_1" file="/xml/List_Catalogs.xml" ruleset="/xml/List_CatalogsRuleset.xml"
schema="/xml/List_Catalogs.xsd" delay="8" tolerance="5"/>
 <request id="req_2" file="/xml/List_To_Do_Lists.xml" schema="/xml/ClientResponse.xsd"
delay="8" tolerance="5"/>
 <request id="req_3" file="/xml/User_Selected_To_Do_List_Items.xml"
schema="/xml/ClientResponse.xsd" delay="8" tolerance="5" url="/some/other/servlet?a=b">
  <variable replace="ToDoList" source=" Admission:req_2" xmlPath="ClientResponse.Service.AFList.It
global=""/>
 </request>
 <request id="req_4" file="/xml/User_Selected_Catalog_Item.xml" rule-
set="/xml/User_Selected_Catalog_ItemRuleset.xml" schema="/xml/ClientResponse.xsd"
delay="8" tolerance="5">
   <variable replace="SelectedCatalog" source=" Admission:req_1" xml-
Path="ClientResponse.Service.AFTree.Leaf(Name=Admission).DataTag" global=""/>
 </request>
 <request id="req_5" file="/xml/User_Selected_To_Do_List_Items.xml"
delay="8" tolerance="5" url="/some/other/servlet?abc=def">
  <variable replace="ToDoList" source="Admission:req_2" xmlPath="ClientResponse.Service.AFList.It
global=""/>
 </request>
</process>
```

Dieser Prozess definiert zunächst 4 Variablen. rnd_1 und rnd_2 werden mit zufälligen Werten aus dem angegebenen Wertebereich belegt, const_1 behält immer den Wert “Vorname_Patient” und thread_id wird auf die Java-Thread-Id des den Test ausführenden Java-Threads gesetzt.

Die erste Anfrage sendet nach 3 bis 13 Millisekunden (8±5) den in /xml/List_Catalogs.xml angegebenen Request ohne Variablenersetzungen vorher durchzuführen. Das Ergebniss wird durch das Schema in /xml/List_Catalogs.xsd und anschließend durch die Schematron-Regeln in /xml/List_CatalogsRuleset.xml validiert.

Nach wieder 3-13ms wird nun /xml/List_To_Do_Lists.xml ausgeführt aber nur gegen das Schema für generelle Antworten in /xml/ClientResponse.xsd geprüft, genauso wie die folgende Anfrage. Anfrage req_3 wird jedoch nicht an das gleiche Servlet wie alle anderen Anfragen gesendet, sondern an "/some/other/servlet?a=b" auf dem gleichen Host und Port. Req_4 hat nun wieder einen eigenen Schematron-Regelsatz für seine Antworten, benutzt aber als XML-Schema das generelle ClientResponse.xsd und ersetzt in der Anfrage alle Marker der Form $$SelectedCatalog$ durch den Wert XXX aus <ClientResponse>...<Service>...<AFTree>...<Leaf Name="Admission" DataTag="XXX"> in der Antwort auf die erste Anfrage dieses Prozesses.

Schließlich wird noch eine letzte Anfrage durchgeführt, in welcher das "DataTag"-Attribut des erstes Elementes eines AFList aus der Antwort des zweiten Requestes benutzt wird und die Anfrage an /some/other/servlet gestellt wird.

Jede Anfrage wird durch ein <request>-Tag definiert, "file" gibt dabei den relativen Pfad zur zu sendenden Anfrage an, welcher durch ein entsprechendes Präfix zu einer URL auf einem zentral gestarteten http-Server ergänzt wird, von dem sich die Worker(siehe Kapitel "verteilte Lasttests") die Anfrage holen. Die id-Attribute müssen eindeutig sein, um die Anfrage und ihre Antwort eindeutig referenzieren zu können. "delay" und "tolerance" geben jeweils eine zufällige Verzögerungszeit von delay±tolerance in Millisekunden an, welche vor Absenden der Anfrage abgewartet werden soll, um einen natürliche Benutzer zu simulieren.

Die neu eingefügten Attribute "ruleset" und "schema" geben in der gleichen Art wie "file" ein XML-Schema und/oder einen Schematron-Regelsatz an. Es wird erwartet, dass die im Beispiel zu sehende Vorgehensweise, für viele Anfragen einfach das generelle Schema einer gültigen Antwort, welche keine Fehlermeldung darstellt, einzusetzen, in der Praxis oft vorkommen wird.

Zu ersetzende Variablen werden mit den <variable>-Tags angegeben, nicht ersetzte Variablen in der Anfrage und zu ersetzende Variablen, für welche keine Marken in der Anfrage existieren, führen zu einem Fehler und dem Abbruch des Tests. "replace" gibt für eine Variable den zu ersetzenden Marker an, "global" einen optionalen globalen Namen, unter dem der Wert der Variablen im Weiteren (innerhalb wie außerhalb dieses Prozesses) referenziert werden kann.

Für "source" gibt es nun 3 verschiedene Werte, welche die Bedeutung des "xmlPath"-Tags bestimmen. Falls "source" den Wert "process:request" trägt, *so beschreibt "xmlPath" entweder den Wert eines (Attribut- oder Element-)Knotens* mittels eines sehr einfachen von Calote übernommenen XPath-ähnlichen Sprache oder (was nun neu hinzugekommen ist) *mittels eines wesentlich mächtigeren und standardisiertem XPath-Ausdruckes.* Zuerst wird versucht, den Ausdruck als XPath zu parsen, und im Falle eines Fehlers auf den alten Parser zurückgegriffen. Aufgrund der starken Ähnlichkeit der alten Sprache zu XPath werden keine Probleme bei der allgemeinen Umstellung auf den XPath-Standard für die Prozess-Beschreibungen erwartet.

Falls der Wert von "source" "system" ist, so enthält "xmlPath" einen Ausdruck aus (nicht-globalen)Variablen-Namen und dem Operator "+", welcher

als String-Verkettung ausgewertet wird. Zufälligen Variablen wird bei jeder Auswertung ein neuer Wert zugewiesen. Dies musste im Zuge der Rückwärts-Kompatibilität mit Calote beibehalten werden, obwohl das Gegenteil ein mächtigeres Konstrukt wäre.

Die dritte Möglichkeit besteht in dem Wert "global:varname" wobei "varname" den Name einer einzelnen globalen Variable darstellt, in diesem Fall wird "xmlPath" nicht ausgewertet.

Aufgrund von Namens-Konflikten in vielen existierenden Prozess-Beschreibungen war es nicht möglich, hier eine Vereinfachung zu treffen und schon im zweiten Fall auch globale Variablen-Namen zuzulassen.

Die Vereinfachung, für "xmlPath" nur einen XPath-Ausdruck zuzulassen und die hier definierten Variablen als Variablen in XPath zu definieren, scheiterte an der mangelhaften Dokumentation der Interna des XPath-Parsers des Apache Xerces XML-Parser (siehe [Xerces]).

Um andere als den bisherigen Server besser testen zu können, *wurde das optionale Attribut "url" für den file-Part einer URL eingefügt.* Hiermit lassen sich nun auch beispielsweise login und logout über ein zweites Servlet beschreiben oder mehrere interagierende Server im Zusammenspiel testen. Dieser Mechanismus soll das fest einkompilierte login (siehe Lasttests) der Vorgängerversion ablösen.

4.3 Variablen

Prozess-weite Variablen werden im <system>-Block in der Form <SYSVAR ID="RND_1" TYPE="FLOAT" MIN="1.2" MAX="3.5" VALUE=""/> definiert. *Das Attribut "namespace" wird nicht mehr verwendet und existiert lediglich zur Rückwärts-Kompatibilität.*

Für Variablen gibt es 9 Typen, welche in Tabelle 1 erläutert sind.

type	Beschreibung
int	"value" wird ignoriert und ein zufälliger Ganzzahl-Wert zwischen "min" und "max" zugeordnet
float	"value" wird ignoriert und ein zufälliger Fließkomma-Wert zwischen "min" und "max" zugeordnet
const	"min" und "max" werden ignoriert und der feste Zeichenketten-Wert "value" zugeordnet
timestamp	die Aktuelle Zeit in Millisekunden wird als Integer eingefügt
r_count	Der String "request_{i}" wird eingesetzt, wobei i die Anzahl der Anfragen im Prozess darstellt
count	beginnend mit "min" modulo "max" wird bei jeder Auswertung der Wert inkrementiert
t_id	alle Attribute werden ignoriert und die Java-Thread-ID des den Test ausführenden Treads eingefügt
worker_idx	Die ID des Worker-Prozesses. Erlaubt zusammen mit t_id eine eindeutige Identifikation des Prozesses.
date	Für "min" und "max" wird "rnd"=zufällig erlaubt und "value" stellt ein Pattern für ein Java SimpleDateFormat dar

Tabelle 1: zulässige Typen von Prozessvariablen und ihre Eigenschaften

4.4 Lasttests

Für Lasttests lassen sich mehrere Prozesse zu einem Test zusammenstellen. Hierbei werden von jedem Worker mehrere Listen aus hintereinander auszuführenden Prozessen abgearbeitet. Diese Listen haben das folgende Format:

```
<?xml version="1.0" ?>
<test id="LoadTest">
 <identities file="login_identities_10C.xml"/>
 <loop id="Load_Loop" times="20">
  <process id="Results" file="process1.xml"/>
  <process id="Results" file="process2.xml"/>
 </loop>
</test>
```

In diesem Beispiel wird mit zufälligen login-Identitäten aus login_identities_10C.xml zhintereinander 20 mal zuerst dr PRozess process1.xml und dann process2.xml ausgeführt.

Lasttests lassen sich nur mit Calote selber, nicht aber mit dem Testsatz-Editor durchführen.

Die Datei "login_identities_10C.xml" enthält in diesem Fall eine Reihe von Passwort, User-Name, Firma -Tripeln.

Ein zufälliges Element dieser Liste wird im Lasttest verwendet um sich vor dem Test anzumelden und danach abzumelden.

Login und Logout sind fest einkompiliert und müssen zur Rückwärts-Kompatibilität erhalten bleiben. Beide lassen sich

jeweils vor der Ausführung deaktivieren, in diesem Fall wird <identities...> ignoriert. Um andere als den einkompilierten Server zu testen, müssen login und logout als gewöhnliche Anfragen modelliert sein.

```
<identities id="LoadTest_10_clinics">
 <login user="worker1" password="pwd" company="ES0028"/>
 <login user="worker1" password="pwd" company="ES0025"/>
...
</identities>
```

4.5 globale Variablen

Globale Variablen werden für jeden Worker separat gespeichert. Sie werden mit der ersten Zuweisung eines Wertes über das "global"-Attribut des <variable>-Tags angelegt und können auf die gleiche Weise mit einem neuen Wert versehen werden. Das Referenzieren einer noch nicht angelegten globalen Variablen ist ein Fehler und führt zum Abbruch des Tests.

5 Der Testsatz-Editor

Der neu geschriebene Testsatz-Editor ist ein Werkzeug, welches für alle bis auf die verteilten Lasttests und die automatisch als ant-Task ausgeführten Tests die

bisherige Calote-Oberfläche ersetzen soll. Mit dem Editor lassen sich neue Test-Prozesse anlegen und alte bearbeiten sowie Tests ausführen. Das Bearbeiten von Lasttest-Beschreibungen ist aufgrund der trivialen Syntax dieser Dateien nicht vorgesehen.

5.1 Dialog-Modell

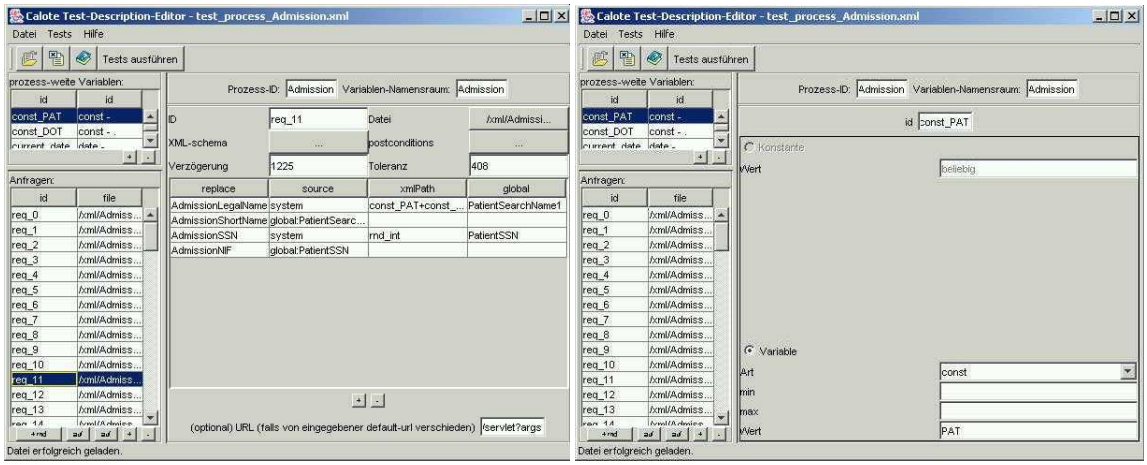

Abbildung 3: Hauptfenster des Testsatz-Editors bei Anderung einer Anfrage und einer Variablen

Der Editor (siehe Abbildung 3) zeigt dem User im Hauptfenster ständig die Anfragen und Prozess-weiten Variablen-Definitionen als Übersicht und erlaubt diese mit entsprechend der Auswahl wechselnden Panels im zentralen Bereich zu ändern.

Sollen Anfragen, Schemata oder Regelsätze geändert werden, steht hierfür ein eigener, in späteren Projekten weiterverwendbarer Editor (siehe Abbildung 4) bereit, welche entsprechend initialisiert geöffnet werden.

Bei diesen wird die bearbeitete XML-Datei ständig mit dem zugrunde liegenden Schema überprüft, um Fehler möglichst sofort dem User zeigen zu können.

Alle momentan gefundenen Fehler werden dem User unter dem Editor aufgelistet und im Quelltext hervorgehoben. Ein Click auf einen der Fehler führt in größeren Dateien sofort an die entsprechende Stelle im Quelltext, so denn dem Fehler zumindest eine Zeile, besser auch eine Spalte, im Quelltext zugeordnet werden kann. Zur besseren Übersicht wird mittels den bei der Schema-Überprüfung anfallender Sax-Events ein Syntax-Highlighting durchgeführt.

Das Ausführen-Fenster (zu sehen in Abbildung 5) startet intern alles, was an Jini-Infrastruktur nötig ist, um mit einem einzelnen lokalen Worker den Test-Prozess durchzuführen. Dabei kann die neu gestartete Instanz keinen im Netz oder auf dem lokalen Rechner ausgeführten Calote beeinflussen, steht aber auch nicht für andere Calote-Master zur Verfügung.

Die normalerweise auf dem Worker lokal geloggten Informationen werden mit entsprechender farblicher Hervorhebung während des Tests angezeigt. Ebenso öffnen sich nach dem Test zusätzliche Tabs, auf denen alle Antworten des Servers

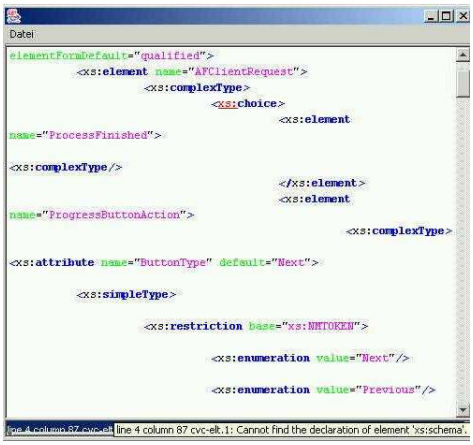

Abbildung 4: Der validierende XML-Editor für Anfragen, Regelsätze und Schemata

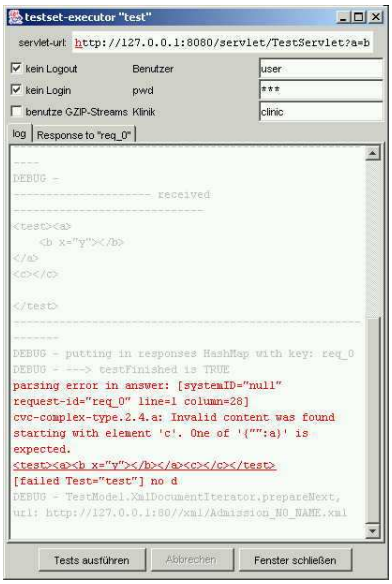

Abbildung 5: Ausführen eines Tests direkt im Testsatz-Editor

betrachtet werden können. Werden in den Ausgaben des Worker Fehlermeldungen erkannt, so wird versucht, das die Fehlerstelle umgebende, Stück XML-Code zu finden und unterhalb der Fehlermeldung zu präsentieren, damit der User nicht erst selber die entsprechende Stelle heraussuchen muss.

Sämtliche Text-Felder in allen Teilen des Editors erlauben eine Suche und die Standard-Operationen der Zwischenablage. Textfelder, welche XML-Daten zeigen, erlauben weiterhin die Selektion eines anderen Schemas, gegen welches das Dokument überprüft werden soll.

5.2 Datenmodell

Als Datenmodell wird intern direkt mit den DOM-Beschreibungen der XML-Dateien gearbeitet. Dies erlaubt ein immer konsistentes Datenmodell, da auch bei Änderungen am DOM-Modell vom Parser eingebundene Schemata und/oder DTDs überprüft werden. Die einzelnen Panels und Tabellen sind unabhängig voneinander und halten sich gegenseitig mittels des notify-Mechanismus der Standard Java object-Klasse über Änderungen der zugrunde liegenden Knoten auf dem Laufenden, da leider in den org.w3c.dom-Klassen keine PropertyListener vorgesehen sind und DOM-implementierungs-spezifischer Code im Sinne einer sauberen Abstrahierung vom eingesetzten Parser nicht in Frage kam.

Durch dieses Vorgehen soll sichergestellt werden, dass spätere Änderungen am Editor mit minimalem Aufwand möglich sind.

Mittels entsprechender Konfiguration des XML-Parsers und entsprechend wenig invasiver Vorgehensweise bei Änderungen am DOM bleiben manuelle Kommentare und Formatierungen in den bearbeiteten Dateien erhalten. Somit lassen sich auch manuell stark bearbeitete Dateien im Editor gefahrlos bearbeiten, was der Akzeptanz des Werkzeugs helfen soll.

5.3 Hilfestellungen

Der Schema-Editor ist unter Verwendung von PropertyRessourceBundles, welche trivial mit einem Text-Editor bearbeitet werden können, vollständig ins Deutsche und Englische übersetzt. Zum Hinzufügen weiterer Sprachen muss lediglich eine neue Datei mit Übersetzungen eingefügt werden (wobei hier der Editor vorzugsweise unicode-kompaktibel sein sollte). Fehlt eine einzelne Übersetzung, wird die übergeordnete Sprache (z.B. zu Deutsch-Schweizerisch, Deutsch) verwendet und, falls dies auch nicht erfolgreich ist, eine Warnung darüber, welche Übersetzung für welche Sprache nachzutragen ist, auf der Konsole ausgegeben und Englisch verwendet.

Weiterhin steht eine HTML-basierende online-Hilfe bereit (siehe 6), welche sowohl in die Benutzung einführt als auch das zugrunde-liegende Datenformat und die Interna für die Entwickler selbst erläutert.

6 verteilte Lasttests

Calote erlaubte es bereits, Lasttests verteilt von mehreren Rechnern aus gegen einen Server auszuführen. Allerdings musste dazu jedes Mal eine vollständige Infrastruktur gestartet werden und es war nur möglich, sämtliche Clients im lokalen Netz für den Lasttest zu verwenden. Da Tests jetzt aber von mehreren

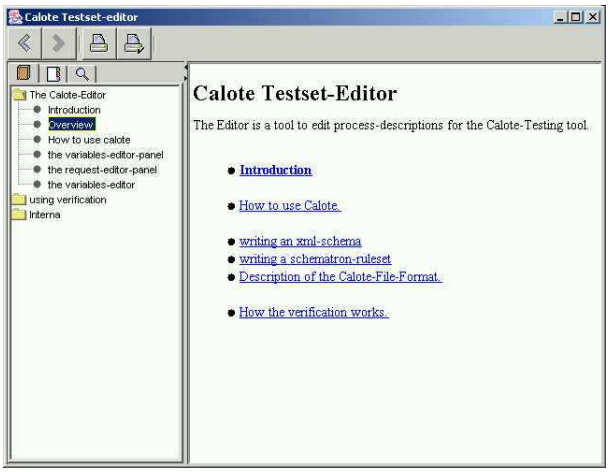

Abbildung 6: Die online-Hilfe des Testsatz-Editors

Entwicklern parallel und unabhängig voneinander durchgeführt werden sollten, war es nötig, hier Änderungen anzubringen. Zunächst möchte ich jedoch eine kurze Einführung in die Jini-Technologie geben, soweit sie zum Verständnis der Arbeitsweise notwendig ist.

6.1 Jini

Jini soll es erlauben, dass sich angebotene Dienste von Geräten in einem Netzwerk selbstständig gegenseitig finden, über ihre Eigenschaften befragen, verwenden und auch wieder ohne Probleme verlieren können (letzteres ist ein komplexeres Problem, als es den Anschein hat). Hierzu kann ein Programm in 2 möglichen Rollen gleichzeitig agieren:

Ein Dienst durchsucht das lokale Netzwerk (und voreingestellte weitere Adressen) nach Broadcasts eines speziellen Registrierungsdienstes und wird sich selbst bei diesem registrieren.

Ein Programm kann nun ebenfalls diesen Registrierungsdienst auf die gleiche Weise finden und über diesen Kenntnis von allen momentan registrierten Diensten erhalten.

Der Registrierungsdienst wird sich auch wie jeder anderen Dienst in einem Jini-Netz selber bei anderen Registraren eintragen, um gefunden zu werden.

Um nur die tatsächlich gesuchten Dienste zu finden, kann eine Suche eingeschränkt werden, in dem Attribute an einen Dienst angehängt werden (oft von einem Administrator ohne Zutun des Dienstes) und zur Suche des Registraren ein Regelsatz, welcher auf die zu findenden Dienste passen muss, übergeben wird.

Zur Strukturierung des Netzes, wie es hier benötigt wird, hingegen wird ein einfacherer Mechanismus in Jini bevorzugt. Hierbei hat jeder Dienst (auch die Registrare) und anfragende Client eine Reihe von Gruppen, in denen er Mitglied ist. (Standardmäßig ist ein Programm nur Mitglied in der public-Gruppe mit der leeren Zeichenkette als Namen). Nur Dienste, welche mindestens eine Gruppe mit dem anfragenden Programm gemeinsam haben, werden gefunden, einschließlich Registrierungs-Diensten.

Damit Dienste nicht ständig im Speicher gehalten werden müssen, wurde der rmid-Server eingeführt. Dieser erlaubt es, einen Dienst zu serialisieren und auf der Festplatte eingefroren zu lagern, bis auf vom Dienst vorgegebenen offenen Ports eine Anfrage für den Dienst kommt.

Da übertragene serialisierte Java-Klassen-Zustände als Nachrichten auch auf der Empfänger-Seite ihre jeweiligen Klassen benötigen, wird mindestens ein kleiner Webserver gestartet. Der Source-Path einer Klasse wird nun immer auf die Klasse auf dem Webserver zeigen, wo alle Worker auch Zugriff darauf haben.

Vorsicht! In Jini gibt es kein eigenes Konzept der Authentifizierung und Sicherung der übertragenen Daten!

Für einen genaueren Überblick über Jini und rmid siehe [Jini].

6.2 Architektur

Calote setzte 4 Programme ein um einen verteilten Lasttest durchzuführen.

Der Calote-Master dient als GUI für den Benutzer und sucht per Broadcast nach Registraren, auf denen sich Calote-Worker eingetragen haben.

Der Registrar reggie und der rmid stammen aus dem verwendeten JDK bzw. dem Jini-Developement-Kit von Sun [Jini].

Der Calote-Worker registriert sich bei allen lokalen Registraren und bietet an, die über Test-Deskriptoren spezifizierten Tests auszuführen.

Hierbei werden die eigentlichen Abfragen nicht im Deskriptor übermittelt, sondern als URL übergeben und der Worker holt sie sich von einem im Calote-Master enthaltenen Webserver. Die Antworten und Anfragen werden vom Client lokal per log4j aufgezeichnet, jedoch nicht dem Server übergeben. Dies würde zu Verzögerungen und damit verfälschten Antwortzeiten sowie einer nicht beherrschbaren Log-Flut führen.

Hier wurden nun zur gleichzeitigen Verwendung durch mehrere Entwickler Kommandozeilen-Optionen eingeführt um Master, Client und optional reggie (Sun Microsystem's Referenz-Implementierung eines Registrars) mit einer Liste von Gruppen zu starten. Hierbei wird angenommen, dass jeder Entwickler eine eigene Gruppe benutzt und gemeinsam verwendete Clients in einer Arbeitsgruppe zusätzlich auch einer gemeinsamen Gruppe angehören.

Weiterhin wurden die Deskriptoren angepasst, um auch die zusätzlichen Test-Beschreibungen zu beinhalten und vom eingebauten Http-Server im Master zentral zu holen.

Dem Client wurde die Möglichkeit gegeben, die im Speicher während des Tests aufbewahrten Antworten auf Anfrage zu liefern. Aufgrund der großen Menge an Anfragen und der Häufigkeit, mit der jeder einzelne Fehler in einem Lasttest auftreten wird, werden sie jedoch nur im Einzel-Test im Testsatz-Editor dem User präsentiert.

Großer Wert musste darauf gelegt werden, dass bisherige Anfragen ohne die neuen Tests weiterhin möglichst exakt die Antwortzeiten der alten Version messen, um vergleichbare Messwerte zur Bewertung der getesteten existierenden Server-Funktionen zu haben. Hierfür wird eine Validator-Klasse für das Validieren der Ergebnisse während des Parsens und das Hinzufügen der zusätzlichen Fehlermeldungen zur Fehler-Liste im Calote-Worker nur dann (vor Beginn der Zeitmessung) instantiiert, wenn auch Schema- oder Postcondition-Tests spezifiziert wurden. Die Fehlerliste wird erst nach Ende der Zeitmessung aktualisiert und die Schema und Schematron-Regelsätze werden vor Beginn der Zeitmessung

geladen und alles vorbereitet, um mit den zusätzlichen Tests möglichst nahe an den Zeit-werten ohne diese in der alten Version zu liegen. Das veränderte Parsen der Test-Beschreibungen mit XPath und den neuen Attributen sowie die Verwendung eines anderen Parsers scheinen keine Auswirkungen innerhalb der Messtoleranz zu zeigen.

7 Fazit

7.1 erreichte Funktionalität

"Im Zuge dieser Studienarbeit sollte ein Test-Werkzeug entwickelt werden, welches sich mit dem bestehenden Calote integriert und zusätzlich zu den Lasttests auch Tests der Antworten des zu testenden Servers auf Korrektheit erlaubt. Weiterhin war es wünschenswert, die bestehenden Test-Definitionen ohne größere Änderungen weiterzuverwenden und nur einen Satz an Test-Anfragen für den Lasttest und die neuen Tests zu pflegen. Ferner sollte das neue Werkzeug möglichst intuitiv zu bedienen sein und die Fehler-Anfälligkeit der bestehenden Art der Testsatz-Definition verringern. Eine Integration in den automatischen Build-Prozess und/oder den Lasttest wurde ausdrücklich gewünscht, um den zusätzlichen Arbeitsaufwand für die Tests zu minimieren. Weiterhin sollte das Werkzeug nicht mehr nur zum Testen eines einzelnen spezialisierten Dienstes verwendbar sein."

Die Schema- und Nachbedingungs- Tests wurden erfolgreich in Calote integriert, wobei das Datenformat nur minimal und in beiden Richtungen kompatibel zur Vorgängerversion erweitert wurde. Hierbei konnte auf standardisierte und weit verbreitete Formate zurückgegriffen und der Wartungsaufwand durch Verwendung von frei verwendbaren externen Bibliotheken minimiert werden.

Die bereits gemessenen Antwortzeiten bleiben unter Verwendung der gleichen Tests weiterhin mit der Vorgängerversion vergleichbar, was eine kontinuierliche, konsistente Dokumentation der bisherigen Test-Subjekte sicherstellt.

Die Ziele der Vermeidung einer doppelten Führung von Lasttest- und Funktionstest-Definitionen sowie der Weiter-Verwendung existierender Test-Definitionen wurde erreicht.

Das Erreichen des Zieles der intuitiven Bedienung wird erst der produktive Einsatz-Alltag des Werkzeugs zeigen, jedoch haben einzelne Tests durch mehrere Entwickler ohne Vorkenntnisse gezeigt, dass sich alle aufkommenden Fragen schnell über die online-Hilfe lösen ließen, die Einarbeitungszeit und Fehlerrate gegenüber dem bisherigen Werkzeug bei den gleichen Personen deutlich geringer ausfiel und der neue Testsatz-Editor vor dem alten textuellen Beschreiben der Tests bevorzugt wurde.

Die Integration in den Build-Prozess ist noch nicht im Einsatz, genügt jedoch den momentanen Anforderungen.

Die Generalisierung der Test-Vorgänge auf andersartige Server als das bisher einzige Calote-Subjekt ist grundsätzlich erreicht, es bedarf jedoch noch einer großflächigen Umstellung der Test-Definitionen, um den alten login-Mechanismus nicht länger zu verwenden, bevor dieser Punkt vervollständigt werden kann. Momentan besteht die Möglichkeit, andere Server zu testen, nur als Option und aus Kompatibilitätsgründen mussten sowohl der subjekt-spezifische login-Mechanismus als auch einige Calote-interne Spezial-Behandlungen für bestimm-

te Arten von Antworten des bisherigen Subjekts beibehalten werden.

7.2 Zukünftige Erweiterungen

Im Zuge einer weiteren Version halte ich eine generelle Überarbeitung des Datenformats für sinnvoll, was mit den Möglichkeiten von XPath als Alternative zum bisherigen eigenen Parser für Variablen-Wert-Zuordnungen bereits begonnen wurde. Hier steht das w3c-konforme Einblenden der globalen und Prozess-Variablen, eine Vereinfachung der Variablen-Deklarationen sowie die Möglichkeit, dynamisch außer Anfragen auch URLs als Ziel dieser Anfragen zu generieren im Vordergrund.

Eine weitere Integration in die automatischen unit-tests wäre denkbar, wobei jUnit-Tests die Möglichkeit erhalten, Anfragen zu stellen, deren Ergebnisse überprüft werden. Auf diese Weise könnte der jUnit-Test direkte Manipulationen an der Test-Datenbank durchführen und über einen Calote-Worker-Thread und die entwickelte Validator-Klasse die dadurch veränderten Antworten des Servers überprüfen.

Die Dokumentation sollte um eine Schritt-für-Schritt-Anleitung erweitert werden, welche einen minimalen Test aufbaut und dann in weiteren Schritten dem User zusätzliche Mittel um Variablen, Schemata und Nachbedingungen bekannt macht.

Die zusätzlichen visuellen Möglichkeiten des Executors im Testbeschreibungs-Editor sollten in geeigneter Form in Calote integriert werden, um auch bei Fehlern in Lasttests möglichst schnell das eigentliche Problem erfassen zu können.

Logs von Testläufen sollten zur Dokumentation von Problemen auch bei Funktionstests archivierenbar gemacht werden und zu einem späteren Zeitpunkt auch wieder zu laden sein.

8 Literatur

[jUnit]: Erich Gamma, Kent Beck http://www.junit.org

[Jini]: Sun Microsystems http://www.jini.org und http://java.sun.com/products/jini (2004-01-14)

[XML-Schema]: W3C 2001 http://www.w3.org/XML/Schema (2004-01-14)

[Xerces] XML-Parser: Apache-Group http://xml.apache.org/xerces2-j/index.html

[Xalan] XSLT-Transformator: Apache-Group http://xml.apache.org/xalan2-j/index.html

[Ant] build-tool: http://xml.apache.org (2004-01-14)

[Sun-XML-Parser]: Sun-Microsystems http://java.sun.com/xml/ (2004-01-14)

[ISO]: International Organization for Standardization http://www.iso.org (2004-01-14)

[Schematron]: 1.5: Rick Jelliffe. 2003. http://www.ascc.net/xml/resource/schematron/schematron.html (2004-01-14)

[XPath]: James Clark und Steve DeRose(Inso Corp. und Brown University) für das W3C, 1999 "XML Path Language (XPath) Version 1.0" http://www.w3.org/TR/xpath (2004-01-14)

[SOAP]: W3C XML Protocol Working Group, 2003 “Simple Object Access Protocol (SOAP) 1.2” http://www.w3.org/TR/SOAP/ (2004-01-14)