Studienarbeit

jETI: Redesign der ETI-Plattform
Eine Client/Server-Architektur für entfernte Ausführung dateibasierter Tools

4. Februar 2009

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5 für Programmiersysteme

Gutachter

Prof. Dr. Bernhard Steffen
Dipl. Inform. Christian Kubczak

Autor

Stefan Naujokat

Inhaltsverzeichnis

1 Einleitung

Seit 1997 existiert die *Electronic Tool Integration Platform*[10] (ETI). Dabei handelt es sich um eine Experimentierplattform für Tools[1], die entfernt ausgeführt werden können. Im Vordergrund steht hier der einfache Zugang zu einer Vielzahl von Diensten, die man problemlos verwenden kann, statt sich jeweils in eine komplizierte Programmbedienung einarbeiten zu müssen. Weiterhin sollte eine Community aufgebaut werden, um möglichst viele Tools vielen Menschen zur Verfügung stellen zu können. Aufgrund des für Software-Verhältnisse fortgeschrittenen Alters sind die technischen Standards und Programmierpraktiken überholt, sodass eine vollständige Neuimplementierung des Systems gerechtfertigt war. Gegenstand der vorliegenden Arbeit ist die Motivation und Protokollierung dieser Neuimplementierung.

Aufgrund der Plattformunabhängigkeit ist Java[12] als zu verwendene Programmiersprache ausgewählt worden. Zudem wird die zusätzlich geforderte Integration in das *jABC*[13] dadurch erleichtert. Daraus ergibt sich auch mit *jETI-Platform* der Name für das neue System.

Im Folgenden werden die Probleme der alten Plattform kurz erläutert, gefolgt von einer daraus resultierenden Liste an Anforderungen. Die danach folgenden Abschnitte beschreiben tiefergehend die Konzepte und Ideen der Einzelkomponenten, aus denen das Gesamtkonzept *jETI* besteht.

Wegen der guten Visualisierbarkeit und der unkompliziertern Kommandozeilenbefehle verwenden die hier gezeigten Beispiele das Programm `convert` aus der ImageMagick-Familie[4]. Dennoch sei vorab erwähnt, dass der Funktionsumfang der jETI-Platform über das einfache Drehen und Manipulieren von Bildern hinausgeht (siehe zum Beispiel [5] und [3]).

2 Anforderungsanalyse

Dieser Abschnitt beschreibt zunächst die ETI-Plattform, indem wichtige Konzepte erläutert werden. Mit diesen Konzepten gehen allerdings einige Einschränkungen einher, die in dem zweiten Teil näher erläutert werden. Gleichzeitig werden damit auch die Ideen und Konzepte für die jETI-Plattform motiviert.

[1] Aktuell wäre der Begriff *Service* angemessener. Dennoch wird im Folgenden die Bezeichnung *Tool* verwendet, wenn es um ETI-Services geht, da dies der ursprünglichen Benennung des Projekts entspricht.

2.1 Bestandsaufnahme

Wie bereits erwähnt, stand bei ETI im Vordergrund, eine Online-Plattform zum schnellen Ausprobieren von Tools zur Verfügung zu stellen. Je nach Erfahrungsgrad des Anwenders sollte die Granularität der Spezifikation anpassbar sein. So sollten weniger erfahrene Anwender die Möglichkeit bekommen, ihre Anforderung *lose* zu spezifizieren. Die konkrete Ausführungssequenz wurde dann mittels Synthese[10] vervollständigt. Erfahrene Anwender hingegen konnten direkt die Sequenz vollständig spezifizieren.

Ein ETI-Tool besteht aus dem Tripel `t = (inputType, toolname, outputType)`. Eine ETI-Sequenz ist eine Hintereinanderausführung von kompatiblen Tools, bei denen der `outputType` von $\texttt{t}_i$ dem `inputType` von $\texttt{t}_{i+1}$ entsprechen muss. Die Eingabe des ersten sowie die Ausgabe des letzten Tools bilden hierbei jeweils die Ein- und Ausgabe der gesamten Sequenz.

Weiterhin wird unter Umständen ein erhöhter Integrationsaufwand in Kauf genommen, um die Verwendung und Ausführung einfach zu halten. Ausführungsprozesse sind daher in der Koordinationssprache *HLL* (*High Level Language*) formuliert, die für den User allerdings transparent ist. Der große Aufwand beim Integrieren von neuen Tools entsteht dadurch, dass Adapter in HLL geschrieben werden müssen, welche die Toolausführung anstoßen und gleichzeitig die Typenkonvertierungen vorher und nachher implementieren[1].

2.2 Erweiterungen

Aus den Eigenschaften des ETI-Projekts ergeben sich die Anforderungen an die neue Plattform. Im Vordergrund hat hierbei die Entwicklung einer möglichst flexiblen Umgebung gestanden, die es ermöglichen soll, eine große Auswahl von Tools entfernt auszuführen. Besonderer Fokus ist dabei auf solche Tools gelegt worden, die per Kommandozeile aufrufbar sind und auf Dateien arbeiten, da in diesem Punkt häufig verbreitete Techniken wie Web Services[17], RMI[11] oder CORBA[9] keine unmittelbare Unterstützung liefern.

Die folgenden Abschnitte beschreiben die wichtigsten Änderungen von jETI gegenüber der alten Plattform. Dabei werden jene Aspekte behandelt, die die essenziellen Kritikpunkte an ETI darstellen.

Der gesamte Themenbereich der Synthese wird nicht an dieser Stelle, sondern in weiteren Arbeiten behandelt.

2.2.1 Integrationskonzept

Die komplizierte Integration von Tools, die zudem nur an einer Stelle – nämlich bei den Betreibern von ETI – durchgeführt werden konnte, sorgte dafür, dass nur sehr wenige Tools in ETI integriert wurden und Updates auf neue Versionen dieser Tools selten eingepflegt werden konnten. Daher ist eine der Kernideen für die Neuentwicklung gewesen, diesen Integrationsprozess zu vereinfachen sowie auf verschiedene Rollen zu verteilen.

Ergebnis dieser Verteilungsidee ist die Rolle des *Tool Providers*. Dieser betreibt einen *Toolserver*, für welchen er die Aufgabe übernimmt, die Tools zu integrieren. Als Experte für „sein Tool“ fällt für ihn der Aufwand aufgrund nicht benötigter Einarbeitung in das Tool geringer aus. Dies setzt allerdings voraus, dass der Integrationsprozess für den Toolexperten, der nicht zwangsläufig Programmierkenntnisse besitzen muss, vereinfacht wird. Daraus ist die Idee entstanden, Tools mittels eines *Tool Descriptors* in XML zu beschreiben, dessen Verwendung nicht voraussetzt, dass man programmieren kann. Um den Prozess darüber hinaus zu vereinfachen, sollte es auch einen HTML-GUI-Editor (*Tool Configurator*) für diese Beschreibung geben. Wegen der Existenz mehrerer Tool Provider ist ferner ein System zur verteilten Ausführung auf unterschiedlichen Toolservern benötigt worden.

Der ehemalige *ETI Core* muss nun nur noch eine Verwaltungsrolle für die vielen Tool Provider übernehmen. Dieser Aufwand ist für eine Rolle sehr viel tragbarer als die vollständige Integration aller Tools, die für jETI zur Verfügung gestellt werden sollen. Gleichzeitig kann es viele Toolserver geben, die jeweils von unterschiedlichen Personen gewartet werden.

2.2.2 Parametrisierbarkeit von Tools

Aus der Tripel-Definition ergibt sich für ETI-Tools die Einschränkung, dass sie genau eine Datei als Eingabe erwarten sowie genau eine Datei als Ausgabe erzeugen. Eine Parametrisierung der Tools oder variableres Input/Output-Verhalten kann allenfalls schlecht simuliert werden[2]. Hier soll das neue System deutlich flexibler einzusetzen sein.

Darüber hinaus schien es sinnvoll zu fordern, dass Parameter selbst näher spezifiziert werden können. So soll es möglich sein, dass ein Parameter optional ist und gegebenenfalls mit einem Standardwert belegt werden kann. Eine vollständige Erläuterung der Parametertypen findet sich in Abschnitt 5.1 auf Seite 8.

[2] Wenn zum Beispiel ein Parameter mit drei möglichen Werten existierte, konnte man daraus drei verschiedene Tools mit festen Werten erzeugen. Das hier beispielhaft verwendete Rotate-Tool würde dabei allerdings in 359 unterschiedlichen Tools resultieren, vorausgesetzt die Drehwinkel sind auf ganze Zahlen beschränkt.

2.2.3 Anbindung an das jABC

Das ETI-System war zum Teil in das *Application Building Center* (ABC) integriert. Da dieses nicht mehr weiterentwickelt wird und auch hiervon eine auf Java basierende Neuentwicklung existiert, sollte auch jETI in das jABC einbindbar sein. Dieses Plugin ermöglicht einem Nutzer, jETI-Tools transparent in ausführbaren Graphen zu verwenden. Es basiert im Wesentlichen auf der Diplomarbeit von Marc Njoku[8], wurde aber darüber hinaus im Rahmen dieser Arbeit noch erweitert.

3 Das jETI-Kommunikationsprotokoll

Das Kommunikationsprotokoll bildet die Grundlage für alle Aufrufe, die an einem Toolserver erfolgen. Es ist grundsätzlich unabhängig von bestimmten Plattformen. Im Rahmen dieser Arbeit sind allerdings ausschließlich auf Java basierende Lösungen entstanden. Dabei baut die Standardimplementierung auf Web Services auf und verwendet dazu das Axis-Framework[15] (Version 1.4).

3.1 Store, Exec, Retrieve

Beim Design des Kommunikationsprotokolls für jETI-Tool-Aufrufe wurde zunächst folgende dreischrittige Befehlsfolge als Standardweg festgesetzt:

- Senden der Eingabedatei(en) an den Server (*store*)
- Aufrufen des Tools (*exec*)
- Herunterladen der Ausgabedatei(en) vom Server (*retrieve*)

Dabei sollte es auch möglich sein, mehrere Tools direkt hintereinander auszuführen, wobei zum Beispiel die Ausgabe des ersten Tools als Eingabe für das zweite Tool dient, ohne dieses Zwischenergebnis zum Client zu übertragen und direkt darauf wieder unverändert zum Server hochzuladen.

Zu diesem Zweck verwaltet der Server für jede Session ein virtuelles Filesystem, auf welchem der Client beliebig (mittels *store*-Befehl) schreiben und (mittels *retrieve*) lesen darf, und welches als Basis für Fileparameter in den Tools verwendet wird. Das Starten einer Session geschieht implizit beim ersten Aufruf von *store*. Allerdings ist geplant, dies zukünftig noch durch eine separate Methode zu ersetzen, die dann zum Beispiel auch Benutzerauthentifizierung mittels eines Passworts ermöglicht.

Die Entscheidung, welche Dateien wann übertragen werden, wird dem Benutzer des Clients – und damit hier dem jETI-Plugin für das jABC – überlassen. Dadurch kann ein möglichst effizienter Datentransfer gewährleistet werden, insbeson-

dere dann, wenn mehrere Tools auf demselben Server ausgeführt werden. Die Methoden zur effizienten Datenübertragung werden im entsprechenden Abschnitt des Plugin-Kapitels näher erläutert.

Um den Datentransfer weiterhin zu optimieren, wurde im Laufe der Entwicklung das Protokoll noch um den Befehl *forward* ergänzt. Mit diesem ist es möglich, Dateien zwischen zwei Toolservern direkt zu übertragen, ohne dass diese den Umweg über den Client machen müssen. Dies ist zum Beispiel dann nötig, wenn die beiden oben genannten Tools nicht bei demselben Toolprovider liegen, aber dennoch das erste Tool die Eingabe für das zweite Tool liefert. Es wird lediglich vorausgesetzt, dass der Client bereits auf beiden Servern eine gültige Session geöffnet hat.

3.2 Parameter von Tools

Die Toolaufrufe können durch Parameter angepasst werden. Diese Parameter haben immer einen pro Tool eindeutigen Namen sowie einen Wert, der in der Regel vom Client gesendet wird. In beiden Fällen handelt es sich um Objekte vom Typ `String`. Um weitere Javaklassen als Parameter zu ermöglichen, wird daher vorausgesetzt, dass diese mittels `toString()` serialisierbar sind und einen Konstruktor besitzen, welcher als einzigen Parameter einen String entgegennimmt. Bei den Standarddatentypen (`Long`, `Double`, `Integer`, `String`, ...) ist dies der Fall. Der Typ, in den der übergebene `String` deserialisiert werden muss, ergibt sich aus der Definition in der `tools.xml`. Sind diese Voraussetzungen für einen gewünschten Parametertyp nicht erfüllt, so muss er mittels eines geeigneten Wrappers an diese Konvention angepasst werden.

3.3 Die Client-API

Die Client-API, welche die oben erläuterten Anforderungen erfüllt, wird durch das Interface `EtiConnection` (siehe Anhang A auf Seite 20) festgelegt. Eine konkrete Implementierung ist abhängig von dem jeweiligen Protokoll, auf dem das jETI-Protokoll aufsetzt.

Derzeit existieren zwei Implementierungen dieses Protokolls. Dabei handelt es sich zum einen um eine auf SOAP basierende Web-Service-Implementierung, zum anderen um eine auf simplen TCP-Streams basierende Lightweight-Implementierung. Mittels letzterer wurde unter anderem ermöglicht, den jETI-Client sogar auf Java-ME-fähigen Multifunktionsdruckern (MFPs) auszuführen, obwohl diese in ihren Möglichkeiten auf sehr grundlegende Features beschränkt sind[3].

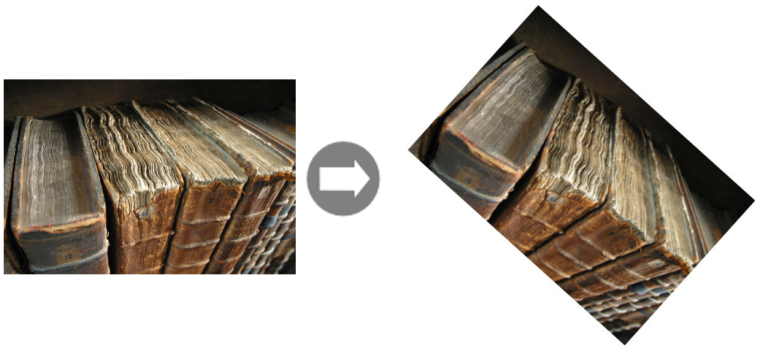

Abbildung 1: Drehen des Beispielbildes um 42°

3.4 Beispiel: Bildmanipulation

Das hier vorgestellte Beispiel verwendet einen einfachen Dienst, mit dem Bilder manipuliert (darunter u.a. auch gedreht) werden können (siehe Abbildung 1[3]). Hierzu wird die Datei an den Toolserver gesendet. Dort wird das jETI-Tool `convert` mit dem Parameter `ROTATEANGLE=42` aufgerufen, worauf das veränderte Bild wieder vom Server heruntergeladen wird. Der hierfür zu verwendene Javacode befindet sich in Anhang B auf Seite 23. An dieser Stelle wird bereits die Sinnhaftigkeit eines jABC-Plugins deutlich, da immer noch ein Programmierer vonnöten wäre, um die jETI-Tools zu verwenden.

4 Serverarchitektur

Der Toolserver beinhaltet verschiedene Einzelkomponenten, die in den folgenden Abschnitten näher erläutert werden. Dieser Abschnitt soll daher einen ersten Eindruck von dem allgemeinen Aufbau erläutern. Abbildung 2 auf der nächsten Seite verdeutlicht, wie die einzelnen Komponenten zueinander in Beziehung stehen.

Die Ausführung wird vom Tool Executor zur Verfügung gestellt. Dieser kann aber nicht direkt von außen angesprochen werden, sondern nur über die Connectoren, die gemeinsam mit der dazugehörigen Client-Klasse das zuvor vorgestellte Kommunikationsprotokoll implementieren.

Die Informationen über ausführbare Tools wiederum entnimmt der Executor dem Tool Descriptor. Diese Datei kann entweder direkt mit einem beliebigen Texteditor oder mit dem Tool Configurator, dessen Verwendung keine genauen Kenntnisse über das verwendete Format voraussetzt, bearbeitet werden.

[3]Das verwendete Bild unterliegt der GFDL (siehe [16])

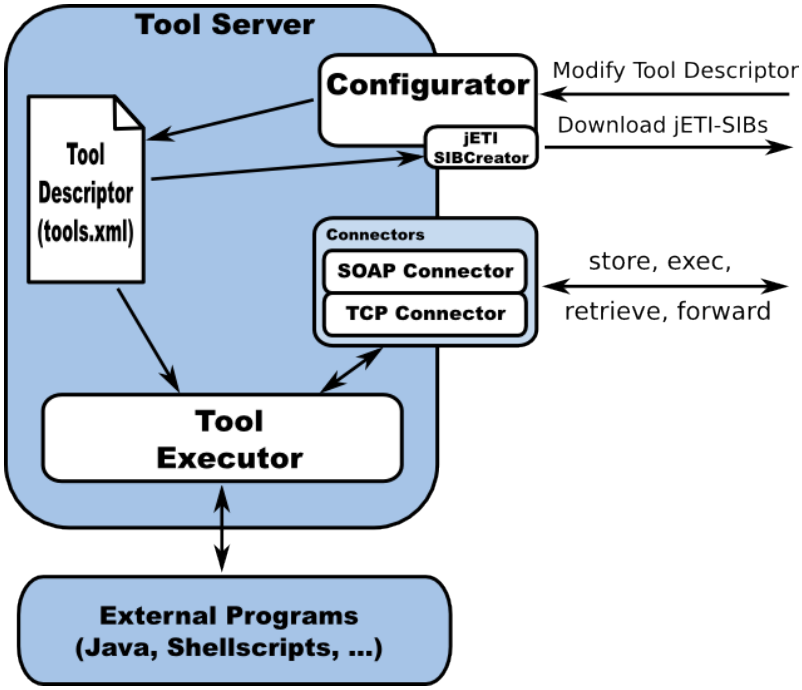

Abbildung 2: Architektur des jETI Toolservers

5 Tool Executor

Der Tool Executor bildet das Kernstück des Toolservers. Er stellt eine in hohem Maße anpassbare Ausführungsschicht für nahezu alle Tools zur Verfügung, sodass diese über das zuvor beschriebene Protokoll ausgeführt werden können. Die Konfiguration der zur Verfügung gestellten Tools wird mittels einer speziellen XML-Datei, dem Tool Descriptor, realisiert[4].

Einer der Grundgedanken des Toolserver ist der, dass beliebige Methoden aus beliebigen Javaklassen als jETI-Tool zur Verfügung gestellt werden können. Der Name der Klasse, wie auch der der Methode, werden in der `tools.xml` angegeben. Bei der Ausführung sucht der Tool Executor dann nach der Klasse sowie der passenden Methode. Hierbei ergibt sich die Signatur, mit der nach der Methode gesucht wird, teilweise aus der Toolbeschreibung, zum Teil aber auch aus den vom Client gesendeten Parametern (vergleiche hierzu das Beispiel in Abschnitt „Der Weg einer Ausführung“ auf Seite 11).

[4]Nachfolgend wird der Tool Descriptor zur Vereinfachung teilweise auch als *tools.xml* benannt.

Parametertyp	`required`	`default`	`value`	`name`
Statischer Parameter	-	-	•	-
Verpflichtender Parameter	•	-	-	•
Optionaler P. mit Defaultwert	-	•	-	•
Optionaler Parameter	-	-	-	•

Tabelle 1: Die vier Parametertypen

5.1 Tool Descriptor

Die `tools.xml` enthält sämtliche Informationen, die sowohl Server als auch Client benötigen, um ein Tool auszuführen.

Auf oberster Ebene muss es ein `<etitoolserver>`-Tag geben, welches mehrere `<tool></tool>`-Blöcke enthalten darf. In `<etitoolserver>` gibt es ein verpflichtendes Attribut namens `serverURI`, welches die Adresse des Servers angibt. Innerhalb dieser URI ist auch das zu verwendende Protokoll kodiert, damit der Client weiß, wo er mit welchem Protokoll den Server erreichen kann.

Ein `<tool>`-Element beschreibt genau eins der Tools, die vom Toolserver zur Verfügung gestellt werden sollen. Es kennt vier Attribute:

`name` Ein über alle Tools des Servers eindeutiger Name, unter dem das Tool nach außen identifiziert werden kann.

`class` Der Name der Klasse, aus der die Methode ausgeführt werden soll, inklusive vollständigem Paketpfad.

`method` Der Name der auszuführenden Methode.

`active` Gibt an, ob das Tool aktuell verfügbar ist. Wird vom Client ein nicht verfügbares Tool aufzurufen versucht, resultiert dies in derselben Fehlermeldung, die auch erzeugt wird, wenn ein überhaupt nicht vorhandenes Tool aufgerufen wird. Mögliche Werte sind `true` und `false`.

Der Hauptbestandteil einer Tooldeklaration besteht hingegen aus den Parametern, die in gewissem Maße gruppiert und geschachtelt werden können, um damit diverse Verhaltensweisen zu erzeugen.

Grundsätzlich werden vier verschiedene Parametertypen unterschieden, die innerhalb des `<parameter>`-Elements mit den Attributen `required`, `default`, `value` und `name` modelliert werden. Die Belegung der Attribute kann Tabelle 1 entnommen werden.

Statischer Parameter

Dieser Parameter spezifizert einen Wert, der nicht vom Client zu beeinflussen ist und statische Eingaben an die Methode ermöglicht. Es handelt sich also um einen Parameter in der aufgerufenen Methode des Tools, nicht aber um einen Parameter für das jETI-Tool. Da dieser Parameter nicht für den Client bestimmt ist, hat er auch keinen Namen, unter dem er identifiziert werden muss.

Verpflichtender Parameter

Ein verpflichtender Parameter muss vom Client beim Aufruf des Tools mitgesendet werden. Ist dies nicht der Fall, wird das Tool nicht ausgeführt und eine entsprechende Fehlermeldung erzeugt.

Optionaler Parameter mit Defaultwert

Dieser Parameter kann vom Client gesendet werden. Wenn der Client den Parameter nicht mitsendet, wird der Executor den Parameter mit dem Defaultwert instanziieren.

Optionaler Parameter

Wenn dieser Parameter vom Client nicht gesendet wird, dann wird er vom Server auch nicht instanziiert. Dies hat unter Umständen Auswirkungen auf die Signatur, mit der die auszuführende Methode gesucht wird.

5.1.1 Komplexe Parameter

Zur Konstruktion von komplexeren Parametersignaturen stehen dem Tool Provider zwei sogenannte *Parameter-Container*, *Array* und *Union*, zur Verfügung, deren Verwendung und Funktionsweise im Folgenden erläutert werden.

Array

In Anlehnung an das übliche Verständnis von Feldern in Programmiersprachen kann hier eine sortierte Menge unbekannter Größe aus Objekten desselben Typs spezifiziert werden. Damit wird zum Beispiel ermöglicht, dass die Klasse RuntimeUnix beliebig lange Kommandozeilen ausführen kann, da es nur ein Array in seiner Signatur hat. Das Element `<array>` besitzt ein Attribut `class`, welches den Typ des Arrays angibt. Beim Erstellen der `tools.xml` muss der Toolprovider selbst auf die Kompatibilität der Typen achten. Es dürfen also nur Objekte in ein Array eingefügt werden, die in der Typhierarchie unterhalb des spezifizierten Typs des Arrays liegen.

Zur Ausführungszeit des Tools wird das Array dann mit dem angegebenen Typ instanziiert und die Parameter werden der Reihe nach eingefügt. Dabei wird ein

nicht verwendeter Parameter einfach nicht in das Array eingefügt. Im Gegensatz zu einem nicht verwendeten Parameter auf oberster Ebene beeinflusst dieser innerhalb eines Arrays nicht die Suche nach einer passenden auszuführenden Methode.

Eine Schachtelung von Arrays ist derzeit nicht möglich. Ein Array kann also immer nur auf höchster Ebene der Parameterdeklarationen vorkommen.

Union

Mit dem Container Union ist es möglich, Konstrukte zu spezifizieren, die entweder vollständig oder gar nicht verwendet werden.

Dabei gilt die Regel: „Sind nicht alle Parameter, die in der Union als required gesetzt sind, vom Client gesendet, so wird die gesamte Union nicht verwendet."

Aus diesem Grund sind die Parameter, die innerhalb einer Union als „required" festgelegt sind, auf Tool-Ebene eigentlich optional. Der Unterschied zum direkt auf Tool-Ebene definierten optionalen Parameter ist aber die Abhängigkeit der in der Union zusammengefassten Parameter untereinander. Wenn eine Union zum Beispiel zwei „required" Parameter enthält, und der Client schickt nur einen davon, dann wird die gesamte Union nicht verwendet. Es ist also der gleiche Effekt, als hätte der Client diesen einen Parameter auch nicht gesendet.

5.1.2 Datei-Parameter

Die jETI-Plattform ist primär für auf Dateien arbeitende Tools konzipiert worden. Diese benötigen in der Regel absolute Dateinamen, da sie meist nicht in demselben Arbeitsverzeichnis wie der Toolserver operieren. Der Client kennt aber nur zum Sessionverzeichnis relative Dateinamen, die er als Wert für den Parameter sendet. Für diesen Fall gibt es daher einen besonderen Parametertypen. Dieser wird vom Benutzer (auf Client-Seite) mit relativem Pfadnamen benutzt, soll aber beim Aufruf von `toString()` den absoluten Pfad zu der Datei liefern. Damit dieser bekannt ist, setzt der Toolserver beim Instanziieren dieses Parameters den Session-Pfad zusätzlich zu dem über den Konstruktor bekanntgemachten relativen Dateinamen. Dies geschieht allgemein darüber, dass der Toolserver bei allen Parametern, die das Interface `FileReference` erweitern, die Methode `addPathPrefix(String)` mit dem Session-Verzeichnis aufruft. Der Aufruf von `toString()` auf diesen Objekten sollte dann den vollständigen Pfad zu der Datei liefern, was bei den Standardimplementierungen `InputFileReference` und `OutputFileReference` der Fall ist.

5.2 Der Weg einer Ausführung

Dieser Abschnitt beschreibt anhand des vorgestellten Bildmanipulationsbeispiels, welche Vorgänge im Tool Executor ablaufen, die letztendlich zur Ausführung des Kommandozeilenbefehls führen.

Aus Sicht des Clients besteht das Beispiel aus folgenden drei Einzelschritten:

1. Bild `books.jpg` wird mit `store` auf dem Server gespeichert.
2. Führe Tool `convert` mit folgenden Parametern aus (vgl. Tool Descriptor in Anhang C auf Seite 24):
   - INFILE = books.jpg
   - ROTATEANGLE = 42
   - OUTFILE = rotated_books.jpg
3. Bild `rotated_books.jpg` wird mit `retrieve` vom Server heruntergeladen.

Im Folgenden werden die in Schritt 2 stattfindenden Einzelschritte aufgezeigt und weitergehend erläutert:

- Das Tool `convert` wird aus dem Tool Descriptor ausgelesen.
- Die in der `tools.xml` angegebene Klasse (`RuntimeUnix`) wird mit Reflection instanziiert.
- Die Parameter werden instanziiert:
  - Array vom Typ `Object[]` wird gemäß Toolbeschreibung generiert. in dieses Array werden `-solarize` und `SOLARIZEVALUE` nicht aufgenommen, da sie in einer Union sind und `SOLARIZEVALUE` nicht vom Client gesendet wurde.
  - Auf allen Parametern, die das Interface `FileReference` implementieren, wird `addPathPrefix` mit dem aktuellen Sessionverzeichnis aufgerufen. In diesem Beispiel handelt sich sich dabei um die Objekte zu den Parametern `INFILE` und `OUTFILE`.
- Es wird nach einer Methode in `RuntimeUnix` gesucht, die zu den instanziierten Objekten passt und den Namen trägt, wie sie im Tool Descriptor konfiguriert ist. Da hier auf oberster Ebene ein `Object`-Array definiert ist, wird die Methode `exec(Object[])` gefunden.
- Die gefundene Methode wird mit den zuvor instanziierten Parametern aufgerufen.
- Innerhalb von RuntimeUnix#exec wird nun auf jedem Element des übergebenen Arrays `toString()` aufgerufen und damit der auszuführende Kommandozeilenbefehl zusammengefügt. Zu beachten ist hierbei, dass INFILE zwar mit

”books.jpg” instanziiert wurde, aber beim Aufruf von `toString()` ”/home/user/jeti/toolserver/var/sessions/423823/books.jpg”[5] liefert.

- Der letztendlich ausgeführte Kommandozeilenbefehl lautet also nun:
```
/usr/bin/convert -rotate 42 \
/home/user/jeti/toolserver/var/sessions/423823/books.jpg \
/home/user/jeti/toolserver/var/sessions/423823/rotated_books.jpg
```

Nach diesem Aufruf befindet sich die Ausgabedatei im Session-Verzeichnis und kann mittels `retrieve` und dem relativen Namen `rotated_books.jpg` heruntergeladen werden.

6 Der Tool Configurator

Mit dem Tool Configurator wird auf Seite des Toolservers eine grafische Oberfläche zur Verfügung gestellt, die es dem Tool Integrator ermöglicht, die vom Toolserver bereitgestellten Tools grafisch zu konfigurieren, ohne die `tools.xml` manuell editieren zu müssen.

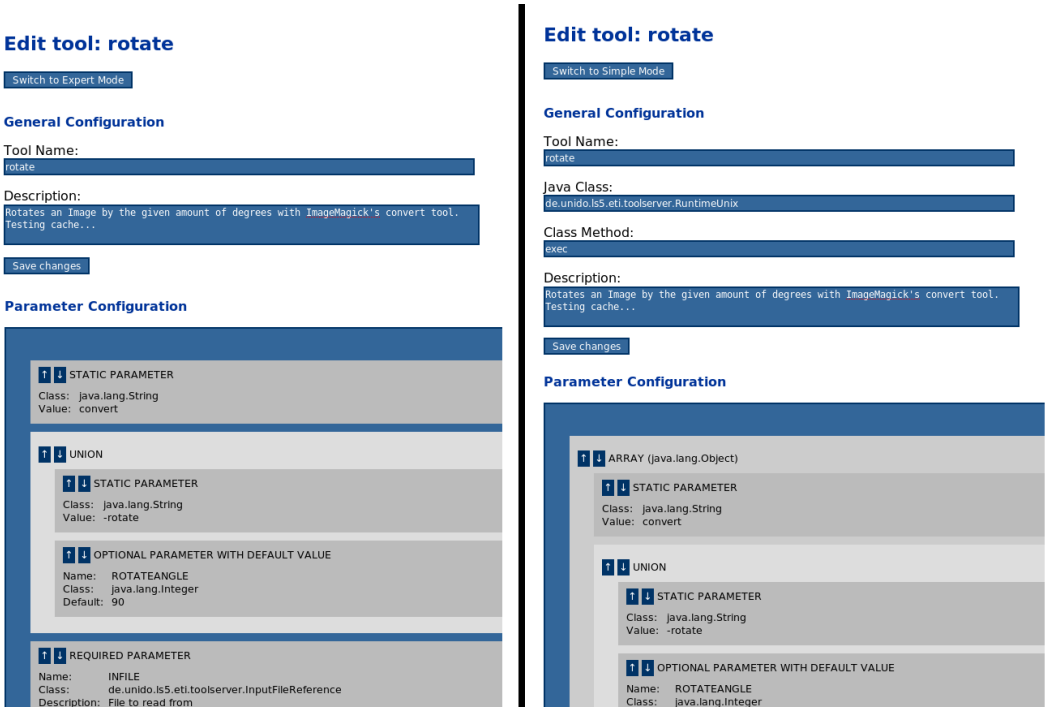

Abbildung 3: Vergleich von *Simple Mode* (links) und *Expert Mode* (rechts).

[5]Das Verzeichnis des Toolservers sowie die Session ID seien hier beispielhaft gesetzt.

Es werden zwei Eingabemodi unterschieden: Im *Expert Mode* bildet der Tool Configurator genau den Tool Descriptor ab, dient also nur als Editierhilfe, die von der konkreten Syntax der resultierenden `tools.xml` abstrahiert. Der *Simple Mode* hingegen zeigt eine Ansicht, mit der Kommandozeilenprogramme, die ausschließlich Standardparameter (`InputFileReference`, `OutputFileReference`, `String`, `Integer`, `Double`) verwenden, vereinfacht bearbeiten werden können. Hier wird keine Auswahl über die Klasse und Methode zugelassen, sondern automatisch `RuntimeUnix#exec` gesetzt, und die Existenz des Array-Parameters auf oberster Ebene versteckt (vgl. Abbildung 3 auf der vorherigen Seite). Darüber hinaus müssen die Parameter nicht mit ihren Klassennamen angegeben werden, sondern können in einer Drop-Down-Box ausgewählt werden.

Mit dem *Simple Mode* des Tool Configurators kann also direkt ein Großteil der Zielgruppentools ohne besondere technische Kenntnisse in das jETI-System integriert werden.

7 Das jABC-Plugin

Das Plugin ermöglicht einem Benutzer des jABC, Ausführungsgraphen mit jETI-SIBs zu modellieren. Abbildung 4 auf der nächsten Seite zeigt, wie das hier verwendete Beispiel als modellierter jABC-Graph aussieht, der gleichzeitig das Ursprungsbild sowie das Ergebnis anzeigt.

Von dem Plugin wird ein eigener Datei-Kontext verwaltet, in dem symbolische Namen verwendet werden können, unabhängig davon, ob die Datei gerade auf einem Toolserver oder lokal liegt. Weiterhin existieren grundlegend zwei Typen von jETI-SIBs: *local* und *remote*. Gemeinsam haben sie, dass sie auf dem genannten Filecontext arbeiten. Diese beiden SIB-Typen werden im Folgenden noch näher beschrieben.

7.1 Remote jETI SIBs

Damit Aufrufe von Tools auf jETI-Toolservern bei der Ausführung eines jABC-Graphen erfolgen können, werden die Remote-SIBs benötigt. Diese SIBs stellen im Prinzip eine Parametersammlung dar, mit denen der Tracer[2] ermitteln kann, auf welchem Server das Tool liegt, und mit welchen Parametern es aufgerufen werden muss.

Trifft der Tracer bei seinem Durchlauf auf ein SIB, welches das Interface `ETI` implementiert, übergibt er die Ausführungskontrolle an die Klasse `ETIPluginInterfaceDelegate`. Diese fragt vom SIB die benötigten Informationen für den entfernten Toolaufruf ab und führt diesen aus.

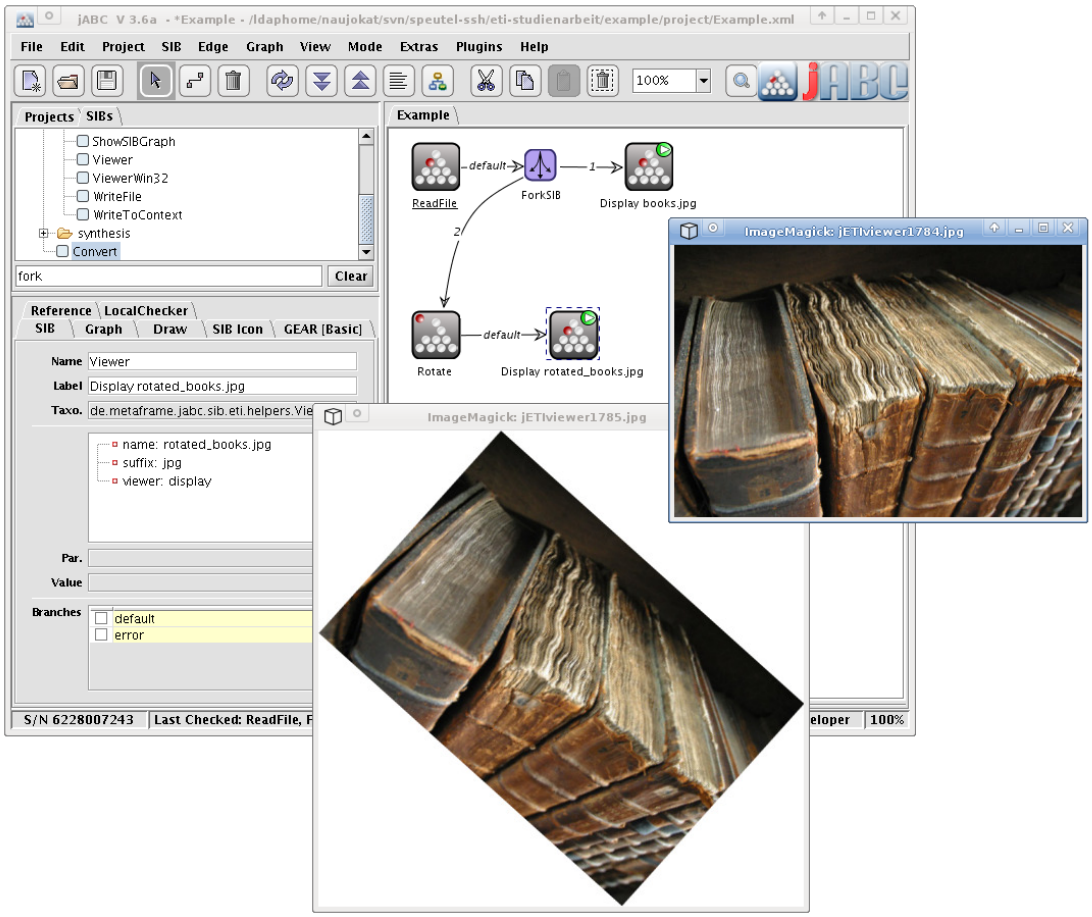

Abbildung 4: Beispiel-Graph nach der Ausführung

Aufgrund aktuell noch vorhandener Einschränkungen seitens des jABC gibt es keine Möglichkeit, optionale Parameter direkt in den jETI-SIBs abzubilden. Es existieren nur zwei Ansätze für Workarounds, die allerdings für jETI nicht implementiert sind. Eine Möglichkeit wäre, für jeden optionalen Parameter zwei SIBs zu erzeugen, nämlich einmal das, welches den Parameter besitzt, und das, welches ihn nicht besitzt. Wenn allerdings für ein Tool mehrere optionale Parameter definiert sind, wächst die Anzahl der daraus resultierenden jETI-SIBs bei diesem Ansatz exponentiell in Anzahl dieser Parameter. Bei der zweiten Möglichkeit wird für jeden optionalen Parameter ein weiterer Parameter vom Typ `Boolean` dem SIB hinzugefügt, welcher angibt, ob der Parameter gesendet werden soll. Diese Lösung ist der erstegenannten deutlich vorzuziehen. Dennoch handelt es sich wegen der Realisierung mittels Namenskonvention um eine umsaubere Umgehung der Konzepte des jABC. Schwieriger wird es weiterhin bei der Umsetzung des Union-Konzeptes. Dieses könnte zum Beispiel mit dem zweiten Ansatz in Kombination mit speziellem Localchecker-Code[6]

realisiert werden, der eine Warnung ausgibt, wenn Union-Parameter nur teilweise gesetzt sind.

7.2 TransferHandler

Der `TransferHandler` ist verantwortlich für die Verwaltung des *jETI File Context*. Dieser Kontext stellt für lokale und entfernte jETI-Tools ein virtuelles Filesystem dar, in welchem der User Dateien referenzieren kann. Hierbei ist transparent, auf welchem Rechner sich diese Datei tatsächlich befindet. Der TransferHandler verwaltet zu jedem virtuellen Dateinamen die URI des Servers, auf dem die jeweils aktuellste Version der Datei liegt. Dabei wird für lokal vorhandene Dateien die feste URI `virtual://local` verwendet. Wird nun eine Datei auf Toolserver A als InputFile verwendet, die letzte aktualisierte Version liegt aber auf Toolserver B, dann wird sie mittels `forward` an A weitergeleitet; Bei lokalen Dateien würde entsprechend `store` verwendet.

Durch Auswählen des Debug-Modus im jETI-Plugin wird der TransferHandler veranlasst, die Map, in der zu den jeweiligen virtuellen Dateinamen die URIs gespeichert werden, in den ExecutionContext des Tracers zu legen. Dieser kann mit dem Debugger des Tracers angezeigt werden. Somit lässt sich das Lazy-Transfer-Verhalten des TransferHandlers nachverfolgen.

7.3 Helper-SIBs

Mit der Palette der jETI-Helper-SIBs liefert das Plugin eine Reihe von lokal arbeitenden SIBs, die häufig für das Erstellen von jETI-Ausführungsgraphen benötigt werden.

Dazu gehören unter anderem:

- Das Einlesen von Dateien vom Filesystem in den jETI-Kontext (`ReadFile`).
- Das Schreiben von Dateien vom jETI-Kontext auf das Filesystem (`WriteFile`).
- Das Einlesen von Dateien aus dem Tracer-Kontext in den ETI-Kontext (`ReadFromContext`).
- Das Schreiben von Dateien aus dem ETI-Kontext in den Tracer-Kontext (`WriteToContext`).
- Das Einlesen von Dateien von einer Internetseite (`ReadFromURL`).
- Einen generischen File-Viewer, der genutzt werden kann, um Dateien, die im jETI-Kontext liegen, anzuzeigen. Dies geschieht, indem das für die Ansicht zu

verwendene Programm als SIB Parameter gesetzt wird (zum Beispiel `display` oder `acroread`) (`Viewer`).

- Ein SIB zur Anzeige von ETI-Exceptions, die vom Plugin im Fehlerfall automatisch in den ExecutionContext geschrieben werden. Dieses SIB kann als Errorkantensenke verwendet werden.

Darüber hinaus sind noch einige weitere lokale Hilfs-SIBs entstanden, welche jeweils einen sehr speziellen Zweck erfüllen (zum Beispiel Spezialvarianten des Viewers). Sie werden an dieser Stelle nicht näher erläutert.

8 Der jETI-Sibcreator

Der jETI-Sibcreator[6] bildet das Brückenstück zwischen Tool Descriptor und Plugin. Mit ihm ist es möglich, automatisch aus der `tools.xml` die dazugehörigen jETI-SIBs zu generieren, die dann direkt im jABC verwendet werden können.

Im Wesentlichen handelt es sich bei dem jETI-Sibcreator um ein Velocity-Template[14] für ein jETI-SIB, welches mit Parametern gefüllt wird, die aus der `tools.xml` entnommen werden. Stellt ein Toolserver mehrere Tools zur Verfügung, werden die dafür erstellten SIBs gemeinsam zu einem JAR gepackt, welches als SIB-Pfad-Eintrag im jABC eingebunden werden kann.

In der aktuellen Implementierung ist der jETI-Sibcreator über den Tool Configurator eingebunden. Hier kann ein Toolserver-Betreiber die generierten SIBs direkt als JAR herunterladen. Zukünftig ist aber vorgesehen, dass der jETI-Sibcreator in den Component Server, die geplante zentrale Verwaltungsinstanz, eingebunden werden soll.

9 Fazit

Mit der jETI-Platform ist ein flexibles und einfach zu verwendendes System zur entfernten Ausführung von Tools innerhalb des jABC entstanden. Allerdings gehen mit dieser Einfachheit zum Teil gewisse Einschränkungen einher. So ist zum Beispiel die Serialisierung mit `toString()` nicht so mächtig wie das generische Marshalling/Unmarshalling-Konzept von Web Services, funktioniert dafür aber zuverlässig und ohne großen Overhead. Weiterhin kam von Tool-Integratoren häufig die Frage: „Was ist, wenn mein Tool sein Ergebnis nicht per Datei ausgibt?“ Da die hier vorgestellte Architektur primär für dateibasierte Tools ausgerichtet ist, ist die

[6] Der jETI-Sibcrator ist nicht zu verwechseln mit dem im jABC verfügbaren SIBCreator-Plugin[7], welches Coderahmen für SIBs während der Arbeit mit dem jABC erstellt.

beschriebene Problematik als Sonderfall zu werten, dessen Umsetzung zwar möglich, aber nicht unbedingt intuitiv ist. Der Grundgedanke hierbei ist das Kapseln solcher Funktionen, zum Beispiel mittels Javaklasse oder per Shell-Script. Dies ist sicherlich eine Einschränkung der Flexibilität, aber durch diese Design-Entscheidung ist das Einbinden von Kommandozeilentools sehr einfach. Da diese hauptsächlich von jETI unterstützt werden sollten, erschien diese Einschränkung während der Planung des Systems als akzeptabel. Mittlerweile existieren auch Ansätze, diese Problematik ohne Wrapper-Klassen/Skripte zu lösen. Diese sind aber zur Zeit nicht implementiert.

Eine weitere aktuell nicht abgedeckte Gruppe von Tools sind jene, die Benutzerinteraktion zur Ausführungszeit benötigen. Dies umfasst Kommandozeilenprogramme, die Eingaben erfordern, aber auch Programme mit grafischer Benutzeroberfläche. Ein Ansatz zur Einbindbarkeit dieser Tools besteht aus der Aufnahme von Maus- und Tastatureingaben, die dann als Skript oder Makro das Tool steuern.

Darüber hinaus sind noch einige Änderungen der Architektur geplant, die bereits teilweise von unterschiedlichen Personen umgesetzt wurden. Dazu gehört eine Authentifizierung mit LDAP, die Implementierung des *Component Server* als zentrale Verwaltungseinheit, kontrollierte Ausführung mittels Zertifikaten, asynchrone Ausführung, der Dateiaustausch mit WebDAV, aber auch z.B. die im alten System vorhandene Synthese von Sequenzen mittels temporallogischer Beschreibungen.

Diese Features hier noch im Detail zu beschreiben würde zum einen mehr als die von mir geleistete Arbeit protokollieren, zum anderen aber auch den Umfang dieses Dokuments sprengen.

Festzuhalten ist, dass die jETI-Plattform keinesfalls ein *fertiges* System ist, sondern einem permanenten, lebendigen Weiterentwicklungsprozess unterliegt.

Literatur

[1] Volker Braun, Tiziana Margaria, and Carsten Weise. Integrating Tools in the ETI Platform. *STTT*, 1(1-2):31–48, 1997.

[2] Markus Doedt. JavaABC Tracer-Plugin. `http://jabc.cs.uni-dortmund.de/opencms/en/plugins/tracer.html`.

[3] Benjamin Hanzelmann. Studienarbeit: Embedded System Workflow Execution with jABC and jAWS. 2008.

[4] ImageMagick Studio LLC. ImageMagick Homepage. `http://www.imagemagick.org`.

[5] Anna-Lena Lamprecht, Tiziana Margaria, and Bernhard Steffen. Seven Variations of an Alignment Workflow - An Illustration of Agile Process Design and Management in Bio-jETI. In Ion I. Mandoiu, Raj Sunderraman, and Alexander Zelikovsky, editors, *ISBRA*, volume 4983 of *Lecture Notes in Computer Science*, pages 445–456. Springer, 2008.

[6] Johannes Neubauer. JavaABC Localchecker-Plugin. `http://jabc.cs.uni-dortmund.de/opencms/en/plugins/local_checker.html`.

[7] Johannes Neubauer. JavaABC SIBCreator-Plugin. `http://jabc.cs.uni-dortmund.de/opencms/en/plugins/SIBCreator.html`.

[8] Marc Njoku. Diplomarbeit: Entwicklung einer Systemerweiterung der jABC-Entwicklungsplattform zur Ansteuerung verteilter Toolsequenzen. 2005.

[9] Object Management Group. Common Object Request Broker Architecture. `http://www.corba.org/`.

[10] Bernhard Steffen, Tiziana Margaria, and Volker Braun. The Electronic Tool Integration Platform: Concepts and Design. *STTT*, 1(1-2):9–30, 1997.

[11] Sun Microsystems. Remote Method Invocation. `http://java.sun.com/javase/technologies/core/basic/rmi/`.

[12] Sun Microsystems. Sun Developer Network (SDN). `http://java.sun.com/`.

[13] Sven Jörges, Christian Kubczak, Ralf Nagel, Tiziana Margaria, and Bernhard Steffen. Model-Driven Development with the jABC. In *HVC - IBM Haifa Verification Conference*, LNCS 4383, Haifa, Israel, October 23-26 2006. IBM, Springer Verlag.

[14] The Apache Software Foundation. The Apache Velocity Project. `http://velocity.apache.org/`.

[15] The Apache Software Foundation. WebServices - Axis. `http://ws.apache.org/axis/`.

[16] Tom Murphy VII. Old book bindings. http://commons.wikimedia.org/w/index.php?title=Image:Old_book_bindings.jpg&oldid=11570279, 2005. [Online; accessed 14-July-2008].

[17] W3C Working Group. Web Service Architecture. http://www.w3.org/TR/ws-arch/.

Anhang

A. `EtiConnection` Interface API

Method Summary

- *endSession*
  `public void` **endSession**`( )`
  - **Usage**
    - End session at jETI server and delete all stored files
  - **Exceptions**
    - `EtiRemoteException` - if the Error was generated by the jETI Server
    - `EtiLocalException` - if the error occured on client-side

- *exec*
  `public void` **exec**`( java.lang.String` **tool**`, java.util.Map` **parameters** `)`
  - **Usage**
    - Executes tool at the server with the given parameters. Note that non-String parameters must be appropriately serialized into Strings.
  - **Parameters**
    - `tool` - the name of the tool to execute
    - `parameters` - mapping from parameter name to value
  - **Exceptions**
    - `EtiLocalException` - if something goes wrong on client side.
    - `EtiRemoteException` - if something goes wrong on server side.

- *forward*
  `public void` **forward**`( java.util.List` **filenames**`,`
  `de.unido.ls5.eti.client.EtiConnection` **toServer** `)`
  - **Usage**
    - Forwards a List of virtual files from the server represented by this connection to the server represented by the toServer connection.
  - **Parameters**
    - `filenames` - list of virtual filenames that shall be forwarded.
    - `toServer` - An existing connection to the server where the files shall be forwarded to.
  - **Exceptions**

        * `EtiLocalException` -
        * `EtiRemoteException` -

- *getServerURI*
  `public URI` **getServerURI**`( )`
  - **Usage**
    * Retrieve the server's URI of this connection
  - **Returns** - the server's URI

- *getSession*
  `public String` **getSession**`( )`
  - **Usage**
    * Retrieve the sessionId of this connection. This should usually onyl be needed for debugging purposes.
  - **Returns** - the session ID at the tool server

- *retrieve*
  `public List` **retrieve**`( java.util.List` **filenames** `)`
  - **Usage**
    * Retrieve the requested files from the server.
  - **Parameters**
    * `filenames` - List of virtual filenames to retrieve.
  - **Returns** - the retrieved files
  - **Exceptions**
    * `EtiLocalException` - if something goes wrong on client side.
    * `EtiRemoteException` - if something goes wrong on server side.

- *setSession*
  `public void` **setSession**`( java.lang.String` **sessionId** `)`
  - **Usage**
    * Sets the session ID of this connection. Method is needed to create a new Connection Object for an already existing Server Session, currently used by EtiServer's forward function
  - **Parameters**
    * `sessionId` - the session Id to set.

- *store*
  `public void` **store**`( java.util.List` **files** `)`

- **Usage**
  - Stores the given List of files to the server
- **Parameters**
  - `files` - the list of files to store
- **Exceptions**
  - `EtiLocalException` - if something goes wrong on client side.
  - `EtiRemoteException` - if something goes wrong on server side.

B. Beispielcode für Clientbenutzung

```
import java.io.*;
import java.net.*;
import java.util.*;

import org.apache.commons.io.IOUtils;

import de.unido.ls5.eti.client.*;

public class Example {

  public static void main(String[] args) {
    try {
      // Create a connection object to the server at localhost
      URI serverURI = new URI("http://localhost:8080/services/ETI");
      EtiConnection myEti =
        EtiConnectionFactory.createConnection(serverURI);

      // Store input files to the server
      List<VirtualFile> inputFiles = new ArrayList<VirtualFile>(1);
      inputFiles.add(
        new FileVirtualFile(new File("books.jpg"), "books.jpg"));
      myEti.store(inputFiles);

      // execute 'convert' with given parameters on the uploaded image
      Map<String, String> parameters = new HashMap<String, String>();
      parameters.put("ROTATEANGLE", "42");
      parameters.put("INFILE", "books.jpg");
      parameters.put("OUTFILE", "rotated_books.jpg");
      myEti.exec("convert", parameters);

      // retrieve the rotated image from the server
      List<String> outputFileNames = new ArrayList<String>(1);
      outputFileNames.add("rotated_books.jpg");
      List<VirtualFile> retrievedFiles = myEti.retrieve(outputFileNames);

      // Write the retrieved image back to a file
      IOUtils.copy(retrievedFiles.get(0).getInputStream(),
          new FileOutputStream(new File("rotated_books.jpg")));

    }
    catch (Exception e) {
      e.printStackTrace();
    }

  }
}
```

C. Beispiel-Tool-Descriptor (tools.xml)

```
<etitoolserver serverURI='http://localhost:8080/services/ETI'>
  <tool name='convert' active='true'
    class='de.unido.ls5.eti.toolserver.RuntimeUnix' method='exec'>
    <description>
      Perform a selection of image conversions on a given
      file using the program 'convert' from the ImageMagick toolset.
    </description>
    <array class='java.lang.Object'>
      <parameter class='java.lang.String' value='convert' />
      <union>
        <parameter class='java.lang.String' value='-rotate' />
        <parameter name='ROTATEANGLE' class='java.lang.Integer'
          required='true' contextExpression='true'
          description='Rotate the image by the given amount of degrees.' />
      </union>
      <union>
        <parameter class='java.lang.String' value='-solarize' />
        <parameter name='SOLARIZEVALUE' class='java.lang.String'
          required='true' contextExpression='true'
          description='Negate the image by replacing every pixel above threshold
          level with its complementary color.' />
      </union>
      <parameter name='INFILE' required='true'
        class='de.unido.ls5.eti.toolserver.InputFileReference'
        description='The image file that is converted' />
      <parameter name='OUTFILE' required='true'
        class='de.unido.ls5.eti.toolserver.OutputFileReference'
        description='The file where the converted image is stored to.' />
    </array>
  </tool>
</etitoolserver>
```