Design und Implementierung ultraleichter Java Clients

Alexander Schmidt
(A.Schmidt@xcc.de)

Thomas Weinstein
(Th.Weinstein@xcc.de)

Xcc Software AG
Bahnhofplatz 8
76137 Karlsruhe
Tel. 0721 93276 – 0
info@xcc.de
http://www.xcc.de

Kurzfassung. Um clientseitiges Java ist es in letzter Zeit ziemlich ruhig geworden. Hauptgrund dafür ist der Ressourcenhunger herkömmlicher Ansätze in Bezug auf Prozessorzeit und Speicherbedarf, außerdem die nicht unerheblichen Probleme bei der Verwendung von Java im Browser. Im nachfolgenden Artikel beschreiben wir deshalb unsere Erfahrungen mit dem Design und der Implementierung ultraleichter Java Clients. Durch einen zweigeteilten Ansatz unter Verwendung einer nur minimal von AWT abhängigen Widgetbibliothek und einer XML basierten, deklarativen Beschreibungssprache bleibt die Größe des eigentlichen Applets unabhängig von der Komplexität der Oberfläche bei konstant ca. 250KB. Das sehr redundante XML lässt sich hervorragend komprimieren und fügt der gesamten vom Server zum Client zu übertragenden Datenmenge kaum etwas hinzu. Die Clients können wahlweise als Java Applikation oder als Applet im Browser betrieben werden. Für die Applets wird dabei nur ein beliebiger Java-fähiger (JDK 1.1.x) Browser vorausgesetzt. Das Ergebnis sind sehr kleine und performante Java Clients, ohne den Overhead von Swing oder AWT, die auch in ressourcenarmen Umgebungen im Produktionsbereich verwendet werden können, aber bei Look & Feel keine Abstriche machen müssen.

1 Einführung

Was vor einigen Jahren als Hype begonnen hat, steht inzwischen kurz davor völlig aus dem Blickfeld des Mainstreams zu geraten. Die Rede ist von Java Applets im Browser. Beim ersten Auftauchen dieser Technologie reagierte die Entwicklergemeinde sehr euphorisch. War es doch erstmals möglich, statische Webinhalte durch dynamische, interaktive Komponenten zu ergänzen, quasi „Programme zu browsen“. Die Ernüchterung stellte sich allerdings bald ein. Anstatt des anfänglich beschworenen WORA (Write Once Run Everywhere) hatte man plötzlich WOTA (Write Once Test Everwhere). Bedingt durch Unterschiede und Fehler der diversen Browserversi-

onen war es nämlich nahezu unmöglich komplexere Anwendungen zu schreiben, die in jedem Java fähigen Browser ablauffähig waren. Die Programmierung von GUIs unter direkter Verwendung der AWT oder Swingklassen ist außerdem sehr aufwendig und die Qualität entspricht oft nicht dem was ein Anwender heutzutage von einer Benutzeroberfläche erwartet.

Statt Oberflächen zu realisieren, wie in Abbildung 1, mußte man die eigenen Vorstellungen von Benutzerinteraktion auf eine Folge von HTML Seiten abbilden, die das dem http Protokoll inhärente Prinzip von Request - Response wiederspiegelten.

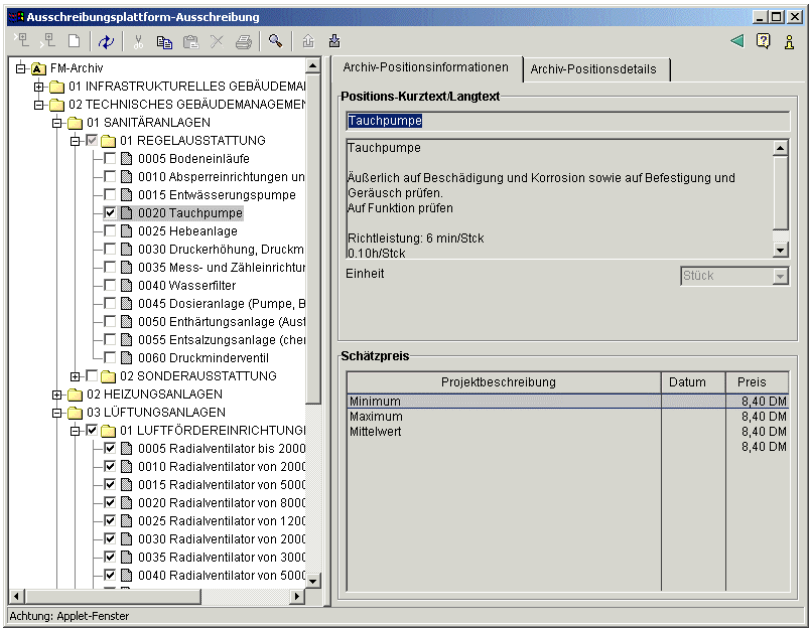


Abbildung 1 - Beispiel für ein Xcc Java Applet

2 Problemstellung

Wo liegen denn eigentlich die Probleme beim Einsatz von Java Applets?

- Die Größe des Appletcodes und die damit stark wachsenden Downloadzeiten, die die Geduld des Anwenders strapazieren. Dies ist bedingt durch:
  - Fehlende direkte Unterstützung von Java Swing in den verbreitetsten Browsern.
  - Designprobleme von Swing, wie z.B. das Listenerkonzept, das zu sehr vielen Klassen führt, die praktisch überhaupt nichts tun.
- Unterschiede und Fehler der Javaimplementierungen der verschiedenen Browser.

- Wenig „ästhetische“ und auf das Wesentlichste beschränkte Kontrollelemente mit Sun’s AWT.
- Die mangelhafte Performanz von AWT- und Swing-Applikationen.
- Hoher Programmieraufwand für GUI Code.

3 Design

Im folgenden werden die wichtigsten Designentscheidungen in Bezug auf den Widgetset und die Beschreibungssprache beschrieben. Die nachfolgende Abbildung zeigt einleitend die prinzipiellen Schritte in der Kommunikation zwischen Client und Server.

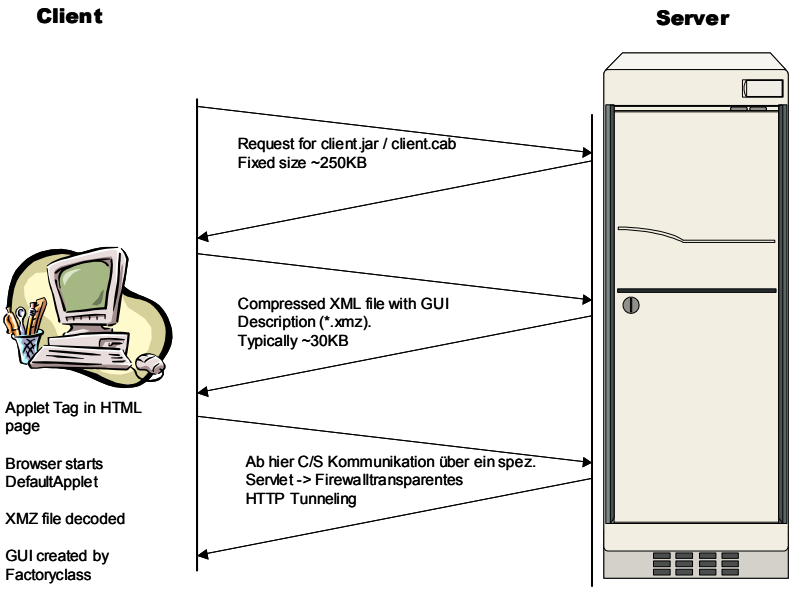


Abbildung 2 - Prinzipielle C/S Kommunikation

3.1 WidgetSet

Es erwies sich als unumgänglich, einen völlig eigenen Widgetset zu implementieren. Zum einen konnten nur hier die unterschiedlichen Browserimplementierungen für den Entwickler transparent behandelt werden, zum anderen deckt sich die Performanz der Standardimplementierungen von Sun, die wahrscheinlich schon auf die nächste Generation von GHz PCs ausgerichtet ist, in keiner Weise mit unseren Ansprüchen.

Ein Java-GUI-Framework für die Verwendung im Browser unterliegt jedoch besonderen Anforderungen und Beschränkungen, welche im Design auch besondere Beachtung finden müssen:

- knappe Systemressourcen: Die realisierten Anwendungen sollen auch auf weniger leistungsfähigen Rechnern mit geringem Speicherausbau funktionsfähig sein.
- langsame Netzverbindung: Die Übertragungszeit einer kompletten Anwendung muss auch bei langsamen Serververbindungen (z.B. über Modem) noch akzeptabel sein.
- Unterschiedliche Browserfabrikate und -versionen: Es kann weder die neueste Java-Release noch eine bestimmte Java-Implementierung vorrausgesetzt werden. Außerdem weisen diverse VM-Implementierungen im Browser insbesondere beim AWT spezifische Mängel auf, welche vor dem Anwendungsentwickler verborgen werden sollen.
- Java-Sandbox: Durch die Securityeinschränkungen in Browserumgebungen können bestimmte APIs nicht verwendet werden, und es sind auch keine nativen Erweiterungen möglich. Eine grundsätzliche Abhängigkeit von Zertifizierung sollte vermieden werden.
- Unterschiedliche GUI Look&Feel Styles: Die verschiedenen Betriebssysteme bzw. deren grafische Oberflächen weisen nicht nur optische Differenzen auf, sondern teilweise auch unterschiedliche Interaktionskonventionen. Dies sollte nach Möglichkeit berücksichtigt werden, um die Effektivität der Anwender nicht zu beeinträchtigen.
- Unterschiedliche Bildschirmauflösungen: Die erstellten Anwendungen sollen sich möglichst einfach den unterschiedlichen Bildschirmauflösungen, Farbtiefen und Schriftgrößen anpassen können.
- Kundenspezifische Anforderungen: Für bestimmte Anwendungen im Internet sind das optische Erscheinungsbild sowie kundenspezifische Designvorgaben von entscheidender Bedeutung. Dies sollte nach Möglichkeit auch einfach umsetzbar sein.

Daraus ergaben sich bereits einige grundsätzliche Vorgaben, wobei die Möglichkeiten zur Minimierung des Codeumfangs sowie die Performanzanforderungen beim Design im Mittelpunkt standen, da dies entscheidendende Faktoren für Eignung des Gesamtsystems waren.

3.1.1 Anbindung an die Javaumgebung

Da die vorhandenen Basis-APIs insbesondere aus dem AWT sowohl in unterschiedlichen Versionen zum Einsatz kommen sollten, als auch auf diversen Plattformen teilweise spezifische Fehler aufwiesen, wurde auf eine direkte Kopplung dieser Klassen verzichtet, und statt dessen die tatsächlich benötigte Funktionalität über Bridgeklassen zur Verfügung gestellt. Damit wurde es möglich, einen einheitlichen Funktionsumfang aller Plattformen zu definieren und somit eine stabile Basis für alle darauf aufbauenden Realisierungen zu schaffen. In dieses API wurden auch Funktionen aufge-

nommen, welche in einigen Zielplattformen nicht oder nur teilweise zur Verfügung standen, jedoch weitestgehend emuliert werden konnten. Das betrifft z.B. das Auslesen der Systemfarben und –fonts, spezielle Eventtypen, sowie diverse Grafikoperationen, welche nur auf bestimmten Plattformen zur Verfügung stehen. Um die Plattformunterschiede in den Bridgeklassen zu egalisieren, mussten hier auch eine Reihe von Fehlerbehebungsroutinen für diverse Browsertypen implementiert werden. Durch diese Grundlage ergaben sich nun weitreichende Freiheiten bezüglich des eigentlichen Frameworkaufbaus.

3.1.2 Primäre GUI-Klassen

Das nachfolgende Diagramm zeigt die Beziehungen der wesentlichen Coreklassen der Widgetbibliothek.

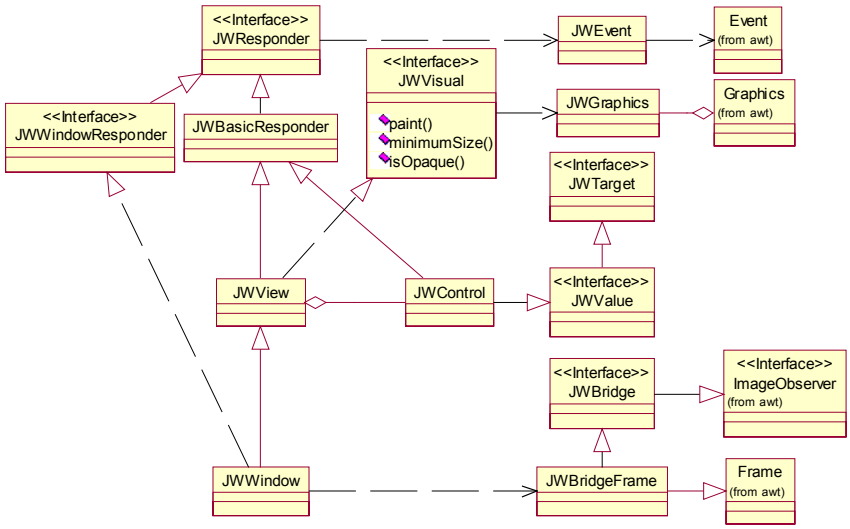


Abbildung 3 - Beziehung der wichtigsten PicoWidget-Klassen

Als Basis der weiteren Designentscheidungen wurden die Erfahrungen mit existierenden nativen Frameworks diverser Systeme analysiert und insbesondere das AppKit-Framework des Nextstep/Openstep-Systems [OpenStep] von Next/Apple als grundlegende Designvorgabe gewählt. Da Java weder über signaturgebundene Polymorphie noch ein Equivalent zu den Categories in Objectiv-C verfügt, dafür jedoch automatische Garbage Collection bietet und außerdem besondere Anforderungen an Codeumfang und Performanz bestanden, wurde das Design zwar an das des AppKit angelehnt, jedoch unabhängig davon neu entworfen.

Basis des GUI-Systems sind das Interface *Visual* und die Klasse *View* welche weitestgehend dem Container im AWT entspricht (ein Äquivalent zu Component gibt es

nicht). Ein *View* implementiert das Visual-Interface und kann beliebig viele weitere Visuals enthalten (z.B. Subviews). Jedes Viewobjekt hat in einer Hierarchie aus anderen Views eine bestimmte Position und kennt seine Koordinaten relativ zum Parentview. Außerdem kann ein *View* primärer Empfänger von Events sein. Ein *View* bzw. eine Ableitung dieser Klasse ist außerdem für das Layout seiner Subvisuals bzw. Subviews verantwortlich. Das *Visual*-Interface definiert Methoden zum Zeichnen eines rechteckigen Bereichs, zur Berechnung der Minimalgröße und zur Ermittlung der Opacity dieses Bereichs. Objekte von Klassen, welche das Visual-Interface implementieren, können direkt als Subvisuals in einen View eingefügt werden. Sie werden dann von dessen Layout positioniert und zum Zeichnen aufgefordert. Da ein Visualobjekt keine Beziehung zu einem Parentview haben muss, kann es gleichzeitig in beliebig vielen Views eingefügt werden. Das Zeichen des Hintergrunds bzw. des Randes (Border) eines Views wird ebenfalls durch Visualobjekte realisiert.

Die Klasse *View* implementiert außerdem das *Responder*-Interface, welches Methoden zur Behandlung primärerer Events (Input) sowie zur Verkettung von Responderobjekten definiert. Die Defaultimplementierung der Eventmethoden leitet die Behandlung an den nächsten Responder weiter. Dadurch entsteht die sogenannte Responderchain. Der nächste Responder eines Views ist normalerweise dessen Parentview. Es können jedoch jederzeit weitere Responderobjekte in die Responderchain eingefügt werden, so dass die Events eines Views an diese Objekte delegiert werden und erst von dort an den Parentview.

Eine weitere Basisklasse, welche das Responderinterface implementiert, ist die Klasse *Control*. Diese Klasse dient als Oberklasse der jeweiligen GUI-Controls (Widgets).
Ein Widget setzt sich normalerweise aus einem Controlobjekt (dem Widget selbst) und mindestens einem Viewobjekt zusammen, welches vom Control benutzt wird. Ein Control ist somit keine Ableitung von View, sondern verwendet üblicherweise ein Objekt einer Standardviewklasse für das Layout und Rendering seiner grafischen Repräsentation. Die eigentlichen Zeichenroutinen werden dann von diversen Visualobjekten implementiert, welche im View des Controls enthalten sind. Es gibt auch Controls, welche keine eigene Viewinstanz haben, sondern an einen beliebigen View angehängt werden z.B. Drag-Handler und KeyEvent-Handler, welche bestimmte Events verwenden, um einen View draggable zu machen oder auf einzelne Keybord-Events zu reagieren.

Die Klasse Window ist eine Ableitung von View, kann jedoch zusätzlich über das Bridge-API in einem AWT-Container eingebettet werden (z.B. AWT-Frame) und bietet die Funktionalität, über das Bridge-API als Top-Level-Window zu agieren. Eine Instanz der Klasse Window kann jedoch auch als Subwindow in einen anderen View eingefügt werden.

Hier noch einige kennzeichnende Zahlen:

Größe in KB	~250
Anzahl Klassen	~160

Zusammen mit weiteren Klassen zur Anbindung an Middleware und Server und (komprimierten) eXPres Beschreibungen der eigentlichen Oberfläche können komplette Clientanwendungen in knapp 300KB untergebracht werden. Der eigentliche Appletcode ist dabei konstant, die entgültige Größe hängt vom Umfang der eXpres Beschreibungen ab.

3.2 Die eXpres Beschreibungssprache

Zur Beschreibung der Benutzeroberfläche wurde eine spezielle Sprache entwickelt. Für das Design der Sprache standen folgende Überlegungen im Vordergrund:

- Die Sprache sollte eine möglichst deklarative Beschreibung der Oberfläche ermöglichen. Sie sollte nicht nur dem Entwickler, sondern, soweit möglich, auch für den Domänenexperten verständlich sein.
- Die Sprache sollte flexibel, erweiterbar und leicht erlernbar sein.
- Die Sprache sollte mächtig genug sein, um eine große Menge von möglichen Oberflächen abzudecken.
- Es sollte möglich sein, auf einfache Art und Weise direkt „Programme“ der Sprache in einem Editor zu bearbeiten oder Oberflächen zu generieren, es sollten aber auch Möglichkeiten für eine einfache Toolunterstützung vorgesehen werden.

Das Ergebnis heißt **eXPres** und ist eine auf XML basierende Beschreibungssprache für Benutzeroberflächen. Im folgenden werden die wesentlichen Eigenschaften von eXPres vorgestellt.
Das Wurzelelement von eXPres ist **<application>** und repräsentiert eine Clientapplikation mit einem oder mehreren Toplevelfenstern. Innerhalb der einzelnen Fenster können hierarchisch eXPres Elemente eingefügt werden. Elemente können dabei zum Einen sichtbar sein, diese Gruppe heißt in eXPres *Visuals*, zum Anderen können aber auch Elemente wie Modelle, Serveranbindungen, Ereignishandler und Bedingungen auftauchen.

Als kurzes Beispiel das unvermeidliche Hello World für eXPres:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE application SYSTEM "client.dtd" [
]>

<application id="app" name="Hello World Example" >
  <shared>
    <font id="fett" style="bold" size="24" />
  </shared>
  <window label="Hello World Demo" id="helloWin"
              x="100" y="100" width="200" height="100">
    <vbox>
      <label label="Hello World" align="center"
font="fett" share="2"/>
      <button label="OK" target="helloWin" action="close"
share="1"/>
```

```
    </vbox>
  </window>
</application>
```

gestartet mit:

```
java de.xcc.client.ClientApp HelloWorld.xml
```



oder als Applet in eine HTML Seite eingebunden ergibt sich dann unter Windows z.B. nebenstehendes Erscheinungsbild. Die aktuelle Implementierung erzeugt dabei, nach dem Decodieren der **eXPres** Datei über Aufrufe an eine Factoryklasse, Objekte der verwendeten Widgetbibliothek. Das heißt, XML spielt nur während des Startups eine Rolle. Zur Laufzeit ist der Client ein ganz normales Java-Programm. Als Parser kommt im Moment noch eine angepasste Version des frei verfügbaren XML Parsers Aelfred zur Verwendung. Eine eigene, spezielle Parserimplementierung steht aber kurz vor dem Abschluß (s.a. Abschnitt 4.2).

3.2.1 Layoutmanagement

Ein Java-Entwickler erwartet natürlich ein automatisches Layoutmanagement und wird hier von eXPres auch nicht enttäuscht. Zur Beschreibung des Layouts wurde eine von Donald Knuth, TeX System [Knuth84] bekannte Metapher gewählt. Die grundlegenden Layoutelemente sind **<hbox>** und **<vbox>**, die wie die Namen schon vermuten lassen, andere Elemente aufnehmen können, die horizontal oder vertikal angeordnet werden. Über Attribute wie `share`, `minwidth`, `minheight`, `insets` etc. lässt sich das Layoutverhalten in weiten Grenzen beeinflussen.

3.2.2 Visuals

Aus der Liste der angebotenen Visuals oder Widgets hier nur ein Auszug: <**tree**>, **<text>**, **<label>**, **<table>**, **<combo>**, **<button>**, **<check>**, **<radio>**, **<table>**, **<tabbox>**, **<image>**. Durch die vorhandenen Primitive können z.B. im Wesentlichen alle Elemente von MS Windows wie Toolbars, Menüs, Tooltips, tabbed Windows, Dialoge, etc. nachgebildet werden. Das Element **<control>** erlaubt es, eigene Widgets oder auch Ableitungen vorhandener Widgets in Form von Javaklassen anzubinden. Diese Klassen müssen dazu nur ein minimales vorgegebenes Interface einhalten. Damit wird es zum Beispiel leicht möglich, spezielle Diagrammtypen einzubinden.

3.2.3 Datenrepräsentation

Zur clientseitigen Repräsentation der Daten wurde ein sehr einfaches, aber trotzdem äußerst flexibles Vorgehen gewählt. Die Daten präsentieren sich dem Client in Form

eines **<model>** Elements. Ein Model kann dabei entweder einen einzelnen Wert (für Textfelder, Combobox, Checkbox, ...), oder eine Stringtabelle (für Tabellen, Trees, ...) enthalten. Die Interpretation des Inhalts bleibt dabei der Modelkomponente des einzelnen Widgets überlassen. Ein Model kann dabei lokale, clientseitige Daten beinhalten, oder über einen sogenannten **<connect>** Daten vom Server. Für den Client ist dies vollständig transparent.

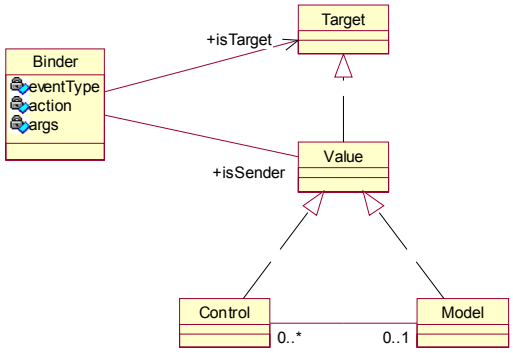


Abbildung 4 - Objektbeziehungen in eXpres

3.2.4 Ereignisbehandlung

An fast jedes Visual können sogenannte **<binder>** angebunden werden. Ein **<binder>** reagiert dabei auf Ereignisse und bringt ein Senderobjekt mit einem Targetobjekt zusammen.
Das folgende Beispiel kopiert z.B. beim Click auf den Button den Inhalt des Sender Textfelds in das Target Textfeld.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<application id="app" name="Test">
  <window label="Binder Example" width="100" height="200">
    <vbox>
      <label label="Sender: "/><text id="s"/>
      <label label="Target: "/><text id="t"/>
      <button label="Copy Sender to Target">
        <binder target="t" action="setValue" arg=”s”/>
      </button>
      </vbox>
  </window>
</application>
```

Ein **<model>** kennt als Target z.B. die Aktionen *update* und *validate*, die gegebenenfalls die Modeldaten durch einen Serverzugriff aktualisieren.

3.2.5 Clientlogik

Auch wenn die eigentliche Businesslogik zu 100% auf Serverseite behandelt wird, gibt es doch Anforderungen an den Client, die auf eine clientseitige Verarbeitung hinauslaufen. So muss z.B. oft in Abhängigkeit von Daten und/oder Ereignissen ein Element disabled bzw. enabled, sichtbar oder unsichtbar gemacht werden. Aus diesem

Grund wurde in eXPres ein Element zur Deklaration von Bedingungen und den daraus abzuleitenden Reaktionen implementiert. **<condition>** beinhaltet dazu als Kindelemente beliebig verschachtelte, komplexe Ausdrücke (**<and>**, **<or>**, **<not>**, **<eq>**, **<neq>,** ...) und erzeugt bei Erfüllung (oder Nichterfüllung) der Bedingung Ereignisse, die über den weiter oben beschriebenen **<binder>** Mechanismus in Aktionen umgesetzt werden können.

3.3 Die Client/Server Kommunikation

Da die Serverseite in diesem Paper nicht weiter betrachtet werden soll, hier nur kurz ein paar Eckpunkte unserer Lösung:

Die Anbindung der Daten und Services des Servers sollte für den Entwickler der GUI Seite möglichst transparent bleiben. Es wurde deshalb ein Paar korrespondierender Interfaces definiert, die die eigentlich Kommunikation zwischen Client und Server abstrahieren. Auf Clientseite gibt es zu diesem Zweck das Interface Connect, die Entsprechung auf Serverseite heißt Gate. Diese beiden Interfaces regeln das komplette Handling der C/S Kommunikation, d.h. Aufruf von Services, Marshalling der Daten, Liefern von Ergebnissen oder Ausnahmen (Exceptions). Anhand des globalen Metamodells und der zu verwendenden Middleware werden die entsprechenden implementierenden Klassen vollautomatisch generiert. So werden für das HTTP Tunneling entsprechende ConnectHttp und GateHttp Klassen erzeugt. Ein Paar dieser Klassen repräsentiert dabei jeweils ein bestimmtes Interface, das wiederum eine Menge von Operationen definiert. Abbildung 5 zeigt die Gesamtarchitektur der Xcc Technologieplattform. Datenbank und Kommunikationstechnologie sind dabei nur exemplarisch zu verstehen.

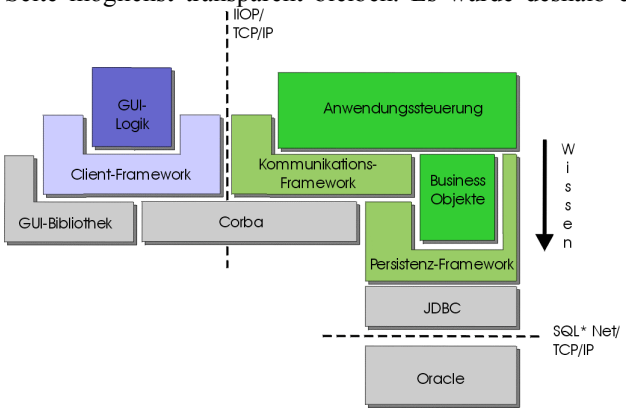


Abbildung 5 - Architektur unserer Technologieplattform

Das Marshalling ist sehr einfach gehalten (nur Strings und Stringtables) aber leicht erweiterbar und überraschend effizient. In die Generierung lassen sich über Konfigurationsparameter Aspekte wie Verschlüsselung etc. einbringen. Momentan existieren Generatoren für „Tunneling“ über http, Corba (Visibroker) und RMI.

4 Implementierung

Es folgen Hinweise auf die von uns gewählten Implementierungen und Erfahrungen mit diesen Lösungen.

4.1 Widgetset

Basierend auf dem Design von NextStep bzw. OpenStep erwiesen sich folgende Dinge als besonders gewinnbringend:

- Alle Klassen haben wohldefinierte Verantwortlichkeiten und delegieren Aufgaben, die außerhalb dieser Verantwortung liegen, an die dafür zuständigen Klassen. Die gesamte Bibliothek folgt somit dem Konzept „Design by Responsibility“. Dadurch entstehen leistungsfähige, optimierte Einzelkomponenten. Das Ergebnis ist eine redundanzfreie, orthogonale Implementierung.
- Minimierung der Erzeugung von Objekten. Das Erzeugen eines Objektes ist in Java eine relativ kostspielige Operation, es lohnt sich daher besonders das Anlegen nur temporär genutzter Objekte zu vermeiden. In diesem Zusammenhang steht auch die Verwendung des Flyweight Pattern um den Memory Footprint der Anwendung möglichst klein zu halten.
- Behandlung von Events über eine Kette von Respondern. Jedes Objekt, das Events eines anderen Objekts verarbeiten will, implementiert das Responderinterface und hängt sich in die Kette der Responder des Objekts ein. Dies verhindert im Gegensatz zu den Listenerklassen von Swing das schnelle Anwachsen der notwendigen Klassen (class bloating) und vereinfacht das dynamische Erstellen von Oberflächen über eine Factory erheblich.

Der eigene Widgetset (PicoWidgets) ist allerdings zu 100% in Java geschrieben und damit völlig plattformunabhängig. Er benutzt als Basis nur sehr wenige Klassen des AWT (Frame, Applet, Graphics), die in fast allen Browserversionen fehlerfrei und stabil implementiert sind. Alle weiteren Operationen, insbesondere das Rendering der Widgets wurde vollkommen neu implementiert. Die PicoWidgets stellen eine Menge von atomaren Elementen zur Verfügung, die durch Factories dann zu den eigentlichen Widgets mit dem gewünschten Look&Feel zusammengebaut werden. Unsere Bibliothek stellt also eine Art GUI Assembler dar, der es erlaubt, effizient fast jede denkbare Benutzeroberfläche zusammen zu bauen.

4.2 XML Kompressor

Obwohl schon ziemlich schnell, erwies sich das Parsen der XML Datei auf Clientseite als ein weiterer optimierungswürdiger Engpass. Ergebnis der Bemühungen ist ein XML-Parser/Kompressor, der durch entsprechende Vorverarbeitung und ausgeklügelte statistische Verfahren die typischerweise anfallenden XML GUI Beschreibungen stark komprimiert (Faktor 25-30) und gleichzeitig ein erneutes Parsen des Ergebnisses

überflüssig macht. Der Decoder implementiert einen endlichen Automaten und kann die komprimierte Datei sehr schnell decodieren.

5 Ausblick

Unsere Lösung ist bei verschiedenen Kunden bereits im produktiven Einsatz und hat sich bisher als sehr stabil erwiesen. Im Moment arbeiten wir am Anschluss an andere Serverarchitekturen, wie z.B. EJB Server und an der Verbesserung der vorhandenen Entwicklerwerkzeuge.

Der Ansatz XML basiert deklarativ Oberflächen zu beschreiben hat sich an anderen Stellen als überraschend effizient erwiesen. Verschiedene Tools im Rahmen unserer Inhouse Technologieplattform (von der die Picowidgets nur einen kleinen Teil ausmachen) wurden verschiedene Tools entwickelt, die auf eXPres Beschreibungen aufsetzten, oder diese automatisch generieren. So unterstützt das neuste Tool das Debuggen von Multithreadinganwendungen. Dabei wird der zu untersuchende Code mit einfachen Aufrufen instrumentiert, die als Ergebnis eine XML Datenstruktur erzeugen. Diese Datenstruktur wird in einem zweiten Schritt vollautomatisch in eine eXPres Beschreibung umgewandelt, die dann die Nebenläufigkeiten der getesteten Anwendung sehr anschaulich visualisiert.

Weitere Tools visualisieren die Packageabhängigkeiten in beliebigen Java Applikationen, erlauben das Abhören von Client/Server Kommunikation, oder unterstützen die Erstellung IDL ähnlicher Schnittstellenbeschreibungen (s.a. nachfolgende Abbildung 6).

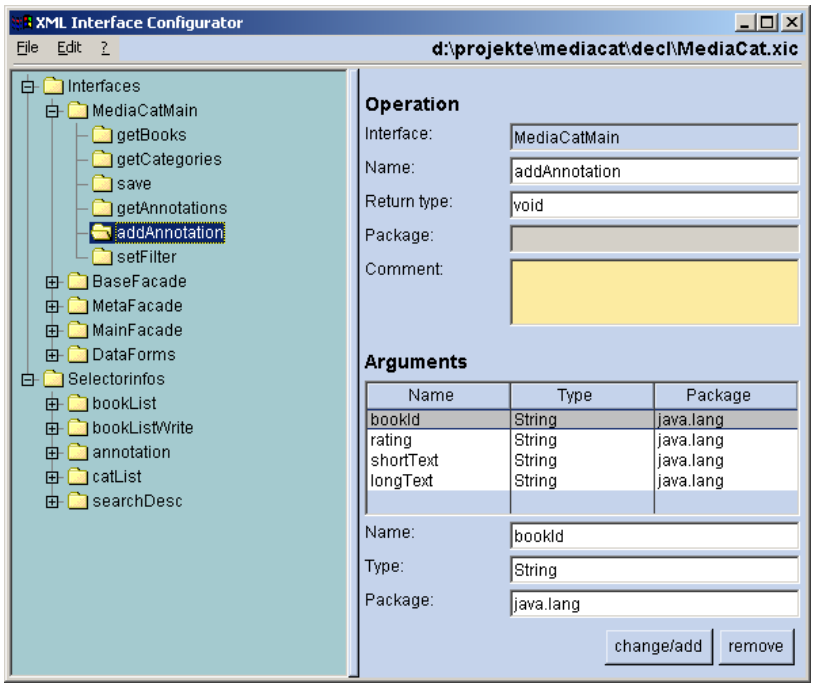


Abbildung 6 - Verwendung von eXPres für Tools

6 Literatur

[DashOPro] Java Optimizer von Preemptive
http://www.preemptive.com/dasho_op.html

[eXPres] eXPres or "How to cook GUIs for Xcc FOOD", Xcc Software intern (info@xcc.de oder **http://www.xcc.de**)

[Knuth84] Donald E. Knuth, The TeXbook, Addison Wesley, 1984

[SunAWT] Abstract Window Toolkit (AWT)
(**http://java.sun.com/j2se/1.3/docs/guide/awt**)

[SunSwing] The Java Swing Connection
(**http://java.sun.com/products/jfc/tsc/index.html**)

[TPP] The pragmatic Programmer, Andrew Hunt und David Thomas, Addison Wesley, 1999