Nachrichtenbasierte Kommunikation zwischen Smalltalk und Java-Komponenten

Interoperabilität mit J2EE-Anwendungen

Bei fast allen größeren Systemen wird heutzutage nicht „auf der Grünen Wiese“ begonnen, sondern ein neues System muss auf die eine oder andere Art mit bereits bestehenden Komponenten kommunizieren, die sehr oft eben nicht in Java entwickelt wurden. Das Thema „Interoperabilität“ behandelt hier die Fragestellung, wie J2EE-Anwendungen mit bestehenden bzw. generell auch mit Nicht-Java-Komponenten (etwa Smalltalk-Komponenten) kommunizieren und umgekehrt.

Dieser Artikel zeigt daher, wie eine kostengünstige und erfolgreiche Kommunikation zwischen Smalltalk- und Java-Komponenten funktioniert. Diese Lösung reicht bis hin zur echten dienstorientierten „Welt“ mit heterogenen Komponenten in verschiedensten Programmiersprachen und Umgebungen.

Ziel einer Anwendungsintegration ist es, funktional ausgerichtete Anwendungen mittels Kommunikation derart zu integrieren, dass sie als ein System geschäftsprozessorientiert arbeitet und dies möglichst effizient und automatisiert. Hierbei geht es um die Interoperabilität zwischen Komponenten. Konzeptionell reichen diese Möglichkeiten der Interoperabilität von einfachen synchronen generischen „Remote“-Funk-

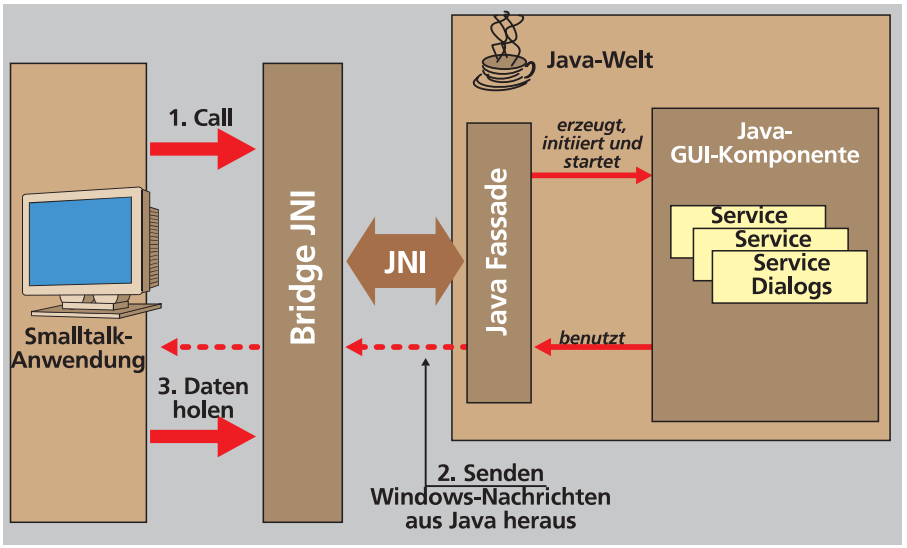

Bild 1: Kommunikationsmuster auf Basis des Bridge-Patterns und Java Native Interface (JNI).

tionsaufrufen, über asynchrone Verarbeitung mittels Nachrichten, bis hin zur echten dienstorientierten Welt mit stark heterogenen Komponenten in verschiedensten Programmiersprachen und Umgebungen, die nicht aus J2EE heraus aufgerufen werden können. Vielmehr kann hier auch umgekehrt z.B. eine Smalltalk Anwendung Enterprise JavaBeans (EJB) als Dienstanbieter nutzen. Die zu kommunizierenden Anwendungen sind häufig getrennt voneinander entwickelt worden und verfügen daher u.a. über eine eigene Präsentationslogik und Fachlichkeit. Hierbei handelt es sich auch um eine Paradigmenintegration, wenn diese Anwendungen auf Basis unterschiedlicher Programmiersprachen entwickelt wurden. Bei der Paradigmenintegration geht es nicht nur um den Aufruf von API (Application Programming Interface)-Methoden, sondern auch um den Aufruf des Systems als eigenständige Komponente und den damit verbundenen Datenaustausch.

In dem vorliegenden Artikel wird in einem Fall beschrieben, wie eine Smalltalk-Anwendung eine Java-Anwendung als eigenständige Windows-Komponente aufruft und mit ihr Daten austauscht. Im anderen Fall nutzt eine Smalltalk-Anwendung Java-Dienste, die mittels EJB-Clients bereits gestellt werden.

Interaktion zwischen Smalltalk-Anwendung und Java

Auf einer Windows-Plattform ergeben sich unter anderem folgende Fragen bei dem Aufrufen einer eigenständigen Java-Anwendung (Java-GUI-Komponente) durch eine Smalltalk-Anwendung:

1. Wie kann eine Smalltalk-Anwendung eine Java-GUI-Komponente starten und verwalten?
2. Wie können die Daten zwischen Java-GUI-Komponente und Smalltalk-Anwendung ausgetauscht werden?

Bei der ersten Frage geht es um die Kontrolle von Smalltalk über Java-GUI-Komponente. Die Smalltalk-Anwendung muss über die Möglichkeit verfügen, den Zustand der Java-GUI-Komponente zu verfolgen und gegebenenfalls zu korrigieren.

Die zweite Frage betrifft die Kommunikation zwischen der Java-GUI-Komponente und Smalltalk: Die Java-Komponente muss der Smalltalk-Anwendung Ergebnisse eines Geschäftsvorfalls zur Verfügung stellen. Ferner muss sie der Smalltalk-Anwendung ihren Zustand (etwa „beendet“), bekannt geben können.

Da wir Windows als Plattform für diesen Anwendungsfall verwenden, muss die Java-GUI-Komponente die Möglichkeit erhalten, die Windows-API zu bedienen. Wir nutzen zur Zeit die Windows-Plattform, weil Windows in der Praxis das am meisten verwendete Betriebssystem für Client-Rechner ist.

In [Cac02] wurde eine auf dem Bridge-Pattern basierende Softwarearchitektur vorgestellt, um die Kommunikation zwischen Java und der Nicht-Java Welt zu implementieren. Diese Lösung ermöglicht einen sicheren Aufruf von Java-Diensten aus der C++-Welt. Wir verwenden diese Lösung wieder, um einen sicheren Aufruf von Java-Diensten aus der Smalltalk-Welt zu erreichen. Auf diesem Weg möchten wir die Kommunikation zwischen Smalltalk und Java realisieren.

Die Java-Komponente kann ihren Zustand über die Windows-API der Smalltalk-Anwendung bekannt geben. Zu diesem Zweck müssen die Windows-Funktionen etwa SendMessageTimeout() umhüllt („wrap“) und aus Java heraus als native (Betriebssystem-)Funktionen aufgerufen werden.

Für diesen Fall ergibt es sich folgendes Kommunikationsmuster, das sich grob in drei Teile gestaltet:

1. Aufruf der Java-Komponente mit Übergabe des Window-Handle des Smalltalk-Hauptfensters. Der Win-

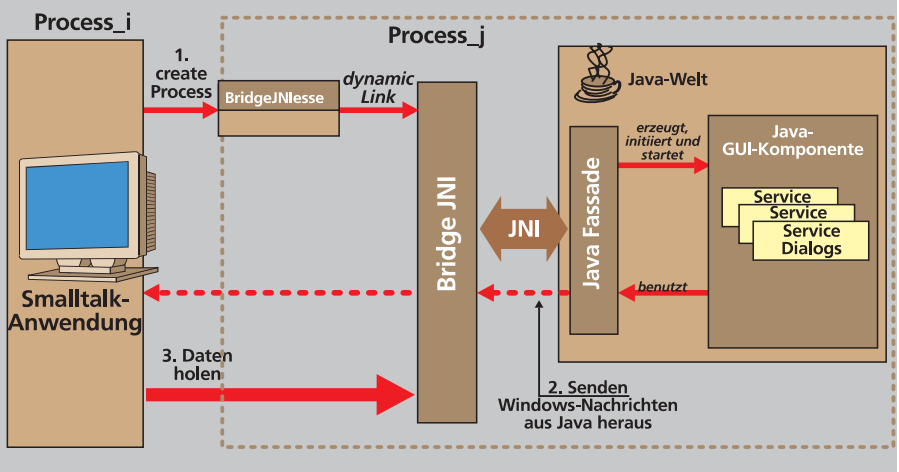

Bild 2: Softwarearchitektur des Kommunikationsmusters mit zwei Prozessen.

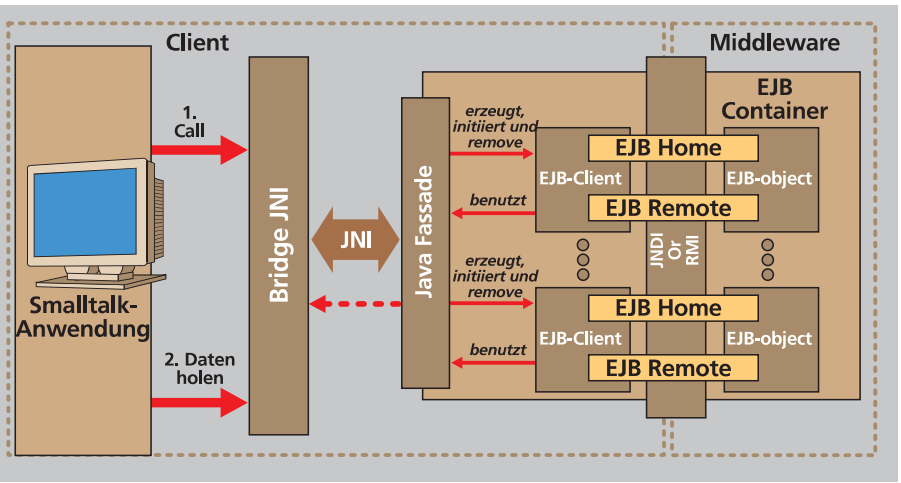

Bild 3: Softwarearchitektur des Kommunikationsmusters für Verteilte Objekte.

dow-Handle des Smalltalk-Hauptfensters repräsentiert den Sender. Der Empfänger kann sich etwa über diesen Handle beim Sender registrieren.

2. Senden einer Window-Message aus Java heraus an Smalltalk mit Übergabe des in Java ermittelten Window-Handle (Hauptfenster der Java-Komponente). Dieser Window-Handle repräsentiert eine Java-Komponente, wodurch der Sender (eine Smalltalk-Anwendung) in der Lage ist, den Empfänger (eine Java-Komponente) zu verwalten. Er kann ihm etwa den Auftrag erteilen, sich zu beenden oder das Java-Hauptfenster in der „Client-Area“ des Smalltalk-Hauptfensters zu „rearrangieren“.

3. Datenaustausch zwischen Java und Smalltalk

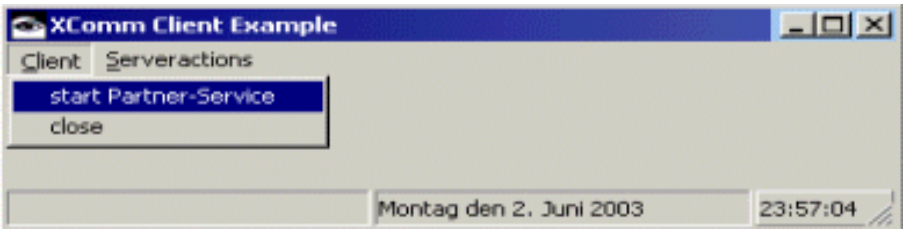

Bild 4: Smalltalk-Hauptfenster einer Beispielsanwendung nach dem Start.

Gemäß [Cac02] sieht dieses Kommunikationsmuster auf Basis des Bridge Patterns und JNI (Java Native Interface) wie in Bild 1 aus. Um eine Kommunikation zwischen den Komponenten zu ermöglichen, müssen beide den Window-Handle der anderen Komponente kennen. Java bekommt den Window-Handle des Smalltalk-Fensters beim Starten der Java-Komponente über die „Shared Library“ BridgeJNI mitgeteilt. Die Fassade der Java-Komponente muss einen Setter (zum Beispiel setWindowHandle(long hWnd)) vorsehen. Diese Methode wird von Smalltalk heraus mit Hilfe des BridgeJNI identifiziert und ausgeführt. In [Cac02] ist beschrieben, wie eine Java-Methode unabhängig von der Anwendungslogik aus der C/C++ Welt identifiziert und ausgeführt wird.

In Java gibt es aber keine Möglichkeit, den „echten“ Betriebssystem-Handle der Swing-GUI zu ermitteln. Das liegt darin begründet, dass Java Plattform übergreifend konzipiert und umgesetzt ist und sich deswegen nicht an eine Plattform (z.B. Windows) anlehnen kann.

Die Lösung liegt auf der Hand: Mittels JNI wird die Windows-Funktion „FindWindow()“ aus Java heraus aufgerufen. In [Sun-1] ist beschrieben, wie man native Funktionen innerhalb einer Java-Komponente spezifizieren kann.

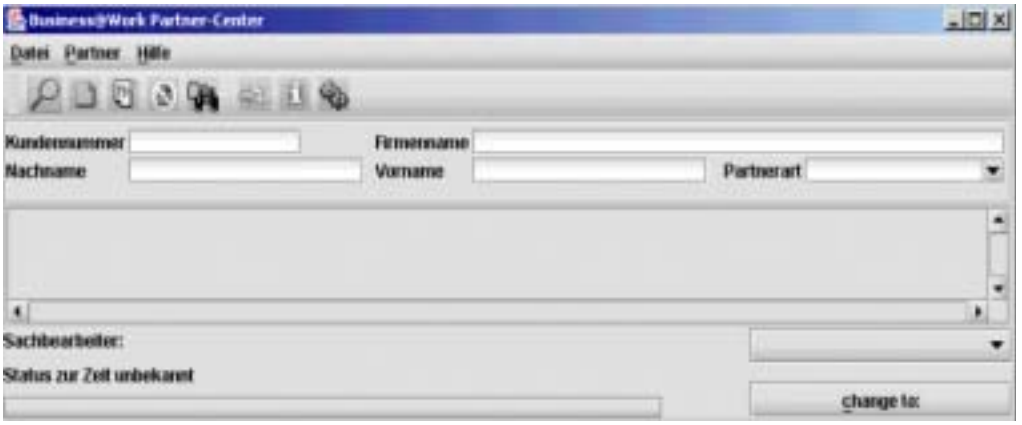

Bild 5: Java-GUI-Komponente (Beispiel) nach dem Start.

Diese native Funktion sucht ein Top-Level-Fenster anhand des vorgebenen Fensternamens und liefert den Window-Handle zurück. Das heißt, anhand der Information des Fenstertitels (sollte in der Prozessliste eindeutig sein) der betreffenden Swing-GUI kann der Window-Handle ermittelt werden. Damit dieser der Smalltalk-Anwendung bekannt gemacht werden kann, muss die Java-Komponente erneut die Windows-Funktion „SendMessageTimeout()“ aufrufen. Diese Funktion sendet eine Nachricht an ein anderes Windows-Fenster und wartet auf eine Antwort innerhalb des spezifizierten Time-Out-Intervalls. Mit dieser nativen Funktion kann eine Nachricht an das Smalltalk-Fenster mit dem Window-Handle des Swing-GUIs gesendet werden. Als eine Möglichkeit, den Windows-Handle der Swing-GUI zu übermitteln, wird die Nachricht „WM_COPYDATA“ und die dazugehörige Struktur „COPYDATASTRUCT“ verwendet. Diese Struktur wird mit den nötigen Daten (diesmal dem Windows-Handle der Swing-GUI) gefüllt und an Smalltalk gesendet. Empfängt Smalltalk diese Nachricht, speichert Smalltalk den Windows-Handle als Instanzvariable in

Bild 6: Ausgabe der empfangenen Nachricht in der Statuszeile.

einem Controller (Model View Controller Prinzip) ab und kann damit auch dem Swing-GUI Nachrichten senden und somit die Java-Komponente verwalten. Der Abschnitt „Message-Dispatching in Smalltalk“ beschreibt, wie Windows-Nachrichten innerhalb einer Smalltalk-Komponente behandelt werden können.

Um Smalltalk über das Schließen der Java-Komponente zu informieren, wird dann beispielsweise die Nachricht „WM_QUIT“ über die Windows-Funktion „PostMessage()“ gesendet. Diese Funktion stellt eine Nachricht in die Warteschlange für den Thread, der das vorgegebene Fenster erzeugt hat, und kehrt zurück, ohne auf eine Antwort zu warten. In [Rich96] befindet sich eine Übersicht zum Thema Windows-API.

Sichere Kommunikation zwischen Smalltalk und Java-GUI-Komponente mittels Multiprocessing

Generell läuft ein Swing-GUI in einem eigenen Thread. Wird die Nachricht „dispose()“ beim Schließen des Swing-Hauptfensters gesendet, wird das Swing-GUI mit dem dazugehörigen Thread beendet, nicht aber der Thread, mit welchem die JVM aus JNI heraus gestartet wurde. Wird beim Schließen des Swing-GUIs die Nachricht „System.exit()“ gesendet, wird die JVM beendet und ebenso die virtuelle Maschine von Smalltalk, da beide Komponenten in einem Prozess laufen. Das ist nicht akzeptabel. Abhilfe schafft die Trennung der Prozesse. Die Java-Komponente bekommt einfach einen eigenen Prozess. Also startet die Smalltalk-Anwendung einen Prozess für die Kommunikation mit der Java-GUI-Komponente über die Windows-API-Funktion „WinCreateProcess()“. Also die Kommunikationskomponente „BridgeJNI.dll“ muss innerhalb eines ausführbaren Programms gekapselt werden, damit eine Smalltalk-Anwendung einen Windows-Prozess mit diesem ausführbaren Programm erzeugen kann.

Diese Entwurfsentscheidung hat den Vorteil, dass die Java-Komponente mit „System.exit()“ verlassen werden kann und der Prozess sauber beendet wird, ohne den Smalltalk-Prozess zu beenden. Bild 2 zeigt die veränderte Softwarearchitektur.

Interaktion zwischen Smalltalk-Anwendungen und Nicht Java-GUI-Komponenten

EJB (Enterprise Java Beans)-Komponenten werden hier stellvertretend benutzt, um die Möglichkeit der Nutzung von nicht GUI-orientierten Java-Diensten in einer Smalltalk-Anwendung zu zeigen. Bekanntlich können Komponenten mittels EJB auch über die Grenzen der Hardware, Betriebssysteme und Middleware genutzt werden. Also können sie Plattform übergreifend kommunizieren. Auch hier müssen folgende Fragen beantwortet werden:

1. Wie kann eine Smalltalk-Anwendung Dienste einer EJB-Komponente nutzen?
2. Wie können die Daten zwischen EJB-Komponenten und Smalltalk-Anwendungen ausgetauscht werden?

Bei der ersten Frage geht es um die Kontrolle von Smalltalk über EJB-Clients. Die Smalltalk-Anwendung muss über die Möglichkeit verfügen, bestimmte EJB-Clients anzusprechen und ggf. zu löschen. Die zweite Frage betrifft die Möglichkeit Ergebnisse eines Dienstes des EJB-Clients abzuholen.

Auch hier findet unseres Muster Verwendung. Bild 3 zeigt die Nutzung des in [Cac02] vorgestellte Softwarearchitektur für eine solche Kommunikation über EJB in einer verteilten Umgebung.

Dieses Muster ist plattformunabhängig, da JNI und EJB architektur- und plattformneutral sind. Jede EJB-Komponente läuft in einem Container ab. Im Container kann eine EJB-Komponente erzeugt und auch wieder gelöscht werden. Bekanntlich stellt der Container zwischen dem Client und den EJB Komponenten eine „Home“- und „Remote“-Schnittstelle für die einheitliche Kommunikation zur Verfügung. Über JNDI (Java Naming and Directory Interface) greift der Client dann auf die „Home“-Schnittstelle zu. Die Fähigkeit wird über die „Java Fassade“ der Smalltalk-Anwendung angeboten.

Message Dispatching in Smalltalk

Um in Smalltalk Nachrichten empfangen zu können, die Java an Smalltalk sendet, müssen so genannte „Event-Hooks“ etabliert werden. Ein „Event-Hook“ ist eine Verankerung im Code, der dann betreten wird, wenn das mit dem Event-Hook vereinbarte Event ausgelöst wird und der Empfänger (immer ein gültiger Window-Handle) dem Prozess zugehörig ist, in dem der Code (mit dem Event-Hook) läuft. Für die Kommunikation zwischen Smalltalk und Java wurde die Funktion „SendMessageTimeout“ in Verbindung mit den Nachrichten „WM_COPYDATA“ oder „WM_QUIT“ verwendet. Wobei die

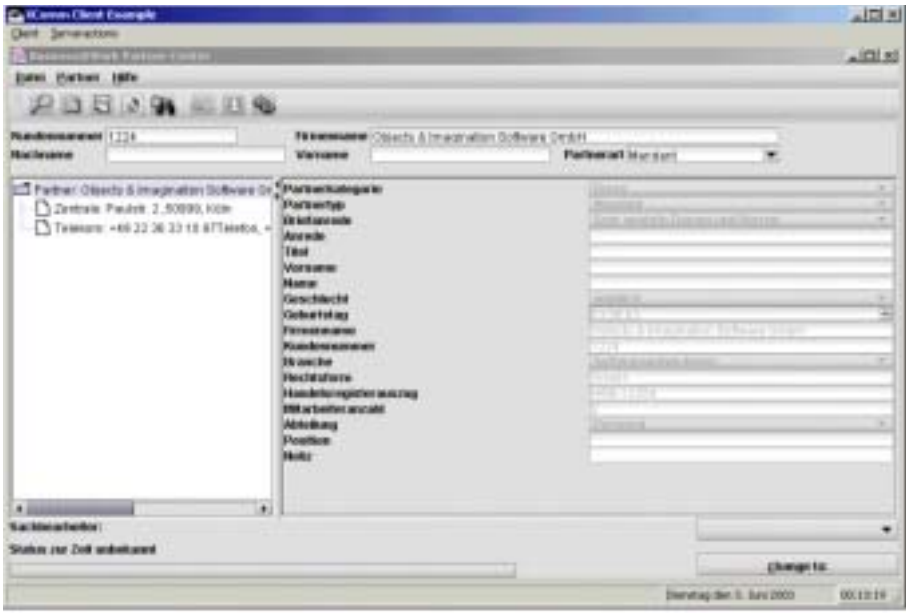

Bild 7: Das Java-Fenster als „echtes Kind“ des Smalltalk-Fensters.

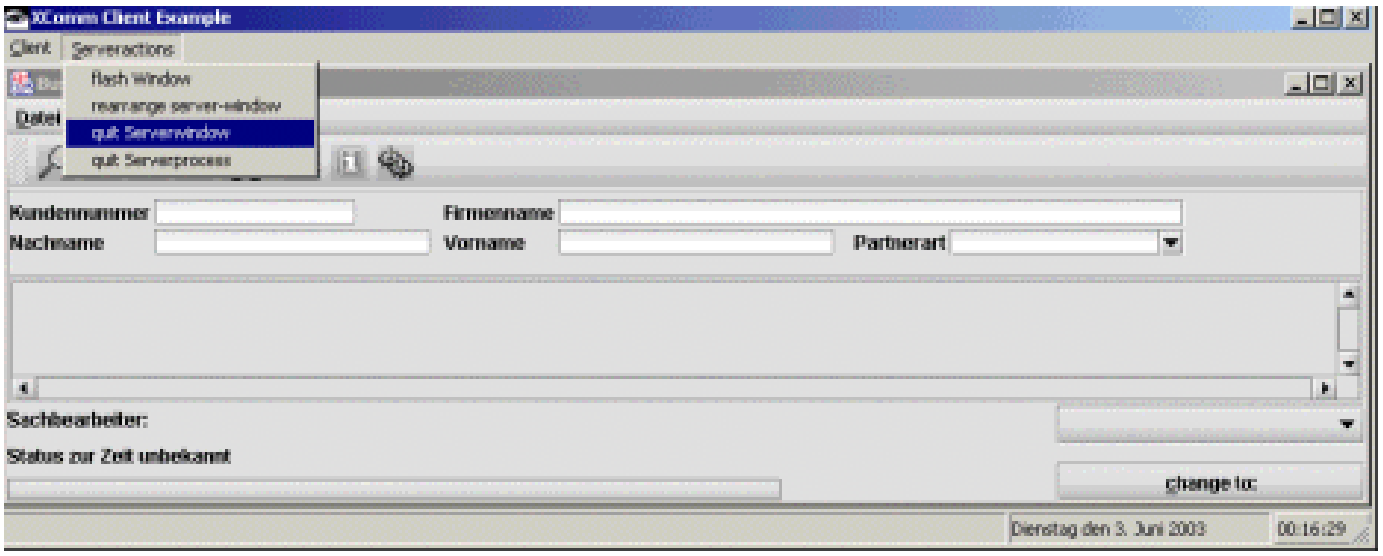

Bild 8: Das Beenden des Java-Fensters (dispose()).

Nachricht „WM_COPYDATA“ als Standardinstrument für den Datenaustausch zwischen zwei Prozessen innerhalb von Windows verwendet wird. Die Nachricht „WM_QUIT“ ist die Standardnachricht beim beenden von Prozessen bzw. Fensterprozeduren. Um die von Smalltalk empfangene Nachricht zu visualisieren, wird diese beim Empfang in die Statuszeile des Smalltalk-Fensters ausgegeben.

Nach dem Start der Smalltalk-Anwendung öffnet sich das Hauptfenster von Smalltalk mit den Java-Services, welche über Smalltalk gestartet werden können. Über den Menüpunkt „Client/start Partner-Service“ lässt die Java-Komponente aufrufen. Durch die beim Start der Java-Komponente gegenseitige Registrierung der Window-Handles lassen sich jetzt ungehindert Nachrichten zwischen diesen Komponenten austauschen. Jede von Smalltalk empfangene Nachricht wird in der Statuszeile ausgegeben.

Da über die Windows-Funktionen eine nahtlose Integration zwischen den Prozessen (Fenstern) möglich ist, kann auch das Java-Hauptfenster völlig unter der Kontrolle des Smalltalk-Fensters gebracht werden. Am deutlichsten kann man das durch eine „Vater-Kind-Beziehung“ machen, in dem das Java-Fenster „echtes“ Kind vom Smalltalk-Fenster wird und sich dieses in der „Client-Area“ des Smalltalk-Fensters darstellt.

Ebenso lässt sich das Java-Fenster, oder gar der ganze Prozess über die Smalltalk-Anwendung beenden (Bild 7). Wird der Prozess beendet, lässt sich dieser erneut Starten und die Java-Komponente öffnet sich erneut (Bild 8).

Fazit

Die in [Cac02] vorgestellte Softwarearchitektur ermöglicht eine generische Interoperabilität zwischen der Nicht-Java- und Java-Welt über JNI. Dieser Artikel hat gezeigt, dass diese Softwarearchitektur auch für eine Interprozess-Kommunikation zwischen Smalltalk- und Java-Komponenten sehr gut geeignet ist. Natürlich kann man RMI (Remote Methode Invocation)-Verfahren[Ibm-1] verwenden, um diese Art von Kommunikation umzusetzen. Leider lassen sie sich schwierig und nicht ohne eine Erweiterung bzw. Anpassung der bestehenden Komponenten anwenden. Darüber hinaus hat dieser Artikel gezeigt, wie Windows-Funktionen, die in Java als native Methoden aufgerufen wurden, als Fensternachrichten in Smalltalk abgearbeitet werden können.

Diese Lösung lässt sich auf andere Plattformen übertragen. Für eine Unix-Plattform kann beispielsweise der message-Mechanismus verwendet werden, um die Interprozess-Kommunikation zwischen Java- und Smalltalk-Komponenten zu realisieren. Die Nachrichten werden an den „Nachrichtenspeicher“ (Message-Queue) geschickt und können von dort abgeholt werden. Eine weitere Möglichkeit der Prozesssynchronisation stellen „Signale“ in der Unix-Plattform dar. Ein Signal kann in diesem Zusammenhang verwendet werden, um „Post-Message“ zu simulieren.

Dr. Ndombe Cacutalua und Frank Krutik
Ndombe.Cacutalua@softlab.de
fkr@objects-imagination.de

Literatur

[Cac02] Dr. Ndombe Cacutalua und Joachim Korittky; Sicherer Aufruf von Java-Diensten aus der C++-Welt auf Basis des Bridge-Pattern; OBJEKTspektrum 05/2002

[Ibm-1] VisualAge Java-RMI-Smalltalk, the ATM Sample from A to Z; http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245418.pdf

[Sun-1] Sun Microsystems; Lesson: Writing a JAVA Program with Native Methods; http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/index.htm

[Sun-2] Sun Microsystems; JNI -JAVA Native Interface; http://java.sun.com/j2se/1.3/docs/guide/jni

[Sun-3] Sun Microsystems; JNI-Jan Native Interface Tutorial; http://java.sun.com/docs/books/tutorial/native1.1/

[Rich96] Richard J. Simon; Win32 Programming API Bible; SAMS, Macmillan Computer Publishing, USA

[Asb-99] Stephen Asbury und Scott R Weiner; Developing Java Enterprise Applications; Wiley 1999.