Nachrichtenbasierte Kommunikation zwischen Java-Komponenten und Smalltalk

von Dr. Ndombe Cacutalua, Frank Krutik, Anton Rommerskirchen

Gut, dass wir gesprochen haben

Interoperabilität bedeutet, dass J2EE-Anwendungen mit bestehenden bzw. mit nicht-Java-Komponenten wie Smalltalk kommunizieren und umgekehrt. Denn bei fast allen größeren Systemen wird heutzutage nicht auf der grünen Wiese zu programmieren begonnen, sondern ein neues System muss auf die eine oder andere Art mit bereits bestehenden, in Java entwickelten Komponenten kommunizieren. Dabei können beispielsweise diese neuen Dienste innerhalb des vorhandenen Systems aufgerufen werden. Aus Kostengründen rückt das Wiederverwenden bestehender oder neuer Komponenten dabei in den Vordergrund. Dieser Artikel zeigt, wie die Kommunikation zwischen Smalltalk- und Java-Komponenten funktionieren kann.

Paradigmenintegration zwischen Smalltalk und Java

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-Funktionsaufrufen, 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 bspw. auch eine Smalltalk-Anwendung EJB als Dienstanbieter nutzen. Die zu kommunizierenden Anwendungen sind häufig getrennt voneinander entwickelt worden und verfügen daher u.a. über eine eigene Präsentationslogik und eigene 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 APIs, sondern auch um den Aufruf des Systems als eigenständige Komponente und den damit verbundenen Datenaustausch.

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

Die zugrundeliegende Softwarearchitektur ermöglicht einen sicheren Aufruf von Java-Diensten aus der nicht-Java-Welt [1]. Wir verwenden diese Lösung, um

einen sicheren Aufruf von Java-Diensten aus der Smalltalk-Welt zu erreichen. Auf diesem Weg realisieren wir die Kommunikation zwischen Smalltalk und Java.

Listing 1

Starten einer JVM über eine JNI-Funktion für JDK ab Version 1.2

```
JavaVMInitArgs bufferInitArgs1_2;
JavaVMOption  options[2];

JavaVM*       pJVM;
JNIEnv*       pEnv;

options[0].optionString = "-Djava.compiler=NONE";
options[1].optionString = getJAVAUserClsPath();
 //user-defined Method for getting CLASSPATH
options[2].optionString = "-verbose:jni ";

bufferInitArgs1_2.version = JNI_VERSION_1_2;
bufferInitArgs1_2.options = options;
bufferInitArgs1_2.nOptions = 3;
bufferInitArgs1_2.ignoreUnrecognized = true;

ErrorCode = JNI_CreateJavaVM(&pJVM, (void **)&pEnv,
                                    &bufferInitArgs1_2);
 if (ErrorCode < 0)
{
 //Fehlerbehandlung
}
```

Listing 2

Code-Fragment einer Java-Klasse

```
package vertragPackage;

public class VersicherungsVertrag
 {
  //Konstruktoren
   public VersicherungsVertrag() { /* something more
                                          than ... */}

 //Dienste
   public void setGesellschaft ( String PstrName)
                          {/* something more than ... */}
  //Attributes
    private String       m_strGesellschaft;

}
```

Listing 3

Finde einer Java-Klasse über eine JNI-Funktion

```
//Finde "VersicherungsVertrag" innerhalb des Packages
//"vertragPackage"
jclass handleAufVertrag = pEnv ->Findclass
                ("vertragPackage/VersicherungsVertrag");
//Fehler Behandlung
 if (pEnv ->ExceptionOccurred())
 {
 //Fehler Behandlung
}
```

Softwarearchitektur: Bridging von JNI-Funktionalitäten

Für die Interoperabilität zwischen Java-Anwendungen und Anwendungen anderen Ursprungs bietet J2EE eine Lösung über das JNI (Java Native Interface). Eine nicht-Java-Anwendung muss mit einer C-Anwendung kommunizieren können. Über JNI kann in eine C/C++-Anwendung die JVM eingebunden werden, um das Laden von Java-Klassen und den Zugriff auf die geladenen Klasseninformationen zu ermöglichen. Genauso lassen sich innerhalb von Java Methoden zur Ausführung bringen und Objekte manipulieren. Das Auftreten von Ausnahmesituationen aus der Java-Welt wird von JNI aufgefangen und weitergeleitet. Abbildung 1 zeigt JNI als Bindeglied zwischen C/C++ und Java.

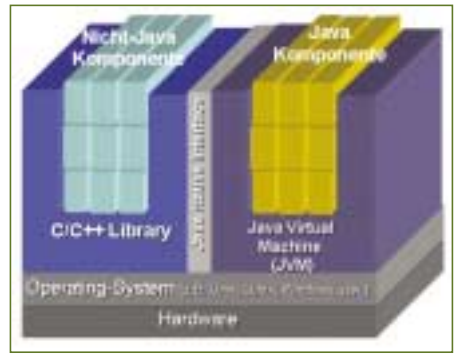

Abb. 1: Java Native Interface (JNI) als Bindeglied zwischen der C++- und der Java-Welt

Vor dem Aufruf eines Java-Diensts aus C/C++ heraus, steht der Start der JVM. Dabei muss die Java-Anwendung bzw. -Komponente im Klassenpfad enthalten sein. Mit dem Zeiger, der auf das JVM-Objekt verweist, lassen sich Java-Klassen instanzieren und auf dieser Basis Dienste aufrufen. Das Vorgehen ist eine Art Fernsteuerung der JVM. Listing 1 zeigt ein Beispiel für das Starten der JVM.

Über Handles (Zeigervariable *pENV*) lassen sich Java-Elemente (Klassen, Methoden usw.) identifizieren und aktivieren (Listing 2). Anhand des Codes in Listing 3 kann die Java-Klasse *VersicherungsVertrag* über JNI-Methode *Findclass()* gesucht und geladen werden. Wenn die Klasse über den bekannten Klassenpfad gefunden ist, liefert die JNI-Methode *Findclass()* einen gültigen Zeiger auf die Java-Klasse. Ansonsten wird der Null-Zeiger zurück geliefert.

Der Umgang mit Handles ist u.a. für die Anwendungsentwicklung zu fehleranfällig. Deswegen sollte die Anwendung frei von dieser Technik sein, woraus sich die Notwendigkeit einer Kapselung der JNI-Funktionalität ergibt. Das Konzept zur Kapselung der JNI-Funktionalität über das Entwurfsmuster Bridge wird hier kurz erläutert.

Die Kernidee beim Bridging von JNI-Funktionalität besteht in der Bereitstellung einer Schnittstelle, die unabhängig von den sie nutzenden Java-Anwendungen JNI-Funktionalität so kapselt, dass jeder Anwendungsclient die Dienste aus der Java-Welt aufrufen kann. Die Nutzung der Schnittstelle erfordert keine Kenntnisse der JNI-Programmiertechnik (Abb. 2).

Die Abstraktions- und Implementierer-Ebene bilden das Bridge-Muster zur JNI-Kapselung. Die Interaktion mit der

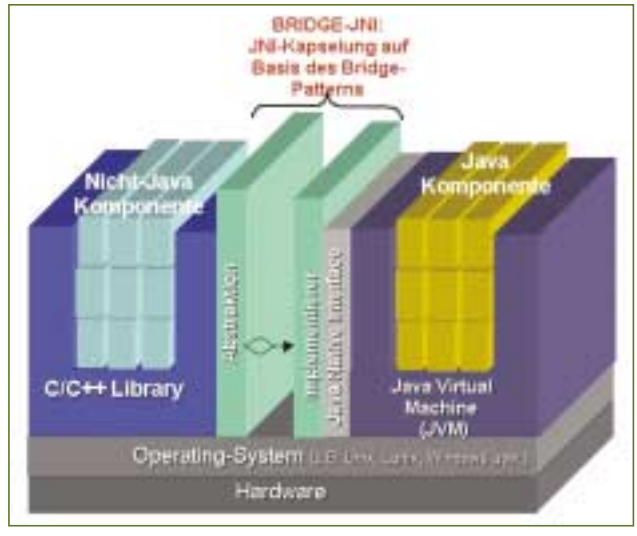

Abb. 2: Architektur basierend auf dem „Bridge"-Muster zur Kapselung von JNI-Funktionalitäten

Anzeige

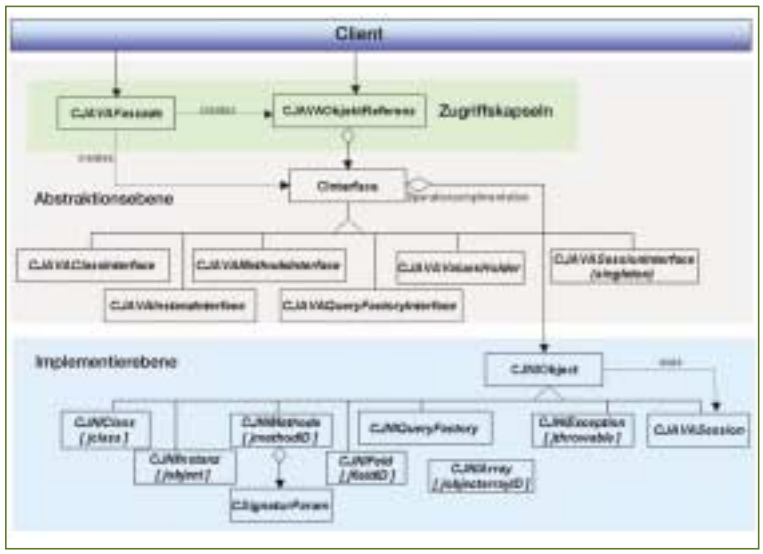

Abb. 3: Ein Snapshot des Klassendiagramms für die Abstraktions- und Implementierer-Ebene

JNI-Funktionalität findet unabhängig von der Logik der nicht-Java Komponente innerhalb der Implementierer-Ebene statt. Darin werden folgende Funktionalitäten gekapselt:

- Starten der JVM: Diese Softwarearchitektur sieht ein klassenbasiertes Erzeugungsmuster vor, das diese Aufgabe übernimmt. Ein solches Muster stellt Fabrikmethoden bereit, welche die Instanzierung und das Starten einer JVM ermöglichen. Um ein mehrfaches Instanzieren und Starten einer JVM zu vermeiden, ist diese Klasse als Singleton umgesetzt. Diese Funktionalität wird in der Klasse *CJAVASession* umgesetzt (Abb. 3).
- Zugriff auf Instanz und Klassenvariable sowie manipulieren von Java-Objekten und Konvertierung von Datentypen (Marshalling/Unmarshalling): Um die Verwaltung von JNI-Variablen (Handles) sicherzustellen, ist für jeden JNI-Objekttyp (*jobject*, *jclass*, *jfieldID*, *jmethodID*, *jobjectarray* und *jthrwoable*) eine Stellvertreterklasse vorgesehen (Abb. 3). Jede Instanz dieser Klassen nimmt Anfragen entgegen, transformiert sie in JNI-Funktionsaufrufe und ruft diese JNI-Funktion auf. Falls notwendig, werden Konvertierung zwischen den verschiedenen Typwelten durchgeführt.
- Lokalisieren und Instanzieren einer Java-Klasse innerhalb eines Packages: Hier werden das Lokalisieren und die Umwandlung von JNI-Fehlern in C++-Ausnahmen anwendungsübergreifend gelöst. Die Klasse, die hierfür vorgesehen ist, stellt Instanzmethoden zur Verfügung, die das Setzen von Suchkriterien (z.B. Klasse-Bezeichnung) ermöglichen.
- Lokalisieren und Aufruf einer Java-Methode (Static und Instanz): Eine Klasse, die das Suchen von Java-Methoden sowie das Aufbereiten der gefundenen übernimmt, stellt Instanzmethoden zur Verfügung, die das Setzen von Signaturen erleichtert (z.B. *setStringParameter()*). Tabelle 1 zeigt ein Mapping zwischen Java-Typen und deren JNI-Signaturen. Bei einer direkten Verwendung solcher Signaturen sind Fehler vorprogrammiert.

Java Datentyp	Signatur
Boolean	Z
Byte	B
Char	C
Short	S
Int	I
Long	L
Float	F
Double	D
Void	V
Objekte	Lklassenpfad;
Type[]	[type
Konstruktor	<init>
Methode type	(arg-types)return-type

Tab. 1.: Mapping zwischen Java-Typen und deren JNI-Signaturen

Die Abstraktions-Ebene ist frei von JNI-Funktionalitäten und bietet eine Zugriffsschicht, die konform zum C++-Programmiermodell ist. In dieser Ebene sind keine Kenntnisse über die JNI-Umsetzungslogik notwendig. Deshalb garantiert diese Ebene die Funktionalitäten:

- *Starten einer JVM unter optionaler Angabe eines Klassenpfads*: Zugriff auf Java-Klassen über den Zugriffspfad (Package/Klassenbezeichnung) und auf Klassenvariablen über Variabelennamen.
- *Aufruf von Methoden (auch überladene Methoden) über Methodennamen*: Klassenspezifische Funktionalität – wie das Erzeugen einer Instanz – muss robust auf Programmierfehler reagieren.

Abbildung 3 zeigt ein Klassendiagramm (Ausschnitt) der Abstraktionsebene, wobei folgende Funktionalitäten verwendet werden:

- *CJAVAFassade* bietet Funktionalitäten zum Starten einer JVM und zum Finden einer Klasse über Namen innerhalb einer Java-Welt. Sie dient auch als Factory für Instanzen der Klasse *CJAVAQueryFactoryInterface*. Jede Methode dieser Klasse liefert einen Zeiger auf die Instanz einer Spezialisierung der Klasse *CInterface*. Dieser Zeiger ist eine Instanz der Klasse *CJAVAObjektReferenz*.
- *CJAVAClassInterface* bietet Dienste zur Erzeugung einer Instanz, zum Aufruf von Klassenmethoden und zur Manipulation von Klassenvariablen an.
- *CJAVAInstanzInterface* bietet Dienste zum Aufruf von Instanzen-Methoden und zur Manipulation von Instanzvariablen.
- *CJAVAQueryFactoryInterface* stellt Dienste zum Finden einer bestimmten Klasse oder Methode zur Verfügung. Eine gefundene Klasse wird als Instanz der Klasse *CJAVAClassInterface* zurückgeliefert. Für eine Methode wird eine Instanz der Klasse *CJAVAMethodInterface* bereitgestellt.
- *CJAVAMethodeInterface* füllt einen Container – als Instanz der Klassen *CJAVAValuesHolder* – mit aktuellen Werten von Übergabeargumente der Methoden. Dieser Container prüft den Datentyp des aktuellen Wertes gegenüber dem spezifizierten Datentyp von Übergabeargumenten.
- *CJAVAObjektReferenz* ist als *smartPointer* umgesetzt, um einen einheitlichen

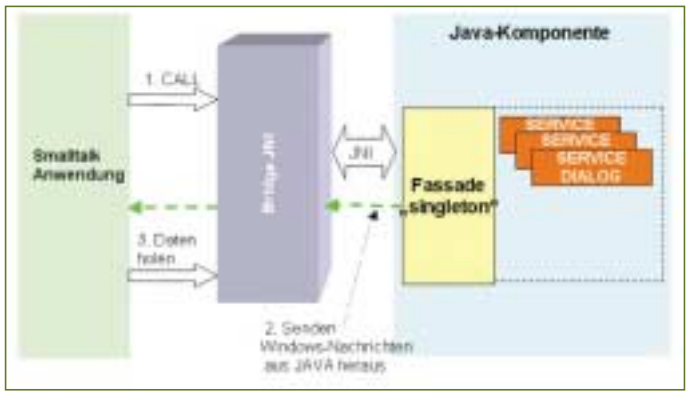

Abb. 4: Kommunikationsmuster auf Basis des BridgingJNI

Zugriff auf Instanzen der Klassen *CJAVAClassInterface*, *CJAVAInstanzInteface*, *CJAVAQueryFactoryInterface* und *CJAVA MethodInterface* sicherzustellen.

Listing 4 zeigt einen Aufruf einer Instanzmethode über BridgeJNI-Funktionalität. Wir betrachten das Beispiel aus Listing 2. Diese Java-Klasse wurde in *BridgingJNIbeispiel.jar* abgelegt.

Interaktion zwischen Smalltalk und Java auf Windows-Plattformen

Hier betrachten wir eine Smalltalk-Anwendung als eine nicht-Java-Komponente, die mit eine Java-Komponente über *BridgeJNI* auf Windows-Plattform kommuniziert.

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

- Wie kann eine Smalltalk-Anwendung eine Java-GUI-Komponente starten und verwalten?
- 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 (z.B. *beendet*), bekannt geben können.

Da wir hier auf der Windows-Plattform arbeiten, muss die Java-GUI-Komponente die Möglichkeit erhalten, das Windows-API zu bedienen. Um einen sicheren Aufruf von Java-Diensten aus der Smalltalk-Welt zu erreichen, verwenden wir die Bridge-basierte Softwarearchitektur, die im letzten Abschnitt angesprochen wurde.

Die Java-Komponente kann ihren Zustand über das Windows-API der Smalltalk-Anwendung bekannt geben. Zu diesem Zweck müssen die Windows-Funktionen (z.B.: *SendMessageTimeout()*) umhüllt (*wrap*) werden und aus Java heraus als native Funktionen aufgerufen werden.

Für diesen Fall ergibt sich folgendes Kommunikationsmuster, das sich grob in drei Teilen gestaltet (Abb. 4):

- Aufruf der Java-Komponente mit Übergabe des Windows-Handle des Smalltalk-Hauptfensters. Der Windows-Handle des Smalltalk-Hauptfensters repräsentiert den Sender, der Empfänger kann sich z.B. über diesen Handle beim Sender registrieren.
- Senden einer Window-Message aus Java heraus an Smalltalk mit Übergabe des in Java ermittelten Windows-Handles (Hauptfenster der Java-Komponente). Dieser Windows-Handle repräsentiert eine Java-Komponente, durch die der Sender (eine Smalltalk-Anwendung) in der Lage ist, den Empfänger (eine Java-Komponente) zu verwalten. Er kann ihm z.B. den Auftrag erteilen, sich zu beenden oder das Java-Hauptfenster in der Client-Area des Smalltalk-Hauptfensters zu rearrangieren.
- Datenaustausch zwischen Java und Smalltalk.

Um eine Kommunikation zwischen den Komponenten zu ermöglichen, müssen beide den Windows-Handle der anderen Komponente kennen. Java bekommt den Windows-Handle des Smalltalk-Fensters beim Starten der Java-Komponente über die Shared Library BridgeJNI mitge-

Listing 4

Aufruf einer Java-Instanzmethode über *BridgeJNI*

```
//1. starte JVM mit dem Klassenpfad
CJavaFassade* pFassade =
CJavaFassade::create("D:\\Beispiel
                          \\BridgingJNIbeispiel.jar; ");
Try
{
 If ( pFassade != 0)
 {

//2. Suche die Klasse "VersicherungsVertrag" innerhalb
//des Package "vertragPackage":
  CjavaObjektReferenz refVertragCls =
        pFassade->findClass("vertragPackage/
                                VersicherungsVertrag");

//3. Erzeuge eine Instanz der Klasse
  CJavaObjektReferenz refVertragInstanz =
                          refVertragCls->createInstanz();

//4. Erzeuge ein QueryFactory Instanz
    CJavaObjektReferenz refQueryInst =
                          pFassade->createQueryFactory();

//5. Setze den SuchKontext
//(innerhalb der Klasse "VersicherungsVertrag")
    refQueryInst -> setKontextClass (refVertragCls);
//6. Setze die Spezifikation für die formalen Parameter.
//Es geht hier um String-Parameter
    refQueryInst -> setStringParam ();

//7. Suche die Void-Methode
    CJavaObjektReferenz refVoidMethod =
        refQueryInst -> getVoidMethodID
                                ("setGesellschaft");

//8. Erzeuge eine Container für die aktuellen Werte
    CJavaObjektReferenz refValueHolders =
         refVoidMethod -> createValuesHolder();

//9. setze den aktuellen Wert
    refValueHolders -> addValue("Versicherer XY");

//10. ruf die Instanzmethode auf
    refVertragInstanz -> callVoidMethod
                          (refVoidMethod, refValueHolders);

  }//end-if
}//end-of-try
Catch(const CJNIException& rExcept){ /* do more than ... */}
```

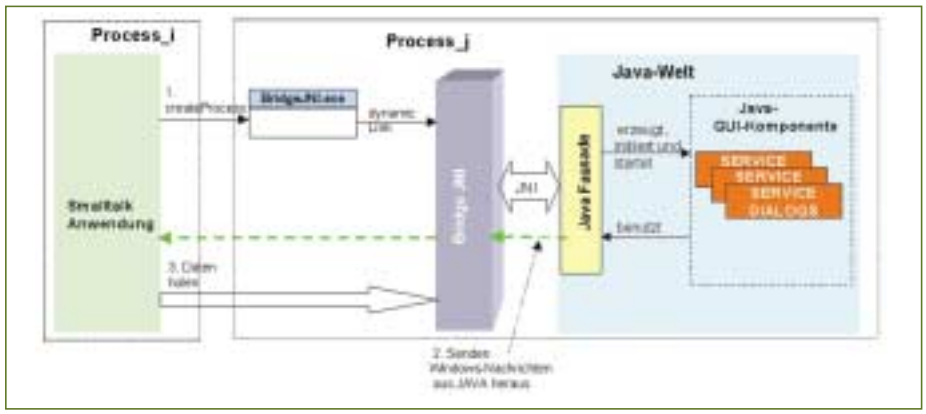

Abb. 5: Softwarearchitektur des Kommunikationsmusters mit zwei Prozessen

teilt. Die Fassade der Java-Komponente muss einen Setter (z.B. *setWindowHandle(long hWnd)*) vorsehen. Diese Methode wird von Smalltalk heraus mit Hilfe des BridgeJNI identifiziert und ausgeführt.

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

Die Lösung liegt auf der Hand [3]: Mittels JNI wird die Windows-Funktion *FindWindow()* aus Java heraus aufgerufen. Diese native Funktion sucht ein Top-Level-Fenster anhand des vorgegebenen Fensternamens und liefert den Windows-Handle zurück. D.h., anhand der Information des Fenstertitels (sollte in der Prozessliste eindeutig sein) der betreffenden Swing-GUI kann der Windows-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 Windows-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 es den Windows-Handle als Instanzvariable in einem Controller (MVC-Prinzip) ab und kann damit auch dem Swing-GUI Nachrichten senden und somit die Java-Komponente verwalten.

Um Smalltalk über das Schließen der Java-Komponente zu informieren, wird z.B. die Nachricht WM_ QUIT über die Windows-Funktion *PostMessage()* gesendet. Diese stellt eine Nachricht in die Warteschlange für den Thread, der das vorgegebene Fenster erzeugt hat, und kehrt zurück, ohne auf eine Antwort zu warten [6].

Listing 5

Java-Spezifikation der Klasse *messageDispatcher*

```
public class messageDispatcher{
 static{
    try{
      System.loadLibrary("BridgeJNI");
    }
    catch(java.lang.UnsatisfiedLinkError ex){
     System.exit(0);
    }
    catch(java.lang.Throwable ex){
     System.exit(0);
    }
 }//end-of- static Block

 public native boolean nPostMessage
   (long hwnd,int uMsg,long wParam,long lParam);
 public native boolean  nSendMessageCOPYDATATimeout
     (long hwnd, int Pumsg, long Pwparam,
                              long Plparam, int PuTimeout);
 public native boolean nSendMessage
                              COPYDATATimeoutWithString
     (long hwnd, int Pumsg, long Pwparam, long Plparam,
                  int PuTimeout,String PstrData);
 public native long nFindWindow(String PjstrWindowTitle);
}
```

Folglich wird für die Kommunikation von Java-Seiten neben der Fassaden-Klasse auch eine *messageDispatcher*-Klasse (Listing 5) benötigt. Diese Klasse kapselt alle benötigten Windows-Funktionen und deklariert diese als native Methoden. Die Umsetzung der nativen Methoden sind in der C++-Komponente *BridgeJNI.dll* gekapselt.

Sichere Kommunikation 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 VM 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()*. 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 (Abb. 5).

Interaktion zwischen Smalltalk und nicht-Java GUI-Komponenten

EJB-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 hinaus genutzt werden. Also können sie plattformübergreifend kommunizie-

Anzeige

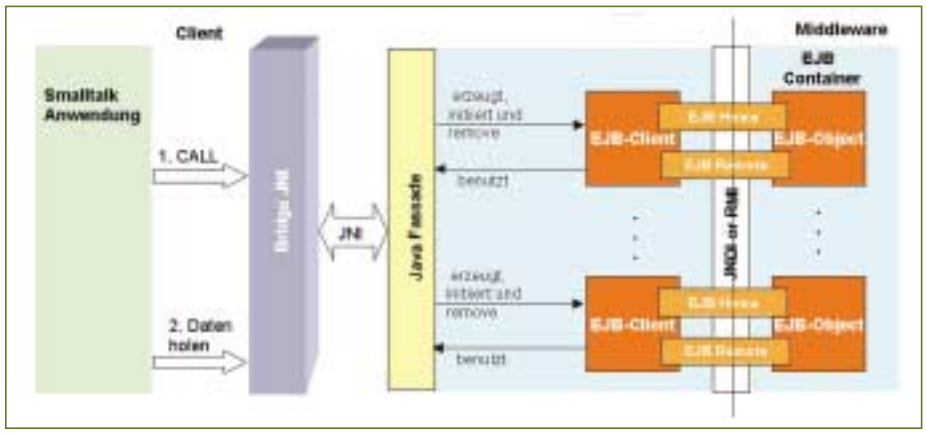

Abb. 6: Softwarearchitektur des Kommunikationsmusters für verteilte Objekte

Listing 6

Smalltalk-Instanzmethode zum Dispatchen des Events *PlatformConstants::WmCopydata*

```
wmCopyData: wParam with: lParam
    | cds |
"Dispatching wird nur durchgeführt,"
"wenn die Shell eine XCommClient-Shell ist"
    self owner rootWidget userData parentPart
                                        isXCommClient ifTrue:[
    "Die Struktur WM_COPYDATASTRUCT
                                (in VAST OSCopydatastruct) wird"
    "mit den im externen Speicher stehenden Daten befüllt"
    cds := ( OSCopydatastruct address: lParam )
                                    abtMoveFromOSMemory.
    self owner rootWidget userData parentPart owner log:
        'Message WM_COPYDATA received with params
                                    dwData:', cds dwData
      printString, ', lpData: ', ((( OSPtr address: cds lpData )
      memcpyFrom: 0 to: cds cbData )asString
      replaceAll: CldtConstants::Nul with: $).
    "Meldung in Statuszeile ausgeben,"
    "damit der Benutzer von der Aktion Notiz nimmt"
    self owner rootWidget userData parentPart owner
    hint: 'Message WM_COPYDATA received with
                                    params dwData:',
    cds dwData printString, ', lpData: ',
                                (((OSPtr address: cds lpData )
    memcpyFrom: 0 to: cds cbData ) asString
    replaceAll: CldtConstants::Nul with: $ ).
    "Hier der wichtigste Punkt: ist die Nachrichtenid 1029"
    "(userdefined Message + 5), ist der Windowhandle von "
    "Java enthalten und wird"
    "in der Instanzvariable serverHwnd gespeichert"
    cds dwData = 1029 ifTrue:[
        self owner rootWidget userData parentPart owner
      serverHwnd: ( OSHwnd value: ((OSPtr address:
                                        cds lpData )
      memcpyFrom: 0 to: cds cbData ) asString asNumber ).
        self owner rootWidget userData parentPart owner
                                    createOSShell.].
    ].
    ^0
```

ren. Auch hier müssen u.a. folgende Fragen beantwortet werden:

- Wie kann eine Smalltalk-Anwendung Dienste einer EJB-Komponente nutzen?
- 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 unser Muster Verwendung (Abb. 6). 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 Client und EJB-Komponenten Home- und Remote-Schnittstelle, wegen einheitlicher 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.

Listing 7

Ausführung der Event-Hook

```
loaded

    "Eventhook für die Nachricht WmQuit"
    ( OSWidget
        eventTableAt: PlatformConstants::WmQuit ) ifNil:[
            OSWidget
                eventTableAt: PlatformConstants::WmQuit
                    put: #wmQuit:with:.].
    "Eventhook für die Nachricht WmCopydata"
    ( OSWidget
        eventTableAt: PlatformConstants::WmCopydata )
ifNil:[     OSWidget
                eventTableAt: PlatformConstants::WmCopydata
                    put: #wmCopyData:with:].
```

Message Dispatching in Smalltalk

Um in Smalltalk Nachrichten aus Java empfangen zu können, müssen so genannte Event-Hooks etabliert werden. Diese stellen eine Verankerung im Code dar, der dann betreten wird, wenn das mit dem Event-Hook vereinbarte Event ausgelöst wird und der Empfänger (immer ein gültiger Windows-Handle) dem Prozess zugehörig ist, in dem der Code 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 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.

Die entsprechende Methode überschreibt die Methode an *OSWidget*, die keine Aktion durchführt (Listing 6).

Listing 7 zeigt den eigentlichen Event-Hook. Dieser wird in der Methode *loaded* des zugehörigen Packages ausgeführt. Diese Methode wird gesendet, wenn das Package von der Smalltalk-Entwicklungsumgebung geladen wurde. Hier sieht man, dass im *eventTable* von *OSWidget* das entsprechende Event eingefügt wird, falls dieses nicht schon registriert wurde.

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. In der linken Icon-Fläche lassen sich die Services starten (Abb. 7). Die Konfiguration der Services werden zur Laufzeit aus einer XML-Datei gelesen und in der Icon-

Fläche dargestellt. Wie auch die Java-Services muss der externe Smalltalk-Service *ODBCSpy* die Methode *setWindowHandle(long hWnd)* implementieren, um den Windows-Handle des Smalltalk-Wirtsfensters zu erhalten.

Durch die beim Start der Java-Komponente gegenseitige Registrierung der Windows-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 (Fenster) möglich ist, kann auch das Java-Hauptfenster völlig unter die Kontrolle des Smalltalk-Fensters gebracht werden. Am deutlichsten kann man das anhand einer Vater-Kind-Beziehung machen, in der 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 (Abb. 7). Wird der Prozess beendet, lässt sich dieser erneut Starten und die Java-Komponente öffnet sich wieder.

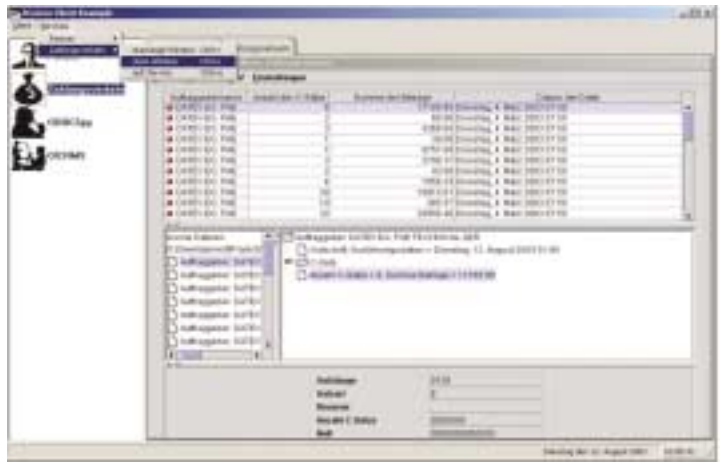

Abb. 7: Das Beenden des Java-Fensters (dispose())

Eine UNIX-Lösungsskizze

Unter Unix muss die Möglichkeit geschaffen werden, die Kommunikation zwischen Java und Smalltalk in der Library Bridge-JNI zu definieren. D.h., die Strategie des Bridging von JNI bleibt erhalten, nur die Implementierung der nativen Funktionen für den Datenaustausch und Nachrichtenaustausch sind verschieden. Die Implementierung dieser nativen Funktionen könnten von der Parametrisierung und der Namensgebung plattformunabhängig gestaltet werden, sodass die Behandlung der unterschiedlichen Plattformen (Unix und Windows) abstrahiert wird. Ein Austausch des Windows-Handles fällt unter Unix weg. Die Implementierung dieser Funktionalitäten bleibt in der Shared Library (Bridge-JNI) gekapselt.

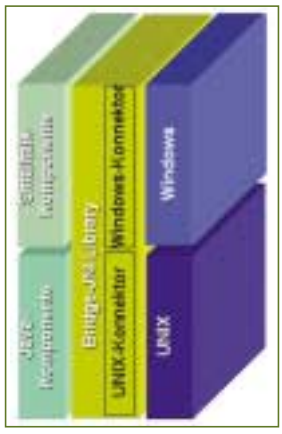

Abb. 8: Abstraktion der nativen Betriebssystemfunktionen durch die Bridge-JNI Library

Für den Datenaustausch eignet sich u.a. die Message-Queue und für den Nachrichtenaustausch Signals. Die Implementierung der Java- und Smalltalkkomponenten bleibt transparent.

Um eine Java-Komponente von Smalltalk aus zu kontrollieren, würde unter Unix die Smalltalk-Anwendung eine Message-Queue anlegen und die Java-Komponente mit der ID der Message-Queue als Parameter starten. Darauf hin kann die Java-Komponente Daten mit der Smalltalk-Anwendung über die Message-Queue austauschen. Über Signals lassen sich vorher festgelegte Nachrichten austauschen, mit denen man den Nachrichtenfluss und den Ablauf der Komponenten steuern kann. Somit wird es z.B. möglich, die Anfrage des Vater-Prozesses (Smalltalk) nach Beenden der Java-Komponente in Java durch ein Veto zu verhindern, da es ansonsten zu einem inkonsistenten Zustand kommen könnte. Abbildung 8 zeigt die Rolle des Bridge-JNI innerhalb der beiden Plattformen.

Die Unterschiede der Plattformen sind sauber in der Bridge-JNI Library gekapselt. D.h., es gibt keine weiteren Änderungen in den Java- bzw. Smalltalkkomponenten. Diese Plattformabhängigkeiten werden in Konnektoren innerhalb der Bridge-JNI abstrahiert.

Fazit

Die vorgestellte Softwarearchitektur ermöglicht eine generische Interoperabilität zwischen der nicht-Java- und der 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-Verfahren verwenden [2], um diese Art von Kommunikation umzusetzen. Leider lassen sie sich nur schwierig und nicht ohne eine Erweiterung bzw. Anpassung der bestehenden Komponenten anwenden.

Darüber hinaus hat dieser Artikel gezeigt, wie Betriebssystemfunktionen, welche in Java als nativen Methoden aufgerufen werden, in der *BridgeJNI-Library* plattformunabhängig gekapselt werden können. Dadurch wird eine kostengünstigere und plattformübergreifende Kommunikation zwischen Smalltalk und Java geschaffen.

Links & Literatur

[1] Dr. Ndombe Cacutalua und Joachim Korittky: „Sicherer Aufruf von Java-Diensten aus der C++-Welt auf Basis des Bridge-Pattern“, Objektspektrum 05/2002

[2] VisualAge Java-RMI-Smalltalk, „the ATM Sample from A to Z”: www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245418.pdf

[3] Sun Microsystems; Lesson: „Writing a Java Program with Native Methods”, java.sun.com/docs/books/tutorial/native1.1/stepbystep/

[4] Sun Microsystems: „JNI –JAVA Native Interface”

[5] Sun Microsystems; „JNI-Jan Native Interface Tutorial”

[6] Richard J. Simon: „Win32 Programming API Bible”, SAMS, Macmillan Computer Publishing, USA

[7] Stephen Asbury und Scott R Weiner: „Developing Java Enterprise Applications”, Wiley 1999