Automatische Vermittlung zwischen Interaktionsstilen

Max Haustein

17. Oktober 2003

Zusammenfassung

Die Wiederverwendung von Softwarekomponenten wird häufig durch Inkompatibilitäten behindert. Ein Problem dabei sind unterschiedliche Interaktionsstile. Es soll ein System entwickelt werden, das automatisch zwischen verschiedenen Stilen vermitteln kann.

1 Einführung

Das Ziel wiederverwendbarer Softwarekomponenten wird seit langer Zeit verfolgt. Die Idee, eine benötigte Komponente anhand geeigneter Suchkriterien zu finden und mit wenigen Handgriffen in das eigene System einzubinden, ist verlockend. Eine solche Wiederverwendung erscheint ökonomisch selbstverständlich, sie ist aus anderen Ingenieurswissenschaften nicht wegzudenken. Darüber hinaus werden Softwaresysteme zunehmend komplexer, so dass eine Neuentwicklung aller Bestandteile von Grund auf im Rahmen eines vertretbaren Zeitplans gar nicht mehr möglich ist [10]. Es liegt also nahe, dass zukünftig vermehrt auch in der Softwaretechnik auf zunehmend höherer Ebene existierende (*of-the-shelf-*) Komponenten miteinander verknüpft und um neue Bestandteile ergänzt werden, um ein neues Systems zu implementieren [4].

In der Praxis ergeben sich bei der Wiederverwendung von Softwarekomponenten jedoch erhebliche Probleme, die gut und gerne den erwünschten positiven Effekt mehr als zunichte machen können [5]. Derartige Probleme haben bisher die Entwicklung von umfangreicheren Komponentenbibliotheken oder -marktplätzen erheblich gebremst und die Idee der Wiederverwendung hat bei einigen deutlich an ihrer ursprünglichen Attraktivität verloren. Für Abhilfe sollen einheitliche Komponentenmodelle sorgen, aber die Entwicklung eines weltweit einheitlichen Modells ist eher unwahrscheinlich und wohl genauso wenig wünschenswert oder sinnvoll, so dass auf absehbare Zeit mit Heterogenität zu rechnen ist. Promiente Vertreter solcher Komponentenmodelle sind derzeit die Enterprise Java Beans von Sun, das Corba Component Model oder Microsofts COM und .NET.

2 Interaktionsstile und Schnittstellenbeschreibung

Ein grundlegendes Problem bei der Koppelung von Komponenten, auf das hier näher eingegangen werden soll, ist die Verwendung unterschiedlicher Interaktionsstile. Verbreitet sind Paradigmen wie die aufrufbasierte Kommunikation, Pipe-and-Filter-Architekturen, ereignisbasierte Systeme oder die Auswertungsmechanismen funktionaler Programmiersprachen. Komponentenmodelle sind meist auf einen bestimmten Interaktionsstil festgelegt oder unterstützen allenfalls zwei (zur Zeit verbreitet etwa alternativ aufruf- oder ereignisbasierte Kommunikation). Die Koppe-

lung von Komponenten mit unterschiedlichen Interaktionsstilen erfolgt meist in individuellen Ad-hoc-Lösungen.

Um die Freiheit bei der Komponentenauswahl zu vergrößern, soll ein System entwickelt werden, das automatischen zwischen verschiedenen Interaktionsstilen vermitteln kann, wie bereits in [7] und [8] beschrieben. Das System soll, basierend auf einer abstrakten Schnittstellenbeschreibung (*Abstract Interface Definition - AID*), entsprechende Vermittler generieren. Die Schnittstellenbeschreibung einer Komponente kann auch nachträglich verfasst werden, so dass bereits vorhandene Komponenten ohne deren Modifikation eingebunden werden können. Sie ist insofern abstrakt, als sie nicht nur von der Implementierung sondern auch vom Interaktionsstil der Komponente abstrahiert.

Die Beschreibung erfolgt in Form von Ein-/Ausgabeereignissen an der Schnittstelle der Komponente, die jedoch nicht mit einer ereignisbasierten Kommunikation zu verwechseln ist. Die verwendeten Ereignisse sind auf einer konzeptionell niedrigeren Ebene angesiedelt und beschreiben lediglich das Auftreten eines Informationsflusses an der Komponentenschnittstelle. Die Beschreibung eines solchen Ereignisses umfasst Initiator, Richtung, eine Bezeichnung und die Typen der Daten, die im Zuge der Verarbeitung des Ereignisses übertragen werden.

Bezüglich der Typen der übertragenen Daten sind bisher lediglich primitive Datentypen und Zeichenkette vorgesehen. Dabei, und insbesondere bei den noch zu integrierenden komplexeren Datentypen, ergeben sich die üblichen Probleme von Schnittstellenbeschreibungen, auf die hier nicht näher eingegangen werden soll.

Durch Initiator und Richtung wird die Aktivität der Interaktionspartner beschrieben. Die Frage ist, welcher der Partner das Ereignis auslöst. Die Partner können sich dabei aktiv oder passiv verhalten. Betrachtet man die Kommunikation zwischen Umgebung und Komponente, so kann der Informationsfluss entweder in die Komponente hinein oder aus ihr heraus gerichtet sein und von der Umgebung oder von der Komponenten ausgelöst werden. Die jeweilige Ausprägung eines Ereignisses an der Schnittstelle wird mit einer Kombination aus *push*/*pull* und *in*/*out* beschrieben.

2.1 Objektorientierte Aufrufe

Ein Methodenaufruf in der objektorientierten Programmierung, bei dem Daten in Form von Parametern an das Objekt (die Komponente) übergeben werden, stellt sich in der abstrakten Schnittstellenbeschreibung als *push in*-Ereignis dar. Besitzt die Methode einen Rückgabewert so gibt es nach Abarbeitung des Methodenrumpfes ein weiteres Ereignis an der Schnittstelle: Das vom Aufrufer angeforderte Ergebnis wird zurückgeliefert. Es handelt sich um ein *pull out*-Ereignis. Eine Java-Komponente mit der (konkreten) Schnittstelle

```
public interface Namensdienst {
    public void anmelden( String name, String adresse );
    public String suchen( String name );
}
```

würde durch folgende abstrakte Schnittstelle beschrieben werden:

```
interface Namensdienst
push in   anmelden String String
push in   suchen String
pull out  suchen String
end
```

Interessant ist hier vor allem, dass die Methode `anmelden` in zwei Ereignissen an der abstrakten Schnittstelle resultiert. Die Namensgleichheit [1] impliziert eine semantische Zusammengehörigkeit der Ereignisse: Mit Auftreten des ersten Ereignisses wird eine Verbindung zwischen Umgebung und Komponente aufgebaut, die durch das zweite Ereignis wieder beendet wird, entsprechend des synchronen Methodenaufrufs, der durch die Java-Schnittstelle beschrieben wird.

2.2 Byte-Ströme

Ein Unix-Programm, das Daten von der Standardeingabe liest (ein Filter im Sinne der Pipe-and-Filter-Architektur), erzeugt dabei *pull in*-Ereignisse an seiner abstrakten Schnittstelle. Die „Komponente“ liest gerade so viele Daten, wie im Programmverlauf benötigt werden und schreibt Ausgaben auf die Standardausgabe, generiert also *push out*-Ereignisse.

Beachtung erfordert die Vielfalt, mit der Unix-Programme die Standardein- und -ausgabekanäle nutzen. Einige Programme arbeiten mit dem reinem Byte-Strom der Standardeingabe, so etwa Kompressionsprogramme. Sie lesen die Daten, ohne sie näher zu interpretieren. In diesem Fall bietet sich eine einfache abstrakte Schnittstelle wie die folgende an:

```
interface Gzip
pull in   stdin  byte
push out  stdout byte
push out  stderr String
end
```

Wie schon bei der Standardfehlerausgabe angedeutet, liegt den Bytes jedoch oft eine Semantik zugrunde, die sich in einer individuellen Schnittstelle ausdrücken lässt, die über die drei Standard-Byte-Ströme hinaus geht. Viele Programme arbeiten textbasiert und lesen Daten zeilenweise ein, z.B. `grep` oder `wc`. Ein Ereignis könnte also im Lesen oder Schreiben einer ganzen Zeile und nicht nur eines Bytes bestehen. Wieder andere Programm arbeiten interaktiv, wie etwa eine Shell. Sie könnte eingebaute Befehle an ihrer abstrakten Schnittstelle anbieten. Letztlich steht und fällt beinahe jedes Unix-Programm mit seiner Konfigurierbarkeit über Kommandozeilenparameter. Auch hier wäre es denkbar, weitere Funktionalität an der abstrakten Schnittstelle zur Verfügung zu stellen.

2.3 Ereignisbasierte Kommunikation

Wie schon erwähnt ist zu beachten, dass die Ereignisse zur Beschreibung der Schnittstelle nicht mit denen in einem ereignisbasierten System zu verwechseln sind. Eine Komponenten in einem solchen System, die das Auftreten eines Ereignisses signalisiert, löst an ihrer Schnittstelle ein *push out*-Ereignis aus.

Viele ereignisbasierte Systeme arbeiten mit unterschiedlichen Ereigniskanälen, bei denen sich einzelne Teilnehmer des Systems registrieren, wenn sie an den dort veröffentlichten Ereignisses interessiert sind. Beispiele sind die *Topic*s von JMS [11] oder die *Event Channel*s des CORBA Notification Services [6]. Andere, wie etwa Siena [2], verlassen sich allein auf die Definition von Filtern, um die Auswahl der empfangenen Nachrichten einzuschränken. Der Empfang von Nachrichten mit einer bestimmten Semantik hängt also im einen Fall vom Kanal ab, über den die Nachricht empfangen wird, im anderen Fall von den Parametern der Nachricht, über denen

[1] Die Zusammengehörigkeit der beiden Ereignisse wird derzeit nur über die gleiche Bezeichnung ausgedrückt, andere Lösungen sind aber denkbar, etwa um Konflikte bei überladenen Methoden zu umgehen.

die Filter definiert werden. Solche Nachrichten unterschiedlicher Semantik würden sich in der abstrakten Schnittstelle als unterschiedliche Operationen niederschlagen.

2.4 Funktionale Programmierung

Die bedarfsgetriebene Erzeugung und Auswertung partieller Listen in funktionalen Programmiersprachen führt zu *pull in*/*pull out* Ereignissen an der Schnittstelle einer entsprechenden Komponente.

Durch die Verwendung von Monaden kann eine Möglichkeit zur Interaktion mit funktional entwickelten Komponenten implementiert werden, die den oben beschriebenen textbasierten oder interaktiven Unix-Programmen recht nahe kommt. Über diesen „Umweg“ lässt sich eine abstrakte Schnittstelle für solche Komponenten definieren.

3 Vermittlung

Die Vermittler gliedern sich in zwei Bestandteile: ein Stellvertreter(*proxy*) repräsentiert die Komponente gegenüber dem Klienten in dessen Umgebung und kommuniziert über ein Vermittlungsprotokoll mit einem Treiber (*driver*), der die Komponente in ihrem „gewohnten“ Interaktionsstil steuert. Somit muss nur für jeden Interaktionsstil eine Abbildung auf das Vermittlungsprotokoll implementiert werden, um beliebige Komponenten miteinander zu koppeln.

Die Vermittlung wird erschwert, wenn Umgebung und Komponenten inkompatibel bezüglich ihres *push*/*pull*-Verhaltens sind. Soll eine Unix-Komponente in einer Umgebung, in der ereignisbasiert kommuniziert wird, eingesetzt werden und soll die Unix-Komponente auf bestimmte Ereignisse reagieren, so stellt die Veröffentlichung des Ereignisses ein *push out*-Ereignis an der abstrakten Schnittstelle dar, das mit einem *pull in*-Ereignis auf der Seite der Unix-Komponente verknüpft werden soll. Beide Interaktionspartner wollen die Informationsübertragung aktiv auslösen, so dass im Rahmen der Vermittlung ein Pufferfunktionalität erforderlich ist. Sind hingegen beide Partner passiv, sollen etwa die Elemente einer Liste, die von einer Haskell-Komponente berechnet werden (*pull out*), als Ereignisse in einem ereignisbasierten System veröffentlicht werden, so bedarf es einer aktiven Instanz. Ohne eine solche Instanz würde es nie zu einem Informationsfluss kommen. Auch sie muss von der Vermittlungsschicht gestellt werden.

4 Verwandte Arbeiten

Allan und Garlan haben die Interaktion zwischen Komponenten formal beschrieben und Konnektoren entwickelt, die Transformationen zwischen den von den Komponenten verwendeten Protokollen vornehmen [1]. Garlan und Spitznagel haben das Konzept der Wrapper näher beleuchtet [13] und gehen dabei auf Spezifikation, Einfluss auf die Komponente, die Verknüpfung von Spezifikation und Implementierung und auf die Wiederverwendung von Wrappern ein. Yellin und Strom haben beschrieben, wie Komponentenschnittstellen um Protokollangaben erweitert werden können und mit Hilfe von Zustandsautomaten für die Komponenten deren Kompatibilität untersucht [15]. Basierend auf den erweiterten Schnittstellen wurden halbautomatisch Adapter generiert. DeLine hat umfangreiche Untersuchungen zu Inkompatibilitäten zwischen Komponenten durchgeführt [3]. Pryce beschreibt die Spezifikation von Interaktionsstilen für Komponenten in verteilten Systemen unabhängig von der Implementierung und der dort verwendeten Programmiersprache [12]. Mehta, Medvidovic und Phadke haben eine umfassende Taxonomie von Softwarekonnektoren aufgestellt und beschreiben deren Eigenschaften und Ausprägungen [9]. Damit

ist eine ausführliche Beschreibung verschiedener Interaktionsstilen verbunden. Yakimovich, Bieman und Basili haben Inkompatibilitäten bezüglich der Interaktion zwischen Komponenten klassifiziert und ein System aufgestellt, das dabei helfen soll, die Integrationskosten für Komponenten abzuschätzen [14].

5 Ausblick

Im Hinblick auf die Praxistauglichkeit ist eine Einbeziehung verbreiteter Komponentensysteme von entscheidender Bedeutung, darunter Suns Enterprise Java Beans (EJB), Microsofts COM und .NET, das Corba Component Model (CCM) und die Open Services Gateway Initiative (OSGi).

Das Verhältnis zu Architekturbeschreibungssprachen muss näher betrachtet werden. Bindung und das Aufsetzten eines lauffähigen Systems erfolgen bisher ad-hoc. Dabei muss auch die Konfigurierbarkeit von Komponenten ermöglicht werden.

Wie bereits erwähnt beschränkt sich das Typsystem bisher auf primitive Datentypen und Zeichenketten. Die Verwendung komplexere Datentypen bei der Schnittstellenbeschreibung steht noch aus. Die Abbildung von der konkreten auf die abstrakte Schnittstelle wurde z.T. nur angedeutet. Die Schnittstellendefinition soll wesentlich flexibler werden, um etwa mehrere abstrakte Operationen auf eine einzelne parametrisierte konkrete abbilden zu können.

Die Nebenläufigkeit zwischen Klienten einer Komponente muss auf geeignete Art und Weise definiert werden.

Literatur

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings: 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press / ACM Press, 1994.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[3] R. DeLine. A catalog of techniques for resolving packaging mismatch. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 44–53, 1999.

[4] D. Garlan. Software architecture: a roadmap. In *ICSE - Future of SE Track*, pages 91–101, 2000.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings: 17th International Conference on Software Engineering*, pages 179–185. IEEE Computer Society Press / ACM Press, 1995.

[6] O. M. Group. Corba notification service. `http://www.omg.org/technology/documents/formal/notificationservice.htm`.

[7] K.-P. Löhr. Towards automatic mediation between heterogeneous software components. In E. Pulvermueller, I. Borne, N. Bouraqadi, P. Cointe, and U. Assmann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[8] K.-P. Löhr. Automatic mediation between incompatible component interaction styles. In *36th Annual Hawaii International Conference on System Sciences (HICSS-36)*, 2003.

[9] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187. ACM Press, June 2000.

[10] B. Meyer. Component and object technology: On to components. *Computer*, 32(1):139–140, Jan. 1999.

[11] S. Microsystems. Java Message Service API (JMS). `http://java.sun.com/products/jms/`.

[12] N. Pryce and S. Crane. Component interaction in distributed systems. In *Fourth International Conference on Configurable Distributed Systems*, pages 71–78. IEEE Press, May 1998.

[13] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 374–384, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[14] D. Yakimovich, J. M. Bieman, and V. R. Basili. Software architecture classification for estimating the cost of COTS integration. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 296–302. IEEE Computer Society Press / ACM Press, 1999.

[15] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, Mar. 1997.