SIPaDIM – Assistenz durch selbstbeschreibende Software

Peter Reiß, Günther Görz

{Reiss, Goerz}@Forsip.de

Abstract: Technische Systeme bieten immer größeren Funktionsumfang, ihre Bedienung wird zu einer Herausforderung für die Benutzer. Wir beschreiben einen Ansatz, der die technischen Systeme um Selbstbeschreibungsfähigkeit erweitert. So geben diese selbst Auskunft über ihre Fähigkeiten und teilen den Benutzern mit, auf welche Weise welche Funktionalität angesteuert werden kann.

1 Einleitung

Computergesteuerte Systeme dringen immer weiter in den Alltag des Menschen vor. Die Vorteile, die durch umfassendere Verarbeitung von Daten und Berechnung von Parametern entstehen, führen dazu, dass selbst in Bereichen, die vor kurzer Zeit noch mit analogen Schaltungen gesteuert wurden, inzwischen digitale Systeme („embedded systems“) eingesetzt werden. Beispiele finden sich vor allem im Automobilbereich, aber auch im Haushalt, etwa bei der Steuerung von Heizung oder Heimelektronik. Software, wie sie heute an den meisten Arbeitsplätzen und in vielen Haushalten verwendet wird, bietet ebenfalls eine enorme Vielfalt an Funktionen – in vielen Fällen eine solche Fülle, dass sie den Anwendern nur in Teilen bekannt ist, so dass viele Funktionen nicht genutzt werden.

2 Problemstellung und Use-Cases

Die Grundfunktionen verwendeter Software bereiten Benutzern keine Schwierigkeiten – etwa, wie man einen Text verfasst, abspeichert und ausdruckt. Weitergehende Kenntnisse der Textverarbeitung sind nicht unbedingt nötig, um die Standardaufgaben zu erfüllen, sie können jedoch je nach Anwendungsgebiet das Leben sehr erleichtern und helfen, Zeit und Arbeit einzusparen. Viele Zusatzfunktionen der Software lernt man aber entweder erst nach Studium des Handbuchs (wenn vorhanden, ansonsten mit Hilfe teuerer Ratgeber-Literatur) oder nach intensiver Benutzung des Programms kennen. Es liegt also der Gedanke nahe, die technischen Systeme mit einer Selbstbeschreibungsfunktionalität auszustatten. Gibt das System selbst Auskunft über seine Fähigkeiten, kann der Benutzer viel schneller lernen, mit vorhandener, bislang aber nicht genutzter Funktionalität unzugehen. In vielen Fällen wird er dadurch erst erfahren, was die Software zu leisten imstande ist. Aber auch für erfahrene Benutzer kann es sinnvoll sein, sich den Weg zu einer im Prinzip bekannten, aber selten genutzten Funktionalität demonstrieren zu lassen.

3 Lösungskonzepte

In einem ersten Schritt wurde der Ansatz verfolgt, unter Verwendung eines automatischen Planers die Klicksequenzen, die in einer komplexen graphischen Oberfläche zum Aktivieren eines Befehls nötig sind, zu berechnen. Diese Klicksequenzen wurden graphisch dargestellt, zusätzlich wurden Erklärungstexte erzeugt und ausgegeben. Im Rahmen des Forschungsprojektes FORSIP wurde dieser Ansatz in Form einer Erweiterung für den Webbrowser Firefox implementiert ([BR07, RLG05]).

Im zweiten Schritt gingen wir über dieses Szenario hinaus und möchten Erklärungsfunktionalität nicht mehr nur zu bereits bestehender Software hinzufügen, sondern diese in die Software selbst integrieren. Wir legen zu Grunde, dass Interaktionen zwischen System und Benutzer – und damit auch die dem Benutzer zur Verfügung stehenden Funktionen – ihre Entsprechungen in der Software haben. Beispielsweise wird in Firefox die Methode `increaseFontSize()` der Klasse `RenderEngine` aufgerufen, wenn auf den entsprechenden Menüpunkt geklickt wird. Diesen Umstand nutzen wir, indem wir interaktionsrelevante Code-Abschnitte mit Meta-Informationen anreichern, die wiederum für Erklärungen herangezogen werden können.

3.1 Selbsterklärende Funktionalität

Die Idee, Programmiersprachen um reflexive Fähigkeiten zu erweitern, ist nicht neu: Smith stellte in [Smi82] ein LISP-System (3LISP) vor, dessen Interpreter in mehreren Instanzen arbeitet, so dass von einer oberen Schicht (Meta-Ebene) aus der Zustand des Interpreters der darunterliegenden Schicht sichtbar ist. Ein Ziel der Arbeit war, Informationen über den Programmablauf zu geben. Maes ([Mae87, MN88]) erweiterte den Reflexionsansatz auf objektorientierte Programmiersprachen. Hier stehen jedoch hauptsächlich programmiertechnische Aspekte wie Debugging, Selbstoptimierung oder Generierung von Programmablaufstatistik im Vordergrund.

Viele moderne Programmiersprachen bieten bereits reflexive Schnittstellen für den Zugriff auf interne Strukturen zur Laufzeit. Da große Teile unseres Systems ohnehin in JAVA™ implementiert sind, nutzen wir das *java reflection api*. Es bietet unter anderem folgende Möglichkeiten: (1) Ermitteln der Klasse, dessen Instanz ein Objekt ist, (2) Ermitteln der Konstruktoren einer Klasse sowie ihrer Methoden, inklusive der Parameter und der Rückgabewerte, (3) Ermitteln der Oberklasse und der implementierten Interfaces, (4) Ermitteln der Annotationen, die eine Klasse und deren Methoden näher beschreiben. Darüber hinaus ist es möglich, über das API zur Laufzeit des Programms Klassen zu instantiieren, Variablenbelegungen durchzuführen und Methoden aufzurufen.

Ein weiteres Werkzeug, das JAVA™ (seit Version 5.0) bietet, sind *annotations*. Diese nehmen eine Zwischenstellung zwischen eigentlichem Quelltext und Kommentaren ein, indem sie einen Mechanismus bieten, ersteren um Meta-Informationen zu erweitern. Die Inhalte der Annotationen sind mit dem Quelltext verknüpft und können – müssen jedoch nicht – die Kompilation oder Ausführung des Programmes beeinflussen. Die

Inhalte der Annotationen sind über die *reflection api* noch zur Laufzeit des Programms zugänglich. Beispiele für Annotationen sind im folgendem Codeausschnitt in dem mit „@“ beginnenden Zeilen zu sehen:

```
@Method(doku="Analysiert eine Wortform morphologisch.")
public String analyzeToString(
   @Param(name="Input" doku="Zu analysierende Wortform")
    String input) {
   return (morph.analyze(input).toString()); }
```

3.2 Benutzerorientierte Programmierung

Die zwei vorgestellten Werkzeuge – *java reflection api* und *annotations* – nutzen wir im Sinne Benutzerorientierter Programmierung. Wir annotieren dazu den Quellcode an den Stellen, die aus Benutzersicht relevante Aufgaben erfüllen, mit menschenlesbaren Kommentaren. Ein verwandtes Vorgehen verwendeten [BBMR89], indem sie in ihrem CLASSIC-System die Möglichkeit gaben, bestimmte Konzepte als erklärungsrelevant auszuzeichnen. Adressat der Annotationen ist in unserem System im Gegensatz zu normalen Quellcode-Kommentaren nicht ein anderer Programmierer, sondern derjenige, der später das Programm aufruft, weswegen die Interaktion mit dem Programmbenutzer schon während der Programmierung eine zentrale Rolle einnimmt. Erklärungswürdiger Code läßt sich in zwei Kategorien einteilen:

1. Programmaktionen, die hinter Benutzerinteraktionsschritten stehen, wie z.B. *event handler*, die auf einen Mausklick auf einen Schaltknopf reagieren. Hier wird in einer Dokumentations-Annotation (siehe obiger Code-Ausschnitt) hinterlegt, welche Funktion der Code aus Anwendersicht im Hinblick auf die ausgeführte Aktion ausführt – und zwar in einer Formulierung, die auch für Anwender ohne Programmierkenntnisse verständlich ist.
2. Über die reine Reaktion auf Benutzereingaben hinaus erfüllen Programme eine bestimmte Aufgabe. Beispielsweise wird ein Dialogsystem eingesetzt, um eine bestimmte Applikation zu steuern, etwa das Navigationsgerät im Automobil. Um beim Benutzer Verständnis für die Funktionsweise des Systems hervorzurufen, ist es sinnvoll, ausgewählte Klassen und Methoden um Dokumentation anzureichern. Diese dokumentierten Teile des Programms dienen als Hintergrundinformation.

Das moderne Programmier-Paradigma der SOA (*Service-orientierte Architektur*), stellt Dienste, die dem Anwender angeboten werden, in den Mittelpunkt der Software-Konzeption. Über definierte Schnittstellen werden Informationen darüber angeboten, was der Service leistet und wie er angesteuert werden kann. Unser Ansatz der Benutzerorientierten Programmierung ist mit SOA eng verwandt. So ist die Beschreibung der Funktionalität der Software essentieller Bestandteil beider Ansätze. Während bei SOA die Ver-

wendung der Informationen auf maschinelle Interaktion ausgerichtet ist, extrahieren wir die Definitionen, zusammen mit menschenlesbaren Kommentaren, um sie dem Benutzer zu präsentieren. Je nach Benutzer und Umgebung kann es unterschiedlich geeignete Präsentationen geben, verschiedene Erklärungsstrategien sind noch Gegenstand unserer Forschungen. Bislang wurde prototypisch eine Art Debugging-Modus implementiert, wobei das Auftreten einer Dokumentations-Annotation zur Ausgabe des entsprechenden Textes auf der Konsole führt.

4 Ergebnisse und Ausblick

Es ist oft nicht einfach, die Grenze zwischen erklärungswürdigen und nicht erklärungsrelevanten Teilen des Codes zu ziehen. Dennoch zeigen die implementierten Prototypen, dass auch bei Benutzern ohne Programmierkenntnisse ein tieferes Verständnis für die Arbeitsweise des Systems entstehen kann, vor allem, wenn die erzeugten Erklärungen auf den Benuzer angepasst werden (Individualisierung). Gibt es beim Benutzer ein solches Verständnis für das System, sind die Systemaktionen deutlicher nachvollziehbar und die Frustration bei nicht gelungener Interaktion ist geringer, vor allem, wenn die Erklärung vom Systemzustand abhängig gemacht wird (Situierung). Gerade in Bereichen, die starke Parallelen zu bekannten Interaktionsvorgängen haben, wie es bei natürlichsprachichen Dialogsystemen der Fall ist, ist es wichtig, dem Anwender klarzumachen, dass das Gegenüber nicht über die Fähigkeiten verfügt, die ein menschlicher Gesprächspartner hat. Hat er ein Bild von der Arbeitsweise des Systems, mag ihm eine Systemreaktion, die bislang nur Unverständnis hervorgerufen hat, plausibel erscheinen, weil das System mitgeteilt hat, auf welche Weise es zu diesem Schritt gekommen ist. Die Anpassung unseres Systems auf verschiedene Benutzergruppen (Personalisierung) wird dazu beitragen, die Verständlichkeit der Systemerklärungen zu verbessern.

Literatur

[BBMR89] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness und Lori Alperin Resnick. CLASSIC: a structural data model for objects. In James Clifford, Hrsg., *Proc. SIGMOD International Conference on Management of Data*, Seiten 58–67, 1989.

[BR07] T. Bertz und P. Reiß. Plan-based Assistance in the Webbrowser Firefox. *Proceedings of AIA 2007*, Seiten 622–624, 2007.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proc. OOPSLA*, Seiten 147–155, ACM, 1987.

[MN88] P. Maes und D. Nardi. *Meta-Level Architectures and Reflection*. Amsterdam, 1988.

[RLG05] P. Reiß, B. Ludwig und G. Görz. Selbstreflexion in einem planbasierten Dialogsystem. *Lecture Notes in Informatics*, P67:256–260, 2005.

[Smi82] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. Dissertation, Massachusetts Institute of Technology, 1982.