

Intelligentes Programmieren: CodeBroker
Ausarbeitung im Rahmen des Proseminars „Künstliche
Intelligenz“ im Sommersemester 2004

Thilo Schmitt

Fakultät für Informatik, Universität Ulm
thilo.schmitt@informatik.uni-ulm.de

Abstract. CodeBroker ist ein sogenanntes „Active Component Repository System“, das mittels dynamischen Vorschlägen von bereits existenten, wieder verwendbaren, kontextspezifischen und vor allem aufgabenrelevanten Komponenten aus der API den Programmierer in seiner Aufgabe unterstützen soll. Dies soll die Produktivität in der Software-Entwicklung und die Qualität der daraus hervorgehenden Erzeugnisse steigern. Im Folgenden werden die Probleme, die bei der Methode der „Wiederverwendung von Programmfragmenten“ auftreten können diskutiert und Grundzüge eines Konzepts zur Lösung dieser Probleme anhand einer konkreten Implementierung aufgezeigt.

Inhaltsverzeichnis

1 Einleitung	2
2 Problemanalyse.....	3
3 Eine Implementierung: CodeBroker	6
3.1 Aufgabenrelevanz.....	6
3.2 Personalisierung	6
3.3 Hilfe durch Beispiele	8
4 Evaluierung	9
5 Related Work und Ausblicke.....	9
5.1 CodeBroker: Grenzen und Verbesserungen	9
5.2 Ähnliche Werkzeuge	10
5.3 Zusammenfassung und Fazit.....	10
6 Literaturangaben / Abbildungsnachweise	11

1 Einleitung

In der Software-Entwicklung gelten alle Anstrengungen einem Ziel: die programmierbare Umsetzung der Lösung einer abstrakt formulierten Problemstellung. Dabei bekommt der Entwickler eine Aufgabe und soll diese möglichst effizient und schnell lösen. Effizienz kann die Geschwindigkeit des Programms sein, die Größe und der vielen mehr. Die Effizienz eines Programms an sich soll hier aber nicht (oder nur wenig) Gegenstand dieser Ausarbeitung sein. Es soll hier vielmehr um Ansätze zur Optimierung des Vorgangs hinsichtlich effizienter Zeitnutzung einer Programm-Erstellung diskutiert werden.

Man stelle sich einmal einen fiktiven Entwickler vor. Dieser bekommt nun einen Auftrag und damit eine neue, eigenständige Problemstellung. Grundsätzlich hat dieser Entwickler nun zwei Möglichkeiten: eine eigenständige Lösung erarbeiten oder Adaption bereits vorhandener Lösungen.

Entweder er geht ganz „naiv“ an die Aufgabe heran und löst sie Schritt für Schritt, Teilaufgabe für Teilaufgabe. Er entwickelt eine völlig neue Lösung von Grund auf. Dies nennt man eine Programm-Erstellung „*from scratch*“ (englisch für „ganz von vorne“). Ein möglicher Vorteil ist, dass die Lösung exakt maßgeschneidert ist und ein Optimum an Aufgabenbezogenheit mitbringt. Der Nachteil allerdings ist mitunter die horrende aufzuwendende Zeit, denn ein solcher Prozess kann lange dauern.

Eine andere Möglichkeit hätte der Entwickler mit der Wiederverwendung ähnlicher, bereits erstellter Programmteile, die allgemein genug sind, um sie zu adaptieren. Zum Beispiel ist es unerheblich, ob durchnummerierte Spielkarten zufällig gemischt werden sollen oder eine Permutation von einigen Zahlen gewünscht ist. Es geht im Prinzip immer um die (zufällige) Durchmischung einer Menge von Zahlen. So könnte der Entwickler hier eine Methode zur Erzeugung von Permutationen für die Mischung der Karten adaptieren. Dies ist im Allgemeinen wesentlich weniger Aufwand als sich ein gutes Verfahren neu überlegen zu müssen. Diese Möglichkeit der Adaption bereits existenter Programmlösungen nennt man „*software reuse*“ oder „*component reuse*“ (englisch für „Wiederverwendung von Software/Komponenten“).

Worin die Vorteile und Schwierigkeiten des *software reuse* bestehen, wie diese Schwierigkeiten zumindest minimiert werden können und wie es möglich ist eine automatisierte Unterstützung für diese Technik bereitzustellen soll in dieser Arbeit diskutiert werden. Konkret wird dies an einer Implementierung eines solchen Systems gezeigt: CodeBroker.

2 Problemanalyse

Die zeiteffizientere Methode Software-Entwicklung zu betreiben die Methode des *software reuse* ist. Aber sind überhaupt wieder verwendbare Komponenten schon vorhanden oder ist die Diskussion ein wenig müßig, da solche Komponenten erst vom Entwickler erstellt werden müssten und dieser erst sehr viel später in die Phase des *software reuse* eintreten könnte (einmal angenommen jener Entwickler würde überhaupt mit einer mehr oder weniger ähnlichen Aufgabenstellung, die er schon einmal gelöst hat, wieder konfrontiert)?

Solche Komponenten-Sammlungen gibt es zu Hauf. Gerade da sich solche Sammlungen als nützlich erwiesen hatten, versuchte man einen Weg zu finden dies formal als Paradigma des Programmierens einzuführen: Es entstanden die objektorientierten Programmiersprachen, die sich per Definition solche Programmfragment-, „Bibliotheken“ nützlich machen. "[Reusable component] library design is [programming] language design." [Stroustrup, 1995]. Das Wissen um und über die Inhalte dieser Bibliotheken ist überdies obligatorisch geworden.

Allerdings sehen sich die Entwickler einem rasanten Wachstum dieser Bibliotheken gegenüber. Die so genannten APIs (*application programmer interface*, engl. für „Schnittstelle für Anwendungsprogrammierer“), die solche Bibliotheken strukturieren und eine Schicht zum „bequemen Zugriff“ zur Verfügung stellen wachsen stetig an. Die folgende Tabelle soll einen kleinen Eindruck für diese Entwicklung anhand der Java Core-API vermitteln.

Tabelle 1. Die Entwicklung der Java Core-API.

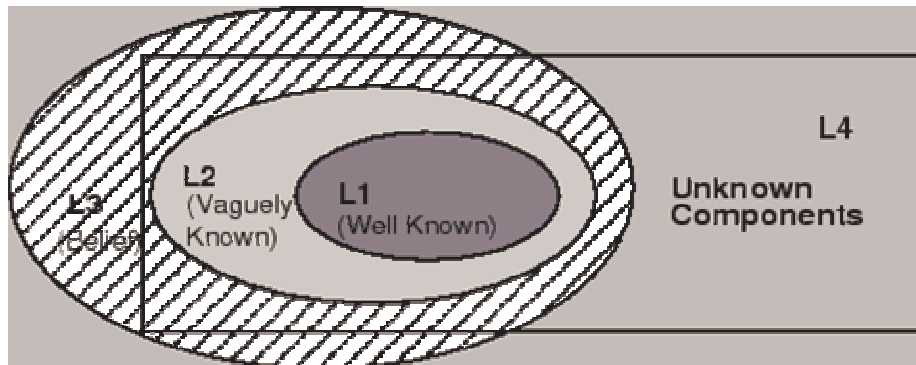
Version	Anz. d. Packages	Anz. d. Klassen	Erscheinungsjahr
Java 1.0	8	211	1996
Java 1.1	23	503	1997
Java 1.2	59	1525	1998
Java 2	über 70	über 2100	1999

Bei diesem Zuwachs, der wohl in näherer Zukunft kaum abbrechen dürfte, kann wohl kein Programmierer mehr behaupten die heutige API in ihrer Gesamtheit und die darin enthaltenen Komponenten als aktives Wissen erfasst zu haben. Umso weniger könnte jemand behaupten, all diese Komponenten in ihren Details zu kennen.

Es bleibt also nur festzustellen, dass die API zu unübersichtlich geworden ist. Ein Entwickler kann nur noch Teilgebiete wirklich gut kennen und noch weniger Komponenten direkt anwenden, da dafür die exakten Eigenschaften jener bekannt sein müssen. Abbildung 1 gibt in Annäherung eine grobe, graphische Darstellung über die Verhältnisse des Wissens eines Programmierers um Komponenten in der API wieder [Ye, 2001].

THILO SCHMITT

Abbildung 1. Wissen eines Programmierers über API-Komponenten.



(Die Menge aller existierenden Komponenten in der API ist durch das Rechteck dargestellt.)

Bei diesen Verhältnissen der „Verteilung“ (aktiven) Wissens ist es umso weniger erstaunlich, dass die Methode des *software reuse* an einigen Faktoren scheitern kann.

Zum einen kann es vorkommen, dass es unzureichende Möglichkeiten gibt (gar nicht vorhanden, zu langwierig, zu kompliziert, etc.) die API zu durchsuchen und damit eine existierende, passende Komponente nicht gefunden wird. Oder der Entwickler findet eine Komponente, kann sie aber nicht verwenden, da er die Arbeitsweise, anzugebende Parameter, etc. nicht versteht, da die Dokumentation zu wenig Auskunft darüber gibt.

Andererseits wird vielleicht eine Komponente gefunden, die nicht zu dem gewünschten Aufgabenfeld passt oder die beispielsweise ein unpassendes Datenformat liefert.

Es kann aber auch vorkommen, dass trotz umfangreicher Recherche eine passende Komponente gar nicht existiert. Und zu letzt gibt es den wohl größten Faktor: Der nicht unternommene Versuch der Recherche.

Abbildung 2 gibt Aufschluss über die Häufigkeit des Auftretens dieser Fehlerfaktoren bei der Methode des *software reuse*. Die dort graphisch dargestellten Daten wurden in einer empirischen Studie gewonnen. [Frakes and Fox, 1995].

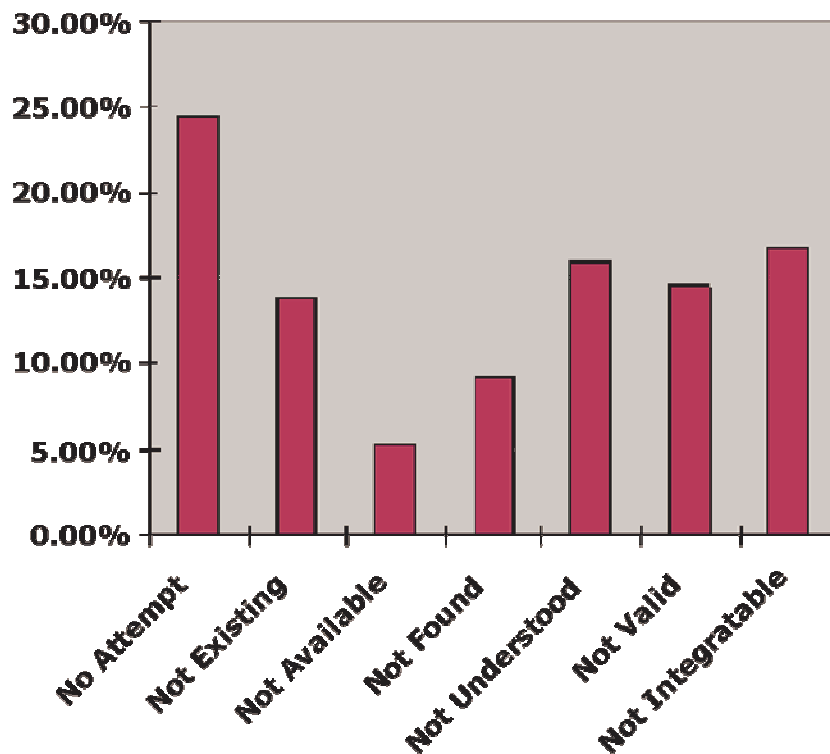
Das Auftreten der oben genannten Faktoren zu vermeiden oder wenigstens zu minimieren muss also das Ziel sein. Es geht um eine Informationsbereitstellung zur richtigen Zeit, im richtigen Kontext und vor allem vollautomatisch. Dies wird durch ein so genanntes *Active Information System* (AIS) erreicht, das aktiv (also während des Arbeitsvorgangs) relevante Informationen liefert.

Frühe Formen davon sind die allseits bekannten „Tip Of The Day“-Varianten, die die Information aber völlig kontextfrei und deplatziert präsentieren. Eine ein wenig

INTELLIGENTES PROGRAMMIEREN: CODEBROKER

fortgeschrittenere Variante – und dies völlig wertfrei bezüglich der Qualität und Sinnhaftigkeit dieses Systems – ist der ebenfalls hinreichend bekannte Office-Assistent von Microsoft. Dieser versucht nun wenigstens eine Kontextbezogenheit herzustellen, wenn auch die Form der Informationsdarbietung vielleicht etwas unglücklich gewählt ist.

Abbildung 2. Faktoren die zum Scheitern von *software reuse* führen.



Hier soll eine speziellere Form eines AIS gezeigt werden: ein so genanntes *Active Component Repository System* (ACRS). Dies ist speziell auf die Bedürfnisse der Software-Entwicklung ausgerichtet und bietet aktiv – also während des eigentlichen Arbeitsvorgangs und ohne weiteres Zutun – Komponenten aus dem dem System bekannten Fundus, der *repository*, zur Disposition und damit zur Wiederverwendung an. Dabei wird auf Kontextsensitivität und andere wichtige Bedingungen geachtet, die den Entwickler unterstützen und nicht mit Informationen überlasten.

Wie das genau geschieht und wie nun eine konkrete Implementierung aussieht, zeigt der nächste Abschnitt.

3 Eine Implementierung: CodeBroker

CodeBroker [URL: <http://www.cs.colorado.edu/~yunwen/codebroker/download.html>] ist ein von Yuwen Ye konzipiertes und implementiertes ACRS, das bei der Entwicklung von Java-Anwendungen helfen soll. Es ist als Plugin für den Editor Emacs mit der "Java Development Environment"-Erweiterung geschrieben und bietet somit eine direkte Integration in die Arbeitsumgebung des Entwicklers. Rein prinzipiell ist ein solches System aber auch für jede andere komponentenbasierte (also meist objektorienterte) Sprache denkbar. Auch die Integration in eine beliebige Entwicklungsumgebung ist grundsätzlich möglich, wenn diese gewissen Anforderungen für Schnittstellen genügt, um den vom Programmierer eingegebenen Text und Programmcode (just-in-time) analysieren zu können.

3.1 Aufgabenrelevanz

Wie stellt CodeBroker sicher, dass die Komponenten, die vorgeschlagen werden, auch zu der entsprechenden Problemstellung passen?

Einerseits wird ein Programmierer stets eine kurze Beschreibung dessen als Kommentar in den Programmtext einfließen lassen, was die darauf folgende Methode bewerkstelligen soll. Hier werden als solche erkannte Schlüsselwörter aus dem Kommentar „herausgezogen“ und für die Suche in der textualen Beschreibung der Komponenten genutzt. Auf dieser Basis bietet CodeBroker zunächst einmal alles an, was per Beschreibung möglichst gut zum Kommentar passt (inklusive einer Bewertung durch Abstufung per Treffergenauigkeit). (*Abbildung 3*) Dies stellt die wichtigste Informationsquelle von CodeBroker dar, da durch Schlüsselwörter der „Bereich“ einer Aufgabe am besten und einfachsten umrissen werden kann.

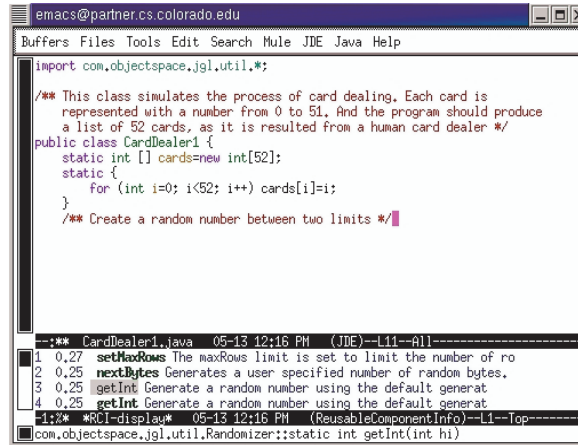
Außerdem kann der Entwickler nun noch durch die Signatur der Methode die Art und Weise festlegen wie Daten ausgetauscht werden sollen. Sollen Fest- oder Gleitkommazahlen verwendet werden und welche Genauigkeit sollen diese haben? Werden etwa Zeichenketten oder gar komplizierte Datenstrukturen als Ausgabe benötigt? Dies wird durch die Methoden-Signatur definiert. Anhand dieser Information kann CodeBroker die zuvor getroffene Auswahl wiederum filtern und zwar dahingehend, dass nur noch Komponenten angeboten werden, die in diesen Kontext passen. (*Abbildung 4*)

3.2 Personalisierung

Nun kann es aber störend sein, dass CodeBroker ständig Komponenten zur Verwendung vorschlägt, die der Programmierer ohnehin in seinem aktiven Wissen hat, und so die Sicht auf andere, vielleicht auch interessante Komponenten verstellt wird. Auch kann es vorkommen, dass Komponenten vorgeschlagen werden, die rein gar nichts mit dem Feld der Problemstellung zu tun haben, aber eben durch die textuale Beschreibung passen würden. Auch diese stören bei der Arbeit.

INTELLIGENTES PROGRAMMIEREN: CODEBROKER

Abbildung 3. CodeBroker nach Eingabe eines Kommentars, der die Methode beschreibt



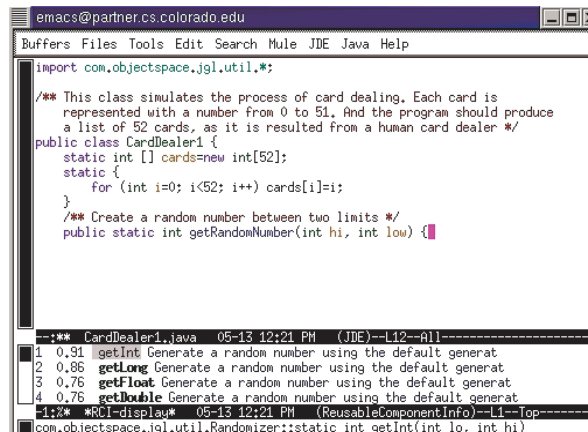
```
emacs@partner.cs.colorado.edu
Buffers Files Tools Edit Search Mule JDE Java Help

import com.objectspace.jgl.util.*;

/** This class simulates the process of card dealing. Each card is
represented with a number from 0 to 51. And the program should produce
a list of 52 cards, as it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
}

---** CardDealer1.java 05-13 12:16 PM (JDE)--L11--All-----
1 0.27 setMaxRows The maxRows limit is set to limit the number of ro
2 0.25 nextBytes Generates a user specified number of random bytes.
3 0.25 getInt Generate a random number using the default generat
4 0.25 getInt Generate a random number using the default generat
---** *RCI-display* 05-13 12:16 PM (ReusableComponentInfo)--L1--Top-----
com.objectspace.jgl.util.Randomizer::static int getInt(int hi)
```

Abbildung 4. CodeBroker's Angebot nach Eingabe der Methoden-Signatur: ein voller Erfolg.



```
emacs@partner.cs.colorado.edu
Buffers Files Tools Edit Search Mule JDE Java Help

import com.objectspace.jgl.util.*;

/** This class simulates the process of card dealing. Each card is
represented with a number from 0 to 51. And the program should produce
a list of 52 cards, as it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
    public static int getRandomNumber(int hi, int low) {}
}

---** CardDealer1.java 05-13 12:21 PM (JDE)--L12--All-----
1 0.31 getInt Generate a random number using the default generat
2 0.86 getLong Generate a random number using the default generat
3 0.76 getFloat Generate a random number using the default generat
4 0.76 getDouble Generate a random number using the default generat
---** *RCI-display* 05-13 12:21 PM (ReusableComponentInfo)--L1--Top-----
com.objectspace.jgl.util.Randomizer::static int getInt(int lo, int hi)
```

Dafür gibt es eine Funktion in CodeBroker, die dafür Sorge trägt, den Programmierer wirklich zu unterstützen und ihm die Möglichkeit gibt dem System eine Art Information über seinen Wissensvorrat mitzuteilen.

Dies geschieht durch den so genannten „Skip Components“-Dialog, der die Möglichkeit bietet einzelne Komponenten, Komponenten-Gruppen oder gar ganze Packages einen von zwei „Modellen“ zuzuordnen: dem *user model* oder dem *discourse model*. (Abbildung 5)

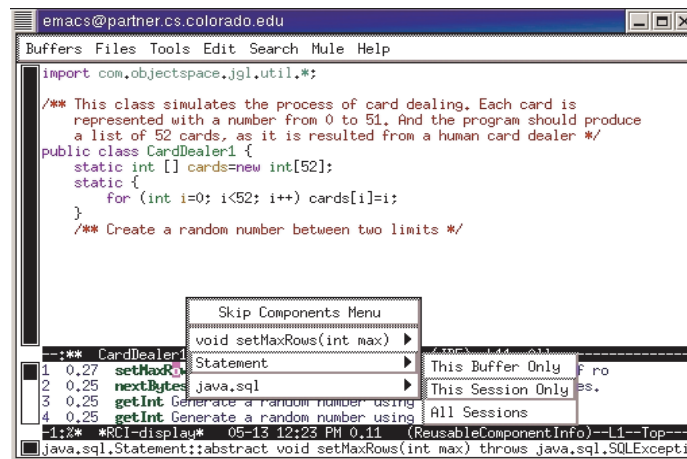
Das *user model* repräsentiert das Wissen eines Programmierers und enthält damit Komponenten/Packages, die CodeBroker grundsätzlich nie vorschlagen muss. (Idealerweise würde das *user model* gerade die Menge L1 in Abbildung 1

THILO SCHMITT

repräsentieren.) Das *discourse model* bezieht sich auf Themengebiete, die im Umfeld der aktuellen Aufgabenstellung ausgeschlossen werden können. Hier finden sich Komponenten, die zwar per Beschreibung und Signatur passen würden, aber nicht in den Kontext des zu bearbeitenden Problems gehören.

Aber nicht nur das aktive Hinzufügen von Komponenten in das *user model* wird durch CodeBroker unterstützt, sondern auch das automatische, welches durch die stete Analyse des Programm-Codes geschieht. Entdeckt CodeBroker eine Komponente, die häufiger vom Programmierer benutzt wird, so fügt das System diese automatisch dem *user model* hinzu, denn es kann davon ausgegangen werden, dass bei mehrfacher Verwendung durch den Programmierer, dieser das Wissen um eben jene Komponente in seinen Erfahrungsschatz aufgenommen hat.

Abbildung 5. Der „Skip Components“-Dialog



3.3 Hilfe durch Beispiele

Wie aus Abbildung 2 deutlich wird, liegt die Häufigkeit des Fehler-Faktors „Einsatzweise/Arbeitsweise der Komponente nicht verstanden“ bei über 15%. Oft schafft ein Beispiel Klarheit. Folgerichtig bietet CodeBroker die Funktion ein Programm-Fragment anzuzeigen, in dem die Komponente bereits erfolgreich eingesetzt wurde. Das setzt allerdings voraus, dass es überhaupt solche Fragmente gibt, was allerdings oft nicht der Fall ist - zumindest was die Standard-API angeht.

Besser sieht es aus, wenn CodeBroker beispielsweise in einem Unternehmen von allen oder wenigstens vielen Entwicklern eingesetzt würde. Dann könnte CodeBroker neu erstellte Programme analysieren und entsprechend indexieren. Dies ist aber in der aktuellen Implementierung so direkt nicht vorgesehen. Man beachte dazu auch Kapitel 5.2 (Grenzen und Verbesserungen).

4 Evaluierung

Das CodeBroker-System wurde selbstredend auf seine Tauglichkeit und Nützlichkeit hin untersucht. Dafür wurden 12 Experimente durchgeführt. Hier waren kleine Java-Anwendungen zu implementieren, die mit verschiedensten API-Komponenten lösbar waren. Diese Experimente wurden auf 5 Entwickler aufgeteilt, deren Kenntnisse und Fertigkeiten in Java variierten: Vom Fortgeschrittenen bis zum Experten.

In den fertiggestellten Programmen wurden 57 API-Komponenten benutzt, wovon 20 durch CodeBroker vorgeschlagen wurden. Von diesen 20 Komponenten kannten die Entwickler 9 gar nicht. Und es wurden sogar Packages benutzt, von denen einige Entwickler nicht gedacht hätten, dass diese existieren.

Schlussendlich gaben die Programmierer Noten auf einer Skala von 1 (nutzlos) bis 10 (äußerst nützlich). Dabei schnitt CodeBroker sehr gut ab. Mit einem Durchschnitt von knapp 7 Punkten bekam das System die Wertungen: 8.5, 8, 7, 7 und 4.

Einige Entwickler merkten an, dass CodeBroker das Arbeiten sehr viel zeitsparender und teilweise interessanter mache. Aber auch Kritik gab es, vornehmlich an der Usability (Benutzerfreundlichkeit) des Systems (Funktionen nur per Maus erreichbar, CodeBroker verbraucht zu viel Fläche des Bildschirms, etc.)

In der Quintessenz allerdings zeigte sich, dass die Entwickler CodeBroker durchaus als gutes, nützliches System mit einem guten Konzept bewerteten.

5 Related Work und Ausblicke

5.1 CodeBroker: Grenzen und Verbesserungen

Eine klare Grenze zeichnet sich für CodeBroker dahingehend ab, dass das System an die bereitgestellten Daten gebunden ist. Bekommt CodeBroker nur wenig Material zur Analyse und Indexierung, so wird ein vernünftiges Arbeiten mit CodeBroker nur recht bedingt möglich sein. Je mehr Dokumente vorliegen, desto besser kann CodeBroker die Texte auf gewisse semantische Eigenschaften und Räume hin überprüfen und so einen sinnvollen Fundus an Schlüsselwörtern für die Textsuche in Beschreibungen aufbauen.

Genau hier liegt auch eine Möglichkeit zur Verbesserung von CodeBroker. Man könnte bessere Algorithmen zum „Vergleich“ von Beschreibungen verwenden, sofern diese existieren und gefunden werden.

THILO SCHMITT

Darüber hinaus wäre eine vernetzte Repository von Vorteil. So hätte CodeBroker die Möglichkeit auf Erfahrungen und Neu-Entwicklungen anderer Programmierer zu reagieren und diese Vorteile an den Anwender direkt weiter zu geben. Es wäre vorstellbar CodeBroker „im *Grid*“ zu organisieren, so dass verteilte Informationen zu einem großen Fundus zusammengefasst werden können. Dies betrifft Komponenten an sich, als auch Dokumente zu deren beispielhaften Einsatz.

Und letztendlich bietet sich noch die Möglichkeit CodeBroker auf eine abstraktere Ebene zu adaptieren. Hier wäre denkbar, dass statt Komponenten gleich ganze Designs oder Pattern (Muster) vorgeschlagen werden. Dazu müssten aber völlig neue Konzepte erarbeitet werden, die festlegen, wie solche Dinge analysiert und repräsentiert werden können und an welchem Punkt der Arbeit das Vorschlagen solcher Objekte sinnvoll ist.

5.2 Ähnliche Werkzeuge

„Remembrance Agent“ ist ein AIS, das den Benutzer mit alten Mails und dazugehörigen Notizen versorgt, die zu der Mail passen, die jener gerade schreibt. So können leichter Bezüge auf früher ausgetauschte Informationen hergestellt werden und diesen werden aus der Menge des gesamten Fundus kontextsensitiv geliefert.

„GURU“ ist ein ACRS, welches auf der Indexierung von Komponenten durch deren sprachlichen Beschreibung aufsetzt. So gesehen ist dieses System ein „halbes“ CodeBroker-System. Es fehlt die Filterung durch Methoden-Signaturen und auch die nahtlose Integration in die Entwicklungsumgebung, die auf der steten Programmcode-Analyse basiert.

Es gibt noch einige ACRS, die ähnliche Funktionalitäten aufweisen. An dieser Stelle sollen einige Namen solcher Systeme genügen: CodeFinder, LaSSIE, ROSA, etc.

5.3 Zusammenfassung und Fazit

CodeBroker ist ein nützliches System, das bei der Anwendungs-Entwicklung enorme Vorteile durch den Nutzen von *software reuse* bringen kann. Allerdings ist meines Erachtens eher das Konzept glücklich gewählt, als das System (künstlich) intelligent. Hier werden lediglich die Probleme des *software reuse* zum Anlass genommen ein Werkzeug dahin gehend zu konzeptionieren, dass es solche Probleme zum Großteil durch geschickten Einsatz verschiedener Mittel minimiert. Wirkliche Intelligenz ist hier nicht am Werk. Einzig und allein das Filtern und Analysieren von Textdokumenten auf mögliche Schlüsselworte hin ist eine in dieser Richtung zu würdigende „Leistung“ des Systems.

Da im Rahmen dieses Proseminars in einer anderen Ausarbeitung das „Verstehen natürlicher Sprache“ behandelt wird, wurde hier auf die Funktionserklärung der eigentlichen Technik des Textverstehens, Zusammenfassens von Texten und Exzerpieren von Schlagwörtern verzichtet.

6 Literaturangaben / Abbildungsnachweise

- [Ye, 2001] Ye, Yuwen (2001). Supporting Component-Based Software Development with Active Component Repository Systems [Dissertation]. Colorado: University of Colorado
- [Ye, 2003] Ye, Yuwen (2003). Programming with an intelligent agent. IEEE Intelligent Systems 18(3):43-47
- [Stroustrup, 1995] Stroustrup, B. (1995). The C++ Programming Language. Addison-Wesley, Reading, MA, 2nd edition.
- [Frakes and Fox, 1995] Frakes, W. B. and Fox, C. J. (1995). Sixteen questions about software reuse. Communications of the ACM, 38(6):75-87.