Verbesserung der Retrievaleffizienz von Softwarekomponentenmärkten

Die Autoren

Oliver Hummel
Colin Atkinson

Dipl.-Inf. Oliver Hummel
Prof. Dr. Colin Atkinson
Universität Mannheim
Lehrstuhl für Softwaretechnik
68161 Mannheim
hummel@informatik.uni-mannheim.de

Eingereicht am 2006-10-11,
nach vier Überarbeitungen angenommen
am 2007-08-17
durch Prof. Dr. Buhl.

1 Einleitung

Die Softwaretechnik wurde in zurückliegenden Jahren häufig für ihre spektakulären Fehlschläge gescholten: der gescheiterte Jungfernflug der Ariane 5, die verspätete Eröffnung des Flughafens in Denver in den USA oder die Probleme bei Toll-Collect und virtuellem Arbeitsmarkt in Deutschland sind nur einige prominente Beispiele, die Millionen oder gar Milliarden von Euro bzw. Dollar gekostet haben. Deren Ursache ist häufig schnell ausgemacht, die vergleichsweise junge Softwaretechnik befinde sich noch nicht auf dem Niveau einer Ingenieursdisziplin, in der die industrialisierte Umsetzung von großen Projekten unter kontrollierten Bedingungen möglich wäre. Andererseits haben auch etablierte Disziplinen mit ähnlichen Problemen zu kämpfen. Die jüngsten Verzögerungen um den neuen Airbus A380 oder die aus dem Rahmen gelaufenen Kosten bei der Modernisierung zahlreicher Stadien für die Fußball-WM 2006 zeigen, dass Softwareentwickler mit ihren Problemen nicht alleine stehen.

Komponenten [Szyp02], Entwurfsmuster [GHJV95] und aspektorientierte Entwicklung [KLMM97] sind nur einige, von anderen Disziplinen inspirierte Ansätze, denen noch in den letzten zehn Jahren zugetraut wurde, die sogenannte „Softwarekrise“ endgültig zu lösen. Inzwischen scheint allerdings die Einsicht anerkannt, dass es kein Allheilmittel [Broo87] für die Softwareentwicklung geben kann, sondern dass alle Ansätze bestenfalls eine weitere Hilfe sein können, um die Softwaretechnik näher an etablierte Ingenieursdisziplinen heranzuführen. Parnas definierte Anfang der 1970er Jahre ein System an dem fünf Entwickler ein Jahr beschäftigt sind, als die Grenze von reiner „Handwerksarbeit“ zu industrialisierter Softwareentwicklung. Mittlerweile werden Softwareprojekte realisiert, die diese Grenze weit überschreiten. Das ebenfalls in den frühen 1970er Jahren umgesetzte Prinzip der Dekomposition einer Applikation in einzelne Module [Parn72], die verschiedene Entwickler unabhängig voneinander bearbeiten können, war dafür eine wichtige Voraussetzung. Die Idee der Komposition von Applikationen durch Wiederverwendung vordefinierter Module ist der nächste logische Schritt und inzwischen gibt es zahlreiche Berichte über damit erzielte Produktivitäts- und Qualitätssteigerungen [z. B. in BaBM96; LeSW87]. Der Begriff Software-Reuse umfasst heute allerdings nicht mehr nur die Wiederverwendung vordefinierter Komponenten, sondern eine ganze Reihe weiterer Ideen wie die bereits genannten Entwurfsmuster [GHJV95] oder den Einsatz von Produktlinien [ClNo02] bei verwandten Applikationen mit gleichartigen Teilen.

Kernpunkte

Softwarekomponentenmärkte konnten sich bisher nicht auf breiter Front durchsetzen. Die Literatur nennt zahlreiche Diffusionsbarrieren, unter anderem ein mangelhaftes Komponentenangebot und schlechte Retrievaleffizienz. Dieser Beitrag begegnet beiden Schwächen unter folgenden veränderten Rahmenbedingungen:

- Open-Source-Software im WWW eröffnet neue Möglichkeiten zur Komponentensuche: Mehrere Millionen Komponenten sind online verfügbar.
- Die Suche von Softwarekomponenten im WWW mit spezialisierten Suchmaschinen ist zwar möglich, aber immer noch zu ungenau.

Der Extreme-Harvesting-Ansatz ermöglicht erstmals präzises Komponentenretrieval aus extrem großen Beständen, basierend auf einer Spezifikation aus Komponentenschnittstelle und zugehörigen Testfällen.

Stichworte: komponentenbasierte Softwareentwicklung, Komponentenretrieval, Wiederverwendung

Der Beitrag befasst sich mit der Wiederverwendung von Komponenten und Services aus entsprechenden Märkten, wie sie bereits von [McIl68] skizziert wurde und trotz beinahe vier Jahrzehnten intensiver Forschung bis heute keinen Eingang in die Entwicklungspraxis gefunden hat. Diese Vorgehensweise erlitt erst kürzlich durch die überraschende Schließung der UDDI Business Registry (UBR) für Werbservices einen erneuten Rückschlag. Deren Abschaltung beruhte nach Angaben der Betreiber [Micr06] zwar darauf, dass der mehrjährige Betrieb der UBR als Service Broker (wie in Bild 1 schematisch gezeigt) als erfolgreicher Konzeptnachweis angesehen werden könne, unabhängige Untersuchungen [HuAt03, 303] zeichnen aber ein anderes Bild. Die UBR enthielt kurz vor ihrer Schließung zwar zahllose Einträge, darunter aber nur rund 400 erreichbare Verweise auf technische Servicebeschreibungen (d. h. WSDL-Dateien), die zur Nutzung eines Werbservice zwingend notwendig sind. Bedenkt man nun, dass bei dieser an sich schon sehr geringen Zahl viele Services nicht mehr oder nicht zuverlässig funktionierten bzw. es sich häufig nur um einfache „Spielzeugbeispiele“ handelte, wird schnell klar, dass man die von den Marketingabteilungen der großen Softwarehäuser beschworenen Business-Services in der UBR vergeblich suchen musste und sie daher nur von geringem Nutzen war.

Aber auch Komponentenmärkte oder -Broker, die analog dem in Bild 1 gezeigten Prinzip funktionieren, sucht man bis heute weitgehend vergeblich. Die wenigen Ausnahmen, wie sie beispielsweise unter http://componentsource.com im WWW zu finden sind, verdeutlichen sehr schnell, wo in der Praxis noch Probleme liegen. Sucht ein Entwickler dort nach einer Komponente, so kann er nur eine textbasierte Suchfunktion benutzen, um unter wenigen hundert Kandidaten Komponenten an Hand einer textuellen Beschreibung zu durchforsten. Dies entspricht etwa der Vorgehensweise mit der ein Käufer bei einem Onlinehändler ein Buch aufstöbern kann. Eine einfache Möglichkeit, Komponenten passend zu ihren Spezifikationen aus dem „Bauplan“, sprich dem (UML-) Design, eines Softwaresystems zu finden und zu bestellen, wie dies z. B. in der KobrA-Methode des Fraunhofer IESE [ABBK02, 251–263]) vorgeschlagen wird, gibt es nicht. Daher verwundert es nicht, wenn z. B. [FrFo96, 276] mangelnde Erfolgsaussichten als einen Hauptgrund für nicht erfolgte Softwarewiederverwendung nennen. Im Mittelpunkt des Beitrages steht daher die Verbesserung des Komponentenretrievals im Hinblick auf die Retrievalgenauigkeit (*Precision*), um die sehr großen Repositorys mit Millionen von Komponenten, wie sie heute in Unternehmen und dank der Open-Source-Bewegung auch im Internet zur Verfügung stehen, für spezifikationsbasierte Wiederverwendung nutzbar zu machen. Zahlreiche weitere, nichttechnische Herausforderungen (wie z. B. Anreize zur Wiederverwendung oder geänderte Rahmenbedingungen für das Management), wie sie nicht nur bei [FrFo96, 276] zu finden sind, müssen aus Platzgründen an dieser Stelle unberücksichtigt bleiben.

Die in den Naturwissenschaften übliche Vorgehensweise, Hypothesen empirisch zu überprüfen, halten die Autoren im vorliegenden Kontext für nicht ausreichend. Die Forschung und auch der Aufbau dieses Beitrags orientieren sich an der von [NuCh90] vorgeschlagenen konstruktivistischen Vorgehensweise, die Systementwicklung per se und damit auch Machbarkeitsnachweise zur Erkenntnisgewinnung ausdrücklich einschließt. Im folgenden Abschnitt 2 werden zunächst einige grundlegende Begriffe und der konzeptuelle Rahmen des Beitrages definiert. Der aktuelle Stand der Technik im Bereich des Komponenten- bzw. Serviceretrievals und die damit einhergehenden Schwächen in Bezug auf die Umsetzung von großen Softwarerepositorys und Komponentenmärkten werden erläutert. In Abschnitt 3 werden der grundlegende Aufbau und die Machbarkeit des sog. Extreme-Harvesting-Ansatzes, mit dessen Hilfe Komponenten oder Services dort gemäß ihrer Spezifikation präzise aufgefunden werden können, diskutiert. In Abschnitt 4 werden neue Erkenntnisse über die Effizienz verschiedener Retrievaltechniken in sehr großen Repositorys vorgestellt, die die Autoren kürzlich mit einer eigenen Komponentensammlung gewinnen konnten. Im darauf folgenden Abschnitt wird diskutiert, an welchen Punkten derzeit eine praktisch einsetzbare Retrievallösung für komplexe Komponenten noch scheitern muss und Einblick in die laufenden Arbeiten der Autoren gegeben. Abschnitt 6 fasst die wichtigsten Kernaussagen zusammen.

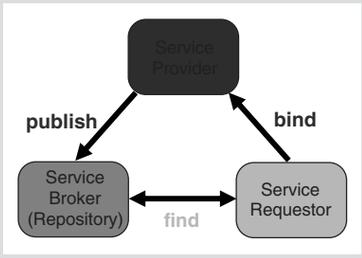

Bild 1 Schematische Darstellung der Brokerage-Architektur für Webservices

2 Grundlagen

Bereits mehrfach wurden Begriffe wie Komponente und Service benutzt, ohne dass diese bisher klar definiert oder voneinander abgegrenzt wurden. Das wird an dieser Stelle nachgeholt, um dabei auch die unterschiedlichen Sichtweisen von Informatik und Wirtschaftsinformatik zu erläutern. Die Autoren orientieren sich weitgehend an der in der Informatik gängigen Komponentendefinition des ECOOP Workshops zu komponentenorientierter Programmierung 1996, die von [Szyp02, 41] wie folgt wiedergegeben ist: „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Interessanterweise können weder die heute verbreiteten objektorientierten Programmiersprachen noch die daraus hervorgegangenen Komponententechnologien wie z. B. Enterprise Java Beans diese Definition vollständig erfüllen. Vor allem die *contractually specified interfaces*, also eine semantische Beschreibung des „provided interface“, und die *explicit context dependencies*, d. h. eine nach außen sichtbare Beschreibung des sog. „required interface“, sind in heutigen Programmiersprachen nicht gegeben, so dass dort am ehesten Klassen als Komponenten betrachtet werden können. Allerdings macht [Szyp02, 41] selbst widersprüchliche Angaben zu dieser Überlegung, während er diese auf S. 38 noch ohne direkten Bezug auf obige Definition verneint, argumentiert er auf S. 285 „a (Java) bean is really a component“. Eine solche Java-Bean ist nun aber nichts anderes als eine Klasse. Die Autoren sind daher der Meinung, dass es gerechtfertig ist, Klassen und analog zu Webservices sogar zustandlose Methoden als feingranulare Komponenten anzuerkennen, sofern eine Spezifikation aus Schnittstelle und Verhaltensbeschreibung ableitbar ist. Diese An-

sicht wird auch von zahlreichen jüngeren Veröffentlichungen wie [IYFY05] oder [YeFi05] und aktuellen komponentenbasierten Entwicklungsmethoden wie KobrA [ABBK02] oder Catalysis [DSWi99] gedeckt, wobei letztere Komponenten hierarchisch komponieren und damit die Klassenebene bei weitem übersteigen können.

Auch für die Wirtschaftsinformatik sind natürlich vor allem komplexere Komponenten interessant, dort wird in diesem Zusammenhang häufig auf die Komponentendefinition von Turowski [Turo02, 1] verwiesen, die der genannten von Szyperski sehr ähnlich ist: Während letztere aber vor allem auf Komponenten in binärer Form zielt, schließt Turowski ausdrücklich weitere Artefakte wie z. B. Spezifikation oder Implementierung mit ein. Die Reusable Asset Specification (RAS) [OMG05] bietet ein entsprechendes, in der Praxis eher selten genutztes, physisches Paketierungsmodell an. Der ebenfalls vor allem in der Wirtschaftsinformatik bekannte Begriff einer Fachkomponente [Turo02, 1] liegt im Vergleich zu den eben genannten, rein technischen Definitionen auf einer höheren Abstraktionsebene und verlangt neben einer technischen Spezifikation auch Funktionalität aus einer betrieblichen Anwendungsdomäne sowie weitergehende Metadaten wie z. B. Qualitäts- und Vermarktungsmerkmale.

Wie eine Komponente kapselt auch ein (Web-)Service [Stal02] Funktionalität hinter einer klar definierten Schnittstelle, ohne dass Nutzer Zugriff auf Interna bekommen. Die in diesem Beitrag diskutierten Ideen sind daher weitgehend analog von Komponenten auf Werbservices bzw. von Komponenten- auf Servicemärkte übertragbar. Der wesentliche Unterschied zwischen beiden liegt darin begründet, dass in einem Servicemarkt unmittelbar der Zugang zu einem plattformunabhängigen Dienst vermittelt wird, während ein Komponentenmarkt nur Komponenten bestimmter Plattformen (bzw. Programmiersprachen) offeriert, die heruntergeladen, ggf. kompiliert und eingebunden werden müssen, bevor sie genutzt werden können.

2.1 Grundlagen des Komponentenretrievals

Da die ersten Softwareretrieval-Ansätze zumeist Ideen aus dem *Information Retrieval (IR)* aufgegriffen haben, liegt es nahe, auch zur Beurteilung der Qualität eines solchen Retrievalsystems auf die im IR [BaRi99, 75] gängigen Maße *Precision* und *Recall* zurückzugreifen, wie dies z. B. auch von Mili et al. [MiMM98] in der bislang ausführlichsten Übersicht zu Softwarebibliotheken getan wird. Die Precision gibt dabei den Anteil von relevanten Ergebnissen im Verhältnis zu allen gelieferten Ergebnissen an. Im Gegensatz dazu steht der Recall, der den Anteil der gelieferten und relevanten Ergebnisse im Verhältnis zu allen im Repository vorhandenen relevanten Ergebnissen angibt. Im Laufe der Jahre wurde eine Unzahl von Retrievaltechniken entwickelt, die von Mili et al. [MiMM98, 360] in sechs weitgehend orthogonale Kategorien eingeteilt werden. Im Folgenden wird auf die Techniken, deren Kenntnis im weiteren Verlauf wichtig sein wird, kurz eingegangen. Wie bereits angedeutet, können textbasierte Verfahren aus dem klassischen IR auch für den Quelltext von Software oder unter Umständen für die Metadaten von Fachkomponenten Verwendung finden, da dort zahlreiche textuelle Elemente, wie z. B. Klassen-, Methoden- und Variablennamen oder Kommentare enthalten sind. Solche Verfahren profitieren vor allem von einfachen Suchanfragen, ähnlich derer, die heute täglich an Suchmaschinen im WWW gesendet werden, leiden gleichzeitig aber unter mangelnder Genauigkeit (Precision). Weitere Probleme gibt es vor allem mit Synonymen (unterschiedliche Worte mit gleicher Bedeutung) und Homonymen (gleiches Wort mit verschiedenen Bedeutungen). Mögliche Lösungsansätze hierfür wurden zwar z. B. bereits in [Alpa80] diskutiert, die sog. semantische Lücke zwischen einer natürlichsprachlichen Beschreibung und der Funktionalität einer Komponente kann aber bis heute nicht zuverlässig überbrückt werden. Außerdem scheitern diese Verfahren bei Blackbox-Komponenten und Services, da bei diesen häufig nicht genügend verwendbarer Text zur Verfügung steht.

Diese Lücke versuchen operationale Verfahren [MiMM98, 376–382) zu vermeiden, indem sie Funktionalität direkt ausführen und dabei ihr Verhalten betrachten. Dazu benötigen sie neben den formalen Parametern auch den Rückgabetyp einer Operation (d. h. ihre Signatur) sowie Tupel aus beliebigen Eingabewerten mit erwarteten Rückgabewerten als Sucheingabe. Ein solcher Ansatz wurde als Behavior Sampling zuerst von Podgurski und Pierce [PoPi93] für seiteneffektfreie C-Funktionen mit primitiven Ein- und Ausgabedatentypen beschrieben. Er bringt allerdings zahlreiche Probleme mit sich, alleine das sog. *Signature Matching* zur Auswahl syntaktisch passender Kandidaten ist bereits eine Herausforderung für sich. Weiterhin können Seiteneffekte und nichtterminierende Operationen zu Problemen führen. Auch für die Frage, wie abstrakte Datentypen in Signaturen beschrieben werden sollen, existiert von Podgurski und Pierce zwar der Vorschlag, diese durch ihre Konstruktoren zu ersetzen, eine praktikable Lösung für objektorientierte Sprachen gibt es dafür aber bislang nicht. Auch die mangelnde Skalierbarkeit für große Bibliotheken ist ein bekannter Schwachpunkt dieses Verfahrens. Das gerade angesprochene Signature Matching wurde von Zaremski und Wing [ZaWi95] genauer untersucht und fällt in der Kategorisierung von Mili et al. [MiMM98, 383–392] unter die sog. denotationalen Verfahren. Wie sein Name bereits andeutet, nutzt es Typen in Signaturen von Operationen, um passende Reuse-Kandidaten zu finden. Entscheidender Nachteil dabei ist, dass z. B. bei Klassen mit einer großen Zahl von Operationen und Parametern auch zahlreiche Permutationen in Frage kommen können. Mit Hilfe dieses Ansatzes allein kann die korrekte Variante aber nicht zweifelsfrei bestimmt werden. Außerdem ist die Wahrscheinlichkeit sehr hoch, in einer großen Komponentensammlung zahlreiche syntaktisch identische Operationen zu finden, die semantisch völlig verschieden sind.

2.2 Kritische Betrachtung der bekannten Ansätze

Interessant ist die nach wie vor große Diskrepanz zwischen den Erwartungen, die durch die vorhandenen Komponentenkonzepte in der (Wirtschafts-)Informatik geweckt werden und den technischen Gegebenheiten moderner Programmiersprachen sowie den in der Literatur vorhandenen Lösungsansätzen zum Komponentenretrieval. Poulin [Poul99] zum Beispiel argumentierte noch vor wenigen Jahren, dass mit den eben beschriebenen Arbeiten, trotz der genannten Einschränkungen, bereits alle wesentlichen Probleme im Zusammenhang mit Reuse-Bibliotheken als gelöst betrachtet werden könnten. In der Praxis erwiesen sich die genannten Ansätze allerdings weitgehend als nicht brauchbar und [Seac99] sowie [MiMM98, 406] rechneten vor knapp zehn Jahren angesichts der technischen Probleme nicht zeitnah mit einer Lösung. Dies mögen gute Gründe für die wenigen Untersuchungen über die tatsächliche Effizienz (d. h. Precision und Recall) von Softwareretrievaltechniken und deren stagnierende Weiterentwicklung

in den letzten Jahren gewesen sein. Schon Mili et al. erkannten die mangelhafte Datenbasis für ihre genannten Studie, wagten dort aber immerhin ungefähre Angaben auf Basis der vorhandenen Literatur und eigenen Schlussfolgerungen. Die wenigen bekannten, konkreten Untersuchungen z. B. von Frakes und Pole [FrPo94] oder Podgurski und Pierce [PoPi93] sind noch älter und verwenden nur sehr kleine Bibliotheken mit etwa einhundert Elementen (Unix-Kommandos bzw. C-Funktionen). In dieser Größenordnung liegt auch die obere Schranke, die nach Poulin [Poul99] verhindern sollte, dass Softwarerepositorys zu nutzlosen Datenspeichern verkommen. Erfolgreiche Reuse-Ansätze aus der Vergangenheit, wie z. B. die „Böblingen Building Blocks", die bis etwa 1990 bei IBM entwickelt wurden [LeSW87] liegen tatsächlich in diesem Bereich.

Offensichtlich hat die technische Entwicklung solche einfachen Bibliothekssysteme aber inzwischen weit überholt, die aktuelle Java Standardbibliothek umfasst zum Beispiel bereits mehrere tausend Klassen, so dass derzeit [z. B. von MXBK05] Prototypen entwickelt werden, die Entwickler in diesem „Dschungel" unterstützen sollen. Ansonsten brachten die vergangenen zehn Jahre nur wenige Neuerungen: Ye und Fischer [YeFi05] verbanden eine Reuse-Bibliothek mit einem proaktiven Empfehlungssystem, das Suchanfragen automatisch und für den Entwickler transparent aus den gerade unter Entwicklung befindlichen Artefakten generiert und mögliche Suchergebnisse unaufgefordert präsentiert. [IYFY05] setzten einen benutzungsbezogenen Ranking-Algorithmus für textbasierte Quelltextsuchen um. Auch in der Wirtschaftsinformatik finden sich zahlreiche Überlegungen zu Komponentenmärkten, auf die z. B. in Fettke et al. [FeLT02] verwiesen wird, aber auch diese bleiben im Hinblick auf die bestehenden technischen Probleme sehr vage, so dass in der Praxis nutzbare Komponentenmärkte weiterhin nicht zu erwarten sind.

Ähnlich düster ist die Lage in der Industrie. Fragt man in Unternehmen nach Softwarerepositorys, so hört man meist von Systemen, die beispielsweise auf dem Concurrent Versions System (CVS), Perforce oder Subversion basieren und lediglich der reinen Versionsverwaltung dienen. Mechanismen, die das gezielte Durchsuchen dieser Softwarespeicher ermöglichten, existieren nicht, so dass selbst innerhalb einer Firma häufig mehrfach funktional identischer Code entwickelt wird. Ähnlich sieht es auch beim weltweit größten Open-Source-Hoster Sourceforge.net aus. Weit über 100.000 Projekte finden sich mittlerweile auf dessen Seiten im Netz, die einzig vorhandene Suchfunktion ist aber nur in der Lage, Projektbeschreibungen nach Stichworten zu durchsuchen; erst eine kürzlich erfolgte Kooperation mit einer Code-Suchmaschine ermöglicht nun immerhin das Durchsuchen von Quelltexten. Neben der bereits erwähnten Webseite Componentsource.com drängen in jüngster Zeit verschiedene solcher Suchmaschinen auf den Markt, die sich zum Ziel gesetzt haben, im WWW verfügbare Open-Source-Projekte durchsuchbar zu machen (z. B. Koders.com & Krugle.com). Ihre Suchfunktionalität arbeitet allerdings nur textbasiert auf einzelnen Quelltextdateien und ist damit noch weit davon entfernt, Komponenten, wie eingangs gefordert, entsprechend einer Spezifikation oder Schnittstellenbeschreibung (wie z. B. aus einem UML-Klassendiagramm) finden zu können.

3 Entwurfsbasiertes Komponentenretrieval

Die grundlegende Idee, wie Komponenten gemäß ihrer Spezifikation gefunden werden können, haben die Autoren erstmals in [HuAt04] vorgestellt. Dort haben sie skizziert, wie eine Kombination aus den zuvor vorgestellten Retrievaltechniken genutzt werden kann, um einfache Komponenten mit großer Präzision liefern zu können. Ein weiteres Kriterium bei der Entwicklung des Ansatzes war eine einfache Verwendbarkeit für den Nutzer, um diesen sowohl von der Erstellung bzw. der Pflege des verwendeten Softwarerepositorys zu entlasten, als auch künftig einen proaktiven Suchablauf zu erlauben. Komponenten werden üblicherweise aus wenigstens zwei verschiedenen Perspektiven beschrieben, die für Komponentenretrieval Verwendung finden können: Eine syntaktische (*interfaces*) und eine semantische Perspektive (*contractually specified*) finden sich auch in heutzutage gängigen Entwicklungsprozessen bzw. bei der Spezifikation von Fachkomponenten wieder. Die Syntax beschreibt die Struktur einzelner Systemteile und wird üblicherweise während des Systemdesigns als UML-Klassendiagramm oder in äquivalenter Notation als Codegerüst oder sog. *Stub* (insbesondere in agilen Methoden wie Extreme Programming, [Beck99]) ausgearbeitet, wie dies in der folgenden Abbildung beispielhaft gezeigt ist (s. Bild 2).

Für die Semantik, also für das Verhalten der Komponente, hat sich bislang noch keine Beschreibungsform durchgesetzt. Üblicherweise werden, angelehnt an „Design by Contract" nach Meyer [Meye91], Vor- und Nachbedingungen für Operationen definiert, wofür von natürlicher Sprache bis hin zu (semi-)formalen Sprachen (wie z. B. der mit UML 2.0 eingeführten Object Constraint Language, OCL) eine ganze Reihe von Möglichkeiten gegeben sind. Ein häufiger Vorbehalt gegen formal semantische Beschreibungen ist allerdings, dass ihre Verwendung ebenso komplex und fehleranfällig sein kann, wie die eigentliche Programmierung. Das Halteproblem macht in der Praxis zudem das Retrieval auf Basis einer formalen Spezifikation unmöglich, da nicht maschinell feststellbar ist, ob eine Komponente ihre Spezifikation tatsächlich einhält.

Inspiriert vom bereits erwähnten Behavior Sampling [PoPi93] sehen die Verfasser in Unit-Tests eine weitere Möglichkeit, das Verhalten einer Komponente zumindest partiell semantisch zu überprüfen. Üblicherweise ist die Erstellung von Testfällen in allen gängigen Softwareprozessen gegen Ende des Entwicklungszyklus vorgesehen, bei den in den letzten Jahren populär gewordenen agilen Entwicklungsmethoden wie Extreme Programming (XP, [Beck99]),

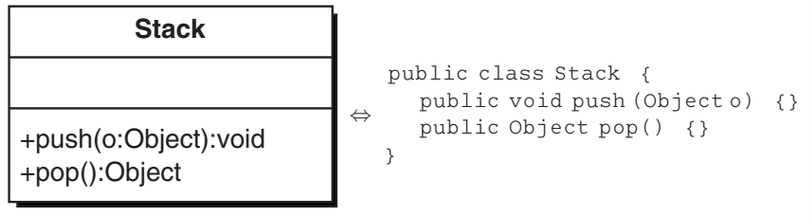

Bild 2 Klassendiagramm einer einfachen Stack-Komponente mit dazu gehörigem Codegerüst

gilt sogar das sog. „test-first“-Prinzip. Dies bedeutet, dass – sobald die Schnittstelle einer Klasse festgelegt wurde – d. h. in XP ihr Stub (vgl. Bild 2 rechts) erstellt wurde, direkt Testfälle für sie geschrieben werden, ohne dass zuvor eine Zeile produktiver Code entwickelt worden wäre. Ein einfacher Testfall für die gezeigte Stack-Komponente unter Verwendung des Quasistandards JUnit (http://junit.org) könnte z. B. wie folgt aussehen:

```
public void testStack() {
    Stack stack1 = new Stack();
    stack1.push( "Lassie" );
    stack1.push( "Flipper" );
    assertTrue( ((String)
        stack1.pop()).
        equals("Flipper") );
    assertTrue( ((String)
        stack1.pop()).
        equals("Lassie") );
}
```

3.1 Funktionsprinzip von Extreme Harvesting

Nach den Erkenntnissen der Verfasser beinhalten Stub und Testfälle bereits alle notwendigen Beschreibungen, um hochpräzises Retrieval in die Praxis umsetzen zu können. Der hier beschriebene Ansatz basiert daher genau auf diesen beiden Elementen und wurde in Anlehnung an XP *Extreme Harvesting* [HuAt04] genannt. Bild 3 zeigt eine schematische Darstellung der Vorgehensweise.

Sobald ein Entwickler Schnittstelle und Testfälle spezifiziert hat (Schritte a und b in Bild 3) kann ein Werkzeug (auch automatisiert im Hintergrund) textuelle Informationen extrahieren, die für eine initiale Suche nach Reuse-Kandidaten erforderlich sind. Eine zusätzliche (evtl. zeitaufwändige) manuelle Definition einer Suchanfrage ist überflüssig. Die Abbildung zeigt in Schritt c eine Suche mit Hilfe einer Komponentensuchmaschine aus dem WWW. In Schritt d untersucht das Tool, ob die gefundenen Kandidaten mit der Suchanfrage syntaktisch übereinstimmen (oder evtl. adaptiert werden können). Für den Fall, dass wenige oder gar keine Kandidaten gefunden werden, kann an dieser Stelle z. B. mit Hilfe von Synonymen oder anderen Heuristiken nach weiteren Kandidaten gesucht werden. In Schritt e werden die Kandidaten mit passenden Signaturen kompiliert. Dabei können bestimmte Fehlermeldungen (wie z. B. fehlende Klassen) automatisch erkannt und häufig direkt behoben werden. Sobald eine Komponente fehlerfrei kompiliert, wird sie, wie in Schritt f am Beispiel von JUnit gezeigt, automatisiert getestet. Durchläuft sie alle Testfälle fehlerfrei, ist eine zur vorgegebenen Spezifikation passende Komponente gefunden, die dem Entwickler angeboten wird. Extreme Harvesting kann somit auch als ein Filterungsprozess verstanden werden, der die Gesamtmenge der vorhandenen Komponenten in einem beliebigen Repository über textbasiertes und syntaktisches Matching so weit verkleinert, bis eine partielle semantische Überprüfung durch Testen in angemessener Zeit durchführbar ist.

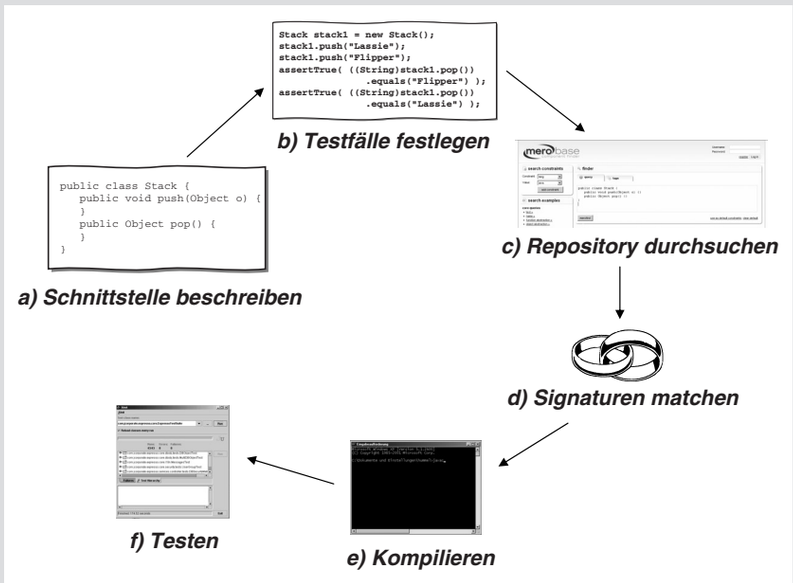

Bild 3 Schematische Darstellung von Extreme Harvesting

3.2 Machbarkeitsnachweis

In Ermangelung eines ausreichend großen, eigenen Repositorys stützte sich der erste Prototyp [HuAt04] auf die reguläre Version der Suchmaschine von Google. Diese verfügt über eine Schnittstelle, um sie aus Programmen heraus ansprechen zu können sowie über nicht offiziell dokumentierte Eigenschaften in ihrem Dateitypenfilter, so dass sie zu diesem Zweck verwendet werden kann. Auf Grund der beschränkten Möglichkeiten der Suchmaschinen und häufig unzuverlässiger Programmierschnittstellen war der praktische Nutzen allerdings stark limitiert. Um diesen Einschränkungen zu entgehen, haben die Autoren zwischenzeitlich eine eigene Komponentenkollektion zusammengestellt, die erstmalig Inhalte aus dem WWW und den Versionierungssystemen der großen Open-Source-Hoster kombiniert und insgesamt knapp 10 Millionen Java-, C- und C#-Komponenten sowie mehrere tausend Werbservices enthält. Dieser Bestand wurde mit Hilfe eines Dual-Core-Servers und verschiedenen Open-Source-Werkzeugen innerhalb mehrerer Monate weitgehend automatisch indexiert und kann seit kurzem unter merobase.com über eine im „google style“ gehaltene Benutzerschnittstelle durchsucht werden.

4 Evaluation

Die Evaluation von großen (Web-)Suchmaschinen ist generell schwierig, da der Recall häufig nicht bestimmbar ist [LeHö07, 165] und aussagekräftige bzw. glaubhafte Beispiele gerade im Reuse-Bereich nur schwer zu finden sind. Bisherige Arbeiten mit ihren relativ kleinen und proprietären Repositorys mussten Versuchspersonen vor Aufgaben stellen, die mit dem Inhalt der Repositorys lösbar waren, um überhaupt verwertbare Ergebnisse zu erhalten. Somit waren positive Aussagen häufig vorherbestimmt [z. B. bei YeFi05]. Mit der heute erreichten Größe der Repo-

sitorys muss allerdings die Precision in den Vordergrund rücken, da beispielsweise eine Suche nach dem Schlüsselwort „stack“ bei Merobase annähernd 200.000 Ergebnisse liefert, so dass eine nähere Beschreibung der gewünschten Art von Stack und ein ausgereiftes Matching dringend geboten sind. So konzentrierten sich Inoue et al. [IYFY05, 219] bereits auf das Ranking der Suchergebnisse, benutzten aber zur Evaluation nur sehr ungenaue Aufgabenstellungen, so dass die Aussagekraft der gewonnenen Ergebnisse trotz eines größeren Repositorys (von ca. 130.000 Komponenten) fraglich bleibt. Die Verfasser denken, dass die vollständige Spezifikation einer Komponente ein logischer Ausgangspunkt für eine Suche ist und haben daher auf ihrer Komponentenkollektion mit mehreren Millionen Einträgen den im Folgenden beschriebenen Vergleich verschiedener Retrievaltechniken durchgeführt.

4.1 Versuchsaufbau und -Durchführung

An Hand von konkreten Signaturen, die bereits in [HuAt06, 308] aus der Literatur gesammelt worden waren, wurde die Precision von fünf Retrievaltechniken miteinander verglichen. Am Beispiel der in Bild 2 gezeigten Stack-Komponente lassen sich für die verschiedenen Retrievalmechanismen in Merobase wie folgt Suchanfragen ableiten. Zunächst können für rein textbasierte Suchen, wie sie die meisten Software-Suchmaschinen im WWW unterstützen, z. B. alle Worte aus der Komponentenschnittstelle extrahiert werden (also: „stack push pop object“). Eine Beschränkung auf Klassen- oder Methodennamen, wie sie auch von Koders und Krugle unterstützt wird, ergibt: „name:stack method:push method:pop object“ als Suchanfrage (die Verfasser nennen das „namensbasiert“). Signature Matching sucht nur nach den Typen aus der Signatur einer Komponente (d. h.: Object → void und void → Object für den Stack). Das „provided interface“ einer Komponenten, wird für schnittstellenbasierte Suchen verwendet, d. h. als Sucheingabe kann direkt der in Bild 2 rechts gezeigte Stub verwendet werden. Extreme Harvesting nutzt zusätzlich noch JUnit-Testfälle, wie beispielhaft in Abschnitt 3 gezeigt, für eine näherungsweise semantische Überprüfung. Merobase unterstützt die vier zuerst genannten Suchverfahren direkt, das letztgenannte wurde aus Sicherheitsgründen in ein externes Tool ausgelagert.

Als relevant erachteten die Autoren ausschließlich den Quelltext von Java-Klassen, die die vorgegebene Spezifikation (also Signatur und dazu erstellte Testfälle) direkt erfüllt haben. Evtl. gefundene Komponenten mit geringfügig abweichenden Schnittstellen (z. B. anderen Namen bzw. Parametertypen oder weiteren Abhängigkeiten) wurden nicht adaptiert und daher nicht als relevant gezählt, auch wenn dies evtl. mit wenigen manuellen Änderungen möglich gewesen wäre. Einzige Ausnahmen waren eine abweichende Groß-/Kleinschreibung und mögliche *Exceptions*, die automatisch angepasst bzw. toleriert wurden. Unter diesen Voraussetzungen liefert ein operationales Verfahren wie Extreme Harvesting (XH) mit ausreichend guten Testfällen immer eine 100 %ige Präzision (was schon [PoPu93, 292] für zufällige Samples gezeigt haben), weshalb es hier eingesetzt wurde, um die von den anderen Techniken gelieferten Kandidaten auf die Einhaltung der vorgegebenen Spezifikation zu überprüfen. Die folgende Tabelle zeigt, wie viele aus maximal den ersten 25 Ergebnissen von Merobase jeweils als relevant getestet wurden. Für Extreme Harvesting wird entsprechend jeweils nur die größte Anzahl von relevanten Ergebnissen in der Tabelle angegeben.

4.2 Diskussion

Die Präzisionsunterschiede zwischen den einzelnen Techniken sind – außer zwischen „Text“ und „Name“ – alle paarweise statistisch signifikant für $\alpha = 0{,}05$. Das etwas überraschende Ergebnis der isPrime-Methode („Text“ liefert hier deutlich mehr Ergebnisse als „Schnittstelle“) lässt sich dadurch erklären, dass merobase für textbasierte Suchen einen optimierten Algorithmus verwendet, der eine andere Ergebnisreihenfolge liefern kann, als der syntaxbezogene Algorithmus. Das allgemein schwache Abschneiden des Signature Matching ist insbesondere durch die Größe des Repositorys (es enthält gut 3,9 Millionen Java-Source-Klassen), in dem gängige Signaturen, wie z. B. int → boolean bei isLeapYear zehntausendfach vorkommen, zu erklären. Insgesamt belegen die Versuchsergebnisse damit die Annahme, dass für die Precision der verschiedenen Techniken im Hinblick auf eine vorgegebene

Tabelle 1 Ergebnisse des Vergleichs verschiedener Retrievaltechniken

Retrievaltechnik Komponente	Signatur	Text	Name	Schnittstelle	XH

`Stack(`
`  push(Object):void`
`  pop():Object`
`)`
	0/25	0/25	3/25	3/25	3

`randomNumber(int,int):int`	0/25	0/25	0/25	5/25	5
`sort(int[]):int[]`	1/25	0/25	0/25	16/25	16
`reverseArray(int[]):int[]`	0/25	0/25	2/25	4/7	4
`isPrime(int):boolean`	0/25	4/25	9/25	5/25	9
`sqrt(double):double`	0/25	1/25	6/25	12/25	12
`isLeapYear(int):boolean`	0/25	8/25	7/25	13/25	13
`randomString(int):String`	0/25	0/25	2/25	3/25	3
`gcd(int,int):int`	0/25	8/25	10/25	12/25	12
`replace(String,String,` `String):String`	1/25	3/25	0/25	18/25	18
`getMinMax(int[]):int[]`	1/25	2/25	2/25	2/4	2
Durchschnitt ⇒ Precision	0,27/25 1,1%	2,36/25 9,4%	3,72/25 14,9%	8,45/21,45 39,4%	8,82 100%
Standardabweichung	2%	12%	15%	21%	–
Varianz	0,03	1,5	2,2	4,6	–

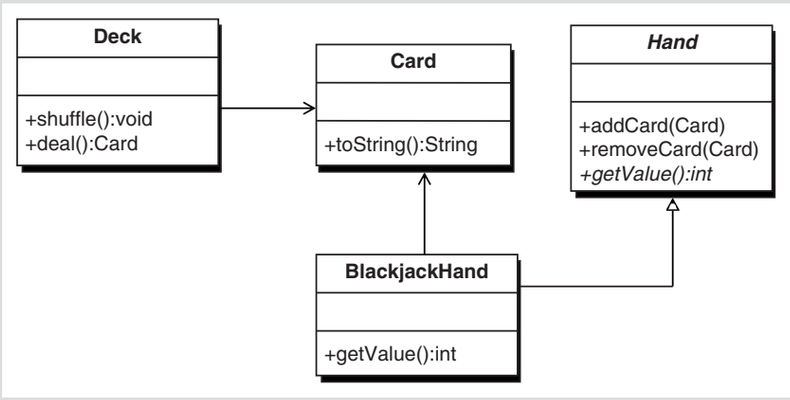

Bild 4 Vereinfachtes Klassendiagramm eines aus dem WWW „geernteten" Blackjack-Ensembles

Spezifikation folgende Rangfolge angenommen werden kann:

Signatur < Text < Name < Schnittstelle < Extreme Harvesting

Es wird erkennbar, dass die in Merobase integrierte schnittstellenbasierte Suche in Sachen Retrievalgenauigkeit einen Schritt in die richtige Richtung bedeutet, zumindest um für den aufwändigen operationalen Teil von Extreme Harvesting bereits mit hoher Wahrscheinlichkeit Kandidaten zu liefern, die auch tatsächlich die geforderte Spezifikation erfüllen. Durch diese Kombination verschiedener Retrievaltechniken sind nach Erfahrungen der Verfasser maximal drei Testfälle – und damit etwa 75 % weniger „Samples" als bei [PoPi93, 292] – absolut ausreichend, um fehlerhafte Kandidaten zuverlässig auszusondern.

Der genaue Einfluss der Testfallqualität auf die Precision bietet natürlich ebenso Stoff für weitere Untersuchungen, wie bisher noch in Planung befindliche, weitere Evaluationsschritte. Diese sehen zunächst ein kontrolliertes Experiment vor, bei dem Gruppen von Studenten mit und ohne Unterstützung eines gerade fertig gestellten, proaktiven Suchwerkzeuges verglichen werden sollen. Die Verfasser erhoffen sich davon den Nachweis, dass Entwickler, die automatisch sinnvolle Vorschläge aus einer großen Komponentenbibliothek erhalten, deutlich produktiver arbeiten können als solche, die alle Aufgaben von Grund auf neu – d. h. nur mit Hilfe von Standardbibliotheken – implementieren müssen.

5 Laufende und zukünftige Arbeiten

Wie zu Beginn bereits diskutiert, sind für einen praktischen Einsatz in Komponentenmärkten auch komplexere (Fach-)Komponenten, die aus zahlreichen Klassen oder Subkomponenten bestehen können, eine unabdingbare Voraussetzung. Das im folgenden Bild gezeigte Blackjack-Ensemble, das aus vier Klassen besteht, ist ein Beispiel für einen solchen Fall. Die Verfasser konnten dafür mit einer einfachen Heuristik (das Tool beginnt die Suche mit einer beliebigen Klasse und versucht Fundstellen für die übrigen Klassen aus den gefundenen URLs herzuleiten) zwei passende Ensembles im WWW finden.

Intuitiv klar ist, dass eine steigende Komplexität von Komponenten dazu führt, dass passende Exemplare schwieriger zu finden und anzupassen sind, was bereits 1997 von Sametinger [Same97, 3] angeführt wurde. Besonders problematisch wird an diesem Punkt allerdings die fehlende Unterstützung für „echte" Komponenten in den heute gängigen Programmiersprachen, die nur Konzepte für Klassen und Packages enthalten. So würde ein Nutzer wahrscheinlich nicht die gezeigten Interna spezifizieren wollen, sondern allenfalls eine Komponente *Blackjack* und deren Funktionalitäten wie z. B. *play,* die allerdings in heutigen Programmiersprachen höchstens immanent in der Nutzerschnittstelle verborgen wären. Wie solche Komponenten in aktuellen, auf einzelne Dateien spezialisierten Suchmaschinen als solche gesucht, erkannt und zurückgeliefert werden könnten, ist in der Literatur bisher noch nicht als Problem erkannt. Auf der Hand liegt zwar die Idee, als Sucheingabe die standardisierte Darstellung eines Klassen- o. Komponentendiagramms (also nach heutigem Stand XMI-Daten) zu verwenden und Ergebnisse gemäß der RAS [OMG05] gepackt zurückzuliefern. Um aber z. B. die genannte Funktionalität der Blackjack-Komponente aus der in Bild 4 gezeigten Realisierung abzuleiten, ist ein umfangreiches Reverse Engineering nötig, das ohne menschliches Eingreifen womöglich überhaupt nicht zu vollbringen sein wird. Völlig offen ist zudem, wie das Zusammensetzen von Komponenten vonstatten gehen soll, wenn sie nicht „am Stück", d. h. im gleichen Paket liegend, gefunden werden können, was mit steigender Komplexität natürlich zunehmend wahrscheinlich wird. Es gibt gegenwärtig weder einen Ansatzpunkt, nach welchem Teil zuerst gesucht werden sollte, noch Strategien, wie solche Kombinationen und evtl. notwendige Schnittstellenanpassungen, bei denen im schlimmsten Fall alle möglichen, d. h. evtl. zehntausende Permutationen erstellt, kompiliert und getestet werden müssten, effektiv durchgespielt werden könnten.

6 Resümee

Softwarekomponentenmärkte wurden zwar bereits 1968 vorhergesagt, funktionieren aber auf Grund zahlreicher Faktoren bis heute nicht wie erwünscht. In diesem Beitrag wurde das Problem des Auffindens von Komponenten in sehr großen Repositorys betrachtet und ein Ansatz vorgestellt, der das Potenzial aufweist, die mangelhafte Precision und Benutzbarkeit älterer Techniken zumindest auf Klassenebene zu überwinden. Dazu wurden Ideen bekannter Techniken zum sog. Extreme Harvesting weiterentwickelt, das eng an moderne, testgetriebene Entwicklungsmethoden, wie Extreme Programming, angelehnt ist. Dadurch ist eine hohe Precision ohne Mehraufwand für den Entwickler gewährleistet, da Extreme Harvesting Suchanfragen aus den Assets generieren kann, die im Verlaufe eines Entwicklungsprozesses ohnehin erstellt werden (sollten). Mit Hilfe der neu erstellten Komponentensammlung, die unter merobase.com von jedermann genutzt werden kann, konnte erstmals in einer Untersuchung belegt werden, dass schnittstellenbasiertes Retrieval zu einer

gegebenen Klassenspezifikation signifikant präzisere Ergebnisse liefert als bisherige Retrievaltechniken und daher gut als Vorfilter für die teure operationale Prüfung in Extreme Harvesting eingesetzt werden kann. Die hohe Precision des Ansatzes geht zwar zu Lasten des Recall, so lange aber genügend Ergebnisse zur Verfügung stehen und die Precision bis zu einem *Cut-Off*-Wert von etwa 20 Ergebnissen hoch genug bleibt, ist das für den Nutzer unerheblich [LeHö07, 165].

Für einen praktischen Einsatz verlangt Extreme Harvesting ferner nach einer abgeschirmten Testumgebung, in der zu testender, fremder Code keinen Schaden anrichten kann. Über den gerade vollendeten Aufbau eines entsprechenden Systems mit Hilfe virtueller Server werden die Verfasser an anderer Stelle berichten. Mögliche rechtliche Probleme, wie z. B. die Nichteinhaltung von Copyright oder Open-Source-Lizenzen, sind zwar ebenfalls nicht von der Hand zu weisen, generell verweisen Suchmaschinen allerdings nur auf öffentlich zugängliche Informationen, für den korrekten Umgang mit gefundenem Material ist allein der Nutzer verantwortlich. Zusammenfassend sehen die Autoren in Extreme Harvesting einen ersten Schritt zu einer besseren Benutzbarkeit von Komponentenmärkten, der allerdings nicht alle Visionen aus Wirtschaftsinformatik und Informatik hinsichtlich der Granularität der unterstützten Komponenten erfüllen kann. Dazu werden künftig noch weitere Forschungen, insbesondere über Aufbau und Adaption von zusammengesetzten Komponenten, notwendig sein.

Literatur

[Alpa80] *Alpar, P.:* Computergestützte interaktive Methodenauswahl. FWI-Verlag, Frankfurt 1980.

[ABBK02] *Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E. et al.:* Component-based Product Line Engineering with UML. Addison Wesley, New York 2002.

[BaBM96] *Basili, V.; Briand, L.; Melo, W.:* How reuse influences productivity in object-oriented systems. In: Communications of the ACM 39 (1996) 10, S. 104–116.

[BaRi99] *Baeza-Yates, R.; Ribeiro-Neto, B.:* Modern Information Retrieval. Addison-Wesley, New York 1999.

[Beck99] *Beck, K.:* Extreme Programming Explained: Embrace Change. Addison-Wesley, New York 1999.

[Broo87] *Brooks, F. P.:* No Silver Bullet – essence and accident in software engineering. In Computer 20 (1987) 4, S. 10–19.

[ClNo02] *Clemens, P.; Northrop, L.:* Software Product Lines: Practices and Patterns. Addison-Wesley, New York 2002.

[DSWi99] *D'Souza, D.; Wills A.:* Objects, Components and Frameworks with UML. The Catalysis Approach. Addison-Wesley, New York, 1999.

[FeLT02] *Fettke, P.; Loos, P.; v. d. Tann, M.:* Entwicklung eines Repositoriums für Fachkomponenten auf Grundlage des Vorschlags der vereinheitlichten Spezifikation von Fachkomponenten – Analyse von Problemen und Diskussion von Lösungsalternativen. In: Modellierung und Spezifikation von Fachkomponenten: 3. Workshop im Rahmen der MKWI 2002, Nürnberg, 2002.

[FrPo94] *Frakes, W. B.; Pole, T. P.*: An empirical study of representation methods for reusable software components. In: IEEE Transactions on Software Engineering 20 (1994) 8, S. 617–630.

[FrFo96] *Frakes, W. B.; Fox, C. J.*: Quality improvement using a software reuse failure modes model. In: IEEE Transactions on Software Engineering 22 (1996) 4, S. 274–279.

[GHJV95] *Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.:* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, New York 1995.

[HuAt04] *Hummel, O.; Atkinson, C.*: Extreme Harvesting: Test Driven Discovery and Reuse of Software Components. In: Proceedings of the International Conference on Information Reuse and Integration, Las Vegas 2004, S. 66–72.

[HuAt06] *Hummel, O.; Atkinson, C:* Using the Web as a Reuse Repository. In: Proceedings of the International Conference on Software Reuse, Torino 2006, S. 298–311.

[IYFY05] *Inoue, K.; Yokomori, R.; Fujiwara, H.; Yamamoto, T. et al.:* Ranking Significance of Software Components Based on Use Relations. In: IEEE Transactions on Software Engineering 31 (2005) 3, S. 213–225.

[KLMM97] *Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C. et al.:* Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming, 1997.

[LeSW87] *Lenz, M.; Schmid, H.; Wolf, P. W.:* Software reuse through building blocks. In *W. Tracz* (Hrsg.): Software Reuse: Emerging Technology, Computer Society Press 1987, S. 100–108.

[LeHö07] *Lewandowski, D.; Höchstötter, N.:* Qualitätsmessung bei Suchmaschinen – System- und nutzerbezogene Evaluationsmaße. Informatik-Spektrum 30 (2007) 3, S. 159–169.

[MXBK05] *Mandelin, D.; Xu, L.; Bodik, R.; Kimelman, D.:* Jungloid mining: helping to navigate the API jungle. In Proceedings of the Conference on Programming Language Design and Implementation, 2005, S. 48–61.

[McIl68] *McIlroy, D.:* Mass-Produced Software Components. In Proceedings of a conference sponsored by the NATO Science Committee, Garmisch 1968.

[Meye91] *Meyer, B.:* Design by Contract, In *Mandrioli, D.; Meyer, B.* (Hrsg.): Advances in Object-Oriented Software Engineering, Prentice-Hall, Upper Saddle River 1991.

[Micr06] Microsoft: UBR shutdown FAQ. http://uddi.microsoft.com/about/FAQshutdown.htm, 2006, Abruf am 2006-10-09.

[MiMM98] *Mili, A.; Mili, R.; Mittermeir, R.:* A Survey of Software Reuse Libraries. In: Annals of Software Engineering 5 (1998), S. 349–414.

[NuCh90] *Nunamaker, J. F.; Chen, M.:* Systems Development in Information System Research. In: Proceedings of the Annual International Conference on System Sciences, Hawaii (1990), S. 631–640.

[OMG05] OMG: Reusable Asset Specification. http://omg.org/technology/documents/formal/ras.htm, 2005, Abruf am 2006-10-09.

[Parn72] *Parnas, D. L.:* On the Criteria to be Used in Decomposing Systems into Modules. In: Communications of the ACM 15 (1972) 12, S. 1053–1058.

[PoPi93] *Podgurski, A.; Pierce, L.:* Retrieving Reusable Software by Sampling Behavior. In: ACM Transactions on Software Engineering and Methodology 2 (1993) 3, S. 286–303.

[Poul99] *Poulin, J.:* Reuse: Been There, Done That. In: Communications of the ACM 42 (1999) 5, S. 98–100.

[Same97] *Sametinger, J.:* Software Engineering with Reusable Components, Springer, Heidelberg 1997.

Abstract

Improving the Retrieval Efficiency of Software Component Markets

Component-based software reuse has been widely accepted as a way of making software development faster, better, and cheaper. However, component markets of the kind envisaged for many decades have not yet become a useful tool in mainstream development. In this article, the authors discuss the underlying problems and present a new approach called "Extreme Harvesting" for test-driven component retrieval. They present examples that demonstrate how this concept works "in vitro", demonstrate its precision with the help of an experiment and discuss further challenges to be solved to make this approach of practical utility.

Keywords: Component-based Software Development, Component Retrieval, Reuse

[Seac99] *Seacord, R.:* Software Engineering Component Repositories, In: Proceedings of the ICSE Workshop on Component-Based Software Engineering, Los Angeles 1999.

[Stal02] *Stal, M.:* Web Services: Beyond Component-Based Computing. In: Communications of the ACM 45 (2002) 10, S. 17–76.

[Szyp02] *Szyperski, C.:* Component Software. Addison-Wesley, New York 2002.

[Turo02] *Turowski, K.* (Hrsg.): Vereinheitlichte Spezifikation von Fachkomponenten: Memorandum des Arbeitskreises Komponentenorientierte betriebliche Anwendungssysteme. Universität Augsburg 2002.

[YeFi05] *Ye, Y.; Fischer, G.:* Reuse-Conducive Development Environments. In: Journal of Automated Software Engineering 12 (2005) 2, S. 199–235.

[ZaWi95] *Zaremski, A. M.; Wing, J. M.:* Signature Matching: A Tool for Using Software Libraries. In: ACM Transactions on Software Engineering and Methodology 4 (1995) 2, S. 146–170.