Code Conjurer: Pulling Reusable Software out of Thin Air

Article *in* IEEE Software · October 2008

DOI: 10.1109/MS.2008.110 · Source: IEEE Xplore

Oliver Hummel
Karlsruhe Institute of Technology
56 PUBLICATIONS **704** CITATIONS
SEE PROFILE

Werner Janjic
Asseco Solutions AG
17 PUBLICATIONS **302** CITATIONS
SEE PROFILE

Colin Atkinson
Universität Mannheim
212 PUBLICATIONS **5,724** CITATIONS
SEE PROFILE

Code Conjurer: Pulling Reusable Software out of Thin Air

Oliver Hummel, *Perot Systems Germany*

Werner Janjic and Colin Atkinson, *University of Mannheim*

A tool that automatically finds and presents suitable reusable software components to developers can help speed the development process.

For many years, the IT industry has sought to accelerate the software development process by assembling new applications from existing software assets. However, true component-based reuse of the form Douglas McIlroy[1] envisaged in the 1960s is still the exception rather than the rule, and most of the systematic software reuse practiced today uses heavyweight approaches such as product-line engineering or domain-specific frameworks. By component, we mean any cohesive and compact unit of software functionality with a well-defined interface—from simple programming language classes to more complex artifacts such as Web services and Enterprise JavaBeans.

Historically, three main reasons explain why component-oriented reuse has failed to take off:

- Not enough good components were around to make it worthwhile. Indeed, during the golden years of software reuse research in the 1980s and 1990s, researchers considered themselves fortunate to have a repository with even a few hundred components.
- The retrieval technologies used to find suitable components matching a user's query were crude and often returned a high proportion of unsuitable components or missed many relevant ones.[2]
- The overhead involved in using the retrieval technology to find suitable components and evaluate their fitness for purpose was too high.

As a result, the balance of effort and risk involved in software reuse always compared unfavorably to building components from scratch.

Recent developments have improved the situation. The rise of the open source movement and cheap, high-bandwidth Internet connectivity have given software developers access to vast swathes of free software, so the number of available components is no longer a significant problem. Also, in the last two years, high-performance code-search engines (such as Koders, Google Code Search, and Merobase) have emerged that provide better ways of retrieving assets from this code base, going beyond simple keyword matching. The third problem regarding retrieval overhead has changed little, however, and is now the main barrier to the routine reuse of software components and the emergence of software component marketplaces.

This is where tools such as Code Conjurer, developed at the University of Mannheim, aim to make a difference (see the "Repository-Driven Reuse Assistance Tools" sidebar for a description of similar tools). As its name implies, from a developer's viewpoint, Code Conjurer effectively "conjures up" software components out of thin air and makes them available with almost no effort on the user's part. It does this by tapping into the vast resource of components offered by a modern code-search engine to deliver high-relevance software reuse recommendations with minimal, if any, disturbance to a developer's normal practices. Moreover, it dramatically reduces the risk and effort involved in

Repository-Driven Reuse Assistance Tools

The idea of accelerating software development by tapping into the knowledge wrapped up in existing components isn't new.[1] Yunwen Ye's CodeBroker was one of the first tools to explore this idea in the form of a proactive invocation service tightly integrated into the well-known Emacs editor.[2] While developers worked on their source code, CodeBroker offered coding suggestions on the basis of information garnered from similar components in the repository. Ye identified two fundamentally distinct ways of getting this information from the repository:

- the classic pull or reactive approach, in which a user actively browses or searches for information, and
- the push or proactive approach, in which a tool monitors the user's activities and offers information it considers useful in a specific context.

However, CodeBroker required users to annotate their components with active comments, significantly increasing the effort involved in developing them, and its repository never grew beyond a few hundred components.

More recent tools have built on CodeBroker's idea of proactive recommendation. For example, Rascal recommends how and when to call the methods of objects from common libraries such as Java Swing, based on an analysis of existing classes.[3] It uses collaborative filtering, similar to that used in online shopping sites, to recommend products on the basis of those similar customers bought.

Tools such as Prospector,[4] ParseWeb,[5] and Strathcona[6] follow a similar path but aren't proactive. They focus on helping developers navigate through the API jungle created by today's standard libraries and frameworks. Because such libraries contain far more assets than even the most sophisticated reuse libraries and tools of a decade ago (for example, Sun's JDK 6 contains more than 3,500 classes in more than 200 packages), such navigation help is urgently required.

Prospector and ParseWeb support developers by recommending method invocation sequences that yield a required destination data type from given input parameter types. Strathcona provides source code examples and structural context for the code fragment under development.

CodeGenie[7] is another recently released Eclipse plug-in that explores the notion of test-driven reuse.[8] However, it isn't proactive and requires developers to manually test all reuse candidates locally in their development environments.

References

1. D. McIlroy, "Mass-Produced Software Components," *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 1969, pp. 138–155.
2. Y. Ye, "Supporting Component-Based Software Development with Active Component Repository Systems," PhD dissertation, Faculty of the Graduate School, Univ. of Colorado, 2001.
3. F. McCarey, M. Ó Cinnéide, and N. Kushmerick, "RASCAL: A Recommender Agent for Agile Reuse," *Artificial Intelligence Rev.*, vol. 24, nos. 3–4, 2005, pp. 253–276.
4. D. Mandelin et al., "Jungloid Mining: Helping to Navigate the API Jungle," *Proc. Conf. Programming Language Design and Implementation*, ACM Press, 2005, pp. 48–61.
5. S. Thummalapenta and T. Xie, "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web," *Proc. Int'l Conf. Automated Software Eng.*, ACM Press, 2007, pp. 204–213.
6. R. Holmes, R.J. Walker, and G.C. Murphy, "Approximate Structural Context Matching: An Approach for Recommending Relevant Examples," *IEEE Trans. Software Eng.*, vol. 32, no. 12, 2006, pp. 952–970.
7. O.A.L. Lemos, S. Bajracharya, and J. Ossher, "CodeGenie: A Tool for Test-Driven Source Code Search," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Lanugages and Applications*, ACM Press, 2007, pp. 917–918.
8. O. Hummel and C. Atkinson, "Extreme Harvesting: Test Driven Discovery and Reuse of Software Components," *Proc. IEEE Int'l Conf. Information Reuse and Integration*, IEEE Press, 2004, pp. 66–72.

exploring reuse opportunities by using standard unit tests created during the normal development process to perform a semantic assessment of the reuse candidates. We first demonstrated the feasibility of test-driven search in 2004.[3] We recently released a beta version of Code Conjurer as a freely available plug-in that seamlessly integrates code search and reuse functionality into the Eclipse Java development environment (see http://codeconjurer.sourceforge.net).

Code-search engines

Code-search engines provide the backbone for the new generation of reuse support tools—for example, CodeGenie relies on Sourcerer[4] while ParseWeb uses Google Code Search. Code Conjurer is driven by the Merobase component search engine, which uses Lucene, Apache's open source information-retrieval library, to index programming language units from various open source code repositories (such as Sourceforge, Google Code, or the Apache projects) as well as the open Web. When crawling for code, Merobase's analysis software identifies the basic abstraction implemented by a module and stores it in a language-agnostic description format. The description's most important element is the abstraction's name, but other key features are also stored, such as method names and parameter signatures.

Table 1 summarizes the code-search engines we were aware of in summer 2007, when we last performed a systematic comparison. We focused on Java components because Code Conjurer currently focuses on the Java language. As the table shows, Merobase currently indexes more than 10 million code modules, giving Code Conjurer access to a re-

Table 1

Overview of specialized code and component search engines

Search engine	No. of indexed files	No. of Java files	Retrieval algorithms
Codase (www.codase.com)	< 1 million	300,000	Keyword matching of hosted open source codes
Codefetch (www.codefetch.com)	< 100,000	< 100,000	Keyword matching of source code in programming books
Component Source (www.componentsource.com)	> 1,000	> 100	Keyword matching of component descriptions in a marketplace
CsourceSearch (www.csourcesearch.net)	1 million	0	Keyword and name matching on popular C/C++ open source packages
Google Code Search (www.google.com/codesearch)	> 10 million	2.5 million	Keyword matching of open source code with regex support
Koders (www.koders.com)	> 1 million	600,000	Keyword and name matching of codes from large open source hosters
Krugle (www.krugle.com)	> 10 million	3.5 million	Keyword and name matching in open source code and search for technical Web pages
O'Reilly Code Search Beta (labs.oreilly.com/code/)	100,000	15,000	Keyword matching and fielded searches on code in O'Reilly programming books
Merobase (www.merobase.com)	> 10 million	8 million	Keyword and name matching, signature matching, and interface-based and test-driven retrieval on open source code, binary components, and Web services
Planet Source Code (www.planetsourcecode.com)	< 100,000	< 50,000	Keyword matching on source code and programming tutorials
Sourcerer (sourcerer.ics.uci.edu/sourcerer/search/index.jsp)	250,000	250,000	Keyword and topological matching on indexed open source code ranked by CodeRank
Spars-J (demo.spars.info)	> 300,000	300,000	Keyword and name matching on open source XML, Java, and JSPs based on component rank and keyword rank algorithms
Ucodit (www.ucodit.com)	> 100,000	> 100,000	Name matching on classes and methods on Java and C open source code

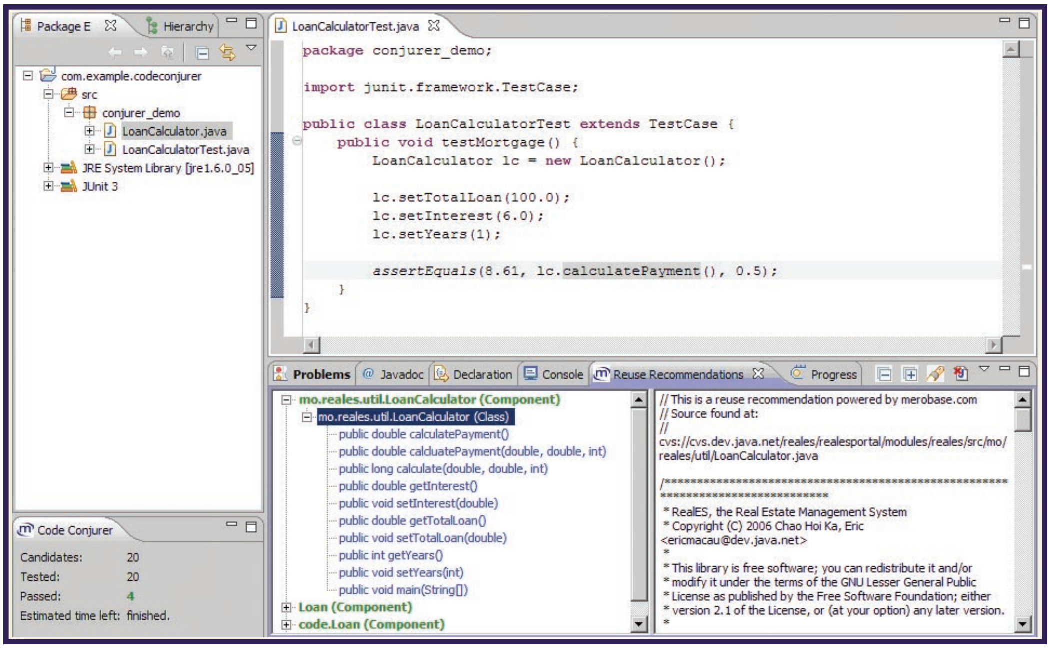
pository several orders of magnitude greater than that of most first-generation reuse recommendation tools (see the "Repository-Driven Reuse Assistance Tools" sidebar).

Traditional component-retrieval techniques are often criticized for being too imprecise or complicated to use. Most of the numbers backing these criticisms, however, are based on experiments with only small collections. To gain a better understanding of these techniques' effectiveness, we implemented several well-known and new retrieval algorithms in Merobase and performed some basic experiments on the large collections it supports.[5] These experiments confirmed that older retrieval approaches, such as keyword or signature matching, are indeed imprecise, as is the name-based matching approach that many search engines still use today. To address this problem, we developed a small query language that lets users define search requests in the style of programming language interface descriptions. For example, to search for components representing object-oriented abstractions, the user simply provides the abstraction's name followed by a list of UML-like function specifications enclosed in brackets. So, the query

```
Customer (
  getAddress():String;
  setAddress(String):void;
)
```

will search for components named `Customer` that offer an operation (a method or function) named `setAddress` with an `In` parameter of type `String` and an operation `getAddress` with an `Out` parameter or return value of type `String`.

Our investigations show that this interface style of query definition delivers a precision of 30 to 50 percent depending on the interface complexity. Nevertheless, traditional text-based search techniques alone clearly can't provide the precision needed to make component reuse a viable



```
package conjurer_demo;

import junit.framework.TestCase;

public class LoanCalculatorTest extends TestCase {
    public void testMortgage() {
        LoanCalculator lc = new LoanCalculator();

        lc.setTotalLoan(100.0);
        lc.setInterest(6.0);
        lc.setYears(1);

        assertEquals(8.61, lc.calculatePayment(), 0.5);
    }
}
```

```
Candidates:             20
Tested:                 20
Passed:                 4
Estimated time left:    finished.
```

```
mo.reales.util.LoanCalculator (Component)
  mo.reales.util.LoanCalculator (Class)
    public double calculatePayment()
    public double calcluatePayment(double, double, int)
    public long calculate(double, double, int)
    public double getInterest()
    public void setInterest(double)
    public double getTotalLoan()
    public void setTotalLoan(double)
    public int getYears()
    public void setYears(int)
    public void main(String[])
Loan (Component)
code.Loan (Component)
```

```
// This is a reuse recommendation powered by merobase.com
// Source found at:
//
cvs://cvs.dev.java.net/reales/realesportal/modules/reales/src/mo/
reales/util/LoanCalculator.java

/*****************************************************************
*****************************
 * RealES, the Real Estate Management System
 * Copyright (C) 2006 Chao Hoi Ka, Eric
<ericmacau@dev.java.net>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
```

Figure 1. Test-driven reuse recommendation and status view. The developer will typically see only the Code Conjurer status view (lower left-hand corner). The Reuse Recommendations window (bottom right) appears when the user asks to see the positively tested components.

proposition, so we also need an additional, fundamentally different approach to enhance search result quality.

Test-driven search

Obviously, the usefulness of the reuse recommendations provided by a tool such as Code Conjurer depends on the recommendations' quality. Users must still examine the components to decide whether or not to reuse them. And users will only deem this effort worthwhile if there's a sufficiently high chance that the proposed components will do what they want. A tool that gets a reputation for making poor recommendations will quickly fall out of use.

Although Merobase finds matching components on the basis of their identifiers and signatures reasonably well, the proportion of unsuitable components in the result set is usually rather high. Fortunately, software is unique among the textual documents indexed by search engines in having executable and observable dynamic behavior.[2] Code Conjurer and Merobase therefore exploit the increasing popularity of agile development methods, which emphasize test-driven development, to dramatically improve the delivered recommendations' quality. They do this by testing the components discovered by the underlying search algorithm and filtering out those that fail. So, Code Conjurer's recommendations are guaranteed to match the user's needs because they've passed the user-defined tests.

Suppose a software engineer is developing a loan calculator component as part of a financial software suite. If the engineer is using an agile development approach such as Extreme Programming, he or she will likely write a test case before developing the production code. The JUnit test shown in the Java editor of the Eclipse screenshot in Figure 1 illustrates this type of reuse.

Normally, the developer sees only the Code Conjurer status view (lower left-hand side). The Reuse Recommendations window on the bottom right only appears when the user requests the positively tested components. Code Conjurer becomes active as soon as the observed test case is (partially) finished, typically when the developer triggers the first "make sure the test case fails" execution. At this point, the tool can send a test-driven search request to the Merobase server, which searches for candidates based on the interface of the class extracted from the test case. Merobase immediately returns information about the number of candidates found to Code Conjurer, which nonintrusively displays this information in the status view. In the meantime, Merobase automatically tests these candidates in a secured virtual machine to filter out those that don't pass the test.

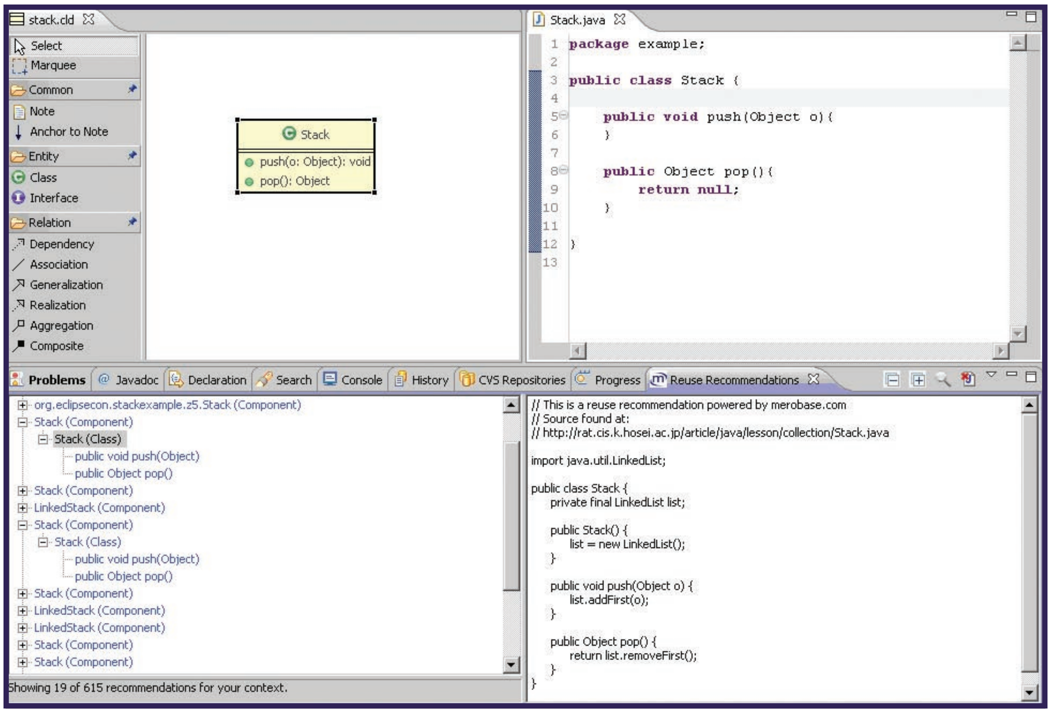
The status view shows the developer immediately whether Code Conjurer has found a matching component suitable for reuse by displaying the number of successfully tested candidates. This feature is especially valuable in an agile context, where test cases are typically developed incrementally. Because Code Conjurer can usually indicate within seconds whether any suitable components are available, a developer can immediately add the next test until no more reusable candidates are available or all required tests have been specified. (We discuss this incremental approach to test-driven reuse in more detail elsewhere.[6])

Table 2

Reusable components found with standard JUnit test cases

	Interface-based matching			Automated adaptation engine		
Desired component	**Candidates**	**Matches**	**Time**	**Candidates**	**Matches**	**Time**
Calculator(sub(int,int):int add(int,int):int mult(int,int):int div(int,int):int)	4	1	19 sec.	23,759	22	20 hrs., 24 min.
Stack(push(Object):void pop():Object)	692	150	26 min.	35,634	611	18 hrs., 23 min.
Matrix (Matrix(int, int) get(int,int):double set(int,int, double):void multiply(Matrix): Matrix)	10	2	23 sec.	137	26	5 min., 25 sec.
ShoppingCart(getItemCount():int getBalance():double addItem(Product):void empty():void removeItem(Product):void)	4	4	26 sec.	12	4	47 sec.
Spreadsheet (put(String,String):void get(String):String)	0	0	3 sec.	22,705	4	15 hrs., 13 min.
ComplexNumber (ComplexNumber(double,double) add(ComplexNumber):ComplexNumber getRealPart():double getImagineryPart():double)	1	0	3 sec.	89	32	1 min., 19 sec.
MortgageCalculator(setRate(double):void setPrincipal(double):void setYears(int):void getMonthlyPayment():double)	0	0	4 sec.	4,265	15	3 hrs., 19 min.

At any time, developers can inspect the candidate list, and if they decide to use a component, Code Conjurer can weave it directly into the project by automatically resolving its dependencies. Table 2 lists the interfaces of some example components found using Code Conjurer. It shows the number of positively tested matches, the total number of candidates, and the total time required to perform the search. For example, for the first component, Code Conjurer found and tested four candidates within 19 seconds but only one of them successfully passed all the test cases. Columns 2 through 4 list results for interface-based matching—that is, where we tested only candidates with the names and signatures defined in the test case. However, as the last three results in the table demonstrate, it's sometimes difficult to anticipate the interface of complex components, so this technique returns few if any



```
package example;

public class Stack {

    public void push(Object o){
    }

    public Object pop(){
        return null;
    }

}
```

```
org.eclipsecon.stackexample.z5.Stack (Component)
Stack (Component)
    Stack (Class)
        public void push(Object)
        public Object pop()
Stack (Component)
LinkedStack (Component)
Stack (Component)
    Stack (Class)
        public void push(Object)
        public Object pop()
Stack (Component)
LinkedStack (Component)
LinkedStack (Component)
Stack (Component)
Stack (Component)
Showing 19 of 615 recommendations for your context.
```

```
// This is a reuse recommendation powered by merobase.com
// Source found at:
// http://rat.cis.k.hosei.ac.jp/article/java/lesson/collection/Stack.java

import java.util.LinkedList;

public class Stack {
    private final LinkedList list;

    public Stack() {
        list = new LinkedList();
    }

    public void push(Object o) {
        list.addFirst(o);
    }

    public Object pop() {
        return list.removeFirst();
    }
}
```

Figure 2. Design-based reuse recommendation example. In the image, a developer has identified a desired API component. If in proactive mode, Code Conjurer issues a search request for that component and displays the results in the Reuse Recommendations box.

candidates. To address this limitation, we developed some heuristics (for example, ignoring object and method names) that can increase the set of testing candidates. In addition, with the aid of an automated adaptation engine, which basically works through all feasible method-mapping permutations, Code Conjurer can recommend components with different interfaces from those required as long as they provide the required functionality. The table also presents the results of these experiments (columns 5 through 7).

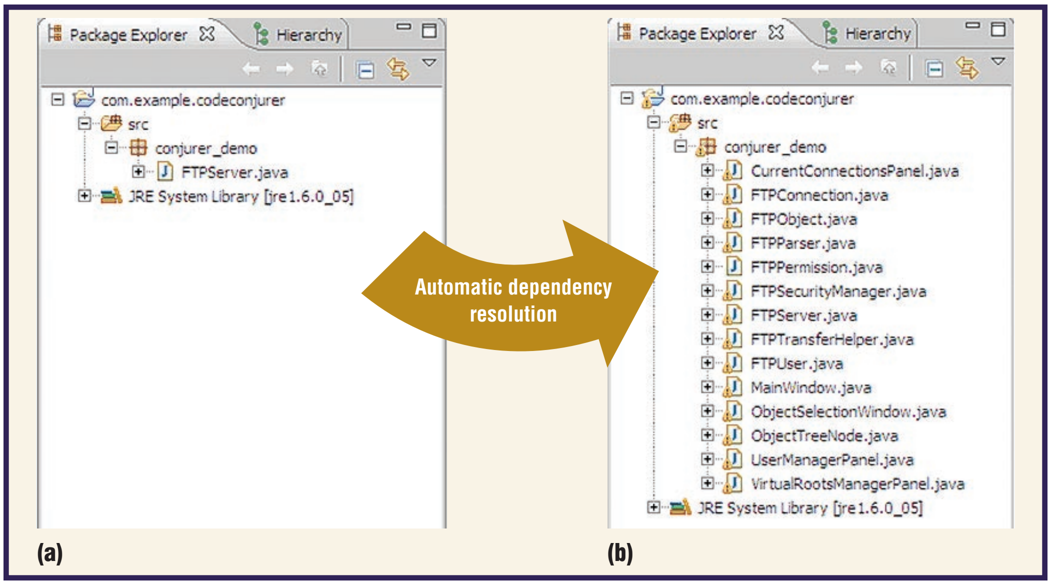
As the table shows, increasing the number of components to test can significantly increase the testing effort. Nevertheless, the results show that the test-driven-reuse approach can deliver useful results (such as the fully functional spreadsheet component shown in the table) and that it makes sense to seamlessly embed the technology into development environments and make it accessible from as many parts of the development process as possible.

Proactive reuse recommendations

Backed up by access to large repositories and better retrieval mechanisms, tools such as Code Conjurer are on the threshold of offering significant support to mainstream software development projects. Beyond the proactive support for test-driven reuse in lightweight development approaches (described in the "Repository-Driven Reuse Assistance Tools" sidebar), Code Conjurer supports design and design-based retrieval in traditional heavyweight processes. Consider a typical development scenario such as the one Figure 2 illustrates, in which developers define the desired component API at the design stage. Code Conjurer can deliver implementation recommendations directly from the component's UML representation and, when set to proactive mode, can issue new search requests each time the developer adds, removes, or changes an interface-defining part of the component. Code Conjurer then presents these components in the lower left Recommendation box. The user can explore any recommendation further by expanding its implementation in the lower right box.

Even if the developer doesn't wish to use one of the components as is, the information embedded in the recommended components can often be useful in improving the design. Code Conjurer not only returns a list of matching components but also analyzes them using various clustering techniques to create a characteristic group picture. Using this information, Code Conjurer can suggest the typical set of methods offered by components matching the partial interface defined by the user. For example, given the stack in Figure 2, Code Conjurer can indicate that the typical set of methods offered by such an abstraction is as follows:

```
public class Stack{
  boolean isEmpty() {}
  Object pop() {}
  void push(Object arg1) {}
  Object top() {}
}
```



```
(a)
com.example.codeconjurer
  src
    conjurer_demo
      FTPServer.java
  JRE System Library [jre1.6.0_05]

(b)
com.example.codeconjurer
  src
    conjurer_demo
      CurrentConnectionsPanel.java
      FTPConnection.java
      FTPObject.java
      FTPParser.java
      FTPPermission.java
      FTPSecurityManager.java
      FTPServer.java
      FTPTransferHelper.java
      FTPUser.java
      MainWindow.java
      ObjectSelectionWindow.java
      ObjectTreeNode.java
      UserManagerPanel.java
      VirtualRootsManagerPanel.java
  JRE System Library [jre1.6.0_05]
```

Figure 3. Eclipse Package Explorer (a) before and (b) after dependency resolution. Code Conjurer's autoresolve feature automatically incorporates the required classes (in this case, an FTPServer component) into a project, starting from the initially retrieved class.

The developer can then easily insert the appropriate operation signatures into the class he or she is working on and, subsequently, obtain reuse recommendations likely to offer the required functionality.

To illustrate Code Conjurer's automated dependency resolution feature, we present an example in a traditional development context requiring an FTPServer component. Because such a component will likely depend on other classes, Code Conjurer's autoresolve feature can automatically incorporate the required classes into the developer's project, starting from the initially retrieved class. Figure 3a shows the Eclipse Package Explorer with the stub of an FTPServer class that the developer has defined. Figure 3b shows the package structure after the developer has chosen an appropriate reuse candidate and Code Conjurer has automatically fetched the missing code on which it depends.

The dependency resolver searches the FTPServer class's immediate context (that is, the same package) for the required dependencies. Obviously, this process can be complex if it can't find the required components at the anticipated places and it must try to recursively find potential matches from other sources.

Open issues

The main advantage of our test-driven approach is the reuse recommendations' high quality. In fact, the recommended components are certain to meet users' needs as defined by their test cases. However, the approach also has some disadvantages.

First, the speed at which it can generate recommendations depends on the number of potential test candidates. We're currently optimizing the selection of candidates that don't fully match the query syntactically to speed up recommendation delivery. Obviously, we can further improve performance by distributing the testing process over a cluster of machines.

In addition, the size of the component pool to which we can apply this technique is reduced because Code Conjurer can't automatically execute all the components in the Merobase repository because of unresolvable dependencies. Another problem is that about 30 percent of all Java source files contain GUI elements, so they will likely require user interaction when executed. Nevertheless, with techniques such as automated dependency resolution, we've already been able to increase the proportion of successfully compilable—and thus potentially executable—components to well over 30 percent.

A third disadvantage is that although precise reuse recommendation technology of the kind supported by Code Conjurer could accelerate software development projects (saving companies a lot of money), having open source code available at one's fingertips might encourage a copy-and-paste mentality among developers. The negative aspects of such programming include a lack of information about code origins and versions (such as bug fixes) and reduced oversight of conformance to licensing requirements.[7] Nevertheless, copy and paste is such a widely practiced reuse technique, even without the availability of code-search engines, that it makes much more sense to try to improve how it's done than to try to stamp it out.

Thus, Code Conjurer and Merobase can help in three main ways. First, by making it easier to find complete, encapsulated code that fulfills a given need, Code Conjurer can encourage a more component-oriented approach (based on complete,

About the Authors

Oliver Hummel is a consultant for Perot Systems Germany. His research interests include software reuse, information retrieval, and software development processes. Hummel received his PhD in software engineering from the University of Mannheim. Contact him at oliver.hummel@ps.net.

Werner Janjic is a PhD student in software engineering at the University of Mannheim. His main research interests are in practical software reuse in the context of agile development and its impact on the software development life cycle. Janjic has a diploma in computer science and business administration from the University of Mannheim. Contact him at janjic@informatik.uni-mannheim.de.

Colin Atkinson is chair of software engineering at the University of Mannheim. His research interests focus on object and component technology and their use in the systematic development of software systems. Atkinson received his PhD in computer science from Imperial College, London. Contact him at atkinson@informatik.uni-mannheim.de.

packaged modules with well-defined APIs) rather than snippet-oriented reuse, based on the scavenging of arbitrary blocks of implementation.

Second, by providing a single, globally accessible index of code resources, Merobase provides a single point of reference that developers can use to identify the origins of reused code (whether snippets or components). Using various duplication-resolution techniques, Merobase can identify different copies of the same code modules by their unique hash value. Although copy-and-paste reusers won't be automatically informed (in a push sense) when developers have changed the source code, they can use the Merobase search facilities to check for changes (in a *pull* sense)—for example, when they're about to release a new version of their application.

Finally, by automatically identifying and analyzing the license information embedded within code modules, search engines can reduce the risk of license mismatches. For example, Merobase includes a license-recognition tool that lets users exclude groups of licenses (such as those with a strong copyleft) from search results. It also uses a license ontology to detect potential incompatibilities between the conditions associated with reuse candidates and a user's existing code base.

Our test-driven recommendation technology effectively trades quantity for quality. It returns fewer query results than previous reuse recommendation tools, but the results are of much higher quality—they do what the developer expects them to do. We believe this is a price worth paying for a tool running in proactive (that is, background) mode that doesn't disturb users unless it finds something that's worthy of their attention. On the other hand, there's the danger that Code Conjurer delivers useful results so infrequently that users will find the service of little value. We don't yet have enough empirical data on the tool's usage to determine whether Code Conjurer has attained the optimal balance. However, we're refining the approach for different kinds of users in different development contexts, with a special focus on agile approaches in which test cases for components are developed incrementally. Even if pure test-driven reuse recommendation doesn't provide the optimal trade-off between quantity and quality by itself, it's just one of the search techniques offered by Code Conjurer and Merobase that can help leverage the large collections of source code accumulated on the Internet and by almost every development company.

Although tools like Code Conjurer are conceptually simple and, when working in fully proactive mode, can be almost invisible to the user, they have the capability to significantly accelerate software engineering projects and to finally usher in a new era in which component-style software reuse is the rule rather than the exception.

References

1. D. McIlroy, "Mass-Produced Software Components," *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 1969, pp. 138–155.
2. A. Mili, R. Mili, and R. Mittermeir, "A Survey of Software Reuse Libraries," *Annals of Software Eng.*, vol. 5, 1998, pp. 349–414.
3. O. Hummel and C. Atkinson, "Extreme Harvesting: Test Driven Discovery and Reuse of Software Components," *Proc. IEEE Int'l Conf. Information Reuse and Integration*, IEEE Press, 2004, pp. 66–72.
4. O.A.L. Lemos, S. Bajracharya, and J. Ossher, "CodeGenie: A Tool for Test-Driven Source Code Search," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications*, ACM Press, 2007, pp. 917–918.
5. O. Hummel, W. Janjic, and C. Atkinson, "Evaluating the Efficiency of Retrieval Methods for Component Repositories," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.*, IEEE Press, 2007, pp. 570–575.
6. O. Hummel and C. Atkinson, "Supporting Agile Reuse through Extreme Harvesting," *Proc. Int'l Conf. Agile Processes in Software Eng. and Extreme Programming*, Springer, 2007, pp. 28–37.
7. W.J. Brown, R.C. Malveau, and H. McCormick, *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.