Integrating S^6 Code Search and Code Bubbles

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

***Abstract*—We wanted to provide a tool for doing code search over open source repositories as part of the Code Bubbles integrated development environment. Integrating code search as a plug-in to Code Bubbles required substantial changes to the S^6 code search engine and the development of appropriate user interfaces in Code Bubbles.**

After briefly reviewing Code Bubbles and the S^6 search engine, this paper describes the integration strategy, the front end for code search, the modifications to the code search engine to handle context-based search, and the user interface for handling the results of the search.

***Index Terms*—Code search, integrated development environments, test-based search, code search in context.**

I. Introduction

Code Bubbles [3,4] is an attempt to redesign the user interface to programming, making the programming environment conform to the programmer's working model. It does this by displaying and manipulating complete *working sets*, collections of task-relevant fragments including code, documentation, test cases, notes, bug reports, and other aspects of programming [9,12]. The fragments in a working set may be contained in multiple files, classes, or other modules, so quick and easy viewing of all these at once is complicated in traditional IDEs. Code Bubbles presents fragments in fully manipulatable interface elements in order to provide an intuitive arrangement of working sets.

Code Bubbles runs as a separate tool on top of Eclipse using a message-based plug-in mechanism [16]. It includes a small Eclipse plug-in which connects to a message bus that the main environment talks to. Integration is achieved using command messages from Code Bubbles to Eclipse and informational messages from Eclipse to Code Bubbles. In addition, Code Bubbles itself provides a plug-in environment that supports a wide range of tools. Plug-ins to Code Bubbles can either be integrated using a traditional toaster-style plug-in model [17] or using the message bus to communicate with the tool running as a separate process.

S^6 is a code search engine that attempts to address several of the problems with current code search technology by effectively automating the multiple tasks the programmer has to do manually in order to use the output of a code search tool [14,15].

S^6 can be used to search for either Java classes or methods. It provides a web-based interface that asks the user to first provide a description of what is wanted in terms of keywords and the semantics of the target code. The latter includes the signature for the target class or method, one or more test cases, and optionally contracts (preconditions and postconditions) and security specifications (e.g. the returned code should not do any file I/O).

Once this data is entered, S^6 processes the request. It first uses the keywords with an existing code search engine (Ohloh, Krugle, or Sourcerer [1]) to get a starting set. It generally takes the first 100-200 files from the search results to build an initial set of solutions. The next step is to apply transformations to each solution to generate new solutions. This is done repeatedly until no more transformations are applicable and no new solutions are generated. These transformations include relatively simply ones such as change the name of the method to match the name in the specified signature or reordering the parameters; moderately complex ones such as replacing a parameter with an assignment or discarding statements that include undefined variables; and complex ones such as extracting functionality from a method by finding a top-level statement computing a value of the return type, doing a backward slice of the code until the only free variables are of the parameter types, and then extracting the resultant code into its own function.

The system next takes all the resultant candidate solutions and does a dependency check. This check first adds other code fragments such as field declarations and auxiliary methods from the initial file that might be needed to make the candidate compile. Then it removes candidate solutions with unmet dependencies that will obviously not compile. For each candidate that passes the dependency check, the system generates a test program that tests that candidate against the user's original test cases, contracts, and security constraints. This test program is compiled and run using Apache Ant [18] and JUnit [5]. The system does an additional pass looking at the output from running the tests, and will try additional transformations if they seem appropriate for example, transformations that handle off-by-one or uppercase/lowercase errors.

Finally, the system takes the candidate solutions that pass all the test cases and passes the resultant code back to the user. It gives the user the option of different formatting styles [13] and different orderings for the results (e.g. fastest to slowest, smallest to largest, least to most complex). It also provides license information for each of the fragments. The user can then take the result, cut and paste it into their program and use it with the confidence it actually compiles and passes their test cases.

One of the enhancements we wanted to make to Code Bubbles was to effectively plug S^6 into the development environment in order to let the programmer search for code rather than writing it from scratch. Here we concentrated on searching for Java methods since this portion of S^6 is more stable and usable.

II. Plugging S6 into Code Bubbles

Plugging S6 into Code Bubbles is neither easy or straightforward. There are several problems or constraints that had to be addressed in order to accomplish the task. These problems are somewhat typical of what is arises when attempting to integrate two complex systems.

The first problem is that S6 is large and memory-intensive. It involves a large code base. Moreover, it typically requires 24-48 gigabytes of memory, and significant CPU resources in order to find a successful match. It is typically run on a large server (16 cores, 64G memory). Running it as part of Code Bubbles or even on the same machine as the one running a programming environment is not practical.

Second, S6 was designed to provide a web-based interface rather than being program callable. While it can be run standalone on a single example, it is much more efficient to run it as a server. The web interface actually communicates with the server using an XmlHttpRequest interface. This provides a hook that lets Code Bubbles use S6 without having to actually run the search tool as part of the environment.

Third, S6 has specific input requirements including keywords, a method or class signature, and test cases. If the programmer wanted to provide these separately each time, they could just use the web interface. The advantage of integrating S6 into Code Bubbles is that much of this information can be derivable automatically assuming the programmer starts by creating a stub method for what they are searching for. In this case the signature is already present, any comments can be used to extract keywords, and, assuming the programmer is using an agile methodology, any existing test cases for that method can be used. At the same time, the programmer might want to provide additional test cases or keywords.

Fourth, S6 generates multiple solutions. Here the environment has to provide a means for selecting among the solutions and then integrating them into their existing code. Each solution can be a single method, in which case it could replace the programmer's stub method. However, the solution can also include auxiliary methods and fields which would also have to be integrated appropriately into the user's code. While S6 provides formatting services, when using a programming environment it makes more sense to use the current style conventions provided by the environment to reformat the returned solution.

The most complex issue in doing the integration is providing a context for the search. Typically S6 looks for code that can run by itself, i.e. it is not dependent on other user classes or methods. This severely limits the types of code that the search engine can be used to find. During normal programming it will often be the case that the target code will need to access other fields and methods of the class it is embedded in. It might also need to access other project or library classes either as parameter or return data types or as intermediate values. In order to make code search useful in these situations we had to create a search context, pass it from Code Bubbles to S6, and then have S6 actually make use of the context both in code transformations and in testing.

The notion of using a context to enable a server to be run as a plug-in is used in other systems, for example, Strathcona [7]. Here the server is run once for a particular instance of the system and communicates with Eclipse directly to build a context from the user's project. Requests are passed to the server which then evaluates them with respect to the previously loaded context. The situation with S6 is different since the S6 server must be able to handle multiple requests coming from different contexts simultaneously. This requires that the client environment build the context on demand and pass the result to the server as part of the request.

There are several other integrations of code search tools for open source code into programming environments. For example, Ohloh provides an eclipse plug-in that lets the programmer start a keyword search from within the environment, effectively duplicating their web interface. This plug-in uses the web XmlHttpRequest interface, similar to what we are doing with S6. CodeGenie uses test cases and signatures to search using Sourcerer for matching code [10,11]. This does not take into account either context or any changes that may need to be made in the returned code. Assieme concentrates on uses of APIs, uses general search engines, and does some corrective work to make the result work [6]. A number of other tools do code search over the current project or an internal code base. Examples here include Codifier for Visual Studio [2] and Code Conjurer for Eclipse [8]. Code Conjurer uses Eclipse quick fix capabilities to modify the code to make it work in the new context.

In the next sections we look at how we addressed these issues while plugging S6 into Code Bubbles. In Section III we look at the user interface for initiating a search. Section IV discusses context-based search. Section V shows the interface for reporting and using the result. We conclude by discussing the open problems with this approach.

III. The Search Request Interface

To use the code search plug-in, programmers first create a stub for the method they want to search for. This stub can include header comments and can be accompanied by JUnit test cases. The programmer then right clicks on the method name and selects the menu option "Code Search for Method Implementation".

The result of this is a code search dialog similar to the S6 web interface but with fields already filled in. An example of this is shown in the top of Figure 1. The keywords are extracted from any comments after discarding common terms. The user is given the choice of entering input-output test cases (shown here), of entering code for a temporary test case, or of selecting from existing test cases for the given routine (shown at the bottom of Figure 1). Once the user is satisfied with the search request, they push the "Start Search" button to initiate the search. This creates an XmlHttpRequest that is sent to the S6 web interface.

IV. Context-Based Search

In order to search for code that makes use of and fits into the user's existing code base, S6 was extended to provide a context-based search. Here, in addition to the search request, the client passes a jar file containing the context in which the search should be done. This file includes:

- Class files for all project code that might be needed to compile and test the resultant fragment, including any user test cases.

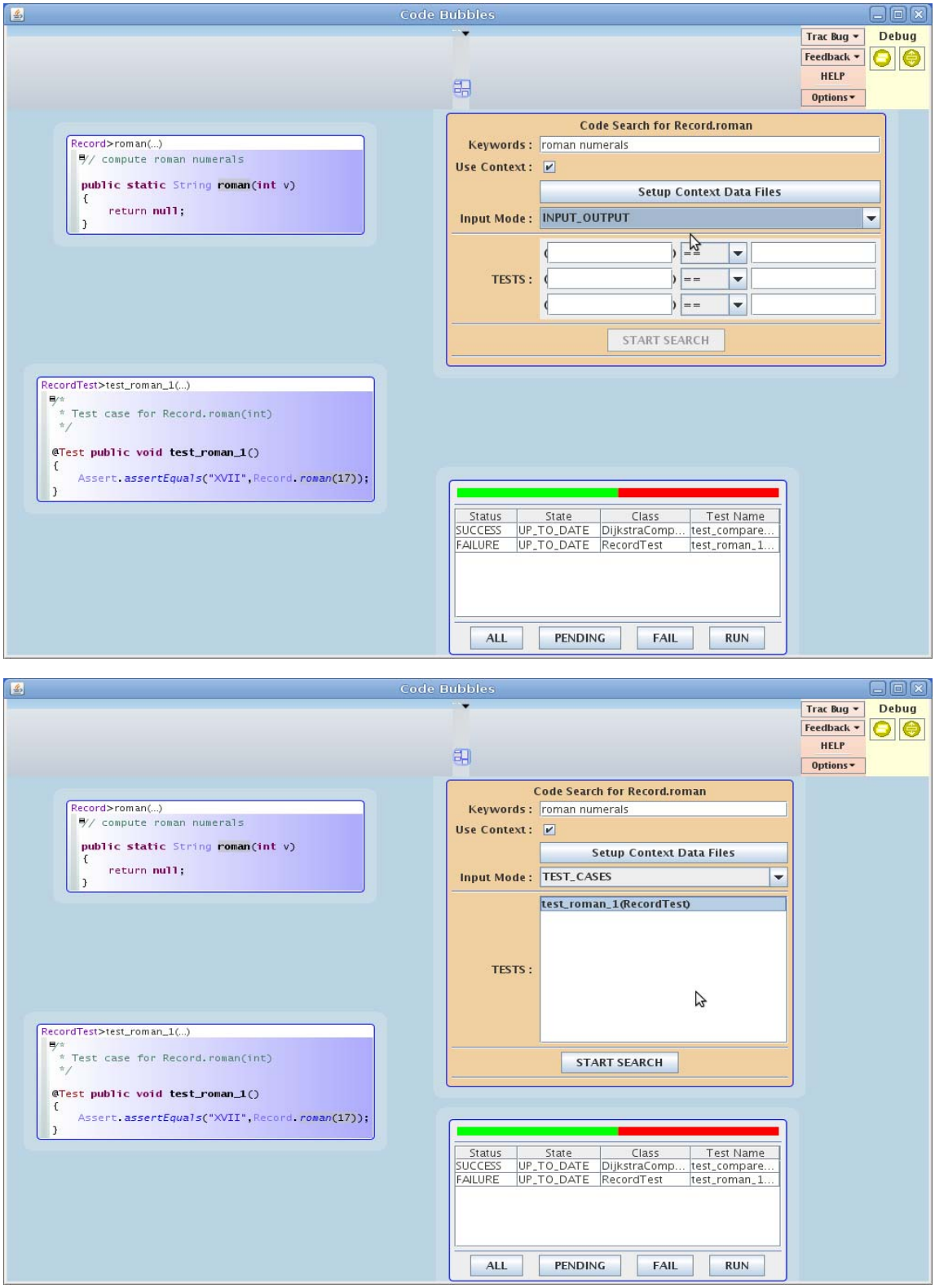

Fig. 1. The user interface for Code Bubbles Code search. In the top figure, The window at the upper left is a stub of the routine to search for. The windov the lower right is the junit test case for this method. The window at the lower left shows test results including the failure of the stub. The dialog on the ri is the search box that results from right clicking on the method and selecting the "Code Search for Method Implementation" option. The bottom figure shows the same situation with the system presenting the existing test cases for use in the search.

- Any library jar files that might be needed to compile and test the resultant fragment.
- The source file the resultant code is to be placed in when searching for a method.

- The name of the class and package where the resultant code should appear.
- The set of imports that should be used for testing.
- Any user data files that are needed for the test programs including the file contents and the path name as accessed by the code.

Context-based search in S^6 is designed to work both with the web interface and with Code Bubbles integration. For the web interface we created an external application using JNLP where the user interactively provided the information. While this works, it is somewhat clumsy to use. The advantage of embedding code search into the programming environment starting with a stub method is that the environment knows the appropriate context and can automatically generate it. The target class and package is specified by the location of the dummy method. The complete user code and libraries are derived from the code and libraries needed for the user's project. While this might include more code and libraries than is required, it is guaranteed to be complete. Extra data files needed for the test case can be included using the appropriate button in the search dialog.

To support transforming the code from its original context in open source code to the user's given context, we created additional transformations inside S^6. These include:

- A context transformation that replaces types in the solution code found by search with types from the passed-in context. For each reference to a class in the solution that refers to a class in the solution package, the transformation attempts to find a corresponding user class. It first constructs the set of fields and methods that are used in the solution code for type to be replaced. Then it considers all classes in the user's context that have methods and fields with similar signatures. For each class in the user's context and each potential mapping of fields and methods it constructs a new solution that uses the user class with the given mapping. This transformation will also handle the 'this' argument to a non-static method, mapping calls to other methods in the original class to possible methods in the user's context class.
- A generalization transformation that will generalize class return and parameter values in the search result to standard Java classes accessible from the user's context. For example, this will replace a return type that is specific to the solution routine with a generic type such as java.lang.Object.
- A name transformation that handles name conflicts between the original code and the user's context by replacing conflicting names in the solution code with new, unique names.
- Transformations that build a new solution from an existing one by removing code that references undefined classes or methods as long as that code is not part of a return statement or is otherwise critical to the method.

The context is also used for testing. Here the solution needs to be integrated into the user's class and run the appropriate class and library binaries from the context. Any JUnit tests from the context cited in the specification are also called as part of the test.

Using these extensions we were able to successfully perform context-based searches. For example, we can find a new method for an existing class (e.g. a square root method for an internal Complex class), and can find methods that access existing object (e.g. a topological sort routine that sorts objects in the user's application). In addition we have used the context to provide file-based test cases, for example for testing file conversion routines.

V. Reporting The Results

After generating the search request, the user interface needs to wait for the search engine to compute the results. Since this could take several minutes, it creates a yellow line at the bottom of the search request box indicating that code search is in progress. Once the search is finished, this is replaced with a green line indicating that the search is finished and a new bubbles containing the results is created as shown in the top of Figure 2.

The result is displayed using a bubble stack, a feature that Code Bubbles typically uses to show the results of a text or identifier search. Each element of the stack lists information about one returned result, indicating the result's source, code size, and a list of the declarations that were returned. The user can click on any of these entries and view the actual code within the bubble stack as seen in the figure. The code fragments can be dragged out into individual bubbles so that the programmer can compare results to choose the preferred one.

Right clicking on an expanded solution, either within or outside the result bubble stack, gives the programmer two options, either to view the license for the resultant code or to use the code to replace the dummy method. Choosing replace automatically replaces the stub code with the returned code the programmer selected, adds any auxiliary definitions such as needed fields or helper methods to the user's class, and formats all the additions. The result can be seen in the bottom of Figure 2. Note that here the test cases now all pass.

VI. Discussion

Our previous work Code Bubbles used two different plug-in mechanisms, a traditional toaster-based plug-in with callbacks and access to the user interface, and a message based plug-in for communicating with Eclipse and with external tools for version management, testing, code analysis, and debugging. However, neither of these technologies was sufficient for integrating code search. Instead, we had to access code search as a web service. This forced us to define a context for code search and to integrate that notion both into Code Bubbles and S^6.

In this work we demonstrated that it is possible to integrate a sophisticated code search tool into a programming environment so that the result seems relatively seamless to the user. We have been using the integrated tool in place of the standard web interface for those cases where we wanted to do code search, but still don't have enough experience to fully evaluate the integrated tool.

There is still a considerable amount of work that is needed to make this integration practical and complete. It first needs to be extended to include searching for classes as well

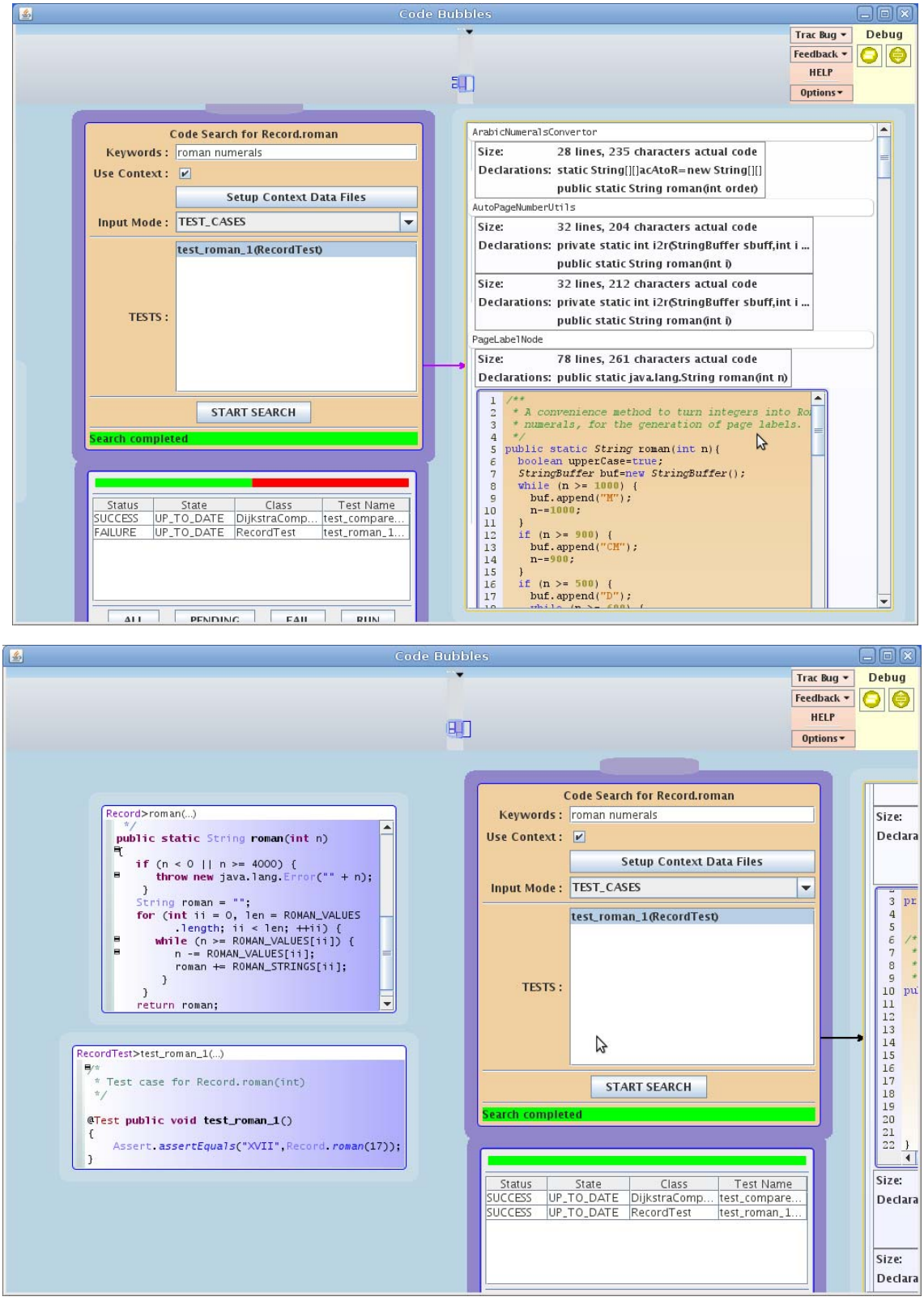

Fig. 2. The top view shows the result of a code search, with bubble stack on the right containing the various results that were returned. Here the user has selected on for further analysis. The bottom view shows the original code after choosing one of the results and using it to replace the original method.

as methods. This is more difficult both because class stubs and class test cases are more complex. It would also require changes to the context transformations within S^6 since these have been designed and tested for methods and would be

inefficient in their current form if used for class transformations.

A second concern is that the underlying code search engines used by S^6 are very sensitive to the proper choice of keywords for the initial search. While taking keywords from comments is a convenience, a better job needs to be done on selecting appropriate words or phrases. Moreover, S^6 itself needs to deal with this situation.

A third change that is needed is to determine a minimal context, possibly splitting the context used for search from that required for compiling and running the test cases. This would make context-based search faster since it would restrict the set of transformations that have to be considered. The problem here is that what is minimal can be difficult to define. We could just include those parts of the system that are needed semantically by the enclosing class and any test classes. (This is relatively easy to compute assuming that Java reflection is not used.) However, this might be either too large (you might not need all of the libraries or it other tests in the test class and what they refer to), or it might be too small since a successful transformation might require access to code that is not currently used by the class.

Acknowledgements

This work is supported by the National Science Foundation grant CCF1130822. Additional support has come from Microsoft and Google.

References

1. Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2006*, pp. 682-682 (October 2006).
2. Andrew Begel, "Codifier: a programmer-centric search user interface," *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).
3. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *International Conference on Software Engineering 2010*, pp. 455-464 (2010).
4. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, pp. 2503-2512 (2010).
5. E. Gamma and K. Beck, "Test infected: Programmers love writing tests," *http://www.junit.org*, (1998).
6. Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proceedings UIST 2007*, pp. 13-22 (October 2007).
7. Reid Holmes and Gail C. Murphy, "Using structural context to recommend source code examples," *International Conference on Software Engineering 05*, pp. 117-125 (May 2005).
8. Caitlin Kelleher and Randy Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys* Vol. **37**(2) pp. 83-137 (June 2005).
9. Andrew J. Ko, Htet Aung, and Brad A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," *Proceedings of the 27th International Conference on Software Engineering*, pp. 126-135 (2005).
10. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Ricardo Morla, Paulo Masiero, Pierre Baldi, and Cristina Lopes, "CodeGenie: using test-cases to search and reuse source code," *ASE '07*, pp. 525-526 (November 2007).
11. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Paulo Masiero, and Cristina Lopes, "Applying test-driven code search to the reuse of auxiliary functionality," *Proceedings ACM Symposium on Applied Computing*, pp. 476-482 (2009).
12. B. A. Meyers, A. J. Ko, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering* Vol. **32**(12) pp. 971-987 (2006).
13. Steven P. Reiss, "Automatic code stylizing," *Proceedings ASE '07*, pp. 74-83 (November 2007).
14. Steven P. Reiss, "Semantics-based code search," *International Conference on Software Engineering 2009*, pp. 243-253 (May 2009).
15. Steven P. Reiss, "Specifying what to search for," *Proceedings SUITE 2009*, (May 2009).
16. Steven P. Reiss, "Plugging in and into Code Bubbles," *Proceedings Workshop on Developing Tools as Plug-ins 2012*, pp. 55-60 (June 2012).
17. Richard Snodgrass and Karen Shannon, "Supporting flexible and efficient tool integration," *Proceedings International Workshop on Advanced Programming Environments*, pp. 290-313 (June 1985).
18. Jesse Tilly and Eric M. Burke, *Ant*: *The Definitive Guide*, O'Reilly (2002).