PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, USA
xie@csc.ncsu.edu

ABSTRACT

Programmers commonly reuse existing frameworks or libraries to reduce software development efforts. One common problem in reusing the existing frameworks or libraries is that the programmers know what type of object that they need, but do not know how to get that object with a specific method sequence. To help programmers to address this issue, we have developed an approach that takes queries of the form "Source object type → Destination object type" as input, and suggests relevant method-invocation sequences that can serve as solutions that yield the destination object from the source object given in the query. Our approach interacts with a code search engine (CSE) to gather relevant code samples and performs static analysis over the gathered samples to extract required sequences. As code samples are collected on demand through CSE, our approach is not limited to queries of any specific set of frameworks or libraries. We have implemented our approach with a tool called PARSEWeb, and conducted four different evaluations to show that our approach is effective in addressing programmers' queries. We also show that PARSEWeb performs better than existing related tools: Prospector and Strathcona.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques—*Object-oriented programming*; D.2.6 [Software Engineering]: Programming Environments—*Integrated environments*;

General Terms: Languages, Experimentation.

Keywords: Code reuse, Code search engine, Code examples, Ranking code samples

1. INTRODUCTION

The primary goal of software development is to deliver high-quality software efficiently and in the least amount of time whenever possible. To achieve the preceding goal, programmers often want to reuse existing frameworks or libraries instead of developing similar code artifacts from scratch. The challenging aspect for programmers in reusing the existing frameworks or libraries is to understand the usage of Application Programming Interfaces (APIs) exposed by those frameworks or libraries, because many of the existing frameworks or libraries are not well-documented. Even when such documentations exist, they are often outdated [9].

In general, the reuse of existing frameworks or libraries involve instantiation of several object types of those frameworks or libraries. For example, consider the programming task of parsing code in a dirty editor (editor whose content is not yet saved) of the Eclipse IDE framework. As a dirty editor is represented as an object of the `IEditorPart` type and the programmer needs an object of `ICompilationUnit` for parsing, the programmer has to identify a method sequence that takes the `IEditorPart` object as input and results in an object of `ICompilationUnit`. One such possible method sequence is shown below:

```
IEditorPart iep = ...
IEditorInput editorInp = iep.getEditorInput();
IWorkingCopyManager wcm = JavaUI.getWorkingCopyManager();
ICompilationUnit icu = wcm.getWorkingCopy(editorInp);
```

The code sample shown above exhibits the difficulties faced by programmers in reusing the existing frameworks or libraries. A programmer unfamiliar to Eclipse may take long time to identify that an `IWorkingCopyManager` object is needed for getting the `ICompilationUnit` object from an object of the `IEditorInput` type. Furthermore, it is not trivial to find an appropriate way of instantiating the `IWorkingCopyManager` object as the instantiation requires a static method invocation on the `JavaUI` class.

In many such situations, programmers know what type of object that they need to instantiate (like `ICompilationUnit`), but do not know how to write code to get that object from a known object type (like `IEditorPart`). For simplicity, we refer the known object type as *Source* and the required object type as *Destination*. Therefore, the proposed problem can be translated to a query of the form "*Source* → *Destination*". There are several existing approaches [6, 11, 14] that address the described problem. But the common issue faced by these existing approaches is that the scope of these approaches is limited to the information available in a fixed (often small) set of applications reusing the frameworks or libraries of interest. Many code search engines (CSE) such as Google [5] and Koders [8] are available on the web. These CSEs can be used to assist programmers by providing relevant code examples with usages of the given query from a large number of publicly accessible source code repositories. For the preceding example, programmers can issue the query "`IEditorPart ICompilationUnit`" to gather relevant

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

```
01:FileName:0_UserBean.java MethodName:ingest  Rank:1 NumberOfOccurrences:6
02:QueueConnectionFactory,createQueueConnection() ReturnType:QueueConnection
03:QueueConnection,createQueueSession(boolean,Session.AUTO_ACKNOWLEDGE) ReturnType:QueueSession
04:QueueSession,createSender(Queue) ReturnType:QueueSender
```

Figure 1: Method sequence suggested by PARSEWeb.

```
01:QueueConnectionFactory qcf;
02:QueueConnection queueConn = qcf.createQueueConnection();
03:QueueSession qs = queueConn.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
04:QueueSender queueSender = qs.createSender(new Queue());
```

Figure 2: Equivalent Java code for the method sequence suggested by PARSEWeb.

code samples with usages of the object types `IEditorPart` and `ICompilationUnit`. However, these CSEs are not quite helpful in addressing the described problem because we observed that Google Code Search Engine (GCSE) [5] returns nearly 100 results for this query and the desired method sequence shown above is present in the 25^{th} source file among those results.

Our approach addresses the described problem by accepting queries of the form "*Source* → *Destination*" and suggests frequently used Method-Invocation Sequences (MIS) that can transform an object of the *Source* type to an object of the *Destination* type. Our approach also suggests relevant code samples that are extracted from a large number of publicly accessible source code repositories. These suggested MISs along with the code samples can help programmers in addressing the described problem and thereby help reduce programmers' effort in reusing existing frameworks or libraries.

We have implemented the proposed approach with a tool called PARSEWeb for helping reuse Java code. PARSEWeb interacts with GCSE [5] to search for code samples with the usages of the given *Source* and *Destination* object types, and downloads the code example results to form a local source code repository. PARSEWeb analyzes the local source code repository to extract different MISs and clusters similar MISs using a sequence postprocessor. These extracted MISs can serve as a solution for the given query. PARSEWeb also sorts the final set of MISs using several ranking heuristics. PARSEWeb uses an additional heuristic called query splitting that helps address the problem where code samples for the given query are split among different source files.

This paper makes the following main contributions:

- An approach for reducing programmers' effort while reusing existing frameworks or libraries by providing frequently used MISs and relevant code samples.
- A technique for collecting relevant code samples dynamically from the web. This contribution has an added advantage of not limiting the scope of the approach to any specific set of frameworks or libraries.
- A technique for analyzing partial code samples through Abstract Syntax Trees (AST) and Directed Acyclic Graphs (DAG) that can handle control-flow information and method inlining.
- An Eclipse plugin tool implemented for the proposed approach and several evaluations to assess the effectiveness of the tool.

The rest of the paper is organized as follows. Section 2 explains the approach through an example. Section 3 presents related work. Section 4 describes key aspects of the approach. Section 5 describes implementation details. Section 6 discusses evaluation results. Section 7 discusses limitations and future work. Finally, Section 8 concludes.

2. EXAMPLE

We next use an example to illustrate our approach and how our approach can help in reducing programmers' effort when reusing existing frameworks or libraries. We use object types `QueueConnectionFactory` and `QueueSender` from the OpenJMS library[1], which is an open source implementation of Sun's Java Message Service API 1.1 Specification. Consider that a programmer has an object of type `QueueConnectionFactory` and does not know how to write code to get a `QueueSender` object, which is required for sending messages.

To address the problem, the programmer can use our PARSEWeb tool (implemented for our approach) in the following way. The programmer first translates the problem into a "`QueueConnectionFactory` → `QueueSender`" query. Given this query, PARSEWeb requests GCSE for relevant code samples with usages of the given *Source* and *Destination* object types, and downloads the code samples to form a local source code repository. The downloaded code samples are often partial and not compilable as GCSE retrieves (and subsequently PARSEWeb downloads) only source files with usages of the given object types instead of entire projects. PARSEWeb analyzes each partial code sample using an Abstract Syntax Tree (AST) and builds a Directed Acyclic Graph (DAG) that represents each given code sample in order to capture control-flow information in the code example. PARSEWeb traverses this DAG to extract MISs that take `QueueConnectionFactory` as input and result in an object of `QueueSender`. The output of PARSEWeb for the given query is shown in Figure 1. The sequence starts with the invocation of the `createQueueConnection` method that results in an instance of `QueueConnection` type. Similarly, by following other method invocations, the method sequence finally results in the `QueueSender` object, which is the desired destination object of the given query.

The sample output also shows additional details such as `FileName`, `MethodName`, `Rank`, and `NumberOfOccurrences`. The details `FileName` and `MethodName` indicate the source file that the programmer can browse to find a relevant code sample for this MIS. For example, a code sample of the given query can be found in method `ingest` of file `0_UserBean.java`. The prefix of the file name gives the index of the source file that contained the suggested solution among the results of GCSE. In this example, the code sample for the suggested method-invocation sequence can be found in the first source file returned by GCSE as the prefix value is zero. Generally, many queries result in more than one possible solution. The `Rank` attribute gives the rank of the corresponding MIS

[1] `http://java.sun.com/products/jms`

among the complete set of results. PARSEWeb derives the rank of a MIS based on the `NumberOfOccurrences` attribute and some other heuristics described in Section 4.4.2. The suggested MIS contains all necessary information for the programmer to write code for getting the *Destination* object from the given *Source* object. The suggested MIS can be transformed to equivalent Java code by introducing required intermediate variables. The code sample suggested along with the MIS can assist programmers in gathering this additional information regarding the intermediate variables. Figure 2 shows equivalent Java code for the suggested MIS.

3. RELATED WORK

The problems faced by programmers in reusing existing frameworks or libraries are addressed by different approaches. Earlier research in this area mainly concentrated on identifying related samples by matching keywords [12] or comments [16]. But solutions suggested based on these approaches often cannot effectively help programmers in reusing the existing code samples.

Mandelin et al. developed Prospector [11], a tool that accepts queries in the form of a pair (T_{in}, T_{out}), where T_{in} and T_{out} are class types, and suggests solutions by traversing all paths among types of API signatures between T_{in} and T_{out}. A solution to the query is a synthesized code sample that takes an input object of type T_{in} and returns an output object of type T_{out}. Their approach uses API signatures for suggesting solutions to the given query. As API signatures are used for addressing the query, Prospector returns many irrelevant examples, as shown in our evaluation. Our approach is different from Prospector as Prospector uses API signatures, whereas our approach uses code samples for solving the given query. This feature helps PARSEWeb in identifying more relevant code samples by giving higher preference to code samples that are often used.

Strathcona developed by Holmes and Murphy [6] maintains an example repository and compares the context of the code under development with the code samples in the example repository, and recommends relevant examples. Both PARSEWeb and Strathcona suggest relevant code samples, but the Strathcona approach is based on heuristics that are generic and are not tuned for addressing the described problem. This limitation often results in irrelevant examples as shown in our evaluation. XSnippet developed by Sahavechaphan and Claypool [14] also tries to address the described problem by suggesting relevant code snippets for the object instantiation task at hand. These suggested code snippets are selected from a sample repository. The major problem with both Strathcona and XSnippet is the availability of limited code samples stored in the repository.

Coogle developed by Sager et al. [13] extends the concept of similarity measures (often used to find similar documents for a given query) to source code repositories. Their approach detects similar Java classes in software projects using tree similarity algorithms. Both PARSEWeb and Coogle use ASTs for parsing Java code, but a structural similarity at the Java class level may not effectively address the described problem. Similar to Strathcona, Coogle may also result in many irrelevant examples.

Another related tool MAPO, developed by Xie and Pei [15], generates frequent usage patterns of an API by extracting and mining code samples from open source repositories through CSEs. Both MAPO and PARSEWeb exploit CSEs for gathering relevant code samples, but MAPO cannot solve queries of the form "*Source* → *Destination*". Programmers need to know the API to be used for using MAPO to identify usage patterns of that API. Moreover, our approach is more effective than MAPO as we consider control-flow information while generating MISs for the given query, whereas MAPO does not consider the control-flow information.

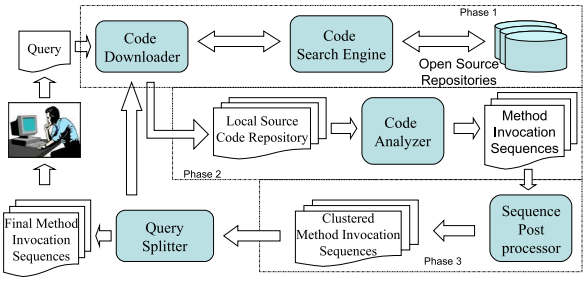

Figure 3: Overview of PARSEWeb

4. APPROACH

Our approach consists of five major components: code search engine, code downloader, code analyzer, sequence postprocessor, and query splitter. Figure 3 shows an overview of all components. Our approach consists of three main phases. These phases may be iterated more than once. In Phase 1, the code downloader accepts a query from the programmer and forms a local source code repository with the code samples collected through CSE. In Phase 2, the code analyzer analyzes the code samples stored in the repository and generates MISs. In Phase 3, the sequence postprocessor clusters similar MISs and ranks the clustered MISs. If the result of the sequence postprocessor consists of any MISs that can serve as a solution for the given query, the query splitter simply outputs the result. If there are no solution MISs in the result generated by the sequence postprocessor, the query splitter instead splits the given query into different sub-queries and iterates all the preceding three phases for each sub-query. Finally, the query splitter gathers results of all sub-queries and generates the final output. We next present the details of each component.

4.1 Code Search Engine

On the web, there are a variety of CSEs[2] available to assist programmers in searching for relevant code samples like Google [5] and Koders [8]. The main reason for using CSEs in our approach is that many new frameworks or libraries emerge from day to day and it is not practical for our own approach to maintain a repository of all the available frameworks or libraries, and extract results of given queries from that repository. Therefore, our approach uses CSEs as an alternative to search in the available open source frameworks or libraries on the web and gathers the relevant code samples on demand. Moreover, our idea of gathering code samples on demand makes our approach independent of any specific set of frameworks or libraries.

In our approach, we used GCSE [5] for collecting relevant code samples for the given query, partly because GCSE provides convenient open APIs for the third-party tools to interact with and it has been consistently improved and main-

[2] `http://gonzui.sourceforge.net/links.html`

tained. However, our approach is independent of the underlying CSE and can be extended easily to any other CSE.

4.2 Code Downloader

The code downloader component accepts queries of the form "*Source* → *Destination*" from the programmer and constructs a request for a CSE. The constructed request contains both *Source* and *Destination* object types. The code downloader component submits the constructed request to CSE and downloads code samples returned by CSE to form a local source code repository. The code samples stored in the local source code repository are often partial and not compilable, because the code downloader downloads only individual source files with usages of the given *Source* and *Destination* object types, instead of entire projects.

4.3 Code Analyzer

The code analyzer component takes code samples stored in the local source code repository as input and analyzes them to extract MISs that can serve as solutions for the given query of the form "*Source* → *Destination*".

As code samples (i.e., source files) stored in the local source code repository are often partial and not compilable, our approach converts each code sample into an intermediate form. We developed the following graph-based in the conversion process to support method inlining and to preserve control-flow information in the intermediate form.

Initially, the code analyzer creates an Abstract Syntax Tree (AST) for each code sample. The code analyzer uses the created AST to build a Directed Acyclic Graph (DAG). A node in the constructed DAG contains a single statement and an edge represents the control flow between the two statements. The primary advantage of DAG is that DAG supports control-flow information through branches and joins, and provides an effective mechanism for identifying paths between any two nodes in the graph. The statements inside loops like *while* and *for* may be executed either several times or zero time. Considering these statements once can help generate the MIS associated with the loop. Therefore, our approach treats the statements inside loops like *while* and *for* as a group of statements that are executed either once or not. While constructing DAG, the code analyzer performs method inlining by replacing the method invocations of the class under analysis with the body of the corresponding method declarations. Our approach cannot perform method inlining if the corresponding method declaration is *abstract*. This method inlining process helps to identify MISs that are split among methods of the class under analysis (shown in Section 6.4). If the current class does not contain the method declaration of a method invocation whose receiver type is `this` (either explicitly or implicitly stated at the call site), the code analyzer assumes that this method belongs to the parent class and associates a special context called parent context (Section 4.3.2) with that node. When needed, the code analyzer can use this parent context for identifying the *Source* object type.

The nodes in the constructed DAG contain only those statements that result in a transformation from one object type to another. In particular, the statements that are included in the intermediate form belong to one of the types described below:

- Method Invocation: Generally, a method invocation with a non-void return type can be considered as a statement that transforms either the receiver type or argument types to the return type. For example, the method invocation `ReturnObj obj1 = RefObj.method(Arg1, Arg2)` can be considered as a statement that transforms objects of type `RefObj`, `Arg1` or `Arg2` to `ReturnObj`.
- Constructor: As a constructor generally takes some arguments, it can be considered as a statement that transforms objects of its argument types to the newly created object type.
- Typecast: A typecast can be considered as a transformation statement, because it performs an explicit transformation from one object type to another.

Along with identifying the preceding statement types, our approach uses several heuristics (Section 4.3.1) to gather additional type information for each statement and this additional type information is associated with the corresponding node in the graph. For example, the additional type information for method-invocation statements include the receiver object type, the return object type, and argument types. When any of receiver, return, or argument types matches with the given *Source* object type, the code analyzer marks the corresponding node as a *Source* node. When the return type of any statement matches with the required *Destination* type, the code analyzer marks the corresponding node as a *Destination* node.

The code analyzer extracts a MIS from the DAG by calculating the shortest path from a *Source* node to a *Destination* node. The shortest path is sufficient as every path from *Source* to *Destination* nodes contain a desired method-invocation sequence. Once a possible sequence is identified from the DAG, the minimization process of the code analyzer extracts a minimal MIS from the possible sequence by eliminating the extra method invocations that are not related to the given query. This minimal MIS is identified by traversing the sequence in the reverse direction from the *Destination* node to the *Source* node by continuously matching the receiver type and argument types of each statement with the return type of the preceding statements. For example, consider a possible sequence for the query "`IEditorPart` → `ICompilationUnit`" (where each statement consists of the receiver type, method name, arguments, and return type):

```
01:IEditorPart,getEditorInput() : IEditorInput
02:CONSTRUCTOR,Shell() : Shell
03:Shell,setText(String) : void
04:JavaUI,getWorkingCopyManager() : IWorkingCopyManager
05:IWorkingCopyManager,connect(IEditorInput) : void
06:IWorkingCopyManager,getWorkingCopy(IEditorInput)
        : ICompilationUnit
```

The minimization process maintains a special set called a look-up set that initially contains only the required *Destination* object. For the given possible sequence, the process starts from Statement 6 and adds the receiver type `IWorkingCopyManager` and the argument type `IEditorInput` to the look-up set, and removes `ICompilationUnit` from the look-up set. The minimization process retains Statement 5 in the minimal MIS as its receiver type matches with one of the types in the look-up set. In Statement 4, the minimization process adds `JavaUI` to the look-up set and removes `IWorkingCopyManager`. The process ignores Statements 3 and 2 as none of its object types match with the object types in the look-up set. The minimization process ends with Statement 1 and generates the minimal MIS as "1,4,5,6". These

minimal MISs are given as input to the sequence postprocessor component (Section 4.4).

4.3.1 Type Resolution

Heuristics play a major role in the static analysis phase of our approach. As the downloaded code samples are often partial and not compilable, our approach relies on these heuristics to gather information like the receiver object type, argument types, and the return object type of each method invocation. Our approach uses five heuristics for identifying the receiver object type and argument types, and uses ten heuristics for identifying the return object type. Our heuristics are based on simple language semantics like object types of left and right hand expressions of an assignment statement are either same or related to each other through inheritance. Due to space limit, we explain only two of our major heuristics used for identifying the return type of a method invocation.

Type Heuristic 1: *The return type of a method-invocation statement contained in an initialization expression is the same as the type of the declared variable.*

Consider the code sample shown below:

```
QueueConnection connect; QueueSession session =
       connect.createQueueSession(false,int)
```

The receiver type of the method `createQueueSession` is the type of `connect` variable. Therefore, the receiver type can be simply inferred by looking at the declaration of the `connect` variable. But as our approach mainly deals with code that is partial and not compilable, it is difficult to get the return type of the method-invocation `createQueueSession`. The reason is the lack of access to method declarations. However, the return type can be inferred from the type of variable `session` on the left hand side of the assignment statement. As the type of variable `session` is `QueueSession`, we can infer that the return type of the method-invocation `createQueueSession` is `QueueSession`.

Type Heuristic 2: *The return type of an outermost method-invocation contained in a return statement is the same as the return type of the enclosing method declaration.*

Consider code sample presented below:

```
public QueueSession test()    { ...
       return connect.createQueueSession(false,int);}
```

In this code sample, the method-invocation statement `createQueueSession` is a part of the return statement of the method declaration. In this scenario, we can infer the return type of this method-invocation from the return type of the method `test`. As the method `test` returns `QueueSession`, we can infer that the return type of the method-invocation `createQueueSession` is also `QueueSession`.

4.3.2 Parent Context

A parent context includes the parent object type, if any, and interfaces implemented by the class in the given code sample. In some of the code samples, we observed that the *Source* object type is not explicitly available in the code sample but is described as a parent class. The code analyzer component handles this aspect by identifying the parent context as possible *Source* object types. Consider the query "`TextEditorAction` → `ITextSelection`" and a related code sample shown below:

```
01:public class OpenAction extends TextEditorAction {
02:  public void run() {
03:      ITextEditor editor = getTextEditor();
04:      ISelectionProvider provider =
              editor.getSelectionProvider();
05:      ITextSelection textSelection = (ITextSelection)
              provider.getSelection();   }  }
```

In the preceding sample, class `OpenAction` declares only a `run` method. Although the code sample includes a solution for the given query, it is not explicitly available as the method declaration of `getTextEditor` is not available. In this case, we can consider that the method `getTextEditor` is defined in either `TextEditorAction` or its parent classes. Therefore, class `TextEditorAction` can be considered as receiver type for the method `getTextEditor`. With this consideration, our approach can identify that this code sample is a possible solution for the given query.

4.4 Sequence Postprocessor

The sequence postprocessor component clusters similar MISs and ranks the clustered MISs. Clustering of MISs helps to identify distinct possible MISs and also reduces the total number of MISs. This reduction of the number of results can help programmers quickly identify the desired MIS for the given query. To further assist programmers, the sequence postprocessor also sorts the clustered results. These sorted results can help programmers identify sequences that are more frequently used for addressing the given query.

4.4.1 Sequence Clustering

We next describe the heuristic used by our approach for identifying and clustering similar MISs. To identify similar MISs, our approach ignores the order of statements in the extracted MISs. Our approach considers MISs with the same set of statements and with a different order as similar. For example, consider MISs "2,3,4,5" and "2,4,3,5" where each number indicates a single statement associated with the node in the constructed DAG. Our approach considers these sequences as similar because different programmers may write intermediate statements in different orders and these statements may be independent from one another.

To further cluster the identified MISs, our approach identifies MISs with minor differences and clusters those identified MISs. We introduce an attribute, called *cluster precision*, which defines the number of statements by which two given MISs differ each other. This attribute is configurable and helps the sequence postprocessor in further clustering the identified MISs. Our approach considers MISs that differ by the given *cluster precision* value as similar, irrespective of the order of statements in those MISs. For example, consider MISs "8,9,6,7" and "8,6,10,7". These two sequences have three common statements (8,6,7) and differ by a single statement. Our approach considers these two MISs as similar under a *cluster precision value* of one, as both the sequences differ by only one method invocation. This heuristic is based on the observation that different MISs in the final set of sequences often contain overloaded forms of the same method invocation.

4.4.2 Sequence Ranking

In general, many queries result in more than one possible solution, and not all solutions are of the same importance to the programmer. To assist the programmer in quickly identifying the desired MISs, our approach uses two ranking heuristics and sorts the final set of MISs.

Ranking Heuristic 1: *Higher the frequency → Higher the rank*

This heuristic is based on the observation that more-used MISs might be more likely to be used compared to less-used MISs. Therefore, MISs with higher frequencies are given a higher preference.

Ranking Heuristic 2: *Shorter the length → Higher the rank*

This ranking heuristic, which was originally proposed in the Prospector [11] approach, is based on the length of the MIS. Shorter sequences are given a higher preference to longer sequences. This heuristic is considered based on the observation that programmers would often tend to use shorter sequences instead of longer ones to achieve their task.

4.5 Query Splitter

Query splitting is an additional heuristic used by our approach to address the problem of lack of code samples in the results of CSE. We observed that a code sample for some of the queries is split among different files instead of having the entire sample in the same file. The query splitting heuristic helps to address this problem by splitting the given query into multiple sub-queries. The algorithm of our approach including the query splitting heuristic is described in Algorithm 1.

Initially, our approach accepts the query of the form "*Source → Destination*" and tries to suggest solutions. If no possible MISs are found, our approach tries to infer the immediate alternate destinations (*AltDest*) by constructing a new query that includes only the *Destination* object type. A query with just the *Destination* object type provides different possible MISs, referred as *DestOnlyMISs*, that result in the object of the *Destination* type. In these *DestOnlyMISs*, the *Source* can be of any object type. Our approach infers the *AltDestSet* by identifying the receiver type and argument types in the last method invocation (*lastMI*) of each MIS in the *DestOnlyMISs* set. The primitive types, such as `int`, are ignored while identifying the *AltDest*. For each of the *AltDest*, new MISs are generated by constructing queries of the form "*Source → AltDest*". The *lastMI* of the earlier sequence is appended to the new set of sequences to generate a complete MIS. In case our approach including the query splitting is not able to suggest any MISs, we simply return the *DestOnlyMISs*.

```
Input: Source and Destination object types
Output: Method-Invocation Sequences
Extract MISs for the Query "Source→Destination";
if MISs are not empty then
    return MISs;
end
//Query Splitting
Extract DestOnlyMISs for the Query "Destination";
for MIS in DestOnlyMISs do
    lastMI = MIS.lastMethodInvocation();
    AltDestSet = ReceiverType and ArgTypes of lastMI;
    Intialize FinalMISs;
    for AltDest in AltDestSet do
        Extract AltMIS for the Query "Source→AltDest";
        Append lastMI to AltMIS;
        Add AltMIS to FinalMISs;
    end
end
if FinalMISs are not empty then
    return FinalMISs
end
else
    return DestOnlyMISs;
end
```

Algorithm 1: Pseudocode of the PARSEWeb algorithm with the query splitting heuristic

5. IMPLEMENTATION

We used Google Code Search Engine (GCSE) [5] as an underlying CSE for the code downloader component. To improve performance, the code downloader uses the multi-threading feature of the Java programming language, and invokes a post processor written in the Perl language to transform the source files returned by GCSE from HTML to Java. Eclipse JDT Compiler [1] is used for building ASTs from Java files. We used Dijkstra's shortest path algorithm from the Jung [7] library to gather the required path from *Source* to *Destination* object types.

We developed an Eclipse plugin, called PARSEWeb[3], that integrates all described aspects of our approach. PARSEWeb displays the suggested MISs for the given query in a tree-structured tabular form. A snapshot of our PARSEWeb output is shown in Figure 4. Each MIS is associated with additional information like rank, frequency, and length. Programmers can browse the relevant code sample of the suggested MIS by double clicking on the corresponding entry.

The current implementation of PARSEWeb shows only the first ten MISs that can serve as a solution for the given query. Furthermore, the query splitter is configured to iterate all three main phases for only the first five elements in *DestOnlyMISs* (shown in Algorithm 1). However, both these parameters are configurable through the property file.

6. EVALUATION

We conducted four different evaluations on PARSEWeb to show that PARSEWeb is effective in solving programmers' queries. In the first evaluation, we showed that PARSEWeb is able to solve programming problems posted in developer forums of existing libraries. In the second evaluation, we showed that PARSEWeb-suggested solutions are available in a real project. We also analyzed the PARSEWeb results with the results of two other related tools: Prospector[4] [11] and Strathcona[5] [6]. Prospector tries to solve the queries related to a specific set of frameworks or libraries by using API signatures. Strathcona tries to suggest similar code examples stored in an example repository by matching the context of the code under development with the samples stored in the example repository. In the third evaluation, we compared PARSEWeb with Prospector. We showed that PARSEWeb performs better than Prospector. Moreover, PARSEWeb is not limited to the queries of any specific set of frameworks or libraries as Prospector is. In the fourth evaluation, we showed the significance of different techniques used in PARSEWeb.

6.1 Programming Problems

The purpose of this evaluation is to show that PARSEWeb is not limited to the queries of any specific set of frameworks or libraries. We collected two programming problems that

[3]Available at `http://ase.csc.ncsu.edu/parseweb/`

[4]`http://snobol.cs.berkeley.edu/prospector/`

[5]`http://strathcona.cpsc.ucalgary.ca/`

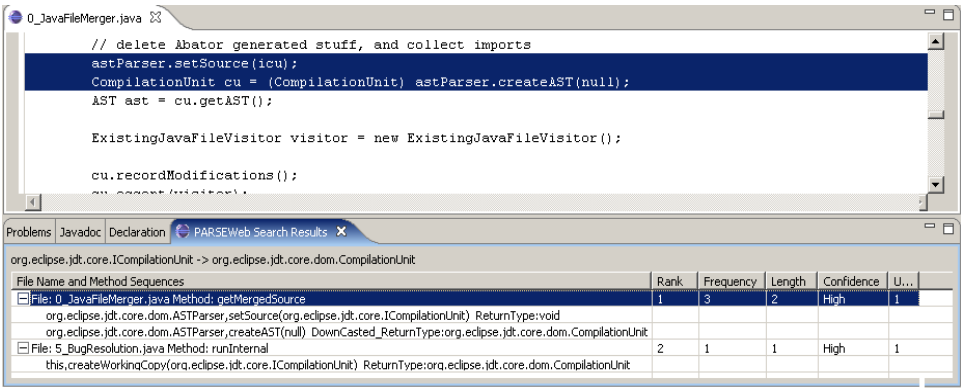

Figure 4: A snapshot of PARSEWeb plugin interface

were posted by developers in forums of existing open source frameworks or libraries and checked whether PARSEWeb is able to suggest solutions for those problems; existing tools such as Prospector and Strathcona cannot address these problems because the queries for these problems fall out of the scope of these two tools: J2SE, Eclipse, and Eclipse GEF (Graphical Editing Framework). The results indicate that PARSEWeb is able to solve these real programming problems.

6.1.1 *Jakarta BCEL User Forum*

The Byte Code Engineering Library (BCEL) provides the ability to analyze, create, and manipulate Java bytecode files. We collected the programming problem "How to disassemble Java byte code" posted in the BCEL forum [3]. The programming problem describes that the programmer is using the BCEL library and has Java byte code under analysis. In the BCEL library, Java byte code is represented through the `Code` class. The programmer wants to obtain a Java assembler command list, which is represented in the form of instructions in the BCEL library. Therefore, we identified the relevant query for the given problem description as "`Code` → `Instruction`". PARSEWeb suggested a solution for the query as shown below:

```
01:FileName:2_RepMIStubGenerator.java MethodName:
   isWriteMethod Rank:1 NumberOfOccurrences:1
02:Code,getCode() ReturnType:#UNKNOWN#
03:CONSTRUCTOR,InstructionList(#UNKNOWN#)
   ReturnType:InstructionList
04:InstructionList,getInstructions()
   ReturnType:Instruction
```

The suggested solution is the same as the response posted in the forum. The programmer can refer to a related code sample by browsing the `isWriteMethod` method in the file `2_RepMIStubGenerator.java`. The original code sample collected from the preceding method is shown below:

```
Code code;
InstructionList il = new InstructionList(code.getCode());
Instruction[] ins = il.getInstructions();
```

In the code sample suggested by PARSEWeb, the return type of `getCode` method is described as `UNKNOWN`. The keyword `UNKNOWN` denotes that the PARSEWeb is not able to infer the return type through its heuristics, because the return type of `getCode` method is not explicitly specified in the code sample. However, PARSEWeb still correctly suggested to pass the return type directly to the constructor of `InstructionList`.

6.1.2 *Dev2Dev Newsgroups*

We applied PARSEWeb on another problem "how to connect db by sessionBean" posted in the Dev2Dev Newsgroups [4]. We transformed the question into the query "`InitialContext` → `Connection`" and used PARSEWeb to obtain the solution. PARSEWeb suggested the following solution, which is the same as the one described in the forum.

```
01:FileName:3_AddrBean.java MethodName:getNextUniqueKey
   Rank:1 NumberOfOccurrences:34
02:InitialContext,lookup(String) ReturnType:DataSource
03:DataSource,getConnection() ReturnType:Connection
```

6.2 Real Project

We next show that PARSEWeb-suggested MISs exist in a real project, and compare the results with those of two related tools: Prospector and Strathcona. As described by Bajracharya et al. [2], there is still a need (but lack) of a benchmark for open source code search that can be used by similar tools for comparing their results. In our evaluation, we used an open source project *Logic* [10] as a subject project. The *Logic* project was developed based on Eclipse Graphical Editing Framework (GEF). The reason for choosing *Logic* for evaluation is that *Logic* is one of the standard example projects delivered with the Eclipse GEF framework.

To be fair in evaluation, we chose all queries from the largest source file ("LogicEditor.java") of the subject project. By choosing the largest file, we can also find many queries that can be used to evaluate all three tools. Within the source file, we picked the first ten available queries of the form "*Source* → *Destination*" from the beginning of the class, and used all three tools to suggest solutions for each query. The query selection process is based on two criteria: a new object type is instantiated from one of the known object types and the selected query is the maximal possible query, which we shall explain next through the code sample extracted from the source file used in the evaluation:

```
01:public void createControl(Composite parent){
02:  PageBook pageBook = new PageBook(parent, SWT.NONE);
03:  Control outline = getViewer().createControl(pageBook);
04:  Canvas overview = new Canvas(pageBook, SWT.NONE);
05:  pageBook.showPage(outline);
06:  configureOutlineViewer();
07:  hookOutlineViewer();
08:  initializeOutlineViewer();}
```

The possible queries that can be extracted from this code sample are "Composite → PageBook", "Composite → Control", and "Composite → Canvas". However, the maximal

Table 1: Evaluation results of programming tasks from the Logic Project

Query		PARSE		PROS		Strath		GCSE
Source	Destination	No	Ra	No	Ra	No	Ra	
IPageSite	IActionBars	1	1	3	1	10	7	1
ActionRegistry	IAction	3	1	4	1	10	3	2
ActionRegistry	ContextMenu Provider	Nil	Nil	2	2	10	3	NA
IPageSite	ISelection Provider	1	1	12	1	10	Nil	5
IPageSite	IToolBar Manager	2	1	12	1	10	6	9
String	ImageDescriptor	10	6	12	Nil	10	Nil	28
Composite	Control	10	2	12	Nil	10	Nil	72
Composite	Canvas	10	5	12	Nil	10	Nil	28
GraphicalViewer	Scrollable Thumbnail	2	1	12	8	10	7	2
GraphicalViewer	IFigure	1	Nil	12	Nil	10	Nil	NA

`PARSE: PARSEWeb, PROS: Prospector, Strath: Strathcona`
`No: Number, Ra: Rank`

possible query among these three queries is "Composite → Canvas", as this query subsumes the other two queries. We consider a task as successful only when the suggested code sample is the same as the code snippet in the corresponding subject project. As our approach tries to suggest solutions from available open source repositories, which may include the subject project under consideration, we excluded the results of PARSEWeb that are suggested from the subject project under consideration.

As both PARSEWeb and Prospector accept the query of the preceding form, we gave constructed queries directly as input. Strathcona compares the context of code given as input and suggests relevant code samples. Therefore, for each evaluation, we built separate driver code that can convey the context of the query. In the driver code, we declared two local variables with the *Source* and *Destination* object types, respectively.

We used PARSEWeb, Prospector, and Strathcona to suggest solutions for each query. The results of our evaluation are shown in Table 1. In Columns "PARSE", "PROS", and "Strath", Sub-columns "No" and "Ra" show the number of results returned by each tool, and rank of the suggested solution that matches with the original source code of the subject project from which the query is constructed. The maximum number of results returned by PARSEWeb, Prospector, and Strathcona are 10, 12, and 10, respectively. The last column "GCSE" shows the index of the source file that contained the solution among the results by GCSE. This index information is extracted by identifying the first source file in which the resultant MIS is found. We found that both PARSEWeb and Prospector performed better than Strathcona. Between PARSEWeb and Prospector, PARSEWeb performed better than Prospector. We next discuss the results of each tool individually.

From the results, we observed that PARSEWeb suggested solutions for all queries except for two. The reason behind the better performance of PARSEWeb is that PARSEWeb suggests solutions from reusable code samples. We inspected queries for which PARSEWeb could not suggest any solution and found the reason is a limitation of our approach in analyzing partial code samples. We elaborate this limitation in Section 7.

Prospector tries to solve the given query using API signatures. Therefore, it can often find some feasible solution for a given query, as it can find a path from the given *Source* to *Destination*. One reason for not getting complete results with Prospector in our evaluation could be that Prospector shows only first twelve results of the given query. Due to this limitation, the required solution might not have shown in the suggested set of solutions. Prospector solves the queries through API signatures and has no knowledge of which MISs are often used compared to other MISs that can also serve as a solution for the given query. PARSEWeb performs better in this scenario because PARSEWeb tries to suggest solutions from reusable code samples and is able to identify MISs that are often used for solving a given query. For example, for query "`Composite` → `Canvas`", the solution is through an additional class called `PageBook`. Although this solution is often used, Prospector is not able to identify the solution as it can be a less favorable solution from the API signature point-of-view.

We suspect that the reason for not getting good results with Strathcona is that Strathcona cannot effectively address the queries of the form "*Source* → *Destination*". We observed that Strathcona generates relevant solutions when the exact API is included in the search context. But our described problem is to identify that API, as the programmer has no knowledge of which API has to be used for solving the query. Moreover, we found that many code samples returned by Strathcona contain both *Source* and *Destination* object types in either `import` statements or in different method declarations. Therefore, those code samples cannot address our query as no MIS can be derived to lead from *Source* to *Destination* object types.

The results shown in Column "GCSE" indicate the problems that may be faced by programmers in using CSEs directly. For example, to find the solution for the seventh task, the programmer has to check 72 files in the results of GCSE.

6.3 Comparison of PARSEWeb and Prospector

We next present the evaluation results of PARSEWeb and Prospector[6] for 12 specific programming tasks. These tasks are based on Eclipse plugin examples from the *Java Developer's Guide to Eclipse* [1] and are the same as the first 12 tasks used by Sahavechaphan and Claypool [14] in evaluating their XSnippet tool. We have not chosen the remaining 5 tasks used in evaluating the XSnippet tool as they are the same as some previous tasks, but differ in the code context where the tasks are executed. As neither PARSEWeb nor Prospector considers the code context, these 5 tasks are redundant to use in our evaluation.

The primary reason for selecting these tasks are that their characteristics include different Java programming aspects like object instantiation via a constructor, a static method, and a non-static method from a parent class. These tasks also require downcasts and have reasonable difficulty levels. For each task, all necessary Java source files and Jar files are provided and code for getting the *Destination* object from the *Source* object is left incomplete. We used open source projects such as `org.eclipse.jdt.ui`, and examples from Eclipse corner articles[7] for creating the necessary en-

[6]We chose only Prospector for detailed comparison because another related tool XSnippet [14] was not available and Strathcona did not perform well in addressing the described problem based on the previous evaluation.

[7]`http://www.eclipse.org/articles/`

Table 2: Evaluation results of programming tasks previously used in evaluating the XSnippet tool

Query		PARSE	PROS
Source	Destination		
ISelection	ICompilationUnit	Yes	No
IStructuredSelection	ICompilationUnit	Yes	Yes
ElementChangedEvent	ICompilationUnit	Yes	Yes
IEditorPart	ICompilationUnit	Yes	Yes
IEditorPart	IEditorInput	Yes	Yes
ViewPart	ISelectionService	Yes	Yes
TextEditorAction	ITextEditor	Yes	No
TextEditorAction	ITextSelection	Yes	No
ITextEditor	ITextSelection	Yes	Yes
AbstractDecorated TextEditor	ProjectViewer	No	No
TextEditor	IDocument	Yes	No
TextEditor	ITextSelection	Yes	Yes

vironment. We used PARSEWeb and Prospector to suggest solutions for each query. A task is considered as successful if the final code can be compiled and executed, and the required functionality is enabled with at least one suggested solution. The task is also considered as successful if the suggested solution acts as a starting point and the final code could be compiled with some additional code. Prospector can generate compilable code for its suggested solutions, but the current implementation of PARSEWeb suggests only the frequent MISs and code samples, but cannot directly generate compilable code. Therefore, we manually transformed the suggested sequences into appropriate code snippets.

The results of our evaluation for the 12 programming tasks are shown in Table 2. PARSEWeb is not able to suggest solution for only one query, whereas Prospector failed to suggest solutions for five queries. This result demonstrates the strength of PARSEWeb as it suggests solutions from reusable code samples gathered from publicly available source code repositories. A summary of percentage of tasks successfully completed by each tool along with the results collected from the XSnippet [14] approach is shown in Figure 5. The x axis shows different tools and the y axis shows the percentage of tasks successfully completed by each tool. The "XSnippet1" and "XSnippet2" entries show two XSnippet query-type techniques: *Type-Based Instantiation Query* (IQ_T) and *Generalized Instantiation Query* (IQ_G), respectively. PARSEWeb performed better than Prospector and XSnippet's IQ_T query type. The results of PARSEWeb are at par with XSnippet's IQ_G query type. However, the IQ_G query type of XSnippet cannot effectively address the issue targeted by PARSEWeb as this query type simply returns the set of all code samples contained in the sample repository that instantiate the given *Destination* object type, irrespective of the *Source* object type. Moreover, XSnippet is also limited to the queries of a specific set of frameworks or libraries.

6.4 Significance of PARSEWeb Techniques

We next show the significance and impact of different techniques used in PARSEWeb. We picked some of the queries in preceding evaluations and analyzed different techniques of our approach. The results of our analysis are shown in Table 3. The table shows the number of identified MISs after applying respective techniques. As shown in the results, the method inlining technique increases the possible number of sequences, whereas the sequence postprocessing

Table 3: Evaluation results of PARSEWeb internal techniques

Query		Simple	Method Inline	Post Process	Query Split
Source	Destination				
TableViewer	TableColumn	21	23	2	2
IWorkbench	IEditorPart	13	17	8	8
IWorkBench Page	IStructured Selection	5	6	1	1
Composite	Control	26	29	24	24
IEditorSite	ISelectionService	Nil	Nil	Nil	2

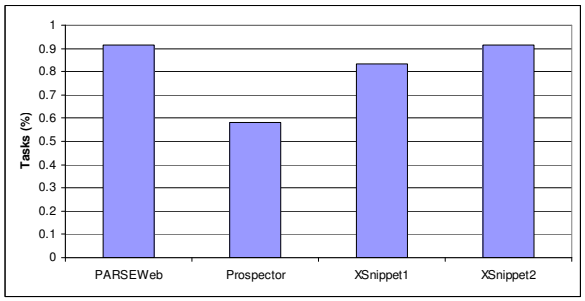

Figure 5: Percentage of tasks successfully completed by PARSEWeb, Prospector, and XSnippet

technique helps in reducing the number of sequences by clustering similar sequences. The query splitting heuristic helps address the lack of code samples by splitting the query into different sub-queries.

6.5 Summary

The primary advantage of PARSEWeb compared to other related tools is that PARSEWeb is not limited to the queries of any specific set of frameworks or libraries. We showed this advantage in the first part of our evaluation. Although Prospector solves the queries of a specific set of frameworks or libraries from the API signatures, we showed that the results of PARSEWeb are better than the results of Prospector. The reason is that Prospector has no knowledge of which MISs are often used compared to other possible sets of sequences. This lack of information often results in irrelevant solutions. Although both PARSEWeb and Strathcona suggest solutions from code samples, the results of PARSEWeb are better than Strathcona because the number of available code samples are limited for Strathcona. Moreover, PARSEWeb has many specialized heuristics compared to Strathcona for helping identify the required MIS. We also showed that GCSE alone cannot handle the queries of the form "*Source* → *Destination*", and showed the significance and impact of different techniques in PARSEWeb.

7. DISCUSSION AND FUTURE WORK

In our current implementation, PARSEWeb interacts with Google Code Search Engine [5] for gathering relevant code samples. Therefore, our current results are dependent on code samples returned by GCSE. We observed that PARSEWeb is not able to suggest solutions for some of the queries due to lack of relevant code samples in the results returned by GCSE. We found this limitation when we evaluated with open source projects *ASTView*[8] and *Flow4j*[9], where Prospector or Strathcona also could not work well. We alleviated

[8] `http://www.eclipse.org/jdt/ui/astview/index.php`
[9] `http://sourceforge.net/projects/flow4jeclipse`

this problem to a certain extent through the query splitting heuristic. To address the problem further, We plan to extend our tool to collect code samples from other CSEs such as Koders [8], and analyze the results to compare different CSEs. We expect that the addition of new source code repositories to CSEs can also help in alleviating the current problem.

As our approach deals with code samples that are often partial and not compilable, there are a few limitations in inferring the information related to method invocations. We explain the limitation of our approach through the code sample shown below:

```
QueueConnectionFactory factory = jndiContext.lookup("t");
QueueSession session = factory.createQueueConnection()
    .createQueueSession(false,Session.AUTOACKNOWLEDGE);
```

In the second expression of this code sample, our heuristics cannot infer the receiver type of the `createQueueSession` method and the return type of the `createQueueConnection` method. The reason is lack of information regarding the intermediate object returned by the `createQueueConnection` method. If this intermediate object is either *Source* or *Destination* of the given query, we cannot suggest this MIS as a solution. Because of this limitation, our approach could not suggest solutions to some of the queries used in evaluation.

Another issue addressed by our approach is the processing of data returned by GCSE [5]. A query such as "`Enumeration` → `Iterator`" results in nearly 22,000 entries. To avoid high runtime costs of downloading and analyzing the code samples, PARSEWeb downloads only a fixed number of samples, say N. This parameter N is made as a configurable parameter. The default value of N is set to 200, which is derived from our experiments and identified to be large enough to make sure that only little useful information is lost during downloading.

The current implementation of PARSEWeb can also accept queries including only the *Destination* object type. Such queries can help programmers if they are not aware of the *Source* object type. However, the number of results returned by PARSEWeb for such queries are often large (10 to 50) due to lack of knowledge of the *Source* object type. Therefore, to reduce the number of results and help programmers in quickly identifying the desired MIS, we plan to extend our tool to infer the *Source* object type from the given code context. We also plan to extend our tool to automatically generate compilable source code from the selected MIS.

8. CONCLUSION

We have developed PARSEWeb, an approach for addressing problems faced by programmers in reusing existing frameworks or libraries. Our approach accepts queries of the form "*Source* → *Destination*" as input and suggests method-invocation sequences that can take the *Source* object type as input and result in the *Destination* object type. Our approach collects the relevant code samples, which include *Source* and *Destination* object types, on demand from CSE. This new idea of collecting sources on demand makes our approach independent of any specific set of frameworks or libraries. Our approach includes several heuristics to deal with partial and non-compilable code downloaded from CSE. Our query splitting heuristic addresses the problem of lack of code samples by splitting the given query into multiple sub-queries. We have conducted four different evaluations on our approach. Through these evaluations, we showed that our approach is effective in addressing the issues faced by programmers and performed better than existing related tools.

Acknowledgments

This work is supported in part by NSF grant CNS-0720641 and ARO grant W911NF-07-1-0431.

9. REFERENCES

[1] J. Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Professional, 2004.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proc. of OOPSLA Companion*, 2006.

[3] Jakarta BCEL user forum, 2001. `http://mail-archives.apache.org/mod_mbox/jakarta-bcel-user/200609.mbox/thread`.

[4] Dev2Dev Newsgroups by developers, for developers, 2006. `http://forums.bea.com/bea/message.jspa?messageID=202265042&tstart=0`.

[5] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[6] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Proc. of ICSE*, pages 117–125, 2005.

[7] Jung the Java Universal Network/Graph Framework, 2005. `http://jung.sourceforge.net/`.

[8] The Koders source code search engine, 2005. `http://www.koders.com`.

[9] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. In *IEEE Software*, pages 35–39, 2003.

[10] Logic Project based on Eclipse GEF, 2006. `http://www.eclipse.org/downloads/download.php?file=/tools/gef/downloads/drops/R-3.2.1-&200609211617/GEF-examples-3.2.1.zip`.

[11] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. of PLDI*, pages 48–61, 2005.

[12] Y. Matsumoto. *A Software Factory: An Overall Approach to Software Production*. In P. Freeman ed., Software Reusability. IEEE CS Press, 1987.

[13] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar Java classes using tree algorithms. In *Proc. of MSR*, pages 65–71, 2006.

[14] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proc. of OOPSLA*, pages 413–430, 2006.

[15] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. of MSR*, pages 54–57, 2006.

[16] Y. Ye and G. Fischer. Supporting reuse by delivering taskrelevant and personalized information. In *Proc. of ICSE*, pages 513–523, 2002.