Code Completion using Keyword Queries in Java

Greg Little and Robert C. Miller
CSAIL MIT
{glittle,rcm}@mit.edu

Abstract

We propose a code completion technique that uses a small number of *keywords*, provided by the user in a code editor, to generate method calls and field references in Java. The keywords are interpreted as a search query over a space of possible expressions that might appear at the current cursor position in the user's code editor. This search space is obtained from the context around the cursor position, including local variables, methods, imported classes, and the type required at the cursor position. The syntactic and type constraints of Java often make it possible to query this space with a small number of keywords, even though the search space may contain thousands of methods, and even though the generated code may require nested method calls and field references. We present two efficient algorithms for converting a keyword query into generated code. The first uses a genetic algorithm to first arrange the user's keywords into an approximation of an abstract syntax tree; the second uses dynamic programming to find method calls that generate each possible type. We evaluate both algorithms using a corpus of keyword queries mined automatically from open-source software. An interesting result from this evaluation is that the information content of method calls in Java is not particularly high: even if punctuation is completely removed and tokens are split and reordered, more than 95% of the method calls we tested could be completely reconstructed based on the remaining keywords and type constraints.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques–program editors, object-oriented programming

General Terms Experimentation, Languages

Keywords Java, Auto-complete, Code Assistants

[Copyright notice will appear here once 'preprint' option is removed.]

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        add line
    }
    return lines;
}
```

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        lines.add(in.readLine());
    }
    return lines;
}
```

Figure 1. Example keyword query completion

1. Introduction

Integrated Development Environments (IDEs) have made significant advancements in helping programmers write code, but much remains to be explored. We are researching the potential of keywords as hints for generating code. For example, imagine a programmer wants to add the next line of text from a stream to a list. We want to let them enter:

add line

and have the computer automatically suggest:

`lines.`**`add`**`(in.read`**`Line`**`());`

More generally, we would like to take a *keyword query* (a string of keywords) and use it to search the space of valid expressions, given the current context in the code. Note that order is unimportant in this query, and some keywords may be omitted. Missing information may be inferred by exploiting type constraints.

Our current prototype is an extension to Eclipse's auto-complete feature that allows the user to enter a keyword query directly into their source code, press Ctrl-Space, and have the keywords replaced with compilable code that is a good match for the keywords, as shown in Figure 1.

This system decreases the cognitive load of coding in several ways. First, keyword queries are shorter than code, and easier to type. Second, the user does not need to recall all the lexical components (e.g. variable names and methods) involved in an expression, since the computer may be able

to infer some of them. Third, the user does not need to type the syntax for calling methods. In fact, the user does even need to *know* the syntax, which may be useful for users who switch between languages often.

This programming paradigm is analogous to searching the web. We would like to simulate the concept of a web page containing every legal Java expression in the current context. The user then needs to enter a standard query for the expression they want. Note that we do not anticipate users trying to write very long or nested lines of code this way. If they wanted to build a large nested expression, we would expect them to only enter keywords relevant to the lowest part of it, and the system could suggest expression templates with holes that need to be filled in.

In this paper, we propose two algorithms for finding keyword query completions quickly. This work builds on keyword programming techniques in Chickenfoot [4] and Koala [3], but scales the algorithms to the larger domain of Java.

This work is similar to Prospector [5] and XSnippet [8], in that it suggests Java code given a query. However, our system uses keywords to guide the search. We also generate code, as opposed to search through existing snippets. This broadens our scope to small projects that have little code to search through.

Our key contributions are:

- Two algorithms for translating keyword queries into Java code efficiently.
- An Eclipse Plug-in that allows users to perform keyword query completions in their Java source code.
- An evaluation of the algorithms and plug-in on an artificial corpus. (We are able to reconstruct 95% of the expressions we obfuscate).

2. Model

The scenario we want to model is the following: the user is at some location in their source code, and they have entered a keyword query. They intend the keywords to produce a Java expression. In order to find the expression, we need to model the available methods, fields and local variables, and how they fit together using Java's type system. The resulting model defines the search space.

Our model M is the triple (T, L, F), where T is a set of types, L is a set of labels, and F is a set of functions.

2.1 Type Set: T

Each type is represented by a unique name. We get this from Java's fully qualified name for the type. Examples include `int` and `java.lang.Object`.

We include all the primitive types and classes referenced in the current source file, as well as object types that are accessible with at most 2 method calls. This number is arbitrary, but all the experiments on our algorithms assume that we are building nested expressions with depth at most 2, e.g. `func0(func1(func2()))`.

We also define $sub(t)$ to be the set of both direct and indirect subtypes of t. This set includes t itself, and anything assignment-compatible with t. We also include a universal supertype $\top$, such that $sub(\top) = T$. This is used when we want to place no restriction on the resulting type of an expression.

2.2 Label Set: L

Each label is a sequence of keywords. We use labels to represent method names, so that we can match them against the keywords in a query.

To get the keywords from a method name, we break up the name at capitalization boundaries. For instance, the method name `currentTimeMillis` is represented with the label (**current**, **time**, **millis**). Note that capitalization is ignored when labels are match against the user's keywords.

2.3 Function Set: F

Functions are used to model each component in a Java expression that we want to match against the user's keyword query, including methods, fields, and local variables.

We define a function as a tuple in $T \times L \times T \times ... \times T$. The first T is the return type, followed by the label, and all the parameter types. As an example, the Java function:

```
String toString(int i, int radix)
```

is modeled as

(`java.lang.String`, (**to**, **string**), `int`, `int`)

For convenience, we'll also define $ret(f)$, $label(f)$ and $params(f)$ to be the return type, label, and parameter types, respectively, of a function f.

2.4 Function Tree

The purpose of defining types, labels and functions is to model Java expressions that the user may intend with their keyword query. We model a Java expression as a function tree. Each node in the tree is associated with a function from F, and obeys certain type constraints.

In particular, a node is an n-tuple consisting of an element from F followed by some number of child nodes. For a node n, we'll define $func(n)$ to be the function, and $children(n)$ to be the list of child nodes. We'll require that the number of children in a node be equal to the number of parameter types of the function, i.e., $|children(n)| = |params(func(n))|$. We'll also require that the return types from the children "fit" into the parameters, i.e.,
$\forall_i ret(func(children(n)_i)) \in sub(params(func(n))_i)$.

Note that in the end, the system will render the function tree as a syntactically-correct and type-correct Java expression.

2.5 Java Mapping

We now provide the particulars for mapping various Java elements to T, L and F. Most of these mappings are natural and straightforward.

2.5.1 Primitive Types

Primitive types are modeled as types in T, using their traditional names (int, long, double, etc...). Note that because of boxing and unboxing in Java 1.5, we include java.lang.Integer in sub(int).

2.5.2 Classes

A class or interface c is modeled as a type in T, using its fully qualified name (e.g. java.lang.String). Any class that is assignment compatible with c is added to $sub(c)$, including any classes that extend or implement c.

2.5.3 Methods

Methods are modeled as functions that take their receiver object as the first parameter. For instance, the method:

```
public Object get(int index)
```

of java.util.Vector is modeled as:

(java.lang.Object, (**get**), java.util.Vector, int)

2.5.4 Fields

Fields become functions that return the type of the field, and take their object as a parameter. For instance, the field:

```
public int x
```

of java.awt.Point is modeled as:

(int, (**x**), java.awt.Point)

2.5.5 Local Variables

Local variables are simply functions that return the type of the variable and take no parameters, e.g., the local variable int i inside a for-loop is modeled as (int, (**i**)).

2.5.6 Constructors

Constructors are modeled as functions that return the type of object they construct. We use the keyword **new** and the name of the class as the function label, e.g., the constructor for java.lang.Integer that takes a primitive int as a parameter is represented by:

(java.lang.Integer, (**new**, **integer**), int)

2.5.7 Members

Member methods and fields of the class containing the keyword query are associated with an additional function, to support the Java syntax of accessing these members with an assumed this token. The new function doesn't require the object as the first parameter. For instance, if we are writing code inside java.awt.Point, we would create a function for the field x like this: (int, (**x**)). Note that we can model the keyword this with the additional function (java.awt.Point, (**this**)).

2.5.8 Statics

The Java syntax for calling methods does not usually require writing type names, which is why we do not include type names in the label when modeling methods. However, type names are used to disambiguate static methods and fields, since they have no receiver object.

To support this syntax, we model static methods and fields with an additional function. Our strategy is to omit the object as the first parameter, but add the class name as part of the function label. For instance

```
static double sin(double a)
```

in java.lang.Math is modeled by adding both:

(double, (**sin**), java.lang.Math, double), and

(double, (**math**, **sin**), double)

We probably don't need to keep the first version in the case of java.lang.Math, since this object cannot be instantiated, but in general, we keep both versions since Java permits calling static methods on live objects.

An alternate approach is to have two types, instead of two functions. For instance, we could add the additional type static:java.lang.Math to T, and then have one function for sin, namely

(double, (**sin**), static:java.lang.Math, double)

We would then make java.lang.Math a subtype of static:java.lang.Math, and we would add a special constructor function for static types, like

(static:java.lang.Math, (**math**))

.

2.5.9 Generics

We support generics explicitly, i.e., we create a new type in T for each instantiation of a generic class or method. For instance, if the current source file contains a reference to both Vector<String> and Vector<Integer>, then we add both of these types to T. We also add all the methods for Vector<String> separately from the methods for Vector<Integer>. For example, the get method would produce both

(String, (**get**), Vector<String>, int), and

(Integer, (**get**), Vector<Integer>, int)

The motivation behind this approach is to keep the model simple, and programming language agnostic. In practice, it does not explode the type system too much, since relatively few separate instantiations are visible at a time.

An alternate approach is to simply erase generics, and treat them as Java 1.4 style objects, e.g., treat Vector<String>

as just `Vector`. The downside is that the search problem becomes less constrained. On the other hand, T and F are smaller, making the algorithms faster.

2.5.10 Other Mappings

We have experimented with additional mappings, although we have not yet done a formal evaluation of them. These include numeric and string literals, variable assignment, and array indexing. We have also considered ways to model control flow. We discuss these mapping in more detail in the Extensions section.

3. Problem

Now that we have a formal model of the domain, we can provide a formal statement of the problem that our algorithm must solve.

The input to the algorithm consists of a model M, and a keyword query. We also supply a desired return type, which we make as specific as possible given the source code around the keyword query. (If any type is possible, we supply $\top$ as the desired return type).

The output is a valid function tree, or possibly more than one. The root of the tree must return the desired type, and the tree should be a good match for the keywords according to some metric.

Choosing a good similarity metric between a function tree and a keyword query is the real challenge. We need a metric that matches human intuition, as well as a metric that is easy to evaluate algorithmically.

Our metric is based on the simple idea that each input keyword is worth 1 point, and a function tree earns that point if it "explains" the keyword by matching it with a keyword in the label of one of the functions in the tree. Note that the details of the metric vary somewhat between the two algorithms presented below, and depend on their implementations and heuristics.

4. Algorithms

We have developed two different algorithms for solving this problem. The first algorithm uses a genetic algorithm to explore possible ways to arrange keywords into trees. The second algorithm is a bottom up dynamic programming algorithm that keeps track of the best functions to achieve different types at each level of the tree.

The first algorithm assumes that the input keywords will adhere to phrase structure, i.e., each subtree of the function tree will map to a contiguous group of keywords. This assumption helps constrain the search, but our implementation does not support function inference. The second algorithm ignores the order of the keywords completely, but it can infer functions not associated with any keywords.

4.1 Algorithm 1: Keyword Tree

In the Keyword Tree algorithm, we define a genome that arranges keywords into a parse tree. We then score the parse tree based on the function trees we can create from it. Finally, we use a genetic algorithm to search over the space of possible parse trees for the one that can produce the highest scoring function tree.

4.1.1 Genome

Although genetic algorithms can be applied to trees directly, this can be complicated. Our algorithm uses a simple linear genome that specifies how to arrange the keywords into a parse tree. This allows us to use standard cross-over to merge genes.

Consider that we have n keywords, $k_1, k_2, ..., k_n$. To start, we assume that each keyword is a singleton tree. The genome consists of $n-1$ components $c_1, c_2, ..., c_{n-1}$. Each component can be thought of as existing between two keywords. In particular, we define $left(c_i) = k_i$ and $right(c_i) = k_{i+1}$. Each component c is a 3-tuple: $(rule, order, index)$ where:

- $rule(c)$ is an element of $\{\leftrightarrow, \nearrow, \nwarrow\}$ that defines the relationship between $left(c)$ and $right(c)$. The $\leftrightarrow$ tells us to make $left(c)$ and $right(c)$ part of the same node. If we let $root(a)$ denote the root node of the tree containing a, then the $\nearrow$ tells us to make $root(left(c))$ a child of $root(right(c))$ (and $\nwarrow$ has the opposite meaning).
- $order(c)$ is a real number between 0 and 1 that determines the order in which to apply the rules. We generate these randomly, and assume each $order$ value will be unique.
- $index(c)$ is also number in the range 1 to $n-1$ that specifies where to insert nodes when we make one node the child of another node. If a node has n_c children, we insert at position $index \mod (n_c + 1)$.

Figure 2 is psuedocode for constructing the parse tree. Let $root(a)$ denote the root node of the tree containing a. Let $combine(a, b)$ have the side effect of combining the nodes a and b into a single node. Let $insert(a, b, i)$ have the side effect of inserting b into a at position i (using 0-based indexing).

Here is an example to illustrate. Consider the the following keyword query, based on the Java expression `boxes.addBox(b)`:

boxes add box b

There are 4 keywords, so there are 3 components in the genome. Let's say the components are: ($\nearrow$, 0.3, 2), ($\leftrightarrow$, 0.2, 3), and ($\nwarrow$, 0.1, 1). We can visualize these sitting between the keywords:

boxes $_{(\nearrow, 0.3, 2)}$ **add** $_{(\leftrightarrow, 0.2, 3)}$ **box** $_{(\nwarrow, 0.1, 1)}$ **b**

In the first pass, we use the components with a $rule$ of $\leftrightarrow$ to combine neighboring keywords into single nodes. We

$$
\begin{array}{l}
C \leftarrow \{c_1, c_2, ..., c_{n-1}\}\\
\textbf{for each } c \in C\\
\quad \textbf{do} \begin{cases} \textbf{if } rule(c) = \leftrightarrow \\ \quad \textbf{then} \begin{cases} combine(root(left(c)), root(right(c))) \\ C \leftarrow C - \{c\} \end{cases} \end{cases}\\
\textbf{while } C \neq \emptyset\\
\quad \textbf{do} \begin{cases} \text{/* get component with smallest order */} \\ c \in \{a \mid a \in C \textbf{ and } \forall_i \; order(a) \leq order(a_i)\} \\ \text{/* apply the rule for } c \text{ */} \\ \textbf{if } rule(c) = \nearrow \\ \quad \textbf{then} \begin{cases} parent \leftarrow root(right(c)) \\ child \leftarrow root(left(c)) \end{cases} \\ \quad \textbf{else if } rule(c) = \nwarrow \\ \quad \textbf{then} \begin{cases} parent \leftarrow root(left(c)) \\ child \leftarrow root(right(c)) \end{cases} \\ i \leftarrow index(c) \mod (|children(parent)| + 1) \\ insert(parent, child, i) \\ C \leftarrow C - \{c\} \end{cases}
\end{array}
$$

Figure 2. Pseudocode for constructing a parse tree.

see such a component between **add** and **box**, so we combine them into a single node:

boxes $_{(\nearrow,\, 0.3,\, 2)}$ **add-box** $_{(\nwarrow,\, 0.1,\, 1)}$ **b**

Next, we find the remaining component with the lowest $order$, which is ($\nwarrow$, 0.1, 1) between **add-box** and **b**. The $rule$ is $\nwarrow$, so we make **b** a child of **add-box**. In this case there is only one place to insert it, so we can ignore the $index$. We'll denote making **a** the child of **b** with **b(a)**, giving us:

boxes $_{(\nearrow,\, 0.3,\, 2)}$ **add-box(b)**

The final component tells us to make **boxes** a child of **add-box**. Since **add-box** already has a child **b**, we use the $index$ of 2 to determine where to insert the new child. We take $index$ modulus one more than the number of children already present, which is written as $2 \mod (1 + 1)$. This evaluates to 0, so we insert **boxes** as the new first child of **add-box**, giving us:

add-box(boxes, b)

Note that this parse tree has the same structure as the abstract syntax tree (AST) for the original Java expression `boxes.addBox(b)`, which is good. This shows us that the genome is expressive enough to build this tree from the keywords. We would like to know that in general, the genome is expressive enough to build any valid parse tree from the keywords.

Proof Sketch: The idea of the proof is to construct the component values given a desired tree. First we note that every edge in the tree represents a component. (There are also components between the keywords in a single node, but these are trivial to deal with; we just set the $rule$ for each of these components to $\leftrightarrow$). Now we take any one of the edges connected to the root, and we find the corresponding component c. Then we set $order(c)$ to the largest unused $order$ value, and we set the component $index(c)$ to the 0-based index of the child connected to this edge. Then we set $rule$ to $\nearrow$ if $root(right(c))$ is the root node, and we set $rule$ to $\nwarrow$ if $root(left(c))$ is the root node.

Next, we remove this edge from the tree and repeat the process (now we are allowed to select any edge from any of the remaining roots in the resulting forest of trees). It is simple to show that applying these construction rules in the reverse order will build up the tree exactly the way we tore it down. ■

4.1.2 Fitness Function

A parse tree defines a tree structure, but it does not specify which function we should use at each node. Consider the parse tree from our previous example:

add-box(boxes, b)

This has the same structure as these function trees:

A. addBox(boxes(), b()),

B. removeBox(boxes(), b()), and

C. synchronizedAddBox(boxes(), b())

However, we would like to say that it corresponds more to **A** than to **B**, because the root node of **A** matches more keywords with the root node of the parse tree. Now **A** and **C** both match the same number of keywords, but we would like to say that **C** is worse because it includes an unmatched keyword **synchronized**.

We formalize this notion in the way we score a function f with a node n. First we let $key(n)$ represent the list of keywords in n. The score equals the number of shared keywords between $key(n)$ and $label(f)$, minus a small amount (0.01) for each keyword that is not shared. Note that $key(n)$ and $label(f)$ are lists, and might contain repeated keywords. If a keyword k appears x times in $key(n)$ and y times in $label(f)$, then we award $\min(x, y)$ points for this keyword, and subtract $0.01 * (\max(x, y) - \min(x, y))$ points.

To determine the score for the entire parse tree, we define $score(n)$ as the score for the parse tree rooted at n. We also define $score(n, t)$ as the score of the best function tree rooted at n that returns type t. It is often true that $score(n) = \max_{t \in T} score(n, t)$, but not always, since we are not always able to find a valid function tree that corresponds with the parse tree.

We calculate $score(n)$ and $score(n, t)$ recursively. To process node n, we first process the children of n. Next, for each function f, we will calculate a score s. We will update $score(n)$ and $score(n, ret(f))$ based on s.

We initialize s with the score calculated between $label(f)$ and $key(n)$, as described above. To speed up the algorithm,

we skip the function if $s \leq 0$. Next, we add a point to s for each genome component with rule $\leftrightarrow$ that went into forming n.

We then add points to s associated with each child n' that is associated with a parameter type p from $params(f)$. If $score(n', t)$ is defined for some $t \in sub(p)$, then we add $\max_{t \in sub(p)} score(n', t)$ to s, in addition to a point for the genome component that attached this child.

If the child doesn't return any types we can use, or is not associated with a parameter type (because we have more children than parameters), then we don't add a point for the genome component that attached this child. However, we do add $score(n')$ to s, since we may be able to use this subtree when the genes get mutated or merged with other genes.

Finally, we update $score(n)$ to $\max(s, score(n))$. We also update $score(n, ret(f))$ to $\max(s, score(n, ret(f)))$, but only if we found children returning the proper types for each parameter of f.

4.1.3 Genetic Algorithm

We create an initial population of 100 genomes, and initialize each genome with random values for each component. Each subsequent generation has only 10 genomes. We run the algorithm for 100 generations. Each generation is created from the previous generation. The best genome from the previous generation is copied directly into the new one, while every other genome is created from two randomly selected parents: the previous generation is put in order of best fitness score, and the index of each parent is chosen using a normal distribution in the following equation: $|\lfloor N(0, 3^2) \rfloor| \mod 10$.

New genomes are created using cross-over and mutation. We choose one cross-over point. Components before the cross-over point are copied from one parent, and components after the cross-over point are copied from the other. Each element of each component has a 20% chance of mutating to a random value.

4.1.4 Extraction

After running the genetic algorithm, we end up with a high-scoring parse tree. Now we need to extract the best function tree from it. We do this by adding additional bookkeeping to the scoring algorithm to keep track of which function achieves $score(n, t)$ at node n for a given type t.

Given this information, we use a simple recursive algorithm to build the function tree. It looks at the root node to find the best function that returns the desired type (or any subtype). Then we repeat this process recursively to find the best function that achieves the desired parameter type from each child.

4.1.5 Limitations

This algorithm will only consider trees that obey phrase structure, i.e., each subtree is constrained to be a contiguous group of keywords. This means that if a users enters **my print message**, when they mean `print(myMessage)`, the system will not consider the tree **print(myMessage)**, since this tree groups **my** and **message** into a subtree, whereas they are not contiguous in the input. However, the algorithm may still make the proper suggestion if it cannot find a use for the isolated keyword **my**. For instance, it could find the tree **myPrint(message)**, which may still result in the function tree `print(myMessage)` (since **myPrint** is a pretty good match for **print**, and **message** for **myMessage**).

A bigger limitation is that this algorithm does not support function inference. This means that a user could not type **print "hello world"**, and expect the system to generate `System.out.println("hello world")`, since the system cannot infer the field `System.out`. Our next algorithm overcomes both of these limitations.

4.2 Algorithm 2: Bottom-Up

This algorithm can be thought of as a dynamic program where we are filling out a table of the form $func(t, i)$, which tells us which function to use to achieve type t, if the function tree can be at most height i. It also tells us the score we expect to achieve with the resulting function tree.

Calculating $func(t, 0)$ for all $t \in T$ is relatively easy. We only consider functions that take no parameters, since our tree height is bounded by 0. Then for each such function f, we give it a score based on its match to the user's keywords, and we associate this score with $ret(f)$. Then for each $t \in T$, we update $func(t, 0)$ with the best score associated with any subtype of t.

Instead of a scalar value for the score, we use an Explanation Vector. We'll explain what this is before talking about the next iteration of the dynamic program.

4.2.1 Explanation Vector

The idea of the Explanation Vector is to encode how well we have explained the input keywords. If we have n keywords $k_1, k_2, ..., k_n$, then the Explanation Vector has $n+1$ elements $e_0, e_1, e_2, ..., e_n$. Each element e_i represents how well we have explained the keyword k_i on a scale of 0 to 1; except e_0, which represents explanatory power not associated with any particular keyword. When we add two Explanation Vectors together, we ensure that the resulting elements $e_1, e_2, ..., e_n$ are capped at 1, since the most we can explain a particular keyword is 1.

Before we do anything else, we calculate an Explanation Vector $expl(f)$ for each function $f \in F$. In the common case, We set e_i to 1 if $label(f)$ contains k_i. For instance, if the input is:

is boxes empty

and the function f is (`boolean`, (**is**, **empty**), `List`), then $expl(f)$ would be:

$(e_0, 1_{\textbf{is}}, 0_{\textbf{boxes}}, 1_{\textbf{empty}})$

In the Keyword Tree algorithm, we penalized unmatched keywords. We do the same here by setting e_0 to $-0.01x$, where x is the number of words appearing in either the input or $label(f)$, but not both. In this case, we set e_0 to -0.01, since the word **boxes** does not appear in $label(f)$.

Now consider the input:

node parent remove node

where `node` is a local variable modeled with the function (`TreeNode`, (**node**)). Since **node** appears twice in the input, we distribute our explanation of the word node between them:

$(e_0, 0.5_{\textbf{node}}, 0_{\textbf{parent}}, 0_{\textbf{remove}}, 0.5_{\textbf{node}})$

In general, we set $e_i = \max(\frac{x}{y}, 1)$, where x is the number of times k_i appears in $label(f)$, and y is the number of times k_i appears in the input.

In this case we set e_0 to -0.03, since there are three words that appear in the input, but not in the function label (we include one of the **node** keywords in this count, since it only appears once in the label).

4.2.2 Next Iteration

In the next iteration, we are trying to compute $func(t, i)$ for all $t \in T$. The basic idea is to consider each function f, and calculate an Explanation Vector by summing the Explanation Vector for f itself, plus the Explanation Vector for each parameter type p found in $func(p, i-1)$.

We can do this, but there is a problem. We no longer know that we have the optimal Explanation Vector possible for this function at this height. Consider the following input:

add x y

and assume we have the functions:

(`int`, (**add**), `int`, `int`),

(`int`, (**x**)), and

(`int`, (**y**)),

If we look in $func(\texttt{int}, 0)$, we are likely to see either (`int`, (**x**)), or (`int`, (**y**)). Let's assume it is (`int`, (**x**)). Now consider what happens in the next iteration when we are processing the function (`int`, (**add**), `int`, `int`). We take the Explanation Vector (-0.02, $1_{\textbf{add}}$, $0_{\textbf{x}}$, $0_{\textbf{y}}$), and we add the Explanation Vector found in $func(\texttt{int}, 0)$, which is (-0.02, $0_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$). This gives us (-0.04, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$).

Now we want to add the Explanation Vector for the second parameter, which is also type `int`. We look in $func(\texttt{int}, 0)$ and find (-0.02, $0_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$) again. When we add it, we get (-0.06, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$), since the keyword components are capped at 1.

But what if we had found the Explanation Vector for (`int`, (**y**))? Then we could have gotten (-0.06, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $1_{\textbf{y}}$), which is better.

To get around this problem, we store the top r functions at each $func(t, i)$, where r is an arbitrary constant.

```
procedure EXTRACT TREE(t, h, e)
 for each f ∈ func(t, i) where i ≤ h
    do { /* create tuple for function tree node */
         n ← (f)
         e_n ← e + expl(f)
         /* put most specific types first */
         P ← SORT(params(f))
         for each p ∈ P
           do { n_p, e_p ← EXTRACT TREE(p, i − 1, e_n)
                /* add n_p as a child of n */
                n ← append(n, n_p) }
         if e_n > best_e
           then { best_e ← e_n
                  best_n ← n } }
 return (best_n, best_e)
```

Figure 3. Pseudocode to extract a function tree.

In our experiments, we chose $r = 3$, except in the case of $func(\texttt{java.lang.Object}, i)$, where we keep the top 5 (since many functions return this type).

Now when we are considering function f at height i, and we are adding Explanation Vectors for the parameters, we are greedy: we add the Explanation Vector that increases our final vector the most, and then we move on to the next parameter. Note that if our parameter type is p, we consider all the Explanation Vectors in each $func(p, j)$ where $j < i$.

4.2.3 Extraction

After we have run the dynamic program to some arbitrary height h (in our case, $h = 2$), we need to extract a function tree.

We use a greedy recursive algorithm (see Figure 3) which takes the following parameters: a desired return type t, a maximum height h, and an Explanation Vector e (representing what we have explained so far). The function returns a new function tree, and an Explanation Vector. Note that we sort the parameters such that the most specific types appear first (t_1 is more specific than t_2 if $|sub(t_1)| < |sub(t_2)|$).

4.2.4 Running Time

We have not yet formally analyzed the running time of these algorithms, but they are both able to generate function trees in well under a second with thousands of functions in F, hundreds of types in T, and up to 6 keywords. With more than 6 keywords, the Keyword Tree algorithm starts to take longer than a second. The Bottom-up algorithm remains about 0.5 seconds even for large inputs (more 12 tokens). These figures are provided in more detail in the evaluation.

5. Eclipse Plug-in

With the Eclipse Plug-in installed, the user may enter keyword queries directly into their source code. Here we show the user entering the keywords **add line**:

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        add line
    }
    return lines;
}
```

Next, the user presses Ctrl-Space to bring up Eclipse's auto-complete menu, we add a hook that does the following:

1. It updates the model M with local variables in the current context. For instance, it sees the local variable `lines`, so it adds the function (`List<String>`, (**lines**)). The model is initialized in the background based on the classes named in the current source file. For instance, when the system sees the class name `List<String>`, it adds all the methods and fields associated with `List<String>`.
2. It figures out where the keyword query begins (it assumes that it ends at the cursor). It uses some heuristics to make this determination, including the nearest Java compilation error, which occurs on the keyword **add** in this example.
3. It tries to determine what Java types are valid for an expression at this location. In this example, the keywords are not nested in a subexpression, so any return type is valid. If the user had instead typed the keywords **in ready** into the condition for the `while` loop, then the system would expect a `boolean` return type.

The system now has all the information it needs to use one of our algorithms. The algorithm will suggest a valid function tree given the constraints. In this example, the Bottom-up algorithm finds the tree **add(lines, readLine(in))**. The system then renders this in Java syntax as:

```
lines.add(in.readLine())
```

Since it is the only completion available, Eclipse automatically replaces the keyword query in the source code:

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        lines.add(in.readLine());
    }
    return lines;
}
```

If the user is not satisfied with the result, they can press Ctrl-Z to undo it. In the future, we plan to support multiple suggestions in the case of ambiguity.

6. Evaluation

We evaluated the plug-in and the algorithms on an artificial corpus. The corpus consists of a variety of open source Java projects. We create artificial keyword queries by finding method calls and obfuscating them (removing punctuation and rearranging keywords). We then pass these keywords to the Plug-in, and record whether it generates the original method call.

Project	Class Files	LOC	Test Sites
Azureus	2277	339628	82006
Buddi	128	27503	7807
CAROL	138	18343	2478
Dnsjava	123	17485	2900
Jakarta CC	41	10082	1806
jEdit	435	124667	25875
jMemorize	95	14771	2604
Jmol	281	88098	44478
JRuby	427	72030	19198
Radeox	179	10076	1304
RSSOwl	201	71097	23685
Sphinx	268	67338	13217
TV-Browser	760	119518	29255
Zimbra	1373	256472	76954

Table 1. Project Statistics

6.1 Projects

We selected 14 projects from popular open source web sites, including sourceforge.net, codehaus.org, and objectweb.org. Projects were selected based on popularity, and our ability to compile them using Eclipse. Our projects include:

1. **Azureus**, an implementation of the BitTorrent protocol.
2. **Buddi**, a program to manage personal finances and budgets.
3. **CAROL**, a library for abstracting away different RMI (Remote Method Invocation) implementations.
4. **Dnsjava**, a Java implementation of the DNS protocol.
5. **Jakarta Commons Codec**, an implementation of common encoders and decoders.
6. **jEdit**, a configurable text editor for programmers.
7. **jMemorize**, a tool involving simulated flashcards to help memorize facts.
8. **Jmol**, a tool for viewing chemical structures in 3D.
9. **JRuby**, an implementation of the Ruby programming language in Java.
10. **Radeox**, an API for rendering wiki markup.
11. **RSSOwl**, a newsreader supporting RSS.
12. **Sphinx**, a speech recognition system.
13. **TV-Browser**, an extensible TV-guide program.
14. **Zimbra**, a set of tools involving instant messaging.

Table 1 shows how many class files and non-blank lines of code each project contains. We also report the number of possible test sites, which we discuss in the next section.

```
public IRubyObject callMethod(RubyModule context, String name, IRubyObject[] args,
        CallType callType) {
    ⋮
    if (method.isUndefined() ||
    ⋮
        IRubyObject[] newArgs = new IRubyObject[args.length + 1];
        System.arraycopy(args, 0, newArgs, 1, args.length);
        newArgs[0] = RubySymbol.newSymbol(getRuntime(), name);

        return callMethod("method_missing", newArgs);
    }
    ⋮
}
```

Figure 4. Example Test Site

6.2 Tests

Each test is conducted on a method call, variable reference or constructor call. We only consider expressions of depth 2 or less, and we make sure that they involve only the Java constructs supported by our model. For example, these include local variables and static fields, but do not include literals or casts. We also exclude expressions inside of inner classes since it simplifies our automated testing algorithm. Finally, we discard test sites with only one keyword as trivial.

Figure 4 shows a valid test site highlighted in the JRuby project. This example is depth 1, because the call to `getRuntime()` is nested within the call to `newSymbol()`. Note that we count nested expressions as valid test sites as well, e.g., `getRuntime()` in this example would be counted as an additional test site.

To perform each test, we obfuscate the method call by removing punctuation, splitting camel-case identifiers, and rearranging keywords (while still maintaining phrase structure). We then treat this obfuscated code as a keyword query, which we pass on to the plug-in. (Note that in these tests, we tell the plug-in explicitly where the keyword query starts and ends).

For example, the method call highlighted in Figure 4 is obfuscated to the following keyword query:

name runtime get symbol symbol ruby new

The plug-in observes the location of this command in an assignment statement to `newArgs[0]`. From this, it detects the required return type:

```
org.jruby.runtime.builtin.IRubyObject
```

The plug-in then passes the keyword query and this return type to one of the algorithms. In this example, it uses the Bottom-up algorithm, and the plug-in returns the Java code:

```
RubySymbol.newSymbol(getRuntime(), name)
```

We compare this string with the original source code (ignoring whitespace), and if it matches exactly, we record the test as a success. We also include other information about the test, including:

- **# Keywords:** the number of keywords in the keyword query.

# Keywords	Samples
2	3330
3	1997
4	1045
5	634
6	397
7	206
8	167
9	86
10	54
11	38
$\geq$ 12	46
⋮	⋮
43	1

Table 2. Samples given # Keywords

- **time:** how many seconds the algorithm spent searching for a function tree. This does not include the time taken to construct the model. Tests were run in Eclipse 3.2 with Java 1.6 on an AMD Athlon X2 (Dual Core) 4200+ with 1.5GB RAM (the algorithms are single threaded).
- **$|T|$:** the number of types in the model constructed at this test site.
- **$|F|$:** the number of functions in the model constructed at this test site.

6.3 Results

The results presented here were obtained by randomly sampling 500 test sites from each project (except Zimbra, which is really composed of 3 projects, and we sampled 500 from each of them). This gives us 8000 test sites. For each test site, we tested both algorithms.

Table 2 shows how many samples we have for different keyword query lengths. Because we do not have many samples for large lengths, we will group all the samples of length 12 or more when we plot graphs against keyword length.

Figure 5 shows the accuracy of each algorithm given a number of keywords. Both algorithms have over 90% accuracy for inputs of 4 keywords or less. In most cases, the Keyword Tree algorithm is significantly more accurate than the Bottom-up algorithm. This may be due in part to the fact that the Keyword Tree algorithm takes advantage of phrase structure, which is something we preserve in the obfuscation.

Figure 6 shows how long each algorithm spent processing inputs of various lengths. The Bottom-up algorithm is slower for very small inputs, but grows at a slow rate, and remains under 500 milliseconds even for large inputs. The Keyword Tree algorithm doesn't scale as well to large inputs, taking more than 4 seconds for inputs 12 keywords long.

Another factor contributing to running time is the size of T and F in the model. Table 7 shows the average size of T

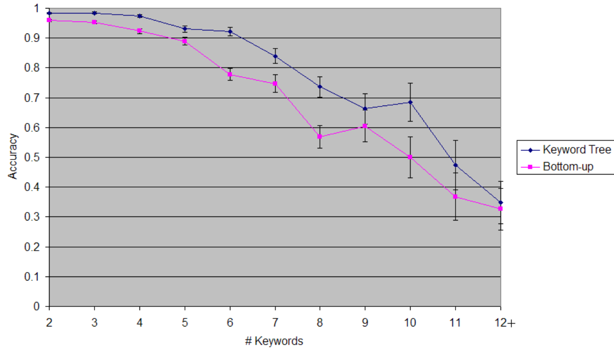

Figure 5. Accuracy given # Keywords

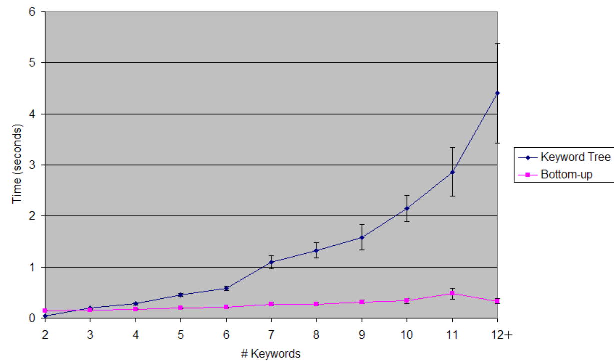

Figure 6. Time given # Keywords

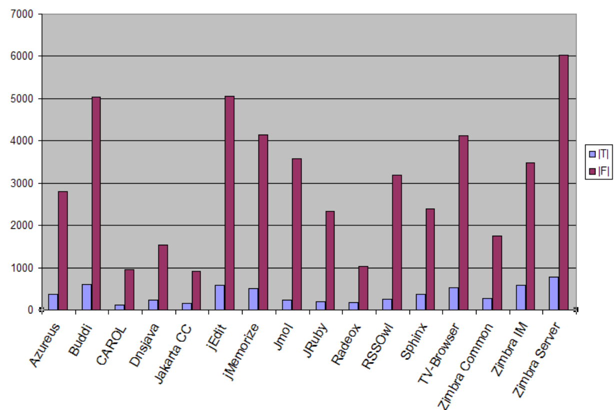

Figure 7. Size of T and F for different projects.

and F for each project. The average size of T ranges from 100 to 800, while the average size of F ranges from 900 to 6000. Figure 8 shows running time as a function of the size of F. We see that the Bottom-up algorithm takes about 1 second when F contains 14000 functions.

To test the inference capabilities of the Bottom-up algorithm, we ran another set of tests on the same corpus. We considered only test sites with nested expressions (i.e. the resulting function tree had at least two nodes). This ensured

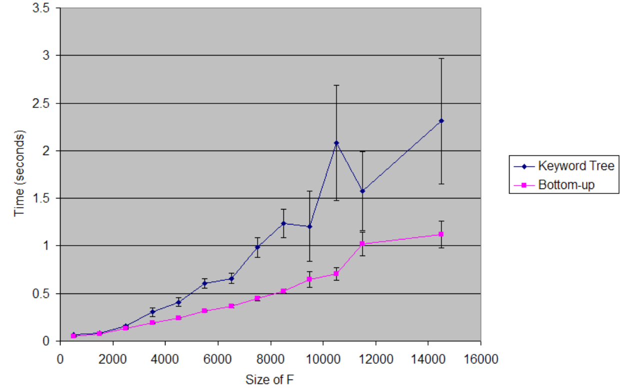

Figure 8. Time given size of F

Keywords provided	Keywords in expression						
	2	3	4	5	6	7	8
0	0.042	0.032	0.028	0.041	0.009	0.012	0.007
1	0.348	0.271	0.239	0.169	0.108	0.073	0.031
2	0.964	0.77	0.562	0.405	0.284	0.26	0.139
3		0.946	0.796	0.647	0.487	0.421	0.322
4			0.895	0.777	0.609	0.467	0.36
5				0.834	0.704	0.593	0.435
6					0.75	0.638	0.6
7						0.683	0.596
8							0.623

Figure 9. Accuracy of inference (Bottom-up algorithm)

that when we provide one keyword, we would definitely have to infer something.

Again, we randomly sampled 500 test sites from each project. At each test site, we ran the Bottom-up algorithm with the empty string as input. Next, we chose the most unique keyword in the expression (according to the frequency counts in L), and ran the algorithm on this. We kept adding the next most unique keyword from the expression to the input. The left side of Figure 9 shows the number of keywords we provided as input. The table shows the accuracy for different expression lengths (measured in keywords).

6.4 Discussion

The tests we ran give us a rough upper and lower bound on the performance we can expect from these algorithms. In the first test, we provide the algorithms with as much information as they are capable of using. In the second test (only conducted on the Bottom-up algorithm), we began by providing the algorithm with as little information as possible (i.e. 0 keywords).

We believe the results are promising. The accuracies are certainly high for small inputs, and there are many accuracies over 50% for relatively challenging tasks, like inferring a 4 keyword expression from 2 keywords.

Our hope is that these accuracies would improve even more if we provided a list of results, rather than the single

best result. In practice, a user would probably be satisfied to look through 5 results if they only had to type a couple keyword to obtain them.

7. Extensions

We have considered mappings for other constructs. For some of these, we believe the most natural implementation may involve complicating the type system. For instance, adding the ability to specify constraints between parameter types, like "the return type must be the same as the first parameter."

7.1 Literals:

Numeric literals can be added by expanding the notion of a function name to include regular expressions. For instance, integer literals could become (`int`, **[-+]?[0-9]+**).

String literals can also be represented as regular expressions, which would require the string to begin and end with a quote character. Ideally, we want to expand the notion of strings to not require quotes. Both Chickenfoot and Koala support this. Koala's approach is to deal with unquoted strings as a post-processing step, and this may be the best approach in this system.

7.2 Operators:

Operators can map to functions in the natural manner, but we require multiple versions of them to support all the primitive types that use them; e.g. we require (`int`, **+**, `int`, `int`) separate from (`long`, **+**, `long`, `long`). It might seem like we could just have (`double`, **+**, `double`, `double`), since all the other numeric primitives are subtypes of `double`. However, this wouldn't allow us to add two numbers, and pass the result to a function that requires an `int`.

7.3 Assignment:

Assignment gets a little more complicated. Say we want to allow the assignment `x = y`, where `x` is an `int` and `y` is a `short`. We could add (`int`, **=**, `int`, `int`). Unfortunately, this doesn't prevent us from passing subtypes of `int` to the left-hand side of the assignment, so this wouldn't prevent `y = x`. It also wouldn't prevent `5 = x`.

One approach we have tried is adding a special set-function for each variable. In this example, we would add (`int`, **x =**, `int`). Note that the function name includes the =. This seems to work, but it does require adding lots of new functions.

A cleaner solution might involve adding more types; in particular, an "assignable" or "reference" type. So the variable `int x` would be accessible with the function (`ref:int`, **x**). Then we would have a function for integer assignment: (`int`, **=**, `ref:int`, `int`). We would also make `ref:int` a subtype of `int`, so that we could still use `x` on the right-hand-side of an assignment. This technique still requires an assignment function for each type, but not for each variable.

7.4 Array Indexing:

Array indexing (e.g. `a[i]`) is possible with functions of the form (`String`, **[]**, `String[]`, `int`). This technique requires adding a function like this for each type of array that we see. A potentially cleaner alternative would involve adding the function (t, **[]**, t`[]`, `int`), along with the machinery in the algorithm to constrain the t's to be equal.

7.5 Other:

We have thought about how we would add constructs like control flow and class declarations. One paradigm we are interested in involves supporting line-by-line completions of these constructs. For instance, the user might enter the keywords **if x == y**, and the system might suggest "`if (x == y) {`", or create a template of an `if` statement with `x == y` filled in. The function for `if` would then look like (`void`, **if**, `boolean`). Note that this function is not a function in the traditional sense of something the computer can execute; rather, it is a function that generates code for the user to work with. This is fine for our system since it is a code completer, and not an actual interpreter.

8. Related Work

This work builds on our earlier efforts to use keywords for scripting – i.e., where each command in a script program is represented by a set of keywords. This approach was used in Chickenfoot [4] and Koala [3]. The algorithms used in those systems were also capable of translating a sequence of keywords into function calls over some API, but the APIs used were very small, on the order of 20 functions. Koala's algorithm actually enumerates all the possible function trees, and then matches them to the entire input sequence (as opposed to the method used in Chickenfoot, which tries to build trees out of the input sequence). This naive approach only works when the number of possible function trees is extremely small (which was true for Chickenfoot and Koala, because they operate on web pages). Compared to Chickenfoot and Koala, the novel contribution of the current paper is the application of this technique to Java, a general purpose programming language with many more possible functions, making the algorithmic problem more difficult.

This work is also related to work on searching for examples in a large corpus of existing code. This work can be distinguished by the kind of *query* provided by the user. For example, Prospector [5] takes two Java types as input, and returns snippets of code that convert from one type to the other. Prospector is most useful when the creation of a particular type from another type is non-obvious (i.e. you can't simply pass it to the constructor, and other initialization steps may be involved). Another system, XSnippet [8], retrieves snippets based on context, e.g., all the available types from local variables. However, the query is still for a snippet of code that achieves a given type, and the intent is still for large systems where the creation of certain types is nontriv-

ial. A third approach, automatic method completion [1], uses a partially-implemented method body to search for snippets of code that could complete that method body.

The key differences between our approach and these other systems are:

1. The user's input is not restricted to a type, although it *is* constrained by types available is local context. Also, the output code may be arbitrary, not just code to obtain an object of a certain type. For instance, you could use our system to enter code on a blank line, where there is no restriction on the return type.
2. Our system uses a guided search, based on the keywords provided by the user. These keywords can match methods, variables and fields that may be used in the expression.
3. Our system generates new code, and does not require a corpus of existing code to mine for snippets. In particular, users could benefit from our system in very small projects that they are just starting.

There is also substantial work on searching for reusable code in software repositories using various kinds of queries provided by the user, including method and class signatures [6, 12], specifications [2, 13], metadata attributes [7], identifier usage [9], and documentation comments [10, 11]. These systems are aimed at the problem of identifying and selecting *components* to reuse to solve a programming problem. Our system, on the other hand, is aimed at the coding task itself, and seeks to streamline the generation of correct code that *uses* already-selected components.

9. Conclusions and Future Work

We have presented a novel technique for code-completion in Java, in which the user provides a keyword query and the system generates type-correct code that matches those keywords. We presented a formal model for the space over which the keyword search is done, and gave two efficient search algorithms. Using example queries automatically generated from a corpus of open-source software, we found that the type constraints of Java ensure that a small number of keywords is often sufficient to generate the correct method calls.

Automatically-generated queries are helpful for experimentation but may not accurately reflect how real users would query the system. We are in the process of doing an evaluation with keyword queries provided by human programmers. One important difference we have already seen is that human queries often use synonyms, e.g. **add** when the actual method name is `append()`. Users may also misspell keywords, or provide only a prefix of the necessary keyword (as if they were using conventional method name completion). We are exploring ways to extend the scoring metric to support more approximate matching, including spelling correction, synonym matching, and prefix matching.

Another improvement to the search algorithm would take into account the probability of each function, either *a priori* or conditioned on the context, when computing its score. For instance, if `Vector.remove()` is called much more often than `Vector.removeAllElements()`, then the `remove` method deserves some *a priori* bonus. Probabilities could be estimated from a large corpus of open source software, or from the user's own project.

We also plan to apply these principles to other programming languages, so that multiple-language programming becomes less arduous. Ideally, a keyword query like **add x list** should generate appropriate code to add `x` to `list`, regardless of the particular language or API in use.

The long-term goal for this work is to simplify the usability barriers of programming, such as forming the correct syntax and naming code elements precisely. Reducing these barriers will allow novice programmers to learn more easily, experts to transition between different languages and different APIs more adroitly, and all programmers to write code more productively.

Acknowledgments

This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer as part of the TParty project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Rosco Hill and Joe Rideout. Automatic Method Completion. *Proceedings of Automated Software Engineering (ASE 2004)*, pp. 228–235.

[2] J.-J. Jeng and B. H. C. Cheng. Specification Matching for Software Reuse: A Foundation. In *Proceedings of the 1995 Symposium on Software reusability*, pp. 97–105, 1995.

[3] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of CHI 2007*, to appear.

[4] Greg Little, and Robert C. Miller. Translating Keyword Commands into Executable Code. *Proceedings of User Interface Software & Technology (UIST 2006)*, pp. 135–144.

[5] David Mandelin, Lin Xu, Rastislav Bodik, Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 48–61.

[6] M. Rittri. Retrieving library identifiers via equational matching of types. *Proceedings of the tenth international conference on Automated deduction*, pp. 603–617, 1990.

[7] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.

[8] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining For Sample Code. *Proceedings of the 21st annual*

ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006), pp. 413–430.

[9] N. Tansalarak and K. T. Claypool. Finding a Needle in the Haystack: A Technique for Ranking Matches between Components. In *Proceedings of the 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005): Software Components at Work*, May 2005.

[10] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *International Symposium on Foundations of Software Engineering*q, pp. 60–68, November 2000.

[11] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pp. 513–523, May 2002.

[12] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.

[13] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.