XSnippet: Mining For Sample Code

Naiyana Sahavechaphan
National Electronics and Computer Technology Center, Thailand
naiyana.sahavechaphan@nectec.or.th

Kajal Claypool
Oracle Corporation, Nashua, NH, USA
Kajal.Claypool@oracle.com

ABSTRACT

It is common practice for software developers to use *examples* to guide development efforts. This largely unwritten, yet standard, practice of "develop by example" is often supported by examples bundled with library or framework packages, provided in textbooks, and made available for download on both official and unofficial web sites. However, the vast number of examples that are embedded in the billions of lines of already developed library and framework code are largely untapped. We have developed *XSnippet*, a context-sensitive code assistant framework that allows developers to query a sample repository for code snippets that are relevant to the programming task at hand. In particular, our work makes three primary contributions. First, a range of queries is provided to allow developers to switch between a context-independent retrieval of code snippets to various degrees of context-sensitive retrieval for object instantiation queries. Second, a novel graph-based code mining algorithm is provided to support the range of queries and enable mining within and across method boundaries. Third, an innovative context-sensitive ranking heuristic is provided that has been experimentally proven to provide better ranking for best-fit code snippets than context-independent heuristics such as shortest path and frequency. Our experimental evaluation has shown that *XSnippet* has significant potential to assist developers, and provides better coverage of tasks and better rankings for best-fit snippets than other code assistant systems.

Categories and Subject Descriptors

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

Keywords

Code Assistants, Code Mining, Ranking Code Samples, Code Reuse

1. INTRODUCTION

For sometime, reuse has been the cornerstone of modern software development. Today's programmer relies on *frameworks* and *libraries*, such as C++ libraries, Java packages, Eclipse packages, and in-house libraries [4, 1, 6, 10, 13, 8], that empower the programmer to create high quality, full-featured applications on-time. But this reliance comes at a price – these libraries and frameworks represent a steep learning curve due to the sheer number of methods, classes, and interfaces. For example, J2SE, the Java standard library, contains thousands of classes many of which have extremely complex APIs that prove to be a significant impediment to re-usability.

It is common practice for software developers to use *examples* to guide development efforts. Examples are used to enable software tasks of various shapes and sizes, such as: How to implement an Eclipse plugin that extends the JDT user interface? How to correctly use `java.io.PushBackInputStream`? How to instantiate an object of `ISelectionService`? The importance of examples is often directly proportional to the complexity of the library or framework being used and the developer's experience with it. This largely unwritten, yet standard, practice of "develop by example" is often facilitated by examples bundled with library or framework packages, provided in textbooks, and made available for download on both official and unofficial web sites. In fact, the availability of *good, reputable* examples can easily become the key factor or the key impediment to the success of a library or a framework.

However, the published examples are the proverbial tip of the iceberg, covering only a small segment of common development tasks. The bulk of the iceberg of examples exists embedded in the billions of lines of code that have already been developed! Recognizing that finding examples (as well as the importance of *examples*) is the challenge, mining techniques [8, 11, 7] have been proposed in the literature to provide code samples based on the context of code being developed by a programmer. Strathcona [8] uses different structural context descriptions (parents, invocations, and types) encapsulated in a class (code under development) to create a set of heuristics. Each heuristic is used to query the code repository, returning a set of methods and classes where the result context matches the query's context. The final result is a set of methods and classes that occur most frequently when applying all proposed heuristics. Prospector [11], on the other hand, defines a query that describes the desired code in terms of input $\mathcal{T}_{in}$ and output type $\mathcal{T}_{out}$. A query returns code snippets that instantiate an object of

WOODSTOCK '97 El Paso, Texas USA

$\mathcal{T}_{out}$ from a given input type $\mathcal{T}_{in}$. Prospector mines signature graphs generated from API specifications and jungloid graphs representing code and ranks the results by the shortest path from $\mathcal{T}_{in}$ to $\mathcal{T}_{out}$.

While these techniques take steps in the right direction, each approach is limited in the *quality* of the code snippets retrieved, resulting often in too many (often irrelevant) code snippets or in some cases too few or no code snippets. Strathcona, for example, does not specialize the heuristics it employs based on the developer's context and its results straddle the extremities – in some cases providing too many irrelevant results while in others over-constraining the context to provide too few or no results. Prospector, while performing better than Strathcona in general, has its own limitations. First, its over-reliance on API information can result in too many irrelevant results. For example, two unrelated paths can be connected by generic classes such as `Object` and `ArrayList` discounting the diversity in their semantics. Second, the context description is limited to only visible input types of fields declared in the boundary of method and class while context information such as the "parent" is ignored thereby missing a set of potentially qualified hits.

This paper presents *XSnippet*, a code assistant system that allows developers to query a sample repository for code snippets relevant to the object instantiation task at hand. Object instantiation is a common problem faced by developers. In the simplest case an object instantiation can be handled by a constructor invocation. Other more complicated ways of instantiating an object include (i) invocation of a static method, often representing a singleton implementation. For example, the static invocation `JavaUI`-*.getWorkingCopyManager()* instantiates an object of type `IWorkingCopyManager`; and (ii) invocation of a sequence of methods that eventually returns the instantiated object. For example, an `ISelectionService` object can be instantiated by the sequence *getViewSite().getWorkbenchWindow().getSelectionService()* in the class that extends `ViewPart`. In particular, *XSnippet* makes the following contributions:

- Range of Queries: *XSnippet* supports a range of object instantiation queries from specialized to generalized. This approach is informed by two hypothesis: 1) a *specialized* query provides *better-fit* code samples, that is samples that have a high degree of pertinence to the developer's code context. The more context used, the more specialized the query and the better the code samples supplied; 2) a *generalized* query provides better *coverage* in that the generalized query can find relevant code samples under a larger number of scenarios, often finding code samples where a specialized query might not.

- Ranking Heuristics: *XSnippet* supports a set of ranking heuristics from context sensitive to context independent. This is informed by two hypothesis: 1) when multiple samples are available for the same query (often the case), ranking heuristics are critical to ensure that the *best-fit* code samples appear as the top snippets; 2) context-sensitive heuristics provide better rankings for the best-fit code snippets than the context-independent heuristics. However, context-independent heuristics can be used to better rank samples within a given group of samples.

- Mining Algorithm: *XSnippet* provides an innovative *code mining* algorithm that accommodates the range of queries by constraining the mining process as needed. The algorithm, based on a graph representation of Java source code, is targeted for instantiation of types via: (i) simple constructor invocation; (ii) static method invocation; and (iii) complex sequence of methods. The algorithm can mine code snippets that exist either within the scope of an individual method or that are spread across the boundaries of two or more methods.

This paper reports on these contributions. We have developed *XSnippet* as an Eclipse plugin that can be invoked from within the Eclipse Java editor. *XSnippet* requires Eclipse 3.0 or greater, and comes with a complete set of instructions for installation. To validate our hypothesis and to evaluate the potential benefits of *XSnippet* a series of experiments have been conducted. In particular, tradeoffs have been studied, examining coverage and ranking for different types of queries ranging from generalized to specialized queries, and the usability of the system via a user study. Head-to-head comparisons have been made between *XSnippet* and the `Prospector` code assistant system [11].

Roadmap: The rest of the paper is organized as follows: Section 2 describes the program context and defines object-instantiation specific queries ranging from specialized to generalized. Section 3 provides the formal model for code snippet mining and Section 4 presents the code snippet mining for answering a user query $\mathcal{Q}$. Section 5 defines a set of ranking heuristics from context sensitive to context independent. Experimental evaluation of *XSnippet* is given in Section 6. We present related work in Section 7 and conclude in Section 8.

2. PROGRAM CONTEXT AND QUERIES

The context of a program is a rich source of semantic information that can be harnessed to provide better, more targeted *code snippets* [1] for solving a programming hurdle. Consider the code snippets shown in Figure 1. Each of these code snippets instantiate an `ICompilationUnit` object, albeit with some differences. For example, snippet A instantiates `ICompilationUnit` from an `ISelection` object, while snippet E instantiates it from a `Map` object. The relevancy of the code snippet to a developer is dependent on the lexically visible types in the developer's code. For example, if the developer's code contains an `IEditorPart` object, then snippet C is possibly the most relevant snippet as it shows how an `ICompilationUnit` object can be instantiated from an `IEditorPart` object.

In this section, we define and classify the context of a developer's code (or the *code context*), and then formally define three different types of object-instantiation specific queries based on the code context. We assume that all queries are invoked within method scope.

2.1 Types of Programming Context

A Java *source class* is defined by its: *parents* – the superclass it (the source class) inherits from, as well as the interfaces it implements; *fields* – the named attributes defined within its' scope; and *methods* – the class behavior

[1] We use the term snippets to imply a fragment of code taken from a larger source code file.

```
A. ISelection selection;
   IStructuredSelection ss = (IStructuredSelection) selection;
   Object obj = ss.getFirstElement();
   IJavaElement je = (IJavaElement) obj;
   IJavaElement ije = je.getAncestor(IJavaElement.COMPILATION_UNIT);
   ICompilationUnit cu = (ICompilationUnit) ije

B. ISelection selection;
   IStructuredSelection ss = (IStructuredSelection) selection;
   Object obj = ss.getFirstElement();
   IFile f = (IFile) obj;
   IJavaElement ije = JavaCore.create(f);
   ICompilationUnit cu = (ICompilationUnit) ije;

C. IEditorPart editor;
   IEditorInput input = editor.getEditorInput();
   IWorkingCopyManager manager = JavaUI.getWorkingCopyManager();
   ICompilationUnit cu = manager.getWorkingCopy(input);

D. JavaEditor editor;
   Object editorInput = SelectionConverter.getInput(editor);
   ICompilationUnit unit = (ICompilationUnit) editorInput;

E. Map fMap;
   IEditorInput input;
   Object obj = fMap.get(input);
   ICompilationUnit unit = (ICompilationUnit) obj;

F. JavaPlugin jp = JavaPlugin.getDefault()
   IWorkingCopyManager manager = jp.getWorkingCopyManager();
   CompilationUnitEditor editor;
   IEditorInput iei = editor.getEditorInput;
   ICompilationUnit unit = manager.getWorkingCopy(iei);
```

Figure 1: Code Snippets that Show Six Different Ways of Instantiating an `ICompilationUnit` Object.

defined at the granularity of methods. We use this basic class structure information to now define *code context* for a given method `m` contained in class `C`.

Parent Context ($\mathcal{CP}$ (`m`)): The *parent context* of a method `m`, denoted as $\mathcal{CP}$ (`m`), is a set containing the superclass extended by its containing source class C, as well as all interfaces implemented by its containing source class C. Thus, $\mathcal{CP}$ (`m`) = $\{S, I_i, I_j, \ldots, I_k\}$, where `m` is a method (`m` $\in C$ and C is the source class), S the superclass of C, and I_i, I_j ... I_k the interfaces implemented by C. The parent context $\mathcal{CP}$ is a *global* context in that it is shared by all methods $\mathtt{m}_i \in C$.

Type Context ($\mathcal{CT}$ (`m`)): The *type context*, denoted as $\mathcal{CT}$ (`m`), is the set of the types of all inherited and local fields, as well as all lexically visible types in the scope of a method `m`. Thus, $\mathcal{CT}$ (`m`) = $\{\{t_s\},\{t_f\},\{t_l\}\}$, where `m` is a method `m` $\in C$, $\{t_s\}$ is the set of types for inherited fields, $\{t_f\}$ is the set of types for all local fields, and $\{t_l\}$ is the set of all lexically visible types within the scope of the method. The sets $\{t_s\}$ and $\{t_f\}$ define a *global* context that is shared by all methods $\mathtt{m}_i \in C$, while $\{t_l\}$ defines a *local* context particular to the method `m`.

Consider the method `m` = *getCompilationUnit()* in the `JavaMetricsView` source class shown in Figure 2. Here, the parent context $\mathcal{CP}$ (`m`) = {`ViewPart`, `ISelectionListener`, `IJavaMetricsListener`}, and the type context $\mathcal{CT}$ (`m`) = {`IStructuredSelection`, `IJavaElement`, `Text`} ∪ all inherited fields in the `ViewPart`.

2.2 Snippet Queries

To support procurement of code snippets for the instantiation of a type t_q from within a method `m`, we define a set of queries that range from generalized queries resulting in all possible code snippets that instantiate t_q to specialized context-based queries that result in more specific code snippets for instantiating t_q.

```
public class JavaMetricsView extends ViewPart
  implements ISelectionListener, IJavaMetricsListener {

  Text message;

  private ICompilationUnit getCompilationUnit
    (IStructuredSelection ss) {
    if (ss.getFirstElement() instanceof IJavaElement) {
      IJavaElement je = (IJavaElement) ss.getFirstElement();

      ... a developer is currently at this position ..

      return (ICompilationUnit) je.getAncestor
        (IJavaElement.COMPILATION_UNIT);
    }
  }
}
```

Figure 2: The `JavaMetricsView` Class showing the Method *getCompilationUnit()*.

Generalized Instantiation Query $\mathcal{IQ}_G$. A generalized instantiation query $\mathcal{IQ}_G$ takes as input a type t_q and returns the set of all snippets s, contained in the sample code repository, that instantiate the type t_q. The query $\mathcal{IQ}_G$ is given in Equation 1.

$$\mathcal{IQ}_G(t_q) = \forall s \in \mathcal{S} : s \ \ contains \ \ t_q \ \ instantiation \quad (1)$$

A generalized instantiation query $\mathcal{IQ}_G$ is useful when (i) the type t_q can be instantiated independent of the code context, for instance via a static method invocation. For example, an `IWorkingCopyManager` object is often instantiated by the invocation of the static method *getWorking-*

CopyManager() of type `JavaUI` irrespective of whether or not the class `JavaUI` exists in the context; and (ii) there are no code snippets available that meet the requirements of the code context. In such cases, any code snippet that shows the instantiation of the type t_q is considered relevant. In general, $\mathcal{IQ}_G$ returns all code snippets that instantiate t_q. Consider the query $\mathcal{IQ}_G$ (`ICompilationUnit`). This query returns all code snippets in the repository, a sampling of which is shown in Figure 1.

A Type-Based Instantiation Query $\mathcal{IQ}_T$. A type-based instantiation query $\mathcal{IQ}_T$ takes as input a type t_q and the type context $\mathcal{CT}$ (`m`), where `m` represents the method in which the query is invoked, and returns the set of all snippets s, contained in the sample code repository, such that the type t_q is instantiated from some type $t_c \in \mathcal{CT}$ (`m`). The query $\mathcal{IQ}_T$ can be defined in terms of $\mathcal{IQ}_G$ as given in Equation 2.

$$\mathcal{IQ}_T(t_q, \mathcal{CT}(\texttt{m})) = \exists s \in \mathcal{IQ}_G(t_q) : \mathcal{T}(s) \cap \mathcal{CT}(\texttt{m}) \quad (2)$$

Here, s denotes a snippet, $\mathcal{T}(s)$ the lexically visible types in the code snippet s and $\mathcal{CT}$ (`m`) denotes the type context of the method `m`.

A type-based instantiation query $\mathcal{IQ}_T$ is useful when the instantiation of a type t_q is required from some visible type t_b in the type context. Consider the code segment shown in Figure 3. To highlight any selected `.java` source in the Eclipse package explorer, the method *selectionChanged()* requires the instantiation of an `ICompilationUnit` object from an object of the type `ISelection`. Here, $\mathcal{CT}$ (*selectionChanged()*) = {`ISelection`, `IWorkbenchPart`, `Text`, `JavaMetrics`} ∪ all inherited fields from `ViewPart`. Based on the code snippets shown in Figure 1, the query $\mathcal{IQ}_T$ (`ICompilationUnit`, $\mathcal{CT}$ (*selectionChanged()*)) will return only code snippets A and B.

```
public class JavaMetricsView extends ViewPart
  implements ISelectionListener, IJavaMetricsListener {

  Text message;
  JavaMetrics jm;

  public void createPartControl(Composite parent) {
    parent.setLayout(new FillLayout());
    message = new Text(parent, SWT.MULTI);
    message.setText(NO_SELECTION_MESSAGE);
    getViewSite().getWorkbenchWindow().getSelectionService().
      addSelectionListener(this);
    jm = new JavaMetrics();
    jm.addListener(this);
  }

  public void setFocus() {
  }

  public void selectionChanged
    (IWorkbenchPart part, ISelection selection) {
    ICompilationUnit cu;
    }
  }
}
```

Figure 3: Code Segment of `JavaMetricsView` Class showing the Object Instantiation of `ICompilationUnit`.

Parent-Based Instantiation Query $\mathcal{IQ}_P$. A well-defined type in a library typically has it own unique functionalities with some dependencies on its parents (direct and indirect superclasses as well as interfaces). It is our observance that a set of types with the same or similar parents tend to have more relevant code snippets than types that do not share a parent. To reflect the relevance of the shared parents (superclass and/or interfaces), a parent-based instantiation query $\mathcal{IQ}_P$ is defined as follows. A parent-based instantiation query $\mathcal{IQ}_P$ takes as input a type t_q and the parent context $\mathcal{CP}$ (`m`), where `m` $\in C$ represents the method in which the query is invoked, and returns the set of all snippets s, contained in the sample code repository, such that the containing class of the snippet C_s and the class C either inherit from the same class or implement the same interfaces.

$$\mathcal{IQ}_P(t_q, \mathcal{CP}(\texttt{m})) = \exists s \in \mathcal{IQ}_G(t_q) : \mathcal{CP}(s) \cap \mathcal{CP}(\texttt{m}) \quad (3)$$

Here, s denotes a snippet, $\mathcal{CP}$ (s) the parent context of the snippet, $\mathcal{CP}$ (`m`) the parent context of the method `m`.

Consider the code segment in Figure 4. Here, an `ICompilationUnit` object in the method *run()* must capture a `.java` source file in the Java editor and add some actions to it. Here, $\mathcal{IQ}_P$ (`ICompilationUnit`, $\mathcal{CT}$ (*run()*) = {`IAction`} ∪ inherited fields from `AddTraceStatementAction`) does not provide any code snippets that instantiate `ICompilationUnit` from `IAction` and other types. However, `ICompilationUnit` can be instantiated based on the parent interface `IEditorActionDelegate`. Thus, while the type-based instantiation query does not provide any results, a parent-based instantiation query $\mathcal{IQ}_P$ (`ICompilationUnit`, $\mathcal{CP}$ (*run()*) = {`IEditorActionDelegate`}) returns the codes snippets C and D shown in Figure 1.

```
public class AddTraceStatementsEditorAction
  extends AddTraceStatementsAction
  implements IEditorActionDelegate {

  public void run (IAction action) {
    ICompilationUnit cu;
  }

  public void setActiveEditor
    (IAction action, IEditorPart targetEditor) {
  }
}
```

Figure 4: Code Segment of `AddTraceStatementsEditorAction` Class showing Object Instantiation of `ICompilationUnit`.

3. THE SOURCE CODE MODEL

While text-based source code is the right medium for developers writing code, the textual format itself has limited applicability for analysis, and for our purposes of mining code snippets. Consider the sequence of method invocations *getViewPart().getWorkbenchWindow().getSelectionService().addSelectionListener(this)* in the method *createPartControl()* of the `JavaMetricsView` class shown in Figure 3. Here, an object of the type `ISelectionService` is instantiated during the invocation of the method *getSelectionService()*. However, the type `ISelectionService` is itself not explicitly denoted in this source code, thereby rendering text-based mining insufficient for answering simple queries such as "how can we instantiate an object of type `ISelectionService`".

In this section, we define the *source code model* ($\mathcal{CM}$) as a directed acyclic graph $\mathcal{CM} = (\mathcal{N}_{CM}, \mathcal{E}_{CM})$. An instance of the source code model (or simply a CM instance), cm_s, corresponds to a source class $\mathcal{C}_s$ defined in a `.java` or `.class` file. The different types of nodes $\mathcal{N}_{CM}$ and edges $\mathcal{E}_{CM}$ defined in $\mathcal{CM}$ in concert capture the structure and behavior of a class, and provide a formal model for the code snippet mining described in later sections.

Java source code encapsulates (i) the *class structure*: represented by the class inheritance hierarchy, the implemented interface hierarchy, and the class's set of members (fields and methods); and (ii) the *class behavior*: represented by the computations specified at the granularity of individual methods. The source code model captures all aspects of the class structure. For class behavior, however, it captures the signature, field declarations, return types, method invocations, and assignment statements. All control statements are ignored.

In this section, we describe the different types of nodes ($\mathcal{N}_{CM}$) and edges ($\mathcal{E}_{CM}$) that comprise the source code model $\mathcal{CM}$, and show by example the construction of a source code model instance (cm_s) of a Java source class. Full details of the transformation algorithm that translates a $\mathcal{C}_s$ to a corresponding instance cm_s can be found at [18].

3.1 Nodes in Source Code Model

A node in the *source code* model $\mathcal{CM}$ can be categorized as *type*, *object* or *method* node. Figure 5 gives the graphic representation of these different node types.

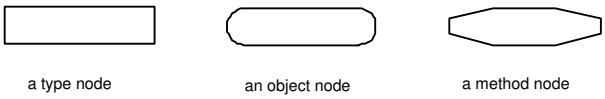

Figure 5: Graphic Representation of the Different Types of Nodes in the Source Code Model $\mathcal{CM}$.

A *type* node can represent (i) a *source class* - the class $\mathcal{C}_s$ for the corresponding source code model instance cm_s; (ii) a *superclass* - the direct superclass $\mathcal{C}_{sp}$ of a specified source class $\mathcal{C}_s$; (iii) an *interface* - the interface $i_s \in \mathcal{I}_s$ implemented by the specified source class $\mathcal{C}_s$; or (iv) a *generic class* - a class $\mathcal{C}_g$ whose static field or method member (including constructor) is referred to or invoked within the specified source class $\mathcal{C}_s$. A *type* node is labeled with its *domain type*[2] where the domain type can be either a class or an interface.

An *object* node can represent (i) a *class field* - a field $f_s \in \mathcal{F}_s$ declared within the scope of the source class $\mathcal{C}_s$; or (ii) a *method field* - a field f_m declared within the scope of a method m_s where the method m_s is contained in the source class $\mathcal{C}_s$. An *object* node is labeled with its *domain type* – a class, an interface or `void`, and its *name* – a label unique within the scope of the method m_s or the source class $\mathcal{C}_s$. An exception to this is the name "*this*" used to denote a reference to the source class $\mathcal{C}_s$.

A *method* node encapsulates the signature as well as the behavior of a method, and is labeled with the method's name and the method's modifier (public, private or protected) using UML notation. The signature of the method is defined by the method's return type and an ordered list of the method's parameter types, both of which are represented by *type* and *object* nodes as previously described. The method's behavior is specified by a set of computations, for example variable declarations and method invocations, that are represented by a source code model sub-instance.

[2]We use the full path to represent the domain type. However, for brevity we use only the class name in this paper.

3.2 Edges in Source Code Model

An edge in the *source code* model $\mathcal{CM}$ can be categorized as *inheritance*, *implement*, *composite*, *method*, *assignment* or *parameter* edge. The *inheritance*, *implement* and *composite* edges together capture the class structure, while the *method*, *assignment* and *parameter* edges represent the class behavior.

An *inheritance* edge represents the relationship between the source class $\mathcal{C}_s$ and its direct superclass $\mathcal{C}_{sp}$. An *implement* edge relates the source class $\mathcal{C}_s$ to its implemented interface $i_s \in \mathcal{I}_s$. A *composite* edge denotes the relationship between the source class $\mathcal{C}_s$ and its declared class field $f_s \in \mathcal{F}_s$ or method $m_s \in \mathcal{M}_s$. Figure 6(b) depicts the class structure for the source code model instance cm_s that is representative of the `JavaMetricsView` source class (Figure 6(a)).

A *method* edge represents the invocation of a method, either a static or a non-static method. The edge is outgoing from the *type* or *object* that represents the type or object handle for the given method m_i, and is incident on the *object* returned by the method m_i. The method edge is labeled with the *method name* to specify the method m_i that is invoked. A special case of the method edge is the object instantiation via the `new` construct. The object instantiation in this case is treated as a static method invocation and is represented by a method edge labeled "*new*" from the input *type* to an output *object*. Figure 7(a)-(b) shows the transformation of method invocation and constructor statements respectively to nodes and method edges in a cm_s.

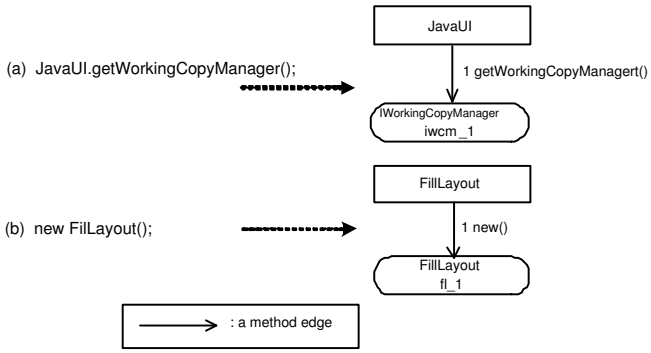

Figure 7: The Transformation of Method Invocation Statements to Nodes and Method Edges.

An *assignment* edge represents the assignment of a value (or address). An assignment statement can be (a) *equivalent* where the two variables have the same domain type; (b) *downcast* where the domain type of the output variable v_o is more specific than that of the input variable v_i; and (c) *upcast* (opposite of a downcast) where the domain type of the output variable v_o is more general than that of input variable v_i. Figure 8(a)-(c) show the mapping of the three different assignments respectively.

A *parameter* edge represents the participation of a node

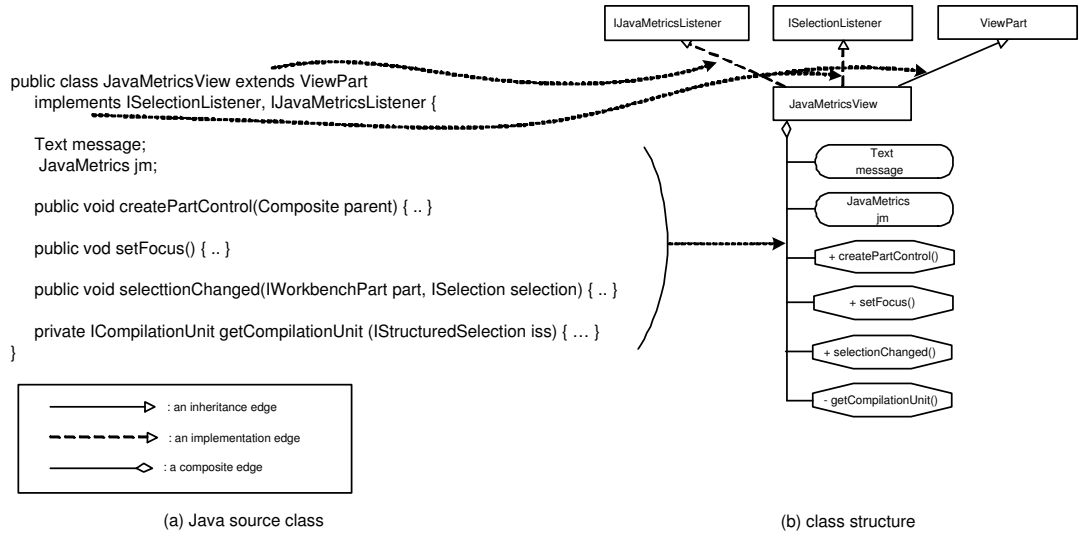

Figure 6: Mapping from Java Source File to Nodes and Structural Edges in cm_s.

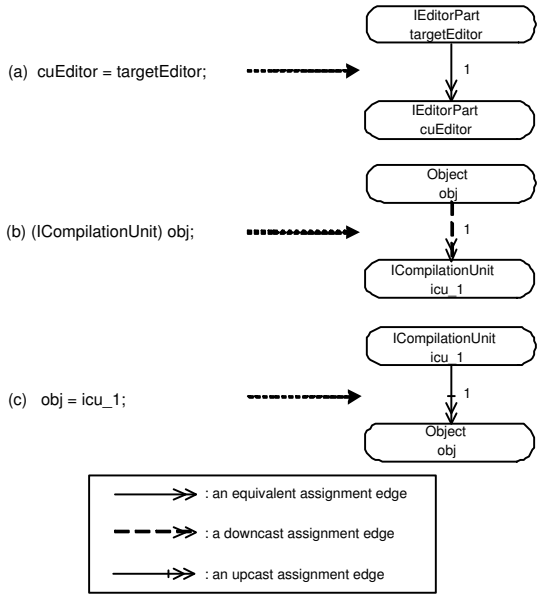

Figure 8: The Mapping of Assignment Statements to Nodes and Assignment Edges in a cm_s.

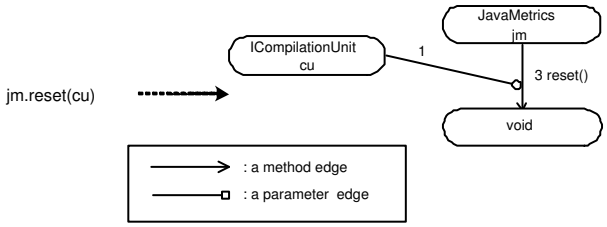

Figure 9: Transformation of the Statement "jm.reset(cu") to Nodes and, Method and Parameter Edges.

n_p as a parameter for a method m_i invoked either in the scope of a method m_s or in a source code model instance cm_s. This edge is unique in that the edge is incident on a method edge that represents the invocation of the method m_i that requires this parameter node n_p. Figure 9 shows the mapping of code "jm.reset(cu)" to nodes and method and parameter edges in a cm_s.

The *method*, *assignment* and *parameter* edges are annotated with a *local order*. For the method and assignment edges, the local order is a monotonically increasing number within the scope of a method m_s and indicates the sequence of computations. For the parameter edges, the local order number is scoped within the method invocation to indicate the order of the parameter in the case of multiple parameters.

4. SNIPPET MINING

The goal of the *snippet mining* is to mine from a given code sample repository all code snippets that satisfy a given user query $\mathcal{Q}$. Figure 10 outlines the high level steps undertaken by the snippet mining process. Given the possibly huge number of code samples in the repository, the *SelectionAgent* pre-selects a set of code model instances cm_i from the code repository based on the user query $\mathcal{Q}$. This pre-selection is based in part on the type of the user query ($\mathcal{IQ}_G$, $\mathcal{IQ}_P$, $\mathcal{IQ}_T$), and in part on a B+ tree index defined on all types declared or referred to in the code sample repository. For $\mathcal{IQ}_G$ and $\mathcal{IQ}_T$ queries all code model instances cm_i that have a reference to the query type t_q are pre-selected using the B+ tree index. For $\mathcal{IQ}_P$ queries, the pre-selected set of code model instances cm_i is limited to those that refer the query type t_q**and** that match the parent context (implement the same interface(s) or extend the same superclass) of the query $\mathcal{Q}$.

The pre-selected code instance set CM is passed to the *MiningAgent* that controls the overall mining process. The

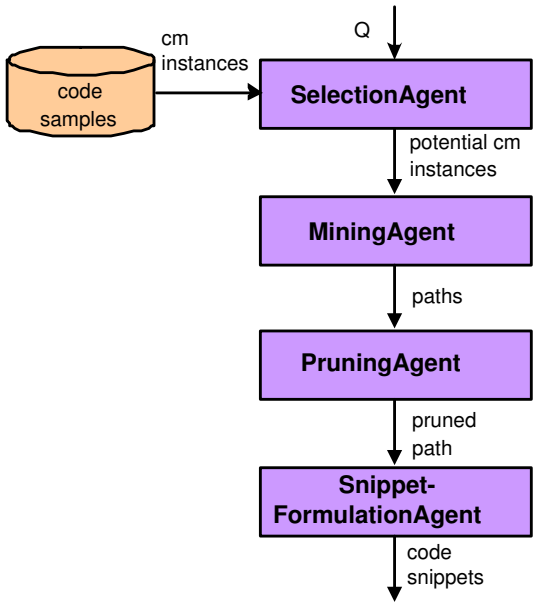

Figure 10: A High-Level View of the Snippet Mining Process.

MiningAgent invokes the BFSMINE algorithm for every code model instance $cm_i \in CM$. The BFSMINE algorithm, a breadth-first mining algorithm, is the crux of our approach. It (BFSMINE algorithm) traverses a code model instance cm_i and produces as output a set of paths P that represent the final code snippets returned to the user. On completion of the BFSMINE phase, the *MiningAgent* passes the collection of the sets of paths $\mathcal{P}$ generated by multiple invocations of the BFSMINE algorithm, to the *PruningAgent*.

The *PruningAgent* removes (i) all duplicate paths $\mathtt{p} \in \mathcal{P}$ without modifying their pre-pruning count; (ii) no-op paths, that is paths that do not encapsulate meaningful functionality. No-op paths include initialization statements such as `ICompilationUnit cu = null`; and (iii) non-compilable and non-executable paths, that is paths that can not be verified for correctness to ensure that they are compilable and executable fragments of code. The *pruned* mining paths are passed to the *SnippetFormulationAgent* that transforms each path $\mathtt{p} \in \mathcal{P}$ to a corresponding Java code snippet based on the transformation rules outlined in Section 3.

The BFSMINE algorithm is a critical component of our approach. We detail this algorithm with examples in the following subsections.

4.1 BFSMINE Algorithm

Figure 11 gives the pseudo-code for the BFSMINE algorithm. Given a user query $\mathcal{Q} = (t_q, \mathcal{CT})$, where t_q is the query type and $\mathcal{CT}$ the code context in which the query is invoked, and a specific code instance cm_i, the BFSMINE algorithm initiates the mining process by identifying the set of nodes $\mathcal{NQ} = \{\mathtt{n_q}\}$ such that $\mathtt{n_q} \in cm_i$ and the domain of the node $\mathtt{n_q}$ is t_q (domain($\mathtt{n_q}$) = t_q). The set of nodes $\mathcal{NQ}$ identifies all instances in the code model instance cm_i where the query type t_q is instantiated. In most cases, the node $\mathtt{n_q} \in \mathcal{NQ}$ is scoped within a method instance $\mathtt{m}_i \in cm_i$. There may be multiple nodes $\mathtt{n_q}$ within the scope of the same method instance $\mathtt{m}_i$. Unless stated otherwise, the BFSMINE algorithm is described in the context of one method instance, cm_m.

```
private MP BFSMINE (Type t_q, Context CT, CM cm_i) {
    MP allMPs;
    for each method m_i ∈ cm_i.getMethods(t_q) {
        for each node n_q ∈ m_i.getNodes(t_q) {
            MP mps = mineIQ(cm_i, m_i, n_q, CT);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}
private MP mineIQ(CM cm_i, Method m_i, Node n_o, Context CT) {
    if n_i.getDomainType() ∈ CT
        return null;
    MP allMPs;
    for each n_i ∈ n_o.getInputNodes() {
        Edge e = n_i.getAnEdgeTo(n_o);
        MP mps = createAPath(n_i, e, n_o);

        MP nextPaths = mineIQ(cm_i, m_i, n_i, CT);
        if nextPaths == null {
            allMPs.addPaths(mps);
        else {
            mps.connectPaths(nextPaths);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}
```

Figure 11: The BFSMINE algorithm.

The goal of the BFSMINE algorithm is to determine for all such instances $\mathtt{n_q}$, types and eventually code segments that instantiate the node $\mathtt{n_q}$ and hence the query type t_q. For each node $\mathtt{n_q} \in \mathcal{NQ}$, the BFSMINE algorithm recursively traces back along all edges e_i,...,e_j incident on the node $\mathtt{n_q}$ [3]. The back traversal terminates when it reaches either (i) node $\mathtt{n}_i \in cm_m$, such that there are no edges $\mathtt{e}_i$ incident on it; or (ii) node $\mathtt{n}_i \in cm_m$, such that the domain of the node $\mathtt{n}_i$ is $\mathtt{t}_i$ (domain($\mathtt{n}_i$) = $\mathtt{t}_i$) and $\mathtt{t}_i$ is a type defined in the context $\mathcal{CT}$ of the user query Q ($\mathtt{t}_i \in \mathcal{CT}$).

Each traversal from $\mathtt{n_q}$ to node $\mathtt{n}_i$ is formulated into a *mining path* $\mathtt{p}$ such that each path records a forward trace from the termination node $\mathtt{n}_i$ to the starting node $\mathtt{n_q}$.

The BFSMINE algorithm terminates when all nodes $\mathtt{n_q} \in \mathcal{NQ}$ have been traced back, and returns the set of paths $\mathcal{P}$ to the *MiningAgent*.

EXAMPLE 1. *To illustrate the working of BFSMINE, consider that the BFSMINE algorithm is invoked with the instantiation query* $\mathcal{IQ}$ = (t_q =`ICompilationUnit`, $\mathcal{CT}$ = { }), *and the code model instance corresponding to class* `JavaMetricsView` *shown in Figure 12. In the first phase of the BFSMINE algorithm, we identify the set of nodes* $\mathcal{NQ}$ = {$\mathtt{n_q}$ } *such that domain*($\mathtt{n_q}$) = `ICompilationUnit`. *In this example,* $\mathcal{NQ}$ = {`ICompilationUnit icu`$_1$ }, *where* $\mathtt{n_q}$ = `ICompilationUnit icu`$_1$ *exists in the code model instance of the method* `getCompilationUnit()` *defined in the* `JavaMetricsView` *class instance.*

The back traversal phase of the BFSMINE algorithm initiates at the node $\mathtt{n_q}$ = `ICompilationUnit icu`$_1$. *As* `ICompilationUnit icu`$_1$ *has two edges incident on it, the BFSMINE algorithm traces back along the edges to the nodes* `IJavaElement ije`$_1$ *and* `IJavaElement ije`$_2$. *At the node*

[3] Note, this does not include the parameter edge that is incident on a method invocation edge.

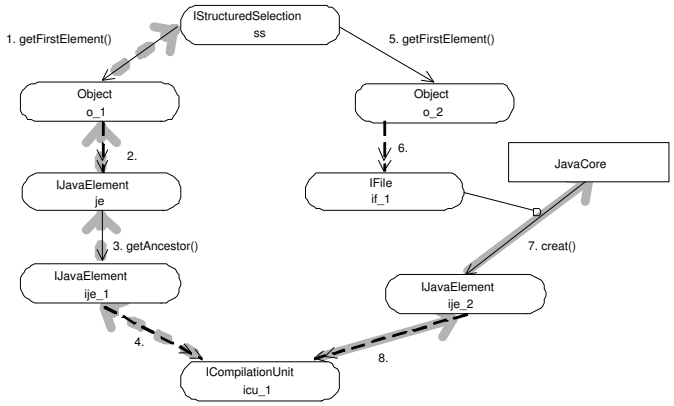

Figure 12: Back Traversal for Query $\mathcal{IQ}$ = (t_q = ICompilationUnit, $\mathcal{CT}$ = { }). The code model instance of method `getCompilationUnit()` in class `JavaMetricsView` is shown. The example highlights the *basic* back traversal of the BFSMINE algorithm.

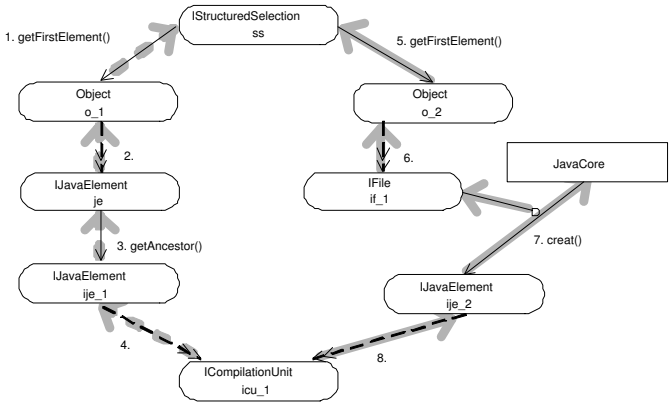

Figure 13: Back Traversal for Query $\mathcal{IQ}$ = (t_q =ICompilationUnit, $\mathcal{CT}$ = { IFile}). The code model instance of method `getCompilationUnit()` in class `JavaMetricsView` is shown. The example highlights the back traversal of all edges.

`IJavaElement ije`$_1$*, the back traversal is applied recursively till it terminates at the node* `IStructuredSelection ss`*. Similarly, the recursive traceback from the node* `IJavaElement ije`$_2$ *terminates at the node* `JavaCore`*. As all paths have been traversed, the traversal phase of the algorithm terminates. The BFSMINE algorithm terminates with the completion of the traversal phase, as there are no other nodes* $\mathtt{n_q} \in \mathcal{NQ}$.

The back traversal is shown in Figure 13 using broad stroked lines. On termination the BFSMINE algorithm outputs two paths: <ss, o$_1$, je, ije$_1$, icu$_1$>*, and* <JavaCore, ije$_2$, icu$_1$>.

*In the above example, the back traversal terminates when nodes with no edges incident on them are reached. The traversal can also terminate if a type specified in the context $\mathcal{CT}$ of the query is reached. Consider the modified query $\mathcal{IQ}$ = (t_q =ICompilationUnit, $\mathcal{CT}$ = {*IJavaElement*}). The query $\mathcal{IQ}$ now has a specified type context $\mathcal{CT}$ = {*IJavaElement*}. The traceback of the paths, in this case, terminates at* `IJavaElement ije`$_1$ *and* `IJavaElement ije`$_2$ *respectively resulting in the final mining paths:* <ije$_1$,icu$_1$>*, and* <ije$_2$,icu$_1$>.

4.2 Extensions to the BFSMINE Algorithm

Consider the example detailed in Figure 12. The BFSMINE algorithm returns two paths: p$_1$ = <ss, o$_1$, je, ije$_1$, icu$_1$> and p$_2$ = <JavaCore, ije$_2$, icu$_1$>. While valid, these paths are *partial* and raise additional questions for the user: How is the object ss obtained? How are parameters of a specific method within the snippet instantiated? The user, in these cases, would likely need to initiate new queries that search the sample repository for possible snippets.

We observe that constraining the traversal phase of the algorithm to (i) edges incident on a node and (ii) within the scope of method boundaries, limits the BFSMINE algorithm to produce only partial snippets in many cases. We now propose extensions to the BFSMINE algorithm to facilitate parameter edge mining as well as across method boundary mining.

Mining Parameter Edges. Traversals along a parameter edge e_{pi} incident on a method edge in the traceback path (as defined in the BFSMINE algorithm) are handled similar to all other types of edges. A back traversal along a parameter edge thus launches a recursive traceback at its initiating parameter node n_p. The key distinction with respect to parameter edges is in the mining paths. While the path itself is constructed as described for the BFSMINE algorithm, each parameter edge traversal represents a *sub-path* defined in the context of the primary path. The *sub-paths* constructed by the parameter edge traversals are appended to the primary path.

The traceback phase and the algorithm terminates as described for BFSMINE algorithm.

EXAMPLE 2. *Consider that the BFSMINE-EXT algorithm (see Figure 14) is invoked with the instantiation query $\mathcal{IQ}$ = (t_q =ICompilationUnit, $\mathcal{CT}$ = {}), and the code model instance corresponding to class* `JavaMetricsView` *shown in Figure 13. The BFSMINE-EXT algorithm traces back over all edges, including the parameter edge* e$_{p1}$ *incident on the method edge* `create ()`*. Traceback along the parameter edge* e$_{p1}$ *results in the sub-path* <ss,o$_2$,if$_1$> *that is appended to the primary traceback path* <JavaCore, ije$_2$, icu$_1$>*. Thus, on completion the BFSMINE-EXT algorithm produces two paths:* p$_1$ = <ss, o$_1$, je, ije$_1$, icu$_1$>*, and* p$_2$ = {<JavaCore, ije$_2$, icu$_1$>, <ss,o$_2$,if$_1$>}

Mining Across Method Boundaries. Traversals across method boundaries are enabled under two conditions.
Case 1: There exists a method parameter node n_{pm} in the code instance of the method $\mathtt{m}_i$ such that n_{pm} represents the parameter object used in the invocation of the method $\mathtt{m}_i$. The method $\mathtt{m}_i$ is assumed to be invoked from within a method $\mathtt{m}_j$. In this case, the traversal of the BFSMINE algorithm is extended to traceback the instantiation of the method parameter node at the invocation point of the method $\mathtt{m}_i$ in method $\mathtt{m}_j$. The method parameter node n_{pm} here provides a "hook" point for tracing back to a hitherto **un-traced** method.

```
private MP BFSMINE-EXT (Type t_q, Context CT, CM cm_i) {
    MP allMPs;
    for each method m_i ∈ cm_i.getMethods(t_q) {
        for each node n_q ∈ m_i.getNodes(t_q) {
            MP mps = mineIQ(cm_i, m_i, n_q, CT);
            allMPs.addPaths(mps);
        } }
    return allMPs;
}
private MP mineIQ(CM cm_i, Method m_i, Node n_o, Context CT) {
    if n_i.getDomainType() ∈ CT
        return null;
    MP allMPs;
    for each n_i ∈ n_o.getInputNodes() {
        Edge e = n_i.getAnEdgeTo(n_o);
        MP mps;
        if isLocalMethodCall(cm_i, n_i, e)
            mps = mineCalledMethod(cm_i, n_i, n_o, e, CT);
        else
            mps = createAPath(n_i, e, n_o);

        if e.getParameters() != null {
            MP paraPaths = mineParameter(cm, m, n_i, e, CT);
            mps.connectPaths(paraPaths);
        }

        if n_i.isMethodParaNode(m_i) {
            MP callerPaths = mineCallerMethod
                        (cm, m, n_i, CT);
            mps.connectPaths(callerPaths);
        }

        MP nextPaths = mineIQ(cm_i, m_i, n_i, CT);
        if nextPaths == null {
            allMPs.addPaths(mps);
        else {
            mps.connectPaths(nextPaths);
            allMPs.addPaths(mps);
        } }
    return allMPs;
}
private MP mineCalledMethod(CM cm_i, Node n_i, Node n_o, Edge e,
                            Context CT) {
    metTypem_called = getALocalMethod(cm_i, e);
    MP allMPs;
    for each node n ∈ m_called.getNodes(n_o.getDomainType()) {
            MP mps = mineIQ(cm_i, m_called, n, CT);
            allMPs.addPaths(mps);
    }
    return allMPs;
}
private MP mineParameter(CM cm, metTypem, Node n, Edge e,
            Context CT) {
    MP allMPs;
    for each parameter node n_p ∈ e.getParameters() {
        MP mps = mineIQ(cm, m, n_p, CT);
        allMPs.connectPaths(mps);
    }
    return allMPs;
}
private MP mineCallerMethod(CM cm, metTypem, Node n,
            Context CT) {
    MP allMPs;
    metTypem_caller = cm.getACallerMethod(m);
    for each node n ∈ m_caller.getNodes(n.getDomainType()) {
        MP mps = mineIQ(cm, m_caller, n, CT);
        allMPS.addPaths(mps);
    }
    return allMPs;
}
```

Figure 14: The BFSMINE-EXT algorithm.

The termination condition for the traversal and the mining path construction is as described for the BFSMINE algorithm.

Case 2: The method edge e_m in the code model instance of the method $\mathtt{m}_i$ encapsulates the invocation of a locally defined method $\mathtt{m}_j$, such that method $\mathtt{m}_j$ returns an object $\mathtt{O}_j$ of interest during traversal. The BFSMINE algorithm is automatically invoked for all such local methods. To avoid infinite recursion, this automatic invocation of the BFSMINE algorithm is restricted to hitherto **un-traced** local methods $\mathtt{m}_j$.

EXAMPLE 3. *Consider the query* $\mathcal{IQ} = (t_q =$ `ICompilationUnit`, $\mathcal{CT} = \{\})$. *Figure 15 shows the code model instance for the* `getCompilationUnit()` *and the* `selectionChanged()` *methods. With the BFSMINE-EXT algorithm, the traversal phase initiates at nodes* `ICompiationUnit` $\mathtt{icu}_1$ *and* `ICompilationUnit` $\mathtt{icu}_2$ *and does not terminate at the* `IStructuredSelection is`, *but instead traces back through the invocation point of the method* `getCompilationUnit()` *in the method* `selectionChanged()` *to the instantiation of the* `getCompilationUnit()`*'s parameter* `IStructuredSelection iss`. *The traversal terminates at the node* `ISelection selection` *in the* `selectionChanged()` *method. The back traversal is schematically depicted in Figure 15 using broad stroked lines. On completion, the BFSMINE-EXT algorithm results in two* more complete *paths:* $\mathtt{p}_1 =$ <`selection`, $\mathtt{iss}_1$, `ss`, $\mathtt{o}_1$, `je`, $\mathtt{ije}_1$, $\mathtt{icu}_1$>, *and* $\mathtt{p}_2 =$ {<`JavaCore`, $\mathtt{ije}_2$, $\mathtt{icu}_1$>, <`selection`, $\mathtt{iss}_1$, `ss`, $\mathtt{o}_2$, $\mathtt{if}_1$>}.

Figure 16 illustrates the traceback when the second cross method boundary traversal condition is invoked. Here, we assume that the BFSMINE-EXT algorithm initiates at node `ICompilationUnit cu` *in method* `selectionChanged()`. *The method edge incident on the node* `ICompilationUnit cu` *encapsulates the invocation of a local method* `getCompilationUnit()`. *In this case, the code model instance of the method* `getCompilationUnit()` *is automatically mined to provide more complete code snippets. On termination, the BFSMINE-EXT algorithm produces two mining paths:* $\mathtt{p}_1 =$ <`selection`, $\mathtt{iss}_1$, `ss`, $\mathtt{o}_1$, `je`, $\mathtt{ije}_1$, $\mathtt{icu}_1$>, *and* $\mathtt{p}_2 =$ {<`JavaCore`, $\mathtt{ije}_2$, $\mathtt{icu}_1$>, <`selection`, $\mathtt{iss}_1$, `ss`, $\mathtt{o}_2$, $\mathtt{if}_1$>}.

5. FIRST IS THE BEST: RANKING THE CODE SNIPPETS

While the *snippet mining* process returns a set of code snippets all of which satisfying a given user query $\mathcal{Q}$, the fit of these code snippets in solving a particular programming task may, however, vary. On average, a user can only be expected to scan the first ten or so code snippets returned by any search or mining process [3]. To best assist developers it is critical that snippets with the potential to be *best-fits* be ranked within the first 10 or so results. In this section, we present three distinct heuristics, *length*, *frequency* and *context*, to rank the set of code snippets returned by the snippet mining process.

5.1 Ranking by Snippet Length

The *length* heuristic ranks snippets by the number of lines of code encapsulated in the code snippets – the lower the number of lines of code, the higher the rank of the code snippet. This length heuristic is similar to the shortest path heuristic applied by Prospector [11]. Figure 17(a) shows the ranking of the code snippets obtained for $\mathcal{Q} = (t_q =$ `ICompilationUnit`, $\mathcal{CT} = \{\})$ using the length heuristic. Here code snippets D and E are ranked above code snippets A, B, C and F as they have fewer lines of code.

5.2 Ranking by Frequency of Occurrence

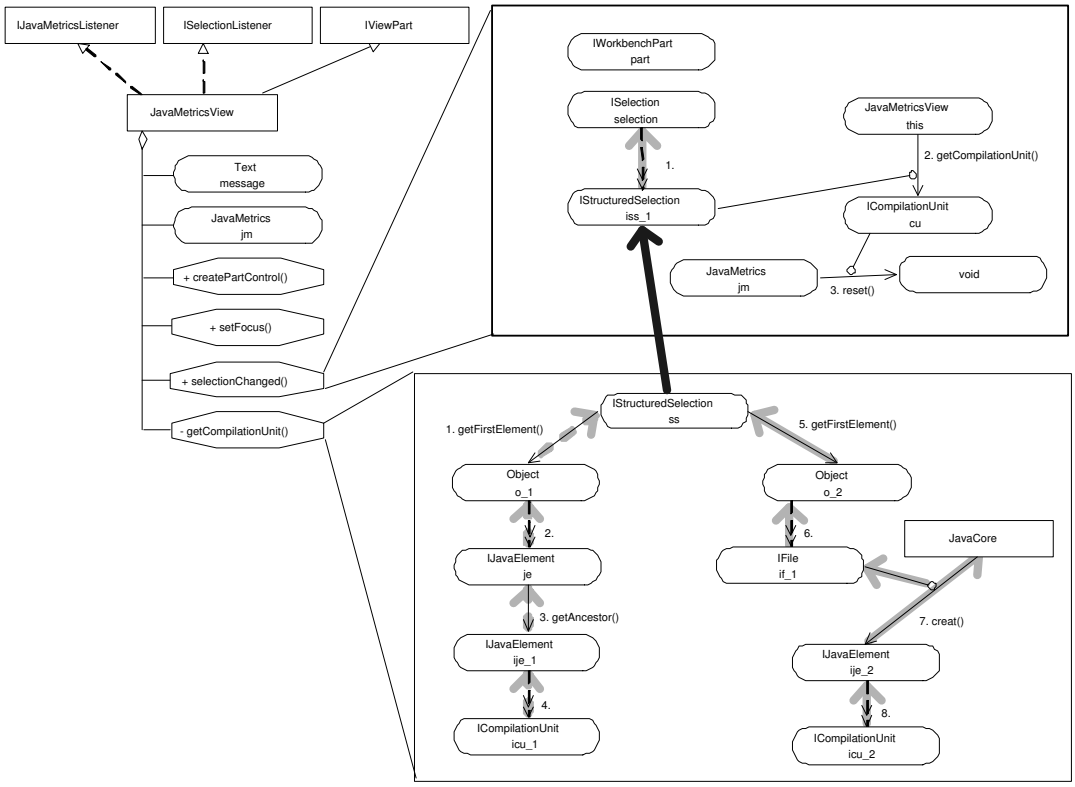

Figure 15: Back Traversal for Query $\mathcal{IQ}$ = (t_q =`ICompilationUnit`, $\mathcal{CT}$ = { ISelection }). The code model instances of methods `getCompilationUnit()` and `selectionChanged()` in class `JavaMetricsView` are shown. The example highlights the back trace across method boundaries via a parameter to the method invocation point.

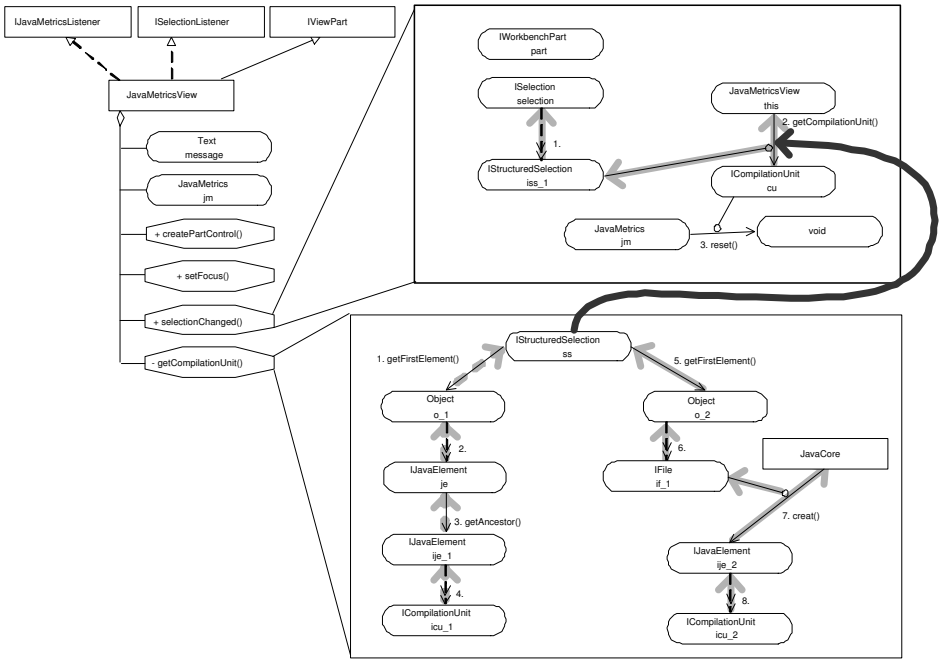

Figure 16: Back Traversal for Query $\mathcal{IQ}$ = (t_q =`ICompilationUnit`, $\mathcal{CT}$ = { }). The code model instances of methods `getCompilationUnit()` and `selectionChanged()` in class `JavaMetricsView` are shown. The example highlights the back trace across method boundaries via the invocation of a local method.

Code Snippets	Length Heuristic (a)	Frequency Heuristic (b)	Context Heuristic (c)
A. ISelection selection; IStructuredSelection ss = (IStructuredSelection) selection; Object obj = ss.getFirstElement(); IJavaElement je = (IJavaElement) obj; IJavaElement ije = je.getAncestor(IJavaElement.COMPILATION_UNIT); ICompilationUnit cu = (ICompilationUnit) ije	5	1	4
B. ISelection selection; IStructuredSelection ss = (IStructuredSelection) selection; Object obj = ss.getFirstElement(); IFile f = (IFile) obj; IJavaElement ije = JavaCore.create(f); ICompilationUnit cu = (ICompilationUnit) ije;	6	2	5
C. IEditorPart editor; IEditorInput input = editor.getEditorInput(); IWorkingCopyManager manager = JavaUI.getWorkingCopyManager(); ICompilationUnit cu = manager.getWorkingCopy(input);	3	4	1
D. JavaEditor editor; Object editorInput = SelectionConverter.getInput(editor); ICompilationUnit unit = (ICompilationUnit) editorInput;	1	3	2
E. Map fMap; IEditorInput input; Object obj = fMap.get(input); ICompilationUnit unit = (ICompilationUnit) obj;	2	5	6
F. JavaPlugin jp = JavaPlugin.getDefault() IWorkingCopyManager manager = jp.getWorkingCopyManager(); CompilationUnitEditor editor; IEditorInput iei = editor.getEditorInput; ICompilationUnit unit = manager.getWorkingCopy(iei);	4	6	3

Figure 17: Code Snippets Obtained for $\mathcal{Q}$ = (t_q = `ICompilationUnit`, $\mathcal{CT}$ = {}) shown with Length, Frequency and Context Rankings.

A common notion, used in many diverse domains, is that of "frequency" – the more number of times a particular result occurs, the more popular the result and hence deserving of a higher rank. In the context of code snippets, *frequency* refers to the number of times identical code snippets written within or across different source classes are mined from the code sample repository in response to a user query $\mathcal{Q}$. Code snippets are ranked based on this frequency measure – the higher the frequency, the higher the rank of the code snippet. Figure 17(b) gives a sample frequency ranking of the code snippets obtained for $\mathcal{Q}$ = (t_q = `ICompilationUnit`, $\mathcal{CT}$ = {}).

5.3 Ranking by Context

Consider the query $\mathcal{Q}$ = (t_q = `ICompilationUnit`, $\mathcal{CT}$ = { `IEditorPart`}) and the code snippets shown in Figure 17. Careful examination of the code snippets shows that snippet C is possibly the best-fit result for this query $\mathcal{Q}$ as `ICompilationUnit` is instantiated indirectly from `IEditorPart`. This code snippet C is, however, ranked 3 by the length heuristic and 4 by the frequency heuristic. In a similar vein, `ICompilationUnit` in code snippets D and F is indirectly instantiated from the types `JavaEditor` and `CompilationUnitEditor` respectively – both specializations of `IEditorPart`. Hence, while not an exact match, the code snippets D and F are relevant to the query $\mathcal{Q}$ as the code context (`IEditorPart`) can be downcasted to either `JavaEditor` or `CompilationUnitEditor` when applying these snippets. The length and frequency heuristics, however, rank them as 1 and 4, and 3 and 6, respectively.

We posit that contextual information can and should be harnessed to provide more effective ranking for a set of code snippets. We now define the *context match* measure – a formal measure for the match between the context of a code snippet and the context of a developer's code. We use this measure to rank the code snippets returned from the snippet mining algorithm.

The *context match* measure, denoted as $\mathcal{M}_{CT}(\mathcal{Q}, s)$, is a quantitative measure of how well the parent $\mathcal{CP}_s$ and type $\mathcal{CT}_s$ contexts of a code snippet s match the parent $\mathcal{CP}_q$ and type $\mathcal{CT}_q$ contexts specified in the query $\mathcal{Q}$. Formally, the *context match* measure is given as:

$$\mathcal{M}_{CT}(\mathcal{Q}, s) = \frac{\mathcal{M}_P(\mathcal{Q}, s) + \mathcal{M}_{VT}(\mathcal{Q}, s)}{2} \tag{4}$$

where $\mathcal{Q}$ represents the query, s a mined snippet, $\mathcal{M}_P(\mathcal{Q}, s)$ the quantitative measure of the match between the direct parents (superclass and interfaces) of $\mathcal{Q}$ and s, and $\mathcal{M}_{VT}(\mathcal{Q}, s)$ the value quantifying the match between the lexically visible types encapsulated in $\mathcal{Q}$ and s.

The parent context match $\mathcal{M}_P(\mathcal{Q},s)$ is defined as the average of the match between the superclasses $\mathcal{M}_S(\mathcal{Q},s)$ and the match between the interfaces $\mathcal{M}_I(\mathcal{Q},s)$ of $\mathcal{Q}$ and s respectively, and is given as:

$$\mathcal{M}_P(\mathcal{Q}, s) = \frac{\mathcal{M}_S(\mathcal{Q}, s) + \mathcal{M}_I(\mathcal{Q}, s)}{2} \tag{5}$$

where

$$\mathcal{M}_S(\mathcal{Q}, s) = \mathcal{M}_T(superclass(\mathcal{Q}), superclass(s)) \tag{6}$$

and

$$\mathcal{M}_I(\mathcal{Q}, s) = \frac{I + S}{|intf(\mathcal{Q})| + |intf(s)|} \quad (7)$$

where

$$I = \sum_{i_q \in intf(\mathcal{Q})} [max_{i_s \in intf(s)} \mathcal{M}_T(i_q, i_s)] \quad (8)$$

$$S = \sum_{i_s \in intf(s)} [max_{i_q \in intf(\mathcal{Q})} \mathcal{M}_T(i_s, i_q)] \quad (9)$$

Here, *superclass* and *intf* generically refer to superclass and interfaces respectively, while $i_q \in intf(\mathcal{Q})$ and $i_s \in intf(s)$ are specific instances of the interfaces. The term $\mathcal{M}_T(t_c, t_s)$ denotes the match between two given domain types, and $\mathcal{M}_I(\mathcal{Q},s)$ the average match between the two sets of interfaces – $intf(\mathcal{Q})$ and $intf(s)$

The type context match $\mathcal{M}_{VT}(\mathcal{Q},s)$, the match of the lexically visible types in the query $\mathcal{Q}$ and code snippet s, is given as:

$$\mathcal{M}_{VT}(\mathcal{Q}, s) = \frac{\sum_{t_c \in type(\mathcal{Q})} [max_{t_s \in type(s)} \mathcal{M}_T(t_c, t_s)]}{|type(\mathcal{Q})|} \quad (10)$$

where *type* denotes the type context encapsulated in $\mathcal{Q}$ and s, $t_c \in type(\mathcal{Q})$ and $t_s \in type(s)$ are specific types, and $\mathcal{M}_T(\mathcal{Q},s)$ the match between two given domain types.

The type context match $\mathcal{M}_{VT}$ is *asymmetric* – it computes the average of the best similarity of each t_c with $t_s \in type(s)$ providing the best fit from the perspective of the query. The type match $\mathcal{M}_T(t_c, t_s)$ is computed using a type match algorithm [17].

6. EXPERIMENTAL EVALUATION

XSnippet is a code assistant framework that enables developers to retrieve candidate code snippets for solving a particular programming task without writing explicit queries. Figure 18 gives an architectural overview of the `XSnippet` framework, and highlights its three key components, *Query Formulation*, *Snippet Mining* and *Ranking*. The `XSnippet` system has been developed in Java (SDK 2.0) as an Eclipse plugin, and can be invoked from within the Eclipse Java editor. Figure 19 shows a snapshot of the `XSnippet` system that extends the Eclipse JDT user interface by adding a new action "`XSnippet`: query" to the pop-up menu of the Java editor for initiating the query, and a new view "`XSnippet`: result" for displaying the returned code snippets. The *XSnippet* framework is available for download as an Eclipse plugin at `http://www.cs.uml.edu/~dsl`.

A series of experiments were conducted to evaluate the potential benefits of the *XSnippet* system. In particular, the experiments were designed to test the following hypotheses:
Hypothesis 1: Generalized queries, $\mathcal{IQ}_G$, provide better coverage of tasks than the specialized queries $\mathcal{IQ}_T$ and $\mathcal{IQ}_P$ (see Section 6.2 for results).
Hypothesis 2: Context-sensitive ranking heuristic provides better ranks for best-fit code snippets than context-independent heuristics. Similarly, context-independent heuristic degrade sharply with the increase in repository size (see Section 6.3 for results).
Hypothesis 3: Specialized queries combined with context-sensitive ranking heuristics provide better rank ordering for best-fit code snippets than generalized queries using context-sensitive ranking heuristics (see Section 6.4 for results).
Hypothesis 4: The *XSnippet* system provides significant assistance to developers, enabling them to efficiently complete a large variety of programming tasks (see Section 6.5 for results).
Hypothesis 5: The context-dependent approach of the *XSnippet* system allows developers to complete more tasks than other previously proposed approaches (see Section 6.6 for results).

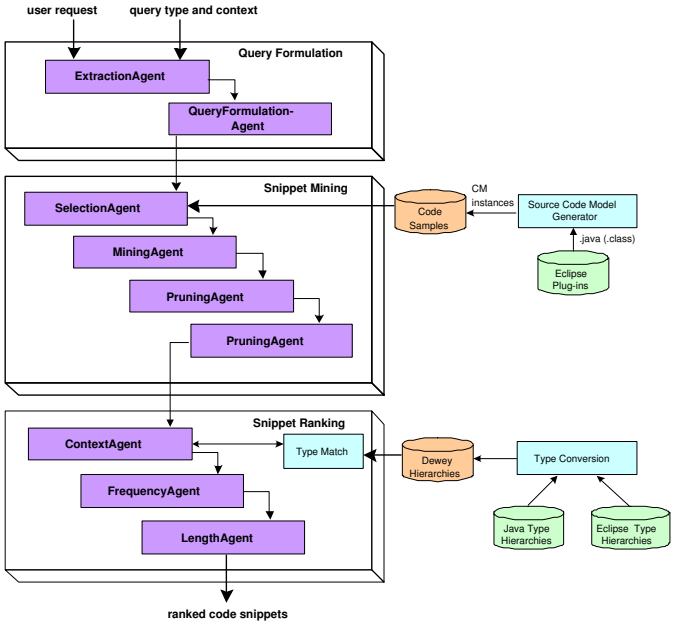

Figure 18: The Architectural Overview of the *XSnippet* framework.

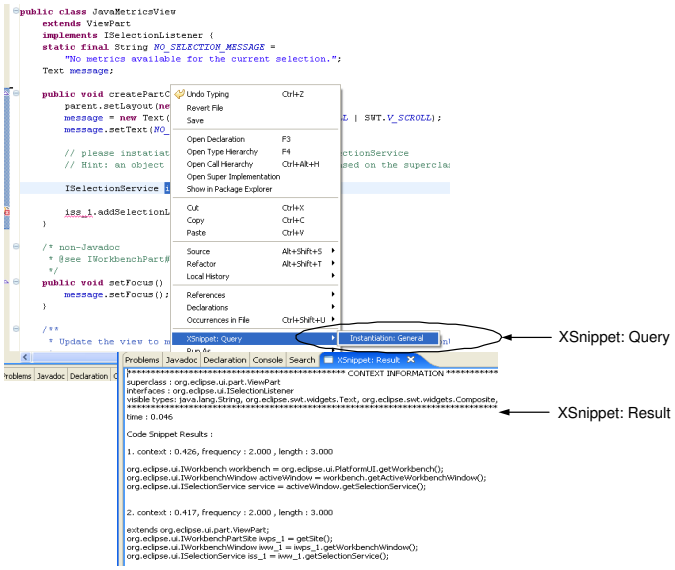

Figure 19: A Snapshot of the `XSnippet` Interface.

6.1 Setup

The *XSnippet* system was deployed on a standalone PC Pentium IV 2.8 GHz with 1 GB RAM running Microsoft Windows XP and Eclipse 3.1. The repository used for the experiments contained approximately 2,000 Java class files and 22,000 methods extracted from two standard Eclipse plugins: `org.eclipse.jdt.ui` and `org.eclipse.debug.ui`. These Java class files were automatically transformed into their corresponding source code model instances and subsequently loaded into the example repository.

To evaluate the *XSnippet* system, we designed 17 object-instantiation specific programming tasks. This was a sufficiently large set to allow measurements of the effects of different parameters, including the context of the code, sample availability, difficulty level, and number of queries required to complete tasks. All tasks were based on the Eclipse plugin examples from *The Java(TM) Developer's Guide to ECLIPSE (The 2nd Edition)* [2]. Table 1 highlights the primary characteristics of the tasks in terms of the type being queried (t_q), the parent ($\mathcal{CT}_P$) and type ($\mathcal{CT}_T$) contexts specified in the provided source class $\mathcal{C}_s$, and the base type t_b from which t_q must be directly (or indirectly) instantiated. For each task, the declaration and usage of type t_q together with all necessary Java source classes and jar files were provided with the object instantiation of type t_q left incomplete. That is, the code was not compilable so the task was to complete the code with the object instantiation.

It should be noted that the code snippets returned by the *XSnippet* system varied in that: (i) some code snippets could be seamlessly integrated with the code under development, thereby enabling the task to be completed at once; (ii) some code snippets introduced new objects into the context, requiring one or more additional queries to complete the task; and (iii) some code snippets encapsulated the relaxed type t'_c of the type t_c in the code context, requiring type modifications based on the type hierarchy to complete the task. Code snippets in any one of the above three categories were considered to be *desirable* as long as they allowed the task to be completed. A task was considered to be *complete* if the code could be compiled and executed, and it enabled the required functionality.

6.2 Effect of Query Types on Task Completion

The first set of experiments measured the number of tasks out of the possible 17 tasks that could be completed using the code snippets returned by the `XSnippet` system. The type of the instantiation queries was varied from the generic query $\mathcal{IQ}_G$ to the specialized *type-based* $\mathcal{IQ}_T$ and *parent-based* queries $\mathcal{IQ}_P$, and the number of tasks that could be completed under each of the query types was measured. The repository was set to contain all of the source code model instances as detailed in Section 6.1.

Figure 20 shows the percentage of tasks completed by the different types of queries – $\mathcal{IQ}_G$, $\mathcal{IQ}_T$ and $\mathcal{IQ}_P$. The x-axis has the different query types and the y-axis is the percentage of tasks completed, with 100% representing all 17 tasks. The query $\mathcal{IQ}_G$ performed the best completing all 17 (100%) tasks, the query $\mathcal{IQ}_T$ completed 11 (65%) tasks, while $\mathcal{IQ}_P$ completed 12 (70%) tasks.

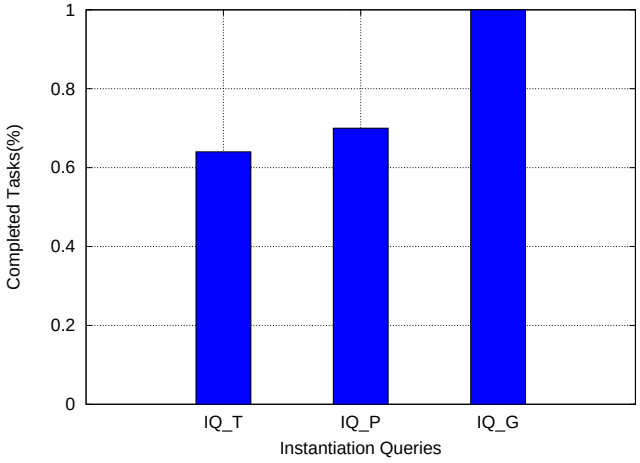

Figure 20: The Percentage of Tasks Supported by the Different Types of Instantiation Queries - $\mathcal{IQ}_G$, $\mathcal{IQ}_T$, and $\mathcal{IQ}_P$.

Furthermore, the characteristics of the tasks supported by each of the query types were analyzed. Figure 21 summarizes these characteristics and relates them to the characteristics of the best-fit code snippets mined from the example repository. The left shows the repository characteristics. The middle column shows the distribution of the tasks that adhere to the repository and code context characteristics. The tasks themselves are broken down into two groups – one where t_b exists in the code context and one where t_b does not. The vertical lines on the right represent the results, in this case coverage of tasks by each query type.

The query $\mathcal{IQ}_T$ was able to support all tasks where (i) the provided source class $\mathcal{C}_s$ encapsulated the base type t_b; and (ii) snippets where the t_q was instantiated from t_b ($t_b \rightarrow t_q$) were mined from the example repository. Twelve (12) tasks out of the total 17 tasks had the type t_b in its code context. Out of these only 10 tasks had snippets where the condition $t_b \rightarrow t_q$ was true. Also, the query $\mathcal{IQ}_T$ was able to

No.	t_q	$\mathcal{CT}_P$	$\mathcal{CT}_T$	t_b
1.	ICompilableUnit	ViewPart, ISelectionListener	Text, JavaMetrics, IWorkbenchPart, ISelection	ISelection
2.	ICompilableUnit	ViewPart, ISelectionListener	Text, JavaMetrics, IWorkbenchPart, ISelection, IStructuredSelection	IStructuredSelection
3.	ICompilationUnit	IElementChangedListener	ICompilationUnit, List, ElementChangedEvent	ElementChangedEvent
4.	ICompilationUnit	AddTraceStatementsAction, IEditorActionDelegate	IEditorPart, IAction	IEditorPart
5.	IEditorInput	AddTraceStatementsAction, IEditorActionDelegate	IEditorPart, IAction	IEditorPart
6.	ISelectionService	ViewPart, ISelectionListener	Text, Composite	ViewPart
7.	ITextEditor	TextEditorAction	-	TextEditorAction
8.	ITextSelection	TextEditorAction	-	TextEditorAction
9.	ITextSelection	TextEditorAction	ITextEditor	ITextEditor
10.	ProjectViewer	AbstractDecoratedTextEditor	SQLCodeScanner, ProjectionSupport Composite	AbstractDecoratedTextEditor
11.	IDocument	IEditorActionDelegate	IAction, TextEditor	TextEditor
12.	ITextSelection	IEditorActionDelegate	TextEditor, IAction, IDocument	TextEditor
13.	ICompilationUnit	AddTraceStatementsAction, IEditorActionDelegate	IAction	JavaEditor
14.	IDocument	IEditorActionDelegate	IAction	TextEditor
15.	IEditorInput	AddTraceStatementsAction, IEditorActionDelegate	IAction	IEditorPart
16.	ITextSelection	IEditorActionDelegate	IAction	TextEditor
17.	IWorkingCopyManager	AddTraceStatementsAction, IEditorActionDelegate	IAction	JavaUI

Table 1: Characteristics of the 17 Programming Tasks.

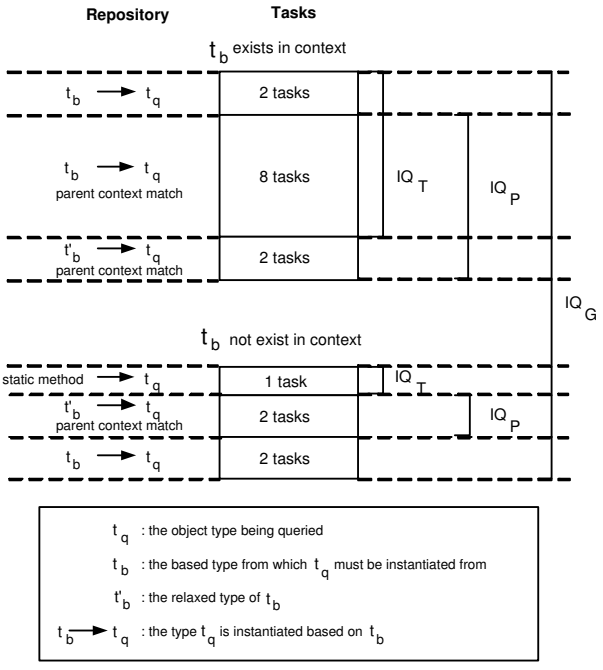

Figure 21: Characteristics of the Tasks Supported by the Different Types of Instantiation Queries.

support tasks that required object instantiation via either a constructor or a static method invocation irrespective of the existence of base type t_b in the code context. The 17^{th} task required an object instantiation based on the static method invocation of the `JavaUI` class. This task was supported by the $\mathcal{IQ}_T$ query. In summary, the query $\mathcal{IQ}_T$ supported 11 tasks – 10 tasks requiring object instantiation of type t_q based on the base type t_b in the code context and 1 task requiring a static method invocation.

The query $\mathcal{IQ}_P$ supported all tasks where the parent context of the provided source class $\mathcal{C}_s$ matched the parent context of the class encapsulating the best-fit code snippet. The existence of the base type t_b in the code context, and the constraint $t_b \rightarrow t_q$ had no impact on $\mathcal{IQ}_P$ performance. Twelve (12) tasks out of the possible 17 had a parent context match between the source class $\mathcal{C}_s$ and the class encapsulating the code snippets, and were completed by $\mathcal{IQ}_P$. Out of these 12 tasks, the t_b existed in the code context for only 10 tasks, and the constraint $t_b \rightarrow t_q$ held for only 8 of the tasks.

The query $\mathcal{IQ}_G$ supported all tasks. The constraint $t_b \rightarrow t_q$ and on the existence of t_b in the code context had no impact on the overall performance of $\mathcal{IQ}_G$. The results returned by $\mathcal{IQ}_G$ were a superset of the results returned by $\mathcal{IQ}_T$ and $\mathcal{IQ}_P$.

6.3 Impact of Contextual Information on Ranking

The second set of experiments measured the effectiveness of the different ranking heuristics for different types of queries. Here, effectiveness is defined as the ability of the ranking heuristic to return the best-fit code snippet in the top-k results. The best-fit code snippet for each of the 17 tasks was determined apriori, and these were used to evaluate the different ranking heuristics. In addition, a *combination* of these ranking heuristics to hierarchically rank the results first by context then within a group by frequency and subsequently length was introduced and evaluated. All experiments were conducted using the 17 tasks, the three query types and the complete repository set up as described in Section 6.1.

Due to space constraints, the ranking heuristics results for only the $\mathcal{IQ}_G$ is reported. Results for the $\mathcal{IQ}_T$ and $\mathcal{IQ}_P$

were similar and can be found in [18].

Figure 22 shows the cumulative distribution of the best-fit code snippet ranking for the different ranking heuristics. The results are reported for all 17 tasks supported by $\mathcal{IQ}_G$. The x-axis is the ranking given as integer numbers and the y-axis is the cumulative distribution. The combination ranking heuristic performed the best with the best-fit code snippet for all tasks ranked within the top-33. For 35% of the tasks, the combination heuristic ranked the best-fit code snippet first. The context ranking heuristic ranked the best-fit code snippet for all tasks within the top-80. For 35% of the tasks, however, the best-fit code snippets ranked first, a result similar to the combination ranking heuristic. Additionally, for 88% of the tasks the context and combination heuristics returned the same rank for the best-fit code snippet. The frequency ranking heuristic placed the best-fit code snippet for all tasks within the top-126, with 23% of the tasks having the best-fit code snippet first. The length ranking heuristic performed the worst, placing the best-fit code snippet first for only 5% of the tasks.

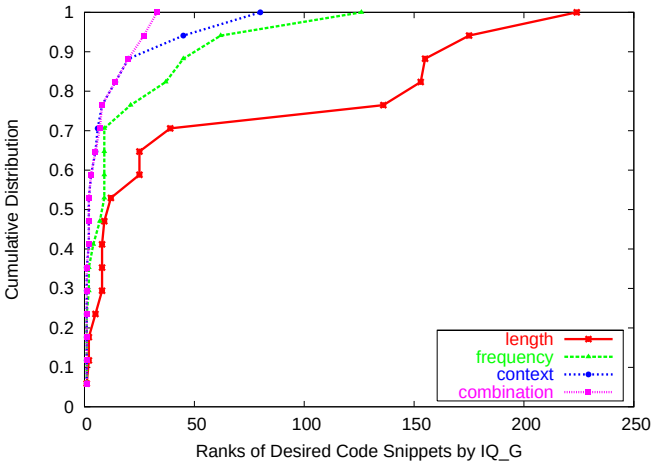

Figure 22: Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using $\mathcal{IQ}_G$.

Furthermore, the effect of the number of code snippets returned by the query $\mathcal{IQ}_G$ on the stability of the ranking heuristic was analyzed. Figure 23 shows the average ranking of the best-fit code snippets for versus the number of result code snippets returned. The x-axis is the number of code snippets returned by the query and the y-axis is the rank of the best-fit code snippet. The performance of the length and frequency heuristics degraded sharply for larger numbers result snippets. The combination and context heuristics were relatively stable, even for larger numbers of returned results. This performance difference was attributed to the sensitivity of the length and frequency heuristics to the size and richness of the example repository.

6.4 Effects of Query Types on Ranking

The third set of experiments measured the rank of the best-fit code snippets (for a given task) for different query types – $\mathcal{IQ}_T$, $\mathcal{IQ}_P$ and $\mathcal{IQ}_G$. To level the playing field, the experiments were conducted using only the 6 tasks sup-

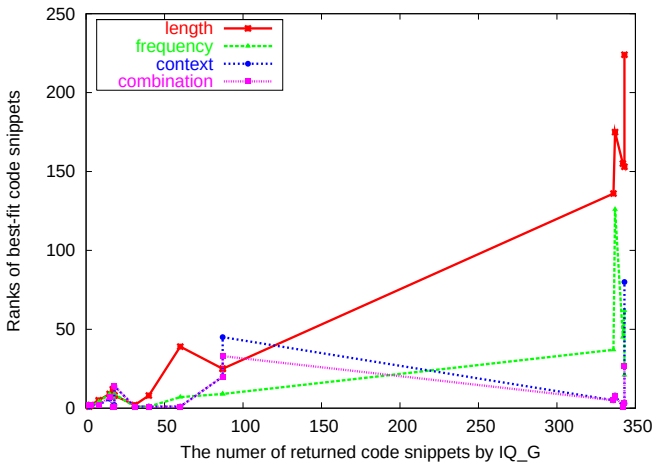

Figure 23: Variation in the Average Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $\mathcal{IQ}_G$.

	1^{st}	2^{nd}	3^{rd}
$\mathcal{IQ}_T$	5	0	1
$\mathcal{IQ}_P$	4	1	1
$\mathcal{IQ}_G$	3	2	1

Table 2: Distribution of the Best-Fit Ranks for Different Query Types.

ported by all three query types. The ranking heuristic was fixed to the combination heuristic for all queries.

Table 2 shows the distribution of the ranks of the best-fit code snippets for the 6 tasks supported by the $\mathcal{IQ}_T$, $\mathcal{IQ}_P$ and $\mathcal{IQ}_G$ queries. Query $\mathcal{IQ}_T$ performed the best, having the the best-fit code snippet ranked first 5 out of 6 times. Query $\mathcal{IQ}_P$ did the next best, with one fewer top-ranked queries. Query $\mathcal{IQ}_G$ performed the worst returning the best-fit code snippet in first only about 1/2 the time. This difference in performance was attributed to the fact that $\mathcal{IQ}_T$ and $\mathcal{IQ}_P$ filtered out non-candidate code snippets as part of the mining and snippet selection process while $\mathcal{IQ}_G$ returned all code snippets mined from the example repository. This effectively deteriorated the overall ranking for $\mathcal{IQ}_G$. In summary, specialized queries ($\mathcal{IQ}_T$ and $\mathcal{IQ}_P$) provide higher ranking best-fit code snippets when compared to the generalized query $\mathcal{IQ}_G$.

6.5 Analysis of *XSnippet* on Assisting Developers

The fourth set of experiments analyzed the use of *XSnippet* in assisting developers to complete their programming tasks. Four tasks with varying degrees of difficulty were designed for the study. Table 3 gives a brief description of the four tasks together with a *difficulty* rating, as well as the ranking of the best-fit snippets returned by the system. Detailed descriptions of the tasks can be found in [18]. For each task, the declaration and usage of type t_q together with all necessary Java source files, jar files and instructions on code execution were provided, but the object instantiation

of the type t_q was left incomplete. In all four tasks the provided code could not be compiled without completing the task modifications.

Volunteers from the UMass-Lowell population were solicited to participate in the study. Initially, one group of users was created to complete the tasks without the use of *XSnippet*. These users were free to use the default Eclipse Java code assistant tool, the Eclipse API browser, search online via Google or use any other means available via the Internet. However, the users struggled for over one hour on the first task alone and, still unable to complete it, became frustrated and quit. It was therefore assumed for the majority of the users that they would be unable to complete the tasks without the use of *XSnippet*.

For the remainder of the study, each participant was trained to use the *XSnippet* system prior to them conducting the study. Participants were then randomly assigned to one of two possible groups. The first group was provided *hints* on the t_b from which an object of the query type t_q should be instantiated. These hints appeared as comments in the provided source code. The second group was not given any hints. For each participant, the number of tasks completed together with the development time to complete each task was recorded. In addition, the strategy employed to select the result snippet by each participant was monitored.

A special test harness was developed to conduct the user study. The test harness allowed participants to enter demographic and programming experience information at the start of the study. For each task, the test harness provided a screen with a brief description of the task, prior to launching the Eclipse development environment pre-set with all required files for the task. The test harness automatically recorded the time taken for the participants to complete each task. On completion of each task, participants were asked to fill a brief questionnaire that solicited information on the usefulness of the *XSnippet* system.

All experiments were conducted using the 4 tasks, the generic instantiation query $\mathcal{IQ}_G$, the *combination* ranking heuristic, and the complete repository setup as described in Section 6.1.

Participant Statistics. Sixteen participants took part in the study. Out of these, seven participants were randomly binned into the *Hint* group and nine participants were placed in the *NoHint* group. Most of the participants were Computer Science graduate students with one CS undergraduate in each group. The participants had comparable programming experience (C++ and Java) with most participants having on average about 1 year experience in one or both languages. Only 6 of the participants had used Eclipse as a development environment prior to conducting the study, and none of the participants had any experience with developing Eclipse plugins.

Given the participant population and distribution of programming experience, the *hint* group approximated the more knowledgeable and experienced developers, and the *no hint* group approximated the novice developers.

Tasks Completed. For each participant the number of tasks completed was tracked. Figure 24 summarizes the results for task completion by participants in each group. The x-axis has the tasks and the y-axis is the percentage of participants that completed each of four tasks.

In the *hint* group, all participants completed Task A, B, and D. This 100% completion rate is helped in part by the provided *hints* – the participants used the provided hints to select the best-fit code snippet as opposed to utilizing the snippet ranking. For Task C, type modification in the code snippet was required to complete the task. Participants with experience and knowledge of Java type hierarchy (about 80% of participants) were able to complete the task. In the *no hint* group, all participants completed Task A and B, 67% of the participants completed Task C, and 56% of the participants completed Task D. Participants in the *no hint* group were observed to use ranking of the code snippets as their primary criteria for selection of snippets. For Task D, the best-fit code snippet was ranked 27 accounting for the fewer number of participants that completed the task. For Task C participants had similar problems with the type modification required in the code snippet.

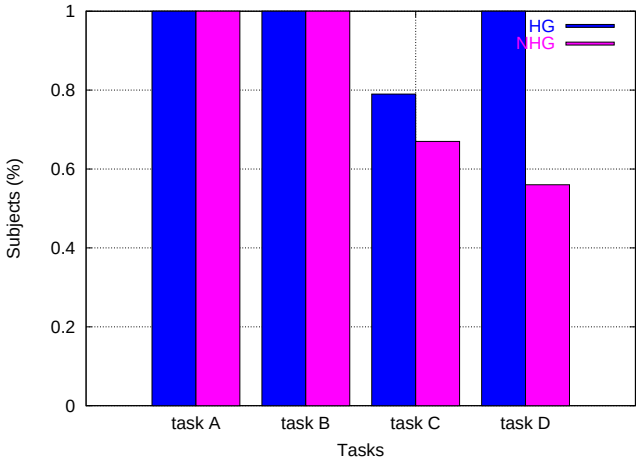

Figure 24: Percentage of Participants that Completed the Four Given Tasks in the *Hint* and *No Hint* Groups.

Task Development Time. For each participant the development time for each task was tracked. The development time included (i) the Eclipse launch time; (ii) the query process time; (iii) the time taken for the participant to select a code snippet from the query results; and (iv) the time taken to modify and test the provided source code.

Figure 25 shows the time taken by each participant to complete the provided task. Development times for participants that did not complete the task were discarded. The x-axis has the tasks and the y-axis is the development time in minutes. The bars for each task represent the *hint* and *no hint* groups with the individual participant times marked on the bars. The lines across the tasks represents the average time taken by the participants to complete each task in the *hint* and *no hint* groups.

Participants in both groups reported similar development times for completing Task A and Task B. For Task C and Task D, participants in the *hint* group fared better – selecting the best-fit code snippet based on the provided hints. The lower average development times for Task C and D for the *hint* group are representative of the improvement due to

Task	Description	$\mathcal{CT}_P$	$\mathcal{CT}_T$	Difficulty	Best-Fit Snippet Ranking
A	Instantiate `ISelectionService` that tracks the selection within the package explorer	`ViewPart`, `ISelectionListener`	`Text`, `Composite`	Easy	1 or 2
B	Instantiate `ICompilationUnit` that represents the .java file selected from the package explorer	`ViewPart`, `ISelectionListener`	`TextEditor`, `IAction`, `IDocument`	Easy-Medium	7 or 8
C	Instantiate `ITextSelection` that represents the text selected from the Text editor	`IEditorActionDelegate`		Hard	6, 7 or 10
D	Instantiate `ICompilationUnit` that represents the .java file appearing on the Java editor	`AddTraceStatementsAction`, `IEditorActionDelegate`	`IAction`	Medium-Hard	27

Table 3: Brief Description of the Fours Tasks Used for the User Study.

hints. Most participants in the *no hint* group tried several code snippets for Task C and Task D before hitting upon the best-fit code snippet. This was attributed to the fact that (i) for task C, there were two or more code snippets that encapsulated the lexically visible types in the code context and were ranked higher than the best-fit code snippets; and (ii) for Task D, code snippets returned by the query $\mathcal{IQ}_G$ did not have context match with the provided source code.

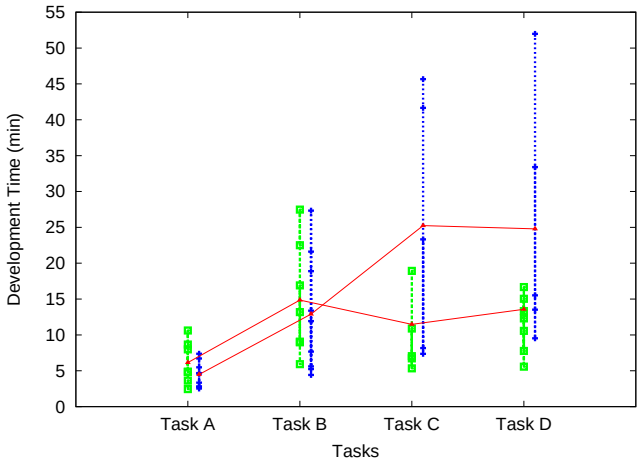

Figure 25: Development Time (Individual and Average) For Completing the Four Tasks. The results are categorized by the two groups.

The 17 tasks, shown earlier in Table 1, were used for comparison. For each task, the $\mathcal{IQ}_T$ and $\mathcal{IQ}_G$ in the *XSnippet* system and the $\mathcal{IQ}'_T$ in the Prospector system, were used to return 12 results. Each result was inserted in turn into the code, compiled and executed to check if the desired functionality was achieved. If the desired functionality was achieved for at least 1 of the 12 returned results, the task was counted as completed.

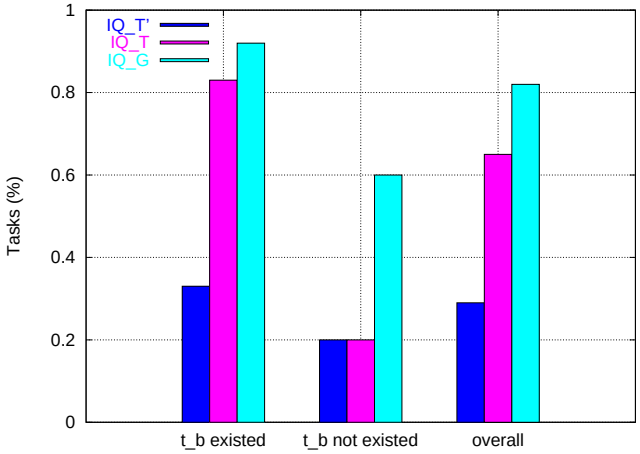

Figure 26: Percentage of Tasks Completed By Prospectors $\mathcal{IQ}'_T$ and *XSnippet* $\mathcal{IQ}_T$ and $\mathcal{IQ}_G$. The results are categorized by the existence of t_b in the code context.

6.6 Comparison with Prospector

The last set of experiments was designed to compare the *XSnippet* system with another code assistant system. The Prospector [11] was chosen for comparison as the Strathcona [8] system was not available.

The repository for *XSnippet* was setup as described in Section 6.1. The default repository for the Prospector system was used. This repository contained complete source code from Eclipse 3.0 standard plugins, GEF plugins and Eclipse/GEF standard examples. In essence, the Prospector repository was a superset of the *XSnippet* repository. Moreover, as Prospector only reports the top 12 code snippets, the number of code snippets returned by *XSnippet* was also limited to 12.

The experiments measured the percentage of tasks that could be completed using Prospector and *XSnippet*.

Figure 26 shows the percentage of tasks completed by $\mathcal{IQ}_T$, $\mathcal{IQ}_G$ and $\mathcal{IQ}'_T$. The primary task characteristic used for analyzing the results was the existence (or lack of) of the base type t_b in the code context. The x-axis plots the two main criteria used for the comparison, and the y-axis plots the percentage of tasks completed by each query type. The *XSnippet* $\mathcal{IQ}_T$ performed better than Prospectors' $\mathcal{IQ}'_T$ for tasks where the t_b existed in the code context. This performance difference can be attributed to the fact that Prospector query evaluation is limited to the lexically visible types within the class boundary and superclass information is discarded. The *XSnippet* $\mathcal{IQ}_T$ and Prospectors' $\mathcal{IQ}'_T$ performed similarly for the tasks where t_b did not exist in the code context. The query $\mathcal{IQ}_G$ performed the best and provided code snippets for 82% tasks irrespective of whether or not the base type t_b existed in the code context.

7. RELATED WORK

Most of previous research on software reuse has focused on the component matching to discover relevant components (classes or methods) as a whole that satisfy a given user query $\mathcal{Q}$. These approaches include keyword-based [12], faceted [14], signature matching [21], specification matching [22, 9], test cases [5], comment matching [20, 19] and name matching [17]. However, while they are a necessary step for identifying possible reusable components from a set of requirements, these works are orthogonal to software reuse that assists by providing code samples.

Holmes et al. [8] have developed Strathcona, an Eclipse plug-in, that enables location of relevant code in an example repository. Their approach is based on six heuristics that match the structural context descriptions encapsulated in the developer code with that encapsulated in the example code. The result is a set of examples (source code examples) that occur most frequently when applying all heuristics. This approach while a good step forward, has some drawbacks: (i) each heuristic is *generic*, that is it is not tuned to a particular task of object instantiation or method invocation. This results often in irrelevant examples; and (ii) each heuristic utilizes all defined context, irrespective of whether the context is relevant or not. This over-constraining of the heuristic can result in too few examples or sometimes no examples. In our work, we use similar contextual information, but tie it with specialized queries thereby providing flexibility without the unnecessary constraints.

Mandelin et al. [11] have developed techniques for automatically synthesizing the *Jungloid* graph based on a query that describes the desired code in terms of input and output types. Each result corresponds to an object instantiation of a query output type $\mathcal{T}_{out}$ derived from a given input type $\mathcal{T}_{in}$. The Jungloid graph is created using both API method signatures and a corpus of sample client programs, and consists of chains of objects connected via method calls. The retrieval is accomplished by traversing a set of paths from $\mathcal{T}_{in}$ to $\mathcal{T}_{out}$ where each path represents a code fragment and a set of paths in turn composes all code fragments to form a code snippet. The code snippets returned by this traversal process are ranked using the length of the paths with the shortest path ranked first. While Prospector provides more refined, object-instantiation specific queries than Strathcona, it still returns many irrelevant examples or in some cases too few qualified examples. This is primarily because (i) API signatures can be over-used. For example, two code segments may be connected by common types such as `List` and `Set`. However, each may expect to contain completely different object types; and (ii) the context description is limited to only the visible input types declared within the boundary of the method and class. The parent context, as described in Section 2, is ignored missing a set of potentially qualified hits. In our work, we explicitly address the above two limitations, and also provide more generalized query types. In addition, we have developed a context-sensitive ranking heuristic that as per our evaluation, provides better ranking of the best-fit code snippets than the shortest path heuristic used in Prospector.

Hill et al. [7] have developed a method completion tool as a plugin for jEdit 4.2. The tool automatically completes a method body based on the current context of a method being developed using machine learning. Their approach defines each method as a 154-dimensional vector: the number of lines of code, the number of arguments, a hash of the return type, the cyclomatic complexity, and the frequency count of each of the 150 Java Language token types. The vector v_q of a method being queried is compared to a set of pre-computed vectors v_t of methods in the example repository, and the best method match is returned to the developer, such that the difference between v_q and v_t is small. This enables the developer to see similar methods. The developer can choose to complete the current method. This approach suffers from some of the same drawbacks as Strathcona – the approach is general to all types of queries and can not be used to ask a more directed query. Moreover, the approach is inflexible using a complete vector comparison at all times, and potentially missing out on partial matches that may be of use. Like Prospector, this approach limits the context to just the type context and does not make use of the parent context.

8. CONCLUSIONS AND FUTURE WORK

This paper present *XSnippet*, an Eclipse plugin that allows developers to query for relevant code snippets from a sample code repository. Here, relevance is defined by the context of the code, both in terms of the parents of the class under development as well as lexically visible types. Figure 27 highlights the primary contributions of this work. *Queries*, invoked from the Java editor, can range from the generalized object instantiation query that returns all possible code snippets for the instantiation of a type, to the more specialized object instantiation queries that return either parent-relevant or type-relevant results. Queries are passed on to a graph-based *Snippet Mining* module that mines for paths that meet the requirement of the specified query. Paths here can be either within the method scope or outside of the method boundaries, ensuring that relevant code snippets that are spread across methods are discovered. All selected snippets are passed on to the *Ranking* module that supports four types of ranking heuristics: context-sensitive ranking, frequency-based ranking, length-based ranking, and a ranking heuristic that combines the three.

Our experimental evaluation of the *XSnippet* system has shown: 1) a *generalized* query provides better *coverage* of tasks than does a specialized query. In our experiments, the generalized query $\mathcal{IQ}_G$ was able to cover (provide best-fit snippets) for all tasks; 2) a *specialized* query provides *better-fit* code samples at a higher rank when used with a context-sensitive ranking algorithm; 3) context-sensitive ranking performs significantly better than context-independent ranking and is less susceptible to variations in the size of the repository; 4) *XSnippet* has the potential to assist developers by providing sample code snippets relevant to their task at hand, and to help decrease the overall development time; and 5) *XSnippet* provides better coverage of tasks and better ranking for best-fit snippets than other code assistant systems.

Future Work: There are a number of directions in which this work can be extended. An immediate direction is a more extensive user study that would go beyond general usage to investigating the impact of such a code assistant framework for actual development tasks. Another direction is to extend the queries beyond object instantiation to queries that pertain to the correct usage of methods. These correspond to discovering embedded sequences of methods that

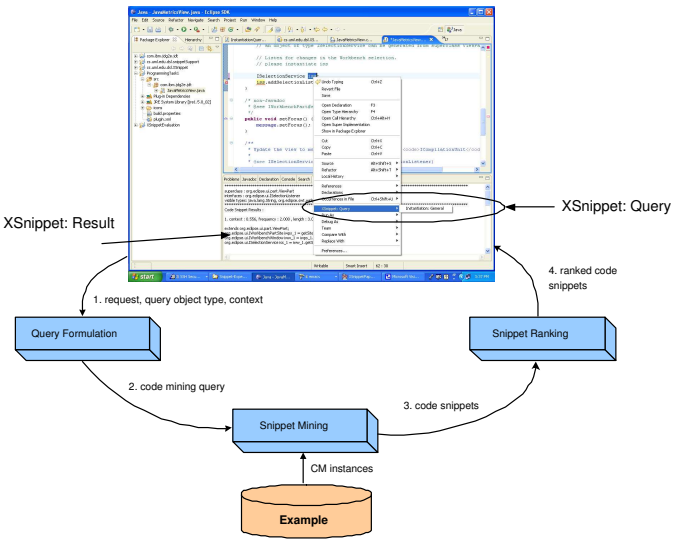

Figure 27: An Overview of the *XSnippet* Framework.

must be used together to ensure proper method behavior such as: 1) methods linked by the life cycle of their classes. For example, to ensure correct behavior of an object of type `IWorkingCopyManager` the methods *connect*, *getWorkingCopy* and *disconnect* must be used together; and 2) methods commonly interconnected by an invocation path. For example, method *apply* of the `TextEdit` class is commonly used in concert with the method *rewriteAST* of the class `ASTRewrite` to apply the edit-tree returned by the method *rewriteAST()* to the given document.

9. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. How reuse influences productivity in object-oriented systems. *Communications of ACM*, 39(10):104–116, 1996.

[2] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java(TM) Developer's Guide to ECLIPSE (The 2nd Edition)*. Addison-Wesley Professional, 2004.

[3] O. Drori. Algorithm for Documents Ranking: Idea and Simulation Results. In *Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering*, pages 99 – 102. ACM Press New York, NY, USA, 2002.

[4] P. Freeman. *Software Reusability*. IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 USA, 1987.

[5] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins. Software Component Search. *Journal of Systems Integration*, 6(1/2):93–134, March 1996.

[6] G. T. Heineman and W. T. Councill. *Component-based Software Engineering*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.

[7] R. Hill and J. Rideout. Automatic method completion. In *The 16th IEEE International Conference on Automated Software Engineering*, pages 228–235, 2004.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*. ACM Press, 2005.

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

[10] G. T. Leavens and M. Sitaraman. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[11] D. Mandelin, L. Xu, R. Bodk, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 2005.

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

[13] O. Nierstrasz and T. D. Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, 1995.

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

[15] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *ACM SIGMIS Database*, 34(3):8–24, 2003.

[16] N. Tansalarak and K. T. Claypool. QoM: Qualitative and Quantitative Schema Match Measure. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER 2003)*, October 2003.

[17] 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.

[18] N. Tansalarak and K. T. Claypool. XSnippet: A Code Assistant Framework. Technical Report 2006-xxx, Department of Computer Science, University of Massachusetts - Lowell, March 2006. Available at http://www.cs.uml.edu/techrpts/reports.jsp.

[19] Y. Ye and G. Fischer. Supporting reuse by delivering taskrelevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, pages 513–523. ACM Press, May 2002.

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

[21] A. M. Zaremski and J. M. Wing. Signature Matching: a Tool for Using Software Libraries. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 146–170. ACM Press, 1995.

[22] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 333–369. ACM Press, 1997.