Using Structural Context to Recommend Source Code Examples

Reid Holmes and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4
rtholmes,murphy@cs.ubc.ca

ABSTRACT

When coding to a framework, developers often become stuck, unsure of which class to subclass, which objects to instantiate and which methods to call. Example code that demonstrates the use of the framework can help developers make progress on their task. In this paper, we describe an approach for locating relevant code in an example repository that is based on heuristically matching the structure of the code under development to the example code. Our tool improves on existing approaches in two ways. First, the structural context needed to query the repository is extracted automatically from the code, freeing the developer from learning a query language or from writing their code in a particular style. Second, the repository can be generated easily from existing applications. We demonstrate the utility of this approach by reporting on a case study involving two subjects completing four programming tasks within the Eclipse integrated development environment framework.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Program Editors; D.2.3 [**Coding Tools and Techniques**]: Object-Oriented Programming; D.2.6 [**Programming Environments**]: Programmer Workbench

General Terms

Languages, Experimentation

Keywords

recommender, examples, software structure, development environment framework

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

1. INTRODUCTION

Frameworks allow software developers to create full-featured applications with less effort. Achieving this benefit requires a developer to use the framework appropriately: subclassing particular classes, instantiating appropriate objects, and calling methods according to established protocols. Some of these constraints on framework use are described in design or API documents; others are specified through code examples crafted specifically to demonstrate particular features of the framework. It is seldom the case that the documentation and examples provided with a large framework are sufficient for a developer to use the framework effectively. All too often, developers become stuck when trying to use the framework, unsure of how to make progress on a programming task.

To help unstick developers caught in this situation, researchers have advocated the establishment of example repositories to house examples of a framework's use (e.g. [10, 11, 14]). These approaches differ in the means that a developer uses to retrieve relevant examples from the repository; developers must either learn a new query language [3], have an idea of what type of example would likely help them with their task [8], or write their source in a style that conforms to that of the example repository [14]. All of these approaches make it too hard for a developer to locate and incorporate examples from the repository.

To ease the burden on the developer, we describe an approach that uses the structure of the code under development to find relevant examples in a repository. Our approach has two advantages. First, the *structural context* that is used to form a query is extracted automatically from the code a developer is writing. A developer who wishes to search the repository need only issue a search request, such as through a keystroke, to find a list of related examples. The developer need not learn a new query language, nor must the developer code to particular standards to enable a search to be conducted. Second, the repository of examples is extracted automatically from existing applications that use the framework. Specific work need not be performed to craft the examples for the repository.

To investigate this approach, we built the Strathcona tool. The client portion of this tool, a plug-in for the Eclipse integrated development environment (IDE),[1] extracts the structural context of the code on which a developer is working

[1] `eclipse.org`

when the developer requests examples. The server portion of the tool houses the example repository and selects examples to be returned using a set of structural matching heuristics. In our approach, an example is a subset of one of the applications stored in the repository, consisting of a set of relevant classes and relationships. The developer is presented with a structural overview of each example using a compact visual representation. The developer can access a rationale for why the example has been returned, as well as the source for the example.

In evaluating our approach, the key question of interest was whether our structural matching heuristics can return examples that a developer finds useful. As it is only possible to understand if an example is useful in the context of a task, we performed a qualitative evaluation in which two subjects replicated four cases; each case consisted of a programming task related to writing plug-ins for Eclipse. We chose Eclipse as the framework for evaluation because it is a large framework with substantial example code available, both in form of third-party plug-ins as well as the plug-in code that collectively makes up the Eclipse system. For the evaluation, the Strathcona repository was populated with the Eclipse system code, because these plug-ins should be good examples of the use of the framework. This code comprised approximately 1.5MLOC. In all but one instance in which there were relevant examples in the repository, the subjects in our study were able to access the relevant examples, understand them, and complete the programming task. These results provide initial evidence that structural matching is appropriate to deliver relevant examples to help ease framework use.

We begin the paper with a scenario describing the tool's use (Section 2). Next, we compare our approach to other efforts (Section 3), describe our approach and tool in detail (Section 4), and present our evaluation (Section 5). We conclude the paper with a discussion of open issues (Section 6) before summarizing (Section 7).

2. SAMPLE SCENARIO

The Eclipse user interface includes a status line that reports information about the status of the environment to the user. For example, when the user selects a number of items from a tree view in Eclipse, the status line shows the number of selected items. Consider a developer who is writing an Eclipse plug-in and who wants to display a message on the status line. The first place a developer might look for help with this task is the Eclipse documentation. Checking this resource, the developer finds a reference to an interface called `IStatusLineManager`. Looking at the API documentation for `IStatusLineManager`, the developer finds the seemingly appropriately named method, `setMessage(String)`, but the documentation does not describe how to get a handle to a `StatusLineManager` object needed to call this method. At this point, the developer becomes stuck as to what is the next step needed to complete the task as there is no documentation available to help with the next step to complete the task. This scenario occurred during the development of the Strathcona plug-in.

Strathcona can help a developer in this situation. The developer adds to an existing method in their plug-in, named `updateStatusBar(String)`, the statement `IStatusLineManager.setMessage(String)`.[2] As shown in the top portion of Figure 1(a), the developer then selects `Query Related` from the context menu to request similar examples from Strathcona. The client portion of Strathcona generates a structural context of the code the developer is writing that comprises details of the method being written, `updateStatusBar`, and its containing class `CodeViewer`. In this case, the context also includes the call by `updateStatusBar(String)` to `IStatusLineManager.setMessage(String)` and its fields (`SourceViewer, Action`). This context is sent to the server portion of Strathcona, which returns ten structurally-related examples. Each of these examples consist of three parts: a code snippet, a structural description of the code snippet, and a rationale explaining the relevance of the code snippet to the problem the developer is facing.

Figure 1(b) shows the structure of one of the returned examples, which relates to the code for one of the components within the Eclipse IDE. The rationale for this example describes that it was selected because the `setMessage(String)` method is being called, `IStatusLineManager` is being used by the example, and the example extends `ViewPart` (Figure 1(c)). The developer requests the code for the example, and Strathcona highlights the call chain `getViewSite().getActionBars().getStatusLineManager().setMessage(msg);` as shown in Figure 1(d). The developer assesses this example as useful and attempts to use it by directly copying the statement into their method and changing the argument. Testing this code, the developer finds that it completes the task.

This scenario describes a conceptually simple task of updating a status line. However, even this simple task requires knowledge about the interaction between several types of the framework, including `ViewPart, IViewSite, IActionBars,` and `IStatusLineManager`. This interaction is not described in the Eclipse documentation. Although the interactions may be discovered using the code completion features in Eclipse, the correct sequence of calls is difficult to find as there are 79 methods available across the four classes.

3. RELATED WORK

Our claim in this paper is that the use of structural context to match possible examples in an example repository places fewer constraints and less of a burden on a developer than existing approaches. We focus our comparison to related software example system efforts.

Of these systems, Strathcona most resembles the CodeBroker system [14, 13]. CodeBroker queries a repository automatically after each comment or method signature written by a developer. The queries made to the repository are based on these comments and method signatures. To retrieve matches, a developer must write comments that explain the functionality of the software in terms similar to that of the repository code [14]. When a developer follows this process, CodeBroker may be able to match a more diverse set of examples than Strathcona. However, the effectiveness of this approach may be limited by the need to

[2]Although this snippet does not compile, our system can tolerate incomplete fragments (Section 4.2). In general, to use Strathcona, a developer need not write special code to serve as a query; the tool is designed to work from the code a developer has written simply as part of their task.

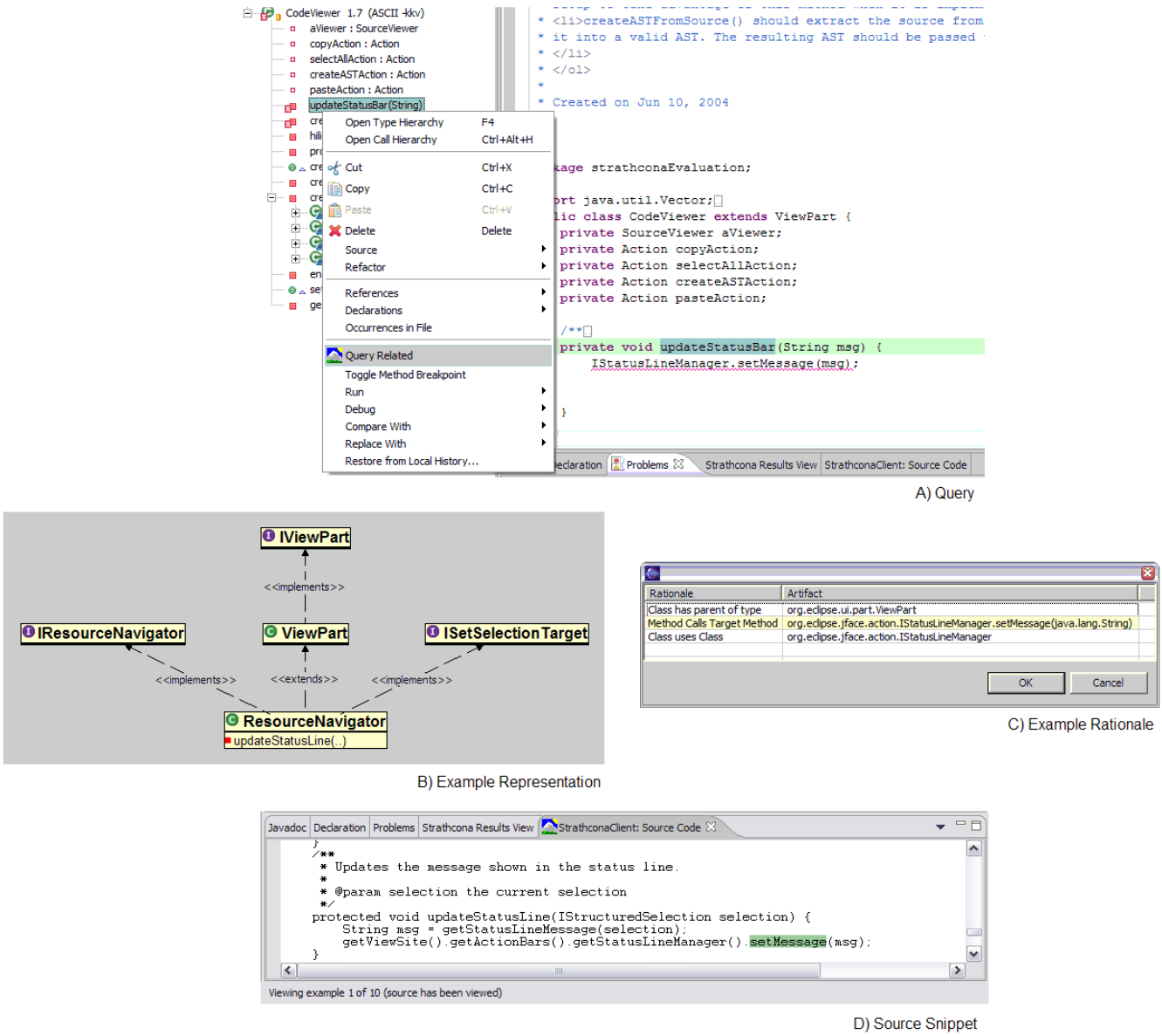

Figure 1: Strathcona Cycle

and difficulty of writing appropriate comments. In comparison, our approach can apply to any code and any framework irrespective of coding conventions since all source code incorporates structure.

The CodeFinder system represents another point in the design space of software example systems by attempting to help developers construct useful queries [3]. The developer formulates a simple text query, executes the query, and is then presented with a list of terms in the repository that are similar to those in the query. Depending on the terms and options selected by the developer, a different set of restrictions is presented to help narrow the search space to a specific class of examples of interest. In contrast to CodeFinder, Strathcona aims to remove the step of formulating the query by creating the query automatically.

Other tools, such as Component Rank [6] and CodeWeb [8, 9], use software structure to determine which parts of a framework are frequently used. Of these tools, CodeWeb is the most similar as it provides information about which classes and methods are frequently used in a framework and how they are used. To provide this information, a developer must populate CodeWeb with applications that are similar to the one which they are developing. Although the intent of CodeWeb is similar to Strathcona, it differs in three ways. First, a developer must find similar applications of interest in advance. Second, the structural attributes are used to compare complete projects against one another, instead of enabling the use of fragments of projects. Third, the need to find applications in advance suggests that a developer would be more likely to engage in the use of CodeWeb at the beginning of the development process as it is based on browsing rather than querying.

The Reuse View Matcher (RVM) provides a set of views describing how an application makes use of a particular class in a framework [12]. This technique relies exclusively on hand-crafted examples which can be time-consuming to create, can be out-of-date with the code, and may not have coverage of all of the classes in the system.

The Automatic Method Completion technique [4] uses machine learning techniques to complete a method body based upon the developer's current context. The approach represents the programming language constructs and named identifiers used in the method as a multi-dimensional vector. This vector is compared to pre-computed vectors for example code on a server, and the best completion for the method of interest is returned. Strathcona differs in returning multiple possible examples for arbitrary blocks of code based on structurally similar features.

The Hipikat tool [2] can recommend relevant development artifacts from a project's history to a developer. One kind of artifact that can be recommended is the source revisions associated with a past change task; these revisions can be considered as an example. Strathcona extends the kinds of examples that can be recommended to a developer by drawing the examples from current uses of a framework, rather than relying solely on the past development history of the framework itself.

4. Strathcona

We describe the implementation of our Strathcona tool in terms of the workflow of a developer using the tool. The process by which the repository is initially populated is described in Section 4.1. When a developer requests examples, a structural context description is generated from the information in the development environment, and is sent to the server (Section 4.2). Upon receipt of the description, the server, according to a set of heuristics, performs queries on the repository that attempt to match the structure described in the query to the structure of the code stored in the repository (Section 4.3). Examples that include code snippets with the best structural matches to the context are returned for perusal, and hopefully use, by the developer (Section 4.5). We conclude the section with a discussion of a performance of Strathcona (Section 4.6).

4.1 Server: Populate Repository

Before the client portion of the tool can be used, the repository, which resides on the server, must be populated with code that uses the framework and from which examples may be selected. A repository manager loads the code into Strathcona using an Eclipse plug-in that extracts the structural information of interest from the code and stores it in the repository database. The repository consists of a relational database that stores the structure of the code: the classes, methods, fields, inheritance relation between classes, the types instantiated by the code, and the calls between the types. There are two restrictions placed on the code used to populate the repository: the code must be parseable by the Eclipse compiler, and the code should represent good usage of the framework. The code may be from multiple applications; portions of applications may also be loaded into the repository. The examples returned by Strathcona are subsets of the code provided to the repository.

4.2 Client: Determining Structural Context

Strathcona relies entirely on the structure of the code being edited by the developer to form a query to the server. The developer requests explicitly examples for a class (C), a method (m), or a field declaration (f). Strathcona parses the source file containing the structural element (C, m or f) and extracts: the (containing) class, the parent class and interfaces of C, the types of fields in C, and calls from m (if m is the requesting target). The precise information used in the query depends on the type of the query; for example the calls from m are only extracted if m is queried.[3] The context extractor uses the Eclipse Java parser, which can tolerate several kinds of programmatic errors. Once extracted, Strathcona forwards the structural context description to the server.

4.3 Server: Matching Structure

When a query containing the structural context description arrives at the server, the server attempts to find structurally similar code in the repository. Strathcona does not attempt to match the structure exactly as this would imply that the precise problem facing the developer, as expressed in application-specific types, exists in the repository. Instead, Strathcona uses a set of heuristics to find relevant parts of the applications to return as examples.

4.4 Structure Matching Heuristics

Strathcona incorporates six heuristics to match a structural context description to the code stored in the repository. Each heuristic relies on different kinds of structural information, and each produces potentially different examples. When a request arrives at the server, all of the heuristics are used to generate examples, and the ten "best" examples are then chosen to be returned to the developer.[4] Currently, Strathcona defines the "best" examples as those that occur most frequently in the set generated from applying all of the heuristics.[5]

We iteratively developed the six heuristics using the source code of several existing third-party plug-ins written for the Eclipse framework. We posited a heuristic, took the source code for the existing plug-ins, deleted sections that used the Eclipse framework, and tested the heuristic to see if any of the returned results would have helped to fill in the code that we had deleted. Through this process, we refined the heuristics to be as simple as possible. We describe each of these heuristics below and discuss why we chose them in Section 6.4.

4.4.1 INHERITANCE *Heuristic*

This heuristic matches on the parents and types of fields of C. Strathcona queries the repository to determine the set of classes C_r, that have the same direct parents (superclasses and interfaces) as C. Strathcona then orders the classes in C_r by the number of matching parents. When two or more classes in C_r match the same number of parents, we query

[3]We ignore any calls or references to the Java library because including these calls shifts the focus away from the framework of interest.

[4]Not every heuristic applies for each request. For example, the CALLS and USES heuristics do not match any examples when there are not any methods declared in the context.

[5]Each heuristic returns its top 20 examples and from these six sets of 20 the "best" are selected

the repository to determine how many of the types of fields in C match the types of fields in the classes from C_r and order the results based on which examples match the most field types. This heuristic was developed for situations when the developer knows which hot spot [7] in the framework to extend or implement, but does not know how to use the hot spot. This heuristic does not rely on any method-level context information.

4.4.2 CALLS *Heuristics*

The CALL heuristics are based on the targets of the calls made from m. In comparison to the INHERITANCE heuristic, the developer must provide more information about how they intend to use the framework. There are three CALLS heuristics.

1. The basic CALLS heuristic returns methods in the repository, the set M_r, that call the same targets as m. To match, a call target must match in both type and method name; we do not consider the inheritance hierarchy in matching call targets. The returned methods are ordered by the number of matched call targets.

   This heuristic sometimes returns large methods that make a number of calls, many of which are not relevant. These large methods are not useful to the developer as it is difficult to extract the portion of the method of interest. This led to the development of the CALLS BEST FIT heuristic.

2. The CALLS BEST FIT heuristic selects from M_r methods with the best ratio of matched to unmatched call targets. This heuristic returns methods only where the ratio of matched to total number of call targets is greater than a threshold (currently 0.4) that we devised through trial and error.

3. The CALLS WITH INHERITANCE heuristic uses more information about the context of m to select potentially useful methods from M_r. The methods this heuristic selects from M_r are those whose containing class share at least one parent with C.

4.4.3 USES *Heuristics*

The USES heuristics are based on the types a developer declares and uses in a method. These heuristics do not require the developer to know specific call targets. There are two USES heuristics.

1. The basic USES heuristic determines the types of the objects referred to by m and finds the set of methods U_r that use the same types. The methods in U_r are ordered by the most number of matches. This heuristic is effective in two cases: when the developer knows which type contains methods of interest (as in the scenario in Section 2), and when the developer has stumbled across the right type, but is using it incorrectly. The USES heuristic frequently returns large sets of examples and should not be considered unless other heuristics also match examples or the other heuristics do not match any examples at all.

2. The USES WITH INHERITANCE heuristic applies more information about the context of m (similar to the CALLS WITH INHERITANCE heuristic) to select potentially useful methods from U_r. The methods selected by this heuristic from U_r are those whose containing class share at least one parent with C.

4.5 Example Presentation

After the heuristics locate related code in the repository Strathcona transforms the code into examples. Strathcona determines how each class returned by a heuristic is related to the structural context of the client and builds a structural description of its use from its collaborating classes and interfaces. This structural description, the code for the class, and the rationale for its selection form the example returned to the client. On the client, the structural description is presented to the user using a limited UML-like class diagram notation (Figures 1(a), 3). This notation presents the classes and interfaces the code extends or implements, and any methods that call or use types of interest. The view does not include any call or usage relationships between the types. When a user requests the rationale for an example, it is presented as a list where each entry includes one of the four reasons for the inclusion of a particular element in the example: class has parent of type, class has field of type, method calls method, or method uses type. The code for the focus class of the example can be viewed so that the developer can investigate relevant portions.

The developer can navigate the returned examples using next and previous buttons. On the status line, Strathcona shows how many times an example has been viewed, and whether or not the developer has viewed the example previously.

4.6 Performance

To support our evaluation of Strathcona and to provide initial experience with its scalability, we populated the repository with the source for all of the Eclipse integrated development environment (Eclipse 3.0 M8). Table 1 summarizes the amount of information in the repository.

Table 1: Number of Structural Relations

Classes	17,456
Methods	124,359
Fields	48,441
Inheritance Relations	15,187
Object Instantions	43,923
Calls Relations	1,066,838
Total	1,316,204

Even with our unoptimized prototype, our approach is scaling well. Building a structural context description is fast, typically taking less than 500ms. Displaying the returned examples is also fast, taking less than 300ms. The average response time for our server[6] on a variety of different example requests is between 4 and 12 seconds. We feel that this is a reasonable delay for developers who are stuck. However, a faster response time would likely aid adoption of the system. Currently, Strathcona runs all of the heuristics on each developer request and combines the results. Further analysis may allow us to determine a priori which heuristics

[6]The server processing the queries was a Pentium 3 800 MHz machine with 1024 MB RAM, and the workstation housing the database was a Pentium 3 1000 MHz machine with 256 MB RAM. Strathcona uses the Postgresql database server to manage the structural database.

would be most effective, allowing us to increase the efficiency of access to the database.

5. EVALUATION

Earlier in the paper, we argued qualitatively that our approach requires less effort on the part of a developer to set-up and query an example repository than existing techniques. However, the overall question remains of whether our structural matching heuristics can produce examples that are helpful to a developer. To evaluate this question, we performed a case study in which we asked two developers to complete four programming tasks that use separate parts of the Eclipse framework: three of these tasks came from our experience developing Strathcona.

5.1 Setup

Two developers (subjects) were asked to complete four programming tasks, each related to building a plug-in using the Eclipse framework. Each of the developers had some plug-in programming experience. Subject 1 had less than one month of Eclipse plug-in programming experience but more than eight years of Java experience. Subject 2 had over six months of Eclipse plug-in programming experience but only eighteen months of experience with Java.

For each task, the subjects were provided a simple description of the task, and a method skeleton within which they could develop their solution. The skeletons were populated with code the developer would likely write to accomplish their task. The code consisted of method calls on types and references to methods on interfaces. These methods and types were identified using the Eclipse documentation and code completion features.

Neither subject knew how to implement any of the assigned tasks. The standard Eclipse Java development tools were available to the subjects as they worked on each task. The tasks were completed in the same order by each subject. A short one page document describing how to use Strathcona and the four tasks assigned was presented to each subject at the start of the exercise. Each subject was given a maximum of three hours to complete the four tasks.

5.2 Results

Table 2 summarizes the results. For each task, we list how many of the ten examples returned by Strathcona the two subjects deemed useful for the task through their direct use of code snippets, the number of examples for which the subjects viewed the source, and whether or not the subject was successful at completing the task. For each task there may have been additional relevant examples but we indicate only the ones that were used by the subjects.

5.2.1 *Task 1: Update Status Line*

The first task involved displaying text in the status line of Eclipse as described in Section 2. This task is conceptually simple, but requires a chain of method calls accessing objects of a variety of types, including `IViewPart`, `IViewSite`, `IActionBars` and `IStatusLineManager`. The subjects were given the same skeleton code as described in Section 2. Each subject found the first example returned useful to complete the task. Both subjects copied code from the example into their editor, changed the variable name for the `setMessage` method and ran the code to test it. The example used was returned by all but the basic USES heuristic.

5.2.2 *Task 2: Create AST*

This task involved building an Abstract Syntax Tree (AST) from a source string. A search of the Eclipse documentation indicated that `ASTParser.setSource(String)` should provide the appropriate functionality. However, three things are required: a factory is needed to create the parser, the parser needs to have access to the appropriate source code, and the AST needs to be generated. The provided skeleton code is shown in Figure 2(a). Both subjects again selected the first example returned (Figure 3) as it demonstrated the use of the method of interest and included code to setup the parser and create the AST (Figure 2(b)). The example they selected was returned by the CALLS, CALLS BEST FIT, and the USES heuristics. The second subject investigated the code snippets for a number of examples before deciding that the first one was the most relevant to the task. As for the first task, both subjects integrated code from the example into their source to complete the task. The code snippet provided with the example contained two extraneous calls that the subjects dealt with differently; one copied all of the code and deleted the extraneous sections in his code while the other subject only copied the sections of code which were relevant (this subject studied the documentation for each method call in the code snippet to figure out what it did before it was copied).

```
a) private void createASTFromSource(String source) {
  ASTParser.setSource(source.toCharArray());

}

b) private CompilationUnit parseCompilationUnit(char[]
       source, String unitName, IJavaProject project) {
  ASTParser parser= ASTParser.newParser(AST.LEVEL_2_0);
  parser.setSource(source);
  parser.setUnitName(unitName);
  parser.setProject(project);
  parser.setResolveBindings(true);
  return (CompilationUnit) parser.createAST(null);
}
```

Figure 2: Task 2 Seed(a), Snippet(b)

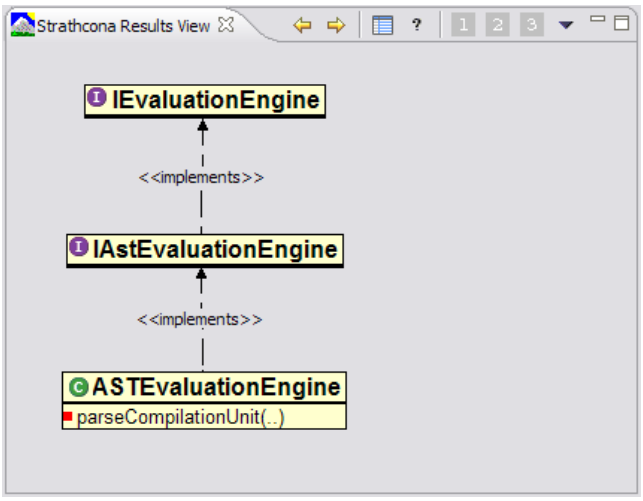

Figure 3: Task 2 UML Representation

Table 2: Results from Evaluation

	Useful Example	Source Viewed	Succeeded at Task
Task 1			
Subject 1	1	1	yes
Subject 2	1	1	yes
Task 2			
Subject 1	1	2	yes
Subject 2	1	6	yes
Task 3			
Subject 1	0	2	yes
Subject 2	0	6	yes
Task 4			
Subject 1	1	2	yes
Subject 2	0	7	partially

5.2.3 Task 3: Highlight Text

This task involved highlighting instances of method invocations in a code viewer using the backend AST representation generated in the second task. Strathcona was not able to return any useful examples for this task. We deliberately included this task to determine if subjects could identify when the returned examples were insufficient. If the subjects are able to recognize unhelpful examples, we have more confidence in their assessment of returned examples.

```
private void hilightRegions(Vector regions) {
  StyleRange[] srs = new StyleRange[regions.size()];

  aViewer.getTextWidget().setStyleRanges(srs);
}
```

Figure 4: Task 3 Seed

Figure 4 shows the skeleton code for this task. The subjects both stopped examining the examples provided within 15 minutes and implemented the feature using the standard IDE tools. Interestingly, both subjects independently decided to use Strathcona as part of this task to find an example of how to create a SWT `Color` object; both subjects used some code from the returned examples to accomplish this portion of the task.

5.2.4 Task 4: Generate Method Signature

This task was the most complex. Using the `ASTVisitor`, the subjects were asked to extract the method signatures for each method in the AST. This task required the use of several Eclipse framework types including `Type, PrimitiveType, ArrayType, Name, SimpleName, QualifiedName, Code, Flags, and SingleVariableDeclaration`. Figure 5 shows some of the most obvious method calls on `MethodDeclaration` that would be needed to complete the task. Subject 1 investigated `MethodDeclaration` using the Eclipse code completion feature to try to derive a working solution before using Strathcona as he was concerned about not finding a relevant example as was the case in task 3. Once he queried Strathcona, he examined the rationale for the first few of the examples carefully before deciding which two examples to investigate. The first two examples returned for this task match four method calls in the code used to query; the remaining examples matched at most two method calls. After investigating the source from the second example, the subject discarded it and moved on to the first example. This example matched several calls as shown by its rationale for selection (Table 3). When viewing the code, this example had one 56 line method highlighted; this method also used several private utility methods that totaled 61 lines of code. He proceeded to copy code from the example in small sections, and completed the task successfully. The example he selected to complete the task was returned by all but the basic USES heuristic. Subject two mistakenly queried the repository on the wrong method and searched through several irrelevant source files before deciding to implement the feature manually. She was partially successful but was unable to extract some parts of the signature from the AST.

```
public boolean visit(MethodDeclaration node) {
   node.getModifiers();
   node.getName().getIdentifier();
   node.parameters();

   return super.visit(node);
}
```

Figure 5: Task 4 Seed

5.3 Summary

Subject one completed all four tasks successfully, finding relevant examples in all three cases for which appropriate examples were returned; subject two completed three out of four tasks, finding relevant examples in two of the possible three cases. In each of the tasks where the subject found a relevant example, source code was copied from the example into the task code. These results show that our tool can deliver relevant and useful examples to developers. They also show a developer can determine when the examples returned are not relevant.

By focusing on a pre-existing framework, by considering cases that have occurred in our own experience, and by using subjects with some but not extensive knowledge of Eclipse, we have focused on the realistic use of a large framework. Since our heuristics rely on structural relationships available in most popular object-oriented languages, we believe that our results will generalize to other frameworks written in other languages. However, further testing is required to determine the applicability of our heuristics to other frameworks and to users more experienced with the framework of interest.

Table 3: Task 4 Rationale

Class has parent of type	org.eclipse.jdt.core.dom.ASTVisitor
Method Calls Target Method	org.eclipse.jdt.core.dom.MethodDeclaration.getName()
Method Calls Target Method	org.eclipse.jdt.core.dom.MethodDeclaration.parameters()
Method Calls Target Method	org.eclipse.jdt.core.dom.SimpleName.getIdentifier()
Method Calls Target Method	org.eclipse.jdt.core.dom.BodyDeclaration.getModifiers()
Class uses Class	org.eclipse.jdt.core.dom.MethodDeclaration
Class uses Class	org.eclipse.jdt.core.dom.SimpleName
Class uses Class	org.eclipse.jdt.core.dom.BodyDeclaration

6. DISCUSSION

We have shown that Strathcona can return relevant code examples to developers using a framework, and that developers can recognize the relevant examples. In this section, we discuss possible pitfalls and limitations of our approach, describe heuristics that we did not find useful, and consider the broader applicability of the approach.

6.1 Examples: Good or Bad?

It may be that the provision of examples to a developer leads to worse code than when examples are not provided. Rosson and Carroll showed, in a study of developers using a Smalltalk framework [12], that developers frequently copied and integrated snippets of code without trying to understand exactly how they worked and executed the resultant code to see the effects of the snippets. Rosson and Carroll call this *debugging into existence*. The developers in our study behaved analogously. As noted by Rosson and Carroll, one potential problem with this strategy is that because simple examples require the least analysis, developers may not have a firm grasp of the different contexts in which a snippet can be used. By returning multiple examples and the rationale for their selection, we hope to alleviate this potential problem and provide the developer with examples for multiple contexts.

Providing examples does have some positive benefits. The use of examples can reduce the amount of typing required to complete a task, or ensure that the details of the code are correct [12]. Anecdotally, we observed that in some cases the presence of an example meant that the code developed was more complete than if it had been written from scratch. For instance, during the fourth task, the developer who successfully completed the assignment, copied some code that checked for array types and added the appropriate notations to the method signatures without knowing what the code did, resulting in a case being taken into account that the developer had not considered. By leveraging the work done by other developers in the past, this developer was able to complete the task with higher quality than if the developer had been working alone.

6.2 Heuristic Performance

To date, our focus has been on the utility of our overall approach: whether structural similarity can be used to return useful examples. In a pilot to the study reported in Section 5, we attempted a more quantitative evaluation of the performance of our heuristics. In the pilot, we asked a developer to rate the examples returned by Strathcona for the four tasks. We found that the developer was unable to provide such a rating because the developer could not assess the value of an example until trying to use it to complete the task [5]. However, completing the task with one example made it impossible to rate the next example given the information learned from completing the task. It may be possible to study the quantitative performance of the heuristics through a larger study in which developers are provided examples from only one kind of heuristic for the same set of tasks; the value of the examples might then be assessed across the set of developers. We have left this more subtle experimentation for future work.

6.3 Missing Examples

We devised our study to include a case in which our tool could not find a useful example in the repository. Choosing to use a large framework for experimentation makes it impossible for us to know if a suitable example exists within the repository that our heuristics were unable to find, although manual searches of the repository also failed to find any relevant examples. Further use of the tool is needed to determine if our heuristics need to be augmented, or combined with other techniques, to improve the finding of examples.

The ability of Strathcona to return useful examples is also dependent upon the quality of the seed code used in a query. If a developer does not have any idea of how to achieve a desired effect with a framework and as such cannot find a seed, or if the developer is on the wrong track and the seed code is incorrect, Strathcona will likely not provide relevant examples. In most cases, we have found it possible to use the documentation to find an appropriate seed. Strathcona fills in the details of how to complete a task that the documentation lacks.

6.4 Heuristic Refinement

We developed the heuristics embedded in Strathcona iteratively as described in Section 4. The final version of the heuristics in Strathcona do not include a number of the approaches we tried but that we did not find useful. We briefly describe the failed approaches.

Example Scoring We tried to develop a scoring system that would assign different values for the different kinds of structural similarity but were unable to find an approach that did not lose general applicability. Our scoring approaches tended to work for one style of code seed but not others. We found that the styles of the seeds differed depending on the stage of development of the code and whether or not the developer had identified reasonable hot spots in the framework from which to begin the task.

Object Instantiations Our heuristics do not treat object instantiations specially; they are treated only as calls to a constructor. We did not find that heuristics that treated these instantiations specially were useful. One reason may be variability in Eclipse as to whether clients or servers in-

stantiate objects. For example, whenever Factory classes are involved the client does not instantiate objects but gets new objects delivered to them by framework objects.

Hierarchies The heuristics do not transitively check the object hierarchy when considering parents, uses, or calls relations. In our initial investigation, we found that these additional targets did not increase the effectiveness of our heuristics and often created much larger, and less relevant, examples. Exploring the inheritance hierarchy more thoroughly may be useful for cases in which the structural context does not map directly to any examples. By not considering the call hierarchy we also potentially miss examples that are split across method boundaries.

6.5 Presenting Examples

We chose to use a compact visual notation to present an example to make it easier for a developer to select which examples to peruse in more detail. This visual notation places a heavy emphasis on inheritance. This additional information about types used and methods called, which is available in the rationale view, may also be useful in the visual representation. The developers in our evaluation used the visual notation we provided to discard examples but always checked the rationale view before deciding if an example should be examined in more detail.

The notation we chose for presenting examples, based on a class diagram, focuses on the structural similarity to the problem at hand. As a developer will often want to know how the objects participating in the example interact, it may be useful to replace or supplement this view with a visualization based on UML collaboration or sequence diagrams.

7. CONCLUSION

The documentation for a framework is typically insufficient to describe all of the ways in which a framework may be used. One way to help a developer use a framework effectively may be to provide a repository of examples. Existing software example repository approaches have two limitations: they impose a large burden on the developer when querying the repository, and building the repository requires either carefully constructed examples or well-commented code.

The approach we describe in this paper overcomes these limitations by locating examples in the repository based on similarity of structure to the code a developer is writing. Examples from the repository are generated automatically from the matched code. Queries are easy to make in this approach because they are formed automatically from the structural information that must appear in the developer's code. For the same reason, the repository can be easily formed automatically. A case study was used to show that structural similarity can be used to deliver helpful examples to developers.

This paper makes two contributions. First, it shows the utility of structural similarity as a basis for a software example repository. Second, it provides an initial set of heuristics to use to determine structural similarity. Further development of the heuristics, possibly using ideas from code clone detection [1], and more extensive testing of the heuristics on other frameworks, is needed.

Acknowledgments

This research was funded, in part by the CSER, IBM and NSERC. We would like to thank the subjects who participated in our study. We would also like to thank Miryung Kim, John Anvik and Andrew Eisenberg for their comments.

8. REFERENCES

[1] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of Int'l Conf. on Soft. Maintenance*, pages 368–377, 1998.

[2] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. of the 25th Int'l Conf. on Software Engineering*, pages 408–418, 2003.

[3] S. Henninger. Retrieving software objects in an example-based programming environment. In *Proc. of the 14th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 251–260, 1991.

[4] R. Hill and J. Rideout. Automatic method completion. In *Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering*, pages 228–235, 2004.

[5] R. Holmes. Using structural context to recommend source code examples. Master's thesis, University of British Columbia, 2004.

[6] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *Proc. of the 25th Int'l Conf. on Software Engineering*, pages 14–24, 2003.

[7] R. E. Johnson. Documenting frameworks using patterns. In *Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 63–72, 1992.

[8] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. of the 22nd Int'l Conf. on Software Engineering*, pages 167–176, 2000.

[9] A. Michail. Code web: Data mining library reuse patterns. In *Proc. of the 23rd Int'l Conf. on Software Engineering*, pages 827–828. IEEE Computer Society, 2001.

[10] L. R. Neal. A system for example-based programming. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 63–68. ACM Press, 1989.

[11] E. Rissland. Examples and learning systems. In *Adaptive Control of Ill-Defined Systems*. Plenum, 1983.

[12] M. B. Rosson and J. M. Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3):219–253, 1996.

[13] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proc. of the 24th Int'l Conf. on Software Engineering*, pages 513–523, 2002.

[14] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *Foundations of Software Engineering*, pages 60–68, 2000.