Using Structural Context to Recommend Source Code Examples

by

Reid Holmes

B.Sc., University of British Columbia, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

August 2004

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 thesis, 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.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor Gail Murphy for introducing me to the world of research while I was an undergraduate, and giving me guidance over the past four years.

I must also thank both Steph Durocher and Matt Brown for providing me with opportunities in which I could focus my thoughts, brainstorm about my thesis, and of course, climb some mountains.

I must also thank Jacek Kisynski, Chris Gray, Andrew Eisenberg, and Doug Janzen for keeping the past two years entertaining and putting up with me.

Reid Holmes

The University of British Columbia
August 2004

To Laura, for making life exciting.

Chapter 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. All too often, some usage constraints remain unstated resulting in a developer becoming stuck when trying to use the framework.

To help unstick developers, several researchers have advocated establishing an example repository that houses a larger set of examples of a framework's use (e.g. [13, 14, 18]). These approaches vary in the means used to retrieve relevant examples from the repository: a developer must either learn a new query language [7], have an idea of what type of example would likely help them finish their current task [11], or write their source in a style that conforms to that of the example repository [17].

The thesis of this research is that the software structure of the code a developer is currently working on can be used to find relevant example code automatically from a software repository. Our approach has two advantages over existing proposals for easing framework use.

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, for instance, through a keystroke. The developer need not learn a new query language, nor must the developer code to particular standards, such as commenting conventions, 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 appropriate examples.

To help investigate this approach, we built the Strathcona tool. The client portion of this tool, which is a plugin for the Eclipse integrated development environment,[1] extracts the structural context of the code a developer is working on when the developer issues a request. 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 consists of a set of classes and relevant relationships; an example is a subset of one of the applications used to populate the repository. The client uses a compact visual representation to present the structure of returned examples to the developer. A rationale for the selection of the examples is also provided.

The key question of interest in evaluating our approach was whether the structural matching heuristics encoded in Strathcona can return useful examples to a developer. We chose to perform a qualitative evaluation in which two subjects replicated four cases; each case consisted of a programming task related to writing plugins for Eclipse. For this evaluation, we populated the Strathcona repository with the source for the Eclipse environment, which comprises approximately 1.5MLOC. This evaluation style is possible because Eclipse is an open-source system, and because the Eclipse environment consists of a set of plugins

[1]`eclipse.org`

that collectively comprise the Eclipse framework. In all but one instance in which there were relevant examples in the repository, the subjects were able to access the relevant examples, understand the examples, and complete the programming task, providing initial evidence that structural matching is appropriate to help ease framework use.

We begin the thesis with three scenarios describing the tool's use (Chapter 2). Next, we compare our approach to other efforts (Chapter 3), describe our approach and tool in detail (Chapter 4), and present our evaluation (Chapter 5). We conclude the thesis with a discussion of open issues (Chapter 6). We follow with a short description of future work in Chapter 7 before summarizing (Chapter 8).

Chapter 2

Sample Scenarios

Developers who are new to a framework sometimes find themselves overwhelmed when they try to accomplish development tasks [2]. One way to alleviate the difficulty of learning how to use the framework is to provide documentation. However, it is difficult to provide documentation that covers all cases of framework use. Our Strathcona tool attempts to help a developer for whom the documentation is insufficient. To provide an overview of how Strathcona can help a developer, we describe three scenarios of possible use.

The first scenario demonstrates a situation in which a simple task, which requires a good understanding of the framework and its internal parts, becomes difficult because of a lack of documentation. This scenario is also the first task which we asked the developers to accomplish in our evaluation of Strathcona (Chapter 5).

The second scenario highlights a situation where documentation is available for a task but the developer wants some additional examples to ensure that they have a solid understanding of how their solution interacts with the framework.

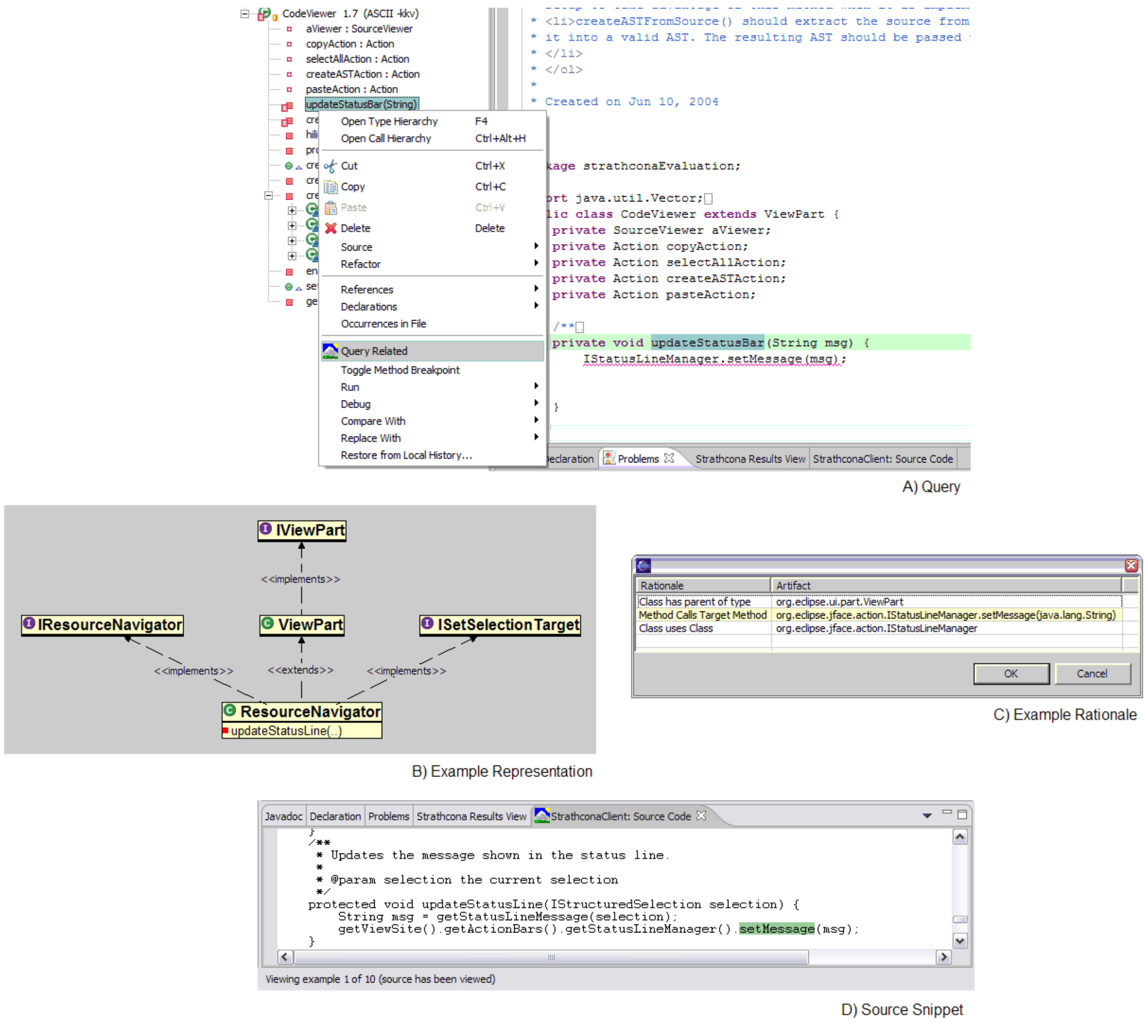
The third scenario shows how, from even a minimal start on a difficult task, Strathcona can direct a developer to an example which can help with a complicated task. This scenario is the fourth task of our evaluation.

2.1 Scenario 1

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 the tree view in Eclipse, the status line shows number of selected items. Consider a developer who, when writing a plugin for Eclipse, wants to update this status line with a new message. The first place a developer might look to determine how to accomplish this task is in the Eclipse documentation. Checking this resource, the developer finds a reference to an interface called `IStatusLineManager`. The API documentation for `IStatusLineManager` provides a method, named`setMessage(String)`, that sounds appropriate, but the documentation does not describe how to get a handle to a `StatusLineManager` object needed to call this method. The developer—in this case myself when writing the Strathcona plugin—becomes stuck.

The Strathcona tool can help the developer become unstuck. The developer adds into a method, say `updateStatusLine`, within the source code for their plugin a temporary statement `IStatusLineManager.setMessage(String)`[1]. From Eclipse's package explorer, which shows the structure of the code in a file, the developer right clicks on the icon for `updateStatusLine` and requests `Query Related` as shown in Figure 2.1(a). The client portion of Strathcona generates a structural context of the code a developer is writing. This context comprises details of the class that contains the method including its field types, the types of its parents, and the calls from the method of interest, `updateStatusLine`. In this case, the context includes the parent of the class containing `updateStatusLine`, namely `ViewPart`, a field of type `CodeViewer`, and the method `updateStatusLine` that contains a call to `IStatusLineManager.setMessage(String)`. This context is sent to the server portion of Strathcona which returns ten examples that are structurally related to the code

[1]While this snippet does not compile our system can tolerate incomplete fragments (Chapter 4.2).



```
* <li>createASTFromSource() should extract the source from
* it into a valid AST. The resulting AST should be passed
* </li>
* </ol>
*
* Created on Jun 10, 2004

kage strathconaEvaluation;

ort java.util.Vector;
lic class CodeViewer extends ViewPart {
 private SourceViewer aViewer;
 private Action copyAction;
 private Action selectAllAction;
 private Action createASTAction;
 private Action pasteAction;

 /**
 private void updateStatusBar(String msg) {
     IStatusLineManager.setMessage(msg);


 }
```

Rationale	Artifact
Class has parent of type	org.eclipse.ui.part.ViewPart
Method Calls Target Method	org.eclipse.jface.action.IStatusLineManager.setMessage(java.lang.String)
Class uses Class	org.eclipse.jface.action.IStatusLineManager

```
}
/**
 * Updates the message shown in the status line.
 *
 * @param selection the current selection
 */
protected void updateStatusLine(IStructuredSelection selection) {
    String msg = getStatusLineMessage(selection);
    getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
}
```

Viewing example 1 of 10 (source has been viewed)

Figure 2.1: *Strathcona Cycle*

being written. Each of these examples consist of three parts: a structural description, a code snippet, and a rationale explaining the relevance of the code snippet to the problem the developer is facing.

Figure 2.1(b) shows the structure of one of the returned examples, which relates to the code for the resource navigator within Eclipse. The rationale returned with the example describes that this example code snippet was selected because the `setMessage(String)` method is being called, `IStatusLineManager` is being used by the example, and the example extends `ViewPart` (Figure 2.1(c)). The developer asks to view the code, and Strathcona highlights the call chain `getViewSite().getActionBars().getStatusLineManager().setMessage(message);` as shown in Figure 2.1(d). This call chain extracts an `IViewSite` from `ViewPart` and uses the `IViewSite` to get a handle to `IActionBars` which then can be used to get a handle to the object which implements `IStatusLineManager`. The developer attempts to solve their task by copying this sequence of calls into `updateStatusLine`, and changing the name passed as an argument. This code completes the task.

Although this scenario describes a conceptually simple task of updating a status line, some level of knowledge about the interaction between several types, `ViewPart, IViewSite, IActionBars, IStatusLineManager`, is needed. The interactions are not described in the provided Eclipse documentation. Although the interactions can be discovered manually through the code completion features in Eclipse, discovering the correct sequence of calls is difficult as there are 79 methods available to be called across the three classes.

2.2 Scenario 2

Version 3.0 of Eclipse changes the way background processes are managed in the system. The `Job` class represents units of runnable work that can be scheduled to run in the background. A developer is given the task of migrating a legacy plugin to Eclipse 3.0. From the documentation, the developer determines that the `Job` class must be extended to get their tool to run in the background. The Eclipse help documentation provides an example of how to extend this class. The only abstract method in `Job` is `run(IProgressMonitor monitor)`. However, the developer is confused about how `ProgressMonitors` interact with `Jobs`.

To help understand this interaction, the developer copies source from the example provided by the Eclipse documentation into `run(IProgressMonitor)` (provided by extending

`Job`) and queries Strathcona. Strathcona then provides the developer with 10 examples of how other plugins have used the Job infrastructure to manipulate progress monitors. With these examples the developer is able to implement their `Job` and properly instantiate its `IProgressMonitor`.

2.3 Scenario 3

The Eclipse framework includes many packages specialized towards Java development, including one for manipulating the code as an abstract syntax tree (AST). Consider a developer trying to extract the method signatures for all of the methods in the code from the `AST`. By looking at the Java documentation (such as javadoc), the developer can see that `MethodDeclaration` objects in the AST represent the information of interest. By implementing an `ASTVisitor` the developer can visit every `MethodDeclaration` node in the tree. There is one example in the documentation about generating a `MethodDeclaration` but no examples of how to extract information from one.

Using the code completion features of Eclipse, the developer quickly identifies several methods of interest, including `getName(), getReturnType(), and parameters()`. The developer adds these method calls to the editor for which the solution to the task is being written and, after examining the Eclipse help documentation and finding it insufficient for this task, queries Strathcona.

Strathcona returns ten examples that use the methods listed by the developer as being of interest. Five of the examples have an `ASTVisitor` as a parent class, call one of the methods of interest, and use the same types as the methods of interest. The remaining five examples match various amounts of code from the developers editor. The snippet from first example returned is for a method called `buildMethodDeclaration(MethodDeclaration)` which returns a `StringBuffer`. This method, and the private methods it uses, comprise approximately 120 lines of code. The developer starts copying at the beginning of the method, copying private methods as the need arises. Some parts of the code in the snippet do not apply to this task as the snippet also examines the body of the method; the developer skips these sections of the code. The developer executes the new code and checks to ensure that the task has been successfully accomplished. Not only does the code work, but the snippet took into consideration cases where the method declaration represents a constructor as well as when parameters to a method is an array. By copying code from the provided snippet, the developer ends up with an implementation that is likely more robust than one which they may have generated manually.

Chapter 3

Related Work

3.1 Framework Documentation Approaches

Several researchers have suggested forms of documentation to ease the use of frameworks (e.g., [1, 6, 9]). A major drawback of this approach is the effort required to document the framework for the multitude of ways in which a large framework may be used. Instead, others have suggested encoding information about the intended use of the framework within the framework itself such as Hooks [5]. Hooks identify constraints that must be followed by the developer, design changes that are required in the developers application, and the effect that using a hook will have on the framework itself. However, these hooks must be defined manually by the framework developer.

One way to overcome the need to write and access documentation, whether external or internal to the framework, is to provide hints to the developer based on the structure of the code comprising the framework, as is available through the code completion features of most modern development environments. Code completion helps a developer complete a programming statement in an editor, for instance, by analyzing the structure of a type being accessed and offering the possible operations that can be called on that type. Using completion can make it easier to access the classes and methods of a framework. However, code completion typically uses a narrow context, namely a programming statement, as the source of information from which to search; as a result, the approach can return only small code snippets. In contrast we use as a larger context, such as the class containing a method and its parents, to provide larger code snippets that include more information than the next available command.

3.2 Software Example Systems

Our claim in this thesis 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 [18, 17]. CodeBroker queries a repository automatically after each comment or method signature a developer writes. The queries made to the repository are based on the 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 [18]. When this approach is taken and when appropriate technical synonyms in the comments are incorporated, CodeBroker may be able to match a more diverse set of examples than Strathcona. However the effectiveness of this approach may be limited because of the difficulty of writing appropriate comments. In comparison, our approach can apply to 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 [7]. 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 CodeFinder, Strathcona aims to remove the step of formulating the query by creating the query automatically for the developer.

Other tools, such as Component Rank [8] and CodeWeb [11, 12], use software structure to determine which parts of frameworks are frequently used. Of these CodeWeb 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 and other, not 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 [16]. The RVM tool provides an animation of different scenarios of use of the class under investigation. 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 Hipikat tool [4] 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.

Our approach of using program snippets as a query language is similar to that of Java Template Miner [19], which allows developers to either use a keyword search or a program template, similar to Strathcona's snippets, to query the repository. However, the searching mechanism used by the Template Miner is based primarily on keywords automatically mined out of the source when it is indexed. The templates provided as queries are also mined for key words and these are matched to find similar examples. Thus, the tool provides a limited form of signature matching as in [20]. However, the Java Template Miner does not try to extract type information from the source as Strathcona does, and the Java Template Miner does not consider types other than Java primitive types.

3.3 Program Databases

The back end of the system supporting Strathcona is essentially a program database (Chapter 4.7.1). There are many similarities between the infrastructure supporting Strathcona and Chen's CIA [3]. Chen identifies several traits for a program database, including separating data extraction from presentation, keeping only object interaction relationships in the database, and keeping the source files separate from the database. We utilized all three of these ideas keeping the database small and flexible without limiting any of our analysis.

Chapter 4

Example Recommender

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 Chapter 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 (Chapter 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 (Chapter 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 (Chapter 4.5). We conclude the Chapter with a discussion of a performance of Strathcona (Chapter 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 can request related 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.[1] 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 structural matches in the repository. Strathcona does not attempt *exact* structural matches as this would imply the precise problem facing the developer, as

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

expressed in application-specific types, has already been solved by an application in the repository. Instead, Strathcona uses a set of heuristics to find relevant parts of the applications to return as examples to the client.

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.[2] Currently, Strathcona defines the "best" examples as those that occur most frequently in the set generated from applying all of the heuristics.[3]

We developed the six heuristics iteratively 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 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 Chapter 6.3.

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 the

[2]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.

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

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 [9] 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 Chapter 2.1), 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 2.1(a), 5.2). 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 portions relevant to their current context.

The developer can navigate the presented examples using next and previous buttons. On the status line, Strathcona shows how many times an example has been seen, 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 4.1 summarizes the amount of information in the repository.

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

Table 4.1: *Number of Structural Relations*

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

4.7 Strathcona Implementation

The Strathcona back end, or application server, receives requests over HTTP from clients in XML format and returns examples in the same format to them. As each request is received it is unpacked, checked to determine the type, and processed by a `ServerDaemon` (See Figure 4.1). These requests can either be example requests (context documents) or source requests. Context documents are turned into a series of queries and sent to the heuristics engine while source requests are turned into a query requesting a single source file. The heuristics have been described in Chapter 4. These queries are then sent to the Database server for servicing. The purpose of this section is to describe the infrastructure used by the heuristics to generate examples.

[4]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.

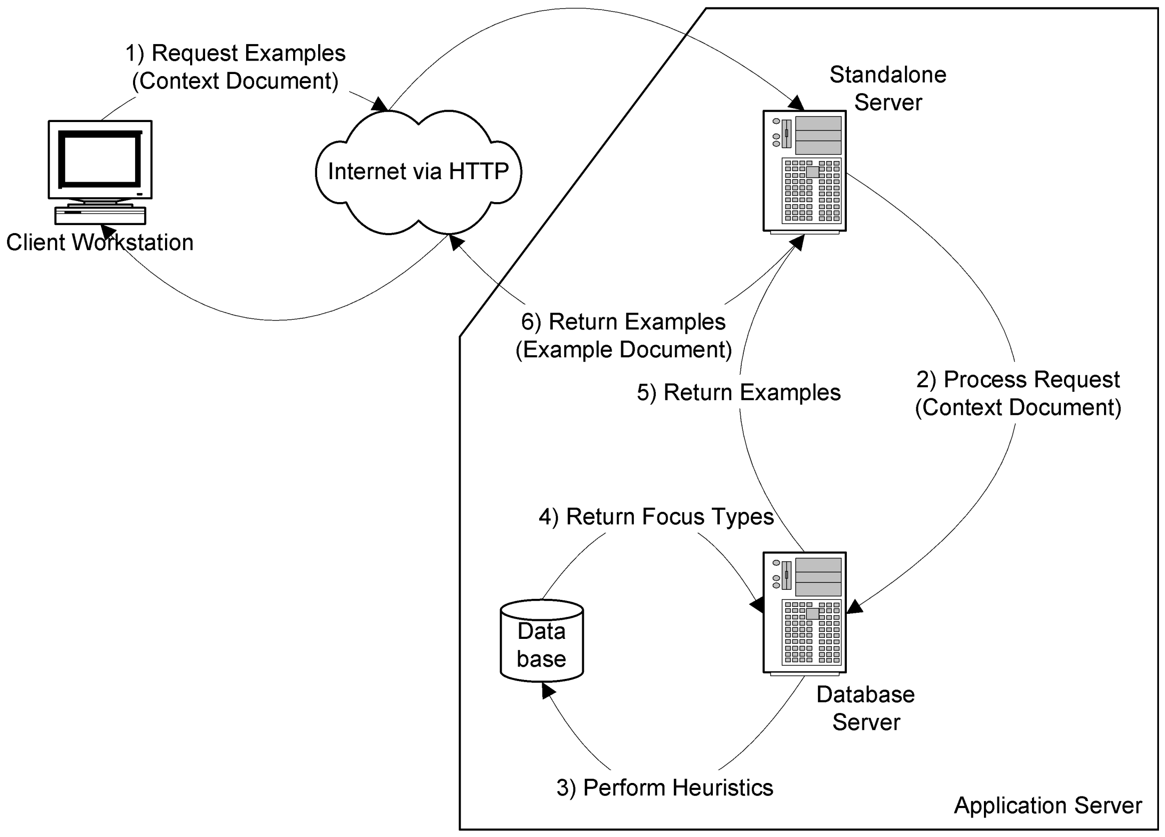


Figure 4.1: *Strathcona Architecture*

4.7.1 Database Server

The Strathcona database is comprised of eight tables and twenty-two indexes. The tables consist of ScopeModifier, Calls, Creates, ClassHierarchy, FieldTable, Methods, Class, and Sources. These tables are described below in Chapter 4.7.3 and in more detail in Appendix A. With the exceptions of ScopeModifier, Sources, and Creates each of the tables is used by each run of the heuristics. Only the structural information for the programs is stored in the tables, the source for the examples is stored on the file system of the database server.

The database is heavily optimised for read-only access because except for loading code into the repository, Strathcona does not write anything to the database. Even though the database can be populated online we found it most efficient to disable the backend server when inserting new programs.

4.7.2 Database Generation

A program extractor for Strathcona has been developed and implemented as an Eclipse plugin. The extractor is based upon the FEAT [15] system. The extractor crawls the program database of a project created by FEAT[5], extracts all relevant pieces of program structure, and stores them in a non-normalized XML file. The use of FEAT allows any program loaded in an Eclipse workbench to be loaded into Strathcona and added to the structural database (only two mouse clicks are required). The XML representation is then automatically added to the database. We store the structure of a project in the intermediate XML format so that programs can be added to the database in batch mode and so the database can be repopulated without analyzing the programs structure repeatedly. It should be noted that each XML file stores the full structure required for that program; for instance the structural information about the parents of each class or stored, even if those parents have been extracted for another program before. This approach allows complete program databases to be generated with any sets of XML files and avoids adding dependencies between the program representations. Extracting the program structure and populating the database is a wholly unoptimized aspect of Strathcona. It typically happens only once before the system is online. Extracting the source for Eclipse 3.0 M8 took one day while populating the database took another day.

4.7.3 The Tables

The `Class` table (Figure A.2) is at the heart of all of the heuristics implemented in Strathcona. This table is linked through foreign keys on `id` to `Method, FieldTable,` and

[5]A project in Eclipse is simply a collection of Java classes

`ClassHierarchy`. Although `Class` stores many attributes, most of them are unindexed as they are not searched upon by any of the heuristics. In fact `isAbstract, isStatic, and scope` are not used by any heuristics. As mentioned above, `ownedBy` links the class to its physical location on disk. However, in the current implementation of the Strathcona `ownedBy` also references the `Sources` table, although this link should be via a new field in a robust implementation. The `Class` table also stores all of the interfaces in the system. Special classes have been manually added for Java primitive types including void in order to support primitive field and method return types.

`ClassHierarchy` (Figure A.2) establishes the parent-child relationships between classes and interfaces. It stores two foreign keys, both to the `Class` table. Both `superClassId` and `subClassId` are indexed even though the heuristics in the system only look for similar parents.

`FieldTable` (Figure A.2) stores the fields for every class. The FieldTable table has an inconsistent name because Field is a reserved word for the Postgresql database. The `hostType` field in the table is a foreign key which references the class in that the field is defined.

The `Method` (Figure A.4) table stores all of the information about the methods in the database. The `parentType` field is a foreign key relating to the class in which the method is defined. The `overridesMethod` was added at a later iteration as a foreign key to record the method from which a method overrides, if any. `isAbstract, isStatic, scope, and returnType` are not currently used by any heuristics.

The `Calls` table (Figure A.4) plays an integral role in all of the calls and uses heuristics. Each tuple in this table simply links two methods together. Through these tuples it is easy to find either all of the calls made by a method or all of the calls made to a method. To find the uses relationships, all of the methods of a class are determined and from this the same query can be made on this set rather than on just a single method.

The `ScopeModifier` table (Figure A.1) stores the scopes possible `public, protected`, or `private` for the methods, classes, and fields in the repository. The Class, Method, and FieldTable tables all have foreign keys into the ScopeModifier table. Although we store this information, the final version of our heuristics do not make use of the information.

The `Sources` table (Figure A.8) tracks which projects (in our case Plugins) are in the repository as well as when each project was added, what version is indexed, and where on disk the source root for the classes is stored. This table is used to locate the source location on disk for each source file indexed by Strathcona. The other information in the table is only stored for accounting purposes — to keep track of what is actually stored in the repository at any point in time.

`Creates` is another table (Figure A.6) that is not currently used by any heuristic. The information in this table is redundant due to the fact that object instantiations are also stored in the `Calls` table as method calls on initializers. This table was created to improve the efficiency of heuristics that use object instantiations in their analysis; however in the end all of these heuristics were removed from the system.

Due to the fact that syntax highlighting cannot fully work without a whole project's context one more table, `SourceLocation` should be defined. This table would reference every method call, field access, and class reference source range in the system. It would track these elements as well as which files, starting character and stopping character for each of these artifacts. This would allow the proper syntax highlighting queues to be stored during the extraction process rather than trying to infer them during the query process.

4.7.4 Queries

To support the heuristically-based matching, Strathcona defines several internal queries on the database. These queries form two categories: those which can be pre compiled because they take a static number of parameters, and those which must be built on the fly because they take a variable number of arguments. The queries provided by Strathcona are listed in Appendix A.

4.7.5 Pre-Built Queries

The pre-built queries (Table B.1) are pre-compiled by the JDBC library so they can operate as quickly as possible. When the `ServerDaemon` is started, one instance of each of the queries is created and is reused throughout the lifecycle of the daemon. Many of these queries involve joins and subqueries which take advantage of the indexes in the database to allow them to execute more quickly. As of the time of this work sub query support was much more advanced in Postgresql compared to both Mysql and Hsqldb which led us to use this database for this work. These queries are listed in Table B.1.

4.7.6 On-the-Fly Queries

Each of these queries (Table B.2) take a variable number of arguments necessitating that they be built on-the-fly during query execution. These queries can be created through combinations of the above pre-built queries, however the on-the-fly queries were created for simplicity of execution after we had determined their value for the heuristics. The performance of these queries decreases as the number of arguments provided to them increases.

4.8 Example Generation

After the queries comprising a heuristic have completed, the result is a list of example types, or *focus types*. Before these focus types can be sent back to the client they must be populated with additional information so that the client has more information to act upon than a single type name. The rationale for why each example was selected, and by which heuristics it was selected must also be gathered and added to the example. We do not report the information about which heuristic selected an example to the developer; we use this information in conjunction with user feedback to see which heuristics are the most effective.

4.8.1 Populating the Example

In addition to the information added in Chapter 4.8 the example must be populated with its parents, children, methods, fields, etc. During early testing we found that showing all of the parents, fields, and methods was too much to display in a compact manner. Also, we found that much of this information is not significant to the task at hand.

We trimmed the children of focus types, any method that is not specifically recommended by a heuristic, all the fields, and any parent that are not directly implemented or extended by the example. This approach does not guarantee that important information is preserved, but in my experience it quickly pares down the information space without a noticeable loss in understanding.

As can be seen in Figures 4.2 and 4.4 examples which have all of their non-Java parents included can provide more information than is necessary to the understanding of the relationship between `ResourceNavigator` and `TaskList` to the Eclipse framework. Figures 4.3 and 4.5 show the trimmed examples.

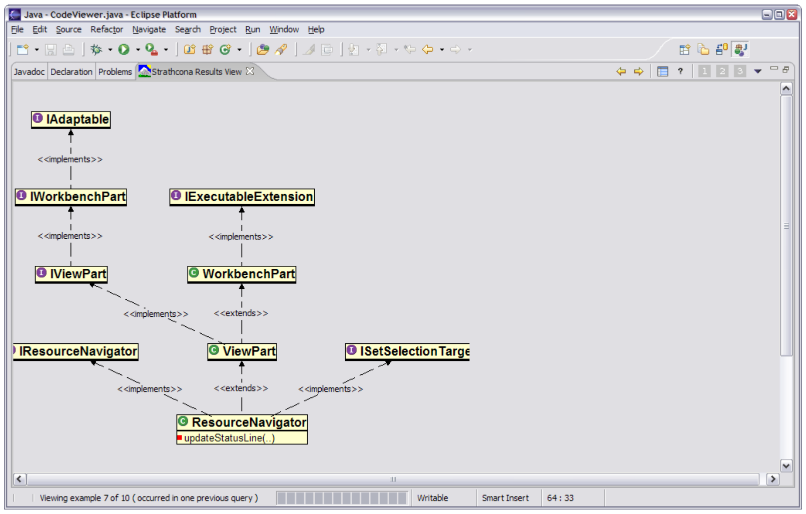


Figure 4.2: *Resource Navigator Example Without Parent Trimming*

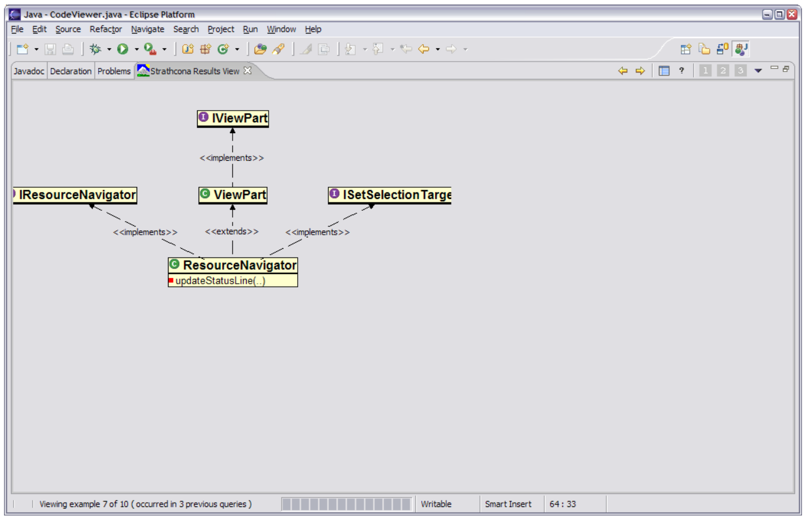
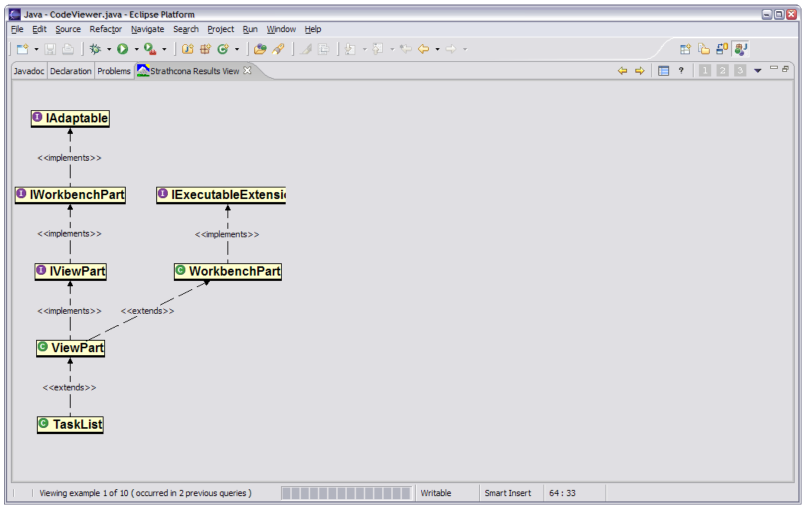
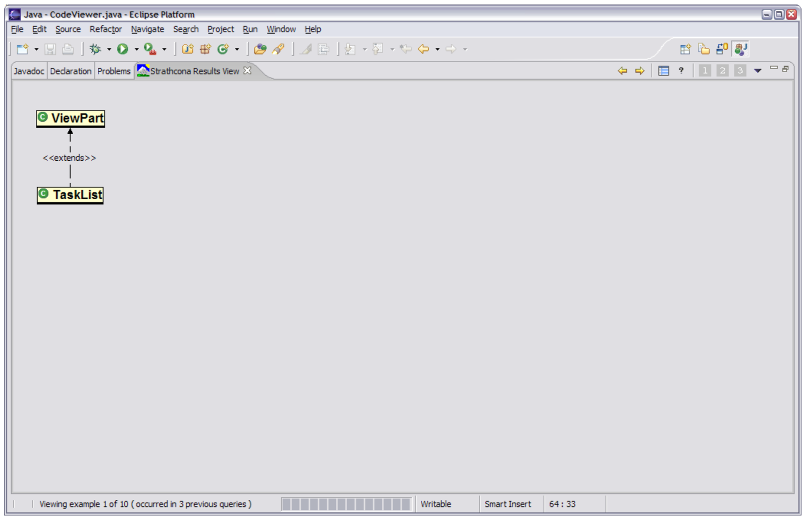

Java - CodeViewer.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Javadoc Declaration Problems Strathcona Results View
IViewPart
<<implements>>
IResourceNavigator
ViewPart
ISetSelectionTarge
<<implements>>
<<extends>>
<<implements>>
ResourceNavigator
updateStatusLine(..)
Viewing example 7 of 10 (occurred in 3 previous queries)
Writable
Smart Insert
64 : 33

Figure 4.3: *Resource Navigator Example With Parent Trimming*



Viewing example 1 of 10 (occurred in 2 previous queries)

Figure 4.4: *TaskList Example Without Parent Trimming*


Java - CodeViewer.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Javadoc Declaration Problems Strathcona Results View
ViewPart
<<extends>>
TaskList
Writable
Smart Insert
64 : 33

Viewing example 1 of 10 (occurred in 3 previous queries)

Figure 4.5: *TaskList Example With Parent Trimming*

Chapter 5

Evaluation

We have argued qualitatively in Chapter 3 that our approach requires less effort on the part of a developer to set-up and query an example repository. However, the usefulness of our approach depends upon whether the 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 using the Eclipse framework: three out of the four tasks we chose we encountered delays with when 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. Each skeleton was populated with some code, a seed, to serve as a starting point from which the developer could begin working. The seeds consisted of method calls on objects typed as classes or interfaces that could be easily identified using the Eclipse documentation, and in the cases of the third and fourth tasks, also the Eclipse code completion functionality. These seeds were not chosen because they provide the correct structural query but because they are the most relevant according to the Eclipse documentation.

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.

We recorded a screen capture of the subjects actions as they worked on the tasks. We also mirrored their workspace onto another machine so we could observe in real time without watching over their shoulders. Comments they made as they worked on their tasks as well as timing from the start of the task were recorded for informational purposes.

Before we ran the evaluation we performed a pilot once with an experienced developer to ensure that the tasks were possible and that our time expectations were realistic.

5.2 Results

The results are summarized in Table 5.1. For each task, we list how many examples were rated as useful for the task by the two developers, the number of examples for which the developers viewed the source, and whether or not a developer was successful at completing the task.

	Useful Example	Source Viewed	Succeeded at Task
Task 1			
S1	1	1	yes
S2	1	1	yes
Task 2			
S1	1	2	yes
S2	1	6	yes
Task 3			
S1	0	2	yes
S2	0	6	yes
Task 4			
S1	1	2	yes
S2	0	7	no

Table 5.1: *Results from Evaluation*

5.3 Task 1

The first task involved displaying text in the status bar of Eclipse as described in the scenario in Chapter 2.1. This is a simple task but requires the use of three intermediate parts of Eclipse (IWorkSpace, IActionBars, and IStatusLineManager). As described in the scenario, the developers were given as a seed, a call to `IStatusLineManager.setMessage(String)`. Each subject found the same example, the first one returned, useful to complete the task. Both subjects copied the code snippet from the example into their editor, changed the variable name for the `setMessage` and ran the code to test it. This example was returned by all but the basic USES heuristic.

5.4 Task 2

This task involved building an AST from a source string. A search of the Eclipse help reveals that `ASTParser.setSource(String)` should provide the appropriate functionality. However, 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 seed is shown in Figure 5.1. Both subjects again selected the first example returned (Figure 5.2) as it demonstrated the use of the seed method and included code to setup the parser and create the AST. The example they selected was returned by the CALLS, CALLS BEST FIT, and the USES heuristics and was returned for the reasons shown in Table 5.2. The second subject investigated a number of example snippets before deciding that the first one was the most relevant to the task. The snippet contained two extraneous calls which the subjects dealt with differently (one copied all of the code and deleted the extraneous sections in his code while the other only copied the sections of code which were relevant). Both subjects integrated the source from the snippets into their code to complete the task.

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

}
```

Figure 5.1: *Task 2 Seed*

Method Calls Target Method	org.eclipse.jdt.core.dom.ASTParser.setSource(char[])
Class uses Class	org.eclipse.jdt.core.dom.ASTParser

Table 5.2: *Task 2 Rationale*

5.5 Task 3

This task was included to see how the developers would react when Strathcona failed to return any valid examples. The 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.

Figure 5.3 shows the seed 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 to find an

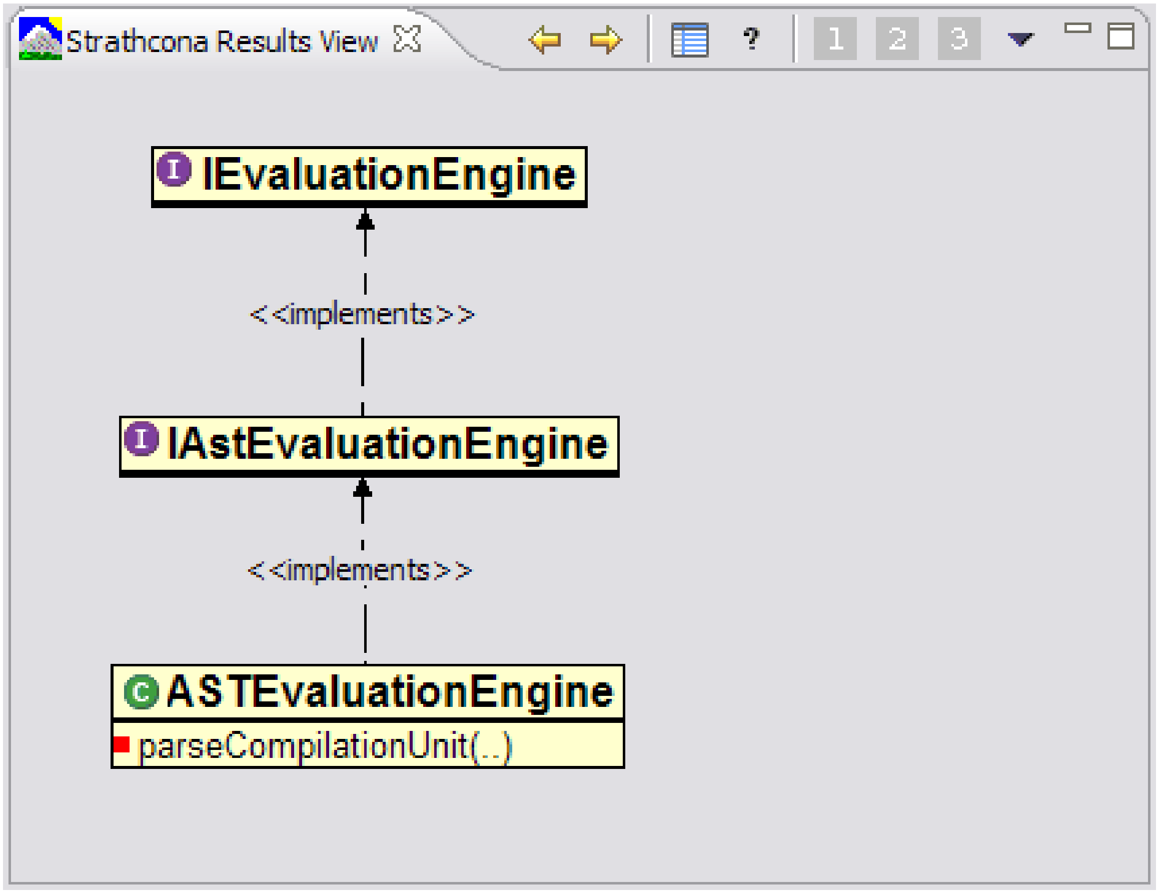


Figure 5.2: *Task 2 UML Representation*

example of how to create a SWT `Color` object and used some example code to accomplish this portion of the task.

5.6 Task 4

This task was the most complex. Using the `ASTVisitor` the subjects were to extract the method signatures for each method from the AST. This required the use of several different Eclipse framework types to efficiently complete the task including `Type`, `PrimitiveType`, `ArrayType`, `Name`, `SimpleName`, `QualifiedName`, `Code`, `Flags`, and `SingleVariableDeclaration`. The seed provided consisted of three of the most obvious method calls on MethodDeclaration which would be needed to complete the task as shown in Figure 5.4. Subject 1

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

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

Figure 5.3: *Task 3 Seed*

investigated MethodDeclaration using the Eclipse auto complete feature to try to derive a working solution before querying the tool as he was concerned about not finding a relevant example as was the case for the task before. Once he queried he examined the rationale for the first few of the examples carefully before deciding which two examples to investigate. With the exception of these first two examples (which both matched 4 seed method calls) the other examples matched at most 2 method calls. After investigating the source from the second example he discarded it and moved on to the first. This example (Figure 5.5) matched several calls from the seed as shown by its rationale for selection (Table 5.3). He proceeded to copy code from the example in small sections. The task was successfully completed. The example he selected to complete the task with was returned by all but the basic USES heuristic. Subject two accidentally queried the repository on the wrong method and searched through several irrelevant source files before deciding to implement the feature manually. He was partially successful but was unable to figure out some of the main details to create the signature successfully.

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

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

Figure 5.4: *Task 4 Seed*

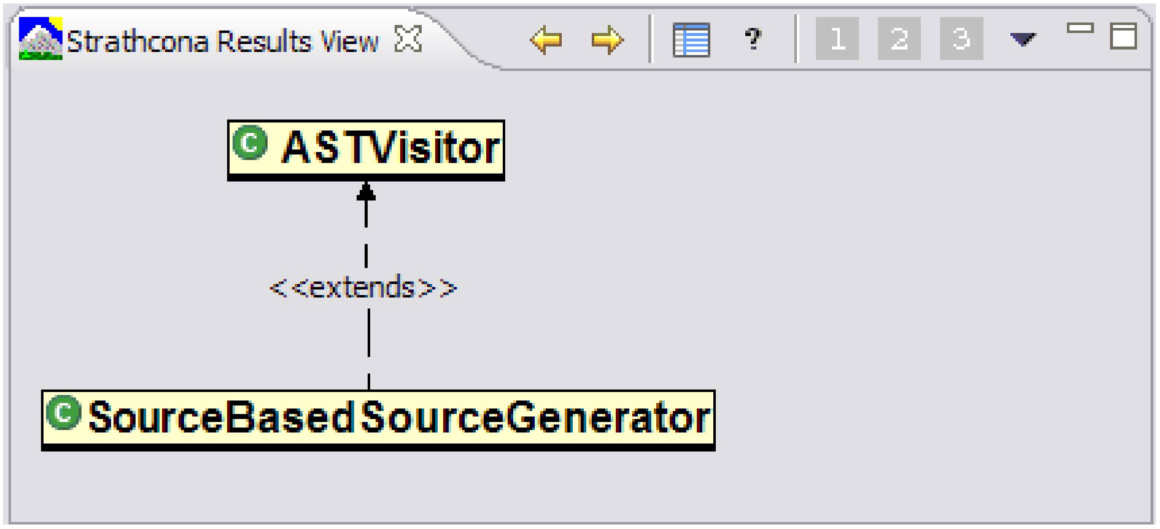


Figure 5.5: *Task 4 UML Representation*

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

Table 5.3: *Task 4 Rationale*

5.7 Summary

Subject 1 completed all 4 tasks successfully (while finding relevant examples for 3 of them) while subject 2 completed 3 out of 4 (finding relevant examples for 2 of the 4 tasks). In each of the tasks where the subject found a relevant example source code was copied from the example snippet into the task code. These results show that our tool can deliver relevant and useful examples to developers.

By focusing on a pre-existing framework, by considering cases that have occurred in our own experience of trying to use Eclipse, 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, there is reason to 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 users with additional experience with the framework of interest.

Chapter 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 Chapter, 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 [16], 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 [16]. 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 Missing Examples

Using a large framework, such as Eclipse, for experimentation makes it impossible to report recall and precision values for the queries we have considered. We know from the query results we have analyzed that precision is less than 100% (i.e., examples are returned that are not useful), but we have also shown in the study that precision is non-zero since we can return useful examples.

Our ability to recall useful examples from the repository is more difficult to judge. We would require extensive knowledge of the million plus lines of Eclipse code to determine if there is a useful example that could be returned for task thee in our study that we missed with our heuristics. Other approaches, such as the generalized and specialized match techniques of Zaremski and Wing [20], might complement our structural heuristics if this is the case.

The ability of Strathcona to return useful examples is also dependent upon the quality of the seed code to which matches are requested. 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 not likely provide relevant examples. As we have described in our case study, it is possible to use the documentation, in most cases, to find an appropriate seed. Strathcona fills in the details of how to complete a task that the documentation lacks.

6.3 Heuristic Refinement

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

Example Scoring Attempts we made to select the examples to return based on a scoring system in which the reasons for which an example was selected were rated at different values was problematic. This approach to scoring did not work because of the differences in the structural contexts forming the queries between tasks and developers. The reasons for the differences included 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 consider which objects are created inside of m. Although we identify object instantiations as a method call to a constructor and can match on these method calls, we did not find that heuristics that matched specifically on object instantiations were useful. One reason may be that in Eclipse some parts of the framework take care of instantiation while other parts leave this up to the client. For example whenever Factory classes are involved, the client does not instantiate objects but gets new objects delivered to them.

Hierarchies The heuristics do not transitively check the object hierarchy when considering parents, uses, or calls relations. For example, a method call on `ViewPart` does not also count as a call on `IViewPart`. 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 directly map to any examples. By not considering the call hierarchy we potentially miss examples that are split across method boundaries. For example, one example may accomplish a task with calls to three private methods, however our heuristics would consider each of these methods independently instead of treating them as one unit.

Heuristic	Subject 1	Subject 2	Difference
Task 1			
Inheritance	6/8	6/8	0
Task 2			
Calls	9/10	9/10	0
Calls Best Fit	7/10	10/10	+3
Calls with Inheritance	1/5	5/5	+4
Uses	2/10	5/10	+3
Uses with Inheritance	1/5	5/5	+4
Task 3			
Inheritance	3/10	8/10	+5
Calls	2/10	2/10	0
Calls Best Fit	6/10	2/10	-4
Calls with Inheritance	6/7	5/7	-1
Uses	4/10	2/10	-2
Uses with Inheritance	6/8	5/8	-1

Table 6.1: *Results from individual heuristic study*

Heuristic Evaluation We tried to evaluate each of the heuristics individually (Results Table 6.3). We had two developers check the results of each heuristic from three tasks and mark each example from each heuristic as useful or not. The variability between the developers can be shown in the results table. However, looking at the developers separately we found that they were not consistent in which examples they marked useful and as such we decided to abandon this type of analysis.

6.4 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 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.

6.5 Other Considerations

Open Source The OSS nature of the Eclipse system enabled us to easily populate the repository with high quality examples. In order to populate the repository we needed access to the source code of any applications using the framework. This naturally lends itself to open source frameworks and more importantly to open source applications using these frameworks. However, in the case of Eclipse it can also be seen that just by using the framework itself many examples can be extracted even though many plugins and applications implemented in Eclipse are not open source themselves. This could possibly lead to licensing issues when closed source plugins start incorporating code fragments from open source applications; this issue is beyond the scope of this thesis.

The nature of Eclipse itself must be considered in determining the applicability of our approach to other frameworks. As Eclipse is self-hosting, that is with the exception of the core components it is implemented as a series of plugins itself, simply indexing the framework provided us with a wealth of examples from which to work. Also, since those designing the framework are experts, their usage of the framework should be appropriate, providing a supply of high-quality examples. Also, the style in which Eclipse is implemented may have affected which heuristics were effective and which were not as noted in Chapter 6.3.

UI Limitations Strathcona's user interface is incomplete. Although the visual notation shows how the example fits into the framework through inheritance, it does not show which framework types are used, nor does it show what methods are called. This information is available in the rationale view and may be useful for it to be integrated on the visual representation. The users often used the visual notation we provided to discard examples but they would always check the rationale view before deciding if an example should be examined in more detail.

Other Heuristic Development Techniques Our approach for creating the heuristics for Strathcona was somewhat ad hoc as we went tried different ideas as they occurred to us and we browsed our structure repository. Machine learning and data mining techniques may be useful for determining some other general structural relationships between portions of software but these were not investigated. In order to attempt these techniques we would need to overcome the example scoring problem outlined above.

Example Quality Our heuristics were designed to be as general as problem. However, it is still possible that they have been trained on the Eclipse framework and would require modification for other frameworks. Although we believe Eclipse should provide a high-quality example of how to use itself, we cannot make any general statements about the quality of this code. All automatic approaches to generating software repositories suffer from this problem as only developers experienced with a framework can say whether or not an example is appropriate. It should be noted however that due to the low cost required to enter new applications into the repository organizations can keep even their most current projects indexed and available for diverse teams to use as examples.

Copy and Paste Programming Strathcona provides source snippets to developers which are easily copied to theer development workspace. We hope that these snippets lead to greater understanding of the framework. With proper tool support, such as those recommended in [10], the negative connotations associated with copy and paste programming may be addressed but this is beyond the scope of this thesis.

Eclipse Text Search Using the Eclipse search tools it is possible to easily search through several projects at a time looking for specific text fragments. For instance, searching for `setSource` as a developer may do for Task 2 retrieves 67 examples. Of these examples only a handful (less than 5) return uses of `ASTParser.setSource` while the rest use methods which are not of interest. There is no way to restrict the search to work only on structure which means that relevant examples are frequently overshadowed by false-positives.

Chapter 7

Future Work

This thesis has described a technique to automatically select examples based on software structural similarity. Further evaluation is necessary to determine the effectiveness of the examples provided by Strathcona on real development tasks. Before this evaluation can take place additional work needs to be done to improve the interaction model of the system. The usability of the tool could be improved by providing proper syntax highlighting for the code for selected examples viewed by a developer. Also, the provision of a more complete compact visual representation of each example might reduce the need for the developer to use the rationale view. Taking advantage of source folding or some form of fish eye view could be used to help hide the unnecessary details in the source view as well to allow the developer to concentrate only on the relevant portions of the source code. Automatically modifying code copied from the source view into the developers workspace (similar to templates) could also aid the transition from example to the current task.

Strathcona provides an easily extensible framework for evaluating heuristics on software structure. Additional heuristics should be investigated that take advantage of more context information available to developers. Specifically, heuristics should be investigated that can locate examples that are analogous to the context but not structurally related, as described in the discussion section. Developing a method of clustering examples to improve the breadth of the examples returned to the developer could also be a benefit in situations where the developers context has minimal structural information.

Chapter 8

Conclusion

Frameworks impose a large learning curve upon new developers. Framework documentation can help these developers become familiar with the system but it is often incomplete and cannot predict every task a developer may wish to complete.

Searching example repositories using free text queries, source code comments, or structured query builders all impose additional query overhead on developers and require that they build queries in a way that matches how the repository is constructed. In this thesis we have proposed using software structure, from a developer's editor, to build an implicit query into an automatically generated software repository. We believe our approach reduces developer overhead and insulates them from the details of the repository. We developed Strathcona, a tool that uses a set of heuristics to return related examples based on structural similarities between a development context and the example repository.

One case study with two developers was used to provide initial feedback on the effectiveness of structural queries for returning relevant examples. We found that in three of four tasks provided to developers our tool was able to return examples which the developers could directly use to complete their task. From this we conclude that structural hints can be used to successfully deliver examples to developers, although further investigation is needed to determine the effectiveness of these examples.

Bibliography

[1] G. Butler and P. D'enomm'ee. Documenting frameworks to assist application developers, 1997.

[2] Greg Butler, Rudolf K. Keller, and Hafedh Mili. A framework for framework documentation. *ACM Computing Surveys*, 32(1es):15, 2000.

[3] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *TSE*, 16(3):325–334, 1990.

[4] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418. IEEE Computer Society, 2003.

[5] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering*, pages 491–501. ACM Press, 1997.

[6] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–180. ACM Press, 1990.

[7] Scott Henninger. Retrieving software objects in an example-based programming environment. In *Proceedings of the 14th annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 251–260. ACM Press, 1991.

[8] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: Relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, pages 14–24. IEEE Computer Society, 2003.

[9] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 63–72, New York, NY, 1992. ACM Press.

[10] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *International Symposium on Empirical Software Engineering*. IEEE Computer Society, 2004.

[11] Amir Michail. Data mining library reuse patterns using generalized association rules. In *International Conference on Software Engineering*, pages 167–176, 2000.

[12] Amir Michail. Code web: Data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 827–828. IEEE Computer Society, 2001.

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

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

[15] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM Press, 2002.

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

[17] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, pages 513–523. ACM Press, 2002.

[18] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Foundations of Software Engineering*, pages 60–68, 2000.

[19] Yuhanis Yusof and Omer F. Rana. Template mining in source-code digital libraries. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 122–127, 2004.

[20] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions Software Engineering Methodologies*, 4(2):146–170, 1995.

Appendix A

Database Schema

This is the schema for the database we used. It is only valid for Postgresql but can be easily modified to work with any other SQL database (we created versions for Mysql and Hsqldb).

```
CREATE TABLE ScopeModifier (
        id serial primary key,
        name VARCHAR(255)
);
CREATE UNIQUE INDEX UniqueScopeModifer ON ScopeModifier(name);
```

Figure A.1: *ScopeModifier Table*

```
CREATE TABLE Class (
        id serial primary key,
        name VARCHAR(255),
        ownedBy VARCHAR(511),
        scope integer,
        isAbstract boolean,
        isStatic boolean,
        isInterface boolean
);
CREATE UNIQUE INDEX UniqueClass ON
        Class(name);
CREATE INDEX ClassName ON
        Class(name);
```

Figure A.2: *Class Table*

```
CREATE TABLE ClassHierarchy (
        id serial primary key,
        subClassId integer,
        superClassId integer,
        FOREIGN KEY(subClassId) REFERENCES Class(id),
        FOREIGN KEY(superClassId) REFERENCES Class(id)
);
CREATE UNIQUE INDEX HierarchyRelationship ON
        ClassHierarchy(subClassId, superClassId);
CREATE INDEX HierarchySubIndex ON
        ClassHierarchy(subClassId);
CREATE INDEX HierarchySuperIndex ON
        ClassHierarchy(superClassId);
```

Figure A.3: *Class Hierarchy Table*

```
CREATE TABLE Method (
        id serial primary key,
        name VARCHAR(511),
        parentType integer,
        returnType integer,
        overridesMethod integer,
        isAbstract boolean,
        isStatic boolean,
        scope integer,
        FOREIGN KEY(parentType) REFERENCES Class(id),
        FOREIGN KEY(returnType) REFERENCES Class(id)
);
CREATE UNIQUE INDEX UniqueMethod ON
        Method(name);
CREATE INDEX MethodName ON
        Method(name);
CREATE INDEX MethodReturn ON
        Method(returnType);
CREATE INDEX MethodParent ON
        Method(parentType);
```

Figure A.4: *Method Table*

```
CREATE TABLE Calls (
        id serial primary key,
        hostMethod integer,
        targetMethod integer,
        numberOfTimes integer,
        FOREIGN KEY(hostMethod) REFERENCES Method(id),
        FOREIGN KEY(targetMethod) REFERENCES Method(id)
);
CREATE UNIQUE INDEX UniqueCalls ON
        Calls(hostMethod, targetMethod);
CREATE INDEX CallsFrom ON
        Calls(hostMethod);
CREATE INDEX CallsTo ON
        Calls(targetMethod);
CREATE INDEX HostMethod ON
        Calls(hostMethod);
```

Figure A.5: *Calls Table*

```
CREATE TABLE Creates (
        id serial primary key,
        hostMethod integer,
        targetType integer,
        numberOfTimes integer,
        FOREIGN KEY(hostMethod) REFERENCES Method(id),
        FOREIGN KEY(targetType) REFERENCES Class(id)
);
CREATE UNIQUE INDEX UniqueCreates ON
        Creates(hostMethod, targetType);
CREATE INDEX CreatedBy ON
        Creates(hostMethod);
CREATE INDEX Created ON
        Creates(targetType);
```

Figure A.6: *Creates Table*

```
CREATE TABLE FieldTable (
        id serial primary key,
        hostType integer,
        targetType integer,
        name VARCHAR(255),
        scope integer,
        isStatic boolean,
        FOREIGN KEY(hostType) REFERENCES Class(id),
        FOREIGN KEY(targetType) REFERENCES Class(id)
);
CREATE UNIQUE INDEX UniqueField ON
        FieldTable(hostType, name);
CREATE INDEX FieldHost ON
        FieldTable(hostType);
CREATE INDEX FieldType ON
        FieldTable(targetType);
CREATE INDEX FieldName ON
        FieldTable(name);
```

Figure A.7: *Field Table*

```
CREATE TABLE Sources (
        id serial primary key,
        name VARCHAR(255),
        version VARCHAR(255),
        url VARCHAR(255),
        dateAdded VARCHAR(255),
        srcLoc VARCHAR(255)
);
CREATE INDEX SourceName on
        Sources(name);
```

Figure A.8: *Sources Table*

Appendix B

Database Queries

Group Queries
Select all Classes
Entity retrieval queries
Find class by type name
Find method by method name
Find field by field name
Queries on Types
Find all method names declared by type
Find all field types declared by type
Find all field names declared by type
Find all types that instantiate this type
Find all types that are instantiated by this type
Find all types overridden by this type
Find all methods overridden by this type
Find source file location by type
Find all parent types for this type
Find all child types for this type
Find all types with parent of type
Find all types with child of type
Find all types with field of type
Find all types used by this type
Find all types using this type
Queries on Methods
Find return type for method
Find types created by method
Find declaring type for method
Find method overridden by method
Find type overridden by method
Find all methods called by method
Find all methods calling method
Queries on fields
Find field type by field name
Find type declaring field name
Boolean Queries
Type has field of type
Type has parent of type
Type uses type
Method calls method
Counting Methods
Number of methods called by method

Table B.1: *Pre compiled queries*

Aggregate Queries
•Find methods which match the most method calls in $m_0..m_n$ sorted by number of matched method calls
•Find methods which match the most method calls in $m_0..m_n$ AND have a common parent to the declaring type of the query method sorted by number of matched method calls
•Find types which use the same local types in $u_0..u_n$ sorted by number of matched uses
•Find types which use the same local types in $u_0..u_n$ AND have a common parent to the query type sorted by number of matched uses
•Find types which match the same parent types in $c_0..c_n$ sorted by number of matched parents

Table B.2: *On the fly queries*

Appendix C

Pilot Document

C.1 Exercise Rationale

The purpose of this exercise is to evaluate whether or not my tool can help you to complete a programming task.

C.1.1 Tool

Whenever you are stuck you can query the tool and it will provide you relevant programming examples which will help you with what you're working on. To make the query you write code in the editor and query it from the package manager. Each task below contains a programming 'seed'. This seed represents a small bit of knowledge in the form of code which you could have found using eclipse help. The goal of the tool is to augment the help and provide context for the information that is in that documentation.

In order to query you need to right click on the method you are working on in the package explorer for your editor and choose Query Related (the icon is a little mountain) as shown in Figure C.1. This then takes a snapshot of what you're doing and sends it to the server for analysis. Once the results have been computed (can take between 1 and 30 seconds) the Strathcona Results View is populated for you.

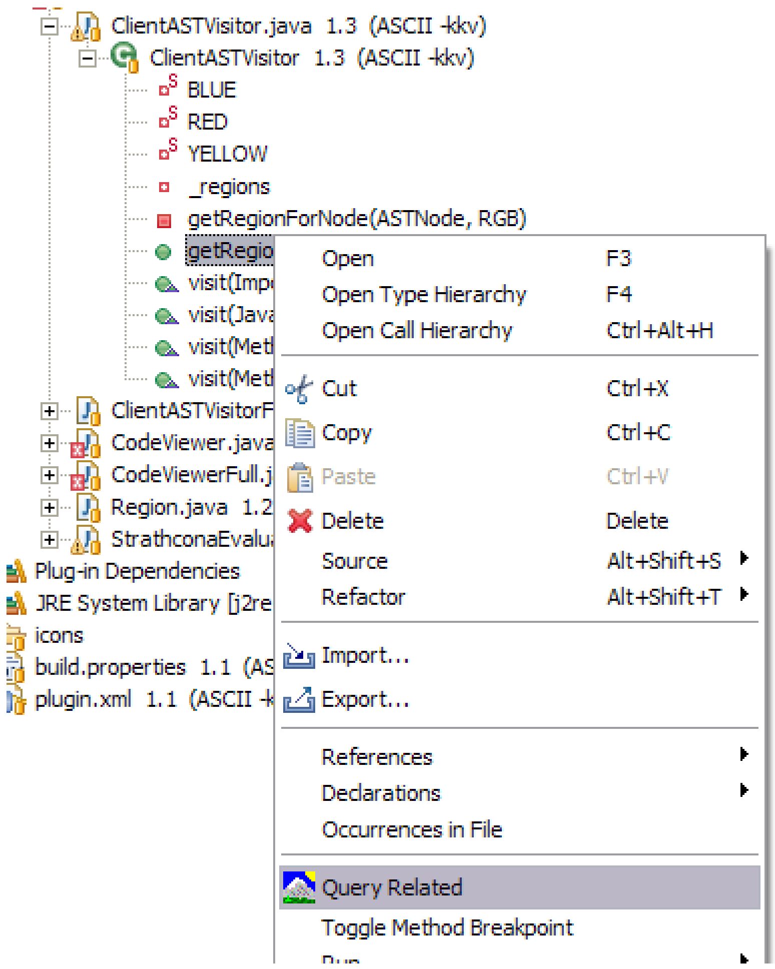


Figure C.1: *How to Query When You Are Stuck*

The results view contains a graphical representation of an example. These examples have been selected by comparing your current work to work in the repository and returning the most related artifacts to the view. The examples are ranked in the order the system thinks is most pertinent. Each example is a piece of the eclipse repository itself.

This results view has a number of buttons designed to help you navigate the examples, learn more about them, and rate whether or not they were useful (See Figure C.2). The two arrow buttons allow you to navigate back and forth between the examples. As you do this the status bar updates so you know where in the example list you are as well as whether or not you've examined the example in depth or ranked it in the past. The status line also lets you know if the current example was returned by a previous query so you can keep track of recurrent examples. Next is the Rationale Button which allows you to see exactly why the example was returned to you. The source button allows you to view the source for your example. For some files this can take a long time as the file needs to be transmitted over the network. The numbered buttons you rank the examples you used.

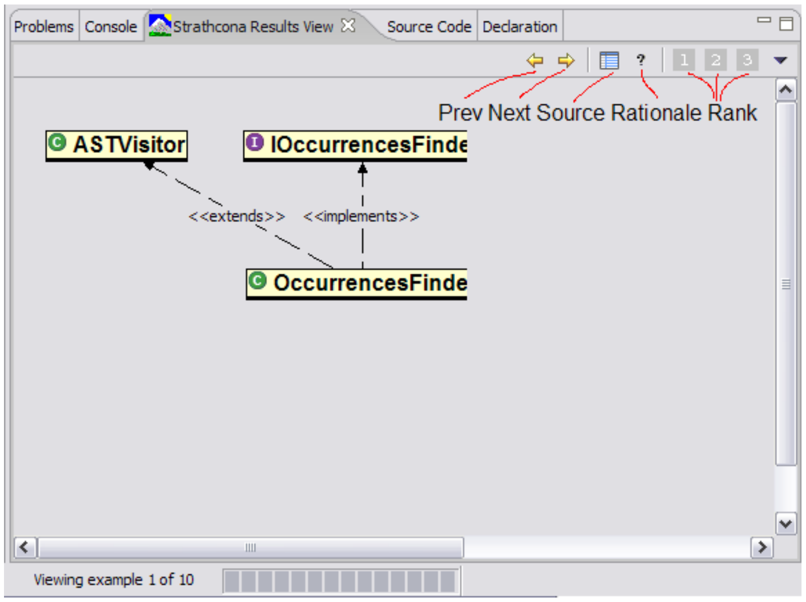
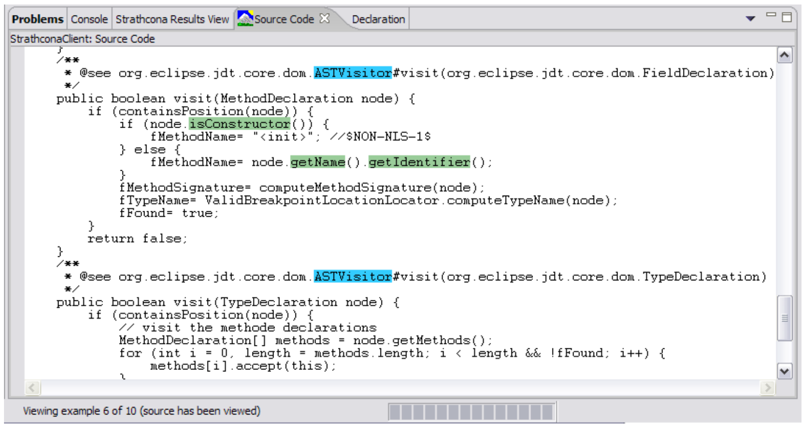


Figure C.2: *Annotated Strathcona Results View*

Ultimately you want to find some source which helps you to do what you're trying to do. When you click to look at the source code it is returned in a hilighted form (Figure C.3). Although the hilighting is simple it should give you a quick idea of which areas in the source to look at. Types are hilighted in blue while method calls are hilighted in green.


Problems
Console
Strathcona Results View
Source Code
Declaration
StrathconaClient: Source Code
Viewing example 6 of 10 (source has been viewed)

```
    }
    /**
     * @see org.eclipse.jdt.core.dom.ASTVisitor#visit(org.eclipse.jdt.core.dom.FieldDeclaration)
     */
    public boolean visit(MethodDeclaration node) {
        if (containsPosition(node)) {
            if (node.isConstructor()) {
                fMethodName= "<init>"; //$NON-NLS-1$
            } else {
                fMethodName= node.getName().getIdentifier();
            }
            fMethodSignature= computeMethodSignature(node);
            fTypeName= ValidBreakpointLocationLocator.computeTypeName(node);
            fFound= true;
        }
        return false;
    }
    /**
     * @see org.eclipse.jdt.core.dom.ASTVisitor#visit(org.eclipse.jdt.core.dom.TypeDeclaration)
     */
    public boolean visit(TypeDeclaration node) {
        if (containsPosition(node)) {
            // visit the methode declarations
            MethodDeclaration[] methods = node.getMethods();
            for (int i = 0, length = methods.length; i < length && !fFound; i++) {
                methods[i].accept(this);
            }
```

Figure C.3: *Annotated Source Snippet*

C.1.2 How to Use the Tool

Start by querying the repository with the seeds provided. If you use any of the examples to complete your task please rank them with the numbered buttons so we know which one(s) are useful. Note that an example can be ranked only once and only one example can be assigned to each number grade.

As you look through the example list the rationale view can be useful to make sure that the example you're looking at has the features you are interested in. For instance if you're interested in specific examples which use a particular method call or type check the rationale view to make sure the example meets these requirements. Some of the heuristics in the backend make different types of guesses to generate the examples and some of them may not be useful to you.

Often the syntax hilighting in the code view is incomplete so don't rely on it 100 percent. If you look at the rationale view before looking at the source you will have a better idea of the types of methods/calls you should be on the lookout for. Code can be copied from the source window into your current editor via the standard cut/paste interfaces. Often it is useful to copy the code in its entirety to TextPad (it is open below) which you can then use to search using F5.

Whenever you want to test your code just startup the runtime workbench. The view can be activated using the show view option in the perspective view (it is called the Strathcona Evaluation View). By pasting any valid code into the view you can see if your code is working or not. Task

C.2 The Tasks

You are going to implement a CodeViewer View. This view will allow you to paste code into it, have it be parsed into an AST, and have various elements of the source be hilighted. The framework for this task has been implemented in CodeViewer.java. Part of the AST task is located in ClientASTVisitor.java. This task can be broken into four sub components which you should do in the order they are given.

Do each task in order. Begin by querying the repository and take it from there. If you use an example or find something particularly useful be sure to flag it with the ranking buttons. Each task has a time limit in order to help you from getting stuck on one problem.

C.2.1 Task 1 Provide View Status (UpdateStatusLine(String))

Users of the view should be kept apprised of changes in the state of the editor. Whenever an AST is built or code is selected in, copied from, or pasted into the view the status line will be updated. However, the method which actually puts the message on the status are needs to be implemented. A starting seed has been provided in the method from which you can begin querying. 30 minutes have been allocated to this task.

C.2.2 Task 2 Parse Source Document (CreateASTFromSource())

After code has been pasted into the view we want to be able to hilight different aspects of it. In order to do this we need to convert the textual representation of the source into an AST. Implement this method. Again, a starting seed has been provided. The last method call in this method should be to processAST(ICompilationUnit). 30 minutes have been allocated to this task.

C.2.3 Task 3 Source Hilighting (HilightRegions(Vector))

ProcessAST checks to make sure the AST doesn't have any catastrophic errors and then visits all of the nodes to determine which ones need hilighting. Our hilighting engine hilights method calls as blue, import statements in yellow, and javadoc comments in red. However, the CodeViewer doesn't know anything about hilighting. Implement HilightRegions(Vector) to change the text style for the regions specified in the parameter to the method. The foreground colour (text colour) should be changed while the background colour should stay the same. Don't worry about populating the regions vector, this is already taken care of in ClientASTVisitor. 60 minutes have been allocated to this task.

C.2.4 Task 4 Method Listing (visit(MethodDeclaration node))

Some information requires extra work to get out of an AST. Implement the visit(MethodDeclaration) method in ClientASTVisitor. This method should print the method signature of the message to the console. The format of the signature should be:

return-type methodName([paramType paramName,paramType paramName]) [throws exceptionType , exceptionType]

A seed has been placed in the method to get you started. 75 minutes have been allocated to this task.

C.3 Post Exercise

There isn't anything formal to do after you've completed the tasks but I would like your feedback on the tool. Specifically I'd like to know if the examples presented to you were useful and which types of examples were useful or not useful. Also, any UI/usability improvements would be appreciated.