Unsticking Eclipse plug-in Development

Reid Holmes and Gail C. Murphy
Department of Computer Science
University of British Columbia
rtholmes,murphy@cs.ubc.ca

October 5, 2004

Abstract

Eclipse is an open and extensible integrated development environment that allows third-party developers to integrate their plug-ins (tools) deeply with features shipped with the environment. Building a plug-in requires a developer to use Eclipse as a framework, extending and implementing classes in the Eclipse API, and calling appropriate methods on Eclipse objects using appropriate protocols. All too often, when trying to build a plug-in, we have seen developers become stuck, unable to make progress on a programming task associated with their plug-in. To help unstick developers in this situation, we have built the Strathcona tool that can provide examples about the use of the Eclipse framework based on the structure of the code on which the developer is currently working. In this paper, we provide an overview of the Strathcona, and describe improvements to aid the usability and adoption of the tool.

1 Introduction

Eclipse is an open and extensible integrated development environment that allows third-party developers to integrate their plug-ins (tools) deeply with features shipped with the environment. For instance, the AJDT plug-in integrates deeply with the Java tooling in Eclipse, annotating views of Java programs with information about declared aspects. This ability to integrate third-party plug-ins seamlessly benefits both plug-in developers and users of the Eclipse environment: plug-in developers benefit through improved productivity and the capability to deliver a robust, full-featured tool; the users benefit by gaining access to more functionality within the same environment.

Building a plug-in requires a developer to use Eclipse as a framework, extending and implementing classes in the Eclipse API, and calling methods on Eclipse objects using appropriate methods. A developer has two basic sources of information available about how to build a plug-in: documentation in various forms including web articles, Eclipse help, and books, and the source for Eclipse itself as well as open-source third-party plug-ins. Although there is an extensive amount of documentation available for Eclipse, we have often found that the programming task we wish to complete is either not listed in the documentation or is too difficult to locate. Frequently, though, we have found that example snippets posted to newsgroups or other similar fora have helped us solve the task. The difficulty lies in being able to find the code snippet of interest.

To help developers more easily locate a useful snippet, we have developed Strathcona, a tool that can deliver relevant examples, including code snippets, to a developer based on the structural context of the code which the developer is currently writing. The idea of delivering examples to help developers is not a new idea. However, past systems either required a developer to learn a query language to be able to retrieve examples [2], write their source in a way that conformed to the examples in the repository [7], have an idea of what type of example would likely help them with their task [4], or required specialized examples to be crafted for the repository [5]. Strathcona overcomes these limitations as the query is generated automatically from the structure of the code a developer is writing, and the repository is built simply from code that uses the framework of interest. Strathcona selects code from the repository based on structural similarity us-
ing a set of heuristics we have developed. A developer accesses the repository using an Eclipse plug-in; currently the repository is populated with the code for the Eclipse environment. With this configuration, Strathcona can provide examples for developers who become stuck when writing an Eclipse plug-in.

In this paper, we focus on how Strathcona can help developers, and how developers use the tool. Details on the internals of Strathcona and on an evaluation of its use are available elsewhere [3].

2 Sample Scenario

The Abstract Syntax Tree (AST) contains a programmatic view of a class in Eclipse. Developers who wish to extract information from it must typically define an `ASTVisitor` which traverses this data structure and extracts the artifacts the developer requires. In this scenario we describe a task whereby the developer wishes to extract the relevant information about, and construct a string representation of, each method listed in the AST. The method signature is to be comprised of the methods visibility, its modifiers, its return type, name, parameter types and any exceptions it throws. Examining the Eclipse help for `ASTVisitor` the developer learns of `MethodDeclaration` which upon investigation represents all of the data in necessary from the AST to complete the task.

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

Figure 1: Scenario Seed

The developer quickly identifies several methods needed to complete the task including `getModifiers()`, `getName()`, and `parameters()`. However, the developer does not know how to make use of these types to turn them into the string representation they want and becomes stuck. Figure 1 shows what the developer has written at this point. The developer then queries Strathcona by selecting the method they are writing, `visit(MethodDeclaration)`, from the package explorer and choosing Query Related. A query, based on the structure of the code in Figure 1 is sent to the server and processed.

![Strathcona Result and Rationale Views](image)

Strathcona returns a list of ten examples to the developer. Each of these examples consists of a visual representation of how the example fits into the Eclipse framework (top of Figure 2), the rationale (bottom of Figure 2) for why the example was selected, and a code snippet (Figure 3). The examples returned by Strathcona are ranked in order with the most similar examples listed first. The developer examines the rationale for the first example and finds that it calls every method that they have identified as pertinent for the task. Examining the code snippet for this example, they find a method that contains many highlighted method calls corresponding to the code they have written and upon which they queried. Perusing this method, the developer finds that it constructs a textual representation of the method. The developer copies code from the snippet into their system. Upon compilation, they find references to several private methods in the snippet. Copying these methods to the working code resolves all remaining compilation problems. A more complete examination of the copied code reveals parts that print out information and that add punctuation to the representation, the developer comments these lines out and changes the return type of the method. These actions complete the task. Over 100 lines of code are used from the provided example.
Strathcona is comprised of a client and a server. The server portion houses the repository and services requests for examples. The client formulates these requests and displays the resultant examples.

### Populating the Server

Our current prototype is populated with the entire structure of the Eclipse 3.0 M8 SDK. Table 1 shows the size of the data we represent in the repository. To add a project to the Strathcona database the repository administrator imports it into an Eclipse workspace and launches the Strathcona extraction perspective. The administrator then selects the projects to add to the database and clicks a single button to accomplish this action. The projects are added one at a time and can be cancelled at any time.

### Generating the Context

When the developer requests examples, the structural context of the current editor is extracted, formatted and sent to the server. The context includes the type of the class being edited, the parents of that class, and the types of all of its fields. If a method was the target of the query, it is also added to the context as well as the methods that it calls.

### Example Matching

The server uses a set of heuristics to select examples related to the structural context described in a query. The six heuristics (CALLS, CALLS BEST FIT, CALLS WITH INHERITANCE, USES and USES WITH INHERITANCE) each

---

**Table 1: Size of Example Repository**

<table>
<thead>
<tr>
<th>Category</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>17,456</td>
</tr>
<tr>
<td>Methods</td>
<td>124,359</td>
</tr>
<tr>
<td>Fields</td>
<td>48,441</td>
</tr>
<tr>
<td>Inheritance Relations</td>
<td>15,187</td>
</tr>
<tr>
<td>Object Instantiations</td>
<td>43,923</td>
</tr>
<tr>
<td>Calls Relations</td>
<td>1,066,838</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1,316,204</td>
</tr>
</tbody>
</table>
return a set of results. Ten examples determined to be best are selected from these sets and returned to the client.

The INHERITANCE heuristic is designed for situations when the developer has already determined how their class fits into the inheritance hierarchy. The CALLS heuristics are designed for situations when the developer knows some of the calls which need to be made but not all of them. The USES heuristics are designed for situations when the developer knows what types they want to use but not how they interact.

Presentation Each example returned from the repository is populated with relevant structural information as well as the rationale for why it was chosen. The client uses the structural description of the example to construct a visual representation of the example (e.g., top of Figure ??). This representation shows how the example fits into the Eclipse inheritance hierarchy. The Strathcona results view also includes controls to scroll between examples, view the rationale, and request the source snippet for an example. The source snippet appears in a separate text view that highlights the parts of the example that are structurally related to the development context.

Performance Building the query from a developer’s context typically takes less than 500ms, and servicing the query takes between four and twelve seconds.1 As Strathcona is positioned at developers who are stuck we feel that this delay waiting for examples is not excessive.

4 Discussion

In this section, we outline the evaluation that we have performed to validate the effectiveness of Strathcona (Section 4.1). We then describe possible improvements to Strathcona’s user interface (Section 4.2) and the tool itself (Section 4.3).

4.1 Evaluation

We have qualitatively tested the effectiveness of Strathcona by having two developers use the tool while working on four Eclipse plug-in programming tasks. One subject found examples to help with all four of the tasks; the second subject found relevant examples for three of the four tasks. This evaluation showed that Strathcona is able to return structurally relevant examples, and developers can recognize and use those examples to help complete programming tasks. Further details are available elsewhere [3].

4.2 Interface Improvements

The structural context view displayed to the developer by Strathcona is minimal. The developer must use the rationale view to understand why an example was selected. The information in the rationale view could be integrated into the structural context view. Specifically, the uses and calls relations could be added to the view. Making the structural diagram interactive to display additional information as a developer hovered over nodes and edges in the diagram is one way to present the information. The code folding features in Eclipse to fold away parts of the source snippet not related to the context could also be used to reduce the amount of information being presented to the developer.

4.3 General Improvements

Several improvements are possible to help Strathcona deliver more relevant examples and to help a developer make use of a relevant example. For instance, collaborative ranking could be added to help developers collectively weed out examples that are ineffective, that do not use the framework properly, or that are confusing. The query mechanism could also be modified to be more tightly integrated with the Eclipse editor to make the use of Strathcona more natural. Lastly, additional tool support in the Strathcona plug-in could help developers integrate code snippets into their code. This support could be as simple as simple renaming refactoring to help modify copied variable names so that they match the types in the developer’s context.

---

1The 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.
5 Related Work

Strathcona aims to reduce the amount of work required by the developer to locate relevant source code from an example repository. Others have taken different approaches to this problem in the past.

The Reuse View Matcher (RVM) [5] provides a set of views to developers that show how to use individual classes in the framework. This technique requires that the framework developer construct the examples, including animations, and include them with the documentation. This places a large burden on the framework developer. Strathcona eliminates this effort by indexing the source structure automatically from applications containing example code. The examples are then generated automatically when a developer’s structural context is matched to the repository.

CodeFinder [2] is a tool that helps a developer construct queries using a wizard-like approach in which the choices made by the developer at each step influence the options given in later steps. Once the developer has constructed the query it is sent to the server to match against examples. Strathcona avoids this query construction phase by generating the query automatically based on the structure available in the development context.

CodeBroker [7, 6] most closely matches the goals of Strathcona. It automatically queries the repository and returns examples to the developer. It is an active system, that continually queries and displays results. Queries are based on both code comments and method signatures. This tool suffers whenever either the framework developer or application developer has failed to write comments in their code. Strathcona avoids difficulties with incomplete or out of date comments by relying on the structure of the code which should always be up to date (at least, if the code is to work).

The Hipikat tool [1] can provide examples, in the form of past source check-ins to the project archives, to developers based on the similarity of a change task upon which they are working to change tasks completed on the project in the past. The developer must determine if the past example provided is still relevant to the current state of the framework.

6 Conclusion

Eclipse is a large framework. Developers attempting to build Eclipse plug-ins sometimes take longer than anticipated to complete a programming task on a plug-in when they lack appropriate documentation or examples about how to use the framework. Our Strathcona tool is intended to unstick these developers by providing them with an appropriate example. Strathcona extracts information about the structure of a developer’s code to use as a query to an example repository. Examples are selected from the repository based on structural similarity. We have demonstrated the utility of the approach to serving examples for Eclipse plug-in developers. Our approach should apply equally well to other frameworks. This testing has been left to future work.

References


