Approximate Structural Context Matching: An Approach to Recommend Relevant Examples

Reid Holmes, Robert J. Walker, *Member*, *IEEE*, and Gail C. Murphy, *Member*, *IEEE Computer Society*

Abstract—When coding to an application programming interface (API), developers often encounter difficulties, unsure of which class to subclass, which objects to instantiate, and which methods to call. Example source code that demonstrates the use of the API can help developers make progress on their task. This paper describes an approach to provide such examples in which the structure of the source code that the developer is writing is matched heuristically to a repository of source code that uses the API. 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. The repository is generated automatically from existing applications, avoiding the need for handcrafted examples. We demonstrate that the approach is effective, efficient, and more reliable than traditional alternatives through four empirical studies.

Index Terms—API usage, structural context, heuristic search, Strathcona, example recommendation.

1 Introduction

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

To help developers find their way, researchers have advocated the establishment of example repositories to house examples of an API's use (e.g., [16], [18], [20]). These approaches differ in the means that a developer uses to retrieve relevant examples from the repository: Developers must either learn a new query language [7], have an idea of what kind of example would likely help them with their task [14], [13], or write their source in a style that conforms to that of the example repository [20]. All of these approaches require a substantial investment on the part of the developer to locate and incorporate examples from the repository.

To ease the burden on the developer, we have developed an approach that uses the structure of the source code under development to find relevant examples in a repository. We first extract a set of structural facts about the context of an indicated source code fragment. We then use this automatically formed *structural context* to search a source code repository heuristically for examples with similar structural context. The best results from the search are returned to the developer for consideration.

Our approach has two key differences from previous work. First, the repository automatically stores source code in a form whereby task-specific examples can be generated as needed, rather than providing generic examples or requiring their manual creation and insertion into the repository. Second, the developer need not learn a new query language, nor must the developer code to particular standards to enable a search to be conducted. Instead, the structural context that is used to form a query is extracted automatically from a developer-indicated source code fragment; for example, the fragment of interest can be indicated by a developer highlighting the code in an integrated development environment (IDE).

To investigate our approach, we have built the Strathcona tool. The client portion of Strathcona, a plug-in for the Eclipse IDE,[1] extracts the structural context of a fragment of selected Java source code for which the developer would like to see examples of use. The server portion of Strathcona houses the repository within which it searches for similar structural contexts using an extensible set of structural similarity heuristics. The server forms examples from the similar code that it finds. These examples are returned to the developer for perusal. Each example consists of three parts: a graphical overview illustrating the structural similarity to the queried fragment, a textual description of why the example was selected, and source code with structural details highlighted that are similar to the queried

- *R. Holmes and R.J. Walker are with the Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, Alberta, Canada T2N 1N4. E-mail: {rtholmes, rwalker}@cpsc.ucalgary.ca.*
- *G.C. Murphy is with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, British Columbia, Canada V6T 1Z4. E-mail: murphy@cs.ubc.ca.*

Manuscript received 10 Mar. 2006; revised 29 Aug. 2006; accepted 13 Sept. 2006; published online 14 Nov. 2006.
Recommended for acceptance by W. Griswold and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0062-0306.

1. http://eclipse.org (last accessed: 9 June 2006).

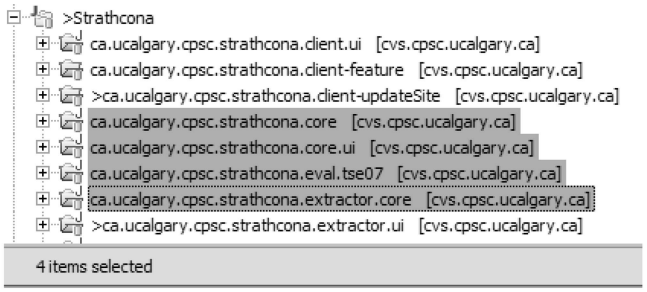

Fig. 1. An example of the behavior of the Eclipse status line.

fragment. The graphical overview and textual rationale description can be used by the developer to quickly decide whether the recommended example is worth examining more closely.

We have performed a series of empirical studies to evaluate the approach and the tool. The results of these studies provide evidence that 1) developers can recognize and use examples located by Strathcona to complete development tasks, 2) the approach provides better results —without needing to formulate explicit queries—than traditional search tools, and 3) the approach is robust in its ability to locate relevant examples for a range of queries.

The paper begins, in Section 2, with a scenario where the use of an example recommendation tool would benefit the developer. Section 3 compares our approach to other efforts. Section 4 describes the concept of approximate structural context matching and how this concept has been instantiated in a concrete tool, called Strathcona, for recommending relevant examples. Section 5 presents our evaluation. Remaining issues and future work are discussed in Section 6.

2 Sample Scenario

The user interface of the Eclipse IDE includes a status line that reports information about the state of the environment to the user. For example, when the user selects some items from a tree view in Eclipse, the status line shows the number of selected items (Fig. 1). Consider a developer who is writing an extension for the Eclipse environment, called a plug-in, and who wants to display a message on the status line from within their view. The first place a developer might look for help with this task is the Eclipse documentation. Checking this resource, the developer finds a reference to an interface called `IStatusLineManager`. Looking at the API documentation for `IStatusLineManager` (Fig. 2), the developer finds the seemingly appropriately named method `setMessage(String)`. The API documentation mentions that there is a concrete implementation `StatusLineManager` but does not provide any clues as to how to obtain an existing instance of that type. The developer attempts to implement a method, named `updateStatusMessage (String msg)`, that calls `IStatusLineManager.setMessage(msg)`. Unfortunately, this method will not compile because `IStatusLineManager` is an interface type, but `setMessage(String)` is an instance method. The API documentation mentions that

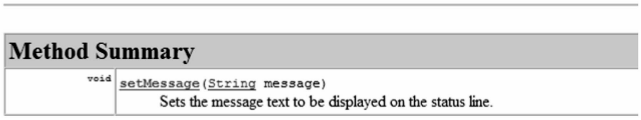

Fig. 2. API documentation for `IStatusLineManager`.

there is a concrete class `StatusLineManager` but contains no obvious clues as to how to obtain an existing object on which to call `setMessage(String)`. At this point, the developer becomes lost, uncertain of the next step needed to complete the task.

This scenario describes a conceptually simple task of updating a status line. However, even this simple task requires knowledge about the interaction between several types of the framework, including `ViewPart`, `IViewSite`, `IActionBars`, and `IStatusLineManager`. This interaction is not described in the Eclipse documentation.[2] Although the interactions may be discovered using the code completion features in Eclipse, the correct sequence of calls is difficult to find as there are a total of 79 methods available across the four classes.

Strathcona can help in this situation. As shown in Fig. 3a, the developer highlights some code they have written and chooses Query Strathcona from the context menu to request similar examples. The client portion of Strathcona extracts the structural context of the source code fragment that the developer has selected. The structural context comprises the structural details about the selected code and its containing class. In this case, the structural context will contain the facts that the selected code calls `IStatusLineManager.setMessage(String)`, that it uses `IStatusLineManager` and `String`, and that its containing class inherits from `ViewPart`. The structural context is sent to the server portion of Strathcona, which returns 10 structurally related examples. Each of these examples consists of three parts: a code fragment, a structural description of the code fragment, and a rationale explaining the relevance of the code fragment to the problem the developer is facing.

Fig. 3b shows a graphical view of the relevant structure for one of the returned examples. The rationale for this example shows that it was selected because its code calls the `IStatusLineManager.setMessage(String)` method, uses `IStatusLineManager`, and extends `ViewPart` (Fig. 3c). The developer requests the code for the example, and Strathcona highlights the call chain `getViewSite().getActionBars().getStatusLineManager().setMessage(msg)` as shown in Fig. 3d. The developer assesses this example as useful and attempts to use it by directly copying the statement into their method. Testing this code, the developer finds that they have completed their task.

3 Related Work

We describe previous efforts in three areas related to our structural context matching approach: other efforts aimed at easing the use of a complex framework or API, efforts

2. As of Version 3.2M5.

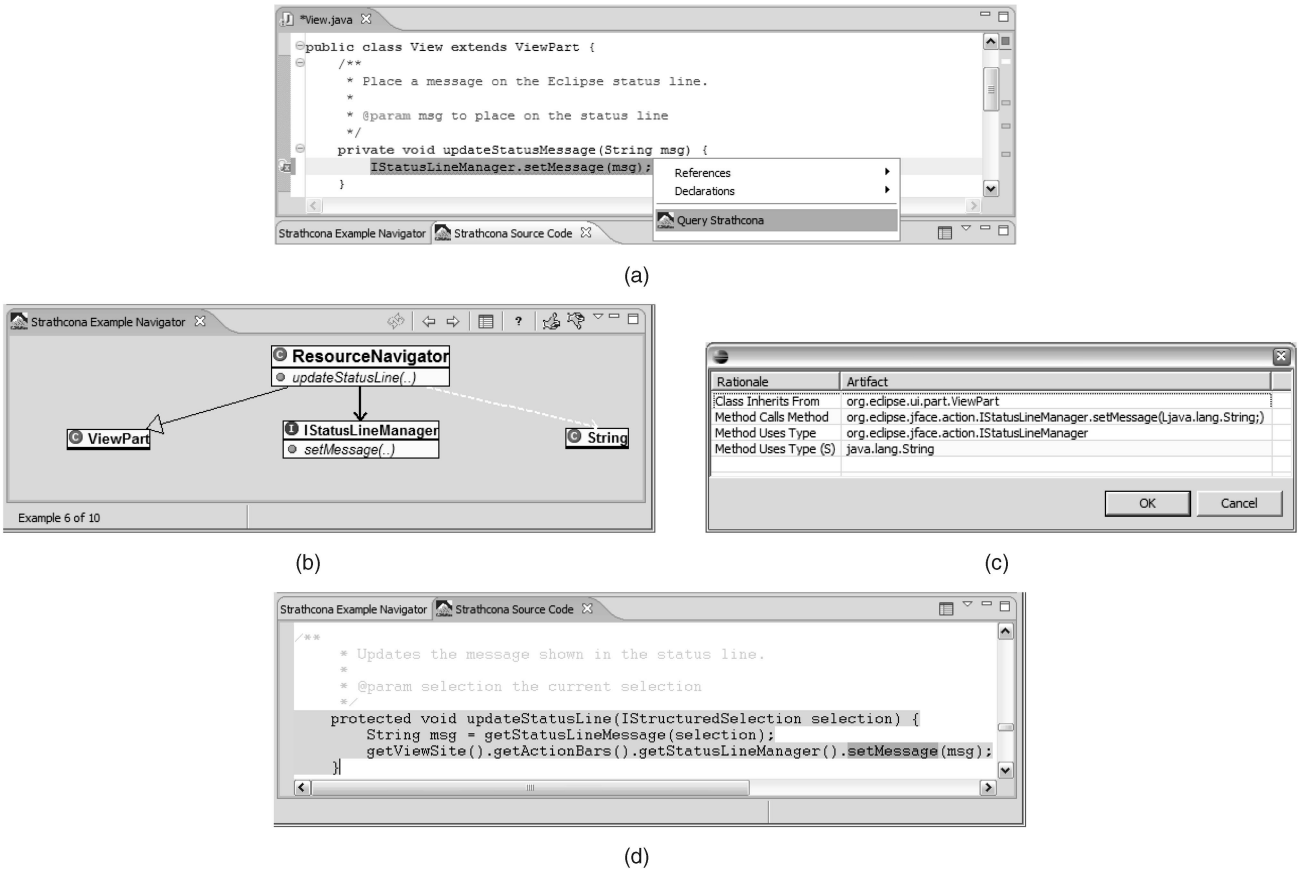

Fig. 3. Querying Strathcona for examples, and viewing results. (a) Querying Strathcona. (b) Graphical view of example's similarity to the queried source. (c) Textual view of rationale for why the example was chosen. (d) Source view.

aimed at finding code to reuse, and efforts aimed at detecting similar code.

3.1 API and Framework Helpers

Several researchers have suggested forms of documentation to ease the use of frameworks (e.g., [3], [6], [12]). However, such an approach requires a significant amount of effort on the part of the original developer to document the framework for the various ways in which a large framework may be used. To reduce this overhead, others have suggested encoding information about the intended use of the framework within the framework itself. For example, hooks [5] concisely enumerate the constraints that must be honored by the developer, the steps they must follow to use a hook, and the effect that using a hook will have on the framework itself. Although hooks are likely easier to evolve with a framework, they must still be defined manually by the framework developer.

In the Reuse View Matcher (RVM), Rosson and Carroll took the approach of providing a set of views within a development environment to describe how an application makes use of a particular class in a framework [19]. This active form of documentation relies on hand-crafted examples that can be time-consuming to create, that can become out-of-date with the code, and that may not describe uses of all of the classes in the framework.

One way to overcome the need to write and access documentation, whether external or internal to the API, is through the automated provision of examples, as in our approach. Jungloids [13] can help in the particular case of determining a possible call chain between a source type and target type. A repository of past uses of the API is mined to help guide an automated search for a call chain that has previously been used. In contrast to our approach, jungloids do not infer the developer's context, but rather require a specific source and target type to be identified. Detailed examples are not returned, merely the "best" call chain that the tool can automatically determine; thus, the developer cannot use that technique to discover additional issues that have lead others to choose particular call chains.

The CodeFinder system addressed the problem of formulating an appropriate query to a repository of examples 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 to CodeFinder, our approach aims to remove the step of formulating the query by creating the query automatically.

Other tools, such as Component Rank [11] and CodeWeb [14], [15], use software structure to determine which parts of a framework are frequently used; knowing which parts are

frequently used might help a developer determine where to begin exploring the framework to perform a desired function. Component Rank aims to find sections of code that are good targets for componentization, rather than finding examples specific to a developer's task. CodeWeb mines a repository for patterns of API use, inferring association rules (e.g., if "`A.m1()`" is called then "`B.m2()`" tends to be called as well); CodeWeb presents its results in a browser-based fashion, requiring that the developer have an idea of what kinds of association rules are of interest. In contrast, our approach targets developers who have become lost in a specific task. Our approach also does not return examples based on their frequency in the repository, but solely on their structural similarity to the source in the developer's context; as a result, examples that use an API in a unique way can be located when they are most similar to the developer's context.

3.2 Finding Code to Reuse

A related problem to helping a developer use an API is finding code that already exists in a library to meet a desired need.

The signature matching approach of Zaremski and Wing [22], for instance, attempts to find library functions to match a need by comparing the signature of a function to be written with signatures of functions in a library. The similarity of signatures is but one structural property of potential interest to a developer seeking relevant examples; our approach generalizes to structural properties other than signatures.

The CodeBroker [20], [21] system queries a repository automatically after each comment or method signature written by a developer. The queries made to the repository are based on these comments and method signatures. To retrieve matches, a developer must write comments that explain the functionality of the software in terms similar to that of the repository code [20]. When a developer follows this process, CodeBroker may be able to match a more diverse set of examples than our structural context approach. However, the effectiveness of the CodeBroker approach may be limited by the need for and difficulty of writing appropriate comments. In comparison, our approach can apply to any code and any framework irrespective of coding conventions since all source code incorporates structure.

The Automatic Method Completion technique [8] uses machine learning techniques to complete a method body based upon the developer's current context. The approach represents the programming language constructs and named identifiers used in the method as a multidimensional vector. This vector is compared to precomputed vectors, for example, code on a server, and the best completion for the method of interest is returned. Our approach differs in returning multiple possible examples for arbitrary blocks of code based on structurally similar features.

Hipikat [4] can recommend relevant development artifacts from a project's history to a developer. One kind of artifact that can be recommended is source revisions associated with a past change task; these revisions can be considered as an example. Our approach 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.

3.3 Detecting Structurally Similar Code

To support the detection of code clones and to support program evolution activities, several efforts have considered techniques to search for structurally similar code. The CloneDr detector of Baxter et al., for example, includes an algorithm for comparing subtrees of AST information [2]. Paul's SCRUPLE system provides a pattern language to describe the structural features of the code of interest to find, such as the code having nested for-loops [17]. The efforts in these areas are aimed at finding large segments of code that are near-identical at a fine-grained level of detail (such as character sequences). Our approach differs in relaxing the degree to which protoexamples must match an indicated source code fragment, concentrating on coarser-grained details (such as the presence of some method calls). In considering coarser-grained details, our approach is more similar to Basit and Jarzabek's technique for detecting the presence of structural clones, "design solutions repeatedly applied by programmers to solve similar problems" [1]. In contrast, our approach forms a structural model from a developer's context to search for relevant examples in the repository.

4 Approximate Structural Context Matching and the Strathcona Tool

In this section, we describe the concept of approximate structural context matching and its realization in the Strathcona example recommendation tool in terms of the main steps involved in determining relevant examples (see Fig. 4). Section 4.1 describes the concept of structural context and how the Strathcona client extracts it from source code selected by the developer; this structural context is sent to the Strathcona server as a query. The server contains a repository of sample source code indexed by structural context facts; the structure of this repository and how it is initially populated are described in Section 4.2. The structural context sent to the server is used to approximately match sample source code in the repository, as described in Section 4.3. The repository does not contain ready-made examples—examples must be formed from the sample source code that has been approximately matched (called protoexamples), as described in Section 4.4. These formed examples are returned to the Strathcona client for presentation to and interpretation by the developer, as explained in Section 4.5. Strathcona's runtime performance is described in Section 4.6. In each section, we describe the behavior of Strathcona through our sample scenario of Section 2.

4.1 Client: Determining Structural Context

As a first step, the developer must select some *fragment* of source code for Strathcona to find relevant examples. Selection is performed by highlighting characters in the

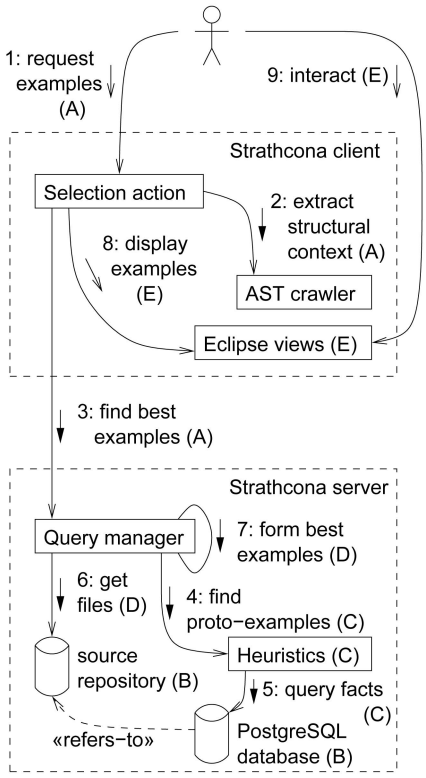

Fig. 4. Overview of the structure of Strathcona. Letters in parentheses indicate the subsections in which the given interaction or component is described. Return messages are not shown.

Eclipse Java source editor using Eclipse's standard manner of interaction.[3] The developer then requests examples from Strathcona via a context menu option.

The Strathcona client extracts the *structural context* of the selected code. The structural context is a partial characterization of the content of the selected code and its place within the greater context of its system. We realize structural contexts as sets of structural facts; the following kinds of facts are extracted by the Strathcona client:

- the method signatures declared within the fragment;
- the names of the types that declare each of those methods, called the declaring types,
- the names and fully qualified types of the fields declared by the declaring types,
- the fully qualified types, methods, and fields referenced by the fragment, and
- the supertypes of the declaring types.

For our sample scenario, the Strathcona client extracts the following structural facts:

- declaring type is `View`,
- declares method `updateStatusMessage(java.lang.String)`,
- extends `org.eclipse.ui.part.ViewPart`,
- calls `org.eclipse.jface.action.IStatusLineManager.setMessage(java.lang.String)`,
- uses `org.eclipse.jface.action.IStatusLineManager`, and
- uses `java.lang.String`.

Some of these facts may not be well-defined if the source code is not yet compilable, i.e., when the source code is still being written. For example, import statements may be missing or ambiguous so that types cannot be resolved, or as yet undeclared methods may be called. We have taken the pragmatic approach of resolving references to the extent provided by the Eclipse Java Development Tools. For instance, in our sample scenario, the code includes a call to an operation on the Java interface type `org.eclipse.jface.action.IStatusLineManager`; this fact is included as-is to see if it matches anything in the repository, rather than attempting to infer how the polymorphic call may resolve.[4]

The Strathcona client packages the structural context as an XML document and sends it to the Strathcona server as a request for relevant examples.

4.2 Server: Repository Structure

The Strathcona server contains a repository of sample source code and a database of structural facts about portions of that source code. We assume examples that are helpful for the developer querying the repository will usually occur within a single method.[5] Thus, the method declarations within the sample code serve as "protoexamples" from which relevant examples can be constructed, as described in Section 4.4.

Before Strathcona can be used, its repository must be populated with sample source code that uses the API(s) of interest. There are two restrictions placed on the code used to populate the repository: The code must be parsable by the Eclipse compiler, and the code should represent good usage of the API(s). The code may be from multiple applications.

Sample source code is loaded into the repository using an Eclipse plug-in. This plug-in walks the AST of the provided code, extracting the structural contexts of all method declarations (in the same manner that the Strathcona client extracts the structural context of a source fragment highlighted by the developer, described in Section 4.1). The facts from these structural contexts are stored in a PostgreSQL relational database; each fact indirectly refers to the method declaration which supports it. There are several tables maintained within the database (see Fig. 5): one each to track the projects contained in the repository, type declarations, method declarations, field declarations, method calls, field accesses, inheritance relationships, and type usages. (This last table is maintained for performance reasons, despite being derivable from the other facts in the database.)

The database tables are fully indexed on the basis of the attributes of the facts stored in them to provide fast search for specific facts. Fact attributes are generally stored as integer IDs instead of strings to avoid the proliferation of

3. In the current implementation, a selected fragment consists of contiguous characters within a single file.

4. See Section 4.3 for a discussion of the implications of this design.

5. Undoubtedly, interesting examples will also occur at a larger-scale. A different realization of approximate structural context matching would be needed to locate those.

```
projecttable:     id (serial; primary key),
                  name (varchar[255]),
                  date (varchar[255])

typetable :       id (serial; primary key),
                  name (varchar[255]),
                  project (integer;
                    foreign key: projecttable.id)

methodtable:      id (serial; primary key)
                  name (varchar[511]),
                  host (integer;
                    foreign key: typetable.id),
                  returnType (integer;
                    foreign key: typetable.id)

fieldtable:       id (serial; primary key),
                  name (varchar[255]),
                  type (integer;
                    foreign key: typetable.id)

callstable:       id (serial; primary key),
                  host (integer;
                    foreign key: methodtable.id),
                  target (integer;
                    foreign key: methodtable.id)

accessestable:    id (serial; primary key),
                  host (integer;
                    foreign key: methodtable.id),
                  target (integer;
                    foreign key: fieldtable.id)

typehierarchy:    id (serial; primary key),
                  child (integer;
                    foreign key: typetable.id),
                  parent (integer;
                    foreign key: typetable.id)

usestable:        id (serial; primary key),
                  host (integer;
                    foreign key: methodtable.id),
                  target (integer;
                    foreign key: typetable.id)
```

Fig. 5. The Strathcona data model.

performance-degrading strings throughout the database. For example, a method declaration would be recorded as a fact (a row) within the method declaration table and method declaration facts include three attributes: the signature (a string), the type declaring the method (an ID as a foreign key into the type declaration table), and the method's ID which is used as a foreign key in other tables.

4.3 Server: Locating Similar Protoexamples

When a query containing the structural context description arrives at the Strathcona server, the server attempts to find structurally similar code in the repository. Strathcona does not attempt to match the structure exactly; structural details that are unique to the developer's application (e.g., the developer-supplied subtypes) cannot be expected to exist in the repository. Instead, Strathcona uses a set of heuristics to find structurally relevant protoexamples.

The server passes the queried structural context independently to each of a set of heuristics. Each heuristic must implement its own concept of similarity between the queried structural context and the structural contexts of protoexamples (i.e., method declarations) in the repository; typically, a given heuristic will emphasize some kinds of facts and deemphasize others. Each heuristic returns a list of protoexamples to the server. To improve performance, the number of protoexamples returned is limited; each heuristic sorts its list according to decreasing similarity to the input structural context and returns, at most, the best 100 protoexamples. These protoexample lists are combined into a single list and sorted in decreasing order of how many heuristics returned each protoexample. The best 10 protoexamples are selected and used to form examples, as explained in Section 4.4.[6]

Our Strathcona server defines four heuristics to match a queried structural context to the structural contexts of protoexamples stored in the repository. Our four heuristics were developed 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 an existing plug-in, 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 general as possible. Each heuristic uses different facts from the structural context to locate matching protoexamples.

INHERITS. This is the simplest of the heuristics. It locates any protoexamples within the repository whose declaring type extends the same type(s) as extended by the input structural context. For our sample scenario, this heuristic searches for method declarations that have `ViewPart` as a supertype; Strathcona returns the 45 method declarations that meet this restriction.

CALLS. This heuristic queries the repository for any protoexamples that make the same method calls as described by the queried structural context. We look for protoexamples where as many method calls as possible can be matched. We rank the returned protoexamples from this heuristic in the order of the number of matching calls they make. For our sample scenario, Strathcona returns the 63 method declarations that call `IStatusLineManager.setMessage(String)`.

USES. This heuristic is essentially a relaxation of the CALLS heuristic. It considers the different types used by the input structural context. A type is considered to be used in situations where it is instantiated, a method is invoked on it, one of its fields is referenced, it is the type of a referenced field, or it is the type of a formal parameter. The protoexamples returned by this heuristic are ordered by the number of type usage facts matching the queried structural context. For our sample scenario, the two uses relationships in the queried structural context (with `String` and `IStatusLineManager`) occur in 92,362 and 204 method declarations, respectively. However, there are only 114 method declarations that use both types; the first[7] 100 of these latter method declarations are returned.

6. These limits were chosen informally. The limit of 100 protoexamples per heuristic is large enough that good protoexamples never appear to be dropped, but small enough that performance is improved. The limit of 10 protoexamples returned to the client was chosen as a reasonable number for the developer to quickly scan through and determine whether the selected fragment will result in useful recommendations.

7. As defined by the PostgreSQL server implementation.

```
select count(*), host from (
  select host from usestable where target in (
    (select id from typetable where name=
     'java.lang.String'),
    (select id from typetable where name=
     'org.eclipse.jface.action.IStatusLineManager'))
) as results group by host order by count(*) desc
limit 100;
```

Fig. 6. The compound PostgreSQL query used by the USES heuristic for the sample scenario.

REFERENCES. This heuristic returns method declarations that use the same field reference(s) as the queried structural context. This heuristic was created because constants are often more significant for their name than their type (which is frequently just an `int` or a `String`). For our sample scenario, the structural context does not reference any fields, and this heuristic does not return any results.

Although each of the heuristics are run independently, their results must be combined in order to rank and select the most relevant matching protoexamples. The Strathcona server ranks matched protoexamples according to the number of heuristics that returned them; this means that any particular protoexample can have a maximum score of 4 in the current implementation. For our sample scenario, the top four method declarations (i.e., protoexamples) were returned by three heuristics, while the next six were returned by only two. This means that of the 45 results from the INHERITS heuristic, the 63 from the CALLS heuristic, and the 114 best results from the USES heuristic, only four method declarations were matched by all three heuristics and, as such, were ranked as most relevant.

The PostgreSQL query language provides efficient queries that can simultaneously locate protoexamples matching certain facts, sort these in order of decreasing similarity, and select the top 100 protoexamples, all in a single query. An example of the compound query used by the USES heuristic is shown in Fig. 6. The IDs for `String` and `IStatusLineManager` are found from the type declaration table (*`typetable`*). Next, uses-facts involving either of these types are located and stored in a *`results`* table. The *`host`* attribute of these facts contains the ID of the method declaration that uses one of these (*`target`*) types. The uses-facts are then grouped by the method declaration ID (*`host`*), the *`results`* are sorted in descending order of how many matched uses-facts refer to the same *`host`*, and the *`results`* are limited to the top 100 matches. The method declaration IDs remaining in the *`results`* table serve as protoexamples to be returned by the USES heuristic. In early implementations of Strathcona, we found that locating protoexamples with the multiple-queries approach instead of the compound-query approach tended to be up to 100 times slower, thereby eliminating interactive rates of response.[8] The compound-query approach has implications for the formation of examples, as described in the following section.

8. The PostgreSQL query planner can optimize compound queries in a way not possible when given many small queries in series.

```
select id from usestable where
  target=<a specific type declaration ID> and
  host=<method declaration ID of a specific proto-example>
```

Fig. 7. Rationale determination by the USES heuristic. This query is run once for each uses-fact in the queried structural context.

4.4 Server: Example Formation

Once the Strathcona server has selected the 10 most relevant protoexamples, it must create examples from each to return to the Strathcona client. From the perspective of the Strathcona server, an example consists of an XML document containing the *rationale* for why this protoexample was selected (a set of strings), the structural facts of the protoexample that are similar to those of the queried structural context, and the signature of the method declaration serving as a protoexample. These details are returned to the Strathcona client for further processing and display (see Section 4.5); the client can also then request the source file from whence the protoexample came.

Rationales are required only to help the developer understand why an example was recommended; therefore, rationale determination can be delayed until the 10 most relevant protoexamples have been determined. In practice, this is more efficient (via the compound-query approach described above) than the alternative of tracking rationales for every matched protoexample.

For each of the 10 protoexamples, the server implementation requests rationales from each of the heuristics; this request includes the protoexample and the queried structural context as arguments. The returned sets of strings are unioned to provide the returned rationale for the example. Each heuristic also returns the facts about the protoexample that it considers similar to the queried structural context; these are combined into an XML document to be returned to the client.

In the current implementations of heuristics, each rationale is a string representing a fact from the queried structural context that is also supported by the selected protoexample. Thus, the content of the set of structural facts returned in the example and the content of the set of rationales will be equivalent.[9] In our sample scenario, the USES heuristic returned 114 method declarations that used both `String` and `IStatusLineManager`; the USES heuristic's rationale for selecting these 114 method declarations would consist of the two statements "`Method uses type IStatusLineManager`" and "`Method uses type String`."[10]

An example of the PostgreSQL query used for determining rationales is shown in Fig. 7 for the USES heuristic. Each uses-fact in the queried structural context would be tested against the protoexample: uses-facts containing (*`target`*) IDs corresponding to the types used in the queried structural context (`String` or `IStatusLineManager` here) and containing the (*`host`*) ID of the protoexample

9. Heuristics could be implemented to provide rationales of other forms, so this redundancy is retained for the sake of flexibility.

10. In the implementation, these type names are actually fully qualified, and the second statement indicates that String is a "secondary" type (one with low specificity) as exemplified in Fig. 3c.

would be queried as shown. If the size of the result of the query were 0, the uses-fact would not match the proto-example and would not be part of the rationale for why the protoexample had been selected. If the size of the result of the query were greater than 0, the uses-fact would be part of the rationale, and the USES heuristic would add the string "`Method uses String`" or "`Method uses IStatus-LineManager`" to its returned rationale. An empty string would be returned if all of these queries were of size 0. The returned strings from all heuristics are concatenated and added as the rationale for a recommended example.

4.5 Client: Example Presentation

The list of generated examples is returned to the Strathcona client for presentation to the developer.

The Strathcona client provides a limited UML-like class diagram representation (e.g., Fig. 3b) to help the developer to quickly judge if a recommended example is worth examining more closely. Since Strathcona does not attempt to judge which facts in the queried structural context are more important, recommended examples may be missing facts that the developer considers crucial. This view displays the structural facts of the returned example, always including the declaring type of the example and the name of the method declaration serving as the basis for the example.

The developer can also view the textual rationale description as a list of strings (e.g., Fig. 3c). In the current implementation of heuristics, the structural facts added to the example are equivalent in content to the strings appearing in the rationale and, thus, the textual rationale and graphical diagram encode the same knowledge.

If the developer wants to see the source code for the example (i.e., if the developer considers the example to be promising), the Strathcona client requests from the server the source for the file containing the protoexample of interest. The client trims the source file returned from the repository to display only those portions that are relevant to the example being displayed (e.g., Fig. 3d), including package declarations, import declarations, and the implementation of the method declaration serving as the basis of the example. Portions of the source that match facts in the queried structural context are highlighted.[11] The developer can also view the untrimmed file if they want to see additional details of the file from which the example originated.

The Strathcona client displays matched examples and their source, one at a time, in a separate view from the one in which the developer initiated the query. The separation of views allows the developer to query, navigate, and investigate each example without losing the context of the task that they were originally trying to solve.

4.6 Server: Performance

Within this paper, we describe the use of two versions of the repository, each of which contained the Eclipse SDK platform and several subsidiary projects that depend on this platform, specifically, the Graphical Editing Framework, Visual Editor, Web Tools, UML2 Metamodel, Eclipse Modeling Framework, and the XML Schema Infoset Model. Our study in Section 5.2 was based upon a version of the repository that had been populated with Eclipse Version 3.0M8. The studies in Sections 5.3 and 5.4 were performed at a later date, and the repository had been updated to Eclipse Version 3.2M5. Table 1 lists the size of each of the tables within this latter database version, the Java source code for which comprises approximately 3.7 MLOC (195 MB); the database itself occupies 462 MB.

TABLE 1
Number of Structural Facts in the Repository

Class declarations	38,827
Field declarations	148,437
Method declarations	321,344
Inheritance relations	39,258
References relations	594,231
Calls relations	887,603
Uses relations	1,124,034
Total	3,153,734

As it is currently implemented, Strathcona seems to scale well. As shown in Table 1, our database contains more than 3,000,000 facts. Strathcona typically takes between 0.3 and 3 seconds (depending on the size of the queried source code fragment) to generate and return its 10 examples. The construction of the structural context for the query and display of the examples are both extremely fast, taking less than 100 ms each.[12]

Because Strathcona's repository should be accessed far more often than having new code added, we have indexed the tables in the database to favor read access. Three factors can slow Strathcona's response rate. First, a large repository could lead to slow responses; to date, we have found this manageable. Even with 3.7 MLOC of code indexed, we are able to return examples in less than 3 seconds in the worst case. Second, the size of the queried structural context might slow the response time. Again, we have found this manageable since most queried contexts tend to be small, less than 10 facts. Finally, the nonspecificity of each fact in the structural context could degrade the performance of the tool. In our running example, we saw that `String` was used 92,362 times in the repository, while the more specific type, `IStatusLineManager`, was used only 204 times. When querying on code that uses only general types such as `String` or `int`, Strathcona takes longer to generate and return examples.

Strathcona's performance might be improved by using knowledge of the queried context to select which heuristics to run, by querying with the most specific facts first and applying less specific facts only if necessary, and by improving the way in which examples are cached for future use. However, we have not implemented these

11. Currently, this highlighting is done lexically by searching the source for the trimmed method for key identifiers appearing in the structural facts of the example. Obviously, this can lead to false highlights, but works sufficiently well in practice for the proof-of-concept implementation.

12. The server processing the queries was a Pentium 3 800 MHz machine with 1 GB RAM, and the workstation housing the database was a Pentium 3 1000 MHz machine with 1 GB RAM.

measures as we have not found them necessary to obtain good performance.

5 Evaluation

In evaluating our approach, we sought to address three key questions. Do our structural matching heuristics return examples that a developer can interpret to complete tasks? Can a developer use Strathcona to locate useful examples with less effort than other alternatives? Can a developer with little specific knowledge of an API reasonably create source code fragments for which Strathcona will return useful examples?

To help answer these questions, we performed four empirical investigations:

1. a qualitative, semicontrolled experiment (Section 5.2) involving two developers who were inexperienced with the API of interest,
2. a quantitative experiment (Section 5.3) into the effectiveness of Strathcona relative to a lexically based search tool (grep) and a structurally aware search tool (Eclipse search),
3. a quantitative experiment (Section 5.4) into the relationship between the quality of fragments supplied by the developer and the relevance of examples returned by Strathcona, and
4. a qualitative case study of the use of Strathcona by an industrial developer as part of their daily work (Section 5.5).

A summary of our findings in relation to the research questions of interest and a discussion of their validity is given in Section 5.6. Two of these studies worked from a set of standard tasks; we begin by describing these standard tasks in Section 5.1.

5.1 Development Tasks

We selected four tasks as useful for evaluating the strengths and weaknesses of Strathcona. These tasks came from our experience developing Strathcona and involve separate parts of the Eclipse API. The repository contained relevant examples for Tasks 1, 2, and 4. At the time the study in Section 5.2 was conducted, no relevant examples existed in the repository for Task 3. For each task, we have considered which details a perfect solution would need to possess and the relative importance of each detail. Examples returned by Strathcona can then be assessed as to whether they provide the full set of these details or some subset, and each example can be graded. Descriptions are given below for each task and which details of a solution matter to complete the task. A grading scheme for examples is described for each task, rating examples from "A" (excellent) to "F" (irrelevant) depending on which details are displayed by an example.

Task 1: Update status line. The first task involves displaying text in the status line of the Eclipse IDE, as described in Section 2. Although this task is conceptually simple, it requires a chain of method calls accessing objects of a variety of types, including `IViewPart`, `IViewSite`, `IActionBars`, and `IStatusLineManager`. By investigating the Eclipse documentation, the developer can discover enough information to construct the code fragment shown in Fig. 8a.

Determining how to access an instance of `IStatusLineManager` to be able to call `setMessage(String)` on it is key to this task. Examples can thus be graded on the basis of whether they demonstrate how to access a series of objects culminating in the call to `setMessage(String)`. The grading scheme is as follows: (A) demonstrates the complete calling sequence from `IViewPart.getViewSite()` to `IViewSite.getActionBars()` to `IActionBars.getStatusLineManager()` to `IStatusLineManager.setMessage()`; (B) does not demonstrate how to access an instance of `IViewSite` but the remainder of the calling sequence is shown; (C) does not demonstrate how to access an instance of either `IViewSite` or `IActionBars`; (D) demonstrates only a call to `setMessage(String)` on an instance of `IStatusLineManager` but gives no hint as to how to obtain this instance; (F) none of the needed information is given.

Task 2: Create AST. This task involves building an Abstract Syntax Tree (AST) from a source string. A search of the Eclipse documentation indicates that `ASTParser.setSource(char[])` should provide the appropriate functionality. However, three things are required: A factory is needed to create the parser, the parser needs to have access to the appropriate source code, and the AST needs to be generated. The fragment shown in Fig. 8b can be determined from the Eclipse documentation.

The grading scheme for examples for this task is as follows: (A) creates a parser, calls `setSource(char[])` on this parser using the source string, and creates a compilation unit; (B) sets the parser's source from some other input (e.g., a classfile or another compilation unit); (C) same as an A or a B but obscured by extraneous functionality; (D) uses the relevant types but does not demonstrate any details of completing the task; (F) completely irrelevant.

Task 3: Highlight source text. The third task involved highlighting instances of method invocations in a code viewer using the AST representation generated in Task 2. To solve this task the developer would need to understand how to manipulate `StyleRanges`, `TextViewers`, and `Colors`. Fig. 8c lists the fragment for this task that can be determined from an examination of the Eclipse documentation.

The grading scheme for examples for this task is as follows: (A) demonstrates how to construct `StyleRange` objects and `Color` objects, and how to apply the `StyleRange` to the `TextViewer`; (B) only shows how to construct and apply `StyleRange` objects; (C) only shows how to apply `StyleRange` objects; (D) uses one or more of the relevant types, but nothing more; (F) is completely irrelevant.

Task 4: Generate method signature. This task was the most complex of the four. Using an `ASTVisitor`, the developer was asked to extract the signature and modifiers for each method declaration in the AST and to output this information as a formatted `String`. This task requires the use of several

```
public class SampleView extends
ViewPart {
    private void setMessage(String msg) {
        IStatusLineManager.setMessage(msg);
    }
}
```

(a)

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

(b)

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

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

(c)

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

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

(d)

Fig. 8. Source fragments identified as relevant to each task. These fragments were provided to the subjects for the study in Section 5.2. (a) Fragment for Task 1. (b) Fragment for Task 2. (c) Fragment for Task 3. (d) Fragment for Task 4.

Eclipse framework types including `Type`, `PrimitiveType`, `ArrayType`, `Name`, `SimpleName`, `QualifiedName`, `Code`, `Flags`, and `SingleVariableDeclaration`. The task is complicated by the fact that formatting the modifiers can be done either manually via a large `case` statement or via a utility method in the `Flags` class; the latter solution is deemed preferable. Fig. 8d shows the fragment for this task that can be determined from the Eclipse documentation.

The grading scheme for examples for this task is as follows: (A) shows how to extract the return type, formal parameters, and modifiers, handles special cases (such as parameters that are arrays), and demonstrates how to format the modifiers via the utility method on the `Flags` class; (B) shows how to extract everything but does not handle special cases and performs modifier formatting via a `case` statement; (C) shows how to extract all the information from the AST, but does not format the `String` for output; (D) uses some of the relevant types or methods, but neither extraction nor formatting is fully dealt with for even standard cases; (F) completely irrelevant.

5.2 Does Strathcona Provide Useful Examples?

Before we can compare Strathcona to other tools, we need to verify that Strathcona can indeed return relevant source code examples to developers. To answer this question, we performed an experiment in which we asked two developers, with little experience in using the Eclipse API, to complete tasks, described in Section 5.1, involved in developing a plug-in for the Eclipse IDE. We monitored the developers to see whether and how they used examples provided by Strathcona.

TABLE 2
Novice Developer Study Results

	Useful examples	Source viewed	Task successful	Rank of used example
Task 1				
Subject 1	1	1	yes	1
Subject 2	1	1	yes	1
Task 2				
Subject 1	1	2	yes	1
Subject 2	1	6	yes	1
Task 3				
Subject 1	0	2	yes	none
Subject 2	0	6	yes	none
Task 4				
Subject 1	1	2	yes	1 (and 2)
Subject 2	0	7	partially	none

Both subjects had some plug-in development experience. Subject 1 had less than one month of Eclipse plug-in development experience but more than eight years of Java experience. Subject 2 had over six months of Eclipse plug-in development experience but only 18 months of experience with Java.

At the start of the exercise, the subjects were each provided with a one-page document describing the four tasks assigned and how to use Strathcona. In addition, a method skeleton was provided for each task within which they could develop their solution. The skeletons were populated with code the developer would likely write to accomplish their task. The code consisted of method calls on types and references to methods on interfaces, as shown in Fig. 8. These methods and types were identified using the Eclipse documentation and code completion features, simulating facts any developer could extract from the documentation. We seeded this initial information to the subjects to keep the time needed to complete the study reasonable.

Neither subject had prior knowledge of how to implement any of the assigned tasks. The standard Eclipse Java Development Tools (Eclipse JDT) was available to the subjects as they worked on each task. The tasks were completed in the same order by each subject. Each subject was given a maximum of three hours to complete the four tasks.

5.2.1 Results

Table 2 summarizes the results. For each task, the table shows how many of the 10 examples returned by Strathcona the two subjects deemed useful for the task through their direct use of code fragments and the number of examples for which the subjects viewed the source. The table also shows whether or not the subject was successful at completing the task. For each task, there may have been additional relevant examples, but we indicate only those that were used by the subjects.

Task 1: Update status line. By examining the graphical diagram of the first returned example, both subjects determined that it would be useful to complete the task.

Both subjects copied code from the example into their editor, changed the argument for the `setMessage` method, and ran the code to test it. The example used was returned by all but the REFERENCES heuristic.

Task 2: Create AST. Both subjects again selected the first example returned as it demonstrated the use of the method of interest and included code to set up the parser and create the AST. Again, both subjects used the graphical diagram to determine that the example warranted closer examination. The example they selected was returned by the CALLS and USES heuristics. The second subject investigated the code snippets for a number of examples before deciding that the first one was the most relevant to the task. As for the first task, both subjects integrated code from the example into their source to complete the task. The code fragment provided with the example contained two extraneous calls that the subjects dealt with differently; one copied all of the code and then deleted the extraneous sections, while the other subject only copied the sections of code that were relevant. (This subject studied the documentation for each method call in the code fragment to figure out what it did before it was copied.)

Task 3: Highlight source text. Strathcona was not able to return any useful examples for this task, as none existed in the repository. We deliberately included this task to determine if subjects could identify when the returned examples were insufficient. If the subjects are able to recognize unhelpful examples, we have more confidence in their assessment of returned examples.

The subjects both stopped examining the examples provided within 15 minutes and implemented the feature using the standard IDE tools. As the developers could not identify relevant examples using the graphical diagram or rationale view, they looked at the source code for several examples. Interestingly, both subjects independently decided to use Strathcona as part of this task to find an example of how to create an SWT `Color` object; both subjects used some code from the returned examples to accomplish this portion of the task.

Task 4: Generate method signature. Subject 1 investigated `MethodDeclaration` using the Eclipse code completion feature to try to derive a working solution before using Strathcona; they expressed concern about not finding a relevant example as had happened for Task 3. After querying Strathcona, the subject examined the graphical diagram and rationale view for the first few of the examples carefully before deciding which two examples to investigate. The subjects used the rationale view more for this task than all the other tasks combined. The structural context contained many elements here, and it was easier for the developer to determine the amount of similarity to their query with the rationale view compared to the graphical diagram. The first two examples returned for this task matched four method calls in the code used to query; the remaining examples matched at most two method calls. After investigating the source from the second example, the subject discarded it and moved on to the first example. This example matched several calls, as shown by its rationale for selection (see Table 3[13]). When viewing the code, this example had one 56-line method highlighted; this method also used several private utility methods that totaled 61 lines of code. The subject proceeded to copy code from the example in small sections and completed the task successfully. The example that the subject selected to complete the task was returned by all but the USES heuristic.

TABLE 3
Task 4 Rationale

Class has parent of type	`ASTVisitor`
Method Calls Target Method	`MethodDeclaration.getName()`
Method Calls Target Method	`MethodDeclaration.parameters()`
Method Calls Target Method	`SimpleName.getIdentifier()`
Method Calls Target Method	`BodyDeclaration.getModifiers()`
Class uses Class	`MethodDeclaration`
Class uses Class	`SimpleName`
Class uses Class	`BodyDeclaration`

Subject 2 mistakenly queried the repository on the wrong method and searched through several irrelevant source files before deciding to implement the feature manually. The subject was partially successful but was unable to extract some parts of the signature from the AST.

Summary. Subject 1 completed all four tasks successfully, finding and using relevant examples in all three cases for which appropriate examples were returned. Subject 2 completed three out of four tasks, finding and using relevant examples in two of the possible three cases. In each of the tasks where the subject found a relevant example, source code was copied from the example into the task code. These results show that our tool can deliver relevant and useful examples to developers. They also show that a developer can determine when the examples returned are not relevant.

5.3 How Does Strathcona Compare to Other Tools?

Existing traditional search tools used by developers can be placed into two categories: those that perform textually based pattern matching and those that incorporate knowledge of syntax and structure. As representatives of these categories, we have considered how Strathcona compares to the use of grep and to the use of Eclipse's search facilities, each of which can be used to perform searches on a large repository of source code. To this end, we considered whether grep and Eclipse Search would perform as well as Strathcona for locating relevant examples; we utilized the development tasks described in Section 5.1 to compare the tools.

Comparing grep to Strathcona. To compare the use of grep to the use of Strathcona, we first wished to determine the manner in which an expert user of grep would use it on these development tasks. To this end, we invited a volunteer who had regularly used grep in industry for many years, to complete the tasks. This volunteer had read about Strathcona and was skeptical that it could outperform the use of grep. We gave the volunteer the same task descriptions and starting fragments as the subjects of the

13. Each name was actually preprended by the package name, `org.eclipse.jdt.core.dom`.

```
find . -name "*java" | xargs grep -l "ViewPart" | xargs grep -l
"IStatusLineManager" | xargs grep -l "setMessage("
```

Fig. 9. Sample query using grep to mimic structural context searches.

study in Section 5.2, plus direct access to the source files stored as part of the repository; details of the volunteer's use of grep were recorded. For Task 1, the volunteer located methods that returned an instance of `IStatusLineManager` and investigated the implementation of these methods to determine what other information was needed to be able to return the instance. Essentially, the volunteer attempted to work backward from the final call in the call chain without considering what the starting place needed to be. After 35 minutes of constructing complex grep queries, the volunteer gave up without having located an example they recognized as useful and without having completed the task; the other tasks were not attempted. From this experience, it seems clear that unconstrained use of grep does not easily lead one to search on the relevant facts.

We then wished to know whether a developer could reasonably mimic the structural-context approach of Strathcona using grep. With an understanding of the structural context approach underlying Strathcona, we formed grep queries for the facts that Strathcona extracted from the sample fragments shown in Fig. 8. The grep query that corresponds to the Task 1 fragment is shown in Fig. 9. These queries locate those files that match every fact from the structural context.

There are two key problems with the results of using grep in this manner. First, any returned file must match *every* fact from the structural context; this is a more stringent definition of similarity than used by Strathcona. Second, these grep queries only return files, whereas Strathcona's results return examples, including a graphical depiction of the structure that is similar and a source code fragment with its similar structure highlighted. Thus, the results of grep are but an initial step toward identifying and judging potential examples.

Thus, we considered whether grep can be used to inform the developer that an example occurs *somewhere* within the body of a file. To this end, we used grep-based queries, like that in Fig. 9, to mimic the search for structural context as performed by Strathcona. Such queries were formulated on the structural contexts from the fragments in Fig. 8; the results are shown in Table 4 relative to Strathcona's results for the same fragments. For each task, the table compares

TABLE 4
Structural-Context-Based Search: Quality Comparison

Task	Tool	A	B	C	D	F	Total Results
1	Strathcona	7	1	0	0	2	10
1	grep-based	2	0	1	3	0	6
2	Strathcona	10	0	0	0	0	10
2	grep-based	37	3	0	1	0	41
3	Strathcona	2	0	1	1	6	10
3	grep-based	2	0	0	0	0	2
4	Strathcona	3	2	1	1	3	10
4	grep-based	3	3	0	4	2	12

TABLE 5
Structural-Context-Based Search: Result Size Comparison

Task	Tool	Results	Average Size	Total Size
1	Strathcona	10	8	75
1	grep-based	6	422	2,532
2	Strathcona	10	26	260
2	grep-based	41	516	21,160
3	Strathcona	10	35	350
3	grep-based	2	1638	3,276
4	Strathcona	10	41	406
4	grep-based	12	790	9,480

the grades assigned to the examples produced by Strathcona (as described in Section 5.1) to the best example occurring in each file returned by the grep-based query; there were no cases of multiple examples existing in the same file. The column for each grade shows how many examples existed with that grade, and what percentage of the total number of results received that grade. The total number of results—i.e., examples returned by Strathcona (always limited to 10 examples) and files returned by grep-based queries—is shown in the last column. This study was conducted on a later version of the repository than that used for the study in Section 5.2: Good examples for Task 3 existed in this later version.

Table 4 shows that grep-based queries for structural context were somewhat effective at locating files containing useful examples. For Task 1, the grep-based query failed to identify the presence of several good examples, but identified a number of low-quality examples instead; since the order of results from a grep-based query is meaningless, all of these results would need to be examined. Strathcona and the grep-based queries are similar for the other tasks, considering that Strathcona will always try to return 10 examples, even when some of those are poor matches —the detailed rationale provided by Strathcona will enable the developer to determine if the match is good or not. As our repository contained over 3 MLOC, we were unable to determine if any examples were missed by either approach. For the two most complex tasks (3 and 4), both techniques located the same A-graded examples.

While Table 4 details the effectiveness of each approach to locate relevant examples, Table 5 shows the amount of information a developer must consider using each approach in order to determine the relevance of each example. Table 5 lists the average and total noncomment, nonblank lines of code returned by each approach. As grep returns whole files for a developer to examine, this count was taken from the whole file, while Strathcona's totals were derived from the highlighted portion of the code shown to the developer. Table 5 shows the large difference in the scale of the information that the developer must navigate in order to determine the relevance of a returned example: on the order of 10 to 100 times as many LOC result from the grep-based approach.

Comparing Eclipse Search to Strathcona. To use Eclipse Search—which does not support compound queries—for the tasks, we had to decide which single fact from the fragments in Fig. 8 was most important to each task we

TABLE 6
Fragment Specificity Results

Case	Facts	Total	Matches	(%)	Average Rank
1	8	256	164	(64%)	4.30
2	12	4096	3944	(96%)	1.30
3	11	2048	1784	(87%)	1.16
4	11	2048	1474	(72%)	1.44

were trying to solve. For each query, we selected the appropriate `Search For` radio button (a choice between Type, Method, Package, Constructor, or Field) and `Limit To` radio button (e.g., Declarations, References, or All Occurrences) in order to return the most relevant results.

For example, we attempted the first task by searching for `IStatusLineManager.setMessage(String)` with the `Search For` flag set to `Methods` and the `Limit To` flag set to `References`. Eclipse Search returned 101 examples, of which only 11 percent were grade A quality; the results were not ordered according to quality. To be tractable for searching for specific examples, Eclipse Search would need to be extended to support compound searches. Until that happens, its results are clearly inferior for this purpose; therefore, we abandoned further investigation of Eclipse Search.

Summary. Existing traditional search approaches are inferior to Strathcona for locating relevant examples of the use of an API. Eclipse Search is not implemented to provide the basic functionality needed to find examples that display multiple facts of interest. Standard ways of using grep do not lead to the location of good examples. While the use of grep can be constrained to mimic a search for structural context, it cannot present examples in a precise manner, but merely points to files—in random order—in which there might be a relevant example. Even when using grep in a structural-context-based manner, its use for discovering relevant examples remains a laborious and time-consuming task.

5.4 Is It Hard to Create Effective Fragments to Query On?

To successfully locate a relevant source code example with Strathcona, the developer needs to come up with a query that references one or more program elements that are involved in potential examples. Our first study (in Section 5.2) did not require the subjects to create source code fragments (although for Task 3, both subjects did create their own source code fragments). To investigate the ability of Strathcona to return appropriate examples for a range of structural context queries (its "robustness"), we performed an analytic experiment. In particular, we wanted to investigate the effect of size and content of queries on Strathcona's responses.

We selected code fragments at random from our repository to serve as the basis for forming queries until we had found four fragments that were each between 10 and 20 lines long. Each of the four fragments was then modified to eliminate information that would allow Strathcona to unfairly locate the original fragment—specifically, references to any private members that a developer could not reasonably utilize within their source fragment were eliminated.[14]

The structural contexts of each of these modified fragments was then extracted. We used every subset of each structural context as a query on the Strathcona server to see if it could locate the original code fragment. We chose to look for an exact match to the original code fragment in the results returned, rather than examine each returned result for relevance, to keep the analysis tractable. This choice represents a worst case in assessing relevance.

Results. Table 6 shows how often a combination of facts retrieved the desired example. For each case, the Facts column indicates how many facts existed in the structural context extracted from the modified fragment, and the Total column represents the number of subsets of the facts within the structural context that were used as queries to the Strathcona server. The Matches column indicates how many of the queries sent to the Strathcona server resulted in the original fragment being returned and the "percent" column indicates the percentage of the total queries that resulted in positive results. Finally, the Average Rank column indicates the average rank (between 1 and 10) of the positive results in the order of returned examples.

The data in this table shows that many different subsets of a structural context can be used to find the same context within the repository. Specifically, in two of the cases, 87 percent and 96 percent of the subsets return the sought after example. Strathcona can successfully match many different structural contexts to find the same example. In the two worst cases—shown in the first and last rows of the table—the correct example was returned only 64 percent and 72 percent of the time, respectively, because the code fragment for these cases used portions of the API that are commonly used by many parts of the system. Each of the remaining cases utilized portions of the API that were less commonly used.

We also used these four cases to investigate whether the size of the context changes the effectiveness of Strathcona. Fig. 10 and Fig. 11 present the results, showing the relationship between the size of the query and how often the original example is retrieved. Fig. 10 treats all facts equally—even those that are obviously poorly specific, such as calls to `String`. Fig. 11 only considers those facts that are of key significance to the task, referred to as primary facts.

Fig. 11 demonstrates that even when the developer provides structural contexts containing only one or two important facts, Strathcona can often return the original source code fragment as an example. Knowing four or more important facts in these cases ensures that this queried

14. The selected and modified fragments consisted of: 1) the method `match(AnnotationTypeDeclaration, Object)` from the class `ASTMatcher`, 2) the method `getVersionedIdentifier()` from the class `Feature`, where lines referencing the private field `versionId` were commented out; 3) the method `searchForRuntimes(IPath, IRuntimeSearchListener, IProgressMonitor)` from the class `RuntimeLocator`; and 4) the method `decodeJRELibraryClasspathEntries (String)` from the class `NewJavaProjectPreferencesPage`, where the highly specific references to the static field `NewJavaProjectPreferencePage_error_decode` on the class `PreferencesMessages` were commented out.

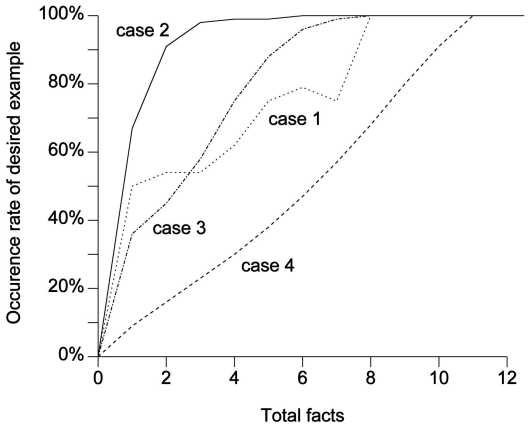

Fig. 10. Structural context size (number of facts) versus query effectiveness (occurrence rate of the desired example).

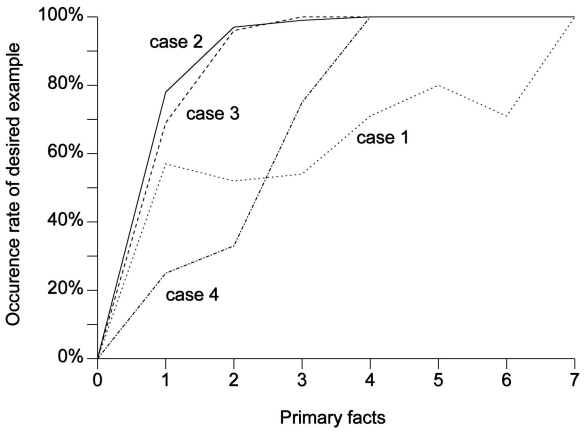

Fig. 11. Structural context size (number of primary facts) versus query effectiveness (occurrence rate of the desired example).

fragment is returned. Fig. 10 shows that, even in the presence of largely irrelevant facts, good examples come back, though with lower probability.

Summary. The results of this study show that our heuristics are robust to various forms of queries, successfully matching various input structural contexts to their original source code fragment. Our analysis also demonstrates that even small input structural contexts can successfully locate relevant examples in many cases. These properties of Strathcona help reduce developer frustration by minimizing situations where a query does not successfully locate a relevant source code fragment. From this evaluation, we can recommend to the developer that small, specific queries are better than larger, more general ones. This fits with how Strathcona is to be used, as a developer who is lost will likely have few extraneous facts at their disposal.

5.5 How Does Strathcona Fit into an Industrial Work Flow?

Industrial developers use a mixture of tools to help them complete their tasks as quickly as possible. To see how Strathcona might be used in this context, we provided the tool to an industrial developer who was unfamiliar with the approach prior to the study. This developer had a specific task he needed to solve: He wanted to create an Eclipse plug-in that provided a visual component to locate all of the HTML files within a certain directory (and its subdirectories) and that could support opening a Compare Editor on any pair of selected files to be able to visually see differences between them.

To create the plug-in, the developer first used wizards provided by Eclipse to create a Sample View. He then used part of the code generated by the wizards—the code that initialized the view (`createPartControl(Composite)`)—to query Strathcona. By looking through the examples returned by Strathcona, he discovered three things: how to declare a simple sorter, the usage of several APIs frequently used in that context (including `setAutoExpandLevel(TreeViewer.ALL_LEVELS)` and `setUseHashLookup(true)`), and how to solve a bug from the previous day (a call to `treeViewer.addSelectionChangedListener(this)` was missing).

He then tried to use Strathcona to find examples that would return the location of the root of his workspace. He needed to identify the call chain `ResourcesPlugin.getWorkspace().getRoot().getLocation()`. He queried Strathcona with an empty method declaration called `getWorkspaceLocation()`. This query did not contain useful structural context; because of this, Strathcona did not locate any relevant examples. To find an example, he instead queried Google with the phrase "compute the root of the Eclipse Workspace." Fortunately, an article had been written that described the solution; the fifth result returned by Google was an article that contained the information he desired.

We believe his attempt to use the empty method declaration as a query indicates he did not fully understand how Strathcona was intended to be used. The terms he queried Google with were conceptual, not containing any framework types or methods. However, a search through the Eclipse API documentation yields types (e.g., `Workspace` and `IWorkspaceRoot`) that can be used to query Strathcona to find relevant examples.

At this point, the developer had discovered how to create a tree view and how to populate it with the HTML files from the workspace. Next, the developer needed to discover how to launch the `CompareEditor`. By using Google, he determined that `CompareUI.openCompareEditor(...)` performed the operation of interest. By querying Strathcona with that fragment, he determined how several other projects have set up and used `CompareEditors`.

To complete his task in a reasonable amount of time, the industrial developer used a variety of tools (Google, Eclipse search, Eclipse autocomplete, and Strathcona). Google frequently pointed him toward some initial useful APIs as it has indexed the Eclipse Javadoc documents, newsgroups, and mailing lists. From these initial Google-provided APIs, the developer was able to query Strathcona. By looking through the examples provided by Strathcona, he noticed other APIs that were frequently used in conjunction with those he was already investigating. The developer also used

Strathcona to query about types of interest that he learned about through other searches; in these cases, Strathcona provided the developer with contextual usage information about how to best set up and use the types. Overall, the developer was able to incorporate Strathcona into his workflow and he found that it provided him with additional information (the examples) that was more difficult to access with traditional techniques.

Based on his experience, the developer suggested two improvements to Strathcona: to support text-based searches on the repository and to provide better support for the navigation of an example's source code (equivalent to that provided by Eclipse for projects in the workspace). These two features are provided by the lexically based, online systems Krugle[15] and Koders.com.[16] However, these systems cannot recommend examples relevant to a developer's current task since they do not leverage the structural information embedded in the source code. Ultimately, combining such features with Strathcona should result in an industrial-strength tool.

5.6 Evaluation Summary

We wanted to answer three questions with our empirical investigations. Do our structural matching heuristics return examples that a developer can interpret to complete tasks? Can a developer use Strathcona to locate useful examples with less effort than other alternatives? Can a developer with little specific knowledge of an API reasonably create source code fragments for which Strathcona will return useful examples? We describe how the combination of results from our studies provide answers to these questions and then discuss the limitations of these empirical investigations.

Aggregate results of empirical studies. Our first question considered whether the structural matching heuristics encoded in Strathcona return examples that a developer can interpret to complete tasks. In essence, this question focuses on the usefulness of the tool's output (i.e., the examples) when it is given good input (i.e., a reasonable query). The results of two of our studies—the first study in which developers solved tasks with seeded queries and the fourth study in which an industrial developer used the tool—provide evidence to support the claim that the results of Strathcona queries can be understood and used by a developer. In the first study, the two developers were able to incorporate code from examples provided by Strathcona that they deemed helpful and they were able to discern when Strathcona did not provide useful examples. In the fourth study, the industrial developer was also able to integrate Strathcona into his development process, successfully using the returned examples to complete portions of his task.

Our second question asks whether a developer can use Strathcona to find useful examples with less effort than alternatives. The second and fourth studies we conducted each provide evidence to suggest that, for a number of cases, Strathcona is able to perform to this expectation. The second study focuses specifically on this question, comparing the results produced by Strathcona to those of grep and Eclipse's structured search. The fourth study, in which an industrial developer used the tool, also shows that when contextual information about an API is needed, Strathcona can be of aid.

Our third question considered whether developers can reasonably form queries to Strathcona. Three of the studies—the first, third, and fourth—address this question. The third study addresses the question directly and analytically, considering the robustness of query formulation to providing reasonable examples. This study showed that Strathcona is robust and can provide useful responses to a range of similar queries. The first and fourth studies address the question indirectly and qualitatively. In the first study, each of the two developers for the third task (which did not return any relevant examples by design from the seeded query) formulated their own queries and used examples from Strathcona returned by those queries. In the fourth study, the industrial developer used Strathcona on five independent queries he formulated naturally as part of his task, using the returned examples from three of these to complete the task.

Validity of empirical studies. Each of the four evaluation activities we performed has limitations.

In the user-based case study (Section 5.2), we chose to have a small number of subjects attempt the use of our approach across a range of tasks rather than have a larger number of subjects attempt one or two tasks. We made this choice because we believe it was more important to first examine the generalizability of the approach across tasks than across developers. In this study, we also chose to provide the subjects appropriate source fragments to query the Strathcona tool. This choice was made in the interest of the time needed to be spent by a subject on the study. To investigate this limitation, we undertook the third study (Section 5.4) to investigate how difficult it might be for a developer to formulate an appropriate query.

Our second study, which compared the effectiveness of Strathcona against grep and Eclipse search (Section 5.3), may be criticized for bias toward Strathcona because we performed the comparisons ourselves. To help address this limitation, we had a grep expert attempt the first task. As we have described, this expert was unable to make sufficient progress within more time (35 min) than most developers might typically allocate to finding an appropriate example. The comparison we performed used a complicated sequence of grep queries in an attempt to ensure that we made as fair a comparison to what grep could possibly return.

The third study tested the robustness of the Strathcona tool against a range of queries for the same example. Our intent in conducting this study was to simulate a range of queries that might be formed by a developer. There are two major limitations of this study. First, we do not know if the queries that are successful in returning the sought after example are representative of the queries that a developer might form. The high percentages of queries returning the desired result do suggest that, even based on random

15. http://krugle.com (last accessed: 9 June 2006).
16. http://koders.com (last accessed: 9 June 2006).

formation of queries, Strathcona can return useful results. The second major limitation is that we chose methods of a particular length, between 10 and 20 lines, to use in this experiment to keep the analysis tractable. The methods we chose were comprised of between 8 and 12 structural facts, while, in our repository, the average method encodes 12.5 structural facts.

All of the evaluations discussed in this section targeted the Eclipse APIs. We chose Eclipse for our evaluations because it is a large framework with many APIs and the Eclipse implementation makes heavy use of its own APIs, providing many sources of examples. We discuss the use of our approach on two other frameworks in Section 6.4.

6 Discussion

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

6.1 Examples: Good or Bad?

It may be that the provision of examples to a developer leads to worse code than when examples are not provided. Rosson and Carroll showed, in a study of developers using a Smalltalk framework [19], 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 [19]. Anecdotally, we observed in our first study (Section 5.2) that 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 in the first study, Subject 1, who successfully completed the task, copied some code that checked for array types and added the appropriate notations to the method signatures without knowing what the code did, resulting in a case being taken into account that the developer had not considered. By leveraging the work done by other developers in the past, this developer was able to complete the task with higher quality than if the developer had been working alone.

6.2 Heuristic Performance

To date, our focus has been on the utility of our overall approach: whether structural similarity can be used to return useful examples. In a pilot to the first study, we attempted a more quantitative evaluation of the performance of our heuristics. In the pilot, we asked a developer to rate the examples returned by Strathcona for the four tasks. We found that the developer was unable to provide such a rating because the developer could not assess the value of an example until trying to use it to complete the task [9]. However, completing the task with one example made it impossible to rate the next example given the information learned from completing the task. It may be possible to study the quantitative performance of the heuristics through a larger study in which developers are provided examples from only one kind of heuristic for the same set of tasks; the value of the examples might then be assessed across the set of developers. We have left this more subtle experimentation for future work.

6.3 Heuristic Refinement

We developed the heuristics embedded in Strathcona iteratively as described in Section 4. The current version of the heuristics in Strathcona do not include a number of the approaches we tried but that we did not find useful. We briefly describe the failed approaches in the first three paragraphs below. We then describe possible improvements in the four subsequent paragraphs.

Complex Heuristics. An earlier version of this work [10] described slightly different heuristics. The current version of the tool has one additional heuristic and three fewer heuristics than the earlier work. The REFERENCES heuristic has been added to help developers find examples that utilize public fields in framework classes. The USES WITH INHERITANCE and CALLS WITH INHERITANCE heuristics have been eliminated as each was equivalent to a combination of other heuristics: the USES and INHERITANCE heuristics, and the CALLS and INHERITANCE heuristics, respectively. The presence of such "nonorthogonal" heuristics weights certain structural facts more heavily than others. The removal of the nonorthogonal heuristics did not alter the recommended examples significantly in our sample tasks but simply improved performance. The CALLS BEST FIT heuristic was removed because it relied on the selection of an arbitrary ratio (the percentage of calls that matched those in the queried context) for its operation; an optimal ratio that was effective for multiple projects could not be determined.[17] Our experience in developing these heuristics suggests that the simplest heuristics are the ones that generalize best across different projects and tasks.

Example Scoring. We tried to develop a scoring system that would assign different values for the different kinds of structural similarity, but we were unable to find an approach that did not lose general applicability. Our scoring approaches tended to work for one style of code fragment but not others. We found that the styles of the fragments differed depending on the stage of development of the code and whether or not the developer had identified

17. The heuristics were changed after the evaluation in Section 5.2 was complete but before the evaluations presented in Sections 5.3 to 5.5 were performed. We have confirmed that, for the evaluation in Section 5.2, the useful examples that are returned continue to be identical to those returned previously. We did not record the details for the unhelpful examples.

reasonable hot spots in the framework from which to begin the task.

Object Instantiations. Our heuristics do not treat object instantiations specially; they are treated only as calls to a constructor. We did not find that heuristics that treated these instantiations specially were useful. One reason may be variability in Eclipse as to whether clients or servers instantiate objects. For example, whenever Factory classes are involved, the client does not instantiate objects but gets new objects delivered to them by framework objects.

Hierarchies. Strathcona's heuristics do not transitively check the type hierarchy when considering inheritance, uses, calls, or references relations. In our initial investigation, we found that these additional targets did not increase the effectiveness of our heuristics. While this seems correct for Eclipse, it does not seem to hold for other frameworks, such as JHotDraw. While Eclipse always refers to objects by their most specific interface (e.g., the code always references `IStatusLineManager`, not `StatusLineManager`), JHotDraw does not follow this convention. For instance, in JHotDraw, types are referred to differently depending on their context of use (e.g., a type is referred to as `CompositeFigure` when relevant and `Drawing` at other times). To best account for these polymorphic issues, our heuristics should be extended to reflect the type hierarchy. As Strathcona's repository currently stores all of the necessary relations to implement this functionality, this change would represent an addition to the extensible set of heuristics already supported by Strathcona.

Match Specificity. We treat each element of every structural query equally. Because of this, our heuristics will treat two examples equally even if one contains a call to a specific relevant API while the other may just use `String`. The results returned by the heuristics would likely improve if we ordered our queries such that the most-specific (that is, those with the least usages in the repository) were queried before the less specific queries. Expanding on this idea, we could use the least-specific query terms to only reinforce already-selected examples. This enhancement would likely improve the relevance of the returned examples.

Unresolvable Types. If a developer is uncertain of which type to use in a code fragment to be queried on, they must currently either insert all possibilities or iterate through the individual possibilities. In some cases, they may desire to insert placeholders rather than refer to real types, or they may not bother to include the appropriate import statements. Strathcona currently assumes that types and method signatures can be fully resolved, so any such use will result in one or more facts that cannot be matched. For example, the sample scenario (Section 2) involved querying on a call to `IStatusLineManager.setMessage(String)` where an import statement (not shown in Fig. 3a) allowed the Eclipse IDE to determine the fully qualified types: `org.eclipse.jface.action.IStatusLineManager.setMessage(java.lang.String)`. If the import statement had not been present, the use of `IStatusLineManager` could not have been resolved to its fully qualified name; as a result, no structural contexts would have matched this type use or this method call. Heuristics could be introduced that search for lexically similar references to deal with this issue.

Alternate Decompositions. Our current heuristics attempt to locate structural contexts that utilize as many of the queried facts as possible. However, developers are free to decompose their systems as they wish and this can lead to situations where our heuristics will fail to locate relevant examples. An example of this would be for the fourth task from Section 5.1. In this case, if the example in the repository used separate private methods for extracting different components of the method signature, Strathcona may not identify it as a relevant example, even though, ultimately, the code may be there to complete the task. To support this type of analysis, our repository would need to record the structural contexts for fragments larger than method scopes, or heuristics would need to be introduced that would combine structural contexts—effectively "widening" the method-level fragments pointed at by the individual structural contexts.

6.4 Experience with Other APIs

Each of the studies described in Section 5 used repositories based on the Eclipse APIs. We also have informal experience in using Strathcona with other APIs, specifically HTTPClient,[18] a package for creating client-side applications that utilize the Hyper-Text Transfer Protocol, and JhotDraw,[19] a Java GUI framework.

The documentation for HTTPClient makes clear how to set up a basic client implementation, but not how to handle a variety of error cases (e.g., protocol violations). A repository seeded with HTTPClient and six applications using it was queried on the basic client implementation, and useful examples were returned demonstrating how to deal with various error cases. No single example dealt with all the error cases, but two examples illustrated the cases of interest. A solution was constructed based on both examples.

For JHotDraw, a task of interest involved setting the Z-order of figures in a drawing. The code under construction had access to a `Figure` instance; examining the API for this type, the methods `setZValue(int)` and `getZValue()` were discovered that appeared relevant. To confirm this hypothesis, a query was made on the fragment `Figure. setZValue(0)`. A number of results came back in the context of the class `CompositeFigure`, where three methods were implemented that called `setZValue(int)`: `sendToLayer(Figure, int)`, `bringToFront(Figure)`, and `sendToBack(Figure)`. Examining the implementations of these methods, each involved detailed management of an explicit ordering of figures within the `CompositeFigure`; we revised our hypothesis, considering calls to these discovered methods to be preferable to direct calls to `setZValue(int)`. To test this hypothesis, we added a call to `CompositeFigure.sendToLayer (Figure, int)` to our code fragment and queried on it. No examples were returned. Manual examination revealed

18. http://jakarta.apache.org/commons/httpclient/ (last accessed: 9 June 2006).
19. http://www.jhotdraw.org/ (last accessed: 9 June 2006).

that `CompositeFigure` is the supertype for `StandardDrawing`, and that `StandardDrawing` implements the interface `Drawing`. Queries for `Drawing.sendToLayer(Figure, int)` returned several examples, supporting our revised hypothesis.

Our use of Strathcona on these two frameworks shows promise that our approach applies to more than just the Eclipse framework used in our evaluations. HTTPClient has a smaller but still useful set of applications that can be used to populate a repository and locate relevant examples. For JHotDraw, Strathcona was not able to find examples that bridge the gap between `CompositeFigure` and `Drawing` because its heuristics do not currently search the type hierarchy, as discussed in Section 6.3. Even so, Strathcona aided us in both these tasks, and the lack of examples returned by the query on `CompositeFigure.sendToLayer(Figure, int)` immediately informed us to look elsewhere for detailed information.

7 Conclusion

In this paper, we have presented an approach to help developers locate contextually relevant source code examples. These examples demonstrate how APIs of interest have been used by other projects and can be used to help a developer learn how to properly use those APIs. Our approach has two key differences from previous work: The repository automatically stores source code in a form whereby task-specific examples can be generated as needed, and the structural context that is used to form a query is extracted automatically from a developer-indicated source code fragment.

There are several ways in which the approach we have presented could be improved. We believe the most significant extensions would be in supporting richer queries to a repository through selection of discontiguous fragments of code, in better utilizing type hierarchy information in locating relevant protoexamples, and in better rankings and summary descriptions of returned examples to allow a developer to more easily determine relevance. In terms of evaluation, given the ability of the approach to return good examples for a wide range of queries, there remains a need to perform additional empirical studies with industrial developers.

We have demonstrated that structural context can be used as a basis for matching structurally relevant contexts. The heuristics we have developed can quickly locate and rank potential matches. From these matched structural contexts, we can derive examples to succinctly demonstrate to the developer those portions of the source code from the repository that are relevant to their task. The approach is effective, efficient, and more robust (in dealing with poor input) than traditional alternatives.

Acknowledgments

This research was funded in part by IBM and in part by the Canadian Natural Sciences and Engineering Research Council. The work benefited from comments by members of the Canadian Consortium for Software Engineering Research. The authors would like to thank the subjects who participated in their studies. They would also like to thank Miryung Kim, John Anvik, Andrew Eisenberg, and the anonymous referees for their comments.

References

[1] H.A. Basit and S. Jarzabek, "Detecting Higher-Level Similarity Patterns in Programs," *Proc. European Conf. Software Eng. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 156-165, 2005.

[2] I.D. Baxter, A. Yahin, L.M.D. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, 1998.

[3] G. Butler and P. Dénommée, "Documenting Frameworks," *Building Application Frameworks: Object-Oriented Foundations of Framework Design,* chapter 21, 1999.

[4] D. Čubranić, G.C. Murphy, J. Singer, and K.S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 446-465, 2005.

[5] G. Froehlich, H.J. Hoover, L. Liu, and P. Sorenson, "Hooking into Object-Oriented Application Frameworks," *Proc. Int'l Conf. Software Eng.,* pp. 491-501, 1997.

[6] R. Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *Proc. European Conf. Object-Oriented Programming and ACM Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 169-180, 1990.

[7] S. Henninger, "Retrieving Software Objects in an Example-Based Programming Environment," *Proc. ACM SIGIR Int'l Conf. Research and Development in Information Retrieval,* pp. 251-260, 1991.

[8] R. Hill and J. Rideout, "Automatic Method Completion," *Proc. IEEE Int'l Conf. Automated Software Eng.,* pp. 228-235, 2004.

[9] R. Holmes, "Using Structural Context to Recommend Source Code Examples," master's thesis, Univ. of British Columbia, 2004.

[10] R. Holmes and G.C. Murphy, "Using Structural Context to Recommend Source Code Examples," *Proc. Int'l Conf. Software Eng.,* pp. 117-125, 2004.

[11] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search," *Proc. Int'l Conf. Software Eng.,* pp. 14-24, 2003.

[12] R.E. Johnson, "Documenting Frameworks Using Patterns," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 63-72, 1992.

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 48-61, 2005.

[14] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," *Proc. Int'l Conf. Software Eng.,* pp. 167-176, 2000.

[15] A. Michail, "Code Web: Data Mining Library Reuse Patterns," *Proc. Int'l Conf. Software Eng.,* pp. 827-828, 2001.

[16] L.R. Neal, "A System for Example-Based Programming," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 63-68, 1989.

[17] S. Paul, "SCRUPLE: A Reengineer's Tool for Source Code Search," *Proc. IBM Centre for Advanced Studies Conf.,* pp. 329-346, 1992.

[18] E. Rissland, "Examples and Learning Systems," *Adaptive Control of Ill-Defined Systems,* 1983.

[19] M.B. Rosson and J.M. Carroll, "The Reuse of Uses in Smalltalk Programming," *ACM Trans. Computer-Human Interaction,* vol. 3, no. 3, pp. 219-253, 1996.

[20] Y. Ye, G. Fischer, and B. Reeves, "Integrating Active Information Delivery and Reuse Repository Systems," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 60-68, 2000.

[21] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," *Proc. Int'l Conf. Software Eng.,* pp. 513-523, 2002.

[22] A.M. Zaremski and J.M. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Trans. Software Eng. and Methodology,* vol. 4, no. 2, pp. 146-170, 1995.

Reid Holmes received the BSc and MSc degrees in computer science from the University of British Columbia in 2002 and 2004, respectively. He is currently a PhD student in the Department of Computer Science at the University of Calgary. His research interests include developer-oriented tool support, example recommendation, and large-scale pragmatic reuse.

Robert J. Walker received the BSc degree in geophysics from the University of British Columbia in 1992 and the BSc, MSc, and PhD degrees in computer science from the University of British Columbia in 1994, 1996, and 2003, respectively. He is currently an assistant professor in the Department of Computer Science at the University of Calgary. His research interests involve software modification, including design, reuse, and aspect-oriented software development. He is a member of the IEEE.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. She is currently a full professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.