The End-to-End Use of Source Code Examples: An Exploratory Study

Reid Holmes
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA, USA
rtholmes@cs.washington.edu

Rylan Cottrell, Robert J. Walker, Jörg Denzinger
Dept. of Computer Science
University of Calgary
Calgary, AB, Canada
cottrell, rwalker, denzinge@cpsc.ucalgary.ca

Abstract

Source code examples are valuable to developers needing to use an unfamiliar application programming interface (API). Numerous approaches exist to help developers locate source code examples; while some of these help the developer to select the most promising examples, none help the developer to reuse the example itself. Without explicit tool support for the complete end-to-end task, the developer can waste time and energy on examples that ultimately fail to be appropriate; as a result, the overhead required to reuse an example can restrict a developer's willingness to investigate multiple examples to find the "best" one for their situation. This paper outlines four case studies involving the end-to-end use of source code examples: we investigate the overhead and pitfalls involved in combining a few state-of-the-art techniques to support the end-to-end use of source code examples.

1. Introduction

Developers frequently use source code examples as the basis for interacting with an application programming interface (API) [13]. A source code example can serve as the explicit origin for a reuse task, when the example fits a developer's context sufficiently well. The process for interacting with source code examples for reuse involves three main phases [12, 5, 3, 6]: *location*, in which the developer queries for a set of examples to consider for relevance to the task; *selection*, in which the developer investigates those examples to identify those that match the requirements and are similar to their own system; and *integration*, in which the developer copies and modifies the chosen example to fit within their own application. These phases can be iterative: for example, there is no guarantee that a selected example will be appropriate until the results of integration are examined. At best, this end-to-end process currently requires the use of independent tools that cover different phases.

Many different approaches exist to help developers locate source code examples (see [10] for an overview). Each of these approaches, by varying means, permits the developer to investigate and to select the example which they deem most promising. However, these approaches present this information in terms of relevance to the developer's query; as such, the choice of examples and the order does not consider the developer's need to find an example that integrates well. Few approaches exist to support the small-scale integration of source code (see [1] for an overview), and these all make assumptions that make them inappropriate for the example integration task. Thus, despite the fact that each tool's design choices make sense *in isolation*, it is unlikely that any simple combination of existing approaches will suffice as end-to-end support for example reuse tasks.

We conduct four case studies into the application of a few of these approaches to the end-to-end process of example reuse; specifically, we look at Google Code Search[1] and the Strathcona example recommendation system [10] for the location and selection phases, and standard Eclipse IDE tools and the Jigsaw small-scale integration tool [1] for the integration phase. Drawing on scenarios described in the literature that are representative of real example reuse tasks, we investigate the costs and challenges associated with the end-to-end process.

2. Related Work

Various researchers have split reuse tasks into phases (e.g., [12, 5, 3]); while minor variations exist, the basic notions of location, selection, and integration are common to all. Frakes and Fox [6] further divide the three phases into seven conditions; they heavily emphasize that failure of *any* of their conditions can lead to failure for the reuse task—thus, rough transitions between the phases is liable to be an impediment to success. Regardless of the choice of categorization, these are the technical barriers that must be overcome for a reuse task to be successful. Despite widespread attention to the phases involved, no work has addressed

[1]http://google.com/codesearch

whether support for all of them was simultaneously needed and *none* has attempted to provide end-to-end support.

The reuse of source code examples is a form of white-box reuse, that has also been described as pragmatic reuse [8]. Various web-based code search engines—like Krugle, Koders, and Google Code Search (GCS)—focus mainly on locating examples without regards to the developer's context. Holmes et al. [10] overview more sophisticated example location systems until 2006 and how their Strathcona system improves upon them; we do not go into further details here. In contrast to example location, Prospector [11], XSnippet [14], and PARSEWeb [16] help developers learn how to use an API by determining possible paths (by computation and/or repository mining) through an API; these approaches are limited to relatively simple call chain problems, in contrast with example locators. Both example locators and call chain recommenders offer little support for the selection phase and none for integration.

Holmes and Walker [8] describe how medium- to large-scale pragmatic reuse tasks are valuable but were poorly supported prior to their Gilligan tool. At those scales, the overhead involved in the planning process used by Gilligan is amortized, leading to significant savings. At the smaller-scale involved in the end-to-end use of source code examples, this overhead is likely to overwhelm any savings.

The Jigsaw [1] system can integrate a method (example) into another context (target system) by considering structural and semantic similarity measures between the example and target context. Source elements from the original context that do not correspond with those in the target, are automatically copied and transformed to fit into the target; this approach can reduce the amount of manual intervention required to integrate a reused example. Cottrell et al. [1] describe how approaches previous to Jigsaw do not suffice for the pragmatic integration of source code examples; we do not repeat the details here. Unfortunately, Jigsaw provides no support for the location or selection phases.

3. Case Studies

We conducted case studies that considered end-to-end reuse tasks including locating, selecting, and integrating examples for specific tasks. Four scenarios were selected from the published literature on source code examples. The scenarios were chosen to possess some complexity and to display some variety in terms of their domain and attributes. Due to space restrictions, the detailed performance of the case studies and the quantitative results have been placed in a separate appendix [7].

3.1. Method

Four scenarios were used in our case studies, one each from the publications on Strathcona [10] (Scenario 1: Compute the signature of a `MethodDeclaration`), XSnippet [14] (Scenario 2: Create an `ICompilationUnit`), PARSEWeb [16] (Scenario 3: Access a text selection), and XFinder [2] (Scenario 4: Create a `TableModel`). For each scenario, we constructed a code skeleton based on the original article's description, to represent the point at which the developer required assistance from an example. We used both Google Code Search (GCS) and the Strathcona example recommendation system [10] to locate potential examples and to aid in selecting examples for integration. Integration was performed both "manually" via standard IDE tools, and via the Jigsaw small-scale reuse system [1].

Locating and selecting examples. We queried GCS using a string manually extracted from the code skeleton; we investigated the returned examples by scanning through the returned source files. Strathcona was queried by selecting the skeleton and invoking a Strathcona search; we investigated these results by considering the UML representation of the example first before investigating the source code directly. We only analyzed the first five returned examples from each location approach; this limit was chosen because some empirical evidence indicates that developers rarely look beyond this limit when searching [15].

Integrating potentially relevant examples. For each example deemed relevant to the task using either location approach, we attempted to integrate the example into our code skeleton. For the integration phase, we independently used the standard tools available in the Eclipse IDE ("manual" approach) and the Jigsaw tool [1].

Manual integration involved copying the relevant code from the example, pasting it into our code skeleton, and modifying it to resolve compilation errors and so it would work in our environment. Jigsaw integration involved selecting the originating source method that the developer wanted to reuse and selecting the target method in the code skeleton; this activated Jigsaw which subsequently copied the required code to our system and modified it to compile within our system.

To quantitatively compare the integration approaches, we record the time required to perform the task (and the time required to setup Jigsaw, where applicable), the lines of code (LOC) ultimately reused from the example, and the number of discrete actions required of the developer to perform the task. These results are reported in the appendix [7].

We use "actions" in this context as an effort indicator that is alternative to time. We counted any specific decision a developer made as an action; for example, if the developer decided to remove a certain method, deleting the method and all calls to it counted as a single action. Copying a method and resolving any dangling dependencies counted as a single action. Any unique modification to the source code counted as an action.

3.2. Observations and Analysis

While performing the case studies we investigated 40 examples and attempted to integrate the most promising 14 of these. From this experience we made a number of observations about the process of locating, selecting, and integrating these types of examples.

Classifying examples. The four scenarios we undertook were derived from the related literature, but in performing these tasks we observed that they fell into two categories. Scenario 2 involved what have been termed call chains, object instantiations [14], and method invocation sequences [16]; these involve short snippets of code that usually demonstrate how to access some functionality in an API. Conversely, feature-oriented scenarios require richer functionality; these are demonstrated in Scenarios 1, 3, and 4. In Scenario 2 we noticed that relevant call chain examples were easier to identify, occurred more frequently because their scope was broad (the same call chain can be used by examples of very different functionality), and were easy to integrate manually as the significant parts were generally short. Deciding that an example was relevant to a feature-oriented scenario was more difficult: more significant functionality is required from them and they are generally larger and more specific.

Location approach shortcomings. GCS sorts examples relative to the query terms that are sent to it, and performs lexical matching within entire files, selecting files where parts of identifiers and comments match the terms; Strathcona ranks examples according to the structural heuristics that are matched on the server. Neither location approach, or any we are aware of, can rank examples in the order that would be most conducive to the developer being able to reuse the example.

Manual integration shortcomings. We observed two main shortcomings of manual integration tasks. First, it was not easy to tell, through manual inspection, what dependencies the example code might have on the rest of its class or the system from which it was extracted. (This is consistent with an earlier study that found that developers often fail to identify source code dependencies using source code editors [9].) Second and more intrusive, the example's context was often different from our code skeleton: fields, local variables, and methods often had to be renamed. For small examples this was straightforward but for larger examples we had to be careful that we were not breaking how the code worked by modifying it.

Jigsaw integration shortcomings. Configuring an example so it could be integrated by Jigsaw often took more time than the actual integration; beyond our controlled environment, it is thus unlikely that Jigsaw would be used in end-to-end example reuse tasks.

3.3. Summary

We found that locating, selecting, and integrating source code examples with existing tool support is approachable but generally cumbersome and slower than necessary. This burden led to situations where we feel that after spending time to integrate one example we would be unlikely to spend more trying to integrate alternatives—in a realistic development setting, at least.

We found 5 significant issues with using these existing approaches as end-to-end support for reusing examples: (1) example repositories that do not contain examples relevant to a scenario are obviously not helpful to that scenario, but such a failing can easily occur if the repository maintenance is not automatic; (2) location techniques have to consider syntax and semantics to locate better examples for reuse; (3) located examples should be ordered relative to ease of integration with the developer's context; (4) overhead for moving examples into the developer's environment and resolving trivial issues has to be reduced or eliminated; and (5) the form of integration needs to consider factors such as: avoiding the alteration of APIs; and iterative issues from copying yet more code to eliminate dangling references.

4. Discussion

A naive response to the need for end-to-end support for example reuse tasks would be to take a couple of the existing approaches and combine them through a bit of engineering. Unfortunately, tools that consider only individual phases of these tasks are not merely incomplete with respect to the end-to-end tasks, but can actively conflict in significant ways, even though they are helpful for the phases of the task they were designed to support.

How can the issues be overcome? More work should be put into the selection phase to increase the chances of investigating the best integration candidate first. This could be achieved either through altering Strathcona to introduce one or more new heuristics [10, Section IV-C] and possibly to remove others, or by analyzing the examples returned by Strathcona and reordering them according to alternative heuristics. The latter option has the advantage of not altering Strathcona but the disadvantage that it cannot so easily add additional examples ignored by Strathcona (if any such exist).

The process of integration should be better automated and more oriented towards the end-to-end task so that developers can more readily try multiple candidate examples to find the one that best meets their needs. This experimentation is crucial to let developers see how an example works within their code, rather than expending more effort in the selection phase choosing the "right" example [4].

Are the case studies biased and lacking in statistical significance? Our investigation simply provides evidence that the end-to-end use of source code examples could be improved; it does not show that no combination of tools could not do better than what we tried. The goal of this exploratory investigation was to gain insight into the requirements of the end-to-end approach.

What is gained by providing end-to-end support? End-to-end support enables the developer to more quickly investigate potentially relevant examples, allowing them the possibility to use the time savings to investigate more examples and perhaps to discover better functionality or issues that were not otherwise apparent (like error cases). The fundamental difference between call chain scenarios and feature-provision scenarios is the richness involved: the more complex but non-standard the functionality, the more likely the need for in-depth investigation of multiple examples. Ignoring the details often leads to making mistakes in the details.

5. Conclusion

End-to-end example reuse tasks involve three phases: location, selection, and integration; with iteration as necessary. No single, existing approach covers all three phases, so we have explored combinations of approaches to provide end-to-end support. We have presented the results from case studies involving four scenarios gleaned from the literature, and have shown that a simple combination of existing approaches does not suffice.

Overall, we identified three key points. (1) Performing the end-to-end tasks via a hodgepodge of tools inhibits the developer's ability to succeed at non-trivial small-scale reuse tasks. (2) Example use tasks fall into two distinct kinds, *call chain problems* that cause few difficulties and *feature-oriented scenarios* that are far more demanding on the developer—differing tool support needs accrue from each kind. (3) Improving total task performance enables developers to investigate more examples in greater depth, thereby enabling them to experiment with different reuse candidates to identify the best one for their task (rather than guess which is best based on superficial appearances).

Without explicit tool support for the complete end-to-end task, the developer can waste a lot of time and energy on investigating and attempting to integrate examples that ultimately fail to be appropriate; as a result, the overhead required can restrict a developer's willingness to investigate multiple examples to find the "best" example for their situation. Reduced quality of the resulting system would ensue.

6. Acknowledgments

This work was supported in part by a Discovery Grant and a Postdoctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada.

References

[1] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 214–225, 2008.

[2] B. Dagenais and H. Ossher. Automatically locating framework extension examples. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 203–213, 2008.

[3] L. Dusink and J. van Katwijk. Reuse dimensions. In *Proc. ACM Symp. Softw. Reusabil.*, pp. 137–149, 1995.

[4] G. Fischer. Cognitive view of reuse and redesign. *IEEE Softw.*, 4(3):60–72, 1987.

[5] G. Fischer, S. Henninger, and D. Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proc. Int'l Conf. Softw. Eng.*, pp. 318–328, 1991.

[6] W. B. Frakes and C. J. Fox. Quality improvement using a software reuse failure modes model. *IEEE Trans. Softw. Eng.*, 22(4):274–279, 1996.

[7] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study—Appendices. Tech. rep. 2009-934-13, Department of Computer Science, University of Calgary, June 2009. http://dx.doi.org/1880/47297.

[8] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proc. Intl Conf. Softw. Eng.*, pp. 447–457, 2007.

[9] R. Holmes and R. J. Walker. Task-specific source code dependency investigation. In *Proc. IEEE Int'l Wkshp. Visualizing Softw. Underst. Analys.*, pp. 100–107, 2007.

[10] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pp. 48–61, 2005.

[12] D. Mcilroy. Mass-produced software components. In *Software Engineering: Report on a Conference by the NATO Science Committee*, pp. 138–155, 1968.

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

[14] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pp. 413–430, 2006.

[15] J. Starke, C. Luce, and J. Sillito. Working with search results. In *Proc. ICSE Wkshp. Search-Driven Dev.: Users Infrastr. Tools Eval.*, 2009. To appear.

[16] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. IEEE/ACM Int'l Conf. Autom. Softw. Eng.*, pp. 204–213, 2007.