Unanticipated Reuse of Large-Scale Software Features

Reid Holmes
Department of Computer Science
University of Calgary
2500 University Dr. NW
Calgary AB Canada T2N 1N4
rtholmes@cpsc.ucalgary.ca

ABSTRACT

Software reuse has been endorsed as a way to reduce development times and costs while increasing software quality and reliability. Techniques designed to encourage software reuse have concentrated on creating reusable software in the form of frameworks, reuse repositories, and component libraries. These approaches do not help a developer who wants to leverage, from an existing system, a complex feature that was not designed to be reusable. We propose an approach that allows developers to investigate the reuse potential of a feature within an existing system, to create a plan for reusing the feature, and to support the transformation of the feature to the developer's project. We believe that by providing explicit support for the reuse of large-scale source code features, the reuse process—and its benefits—can be made accessible to developers.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Program Editors; D.2.13 [**Reusable Software**]: Reuse Models

General Terms

Languages

Keywords

Reuse, software structure, software feature, integrated development environment.

1. INTRODUCTION

Software reuse remains difficult to achieve in unanticipated scenarios [7]. While a system is being developed, designers may identify subsets of features that are likely to change or be added in the future; however, their focus is generally inward-looking. That is, they are concerned about how the product they are creating may change, not how future developers may want to reuse their system. The cost of building software is high enough [1] without designers trying to account for external reuse scenarios. Even though a system may not be designed for external reuse, features within the system can be valuable to other developers who require similar pieces of functionality within their own projects.

The steps that a developer must go through to evaluate an existing software feature for reuse are complex. First, they must explore the source code for the feature they are interested in and determine its boundaries within its system. Second, they need to investigate each of the feature's dependencies within its system to identify the environment the feature requires. Third, they need to determine how each of these dependencies should be managed when the feature is reused in their project. Fourth, they need to adapt the feature's source code, and its dependencies, so it will work within their project. This largely manual process entails that the developer track each dependency, how it is to be handled, and how it should be integrated into their project. For large features the developer needs to account for an overwhelming amount of information in order to successfully complete the reuse task.

Previous research has provided several ways to describe features within systems [9, 13]. Other research has considered how to identify reusable source code [2, 5]. These approaches are not designed to help the developer reuse the features they have identified. Source-to-source transformation research has concentrated on how to transform source from one context to another [14], not how to locate the code to be transformed. While each of these research projects addresses an aspect of what the developer is trying to accomplish while reusing a feature, none provides a comprehensive approach that enables the reuse process.

In contrast, we propose a system by which a developer can navigate an abstract representation of a system to sketch the boundaries of a feature. Through an iterative process the developer then determines how the feature's dependencies should be managed. This augmented sketch then becomes a reuse plan, that we can use to extract the feature and transform it to the developer's current project. Our approach provides developers with intuitive techniques to manage, to evaluate, and to act upon the volumes of information required to successfully complete large-scale reuse tasks.

Our approach is detailed in Section 2. Related research is examined in Section 3. How we will develop and evaluate it is addressed in Section 4.

Copyright is held by the author/owner.
ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

2. THE APPROACH

How our approach helps developers reuse features is illustrated in Figure 1. First the developer needs to determine which feature they are building that they want to reuse from another project. Next, they need to locate an existing project that provides the feature they require. The developer then chooses a location in the project that is relevant to the feature. From that location our tool provides an initial sketch of the feature that the developer can start exploring from. The developer navigates the feature using an abstract visual graph representation, traversing the feature's dependencies and sketching out its boundaries (Section 2.1). The developer then determines how each of the feature's dependencies are relevant to the functionality they wish to reuse and annotates them accordingly (Section 2.1.2). As the developer traverses the feature's dependencies our system recommends artifacts that are both structurally significant and relevant based on the developer's past decisions (Section 2.1.1). The developer can then use our tool to extract and transform the feature's source code to their project based on this plan (Section 2.2).

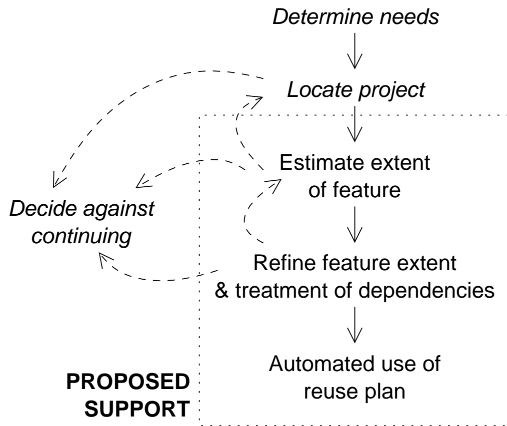

Figure 1: How the tool is used

Our approach does not force the developer to follow a rigid process to plan their reuse task. At any time, they are able to iterate on their previous decisions to converge on a solution that meets their needs.

2.1 Feature Sketching

The purpose of the sketching process is to formulate a comprehensive feature reuse plan. While sketching the feature the developer is trying to (a) delineate the boundaries of the feature, (b) determine how each of its dependencies should be handled, and (c) judge whether the reuse task is feasible.

Our approach enables developers to explore a graph-based representation of a feature. Each node in the graph represents a code fragment (a field, method, class, or package) and each edge represents structural dependencies between the nodes. The nodes and edges are statically-determined by analyzing the structural dependencies within the source code. The developer begins from an initial starting point that they are interested in for the feature (such as a class, method, method call, field reference or combination thereof) and interactively builds up the feature sketch. As the developer navigates through the dependencies in the graph our system ensures they are not overwhelmed with every possible dependency between one node and the next; based on the nodes the developer has investigated and annotated in the past, the graph is updated to only display dependencies the developer needs to make decisions about. For instance, if the developer discovers that the feature they are investigating uses the Java AWT, and their project uses the same widget kit, they can choose to accept all `java.awt.*` dependencies and have them elided from the graph in the future as they know these dependencies will be provided by their system.

By working within an abstraction of the feature, the developer is able to maintain a global overview of the feature, and what it requires, without having to maintain a mental model while jumping between various files. The low-cost nature of the exploratory process is critical to allow the developer to freely and quickly investigate alternate paths through the graph so they can iteratively converge on a sketch of the feature that meets their needs. The developer can make both aggregate and fine-grained decisions (including method call, field reference, method, field, class, and package level); this helps to reduce the scope of the dependencies they need to consider without losing the overall perspective of the feature.

2.1.1 Dependency Recommendation

To further scale our approach and help developers reuse large-scale features, the system will not only display dependencies for the developer but will also make recommendations about specific dependencies to them. The intent of these recommendations is to identify dependencies within the feature that have high reuse costs while hiding those dependencies that have low cost. Using these suggestions, developers can make more informed decisions, as they know more about the potential cost of each dependency; this will help them investigate more complex features, as they will not lose sight of the overall feature amid a myriad of details.

For example, for one feature of interest to us that we informally investigated we found that 90% of the 2700 dependencies involved in the reuse of the feature could be eliminated by removing a single dependency from the sketch. These high-cost dependencies are the ones that we want to bring to the developer's attention.

2.1.2 Feature Annotation

As the developer navigates their feature sketch, there are various decisions they can make about each of the nodes (Table 1) and dependencies (Table 2) that they encounter. The developer can quickly annotate each entity using single keystroke commands, creating a "visual bread crumb" trail revealing trends in the decisions they have already made. Significantly though, our approach leverages these annotations to drive the feature reuse process (Section 2.2).

As the developer annotates the graph they get a quick visual overview of the scope of the reuse task. The number of nodes marked as *reuse* indicates the size of the feature to be reused; the number of dependencies marked as *reimplement* indicates new code the developer needs to write in their system to support the feature; while *mapped* dependencies can be quickly modified to match the APIs in the developer's system. Nodes that are marked *reject* are not considered part of the feature. If the developer needs to see additional details before annotating a dependency they can view the source code that corresponds to the dependency in a view below the feature sketch.

After the developer has created their feature sketch and annotated it, they are left with a reuse plan. The plan describes the feature and how it should be reused in detail. They can then evaluate this plan to determine whether they should proceed with the reuse process, re-examine their annotations and update the plan, or abandon the reuse of the feature altogether.

Action	Description
Reuse	Migrate source code to new system
Reject	Not a part of the feature to be reused

Table 1: Node actions

Action	Description
Preserve	Already provided by the system
Reimplement	Must be implemented by the system
Map	Can be mapped to another API within the system that provides the same functionality

Table 2: Dependency resolution actions

2.2 Feature Extraction and Integration

Our tool can use the developer's reuse plan to automatically proceed with the reuse task. Nodes in the plan that are marked *reuse* are migrated to the new system. The dependencies on these nodes that are marked *preserve* should compile without modification within the new system. The tool helps the developer work through each of the *map* dependencies and assist them in creating maps from the feature's original dependency to new dependencies within the developer's project. Stub classes and methods are created for the developer to implement for those dependencies that are marked for *reimplementation*. After the developer implements the code for these stubs, the reuse task is complete.

2.3 Scenario

A small example demonstrating a quick reuse plan representing a status line feature is given in Figure 2. The oval nodes represent methods, the rectangular nodes are packages, and the edges between them show structural dependencies. This plan was created in less than five minutes and includes many of the salient attributes of the feature. Manually navigating through the source code would have involved many searches and switches between various source files. While using our tool, the developer was able to maintain their context as they quickly explored the feature. The developer has indicated that they will preserve dependencies (dashed lines) on `org.eclipse.swt.*` and `java.lang.*`, as these packages are already used in their project. The developer's project already contains a class providing functionality to manage the status line so `StatusLineManager` is remapped (dotted lines) to the existing class. The developer has also opted to reuse (solid lines) `StatusLine` with the exception of `StatusLine.trim()` which they will re-implement (double lines). They will also implement the colour functionality provided by `JFaceColors`. From this sketch they can quickly see that they will only have to reimplement two methods, three methods will be copied automatically, and two methods can be re-mapped to their own implementations, and all of the dependencies on two high-level packages do not need to be modified.

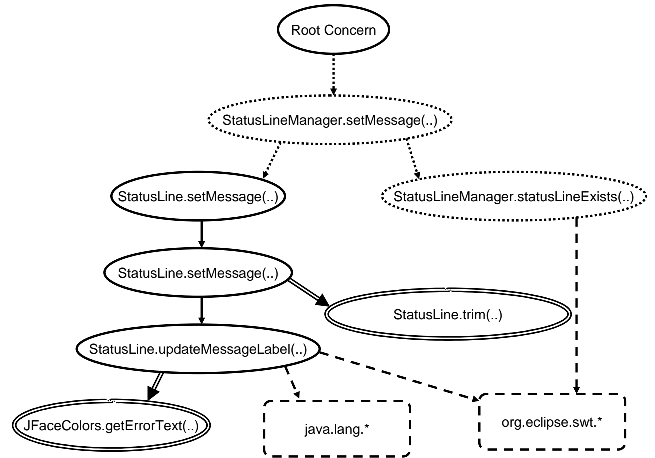

Figure 2: Simple feature reuse plan

3. RELATED WORK

Several approaches have been presented to help developers investigate, model, and reuse source code. Due to the nature of this paper, only a brief synopsis of relevant work will be presented.

Two notable reuse projects include CARE [2] and flow graph program slicing [10]. While both approaches locate features in source code (CARE uses a metrics approach as opposed to program slicing) neither approach assists the developer in reasoning about, or reusing the located feature. Additionally, while these approaches aim to provide reusable features to developers, it does not let them choose the scope of the feature to be reused.

The graph metaphor has been employed by past systems as a basis for program understanding activities [15, 4]. Navigating graphs to locate portions of a system relevant to a specific feature has also been evaluated [4]. These techniques have not been applied to feature reuse tasks. The reflexion model approach allows the developer to iteratively compare their mental model for the structure of a system against what is actually realized in the source code [11]. This approach is similar to ours in the manner in which the developer is encouraged to think, but reflexion models are not designed to help developers reuse features.

Robillard has extended his original research on concern mapping [13] to include the automatic recommendation of program elements [12]. This work demonstrates that reasonable recommendations can be provided to a developer in an iterative semi-automated environment. Several systems exist that recommend source code to be reused from within a repository of code. Both Strathcona [8] and CodeBroker [16] return example code that can be copied by a developer. These approaches both work with code that wasn't designed for reuse, but are only applicable for very small features, as the examples they return cannot be further explored.

The Implicit Context approach developed by Walker and Murphy [14] describes a method to transform a source code artifact from the context in which it was developed to a

new, different context but this work does not provide any means to help a developer identify and isolate features to be transformed.

We will build upon much of this prior research as we develop the tools to implement our approach.

4. METHOD AND EVALUATION

Our approach consists of four major components.

The first extracts the static structure of the software for to be considered for reuse. This builds upon our existing work with the Strathcona Example Recommendation System [8].

The second component handles feature sketching and annotation. This will be implemented from scratch using lessons learned through prior research [15, 4]. We will avoid tangential research into graph layout and edge routing problems.

The third is the dependency recommender which will be an application of Robillard's concern detection tool [12]. Once these first three components have been built, we will perform a comprehensive case study. The purpose of this case study will be to answer three key questions: (a) can developers use our tool to navigate a feature and annotate it easily; (b) do our dependency recommendations improve the scalability of the tool and reduce developer workload; and (c) can the resultant reuse plan be used to make valid decisions about the difficulty of the reuse task.

The fourth component provides the feature transformation functionality of our approach. We will build upon ICon-JDT [3] specifically for the mapping portions of the transformations.

Evaluation for our approach is complicated by the fact that skilled developers are required to perform complex feature reuse tasks. We will compare developer performance for carrying out a reuse task using three treatments: implementing the feature from scratch, manually reusing a feature, and using our approach to reuse the feature. We will monitor several aspects of the developer's performance on these tasks to try to identify the relative difficulty in employing each treatment. Additionally, through our partnership with IBM Ottawa Software Lab we will perform a case study to determine whether our approach can help industrial developers successfully perform reuse tasks on large-scale systems.

4.1 Status

We have begun developing the sketching framework and are performing preliminary analysis to see if our abstract representation sufficiently describes the feature to enable the developer to make informed decisions.

5. CONCLUSION

We have described an approach to help developers reuse complex source code features that were not designed for reuse. While existing techniques have looked at various aspects of this problem, we propose an approach that assists developers through the whole process from the initial stages of investigating a feature through creating a comprehensive reuse plan and finally migrating the feature to within their system. This approach relies on light-weight graphical abstractions and utilizes a system of heuristics to recommend important artifacts to the developer in order to increase scalability. Ultimately, using the reuse plan created by the developer, our tool transforms the feature from its existing system into the developer's project. By supporting developers in this way we hope to increase the adoption of unanticipated source-code reuse within development projects.

6. REFERENCES

[1] V. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora. The software engineering laboratory: an operational software experience factory. In *Proc. Int'l Conf. Softw. Eng.*, pages 370–381, 1992.

[2] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, 1991.

[3] J. J. C. Chang and R. J. Walker. Incomplete resolution of references in eclipse. In *Proc. Eclipse Technology eXchange Workshop Conf. at OOPSLA*, 2005.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proc. Int'l Workshop Progr. Comprehension*, page 241, 2000.

[5] W. Dai. Development of reusable expert system components: preliminary experience. In *Proc. Symp. Softw. Reusability*, pages 238–246, 1995.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[7] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, 1995.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. Int'l Conf. Softw. Eng.*, pages 117–125, 2005.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Progr. Lang. Syst.*, 12(1):26–60, 1990.

[10] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, 1997.

[11] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 18–28, 1995.

[12] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proc. Europ. Softw. Eng. Conf./ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 11–20.

[13] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. Softw. Eng.*, pages 406–416.

[14] R. J. Walker and G. C. Murphy. Implicit context: easing software evolution and reuse. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 69–78, 2000.

[15] J. Wu and M.-A. D. Storey. A multi-perspective software visualization environment. In *Proc. Conf. IBM Centres for Advanced Studies on Collaborative Research*, page 15, 2000.

[16] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proc. Int'l Conf. Softw. Eng.*, pages 513–523, 2002.