Using Refactorings to Automatically Update Component-Based Applications

Danny Dig
University of Illinois
Urbana, IL, US
dig@uiuc.edu

ABSTRACT
Frameworks and libraries change their APIs during evolution. Migrating an application to the new API is tedious and disrupts the development process. Although some tools and techniques have been proposed to solve the evolution of APIs, most updates are done manually. Our goal is to reduce the burden of reuse on maintenance by reducing the cost of adapting to change. We studied the API changes of three frameworks and one library and discovered that over 80% of the changes the break existing applications are refactorings. This suggests that refactoring-based migration tools should be used to effectively update applications. We propose a methodology to automatically and safely update component-based applications with no overhead on the component producers.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *documentation; restructuring, reverse engineering, and reengineering; version control.*

General Terms
Documentation, Economics, Reliability.

Keywords
library, framework, component reuse, software maintenance, backwards compatibility

1. INTRODUCTION
Part of maintaining a software system is updating it to use the latest version of its components. Developers like to reuse software components because it lets them build a system more quickly, but then the system depends on the reused components.

Ideally, the interface to a component never changes. In practice, new versions of software components often change their interfaces and so require systems that use the components to change accordingly. The application engineers might be in the middle of development when the introduction of an updated component could adversely affect costs and schedules. Unless there is a high return-on-investment, application developers will not want to migrate to the new version of the component. At the same time, component producers need to make changes in their designs. They are limited in terms of how much they can refine their designs if they want to maintain backwards compatibility. Clearly there is an impedance mismatch between how application and component developers regard change [11].

An important kind of change to object-oriented software is a refactoring[4]. Refactorings are program transformations that change the structure of a program but not its behavior. Refactorings include changing the names of classes and methods, moving methods and variables from one class to another and splitting methods or classes. A refactoring that changes the interface of an object must change all its clients to use the new interface. When a class library that is reused in many systems is refactored, the systems that reuse it must change. But often those developing the library do not know all the systems that reuse it. The new version of the library is a refactoring from their point of view, but not from the point of view of the application developers who are their customers.

Copyright is held by the author/owner(s).
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

2. GOAL STATEMENT
Our goal is to *reduce the burden of component reuse* on maintenance. This requires either reducing the amount of change or reducing the cost of adapting to change.

We propose a methodology to address the needs of both component and application developers:

1. Application developers want an easy (push-button) and safe (behavior-preserving) way to update component-based applications.
2. Component developers don't want to learn any new language or write any specifications extraneous to the regular component development.

Based on these objectives, the following research questions are investigated:

1. What are the component changes that break compatibility with existing clients? What is a suitable representation for these changes? Can it be gathered automatically? Does this representation carry both the syntax and the semantics of changes? Can it lead to safe, automatic updating of component-based applications?
2. How much of the effort spent on updating component-based applications can be saved?
3. How would component developers evolve the component's design when they don't have to worry about backwards compatibility?

3. APPROACH
Because refactorings carry rich semantics (besides the syntax of changes), they are a suitable representation for the component

evolution and can be used for automated upgrades. A tool will detect and record the changes that happened in the component as refactorings (no overhead on the component developers). Component developers will ship the log of refactorings along with the new release of the component. When an application engineer migrates to a new version of the component, our tool incorporates all the refactorings that were shipped with the component.

3.1 Feasibility Study

To better understand the requirements for migration tools we studied the API changes of three frameworks (Eclipse, Struts, Mortgage) and one library (log4J). We discovered that the changes that break existing applications are not random, but they tend to fall into particular categories. In the four case studies, respectively 84%, 81%, 90% and 97% of the API breaking changes are structural, behavior-preserving transformations (refactorings). Our findings [1] suggest that refactoring-based migration tools should be used to effectively update applications. The application developers will have to carry only a small fraction (less than 20%) of the remaining changes. These are changes that require human expertise.

3.2 Tool Support

We are currently working on CatchUp [3], a migration tool based on a record-and-playback technique. CatchUp is integrated with the Eclipse development environment. As component developers use Eclipse's refactoring engine to refactor their code, CatchUp records these refactorings automatically. This saves the component providers from writing any cumbersome annotations (like the ones proposed by [5, 6, 7]) whose sole purpose is to serve later during client migration. When an application developer upgrades to a new version, CatchUp plays back on the client code all the refactorings that were shipped with the component. This saves the application developer from checking that the incoming changes are safe and from adapting the code to these changes.

To address the cases when CatchUp was not available for performing refactorings or some refactorings were performed manually, we are currently working on automatic detection of refactorings. Current techniques [8, 9, 10] to detect refactorings either are too sensitive to the noise introduced by the deprecate-replace-remove cycle of component development or don't scale up for real-life components. To address the first challenge when deprecated entities coexist with their replacement counterparts for a while, we use a clone detection technique. We are developing RefactoringCrawler [2], a static analyzer based on a hybrid of syntactic and semantic analysis. We employ a fast syntactic analysis on the bulk of the code to create a list of potential candidates (clones). We only have to employ expensive semantic analysis on these candidates. Preliminary evaluation shows that our method scales for real-life libraries and can accurately detect when entities are renamed or moved.

3.3 Evaluation of Results

We will evaluate how our migration tool eases the task of the application developer when upgrading to a new version of the component. We plan to compare the productivity of a control group against that of a group that uses our toolkit. Also we will observe what kind of changes component designers make when they don't have to worry about breaking the compatibility with existing applications. Since our toolkit is integrated entirely with the Eclipse technology, we expect to have a large customer base. Feedback from the Eclipse community will improve the pragmatic aspects of our methodology.

4. CONCLUSION

By introducing a refactoring-based approach to migrate the applications, we remove 80% of the burden of manual upgrades while ensuring behavior preservation. The availability of powerful migration tools will change things for the component designers as well. Without fear of breaking client code, the designers will be bolder to refactor their designs. Given this new found freedom, designers won't have to carry poor design decisions made in the past. They will purge the design to be easier to understand and reuse.

5. REFERENCES

[1] D. Dig and R. Johnson: *The Role of Refactorings in API Evolution*, to appear in Proceedings of International Conference on Software Maintenance (ICSM'05)

[2] D. Dig, C. Comertoglu, D. Marinov, R. Johnson: *Automatic Detection of Refactorings for Frameworks and Libraries*, in Proceedings of Workshop on OO Reengineering (WOOR'05)

[3] J. Henkel and A. Diwan: "*CatchUp! Capturing and Replaying Refactorings to Support API Evolution"*, in Proceedings of International Conference on Software Engineering (ICSE '05)

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring:Improving the Design of Existing Code*, Addison-Wesley, 1999

[5] K. Chow and D. Notkin: *Semi-Automatic Update of Applications in Response to Library Changes*, in Proceedings of ICSM '96, pp 359-368

[6] R. Keller and U. Hlzle: *Binary Component Adaptation*, in Proceedings of ECOOP '98

[7] S. Roock and A. Havenstein: *Refactoring Tags for automatic refactoring of framework*, in Proceedings of Extreme Programming Conference '02

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz: *Finding refactorings via change metrics,* in Proceedings of OOPSLA, 2000.

[9] M. Godfrey and L. Zou: *Using origin analysis to detect merging and splitting of source code entities,* IEEE Transactions on Software Engineering, 2005.

[10] G. Antoniol, M. Di Penta, and E. Merlo: *An automatic approach to identify class evolution discontinuities,* in Proceedings of the 7th International Workshop on Principles of Software Evolution, 2004.

[11] M. Laitinen: *Framework Maintenance: Vendor Viewpoint*, in Object-Oriented Application Frameworks: Problems and Perspectives, M. E. Fayad, D. C. Schmidt, R.E. Johnson (eds), Wiley & Sons, 1999