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 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.

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
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