JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2006; **18**:83–107
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.328

Research

How do APIs evolve? A story of refactoring

Danny Dig*,† and Ralph Johnson

Department of Computer Science,
University of Illinois at Urbana-Champaign,
201 N Goodwin Avenue, Urbana, IL 61801, U.S.A.

SUMMARY

Frameworks and libraries change their APIs. Migrating an application to the new API is tedious and disrupts the development process. Although some tools and ideas have been proposed to solve the evolution of APIs, most updates are done manually. To better understand the requirements for migration tools, we studied the API changes of four frameworks and one library. We discovered that the changes that break existing applications are not random, but tend to fall into particular categories. Over 80% of these changes are refactorings. This suggests that refactoring-based migration tools should be used to update applications. Copyright © 2006 John Wiley & Sons, Ltd.

Received 16 September 2005; Revised 30 November 2005; Accepted 15 December 2005

KEY WORDS: API evolution; refactoring; frameworks; libraries; component reuse; 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 components that they have reused. 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 be changed before the new versions can be used.

Software evolution has long been a topic of study [1]. Others [2,3] have focused on *why* software changes; we want to discover *how* it changes. Our goal is to *reduce the burden of reuse* on maintenance. This requires either reducing the amount of change or reducing the cost of adapting to change.

Component developers do not want to learn a new language or write extra specifications for a component. Application developers want an easy (push-button) and safe (behavior-preserving)

*Correspondence to: Danny Dig, Department of Computer Science, University of Illinois, 201 N Goodwin Avenue, Urbana, IL 61801, U.S.A.
†E-mail: dig@cs.uiuc.edu

Copyright © 2006 John Wiley & Sons, Ltd.

way to update component-based applications. This paper is the beginning of our quest to meet the needs of both component and application developers. What is a suitable representation for the changes that happened in a component? 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? How much of the effort spent on updating component-based applications can be saved?

Although there are principles of software evolution that are true for software in any language, programming languages have an impact on software evolution. We are particularly interested in the evolution of object-oriented components (we refer to both the library and framework as components, unless a distinction is necessary). Classes contain a mixture of private and public methods. The public methods are those that are meant to be used by application programmers. The set of public methods of a class library make up its application programmer interface (API). Changes to private methods and classes do not pose a problem to application developers; they only care about changes to the API.

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 renaming classes or methods, moving methods or variables between classes, and splitting methods or classes. A refactoring that changes the interface of an object must change all of 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. However, those developing the library often do not know all of 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.

The original work on refactoring was motivated by framework evolution. Opdyke and Johnson [5] looked at the Choices operating system and the kind of refactorings that occurred as it evolved. Graver [6] studied an object-oriented compiler framework as it went through three iterations. Tokuda and Batory [7] describe the evolution of two frameworks, focusing on how large architectural changes can be accomplished by a sequence of refactorings.

However, none of these studies determined the fraction of changes that are refactorings. Of the changes that cause problems for maintainers, what fraction are refactorings? Are refactorings as important in practice as these authors imply? The authors all discuss tool support, although usually from the point of view of a component developer, not of a component reuser. However, CatchUp [8] is a tool that uses descriptions of refactorings to help application developers migrate their applications to a new version of a component. How much of the component evolution can be expressed in terms of refactorings? The only way to tell is to look at changes in a component over time and categorize them.

This paper looks at four frameworks and one library (see Table I and Section 2) developed by five different groups. Four are commonly used open source and one is a proprietary framework. All of the case studies are mature software, namely components that have been in production for more than three years. By now they have proven themselves to be useful and therefore acquired a large customer base. At this stage, API changes have the potential to break compatibility with many older applications. Backwards compatibility and different strategies to preserve it are the topic of Section 3.

We analyze and classify the API changes in the five systems. We describe a few techniques used to maintain backwards compatibility (Section 3) while the main focus is on the API changes that break compatibility with older applications (Section 4). We learned that between two versions for each of the five systems we studied, 84%, 81%, 90%, 97%, and 94% of the API breaking changes are refactorings. Most API changes occur as responsibility is shifted between classes (e.g., methods or fields moved around) and collaboration protocol changes (e.g., renaming or changing method signature).

Table I. Size of the studied components in thousand lines of code (KLOC). The number of classes in API denote only those classes that are meant to be reused. ReleaseNotes give the size (in pages) of the documents describing the API changes. The logs were provided by the component developers.

	Eclipse 3.0	Mortgage	Struts 1.2.4	log4j 1.3	JHotDraw 5.0
Size (KLOC)	1923	52	97	62	14
API Classes	2579	174	435	349	134
BreakingChanges	51	11	136	38	58
ReleaseNotes (pages)	24	—	16	4	3

These results made us believe that refactoring plays an important role as mature components evolve. Since more than 80% of the breaking changes in the case studies were refactorings, we propose that tools that record-and-playback refactorings are used to update applications (Section 5).

This paper is an expanded version of a conference paper [9], including one more case study, more types of refactorings, and a detailed analysis of each refactoring. The new case study gave similar results as the others.

2. OVERVIEW OF THE CASE STUDIES

This section briefly describes the components that we used as case studies. We chose four well-known frameworks and libraries from the open-source realm. To check whether the production environment affects the type of API changes, we chose one more proprietary framework. We tried to be unbiased in the selection of the case studies, the main concern being that the systems have good documentation.

For each component we chose for comparison two major releases that span large architectural changes. There are two benefits to choosing major releases as comparison points. First, it is likely that there will be lots of changes between the two versions. Second, it is likely that those changes will be documented thus providing some starting point for a detailed analysis of the API changes.

2.1. Eclipse platform

Eclipse (http://www.eclipse.org) was initially developed by IBM and later released to the open-source community. The Eclipse platform provides many APIs and many different smaller frameworks. The key framework in Eclipse is a plug-in-based framework that can be used to develop and integrate software tools. This framework is often used to develop Integrated Development Environments (IDEs). The Eclipse platform is written in Java.

We chose two major releases of Eclipse, namely 2.1 (March 2003) and 3.0 (June 2004). Eclipse 3.0 came with some major themes that affected the APIs. The *responsiveness* theme ensured that more operations run in the background without blocking the user. New APIs allow long-running operations such as builds and searches to be performed in the background while the user continues to work.

Another major theme in 3.0 is *rich-client platforms*. Eclipse was designed as a universal IDE. However, many components of Eclipse are not particularly specific to IDEs and can be reused in other rich-client applications (e.g., plug-ins, help system, update manager, window-based GUIs). This architectural theme involved factoring-out IDE-specific elements. APIs heavily affected by this change are those that made use of the filesystem resources. For instance, `IWorkbenchPage` is an interface used to open an editor for a file input. All methods that were resource specific (those that dealt with opening editors over files) were removed from the interface. A client who opens an editor for a file should convert it first to a generic editor input. Now the interface can be used by both non-IDE clients (e.g., an electronic mail client that edits the message body) as well as IDE clients.

2.2. Mortgage framework

A large banking corporation in the Midwest has been building a Mortgage framework to leverage existing financial expertise when writing new applications.

The Mortgage framework allows various banking applications developed within the company to communicate with each other and with the existing legacy systems. The framework receives requests from front-end systems or services, evaluates their requirements, and redirects the request to a specific destination, or destinations such as a pricing engine or closing-cost engine. After receiving an appropriate response, the framework refines it for a specific request channel and then forwards it back to the requestor.

When we visited the banking institution, they were finalizing the integration between the mortgage framework and another middleware framework developed independently at another branch of the bank. Frameworks are designed for extension not for integration [10]. As a result of the marriage between the two frameworks, the application developers had to migrate the existing services. The company reported that the whole integration and upgrading process lasted a summer. At the time of writing, there are about 50 services that use the framework.

2.3. Struts framework

Struts (http://www.struts.apache.org) is an open-source framework for building Java Web applications. The framework is a variation of the Model-View-Controller (MVC) design paradigm. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View.

For the Model, Struts can interact with standard data access technologies, such as JDBC and EJB, as well as most any third-party packages, such as Hibernate, iBATIS, or Object Relational Bridge. For the View, Struts works well with JavaServer Pages, as well as Velocity Templates, XSLT, and other presentation systems. Owing to this separation of concerns, Struts can help control change in a Web project and promote job specialization.

We chose for comparison version 1.1 (June 2003), a major past release, and 1.2.4 (September 2004), the latest stable release. All of the API changes reveal consolidation work that was done between the two releases. The framework developers eliminated duplicated code and removed unmaintained or buggy code.

2.4. log4j Library

log4j (http://www.logging.apache.org/log4j) is a popular Java library for enabling logging without modifying the application binary. It allows the developer to control which log statements are output with arbitrary granularity by using external configuration files. Logging does have its drawbacks. It can slow down an application. If too verbose, it can cause scrolling blindness. To alleviate these concerns, log4j is designed to be reliable, fast, and extensible.

log4j uses a logger hierarchy to control which log statements are output. This helps reduce the volume of logged output and minimize the cost of logging. The target of the log output can be a file, an OutputStream, a java.io.Writer, a remote log4j server, or a remote Unix Syslog daemon logger among many other output targets.

We chose for comparison version 1.2 (May 2002) and version 1.3alpha6 (January 2005). The library passed through an expansionary phase and it grew from 30 KLOC to 62 KLOC (KLOC stands for thousand lines of code). The library grew by improving on existing components (such as Chainsaw, a visualization toolkit for loggers) or adding new components (such as support for plugins as a way to extend the library).

2.5. JHotDraw

JHotDraw (http://www.jhotdraw.org/) is a two-dimensional graphics framework for structured drawing editors that is written in Java. It was originally developed in Smalltalk by Kent Beck and Ward Cunningham. Erich Gamma and Thomas Eggenschwiler developed the Java version, then it became an open-source project. The original HotDraw was one of the first projects specifically designed for reuse and labeled as a framework. It was also one of the first systems documented in terms of design patterns [11].

In contrast to the Swing graphics library, JHotDraw defines a basic skeleton for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework has been used to create many different editors from CASE tools to a Pert diagram editor (http://www.jhotdraw.org/survey/applications.html).

We chose for comparison version 4.0 and 5.0. The purpose of 5.0 release was to use features (new at that time) available in JDK 1.1 such as event model, access to resources, serialization, scrolling, printing, etc. It also consolidated the packaging structure and improved support for connectivity of diagrams.

2.6. Collecting the data

The case study components are medium to large sized (Mortgage is 50 KLOC, Eclipse is roughly 20 000 KLOC). It is hard to discover the changes in a large system and many authors suggest that tools should be used to detect changes. For instance, Demeyer *et al.* [12] describe how they used metrics tools to discover refactorings. However, most API changes follow a long deprecate–replace–remove cycle to preserve backwards compatibility. This means that an obsolete API can coexist with the new API for a long time. This introduces enough noise so that tools might mislead us about the exact kind of change that happened.

Copyright © 2006 John Wiley & Sons, Ltd.

Consider for instance a change such as renaming class `Category` to class `Logger` in log4j. In order to maintain compatibility with old clients, class `Logger` (the new name) inherits from class `Category`. The constructor of `Category` became protected so that users cannot create categories directly but invoke instead the creational method `getInstance()`. This method returns instances of the new class `Logger`. Any method in `Category` that returned an object of type `Category` became deprecated. Clients should replace all of the references to `Category` with references to `Logger`. The two classes still coexist, but `Category` will be deleted eventually. Such a three-step change would have been misinterpreted by a tool, but a human expert can easily spot this as a renaming.

For these reasons, we chose instead to do a manual analysis of the API changes. Even for the larger components, this was feasible because we started from the change logs that describe the API changes for each release. For Eclipse we used its help system[‡], the documents called 'Incompatibilities between Eclipse 2.1 and 3.0' and 'Adopting 3.0 mechanisms and API'. For Struts we studied the 'Release Notes' for version 1.2.4[§]. For Log4J we studied 'Preparing for log4j version 1.3'[¶]. For JHotDraw we studied the 'Release Notes' for version 5.0, packaged along with the documentation of the framework.

Sometimes the release documents would be vague, reading for example 'method M in class X is deprecated'. Owing to the deprecate–replace–remove cycle many types of changes are masked by the deprecation mechanism. In those cases we read and compared the two versions of the source code in order to discover the intent behind the deprecation. When a method is deprecated it merely delegates to its replacement method. By reading the code we learned whether the new method is just a renaming of the deprecated method, whether the intent was to move the method to another class, or whether the deprecated method was replaced by a semantically equivalent method that offers better performance.

For the Mortgage framework, for two days we interviewed the framework and application developers and then studied the source code. We classified all of the breaking API changes from the case studies into structural and behavioral changes (qualitative analysis), then we recorded how many times each type of change occurred (quantitative analysis).

The current tool support [12–15] for detecting and classifying structural evolution is very limited: only a few types of refactorings (mostly merging and splitting) were attempted to be detected. Therefore, to do a comprehensive qualitative analysis of the breaking changes, the manual method seems the only alternative. We double-checked our quantitative analysis by using a tool (Van [16]) and heuristics (as in [12]). For each type of refactoring, we wrote queries in Van that return those structures suspected of that specific refactoring. For instance, to detect changes in method parameters' types, we searched for methods that have the same name in both versions of a class, have the same number of arguments, have the same return type but have different signature. After analyzing and eliminating the false positives, the remaining candidates were found among those that were already detected from the change logs. Van found a few other places suspected of refactoring, but the number is less than 4% of those detected by starting from the change logs. Also Van failed to detect several places where a certain refactoring took place. This happened because of the noise introduced by the deprecate–replace–remove cycle described above. We could only cross reference our results for Struts and log4j.

[‡]Section: Eclipse 3.0 Plugin Migration Guide.
[§]http://struts.apache.org/userGuide/release-notes-1.2.4.html
[¶]http://www.qos.ch/logging/preparingFor13.jsp

The tool did not scale up for Eclipse and we do not own the source code of the proprietary Mortgage framework. JHotDraw case study was added at a later date and since Van did not produce new data for the previous case studies, we decided not to run it on JHotDraw.

3. HOW APIS CHANGE

In this section we classify the API changes in respect of how they affect backwards compatibility. Section 3.1 talks about APIs and what does it mean for an API change to break compatibility with applications. In Section 3.2, we describe some techniques used to make API changes without breaking applications. Section 3.3 presents the empirical data gathered from the breaking changes we noticed in the five case studies.

3.1. API changes and compatibility

An API is the interface that a component provides to application developers and its description is part of the component's documentation. The term has been extended to mean any component that is supposed to be reused by clients and thus is expected to be stable.

APIs make use of the visibility rules of the language in which the component was implemented. For instance, in Java or C++ only members that are declared public or protected can be part of the API. However, not all classes or class members that are public are intended to appear in client code.

Usually there are no language features that distinguish between public entities that are intended to be part of the API and public entities that are not. Naming conventions can be used to identify those components that are 'published' (to be reused) from those components that are 'public' but are not intended to be reused [17]. For instance, Eclipse places a public class that is not an API in a package with 'internal' as a prefix. Such a class is fair game to change without notice.

Over time, changes are made to APIs or APIs' behavior. Depending on whether or not they are backwards compatible, API changes can be classified as *non-breaking API changes* or *breaking API changes*.

A *breaking change* is not backwards compatible. It would cause an application built with an older version of the component to fail under a newer version. If the problem is immediately visible, the application fails to compile or link. Or the application might compile fine, but behave differently at runtime. By behavior we mean functional behavior, e.g., the set of observable outputs for a given set of inputs. If the only observable difference is that an application is slightly faster or slower or has a different memory footprint, we do not consider it a breaking change[‖].

A *non-breaking change* is backwards compatible. Such a change can be an enhancement such as the addition of new modules to extend the functionality of the component, or it can be a performance optimization or an error removal.

A seemingly non-breaking change such as fixing a bug in the component might be a breaking change. If the application developers worked around the bug, then when the bug is removed from the component, the application might behave differently.

[‖]We go with a loose definition of failure, but in domains such as embedded systems, our notion of reliability might not be sufficient.

Although there are a number of techniques used to facilitate component changes without breaking the clients [18], breaking API changes happen all the time. Our ultimate goal is to provide migration tools that can incorporate breaking changes.

3.2. Non-breaking API changes

This section discusses changes to the API that preserve the backwards compatibility of the component. We also present some techniques that component developers use to maintain compatibility with older applications.

New features of the component are usually packaged in separate modules and do not affect the existing applications. Developers still have to learn the new APIs in order to use the extensions.

Other times, component designers successfully employ established software-engineering techniques such as information hiding, encapsulation, and abstraction to introduce changes seamlessly. Such ‘under the hood’ changes often include performance optimizations, security improvements, bug fixes, and other implementation details. For instance, the financial institution that we visited made a large architectural change to their enterprise framework. For performance reasons they replaced EJB entity beans with Data Access Objects. They employed information hiding to isolate the changes from the Web services that use the framework. All of the services are calling a Session Bean Facade that offers the same interface to the outside world. For the application developers these changes happened seamlessly without asking for recoding.

In the five case studies we noticed a number of other techniques used to facilitate changes without breaking the client applications.

(1) *Deprecation*. This is used to gracefully degrade old APIs and to warn application developers that certain APIs will not be supported in future versions. Developers can plan ahead for migration and avoid a last minute rush. The parts of the API that are obsolete are marked with special annotations (e.g., `@deprecated` tag in Javadoc). The deprecated API should also provide a description of how applications would migrate to the new API.

(2) *Delegation.* Sometimes it is possible to make the old API implementation delegate to the new implementation so that clients make use of the enhancements. Nevertheless, they should plan to migrate to the new API. For instance, this technique is often used when renaming a method in Struts. In the class `ActionServlet`, method `destroyApplications()` changed its name to `destroyModules`. In order to preserve backwards compatibility, the class supported both methods for a while:

```
/**
 * @deprecated
 * replaced by destroyModules()
 */

protected void destroyApplications() {
        destroyModules();
}
```

(3) *Naming conventions.* Eclipse developers use this style when they extend an interface. For instance, the initial version of interface IMarkerResolution was

```
public interface IMarkerResolution {
    public String getLabel();
    public void run(IMarker marker);
}
```

The new interface adds more methods without breaking existing implementations of `IMarkerResolution`:

```
public interface IMarkerResolution2 extends IMarkerResolution {
    public String getDescription();
    public Image getImage();
}
```

The ‘2’ naming convention suggests that this is an extension to the original interface. The new interface can be used wherever the old one was expected. However, clients will have to use casting to access the new methods.

(4) *Runtime switch.* Dealing with removed classes or removed methods from an interface is a bit trickier. Eclipse provides a Compatibility plugin that ‘adds back’ the removed entities at runtime. For instance, Eclipse 2.1 API offered a method that is removed in version 3.0:

```
public interface IWorkbenchPage {
    IEditorPart openEditor(IFile file); // to be deleted
    ...
}
```

Eclipse 3.0 checks whether a client application is compliant with version 2.1 or 3.0. When an older client makes a call to the removed method, this call is dispatched to an instantiator of the older version of the interface (which is added back and loaded up at runtime).

(5) *COM-style interface query.* A component provides multiple versions of an interface. A client will first query for an interface that it knows how to interact with. If the component still supports the older interface that the client is requesting, the client and the component can communicate. In Eclipse, the initial document interface `IDocument` has acquired several extensions called `IDocumentExtension`, `IDocumentExtension2`, `IDocumentExtension3`. A client checks whether the document returned by `getDocument()` has an interface that it knows how to talk to.

3.3. Breaking API changes

Breaking changes are extremely disturbing in the development life cycle of component-based applications. When application engineers are in the middle of development, upgrading a component could hurt 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 [19].

Table II lists the types of breaking API changes that we observed in the components that we studied. The first column identifies the type of change. Those changes in *italic* font are refactorings. The remaining columns give the number of times each type of change occurred in the components. Columns Eclipse* (E*) and Struts* (S*) deal with ‘recommended’ changes. Component designers marked these as changes that will be enforced in the next major release. Even though technically these

Table II. Types of breaking API changes and the number of these changes in Eclipse (E), Mortgage (M), Struts (S), log4j (L), and JHotDraw (JHD). Eclipse* (E*) and Struts* (S*) denote recommended changes, that is changes that will become breaking changes in future releases. Those changes in *italic* font (upper half of the table) are refactorings.

Type of change	E	E*	M	S	S*	L	JHD
Moved method	16	13	—	11	28	9	—
Moved field	—	45	—	18	2	5	—
Deleted method	2	2	—	24	32	—	2
Changed argument type	5	—	4	18	4	11	—
Changed return type	2	—	1	2	—	2	31
Replaced method call	1	20	—	8	4	—	1
Renamed method	4	—	—	16	5	8	—
New hook method	4	2	2	7	—	—	5
Extra argument	3	2	2	1	1	—	2
Deleted class	—	—	—	9	—	—	1
Extracted interface	—	—	—	—	—	—	7
Renamed field	—	—	—	6	1	—	—
Renamed class	—	1	—	2	—	2	2
Method object	3	—	—	—	—	—	—
Pushed down method	3	—	—	—	—	—	—
Moved class	—	2	—	—	—	—	1
Pulled up method	—	—	—	1	—	—	—
Renamed package	—	—	—	—	—	—	1
Split package	—	—	—	—	—	—	1
Split class	—	—	—	—	—	—	1
New method contract	3	12	1	8	—	1	—
Implement new interface	1	—	1	5	—	—	3
Changed event order	3	—	—	—	—	—	—
New enum constant	1	—	—	—	—	—	—

are not breaking changes for the current release (they were insulated by the deprecation mechanism), we included them to offer the trend of breaking changes that are coming in next versions. Based on how many times each type of change occurred, we sorted the rows so that most popular changes appear first.

4. A CATALOG OF BREAKING API CHANGES

This section categorizes the changes in Table II according to how they affect the semantics of the component. The structural transformations are semantic-preserving changes (refactorings) while the behavioral changes are semantic-modifying changes.

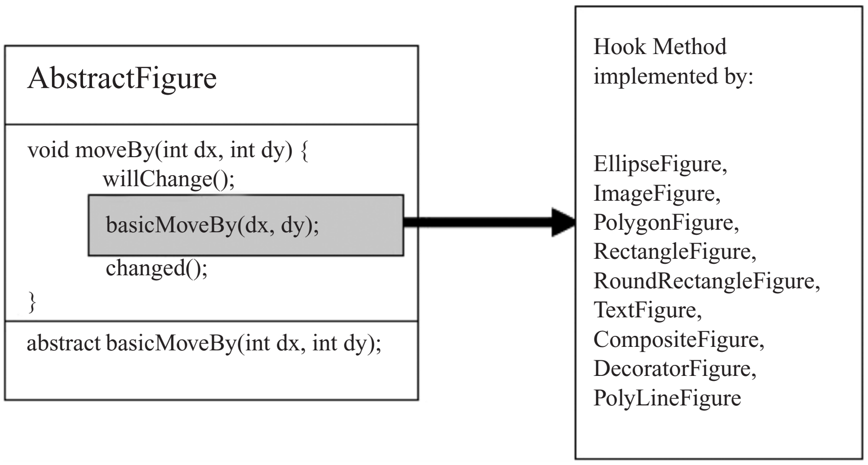


Figure 1. Using the hook method to introduce points of variability.

An API class can have two types of clients: *instantiators* and *extenders*. As for API methods, due to extensive usage of callbacks (hook methods) in frameworks, there are two types of clients of API methods: *callers* and *implementors*. Some changes affect both types of clients while other changes affect only one type of clients.

JHotDraw illustrates some of the subtleties of API changes upon client code. `AbstractFigure` provides default implementation for the figure classes. We encapsulate commonalities among different class figures, by using the Template Method design pattern [20]. Thus, method `moveBy` (referred as template method) contains the fixed algorithm for moving a figure (see Figure 1): first it announces that a figure is about to change something that will affect its displaying box (by calling method `willChange`), then the figure is moved with the specified delta (by calling method `basicMoveBy`), and lastly it informs that a figure modified the area of its display box (by calling method `changed`). The sequence of these three steps is common for all the figures, what changes from figure to figure is encapsulated in the abstract method `basicMoveBy`. Therefore, this method becomes the hinge point in the algorithm, or the *hook method*. Clients wishing to add new types of figures must implement this method. Rather than client code calling the framework, the framework will call this method when the user moves a figure (thus, the callback nature of this method).

Now any change that affects the signature or name of the hook method is potentially changing the behavior of client code. A change to the hook method that breaks the polymorphic overriding with a client method will result in the client method being skipped whenever the template method is invoked. Therefore, fewer client methods get called. The inverse can also happen. A change to the hook method that makes it become the super method of a different client method will result in the client method being called (accidentally) whenever the template method is called. Steyaert *et al.* [21] refer to the first case as *Inconsistent Method* and to the second as *Method Capture*. Since Method Capture and Inconsistent Method do not produce compile errors, one could fallaciously assume that the upgrading was safe.

When the hook method is declared as an abstract method, if the hook method changes so that there are no more client methods overriding it, the compiler signals an *Unimplemented Method* error in the client class.

Any of the Inconsistent Method, Method Capture or Unimplemented Method can occur, depending on the nature of change in the hook method and the existing methods in the client class. Suppose that a client class named `BackgroundFigure` has, among other methods, the following methods:

```
public class BackgroundFigure extends AbstractFigure{
    void basicMoveBy(int dx, int dy){
        // overrides the hook method in the superclass
        ...
    }

    void moveByDelta(int dx, int dy){
        // a helper method
    }
}
```

If the framework designers move the `basicMoveBy` hook method to another class and update its call site in the `moveBy` template method, the `basicMoveBy` in `BackgroundFigure` no longer overrides the hook method. When the template method is called on an instance of `BackgroundFigure`, its `basicMoveBy` does not get called any longer, thus leading to Inconsistent Method. If, instead, framework designers rename the `basicMoveBy` hook method to `moveByDelta`, since there exists a method with this signature in the client class, this method will be (accidentally) called whenever the template method is invoked, therefore leading to Method Capture. Otherwise, if the framework designers rename the abstract hook method such that there is no method in `BackgroundFigure` that overrides it, the compiler would signal an Unimplemented Method error in class `BackgroundFigure`.

4.1. Structural transformations

To improve reusability and maintainability of the component, the code is restructured (refactored). Refactorings affect only the structure of the code and are meant to preserve the functional behavior of the component. Consider, for instance, what happens when a method is renamed.

Component designers rename an instance method in the component. They find and update all the callers and implementors of the method to reflect the new name. For the component itself, this change is safe and does not modify its behavior. However, remote applications that call the renamed method are broken. Thus, a behavior-preserving change (refactoring) for the component might lead to a breaking change for the application.

Most times application code is not available to component developers when they make structural changes. The result is that applications might not even compile with the new version of the component. Once the application developer solves the compile errors, the application's behavior is the same (structural changes in the component do not introduce new behavior). In cases when the structural changes accidentally induced modified behavior (such as in Method Capture or Inconsistent Method described above), the application developer will have to make semantical code changes to counter balance the changes in the component.

Moved method

The most common way that instance methods moved in Eclipse is by becoming static methods in new host classes. The rationale was to move the layer breaking methods into utility classes to preserve the convenience of the old methods. Usually the moved method will take the old home class as an extra argument. This will ensure that the moved method can access public members in the old home class.

In Struts, instance methods remain instance methods after they move to other classes. Old callers of the method ask a factory method for an instance of the new home class and then call the moved method. Other ways that methods got moved are variations of the Move Method refactoring described by Fowler *et al*. [4].

Implementors of the moved method will not compile if they make a call to the super() method. As for the callers of the moved method, they might compile or not depending on whether the method is used polymorphically. If the moved method is declared in a parent class and is overridden in the client class and the method is invoked by sending a message to an object having the type of the client class, the behavior is preserved. However, good object-oriented principles recommend that instance or local variables are declared having the most general supertype. In this case, in a static typed language, a compile error will warn that there is no such a method in the parent class. Moving a hook method out of the parent class results in Inconsistent Method (described above) since the client method overriding the hook is no longer called.

Moved field

Encapsulation requires that the variables that characterize the state of an object are not exposed. However, sometimes fields are publicly exposed either because of convenience or because they represent constants. When fields are placeholders for global constants they are usually declared as static fields. In Eclipse, Struts, and log4j, only fields that were constants moved to another home class. Moving a public field results in compile errors for the clients who access it.

Deleted method

Typically this happens after a method is renamed or moved to another class. For compatibility reasons, component producers support both the old and new method for a while. After all of the references to the old method were replaced, the method is deleted since it is a remnant of the obsolete API. This usually results in compile errors for the clients calling the method. Deleting a hook method results in the Inconsistent Method since the client method overriding the hook is no longer called.

Changed argument type

We observed several kinds of argument type changes.

(1) The type of a method argument is replaced with its supertype to make the method more general. This change may or may not break an existing application depending on whether the application calls any methods that are not visible through the supertype's interface.
(2) The type of method argument is replaced by another type while the relationship between the two is aggregation. This is often the case when replacing a primitive type with an object type

(e.g., in Java replace *int* with *Integer*). Another special case is replacing a type with a collection that contains several elements of the previous type. In order to regard these changes as automated refactorings, one needs to know how to access the member from the wrapper and how to get the proper wrapper for a member. In the Mortgage framework the method `process(String message)` changed to `process(Envelope e)` with `Envelope` encapsulating the `message`. Callers of `process()` will have to pass an Envelope instead, which is obtained from a factory method. The implementors of `process()` should augment their implementation to match the new type. They will first obtain the String message out of the Envelope. This change results in compile errors for the clients calling the method. If the arguments of a hook method are changed, this can result in any of Method Capture, Inconsistent Method, or Unimplemented Method.

Changed return type

This change is very similar to changed argument type. We observed one interesting type change in Eclipse. The return type of `IJavaBreakpointListener.breakpointHit()` was changed from Boolean to integer to allow listeners to vote 'do not care' in addition to 'suspend' and 'do not suspend'. A refactoring tool can only swap primitive types if there is a translation map between the values of the two different types. For the callers that assign the result of the method call or pass it to other method calls, this type of change can produce compile errors depending on what is the relationship between the new returned type and the expected type. For implementors of the changed method, due to the fact that statically types languages such as Java do not allow two methods in the same class to differ only by the return type, accidental Method Capture or Inconsistent method cannot occur. The compiler signals an error that the overridden method and the super class method differ in the return type.

Replaced method call

The clients of a method should call another method that is semantically equivalent and is offered in the same class. When there are no more callers to the original method, it is usually deleted. In Struts for example, clients of `FieldChecks.validateRange(..)` should call `FieldChecks.validateIntRange(..)` instead. If the replaced method was a hook method, this change can result in any of the Method Capture, Inconsistent Method, or Unimplemented Method.

Renamed method, renamed class, renamed field, and renamed package

These are used to give intention-revealing, self-explanatory names to methods, classes, class fields, and packages. These refactorings are well described in refactoring catalogs (see [4]). Renaming a method usually results in compile errors for the callers of the method. Renaming a hook method can result in any of the Method Capture, Inconsistent Method, or Unimplemented Method. Class renamings result in compile errors for both the instantiators and implementors of the class. Field renamings result in compile errors for the clients that access them. Package renamings result in compile errors for clients that use entities from the renamed package.

New hook method

Component producers factor out a method to provide 'hot spots' that are to be specialized by subclasses (see the Template Method in [20]). They add a new hook method in the super class (usually as an abstract method) that all non-abstract subclasses must override. We illustrate this with an example from Struts. Method `validate()` in class `ValidatorForm` calls the newly introduced method `getValidationKey()`:

```
public ActionErrors validate(ActionMapping mapping,HttpServletRequest request)
{
    ...
    String validationKey =
        getValidationKey(mapping, request);
    ...
}

String getValidationKey(ActionMapping mapping, HttpServletRequest request){
        return mapping.getAttribute();
}
```

Subclasses override `getValidationKey()` to provide the desired behavior. It might happen that an existing subclass already has a method with the same signature as the newly introduced hook method. In this case, the method provided by the inheritor gets captured by the parent class even though the inheritor did not intend this, therefore Method Capture occurs. Using a refactoring tool to perform this change would warn one when method capture happens.

Extra argument

Often two methods signatures are very similar, they only differ by an argument. The two methods do similar things, but one method can do extra things by making use of the extra argument. When eliminating duplicated code, the method with fewer arguments will usually be replaced by the method with more arguments. For the call sites of the displaced method, this change appears as if the method gained one more argument. The callers of the old method with fewer arguments will have to call the new method and pass a default value for the extra parameter.

Developers of the Mortgage framework decided that database connections should be reused from a connection pool rather then being created every time a database operation was required. In order to persist an object one would call the following method in the framework:

```
boolean persist(BusinessObject)
```

Inside the `persist` method a database connection would be created. The later version of this method looks as follows:

```
boolean persist(BusinessObject, DBConnection)
```

When a Web service calls this method it will pass along an existing database connection (in case that it owns one). When the null object is passed, the `persist` method will create a connection on the fly.

This change usually produces compile errors for the clients calling the method. When a hook method gains more arguments, any of the Method Capture, Inconsistent Method, or Unimplemented Method can occur.

Deleted class

Component producers delete a class when it is no longer supported or maintained due to lack of resources or because the implementation is too buggy. In Struts several classes acted like containers for particular objects. The container's name would suggest that it contains objects of a certain kind (e.g., ActionMappings holds a collection of ActionMapping objects). In a later version, the containers are superceded by general-purpose collection classes and then deleted. Class deletion results in compile errors for both the instantiators and implementors of the class.

Extracted interface

Component developers extract the signatures of the public methods offered by a class into an interface (in Java, interfaces are first-class entities). Clients of the old class should call its methods through the interface. This change is meant to make component code more extensible since new classes implementing the interface can be passed as a concrete implementation of the interface, without having to change client code. In JHotDraw 5.0, all of the key abstractions are extracted from their previous classes as interfaces. The interface takes the name of the class from which it was extracted. The old class implements the interface in one of two ways: as abstract class or as standard class. Abstract classes (such as `AbstractFigure`) provide default implementation but still need to be subclassed. Standard classes (such as `StandardDrawing`) can be used as is. The new prefix of the class (abstract or standard) makes it easy to distinguish between them. This change does not affect clients that were only invoking methods of the class. However, inheritors of the old classes should be changed to inherit from the corresponding abstract or standard classes, otherwise Method Unimplemented errors are thrown by the compiler.

Method object

This is a variation on Method Object described by Beck [22] and we illustrate it with an example from Eclipse. In class `AbstractDocumentProvider`, the modifier of `saveDocument()` method changed to *final* so that subclasses cannot override it anymore. A new method called `doSaveDocument()` was introduced and all of the code from `saveDocument()` moved to the new method. A `DocumentProviderOperation` object offers an `execute()` method that delegates to `doSaveDocument()`. The new implementation of `saveDocument()` creates an instance of the `DocumentProviderOperation` and then calls its `execute()` method. All previous implementors of `saveDocument()` must override `doSaveDocument()` instead.

Pushed down method

A service is no longer offered by the superclass but only by subclasses. Thus, we say that the corresponding method was pushed down in the class hierarchy. This change results in compile errors for

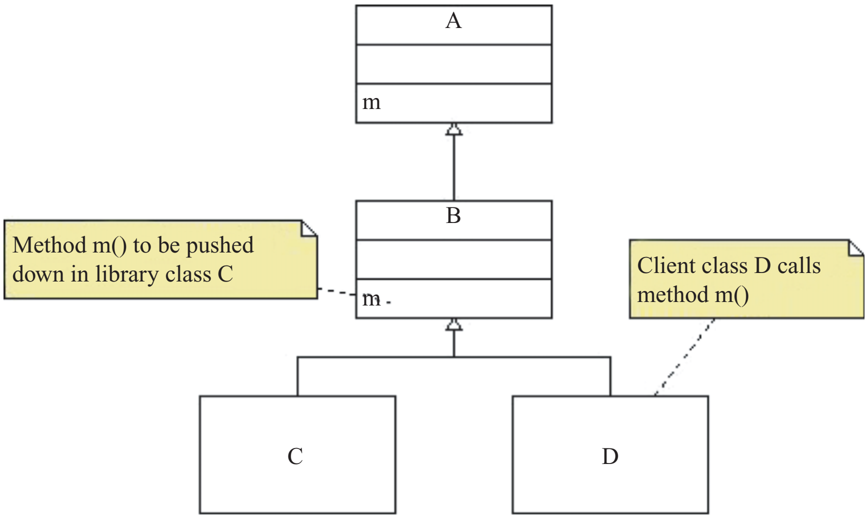

A
m
B
m
Method m() to be pushed down in library class C
Client class D calls method m()
C
D

Figure 2. When pushed down method can introduce a behavioral change.

the old method clients that call the moved method through the interface of its original class. However, a much subtler behavioral change can occur without being spotted by the compiler. Imagine the scenario in Figure 2. On the component side, the superclass `A` defines method `m`. This method is overridden in the subclass `B`. Class `C` is yet another component class that inherits from `B`. On the application side, class `D` inherits from `B` and, therefore, any call that it makes to method `m` gets dispatched to the implementation provided by `B`. In the next version of the component, without being aware of the presence of client class `D`, component providers push down the implementation of `m` from `B` to `C`. The client `D` will compile fine, but when it calls method `m`, the call gets dispatched to the implementation provided by `A`, thus leaving to a different behavior than when using the previous release.

Moved class

A class is moved to a different package in order to increase the cohesiveness of that package. This change breaks both the instantiators and implementors of the moved class.

Pulled up method

A method is moved in the parent class so that everyone can take advantage of the superclass logic. This change will not produce any compile errors, but can lead to an incorrect behavior. Consider the scenario in Figure 3. Before doing the upgrade, when the client class `D` calls method `m`, this call gets dispatched to the implementation provided by superclass `A`. In the next component version, `m()` gets

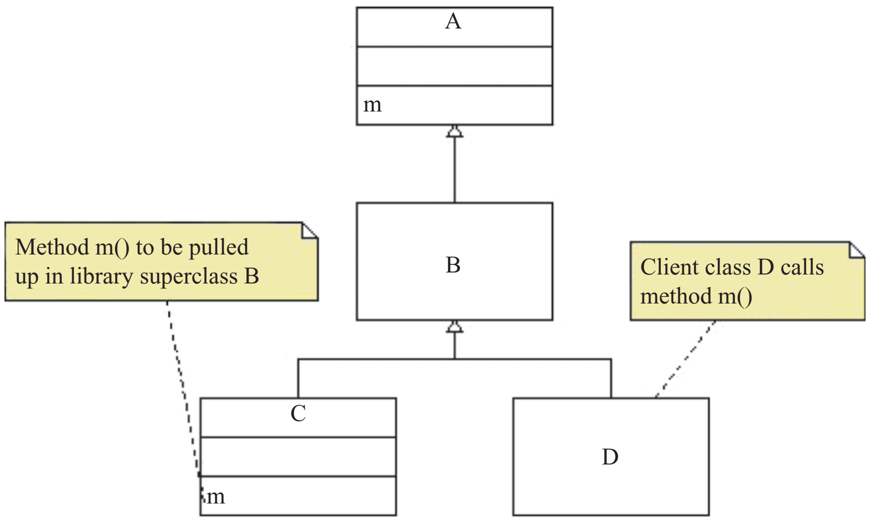

A
m
B
Method m() to be pulled up in library superclass B
Client class D calls method m()
C
m
D

Figure 3. When pulled up method can introduce a behavioral change.

pulled up from `C` to `B`. When client D calls it, the message will be dispatched to the implementation provided by class `B`.

Split package

In order to enhance the cohesiveness of classes in a package, component designers split a package into smaller, more cohesive units. In JHotDraw, a `figures` package was forked out of the `standard` package. This package provides a kit of standard figures and their related handle and tool classes. This change results in compile errors for both instantiators and implementors of the classes that belong to the new package.

Split class

Each class should have only a few responsibilities. When a class acquires too many responsibilities, the initial class cohesiveness will be fractioned into clusters of methods and fields that interact with each other closely. Component designers choose to 'purify' the cluttered class by moving responsibility to other classes. This is usually done by either using object composition or inheritance. In JHotDraw, `AbstractFigure` was split. It no longer keeps track of the attributes of a figure. The dictionary is maintained now in a subclass called `AttributeFigure`. This change allows one to create figure classes without having to use the figure's attributes implementation. Clients who were calling or implementing attributes-related API methods no longer compile.

Table III. Effects of changing method contract on callers and implementors.

What changes	How	Callers	Implementors
Precondition	Weaken	Compatible	**Broken**
Precondition	Strengthen	**Broken**	Compatible
Postcondition	Weaken	**Broken**	Compatible
Postcondition	Strengthen	Compatible	**Broken**

4.2. Behavioral modifications

We saw that structural transformations preserve the behavior of the component, but might cause applications to fail to compile. In contrast, behavioral modifications in the component might cause the application to compile fine with the new version. However, the application will not behave the same because the new version uses different assumptions.

New method contract

A contract is an agreement between the method provider and its clients [23]. The precondition is what the method assumes to be true before execution. A postcondition is what a method guarantees to be true after the method body has executed successfully (presuming that the precondition holds).

Des Rivieres [18] shows the effect of strengthening or weakening preconditions and postconditions on clients of a method as seen in Table III. The first column identifies what part of the contract changes. The second column gives the direction of change: strengthening or weakening the contract. The next two columns show whether the change is backwards compatible or it breaks existing method clients.

Consider the following method offered by the Collection interface:

```
/** @param coll a non-null Collection */
public boolean addAll(Collection coll);
```

Designers think about weakening the precondition so that it is acceptable to pass a null object. The callers of this method are not affected. However, an implementor like that below will throw a `NullPointerException` when it sends `size()` message to a null object:

```
public boolean addAll(Collection coll){
    //an implementation
    int size= coll.size();
    ....
}
```

If the precondition were strengthened (e.g., passed collection should not be shorter than a threshold), some existing callers of the method might not fulfill the requirements thus causing some faulty behavior. The existing implementors will not be affected since they assumed less than what is offered now.

Implement new interface

Developers of the component replace the interface implemented by a class with a different interface (with different contracts). Or they add a new interface to those a class already implements. In Struts, the latest version of class `LabelValueBean` implements a new interface, namely `Comparable`. The class now overrides methods `compareTo(Object)`, `equals(Object)`, and `hashCode()`. Older applications that compared instances of this class for equality might behave differently now that the class provides its own way for equality checks.

Changed events order

Similar to orchestra conductors, frameworks control the code contributed by applications. Usually the applications just respond when the conductor gives them the signal to participate. When the application make assumptions about the order in which the events are generated, it is vulnerable to any change in the sequence of events. For instance, in Eclipse 3.0, selection of items in tables and trees generates the event sequence MouseDown–Selection–MouseUp. In version 2.1, the event order was different under some platforms with the Selection event being generated first, i.e., the sequence Selection–MouseDown–MouseUp.

New enumeration constant

This change affects clients that rely on the set of all possible fields in an enumeration. In Eclipse 2.1, IStatus is an enumeration with four constants: OK, INFO, WARNING, and ERROR. Some clients used a switch case statement to check all the values of an enumeration. They treated the ERROR case in the *default* branch of the switch statement. Eclipse 3.0 adds a new constant, namely CANCEL. When CANCEL is passed around, the old clients will trap the new constant in their *default* branch, thus treating it like the ERROR case.

Miscellaneous

In addition to API changes there are other types of changes that may cause component-based applications to malfunction. Some of these changes might be deployment changes, classloader order changed, changes to build scripts and other configuration files, data format and interpretation changes. We noticed changes in the XML configuration and metadata files in all four studied frameworks. However, these are beyond the scope of this paper.

Table IV is a summary of Table II. The first column lists the components we studied. As we did in Table II, Eclipse* and Struts* denote recommended changes, that is changes that will become breaking changes in the next release. The second column gives the total number of breaking API changes (both structural and behavioral). The last column shows how many of the breaking API changes are refactorings.

Our findings suggest that most API breaking changes are small structural changes. This makes sense because large-scale changes lead to clients abandoning the component. For a component to stay alive, it should change through a series of rather small steps, mostly refactorings.

Table IV. Ratio of refactorings to all breaking API changes. Eclipse* and Struts* denote recommended changes.

Component	Number of breaking changes	Refactorings (%)
Eclipse	51	84
Eclipse*	99	87
Mortgage	11	81
Struts	136	90
Struts*	77	100
Log4J	38	97
JHotDraw	58	94

Table V. The impact of refactorings on backwards compatibility.

	Struts	Log4J
Refactorings	123	37
All other API changes	325	920
Percentage of refactorings	27.4%	3.8%
Impact of refactorings	90%	97%

For Struts and log4j we analyzed what percentage of all API changes (including addition of new API) are represented by refactorings (see Table V).

We used Van [16] to learn the number of addition and deletion of API classes and methods. The second row sums the API methods that were added or deleted from classes that exist in both versions, the number of API classes that were added or deleted in between the two versions, and the number of breaking API changes that are not refactorings. The row 'Percentage of refactorings' depicts how many of all API changes (including non-breaking changes such as the addition of new APIs) are refactorings. The row 'Impact of refactorings' depicts how many of all changes that break existing customers are refactorings. Table V shows that even though refactorings are a small percentage of all API changes (including the addition of an API), they have a large impact on backwards compatibility. Therefore, migration tools should focus on carrying out these types of changes.

5. ON MIGRATING COMPONENT-BASED APPLICATIONS

We found out that between the two versions we studied for each of the five systems, more than 80% of the breaking changes were refactorings. If all of the breaking changes were refactorings, it would make sense to use a refactoring tool to incorporate these changes in the applications. However, we feel that if tools could handle even 80% of the changes, they would reduce the burden of manual upgrades.

A migration tool based on refactorings needs to automate three steps. First, it needs to track the refactorings that happened between two versions of a component (no overhead for component producers). Refactorings can be recorded right at the moment when they were performed on the component. This requires that the refactoring engine automatically logs each performed refactoring. Most refactoring tools, including Eclipse, represent refactorings as objects, but do not necessary record all of the information needed to make these refactorings persistent. Thus, they will need to be changed to create a persistent log of refactorings. Component developers will ship this log of refactorings along with the new release of the component.

Second, at the application's site, the migration tool needs to load the representation of the refactorings into live refactoring objects. If refactorings are already objects, then this might be as simple as reconstructing the object that has previously been stored.

Third, the migration tool plays back the log of refactorings at the application's site. Now because component and application reside in the same workspace, all of the references are correctly updated. During this step, the tool needs to handle those cases when a refactoring cannot be played back in the new context. For instance, at the component site it was possible to rename an API method, but at the application site this renaming results in a conflict because the application already defines a method having the new name.

CatchUp [8] is a research prototype that aims to update applications by recording and playing back the refactorings. However, only a few refactorings have full record and replay support. Also it does not handle conflicts that might arise in step (3). Eclipse release 3.2 Milestone 4 is planned to extend its refactoring engine with full support for record and playback. We plan to evaluate in the future how effective it is to update applications using such an approach.

6. RELATED WORK

To our knowledge no quantitative study has been published about the kind of API changes that occur in components. Several categories of related work can be distinguished and are provided below.

Bansiya [24] and Mattson and Bosch [25] used metrics to assess the stability of frameworks. Their metrics can only detect the effect of changes in the framework and not the exact type of change (e.g., they observed that method argument types have been changed between subsequent versions whereas we observe whether they changed because of adding/removing of parameters or because of changing the argument types).

Mattson and Bosch [3] identified four evolution categories in frameworks: internal reorganization, changing functionality, extending functionality, and reducing functionality. Our findings confirm all four of the evolutions they have been describing.

There exists some limited tool support for detecting and classifying structural evolution. Detection of class splitting and merging was the main target of the tools described in [12–15]. Clone detection can be used to detect some refactorings such as renaming or moved method. Since none of these tools attempted to find all types of structural evolution, we had to analyze the changes manually.

Tool support for upgrading applications has been a long time interest. In [26–28], different annotations are discussed within the component's source code that can be used by tools to upgrade applications. However, writing such annotations is cumbersome. Balaban *et al.* [29] aimed to automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements. A more appealing approach would be if tools could generate this information.

As an alternative to refactorings, Steyaert *et al.* [21] introduced the notion of Reuse Contracts to guarantee structural and behavioral compatibility between frameworks and instantiations. On the same base line, Tourwe and Mens [30] introduced metapatterns and their associated transformations to document the framework changes. Owing to the rich semantics carried in such documentation, automated support for application migration can be possible. We agree that refactoring alone cannot solve all of the migration problems. However, automated refactoring is supported by the most recent IDEs. We showed that refactorings can effectively describe over 80% of the breaking API changes that actually occur in component evolution.

7. CONCLUSIONS AND FUTURE WORK

API changes have an impact on applications. One might argue that library engineers should maintain old versions of the library so that applications built on those versions continue to run. However, this results in version proliferation and high maintenance costs for the producer. In practice, it is application engineers who adapt to the changes in the library.

We looked at one proprietary and three open-source frameworks and one library and studied what changed between two major releases. Then we analyzed those changes in detail and found out that in the five case studies, 84%, 81%, 90%, 97%, and 94% of the API-breaking changes are structural, behavior-preserving transformations (refactorings).

There are several implications of our findings. First, they confirm that refactoring plays an important role in the evolution of components. Second, they offer a ranking of refactorings based on how often they were used in the five systems. Refactoring vendors should prioritize to support the most frequently used refactorings. Third, they suggest that component producers should document the changes in each product release in terms of refactorings. As refactorings carry rich semantics (in addition to the syntax of changes) they can serve as explicit documentation for both manual and automated upgrades. Fourth, migration tools should focus on support to integrate into applications those refactorings performed in the component. Our future work aims to produce such migration tools based on refactorings.

Refactoring engines guarantee that the structural changes they perform will not break the applications. A migration tool based on refactoring engines should be able to do most of the tedious job of upgrading to a new version. Application developers will have to carry only a small fraction (less than 20%) of the remaining changes. These are changes that require human expertise. Future work will evaluate how much of the migration effort is saved by using a refactoring-based migration tool.

Our findings cannot prove without a doubt that the majority of API-breaking changes are refactorings, but they give us the confidence that this is the trend. Also the five case studies are Java systems. Perhaps the programming language has an influence on the types of change that happen in components. It could be that for languages in which the cost of change is relatively low, there would be more changes than in languages where the cost is high. More research and case studies are needed to prove our position.

The availability of powerful migration tools will also change things for the component designers. Without fear that they will break the clients, the designers will be bolder in the kind of changes they can make to their designs. Given this new found freedom, designers will not have to carry bad design decisions made in the past. They will purge the design to be easier to understand and reuse.

ACKNOWLEDGEMENTS

We thank many people who reviewed drafts: Darko Marinov, Johannes Henkel, Amer Diwan, Oscar Nierstrasz, Erich Gamma, Serge Demeyer, Doru Girba, Adam Kiezun, Frank Tip, Riley White, Danny Soroker, Bob Hanmer, anonymous reviewers, and the members of Software Architecture Group at UIUC. Your feedback made a difference.

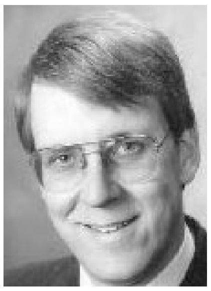
REFERENCES

1. Lientz BP, Swanson EB. *Software Maintenance Management: A Study of the Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley: Reading MA, 1980.
2. Chapin N, Hale J, Khan K, Ramil J, Tan W-G. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2001; **13**(1):3–30.
3. Mattson M, Bosch J. Frameworks as components: A classification of framework evolution. *Proceedings of NWPER'98: Nordic Workshop on Programming Environment Research*. Bergen Print Service: Bergen, Norway, 1998; 16–74.
4. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999.
5. Opdyke WF, Johnson RE. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *Proceedings of SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*. ACM Press: New York NY, 1990.
6. Graver J. The evolution of an object-oriented compiler framework. *Software—Practice and Experience* 1992; **22**(7):519–535.
7. Tokuda L, Batory D. Evolving object-oriented designs with refactorings. *Journal of Automated Software Engineering* 2001; **8**:89–120.
8. Henkel J, Diwan A. CatchUp! Capturing and replaying refactorings to support API evolution. *Proceedings of ICSE'05: International Conference on Software Engineering*. ACM Press: New York NY, 2005; 274–283.
9. Dig D, Johnson R. The role of refactorings in API evolution. *Proceedings of ICSM'05: International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2005; 389–398.
10. Mattsson M, Bosch J, Fayad M. Framework integration. Problems, causes, solutions. *Communications of ACM* 1999; **42**(10):80–87.
11. Johnson R. Documenting frameworks using patterns. *Proceedings of OOPSLA'92: Conference on Object-Oriented Programming, Systems, Languages and Applications*. *ACM SIGPLAN Notices* 1992; **27**(10):63–72.
12. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. *Proceedings of OOPSLA'00: Conference on Object-Oriented Programming, Systems, Languages and Applications*. *ACM SIGPLAN Notices* 2000; **35**(10):166–177.
13. Van Rysselberghe F, Demeyer S. Reconstruction of successful software evolution using clone detection. *Proceedings of IWPSE'03: International Workshop on Principles of Software Evolution*. IEEE Computer Society Press: Los Alamitos CA, 2003; 126–130.
14. Antoniol G, Di Penta M, Merlo E. An automatic approach to identify class evolution discontinuities. *Proceedings of IWPSE'04: The 7th International Workshop on Principles of Software Evolution*. IEEE Computer Society Press: Los Alamitos CA, 2004; 31–40.
15. Godfrey M, Zou L. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 2005; **31**(2):166–181.
16. Girba T, Ducasse S, Lanza M. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. *Proceedings of ICSM'04: International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2004; 40–49.
17. Demeyer S, Ducasse S, Nierstrasz O. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann: San Mateo CA, 2003.
18. Des Rivieres J. Evolving Java-based APIs. *White Paper*, Object Technology International. Available at: http://www.eclipse.org/eclipse/development/java-api-evolution.html [16 March 2006].

19. Laitinen M. Framework maintenance: Vendor viewpoint. *Object-Oriented Application Frameworks: Problems and Perspectives*, Fayad ME, Schmidt DC, Johnson RE (eds). Wiley: New York NY, 1999.
20. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading MA, 1995.
21. Steyaert P, Lucas C, Mens K, D'Hondt T. Reuse contracts: Managing the evolution of reusable assets. *Proceedings of OOPSLA'96: Conference on Object-Oriented Programming, Systems, Languages and Applications*. *ACM SIGPLAN Notices* 1996; **31**(10):268–285.
22. Beck K. *Smalltalk Best Practice Patterns*. Prentice-Hall: Englewood Cliffs NJ, 1997.
23. Meyer B. Design by contract. *Technical Report TR-EI-12/CO*, Interactive Software Engineering, Inc., 1986.
24. Bansiya J. Evaluating application framework architecture structural and functional stability. *Object-Oriented Application Frameworks: Problems and Perspectives*, Fayad ME, Schmidt DC, Johnson RE (eds.). Wiley: New York NY, 1999.
25. Mattson M, Bosch J. Three evaluation methods for object-oriented frameworks evolution—application, assessment and comparison. *Research Report 1999:20*, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden, 1999.
26. Chow K, Notkin D. Semi-automatic update of applications in response to library changes. *Proceedings of ICSM'96: International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1996; 359–368.
27. Keller R, Hlzle U. Binary component adaptation. *Proceedings of ECOOP'98: European Conference on Object-Oriented Programming* (*Lecture Notes in Computer Science*, vol. 1445). Springer: Berlin, 1998; 307–329.
28. Roock S, Havenstein A. Refactoring tags for automatic refactoring of framework dependent applications. *Proceedings of Extreme Programming Conference'02* (*Lecture Notes in Computer Science*, vol. 2418). Springer: Berlin, 2002; 182–185.
29. Balaban I, Tip F, Fuhrer R. Refactoring support for class library migration. *Proceedings of OOPSLA'05: Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM Press: New York NY, 2005; 265–279.
30. Tourwe T, Mens T. Automated support for framework-based software. *Proceedings of ICSM'03: International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2003; 148–157.

AUTHORS' BIOGRAPHIES

Danny Dig is working towards a PhD in Computer Science at the University of Illinois at Urbana-Champaign (UIUC). He got his MS in Computer Science from Polytechnic University of Timisoara, Romania, where he built JavaRefactor, the first open-source refactoring engine for Java. His research interest is in object-oriented software engineering, with an emphasis on techniques and tools for improving software design through refactorings. Currently, refactoring engines operate within a closed-world paradigm, that is, a refactoring engine can update only the source code residing in the same sandbox. His goal is to fit refactorings into an open-world paradigm, where libraries and frameworks are developed at one site and are reused all over the world.



Ralph Johnson is a Research Associate Professor at the University of Illinois at Urbana-Champaign. He received his BA from Knox College in 1977 and his PhD in Computer Science from Cornell University in 1987. He is one of the four coauthors of '*Design Patterns*' and the leader of the group that developed the Smalltalk Refactoring Browser, the first refactoring tool.