The Role of Refactorings in API Evolution

Danny Dig and Ralph Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin
Urbana, IL 61801, USA
{dig, johnson}@cs.uiuc.edu

Abstract

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 three frameworks and one library. We discovered that the changes that break existing applications are not random, but they 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.

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 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 [17]. Others [5, 20] 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) way to update component-based applications. This paper is 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 library and framework as component, unless a distinction is necessary). Classes contain a mixture of private and public methods. The public methods are the ones that are meant to be used by application programmers. The set of public methods of a class library make up its API (Application Programmer Interface). 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[9]. 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 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.

The original work on refactoring was motivated by framework evolution. Opdyke [22] looked at the Choices operating system and the kind of refactorings that occurred as it evolved. Graver [13] studied an object-oriented compiler framework as it went through three iterations. Tokuda and Batory [26] describe the evolution of two frame-

works, 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, though usually from the point of view of a component developer, not of a component reuser. However, CatchUp [14] 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.

In this paper, we look at three frameworks and one library (see Table 1 and Section 2) developed by four different groups. Three are commonly used open source and one is a proprietary framework. All 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.

	Eclipse 3.0	Mortgage	Struts 1.2.4	log4j 1.3
Size(KLOC)	1,923	52	97	62
API Classes	2,579	174	435	349
BreakingChanges	51	11	136	38
ChangeLogs	24	-	16	4

Table 1. Size of the studied components. The number of classes in API denote only those classes that are meant to be reused. ChangeLogs give the size (in pages) of documents describing the API changes. The logs were provided by the component developers.

We analyze and classify the API changes in the four systems (Section 3). Some API changes like expansion of the component through addition of new classes and methods will not affect existing users. We discard these type of API changes and only focus on the API changes that break compatibility with older applications. We learned that for the four systems we studied, respectively 84%, 81%, 90% and 97% 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). These results made us believe that refactoring plays an important role as mature components evolve.

2. Overview of the Case Studies

This section describes briefly the components that we used as case studies. We chose well known frameworks and libraries from both proprietary and open source realm in order to check whether the production environment affects the type of API changes. We were unbiased in the selection of the case studies, only concern being that the systems have decent 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 in between the two versions. Secondly, 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 [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 like 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. a 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 [18]. 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 we write this, there are about 50 services that use the framework.

2.3. Struts Framework

Struts [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, like JDBC and EJB, as well as most any third-party packages, like 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. Because of 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 the API changes reveal consolidation work that was done in between the two releases. The framework developers eliminated duplicated code and removed unmaintained or buggy code.

2.4. log4j Library

log4j [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 30KLOC to 62KLOC. The library grew by improving on existing components (like Chainsaw, a visualization toolkit for loggers) or adding new components (like support for plugins as a way to extend the library).

2.5. Collecting the Data

Our case study components are considered medium to large size (Mortgage is 50 KLOC, Eclipse is roughly 2 million LOC). To tackle the API changes in such large systems we could have used tools. For instance, Demeyer et al. [8] describe how they used metrics tools to discover refactorings. However, because of preserving backward compatibility, most API changes don't happen overnight but follow a long deprecate-replace-remove cycle. Therefore an obsolete API can coexist with the new API for a long time. This introduces enough noise that tools might mislead us about the exact kind of change that happened.

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 can't 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 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[1], the documents called "Incompat-

1 Section: Eclipse 3.0 Plugin Migration Guide

ibilities 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[2]. For Log4J we studied "Preparing for log4j version 1.3"[3] .

Sometimes the documents would be vague, reading for example "method M in class X is deprecated". Because of 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 we interviewed the framework and application developers and then studied the source code. We classified all 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 [8, 28, 1, 12] 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, manual method seems the only alternative. We double-checked our quantitative analysis by using a tool (Van [11]) and heuristics (like in [8]). 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 some 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. The tool did not scale up for Eclipse and we do not own the source code of the proprietary Mortgage framework.

2 http://struts.apache.org/userGuide/release-notes-1.2.4.html

3 http://www.qos.ch/logging/preparingFor13.jsp

3. How APIs Change

This section describes the API changes that occurred in the four case studies. The first subsection talks about APIs and what does it mean for an API change to break compatibility with applications. The subsection BREAKING API CHANGES presents the empirical data gathered from the breaking changes we noticed in the case studies. The following subsections analyze in detail two kinds of breaking changes, namely semantics-preserving changes (structural changes) and semantics-modifying changes (behavioral modifications).

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 [7]. For instance, Eclipse places a public class that is not 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[4].

A **non-breaking change** is backwards compatible. Such a change can be an enhancement like addition of new mod-

4 We go with a loose definition of failure but in domains like embedded systems, our notion of reliability might not be sufficient

ules 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.

Although there are a number of techniques used to facilitate component changes without breaking the clients[23], breaking API changes happen all the time. Our goal is to provide migration tools that can incorporate breaking changes. The next section focuses on these changes.

3.2. Breaking API Changes

From anecdotal experience with components we noticed that breaking changes are perceived as extremely disturbing in the development life cycle of component-based applications. 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[16].

Table 2 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 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.

The next two sections categorize the changes in Table 2 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.

3.3. Structural Transformations

Next we describe the types of structural changes that we noticed in the studied components (Table 2). We describe the changes that occurred in the component instead of giving prescriptions about how one developer should migrate to the new version.

Type of change	E	E*	M	S	S*	L
Moved Method	16	13	-	11	28	9
Moved Field	-	45	-	18	2	5
Deleted Method	2	2	-	24	32	-
ChangedArgumentType	5	-	4	18	4	11
Renamed Method	4	-	-	16	5	8
Replaced Method Call	1	20	-	8	4	-
New Hook Method	4	2	2	7	-	-
Extra Argument	3	2	2	1	1	-
Deleted Class	-	-	-	9	-	-
Renamed Field	-	-	-	6	1	-
Changed Return Type	2	-	1	2	-	2
Renamed Class	-	1	-	2	-	2
Method Object	3	-	-	-	-	-
Pushed Down Method	3	-	-	-	-	-
Moved Class	-	2	-	-	-	-
Pulled Up Method	-	-	-	1	-	-
New Method Contract	3	12	1	8	-	1
Impl.New Interface	1	-	1	5	-	-
Changed Event Order	3	-	-	-	-	-
New Enum Constant	1	-	-	-	-	-

Table 2. Types of BREAKING API CHANGES and the number of these changes in Eclipse (E), Mortgage (M) , Struts (S) and log4j (L). 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.

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 compile with the new version of the component. Once the application developer solves the compile errors, the application's behaviour is the same (structural changes do not introduce new behavior).

MOVED METHOD. The most common way that instance methods moved in Eclipse is by becoming static methods. 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[9].

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 usually they are declared as static fields. In Eclipse, Struts and log4j only fields that were constants moved to another home class.

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 the references to old method were replaced, the method is deleted since it's a remnant of the obsolete API.

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.

RENAMED METHOD, RENAMED CLASS and RENAMED FIELD are used to give intention revealing, self-explanatory names to methods, classes and class fields. These refactorings are well described in refactoring catalogs (see [9]).

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 instead `FieldChecks.validateIntRange(...)`.

NEW HOOK METHOD. Component producers factor out a method to provide "hot spots" that are to be specialized by subclasses (see Template Method in [10]). 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);
    ...
}

public 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 (see Method Capture in [25]). 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, usually the method with fewer arguments will be replaced by the one 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 `persist` method a database connection would be created. The later version of this method looks like:

```
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.

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 superseded by general-purpose collection classes and then deleted.

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 "don't care" in addition to "suspend" and "don't suspend". A refactoring tool can only swap primitive types if there is a translation map between the values of the two different types.

METHOD OBJECT. This is a variation on Method Object described by Beck [4] and we'll 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 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.

What changes	How	Callers	Implementors
Precondition	weaken	compatible	**broken**
Precondition	strengthen	**broken**	compatible
Postcondition	weaken	**broken**	compatible
Postcondition	strengthen	compatible	**broken**

Table 3. Effects of Changing Method Contract on Callers and Implementors

MOVED CLASS. A class is moved to a different package in order to increase the cohesiveness of that package.

PULLED UP METHOD. A method is moved in the parent class so that everyone can take advantage of the superclass logic.

3.4. 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 won't behave the same since the new version uses different assumptions.

NEW METHOD CONTRACT. A contract is an agreement between the method provider and its clients [21]. 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). In frameworks, due to extensive usage of callbacks (hook methods), with regard to contracts we must consider two types of method clients: *callers* and *implementors*.

Des Rivieres [23] shows the effect of strengthening or weakening preconditions and postconditions on clients of a method in Table 3. 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 the one 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 the ones 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 fallible 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 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. Besides 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 three studied frameworks. However, these are beyond the scope of this paper.

Component	# Breaking Changes	% Refactorings
Eclipse	51	84%
Eclipse*	99	87%
Mortgage	11	81%
Struts	136	90%
Struts*	77	100%
Log4J	38	97 %

Table 4. Ratio of refactorings to all breaking API changes. Eclipse* and Struts* denote recommended changes.

	Struts	Log4J
Refactorings	123	37
All Other API Changes	325	920
Percentage of Refactorings	27.4%	3.8%
Impact of Refactorings	90%	97%

Table 5. The impact of refactorings upon backwards compatibility

Table 4 is a summary of Table 2. The first column lists the components we studied. As we did in Table 4, 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.

For Struts and log4j we analyzed what percentage of all API changes (including addition of new API) are represented by refactorings (see Table 5). We used Van [11] to learn the number of addition and deletion of API classes and methods. 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. Row 'Percentage of Refactorings' depicts how many of all API changes (including non-breaking changes like addition of new APIs) are refactorings. Row 'Impact of Refactorings' depicts how many of all changes that break existing customers are refactorings. Table 5 shows that even though refactorings are a small percentage of all API changes (including addition of

API), they have a large impact upon backwards compatibility. Therefore, migration tools should focus on carrying out these types of changes.

4. 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 [3] and Mattson [19] 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 [20] 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 tools described in [8, 28, 1, 12]. Clone detection can be used to detect some refactorings like 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. [6, 15, 24] discuss different annotations within the component's source code that can be used by tools to upgrade applications. However, writing such annotations is cumbersome. Balaban et al.[2] aim 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.

Henkel and Diwan [14] describe CatchUp, a research prototype of a refactoring-based migration tool. We are collaborating with our colleagues from University of Colorado to turn CatchUp into a full-featured, reliable tool. CatchUp is integrated with the Eclipse development environment and uses a record and playback technique. As component developers refactor their code, CatchUp records the refactorings. Along with the new version of the component, its developers ship this log of refactorings. When an application developer upgrades to a new version, CatchUp will playback on the client code all the refactorings that were shipped with the component. Our paper provides the motivation that refactoring-based migration tools are likely to be useful in the migration task due to the large number of refactorings that occur during component evolution.

As an alternative to refactorings, Steyaert et. all [25] introduce the notion of Reuse Contracts to guarantee structural and behavioral compatibility between frameworks and instantiations. On the same base line, Tourwe and Mens[27] introduce metapatterns and their associated transformations to document the framework changes. Because of the rich semantics carried in such documentation, automated support for application migration can be possible. We agree that refactoring alone cannot solve all the migration problems. However, automated refactoring is supported by most recent IDEs. We showed that refactorings can effectively describe over 80% of the breaking API changes that actually occur in component evolution.

5. 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 two open source frameworks and one library and studied what changed in between versions. Then we analyzed those changes in detail and found out that in the four case studies, respectively 84%, 81%, 90%, and 97% 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 four 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. Because refactorings carry rich semantics (besides 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 won't break the applications. A migration tool based on refactoring engines (like CatchUp[14]) 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 breaking API changes are refactorings, but they give us the confidence that this is the trend. More research and case studies are needed to formally prove our position.

The availability of powerful migration tools will change things for the component designers as well. Without fear that they 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 won't have to carry bad design decisions made in the past. They will purge the design to be easier to understand and reuse.

6. 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, anonymous reviewers, and the members of Software Architecture Group at UIUC. Your feedback made a difference.

References

[1] 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 (IWPSE'04).

[2] I. Balaban, F. Tip, and R. Fuhrer: *Refactoring Support for Class Library Migration*, to appear in Proceedings of OOPSLA'05

[3] J. Bansiya: *Evaluating Application Framework Architecture Structural and Functional Stability*, in Object-Oriented Application Frameworks: Problems and Perspectives, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds), Wiley & Sons, 1999.

[4] K. Beck:*Smalltalk Best Practice Patterns*, Prentice Hall, 1997

[5] N. Chapin, J. Hale, K. Khan, J. Ramil, and W.-G. Than: *Types of software evolution and software maintenance*, in Journal of Software Maintenance 13(1): 3-30 (2001)

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

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz: *Object-Oriented Reengineering Patterns*, Morgan Kaufmann Publishers, 2003

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz: *Finding Refactorings via Change Metrics*, in Proceedings of OOPSLA'00, pp166-177

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

[10] E.Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[11] T. Girba , S. Ducasse, and M.Lanza: *Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes*, in Proceedings of ICSM '04, pp 40-49

[12] M. Godfrey and L. Zou: *Using Origin Analysis to Detect Merging and Splitting of Source Code Entities*, in IEEE Transactions on Software Engineering, vol 31(2), 2005, pp. 166-181

[13] J. Graver: *The Evolution of an Object-oriented compiler framework* in Software, Practice & Experience archive 22(7): 519-535 (1992) published by Wiley & Sons

[14] J. Henkel and A. Diwan:*CatchUp! Capturing and Replaying Refactorings to Support API Evolution*, in Proceedings of ICSE '05, pp 274-283

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

[16] 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.

[17] B.P. Lientz and E.B. Swanson: *Software maintenance management: a study of the computer application software in 487 data processing organizations*, Addison-Wesley, 1980.

[18] M. Mattsson, J. Bosch, and M. Fayad: *Framework Integration. Problems, Causes, Solutions*, in Communications of ACM 42(10): 80-87 (1999)

[19] M. Mattson and J. Bosch: *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

[20] M. Mattson and J. Bosch: *Frameworks as Components: A Classification of Framework Evolution*, in Proceedings of NWPER98 Nordic Workshop on Programming Environment Research, Ronneby, Sweden, August 1998, pp. 16-74

[21] B. Meyer: *Design by Contract*, Prentice Hall, 2005

[22] W. F. Opdyke and R.E. Johnson: *Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems*, in Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA '90)

[23] J. Des Rivieres: *Evolving Java-based APIs*, O.T.I, http://www.eclipse.org/eclipse/development/java-api-evolution.html

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

[25] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt: *Reuse Contracts: Managing the Evolution of Reusable Assets*, in Proceedings of OOPSLA '96, 268-285

[26] L. Tokuda and D. Batory: *Evolving Object-Oriented Designs with Refactorings*, in Journal of Automated Software Engineering 8: 8-20 (2001)

[27] T. Tourwe and T. Mens: *Automated support for framework-based software*, in Proceedings of ICSM '03, pp 148- 157

[28] F. Van Rysselberghe and S. Demeyer: *Reconstruction of Successful Software Evolution Using Clone Detection*, in Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '03), pp 126-130