Exploring the Intent behind API Evolution: A Case Study

Conference Paper · October 2011

DOI: 10.1109/WCRE.2011.24 · Source: DBLP

2 authors:

Daqing Hou
Clarkson University
57 PUBLICATIONS **670** CITATIONS
SEE PROFILE

Xiaojia Yao
Clarkson University
2 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Exploring the Intent behind API Evolution: A Case Study

Daqing Hou
Electrical and Computer Engineering
Clarkson University, Potsdam, NY USA 13699
dhou@clarkson.edu

Xiaojia Yao
Computer Science
Clarkson University, Potsdam, NY USA 13699
yaox@clarkson.edu

***Abstract*—Reuse has significantly improved software productivity and quality. An application interacts with a reused system through its Application Programming Interfaces (API). To make the life of the application developer easier, it is desirable for the API to be both sufficiently powerful and stable. Unfortunately, in reality APIs inevitably change, to be more powerful or to remove design flaws. This may create additional work for the application developer to adapt to the changed API. Thus, to counter the negative impacts of API evolution, we need to study how and why APIs are evolved. To that end, we performed a detailed analysis of the evolution of a production API. In particular, we categorized the changes to the API according to its domain semantics and design intent. We discussed the implications of our findings for both API designers and application developers.**

***Keywords*-Frameworks and libraries; APIs; software evolution; design intent; empirical studies.**

I. Introduction

Application frameworks and software libraries have clearly contributed substantially to programmer productivity and software quality [1], [2], [3]. Software frameworks and libraries provide leverage in large part because they are used by many applications through the published API (Application Programming Interfaces). Stable APIs are important because they separate the applications from changes in the frameworks and libraries. That is, as long as there is no change in the syntax or semantics for the subset of APIs that an application depends on, the application can continue to use the updated libraries by simply importing and linking with the new version, without modifications.

However, in reality, frameworks and libraries are indeed changed from time to time. Motivated by the need to assist programmers in the tedious work of adapting the application source code to the library interface changes, many automated tools have been proposed in the literature [4], [5], [6], [7], [8], [9]. Others have studied the evolution of API changes from the perspective of automated refactoring [10], [11]. However, to our knowledge, there does not exist a study that is designed to recover the complete history as well as the intents behind the evolution of a production API. Our work is a first attempt to fill in this gap.

Studying the evolution of production APIs can be valuable in that it will improve our understanding of API design and evolution as well as our ability to plan for and control API changes [12]. More specifically, from multiple such studies, we can incrementally build up a catalog of intents for API change along with concrete instances for each category of intent. To that end, a study like ours can contribute to our general knowledge of API design and evolution.

The results of such a study can be useful for both API designers and application developers. Since critical APIs are important for applications that depend on them, API developers are obligated to deliver high quality APIs. We just cannot casually put together a set of classes and call it an API. Instead, API developers need exemplars and guidelines for designing new APIs. They need to know what are the common problems that they should avoid, how other APIs are changed over time, and how their own API may be changed in the future. To that end, our results can be a useful resource that API designers can refer to. Furthermore, our results can also be used by application developers to comprehend and talk about API changes. Finally, our results may inform the design of new tools to further automate the adaptation of applications to API changes.

In this paper, we report on a case study on how and why APIs evolve by analyzing the evolution of a production API (AWT/Swing) in detail. By drawing on the rich set of data on the evolution of this particular API, we created a catalog of change intents. We also provided concrete instances to help illustrate each kind of intent. This catalog can be used as the beginning of an ontology for talking about API changes.

Our approach is semi-automated, relying mainly on the history of API changes that the AWT/Swing developers recorded in code comments. To accurately interpret the intention behind each API change, we also made use of the design knowledge of the API at both the architectural and detailed design levels. For each API change, we explored API use scenarios to understand the impacts that the API change has on client codes, for example, by searching for evidence on the Internet. Finally, the process of identifying each category of change intent was highly iterative. Often a general change intent is initially motivated by a specific API change. If it is further confirmed by more instances of changes, we then keep it in our catalog.

The remainder of this paper is structured as follows. Section II describes the methodology of our study. Section III reports the results of our case study. Section IV discusses the

implications of our results. Section V summarizes the threats to the validity of this study. Finally, Section VI presents related work, and Section VII concludes the paper.

All the data and the program we used to extract API evolution information can be found at the following URL: **www.clarkson.edu/~dhou/projects/wcre2011.tar.gz**

II. Data Collection and Analysis Methodology

A. Data Characteristics and Collection

In this study, we extracted and analyzed the evolution history of the AWT/Swing API from JDK. We chose to study AWT/Swing because they are the essential components of the JDK and have been updated together with each major version of JDK. They are long lived, production APIs that are used by thousands of developers. Table I depicts the release history of JDK, which spans almost 11 years.

Table I: Major versions of Java and release dates.

Versions	Public Release Dates
JDK 1.0	January 23, 1996
JDK 1.1	February 19, 1997
J2SE 1.2	December 8, 1998
J2SE 1.3	May 8, 2000
J2SE 1.4	February 6, 2002
J2SE 5.0 (1.5)	September 30, 2004
Java SE 6 (1.6)	December 11, 2006

The main source of data for our study is the official JavaDoc for the AWT/Swing API. To study how and why API elements evolve, we need to produce the lists of Java packages, classes, and methods that were newly added, deprecated, or updated. Fortunately, the JDK developers have established and fairly closely followed a coding style [13], so high-quality information about its evolution is made available in the JavaDoc for each version of JDK released. (Because we inspected all the API elements as well as their contexts, we were able to discover several cases where the coding standard was not followed. But we discovered only less than a dozen of such cases and, thus, we are confident that the JavaDoc is a reliable source of information for the evolution of AWT/Swing.) The JavaDoc is automatically generated from the source code comments using the Javadoc tool [13]. We took advantage of this and wrote a Java program to extract the needed information from the AWT/Swing part of the Java SE 6 documentation.

In what follows, we briefly summarize the JDK comment standards that make this extraction possible. The @since tag is used to specify the product version when the Java name was added to the API specification. For example, if a package, class, interface or member was added to the Java 2 Platform, Standard Edition, API Specification at version 1.2, the JDK developer would insert the following comments into the code:

```
/**
* @since 1.2
*/
```

More specifically, when a package is introduced, an @since tag is specified in its package description and each of its classes. When a class (or interface) is introduced, one @since tag is specified in its class description but no @since tags in the members. An @since tag is specified only for members that are added later than the initial version of the class. Thus members added together with a class have no @since tags.

When a method is deprecated, an @deprecated tag should be inserted into its header comment. The @deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Subsequent sentences can also explain why it has been deprecated.

For Javadoc 1.1, the standard format is to create a pair of @deprecated and @see tags. For example:

```
/**
* @deprecated As of JDK 1.1, replaced by setBounds
* @see #setBounds(int,int,int,int)
*/
```

For Javadoc 1.2 and later, the standard format is to use the @deprecated tag and the in-line {@link} tag. This creates the link in-line, where a developer wants it. For example:

```
/**
* @deprecated As of JDK 1.1, replaced by
* {@link #setBounds(int,int,int,int)}
*/
```

The Javadoc 1.2 standard also requires that if the member has no replacement, the argument to @deprecated should be "No replacement".

The Javadoc tool extracts these tags as well as their arguments, from the source code comments and put them into the generated JavaDoc.

Since the JavaDoc is well structured, we wrote a small Java program to take advantage of the structure and extract the needed information using regular expressions. Specifically, for each class in a package, this program takes the html documentation file for the class as input, and outputs the version where the class was added into the package, as well as any new methods and the versions when they were added. Since a later version of JDK documentation subsumes the content of an earlier version, to obtain the data needed by this study, we processed only the official Java SE 6 documentation [1].

In the end, with the assistance of our program, for each version, we created a list of new classes that were added to each existing package, a list of new methods added to each existing class, and a list of deprecated methods as well as their replacements, if there is one. We also created a separate list of new overriding methods (methods that override a supertype method) since overriding a method would not be very interesting from the perspective of API evolution.

[1]The Java SE 6 documentation is available for download at http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html. Last verified: June 25, 2011.

We manually tracked down the package history as shown in Table II. We noticed that few deprecated classes were not documented as required by the JavaDoc standard, so we also manually checked all the class documentation to make sure that we have the complete list of deprecated classes.

B. Analysis Methodology

Our analysis of the resulting data is largely done manually and iteratively. We first partitioned the changed API elements into *deprecated* and *new*. For deprecated API methods and classes, we identified any replacements that they might have. We analyzed the deprecated methods one by one to find out what has caused the deprecation and why. We identified the change intent and categorized the API changes accordingly. We were especially careful in analyzing the deprecated API elements because they may give us valuable lessons as to what has been done wrong in the past.

For newly added API methods and classes, we first grouped them into design features to gain a high level understanding of their purposes and roles in the overall AWT/Swing framework. We then categorized the changes at the feature level as *new*, *enhanced*, or *redesign*.

In addition to the JavaDoc, we made use of additional source of information. Specifically, to group API methods by features, we needed to know the design of AWT/Swing. For each major release of JDK, there is also a brief release note that summarizes the fixed major bugs, changes, new features, and major enhancements introduced in that release. However, the release notes briefly discuss only a small subset of the actual API changes that the API developers consider important. So they alone are insufficient for our study. We occasionally consulted the release notes, mainly as a way to ensure that we have not missed anything significant.

III. Results

This section presents the results of our study. Section III-A discusses the evolution of AWT/Swing at the package and class levels. Section III-B presents the results for deprecated and replaced methods. Section III-C presents the results for new API methods. Finally, Section III-D presents the results for overriding methods.

A. Evolution of packages and classes in AWT/Swing

Table II shows the history of the AWT/Swing packages. As shown, most of the packages were added in 1.2, the version when Swing was first released. For AWT, this evolution path is likely to be the result of a careful prioritization in terms of the importance of each package and its features. For example, the event package was added when the event processing logic was redesigned (see III-C4), since having the right architecture for event processing would have been critical for the smooth growth of the toolkit in the subsequent versions. By contrast, Swing appears to have been designed and planned more up front in that all but one package (plaf.synth) were added in 1.2.

Table II: Java versions when AWT/Swing packages were added. No packages were added for Versions 1.4 and 1.6. AWT packages have java.awt as the prefix for their fully-qualified names, and javax.swing for Swing packages.

1.0	1.1	1.2	1.3	1.5
AWT				
java.awt, image	event, datatransfer	dnd, geom, color, font, image.renderable, print, im	im.spi	
Swing				
		javax.swing, event, plaf, undo, border, text, table, tree, text.html, text.html.parser, text.rtf, colorchooser, filechooser, plaf.basic, plaf.metal, plaf.multi		plaf.synth

Table III shows the numbers of classes added in each version. A new class may be added to an existing package for several reasons: the introduction of a new feature or a new instance of an existing kind of features, a change to an existing feature, or a helper class. We have created a spreadsheet listing all of such new classes and categorized them by design features, which can be downloaded online from the URL provided in Section I. We briefly summarize the breakdown of the newly added classes by features as follows. More details about these classes can be found from the spreadsheet.

- **AWT 1.1** Table III shows four classes were added to AWT in 1.1. Three of them, AWTEvent, EventMulticaster, and EventQueue, were added as part of the redesign of the event processing logic in 1.1. The fourth class, MenuShortcut, would be a helper class.
- **AWT 1.3** The 45 new classes added to AWT in 1.3 include 27 for implementing accessibility, 12 for printing, 2 for new events, 2 new exceptions, and 2 helper classes.
- **AWT 1.4** The 25 new classes for AWT in 1.4 include 10 for image buffer management, 2 new events, 6 for focus management (recall that focus management was redesigned in 1.4), and 7 helper classes.
- **Swing 1.3** 8 classes were added for the action map and input map features, 9 for rendering text in text components, and 8 for plaf support.
- **Swing 1.4** 22 classes were added to offer two new widgets, JSpinner and JTextField, and 3 for the new Spring layout, and 3 for focus management. The remaining new classes were added for miscellaneous secondary purposes.

Table III: Evolution of the numbers of classes for each package in AWT and Swing. **new** represents the number of new classes added in each version, whereas **delta** represents the number of classes added to a package that has already existed.

Packages	1.0	1.1	1.2	1.3	1.4	1.5	1.6	Total
AWT								
java.awt	85	4	1	41	14	2	8	155
image	48	0	0	0	4	0	0	52
event	-	37	2	2	2	0	0	43
datatransfer	-	9	1	1	0	1	0	12
dnd	-	-	22	0	2	0	0	24
geom	-	-	33	0	0	0	3	36
color	-	-	7	0	0	0	0	7
font	-	-	16	1	1	0	1	19
renderable	-	-	7	0	0	0	0	7
print	-	-	10	0	0	0	0	10
im	-	-	4	0	0	0	0	4
im.spi	-	-	-	3	0	0	0	3
new	133	50	105	48	25	3	19	372
delta	-	4	4	45	25	3	19	100
Swing								
javax.swing	-	-	188	10	22	2	17	239
event	-	-	48	0	0	0	1	49
plaf	-	-	43	4	1	0	0	48
undo	-	-	9	0	0	0	0	9
border	-	-	10	0	0	0	0	10
tree	-	-	20	0	0	0	0	20
table	-	-	13	0	0	0	2	15
text	-	-	96	9	10	0	1	116
text.rtf	-	-	1	0	0	0	0	1
text.html	-	-	42	0	1	1	0	44
parser	-	-	10	0	0	0	0	10
plaf.multi	-	-	29	0	2	0	0	31
plaf.metal	-	-	62	5	1	2	0	70
plaf.basic	-	-	185	3	5	0	0	193
colorchooser	-	-	4	0	0	0	0	4
plaf.synth	-	-	-	-	-	9	0	9
new	-	-	760	31	42	14	21	868
delta	-	-	-	31	42	5	21	99

- **Swing 1.6** The 21 classes added to Swing in 1.6 include 9 for sorting and filtering the data in a table, 6 for enhancing drag and drop, 5 for the new group layout, and the SwingWorker class for threading.

Only four classes were deprecated: two classes, FocusManager and DefaultFocusManager, from the javax.swing package, in 1.4; javax.swing.text.DefaultTextUI replaced by plaf.basic.BasicTextUI, and java.awt.Event by AWTEvent, in 1.1. All of them have a replacement. Three of them were deprecated during a large redesign (Section III-C4). The deprecation of the fourth class, DefaultTextUI, might have been caused by a lack of coordination between developers as both this class and its replacement class started to exist in the same version, 1.2, which should not have happened.

Overall, we conclude that the architectures for AWT/Swing have been established and maintained stable since 1.2 (that is, at the end of the first three years). This conclusion is based on data shown in Tables II and III, combined with our general knowledge about a GUI framework. Thus, in the remaining subsections, we focus on characterizing how the API methods have been changed at the detailed design level.

B. Deprecated API methods

As mentioned before, we consider deprecated methods interesting because their evolution may give valuable lessons as to what has been done wrong in API design. As shown in Table IV, 114 methods from AWT and 41 from Swing are deprecated, 108 and 27 of which, respectively, have a replacement. In this section, we grouped these deprecated methods into categories of change intent.

1) Conformance to naming conventions (57/114, 10/41): Establishing and following good naming conventions can facilitate future comprehension and maintenance activities by helping developers set the right expectations for an API element. A large proportion of the API methods were deprecated in order to conform to a set of naming conventions. In what follows, we broke them down into subcategories.

More precise names (7/114, 8/41) A precise method name will help convey the exact meaning of the concept and behavior that the method is intended to implement.

For example, the CheckboxGroup class in AWT is used to group together a set of Checkbox buttons. Exactly one check box button in a CheckboxGroup can be in the "on" state at any given time. Pushing any button sets its state to "on" and forces any other button that is in the "on" state into the "off"

Table IV: Deprecated, updated/replaced, and new methods in AWT and Swing. AWT and Swing added 58 and 53 constructors, as well as 86 and 160 methods that override a supertype method, respectively. Because Swing does not specify the version when a method is deprecated or updated, version 1.3 is used, but we have analyzed these cases as well.

Versions	Deprecated	Updated	New
AWT			
1.1	102	97 (3)	239
1.2	5	1	87
1.3	2	2	73
1.4	1	1	220
1.5	4	4	47
1.6	0	0	66
Total	114	108	734 (+58+86)
Swing			
1.3	41	27	251
1.4	-	-	231
1.5	-	-	49
1.6	-	-	121
Total	41	27	652 (+53+160)

state. In 1.0, getCurrent() and setCurrent(Checkbox) were the API methods for getting and setting the active button for a CheckboxGroup. In 1.1, they were deprecated and getSelectedCheckbox() and setSelectedCheckbox(Checkbox) were recommended as replacements. Clearly, the two new names are more precise than the old ones.

Another example is from the JPasswordField class in Swing, where its method getText() was renamed to the more precise name getPassword().

More concise names (8/114, 0/41) A more concise name will help convey the same semantics, but with fewer words.

The first example is the addItem(String) method in the List class of AWT. This method can be used to insert a string into a list. In 1.1, addItem(String) was deprecated in favor of add(String) for better conciseness.

A second example is the isFocusTraversable() method in the Component class, which tests whether a GUI component can receive the keyboard focus. Again in 1.1, isFocusTraversable() was deprecated in favor of isFocusable() for better conciseness.

Getters and setters (39/114, 1/41) To set and retrieve an attribute from an object, the setter and getter APIs should be named as getXXX and setXXX. For this reason, the countItems() method in the List class was replaced with getItemCount(), and the move(int, int) method in Component was replaced with setLocation(int, int).

API methods for retrieving boolean attributes can be named either as isXXX or by using an indicative verb such as contains. For example, the inside(int, int) method in Component tests whether a given pair of coordinates is contained by a component. contains(int, int) was added as a replacement API method for inside.

Spelling errors ((3/114, 1/41)) As shown in Table V, four API methods were deprecated due to spelling errors.

Table V: Spelling errors in AWT/Swing API method names.

classes	old	new
FontMetrics	getMaxDecent()	getMaxDescent()
RenderContext	preConcetenateTransform(AffineTransform) concetenateTransform (AffineTransform)	preConcatenateTransform(AffineTransform) concatenateTransform (AffineTransform)
HTMLEditorKit. InsertHTMLText Action (**Swing**)	insertAtBoundry()	insertAtBoundary()

2) Simplification (19/114, 2/41): Some API methods were deprecated to reduce the volume of the API. For example, Dialog can be made visible by calling show(), invisible by hide(), or by show(boolean). In 1.1, these three methods were marked as deprecated, and replaced with setVisible(boolean). This kind of simplification would improve API usability.

3) Introduction of new concepts and classes (13/114, 2/41): Existing API methods may be deprecated and replaced due to the introduction of new concepts or classes.

For example, to support internationalization, AWT supports the notion of text orientation. The orientation of a Component can be retrieved using getOrientation(ResourceBundle). However, the parameter ResourceBundle contains more information than what is needed for retrieving the text orientation. In 1.4, a new concept of locale is introduced by a class Locale. The old getOrientation(ResourceBundle) was deprecated in favor of getOrientation(Locale).

A second example is related to setting a cursor for a frame. In 1.0, Frame provides the setCursor(int) method for setting its cursor, where the integer parameter specifies a type of a cursor. This would be based on the assumption that there are an enumerable set of cursors that can be identified with an index. In 1.1, the Cursor class was added, and a replacement API method setCursor(Cursor) was added in Component, a supertype of Frame.

The third example is related to layout management. In 1.0, in order for a widget to be a child of a container and managed by a layout manager, a location must be specified for the widget via the first parameter of addLayoutComponent(String, Component) on a layout manager object. In 1.1, the notion of a location was generalized, from being a String to a constraint of any kind. Thus, addLayoutComponent(Component, Object) was added as a replacement, whose second parameter serves as the constraint. The choice of the signature for the new API method is likely motivated by the desire to maintain backward compatibility with the deprecated one.

4) Reducing coupling (2/114, 3/41): Inappropriate coupling can make comprehension and maintenance unnecessarily difficult. A developer would be surprised to find that things are not located where they are supposed to be.

For example, in 1.0, to obtain the metrics for a font,

one must invoke getFontMetrics(Font) on Toolkit. Toolkit is clearly a less ideal home for this method than Font. In 1.1, Font.getLineMetrics was added as a replacement.

Another example is about retrieving the bounds for the viewport of a scroll pane. The method for this task used to be located in ScrollPaneLayout (getViewportBorderBounds(JScrollPane)). In 1.3 (or later), it was relocated to JScrollPane (getViewportBorderBounds()), a better home.

5) Encapsulation (3/114, 0/41): An API method may be deprecated because it is deemed to expose too much internal information that would better be encapsulated. One example is Component.getPeer(). This method returns the native, system-specific GUI object (also known as the peer) that actually renders the AWT object on a native platform. This exposure is unnecessary and may accidentally break the system if someone tries to manipulate the returned peer object. In 1.1, it was replaced by boolean isDisplayable().

6) Conformance to supertype contracts (2/114, 3/41): The intention for a method in a subclass to override a supertype method is expressed by having the two methods share the identical signature. However, there is not an automated way to enforce this intention. Consequently, a method in a subclass may be wrongly named despite the intention that it should override a supertype method.

For example, the getComponentAtIndex(int) in JMenuBar returns the menu at a given index from a menu bar object. It turns out that Container, a supertype of JMenuBar, has already declared a method named getComponent(int index) for the same purpose. As a result, getComponentAtIndex(int) was deprecated and replaced by the latter. Note that although getComponentAt would have been a better name for the same purpose, Container has already used it for retrieving a component that is located at a given coordinate.

Another example is from JTable. It has a method sizeColumnsToFit(boolean) for calculating the size of the table. But there is already a method doLayout() from a supertype for doing this. Thus, sizeColumnsToFit is deprecated.

7) Deprecation without replacements (6/114, 14/41): It would be especially interesting to know how and why API methods are truly deprecated without replacements since they are likely to be caused by true "design mistakes".

Of the six from AWT, two were protected, subclass APIs that were documented as "never used"; one was an equals() method that the designer considered wrong, probably because it does not conform to the contract for equals(); one was a setter that was considered unnecessary because it is sufficient to set the value via a constructor; and two were convenient methods whose functionalities can be achieved by other methods.

Of the 14 deprecated APIs from Swing, eight were due to design cleanup because the designers identified a different way to achieving the same goal for each of the eight; three were considered redundant because other APIs can be used to achieve the same functionalities; and three were protected, subclass APIs that were documented as "never used".

Table VI: Distributions of 1,386 new AWT/Swing API methods by features.

features	#APIs
new attributes	708
listeners	179
event processing	124
focus	57
graphics	80
text components	38
plaf	65
drag and drop	9
buffer strategy	11
print	11
accessibility	11
actions	24
other	69

8) Redesign of existing features (12/114, 7/41): Some API methods were marked as deprecated as the result of a larger restructuring and redesign effort. Twelve API methods in AWT were deprecated due to the redesign of the event processing subsystem in 1.1. Seven API methods in Swing were deprecated due to the redesign of the new focus management system. See Section III-C4 for details of these two major feature redesigns.

C. New API methods

As shown in Table IV, 1,386 new API methods, 734 for AWT and 652 for Swing, were added. Our main methodology for analyzing the intents behind these new API methods is to raise the level of abstraction. Rather than separately as individual API methods, we study them at the design feature level. When an individual API is also a feature, this methodology does not make much difference. However, when a feature is implemented by more than one API method, the addition of a new API method to the feature can be more appropriately and meaningfully interpreted by considering the intent behind the change to the feature.

To gain insights as to what these methods are for, we first exhaustively grouped them by design features. Table VI summarizes the result of this categorization. (See the online spreadsheet for further details.) The 'new attributes' category represents methods of getters and setters and isXXX for boolean attributes. The 'listeners' category represents those methods for managing event listeners (addXXXListener, removeXXXListener, getXXXListeners, and getListeners). All of the other features are standard to AWT/Swing. The thirteen categories in the table show that the conceptual volume of the new API methods is much smaller than their sheer number. In fact, just the first four categories already contain 1,068 of the 1,386 methods. Finally, although the 69 API methods contained in the 'other' category can be further categorized by adding new features, doing so would make the list of features in Table IV unnecessarily long. In fact, the majority of them are new features specific to individual

widgets such as JTree and JTable for Swing, and Component and Container for AWT.

After categorizing new APIs by design features, we further explored the intent behind adding a new method. Although majority of these new methods were added for introducing new capabilities, some of them were added for different reasons or intent. In the remaining of this section, we present the intents behind adding these new API methods.

1) New features: The majority of the new API methods were added to create new capabilities. In particular, most of the API methods in the 'new attributes' category in Table VI introduce a new feature. In the following, two more examples from events are given for further illustration.

In 1.6, in addition to the coordinates in the component where the mouse is located, MouseEvent provides the location of the mouse on the screen (getLocationOnScreen(), getXOnScreen(), and getYOnScreen()).

In 1.4, FocusEvent provides the component that has just lost its focus to the current component (getOppositeComponent()). WindowEvent returns the other Window involved in this focus or activation change (getOppositeWindow()).

2) Functional enhancement of existing features: When an existing feature is not powerful enough, it will be impossible or difficult to achieve some functionality. New API methods may be added to enhance the existing feature and help achieve the functionality. Many of these features are ADTs.

For example, JTabbedPane is a GUI widget that manages multiple panels/components using multiple tabs. At any instant of time, only one tab can be selected with the corresponding component shown consequently. For each tab, a client may supply a component, a title, and an icon. JTabbedPane provides API methods for setting and getting the content and the index for each tab. In 1.6, support is added so that one can customize the visual display of the tab itself (setTabComponentAt(int, java.awt.Component), getTabComponentAt(int), and indexOfTabComponent(java.awt.Component)).

As a second example, getListeners and getXXXListeners were added in 1.3 and 1.4, respectively, to various Swing components to provide access to the registered listeners. Such access can be necessary in situations where the clients have not kept track of the registered listeners but nevertheless want to manipulate them.

The third example is the addition of getLayoutComponent to BorderLayout (as well as getConstraints). This method allows for retrieving the component that is added to a certain position in a container that uses BorderLayout. This is useful when no references to the component are kept, as the following snippet illustrates:

```
BorderLayout layout = panel.getLayout();
panel.remove(layout.getLayoutComponent(BorderLayout.CENTER));
```

3) Non-functional enhancement of existing features (usability and performance): New API methods may also be added for reasons such as usability and performance. We present two examples for each of usability and performance.

As a first example for usability, a scrollbar controls the position of a smaller view over a larger background content. At any given time, the view shows only a portion of the background content. The position is represented by an attribute called value. The range of the background content is controlled by two attributes minimum and maximum. The size of the view is represented by the attribute visible amount. In 1.0, AWT provides only setValues(int value, int visibleAmount, int minimum, int maximum) for setting these attributes. Thus, in 1.0, to change the maximum value for a scrollbar, the following code snippet is needed:

```
int oldValue = aScrollbar.getValue();
int oldVisibleAmount = aScrollbar.getVisibleAmount();
int oldMinimum = aScrollbar.getMinimum();
aScrollbar.setValues(oldValue, oldVisibleAmount,
oldMinimum, maximum);
```

In 1.1, two more methods, setMinimum and setMaximum, were added. As a result, one can achieve the same functionality with only one line of code:

```
aScrollbar.setMaximum(maximum);
```

Interestingly, in 1.0, there does exist another method, setValue(int value), for setting the value. This shows that the AWT developers probably have already thought about the convenience of setting the value, but somehow they did not apply it to the other three attributes of the scroll bar.

The second example for usability is the addition of the three setters, setPreferredSize, setMaximumSize, and setMinimumSize to Component in 1.5, which significantly shorten the amount of client code needing to be written for setting these three properties for a widget. The corresponding getters already exist since 1.0. Before 1.5, the only way to set these properties is through overriding the getters in a subclass. This is both cumbersome and unintuitive. A quick search in Google Code Search using these getters as keywords revealed several thousands of anonymous subclasses written for this purpose, which could be replaced by simply calling the setters. A similar case is the addition of setComponentPopupMenu (JComponent in 1.5), which also greatly simplifies the code for registering a popup menu on a widget.

The first example for performance is the addition of getX(), getY(), getWidth(), and getHeight() to Component in 1.2. These API methods may appear redundant since the x and y coordinates as well as the height and width of a widget can also be obtained from the return value of other API method calls such as getBounds() and getLocation(). The introduction of these new methods is motivated by consideration of performance because the latter two create a heap object every time they are called whereas the new methods avoid creating such objects.

The second example for performance is the class Segment in javax.swing.text, which represents a fragment of text inside a character array. For efficiency (avoiding the overhead of

copying around characters), the array that backs up the segment is made directly accessible (as a public field). However, the design intent is that the array should be treated as immutable. To support this, in 1.3, several common methods for accessing a sequence were added, for example, setIndex, current, first, last, previous, and next. However, there was not a method for accessing a character at an arbitrary index, or a method for obtaining a subsequence from a segment. In 1.6, getCharAt() and subSequence() were added to support these, respectively.

4) Redesign of existing features: There are two redesign of existing features.

Case 1: The event processing model In AWT/Swing, once generated, GUI events are first pushed into event queues. The Java runtime dispatches these events to the right GUI components one by one. In case that the component is a container, it may have to further dispatch the event to the right child component. Otherwise, the component handles the event. In 1.1 and subsequent versions, this event processing logic has evolved substantially.

The first major change is replacing the inheritance based event handling model with the listener based model. In 1.0, all events are represented by a single Event class, where events are distinguished by an integer id. Each component knows the set of events that it can handle. Component handles each event with a dedicated event handling method, for example, mouseDown(Event, x, y) for the mouse down event, and keyDown(Event, key) for the key down event. Note that x, y, and key are redundant because the same information can be found in Event. Two methods in Component, deliverEvent(Event) and postEvent(Event), contain the switch logic to dispatch an event to a proper handler. This design for event handling is called "the inheritance-based model" because to handle an event differently, a subclass must be created to override the event handling method.

In 1.1, the listener based model is introduced. The Event class is deprecated and replaced with an event type hierarchy with a new root type AWTEvent. For each type of event, a listener is defined for handling that event. More importantly, each Component can have multiple listeners of the same kind. To manage the listeners, a pair of methods, addXXXListener and removeXXXListener, are added to Component, for each event "XXX". Furthermore, deliverEvent(Event) and postEvent(Event) are replaced by processEvent(AWTEvent). For each kind of event "XXX", a method processXXXEvent(XXXEvent) is added to Component, for example, processKeyEvent(KeyEvent) for KeyEvent and processMouseWheelEvent(MouseWheelEvent) for MouseWheelEvent. On receiving an XXX event, processXXXEvent(XXXEvent) invokes all the registered "XXX" listeners.

The further evolution of event handling involves mainly supporting new events as well as adding additional accessor methods. For example, mouse wheel events are supported in 1.4, which results in the addition of MouseWheelEvent and four more methods in Component (processMouseWheelEvent, addMouseWheelListener(), removeMouseWheelListener(), and getMouseWheelListeners()). Furthermore, getListeners() were added in 1.3, and getXXXListeners() in 1.4, to provide access to these listeners. Such access can be necessary in situations where the clients have not kept track of the registered listeners but nevertheless want to manipulate them.

In sum, the redesign of the event handling in AWT enables the listeners model. This redesign has facilitated the growth of the event handling in subsequent versions in that the addition of new APIs reflects the original roles they are designed for. This is consistent with the observation of Aversano et al. [14] made on design patterns.

Case 2: The focus subsystem Focus, also known as the keyboard focus, is the mechanism that determines which of the components in a window will receive keyboard input events. Most of the components, even those primarily operated with the mouse, like buttons, can be operated with the keyboard. A focus manager looks for special keystrokes that change the focus (usually the Tab and Shift-Tab keys), and decides which component will get the focus next.

The focus system in 1.0 allowed components to set the focus to themselves with requestFocus, to change the focus to the next component with nextFocus, and to get notification of other focus changes with gotFocus and lostFocus. JDK 1.1 deprecated and replaced nextFocus with transferFocus, which provides almost the same functionality. 1.1 also deprecated and replaced gotFocus and lostFocus, with addFocusListener, to handle focus change notification.

The focus subsystem before 1.4 was inadequate and suffered from the several major problems. First, the focus traversal order is fixed, and there is no way for applications to change it. Second, it is impossible to query for the currently focused component. In fact, such information was not even maintained by the code, making it an insufficient architecture. Third, the APIs for FocusEvent and WindowEvent were not sufficient because they did not provide a way for determining the "opposite" component involved in the focus or activation change. For example, when a component received a Focus_Gain event, it had no way to know which component had just lost focus. These problems were mainly caused by the incompatibilities between the native focus systems. A lightweight focus architecture is called for to hide these incompatibilities.

The 1.4 focus system for AWT is a complete redesign. The primary changes of the system were the construction of a new centralized KeyboardFocusManager class and a lightweight focus architecture that hides the diverse incompatibilities between the native focus systems. In order to reduce the platform inconsistencies, the focus-related, platform-dependent code is minimized and replaced by fully pluggable and extensible public APIs in the AWT. Unfortunately, the application developers have to sacrifice the

backward compatibility with the existing implementation. The complete list of API methods and classes for focus can be found in [15].

D. Overriding methods

An API method in a subclass overrides a supertype method for two main reasons. It may be added simultaneously with the supertype as part of a new design feature. For example, in 1.6, Swing adds new support for base line for GUI components. As a result, two new methods, getBaseline(int, int) and getBaselineResizeBehavior(), are added to JComponent. Two same named methods are also added to ComponentUI. 15 subclasses of ComponentUI such as BasicLabelUI and BasicPaneUI, override these two methods.

An overriding method may also be added after the supertype has existed, usually for better performance. For example, in 1.6, JFrame, JDialog, JWindow, and JApplet override getGraphics() and repaint() of the supertype Component. These two methods are defined in Component since 1.0. The four top windows override them to improve the performance of graphics rendering.

Another example for better performance is related to DefaultTreeCellRenderer and DefaultTableCellRenderer. According to their JavaDoc, in 1.5, these two classes override five superclass methods, invalidate, validate, revalidate, repaint, and firePropertyChange, solely to improve performance. If not overridden, these frequently called methods would execute code paths that are unnecessary for these two default renderers.

IV. Implications of results

At the global level, we would like to consider the evolution of the AWT/Swing API as being driven by a stable architectural design. The GUI component type hierarchy, the architecture for Swing's pluggable look and feel, and the event processing are established early, since 1.2. The event processing subsystem was re-architected early in 1.1. The new, listener-based design has accommodated the future growth of the event systems smoothly without friction; as our data show, support for new kinds of events and listeners have been added systematically. There were very few classes deprecated. Overall, the number of changed elements is relatively small compared to the size of the whole API, and the majority of them happened in 1.1 (Table IV).

Although a good initial design means a more stable API, local design weaknesses can still cause major overhauls of the API. A large part of API evolution is minor correctives, for example, fixing naming problems (spelling errors, weak names) or issues related to violations of general design principles such as coupling and encapsulation. Establishing a coding standard earlier should have helped avoid many of these API changes in the first place. In fact, the API team appeared to have done so in 1.1. This should also explain why the Swing API contains much fewer such changes (Table IV). Paying more attention to the general design principles would also help. Though much of this is rather intuitive, our empirical data backed the intuition.

Some of our results may be less intuitive. One, API methods may be replaced or added to improve usability or performance (less API methods to learn, or shorter client codes. See the instances presented under *simplification* in Section III-B2 and *non-functional enhancement* in Section III-C3). Two, we show that the API methods for a feature may be added incrementally over time (Section III-C2). It appears that at least some of those instances could have been avoided if, during the initial design, the API designers had strived to be complete, for example, by following simple rules such as always providing both setters and getters. For application developers, knowing these categories would allow them to comprehend API changes more effectively. This is especially important since modern APIs tend to be voluminous. Developers need such assistance to orient themselves in this large, evolving information space.

In general, it is desirable to avoid feature redesign, or at least reduce its time window to minimize the impact that such redesign can have on client codes. Comparing with the event processing logic, the redesign of the focus subsystem in 1.4 appeared to be a little late. Because some high quality GUI applications require the fine control over focus behavior, this delay might have hindered the adoption of the framework. It also created major difficulties for some application developers as they had to create workarounds. However, it is not exactly clear as to what have caused this delay. To reach a reliable conclusion, we would need to know the state of the art of domain understanding at that time and the main sources of inspiration of Swing/AWT. Only then can we speculate on what sort of focus system could have been expected. Perhaps the timeline of revising the focus system was more or less generally known within the API design team. In any event, this case illustrates the negative impact that API redesign may have on the clients and the importance of clean up-front design.

V. Threats to Validity

As a single case study, our study reflects only the evolution of AWT/Swing. There are good reasons to believe that other APIs may follow a different trajectory of evolution. So we need to be careful not to overgeneralize our results.

We rely on the information that AWT/Swing developers put in the JavaDoc. So our results can only be as good as the quality of the JavaDoc. For example, we have recently discovered from reading the JavaDoc for the DefaultTableModel class, that in 1.3, setNumRows was replaced by setRowCount. But the replacement was not documented in the standard way and, thus, missed by our tool. Despite a few of such issues, overall we believe that the AWT/Swing developers have done a high-quality job in keeping track of evolution history of this framework over a decade.

To keep the study simple, we have focused mainly on classes and methods and ignored changes to fields and interfaces. So our results do not apply to the evolution of fields and interfaces.

VI. Related Work

Dig and Johnson manually investigated API changes using the change logs and release notes to study the types of library-side updates that break compatibility with existing client code, and discovered that 80% of such changes are refactorings [10]. Xing and Stroulia used UMLDiff to study API evolution in several systems, and found that about 70% of structural changes are refactorings [11]. Our study is broadly targeted to characterize the overall evolution of one API rather than the role of refactoring in API evolution.

When API methods are changed such that a client program is broken, it can be tedious to bring the client program back to work with the updated API. A series of tools have been proposed to automate this API adaptation process, under various assumptions and with different degrees of support [4], [5], [6], [7], [8], [9]. An empirical study like ours could help reveal new challenges and opportunities in API evolution for the tool designers.

Aversano et al. empirically study the evolution of design patterns in open-source systems and find that the most frequent changes to design patterns are those that align well with the roles of the patterns that are intended by the application [14].

Lawrie, Feild, and Binkley [16] and Deissenboeck and Pizka [17] investigate the concise and consistent naming of identifiers and the automated enforcement of syntactic concise and consistent naming. In this paper, we use the term "precise" to express their notion of conciseness.

Shi et al. study the evolution of Java API documentation, showing the kinds of changes and discussing their implications [18]. Schreck et al. compare the evolution of the quality of JDK documentation and Eclipse documentation [19].

VII. Conclusions

We report a case study of the evolution of the AWT/Swing APIs based on information extracted from the official Java API documentation. We conclude that a stable architecture has played an important role in supporting the smooth evolution of the AWT/Swing API. We also identify a set of API design mistakes and enhancements that can serve as lessons learned for both API designers and application developers.

One future work could be to empirically identify the factors that drive the addition and enhancement of features to an API and to investigate to what extent they can be anticipated by the API designers up front. It would also be interesting to contrast this case of architectural driven evolution with APIs that are evolved using a more agile methodology, such as those in Eclipse [20].

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Commun. ACM*, vol. 39, pp. 104–116, October 1996.

[2] R. W. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 495–510, 2005.

[3] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 13:1–13:31, June 2008.

[4] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *ICSM*, 1996, pp. 359–368.

[5] J. H. Perkins, "Automatically generating refactorings to support api evolution," in *PASTE*, 2005, pp. 111–114.

[6] Z. Xing and E. Stroulia, "API-evolution support with Diff-CatchUp," *IEEE Trans. Software Eng.*, vol. 33, no. 12, pp. 818–836, 2007.

[7] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE*, 2008, pp. 481–490.

[8] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *OOPSLA*, 2010, pp. 302–321.

[9] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in *ICSE*, 2010, pp. 325–334.

[10] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *J. Softw. Maint. Evol.*, vol. 18, pp. 83–107, March 2006.

[11] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," in *ICSM*, 2006, pp. 458–468.

[12] D. Notkin. Position paper: Anticipating change in general-purpose software systems. NSF Workshop on Science of Design: Software-Intensive Systems, November 2-4, 2003, Airlie Center, Virginia. Last verified: April 27, 2011. [Online]. Available: http://www.cs.virginia.edu/~sullivan/sdsis/Program/David%20Notkin.pdf

[13] D. Kramer. How to write doc comments for the javadoc tool. Last verified: April 24, 2011. [Online]. Available: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

[14] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *ESEC-FSE*, 2007, pp. 385–394.

[15] X. Yao, "How do APIs evolve? A study of Java AWT and Swing," Master's thesis, Clarkson University, Potsdam, New York, USA, April 2010.

[16] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *SCAM*, 2006, pp. 139–148.

[17] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, pp. 261–282, September 2006.

[18] L. Shi, H. Zhong, T. Xie, and M. Li, "An Empirical Study on Evolution of API Documentation," in *FASE*, 2011, pp. 416–431.

[19] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in *IWPSE*, 2007, pp. 4–10.

[20] D. Hou, "Studying the evolution of the Eclipse Java editor," in *ETX*, 2007, pp. 65–69.