What Can Programmer Questions Tell Us About Frameworks?

Daqing Hou
9723-88 Ave
Avra Software Lab Inc.
Edmonton, Alberta, Canada
daqing@ieee.org

Kenny Wong and H. James Hoover
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{kenw, hoover}@cs.ualberta.ca

Abstract

In order to make frameworks easier to use we need to better understand the difficulties that programmers have with them. The questions that programmers ask give clues to the quality of design, documentation, and programmer practice.

We describe the method and results of a study on the Java Swing framework. We collected and analyzed a sample of 300 newsgroup questions asked about two Swing components (JButton and JTree), and classified the questions according to the design features of the components. This process revealed key insights that can improve a framework's design, its tutorials, and programmer practice.

1. Introduction

Object-oriented frameworks leverage proven software designs and implementations to reduce development costs and improve software quality. In contrast to earlier reuse techniques based on function and class libraries, frameworks are targeted for particular application domains (such as cellular communications, user interfaces, real-time avionics, and typical business applications). When combined with components, frameworks provide practical large-scale reuse [5].

Object-oriented frameworks are often hard to learn and use. Popular frameworks such as Microsoft Foundation Classes (MFC) and Sun Java Swing have a steady stream of postings to developer newsgroups asking for help. This is such a serious problem that Microsoft itself has set up a team to evaluate the usability of MFC [12].

There can be several reasons for this unfortunate state of affairs. First, software is complex, and so is a framework. It would be naive to expect that learning a framework be an easy task. Second, even though frameworks are usually high-quality, their design may still contain flaws and documentation always needs improvement. Third, most programmers tend to spend as little time as possible to get their work done. This strategy may work occasionally but fail miserably when framework interfaces are complex. Understanding framework usage is both a practical and pressing problem.

Understanding frameworks is different from other program understanding activities. Frameworks are more abstract than ordinary software. Programmers focus mostly on framework interfaces and the interaction between frameworks and applications. Although framework source code is often made available as complementary documentation, it is so large that programmers rely mostly on interface documentation and tutorials for learning a framework. Frequently, partial understanding can be sufficient for the given tasks.

The existing literature has little prior work on framework usage [12]. In the past, research in program understanding has focused mostly on software maintenance and evolution but not reuse [9]. Past studies looked at the construction of mental models in program understanding [9], program understanding behavior during maintenance [10], understanding components [1], and defect repair strategies and behavior [4, 8]. Most of these studies were based on source code. Strategies for using frameworks are absent in the literature [9]. Therefore, a study that explores issues faced by programmers in framework-based development is a useful contribution.

This study explores how the information implicit in programmer questions can suggest improvements to design, documentation, and programmer practice.

We focused on the Java Swing framework because it is mature, documented, and widely used. This study manually collected and classified newsgroup questions from the Swing Forum. Instead of studying over 30 Swing components, we picked two: JButton and JTree. JButton is representative of a simple component which reveals issues related to deep inheritance. JTree represents a typical complex component with rich composition. Since the design of Swing is heavily inheritance-based, these two components

should be enough to cover a variety of usage questions.

We collected and analyzed a sample of 300 questions. Each question was categorized by the design features of the corresponding component. This classification highlighted the most problematic design features. We re-examined these features and reflected on why they were difficult to understand.

1.1. Method

We followed the following process in this study:

1. Identify design features for the subject of study;
2. Collect relevant questions;
3. Analyze and classify each question by the feature that is the root cause of the problem.
4. Reflect on why certain design features are difficult.

1.2. Paper organization

The remainder of this paper is organized as follows. Section 2 introduces parameters of our study, in particular, the design features of JButton and JTree; the Swing Forum; and how we collected, analyzed, and classified questions. Section 3 presents the classification results. In section 4, guided by the classification results, we reflect on issues related to design, documentation, and people. Finally, section 5 concludes the paper and discusses future work.

2. Study parameters

This section introduces the parameters to our study. Subsections 2.1 and 2.2 provide a brief description of the design of JButton and JTree, respectively. Subsection 2.3 introduces our data source (Swing Forum) and the data collection process. Subsection 2.4 addresses issues related to analysis and classification.

To ensure the quality of such a study, it is critical to fully understand the relevant details about the subject framework. For that purpose, we consulted not only Sun's on-line Javadoc API documentation, but also a few books on Swing such as the Swing tutorial [14]. In particular, we compared the identified features with the book to make sure that our features were both complete and consistent with the original design.

2.1. JButton design

Figure 1 depicts JButton's high-level design. To understand which inherited features tended to generate the most questions, we labeled each of the three superclasses with features that the superclass implements. Documentation for these features can be found in [14].

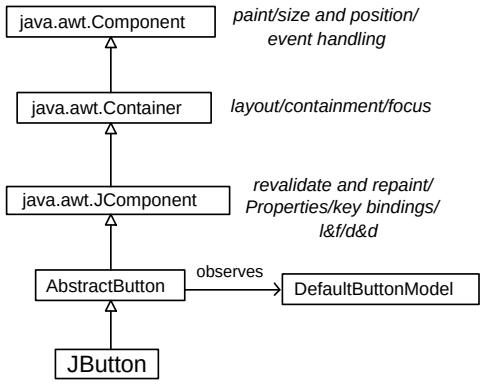


Figure 1. JButton design

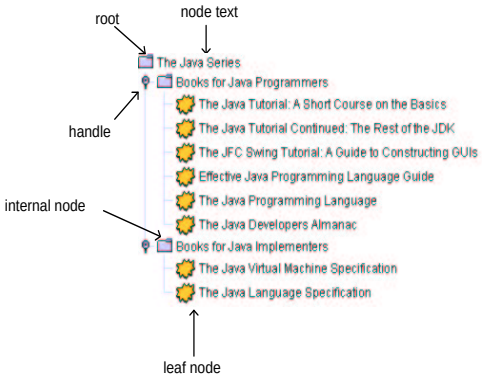


Figure 2. JTree appearance

Similarly to all other Swing components, JButton inherits 336 fields and methods from the three superclasses (Table 1). AbstractButton consists of 24 fields and 80 methods, DefaultButtonModel has 11 fields and 31 methods, and JButton has 14 methods. This brings the total fields and methods of the JButton component to 160.

2.2. JTree design

Figure 2 presents a sample tree picture and Figure 3 depicts a high-level conceptual view of JTree's design. To help build connection with features, we marked the diagram with feature numbers. At the core of this design is the observation relationship between a JTree component and its data model. A tree model is made up of tree nodes that wrap ordinary user objects. Given a tree model, a JTree object draws its tree nodes using a tree cell renderer, which can be customized to produce required visual effects. Similarly, a JTree edits its tree nodes using a tree cell editor. As can be seen from the diagram, (to avoid visual clutter, tree cell editors' dependences are omitted) both renderers and editors know tree nodes and the current visual state of the tree. This is necessary as both may need to customize things such as

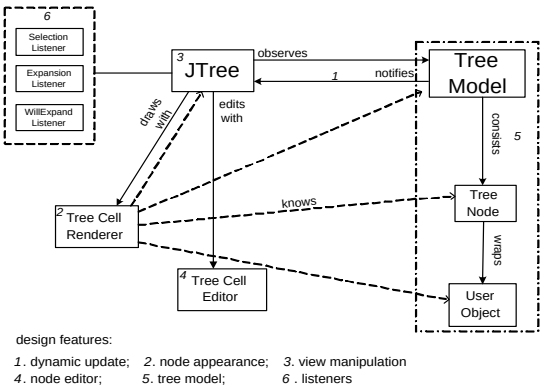


Figure 3. Conceptual design of JTree

Classes	Fields+methods
java.awt.Component	5+138
java.awt.Container	0+51
javax.swing.JComponent	8+134
Total	13+323 (336)

Table 1. Common Swing superclasses

Classes and interfaces	Fields+methods
JTree	31+135
TreeModel	0+8
DefaultTreeModel	3+30
TreeNode	0+7
MutableTreeNode	0+6
DefaultMutableTreeNode	5+54
TreePath	0+15
TreeCellRenderer	0+1
DefaultTreeCellRenderer	10+39
TreeCellEditor	0+8
DefaultTreeCellEditor	13+26
TreeModelListener	0+4
TreeModelEvent	3+9
TreeSelectionListener	0+1
TreeSelectionEvent	4+10
TreeSelectionModel	3+27
DefaultTreeSelectionModel	10+40
TreeExpansionListener	0+2
TreeWillExpandListener	0+2
TreeExpansionEvent	1+2
ExpandVetoException	1+2
Total	84+428 (512)

Table 2. JTree related classes and interfaces

their visual appearance based on these information. For example, one may determine the icon for a leaf node according to the current state of the tree. In addition to rendering and editing, a visual tree component also supports user interaction such as expanding or collapsing an internal node, and view manipulation such as setting the height of rows.

Similarly to JButton, JTree inherits a total of 13 fields and 323 methods (Table 1) from java.awt.Component, java.awt.Container, and javax.swing.JComponent. The JTree class itself consists of 31 fields and 135 methods, and its auxiliary classes and interfaces have 346 fields and methods. See Table 2 for a detailed breakdown of these numbers. Using these numbers as indicators of component complexity, we can see that JTree (512 fields and methods) is much more complicated than JButton (160).

Figure 2.2 shows JTree's class diagram. Given a tree model, a JTree object can visually render a picture for the model. The JTree class can not only render a tree view but also provide interaction control and view manipulation. The data model is characterized by the TreeModel interface. The implementation of JTree class depends on this interface rather than its default implementation class DefaultTreeModel.

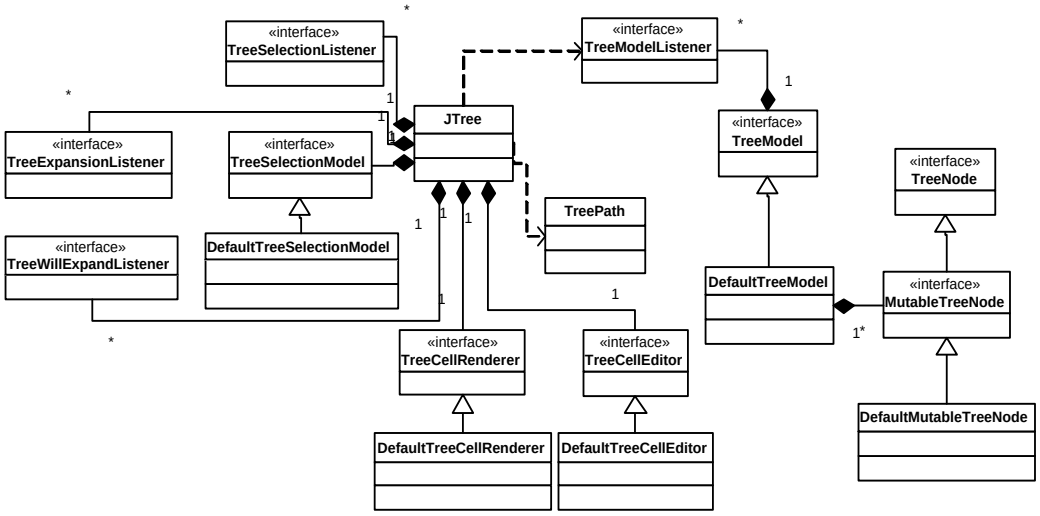
The JTree component involves an extensive set of classes and interfaces. The TreeModel interface defines a protocol for tree data structures. The TreeNode interface defines a protocol for tree nodes. The class TreePath identifies a tree node in the view by the sequence of nodes on the path from the root to the node.

TreeCellRenderer specifies an interface for a renderer object that can actually draw a node for the tree. TreeCellEditor is an interface for an editor that can be used to edit a node interactively. Each JTree must have one TreeCellRenderer and optionally, a TreeCellEditor. Swing provides default implementations for both interfaces, namely, DefaultTreeCellRenderer and DefaultTreeCellEditor.

The TreeModelListener interface specifies event handlers for TreeModelEvent events that a tree model generates. The JTree class can generate two types of events for expanding or collapsing a node; and selecting a node. Classes TreeExpansionEvent and TreeSelectionEvent represent the two events. Finally, A JTree can accept three types of listeners, namely, TreeWillExpandListener, TreeExpansionListener, and TreeSelectionListener.

2.3. Data source and data collection

We gathered questions from Sun's Swing Forum. Sun supports about 50 Web-based Java Forums on the Internet (http://forum.java.sun.com), covering topics from Java programming, through enterprise and distributed computing, to GUI building. The Swing Forum is one of the four most popular. The other three are Java Programming, JSP and JSTL, and New to Java Technology. From July 2, 1997 to



January 19, 2005, there are 210,365 messages in 64,424 topics in the active and archived Swing Forum. This forum is very active. An inspection did not find any single date without postings. There were postings even on Christmas and New Year's day.

The number of messages in the forum was overwhelming for a manual analysis. We narrowed down the scope of messages to those containing the keywords JButton or JTree. We then excluded irrelevant ones. For example, a message may mention JButton incidentally while describing something else. For JTree we also left out illegible messages. A message was deemed illegible if and only if there were no answers to it and despite our best effort, we still could not understand it. We did not filter out messages describing problems of the same nature unless they were duplicated postings by the same author (such duplications were rare).

2.4. Data analysis and classification

To ensure the quality of the analysis, the first author and a colleague first independently analyzed the JTree questions. The discrepancy between their results were then identified and resolved one by one. The analysis for JButton questions was done by the first author only, partly because JButton is much simpler than JTree and partly because after the JTree analysis, he was experienced enough to analyze JButton independently.

Depending on the purpose of a study, one can classify these questions based on either the nature of the problems to be solved or on the solutions. Problem-oriented classification focuses on 'what' programmers want to do with a framework while solution-oriented classification focuses on 'why' programmers' solutions do not work. Since our goal was to understand the interaction issue in using the framework interface, which is tightly related to framework design, we chose a solution-oriented approach. Had we decided to study the functional scope of the framework, then the problem-oriented approach would be more appropriate.

We first filtered out domain-specific questions. For the remaining questions, we learned many solutions from others' replies. This was especially the case at the beginning phase. However, not all questions had obvious answers. We then had to do some research on our own. We made sure that such questions were investigated thoroughly. We found that writing sample programs was an effective way of dealing with such problems. As we spent more time on the framework, our knowledge was incrementally built up. Eventually we were able to solve new problems all on our own. A positive sign of this is that we could identify some widespread but incorrect solutions to some problems. This ability is critical to our classification as the true problem underlying a question can lie deeper than what the authors described on the surface. We dug out the root cause and classified the question accordingly.

3. Results

We collected and analyzed a sample of 110 JButton questions and 190 JTree questions. These questions were then classified by the design features of JButton and JTree. The classification results are summarized in Tables 3 and 4. Note that the left two columns of the tables are the design features that were used. In this section, we introduce the design features and provide selected questions to illustrate the difficulties. For presentation purpose, we have corrected the spelling errors in the original messages. Full details of the collected messages and the classification can be found in [7].

3.1. JButton questions

The 110 JButton questions were broken down into 4 categories: inherited features (37), JButton-specific (19) , interaction with other components (such as JScrollPane, 7), and misc (47).

From the statistics, we can see that for a simple com-

Design features	Sub-features	Questions
inherited features		26
	revalidate and repaint(10)	
	paint (6)	
	focus(3)	
	event handling thread	
	component hierarchy	
	position	
	keyboard	
button-specific		19
layout		11
component interaction		7
misc		47
	design discussion (8)	
	debugging (11)	
	platform bugs (6)	
	basic questions (13)	
	illegible messages (9)	
Total		110

Table 3. JButton questions

ponent like JButton, inherited features were responsible for more questions than component specific features. Another finding is that the forum is a good venue to ask questions, find bugs, and discuss design problems.

Inherited features. Here is an example of a question related to inherited features.

> I am trying to display an image over a JButton inside a JDialog. [...] But the button is invisible. I am sure that it is drawn on the ContentPane since when i click partially/half at the position of the JButton I could see the JButton with the image. Actually I am resizing the JDialog since I have two buttons, one to minimize and the other to maximize the size of the JDailog. All the buttons are in place but not visible neither at the first place nor when the JDialog is resized.

The problem was that the author incorrectly overrode the `paint` method rather than the `paintComponent` in the dialog class.

Button-specific questions. Here is an example of a button-specific question.

> I created a JButton and added an ActionListener. I can activate the button with a mouse left-click but not with a right-click. [...] I added a MouseListener to the button. I got mouse events for both left and right clicks (in mousePressed()), but only the left-click activated the JButton.

The author did not know the existence of a `doClick` method that can be used to activate a button programmatically.

Layout. These questions were related to layout management [14]. JButton happened to be the components mentioned by the authors.

Component interaction. These questions were about JButton interacting with other components. For example, some programmers added buttons directly to top-level containers rather than their content panes; some did not know that in order to obtain a scrolling behavior, a JTree must be put into a JScrollPane.

Misc. The misc category is further divided into design discussion, debugging, basics, platform bugs, and illegible messages. Although these topics were not related to JButton's design features, they shed light on some relevant people issues, which we shall discuss in the next section.

Here is an example of people discussing design issues in the forum. This posting was particularly interesting, followed by 16 replies.

> So, let's say that I do this:
> ```
> JButton b = new JButton("Button");
> final JLabel label = new JLabel("Waiting");
> b.addActionListener( new ActionLis-
> tener(){
> public void actionPerformed(ActionEvent
> e){
> label.setText("Okay");
> }});
> ```
>
> Now let's say that the frame containing the JLabel is destroyed. The JLabel itself will remain in memory until the JButton is destroyed. I have an application in which the equivalent of the JButton is never changed but the equivalent of the JLabel is; therefore, the equivalent of the JButton receives a lot of add...Listener calls. Additionally, the component in my application which corresponds to the JLabel is very large. Does anyone have any recommendations for a clean and graceful way for my component to deregister all of its anonymous listeners without having to keep a registry of who registered what?

Indeed, a table keeping track of which listeners are registered on which buttons can solve the problem.

3.2. JTree questions

Model-view relationship. A tree consists of two kinds of components: one or more views and a data model. The tree model is shared by all the views. Whenever the tree model is changed, programmers must ensure that all views be notified and update their visual appearance to reflect the change.

The tree model of a JTree can be dynamically updated. That is, one can add and remove a tree node; or change the content of a tree node at runtime. To keep a tree's visual appearance in sync with the tree model, the tree model class provides 9 methods for such updates (see Table 5; details about these methods omitted). Programmers must choose a method suitable for their particular tasks. A thorny problem is that many programmers made mistakes when doing this. In the following we give two examples.

> I am using a level 3 JTree in our project. Under certain conditions, I need to remove some leaf nodes from JTree. When the JTree is collapsed, it works fine. But when the JTree is expanded, the removed node remains in the tree, although it is actually removed from the tree model. I have tried several methods like
> `tree.validate()`

Design features	Sub-features	Questions
model-view		44
	notification failures	
	tree model sharing	
node appearance		44
	leaf	
	text and icon	
	tool tips	
	components as renderer	
view manipulation		35
	multi-root tree	
	tree orientation	
	selection/expansion/collapse	
	auto-scrolling	
node editor		20
	enabling and disabling	
	saving editor result	
	appearance	
inherited features		19
	mouse	
	paint	
	JScrollPane	
tree model		13
	node ordering	
	user objects	
tree listeners		10
misc		5
	domain problems	
	design discussion	
Total		190

Table 4. JTree questions

> `clearSelection()`
> I also tried first to collapse the leaf's parent, remove the leaf, and then expand the parent again but in vain.

This programmer did not understand the fine details of the 9 notification methods and failed to notify the JTree that a node had been removed from JTree's tree model. This could be solved by invoking either `nodeStructureChanged` or `removeNodeFromParent`.

Here is the second example.

> I got a JTree object I want to display on two different panes of a JTabPane. Each pane has a panel where the JTree will go. Both panels are identical and start with a simple label saying the tree data is still building. Each pane implements Observer and when the tree has completed loading (from a SQL database), it notifies its observers. So far so good. Here's the problem. On one pane, the tree shows up and works great, ie, it responds to mouse clicks. On the other pane, the tree shows up only occasionally and when it does, it doesn't respond to mouse clicks. Additionally, if I switch to a different pane and come back, the tree is gone. Does anyone have an idea what is going on here?

This problem had two causes. One was that the same JTree object was shared by two different panels. Thus it broke the invariant that a component hierarchy must be a tree. The other was that the programmer seemed to be unaware that a tree model can be shared by multiple trees. The problem could be solved by creating two JTree objects and letting them share the tree model.

insertNodeInto
nodeChanged
nodesChanged
nodeStructureChanged
nodesWereInserted
nodesWereRemoved
reload()
reload(TreeNode node)
removeNodeFromParent

Table 5. Tree model notification methods

Node appearance. JTree uses a tree cell renderer object to draw its nodes. At the conceptual level, whenever a drawing is needed, the following algorithm is executed:

```
for each node in tree model
  draw node with the renderer object
```

Several states are related to the visual aspects of tree nodes. A node can have a icon, a text, and a tooltip. It can be either an internal node or a leaf. A leaf and an internal node require different icons. An internal node can be either expanded or collapsed, indicated by corresponding icons. Finally, a node can be selected and highlighted visually.

The renderer object has a method with many parameters to capture all the relevant state for the node to be drawn:

```
Component
getTreeCellRendererComponent(JTree tree,
Object value,  boolean selected, boolean
expanded,  boolean leaf,  int row,
boolean hasFocus)

Sets the value of the current tree cell
to value.  If selected is true, the
cell will be drawn as if  selected. If
expanded is true the node is currently
expanded and if leaf is true the node
represents a  leaf and if hasFocus is
true the node currently has  focus. tree
is the JTree the receiver is being
configured for. Returns the Component
that the renderer uses to draw the value.
Returns:
the Component that the renderer uses to
draw the value
```

This is a 'big' interface. The renderer object knows everything about the tree and the node to be drawn. As such, it takes on many responsibilities. Almost all adjustments to node appearance involve this renderer object.

Here is one question whose poster did not know that the `selected` parameter in the above method signature means that the node is selected.

> I've got a JTree and what I need is to be able to change the icon on a node when it is selected. I'm using TreeSelectionListener to detect when the node is clicked. I've figured out how to setup different icons for different nodes using a renderer but I'm not sure how to change the icon when a node is selected.

To set the icon, the programmer can do the following in the `getTreeCellRendererComponent` method of `TreeCellRenderer`:

```
if(sel)
  this.setIcon(yourselectedicon);
else this.setIcon(yournonselectedicon);
```

View manipulation. The JTree class itself contains a rich set of methods to manipulate its visual aspects such as selection; node expansion and collapse; tree orientation; and multiple-root tree. Even though most of them are standalone and simple to use, a rather large number of questions were asked in this category.

Node editor. Similarly to tree cell renderers, JTree also uses a tree cell editor to edit nodes. By default, the editor can be activated by a combination of three mouse clicks. Programmers can disable or invoke the editor programmatically. Other questions were about how to save editing results and adjusting an editor's visual appearance. The next example is about invoking an editor.

> I have a JTree with editable nodes and it works fine with the triple click to start the edit (or the click, pause, click). How can I "force" someone into edit mode programmatically? For example, I also have a menubar and I'd like to add an "Edit" menuitem to it, but I don't know how to throw them into the selected node's cell editor.

As the following code snippet shows, the programmer can invoke the editor by calling `startEditingPath`.

```
void actionPerformed(ActionEvent e){
     Path = //getSelectionPath
     tree.startEditingAtPath(path)
}
```

Inherited features. JTree also inherits a set of features from Component, Container, and JComponent. 19 questions are related to how to interact with these features. For example, one asked how to determine if the mouse enters or leaves a node, which can be done by first registering a mouse listener to the JTree and then invoking a method from JTree to determine if the mouse is within the area of a node. Another asked how to set the background of the tree area to some image, which can be solved by overriding `paintComponent`.

Tree model. This category involves everything about setting up a tree data structure. Here is an example of a programmer not knowing how to wrap up a user object with a `DefaultMutableTreeNode` object.

> I would like to populate a JTree with objects. Up to now I have been using a string from the object, as all the examples show how to do, but I need to call functions from that object. I still need the string to appear on the tree nodes naturally. I have found no examples that do this through the several books that I have perused. The DefaultMutableTreeNode only takes an object. I could extend my own version of it, but not sure how to get the tree to display the name strings.

The solution is to return a string from his object's `toString` method and obtain his user object from a DefaultMutableTreeNode by calling `getUserObject`.

Tree listeners and misc. Programmers seemed to understand listeners, and there were only 10 questions in this category. The misc category has 5 questions. Some were about design issues such as how to implement undo for JTree. Others were about the programmers asking for a solution for a task that was not particularly relevant to the framework.

4. Analysis of results

The classification results drew our attention to particularly problematic design features. We carefully re-examined these features and identified a few design issues. We also cross-checked these features with existing documentation [14], and found places where documentation was either incomplete or completely missed out the topic. Finally, analyzing 300 topics also gave us a chance to discover some people issues relevant to the general framework use problem.

In this section, we discuss our findings in terms of three aspects: design, documentation, and people.

4.1. Design issues

Tightly coupled variation points. A variation point provides programmers the ability to hook into a framework. If a variation point depends on too many parts of the framework, it is considered 'tightly coupled'. Framework designers should try to reduce the extent of coupling of such variation points and explicitly document them if they are unavoidable. In the latter case, framework designers could also help by anticipating typical usage scenarios and providing default implementations accordingly.

TreeCellRenderer is tightly coupled with everything related to JTree. This can be understood intuitively by examining the dependences in Figure 3. Note that in the figure, the tree cell renderer node depends on both JTree and TreeModel. At the programming level, this dependence is reflected by the long list of parameters of the `getTreeCellRendererComponent` method. Our classification corroborated this observation: the node appearance feature has as many as 44 questions. Thus this variation point is considered problematic.

Unfortunately, the details of this variation point were not documented carefully enough. The one paragraph of

Javadoc API documentation for the method `getTreeCellRendererComponent` is too brief to cover all its potential usage. Upon examining the Swing tutorial [14], We could find only a brief mention of this variation point on page 444 and a short code snippet on page 445. A second example is the tree cell editor, which is similar to the tree cell renderer in nature, but was not documented in the book at all.

A third example of such a variation point is the layout manager. A layout manager depends on the structure of the GUI component hierarchy and information about the size and position of the components for laying out GUIs. Therefore, it is tightly coupled with the internal working of the Swing framework. But there aren't many questions on layout managers. This is probably because Swing provides a set of seven layout managers and carefully documents each with a how-to section.

De-localized concerns. An object-oriented design such as JTree often consists of a number of classes and interfaces, and the source code for a feature or concern may be scattered in multiple syntactic constructs. How to modularize such concerns is an interesting research topic. But for practitioners, a practical solution is to document such cases carefully, for future maintenance and reuse. Our analysis of programmers' questions revealed that some important de-localized concerns in JTree design were not documented at all. As a result, programmers were left on their own to understand the features. In one case, because a variation point was not documented and thus its existence unknown, an awkward solution became popular in the forum. In the following, we describe two examples of de-localized concerns identified in our study.

For example, the implementation of editing was scattered in three classes: DefaultTreeCellEditor, TreeModel, and JTree. In particular, when an editing session ends, the internal implementation calls TreeModel's `valueForPathChanged` method to transfer the value within the editor to the underlying user object. Thus in order to save the result to a user object, programmers must override this method in a subclass. This important design was not documented. Consequently, programmers often asked why their programs could not correctly save the result of editing.

Another example is related to node appearance. By default, a tree model treats a node as a leaf if the node has no child and displays a leaf icon for the node accordingly. However, sometimes programmers want to be able to programmatically determine if an empty node should be regarded as a leaf or not. For example, it makes sense to display an empty directory of a file system as an internal node rather than a leaf. The logic of determining leaves is de-localized into two classes (4 methods about the AllowsChildren property in DefaultTreeModel and DefaultMutableTreeNode). Correct use of such features requires coordinated actions among all involved parties. Here is an example.

> I'm making a JTree displaying files and directories. [...] I've got a problem with empty dir because they've no child. My question is: how can I set a particular node NOT leaf (as if it had children) even if it is a leaf node in my JTree? I tried the `setAllowsChildren (true)` method on `DefaultMutableTreeNode`, but it does not change the L&F.

This programmer forgot to invoke the related method `setAskAllowsChildren` on on his tree model.

Inherited features. Table 3 shows that 26 out of 110 JButton questions are related to inherited features. A careful re-examination of JButton's inherited features revealed that a large portion of questions (16 out of 26) were related to the paint feature, and the revalidate and repaint feature [14]. These two features are at the core of the framework and difficult to understand for most programmers.

Table 4 shows that 19 out of 190 JTree questions are related to inherited features. The difference between JButton and JTree in their numbers of questions related to inherited features also suggests that there are more requirements for custom buttons than custom trees. For example, one programmer asked about how to create a custom button that could be rotated to a certain direction. Perhaps trees are so difficult to set up that no one considers customization, or JTree just happens to be suited to most users.

The Swing inheritance tree shown in Figure 1 is typical in object-oriented design. We suspect that the substantial number of questions on these inherited features may be due to programmers not fluent enough in object-oriented programming. This is consistent with an observation made by [2]. That is, novice programmers poorly exploit static relationships such as inheritance in their comprehension activities.

Advice: remove special cases whenever possible. Unlike graphical user interfaces, usability for programming interfaces has not been studied much [12]. One piece of advice for framework designers is to remove special cases from their framework whenever possible. Universally held properties have the desirable feature of learn-once and apply-everywhere. Special cases force programmers to remember extra information, and thus impede understanding and increase the chance of making errors. In the following, we describe two anecdotes from Swing to illustrate this point.

The first example is the `getContentPane` method. Top-level containers such as frames, windows, dialogs, and applets consist of a special internal structure [14]. As a result, programmers must not add components directly to these containers. Instead, they must get the content pane of a container first and add their components to the content pane. This is a special case because normally one adds a component to a container by simply calling the add method. Such

a special case surprised programmers and numerous questions were generated in the Swing Forum. The Swing team has corrected this design flaw [13] in J2SE 1.5 such that one can add components directly to top-level containers.

The second example is setting tooltips for tree nodes. The Swing framework has a general way of setting tooltips for components, which hides much of the detail behind a pair of getter and setter methods. However, because JTree was treated differently than other Swing components, it broke this good design. Suddenly, programmers were exposed to the full details of setting up tooltips. Consequently, endless questions were asked in the forum. Although it is not clear how to fix this design, the lesson is clear: adding special cases to a framework causes problems.

4.2. Documentation issues

We started learning Swing with Sun's on-line Javadoc API documentation. Our single biggest complaint about the documentation is its failure to present the logical groupings of methods, both for those within the same class and those across different classes. It is expensive to discover the delocalized concerns. Tools in the spirit of concern graphs [11] could be used to substantially improve such automatically generated documentation.

Recently, we began to use the Swing tutorial book [14]. This tutorial contains a number of nice features. First, it covers every important topic. Second, after providing an architectural overview, the book is organized into how-to sections, each on a topic in a self-contained manner. Our own learning experience is that it is rather expensive to discover a variation point based on only Javadoc and source code. Every non-trivial variation point should have how-to documentation. Third, the tutorial groups APIs logically, not alphabetically. For example, a subset of JTree methods are grouped into tree creation and setup, selection, and showing and hiding nodes. Based on our negative experience with Javadoc, we believe that logically grouped documentation makes it much easier for programmers to learn a framework; they help programmers avoid much of the expensive, bottom-up grouping.

Despite the high quality of the tutorial, it is not perfect. We compared our results with the respective how-to documents for JButton and JTree and identified what important details were missing and where more detail should be added. For example, the tree cell renderer is only discussed on pages 444 and 445, which is insufficient to answer many of the raised questions. The model-view relationship is mentioned only briefly with one example on page 448. For two other important features (editors, and using other components as renderers) we could not find any relevant information from the book.

Our study also helped to identify undocumented details about framework APIs. For example, one programmer invoked the `getHeight` method on a button but the program did not work as expected. The reason turned out to be that `getHeight` returns a meaningful value only after the component has been laid out. The following Javadoc failed to mention this important precondition.

```
public int getHeight()
  Returns the current height
  of this component. This
  method is preferable to writing
  component.getBounds().heightcom-
  ponent.getSize().height
  because it doesn't cause any heap
  allocations.
Returns:
  the current height of this component
Since:
    1.2
```

In terms of comprehension of software frameworks, we feel that doc-driven understanding is more efficient than reverse engineering from source code. Source code should be the last resort for a programmer to consult. Understanding from source or Javadoc-like documents is just too expensive. It needlessly increases the cost of using a framework.

4.3. People issues

The JButton data (see the misc category of Table 3) provides some hints on people issues related to understanding software frameworks.

First, forums in general can be an excellent resource for debugging, learning about platform bugs, and discussing design issues. Bugs contained in a snippet of posted code can be quickly pointed out by peers. Many times we were impressed on how fast someone could respond to a buggy program. The second benefit of posting to the forum is to quickly find out if a problem in one's program is due to a platform bug. People also brought their design issues to the forum and discussed them with their peers, often ending up with good suggestions.

Programmers asked quickly answerable questions about individual APIs. For example, a large number of questions in the category of tree view manipulation were about individual APIs of the JTree class. The forum quickly pointed out the right APIs to posters, thus avoiding a search through a large set of APIs. On the other hand, tools could help to reduce the number of such questions. Development environments such as Eclipse can provide on-demand lightweight documentation of APIs.

Many questions were posted by beginners, such as how to create an action listener. These questions can be recognized by the kind of source code posted, some of which did not even pass the Java compiler. Generally beginners, like college students, were treated with respect. Most of the time, there would be someone providing assistance to beginners.

Most programmers do not systematically study the design of a framework. Good design and documentation can only do so much to make framework use easier. Programmers sometimes need to be informed when the task they are attempting requires a serious effort at obtaining a deeper understanding. Often the forum takes on this educational role.

5. Conclusion and future work

Modern software development is gradually moving towards larger scales of reuse such as software frameworks. A significant obstacle is that learning how to use an object-oriented framework requires a rather large effort from a programmer. Forums provide an ongoing communication channel among designers of a framework and programmers who use it. Forums play a significant role in assisting programmers in the learning task, and informing the framework developers and technical writers of problems in design or documentation.

Studying questions raised by real programmers using a real framework can give deeper insights into the problems of framework usage. But effectively exploiting the content of a forum requires a systematic analysis. This study presents an approach to doing just that. Our solution-oriented classification of forum questions identifies features of the framework that are poorly designed or badly documented, or cases of problematic programmer practice. This information can then be used to guide the future efforts of the framework team.

For example, the historical record in the forum can help the framework developers distinguish between conceptual problems that are common just to novices (and solved by education) from confusing architectural weaknesses that require redesign.

As another example, technical writers can distill common problems and identify areas for which new documentation can be developed. Even a rudimentary classification can be valuable when used to structure a repository of frequently asked questions.

Another problem is the size of framework documentation. Searching for a small piece of text in a huge volume of documentation is painful. We face a dilemma: on the one hand, we need more pages to document useful information; on the other hand, adding more pages increases the retrieval effort. A possible solution is to integrate framework documentation within a modern development environment. Tools like the Framework Constraint Language (FCL) [6] can be extended to automatically calculate a subset of relevant documentation based on what framework features are being used by the programmer's code. For example, if a program does not use JTree, then the tool could hide all JTree related documentation from the programmer. Information retrieval tools in the spirit of Hipikat [3] could also be used to assist programmers.

6. Acknowledgements

This work was funded by Natural Sciences and Engineering Research Council of Canada (NSERC), the Alberta Research Council, and Alberta Ingenuity. Eunice Yin assisted in collecting JTree questions.

References

[1] A. Andrews, S. Ghosh, and E. M. Choi. A Model for Understanding Software Component. In *Proceedings of International Conference on Software Maintenance*, 2002.

[2] J.-M. Burkhardt, F. Detienne, and S. Wiedenbeck. The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. In *Proceedings of IEEE International Workshop of Programming Comprehension*, 1998.

[3] D. Cubranic and G. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proceedings of International Conference on Software Engineering*, May 2003.

[4] M. Eisenstadt. My Hariest Bug War Stories. *Communications of the ACM*, 40(4):30–37, 1997.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. O. Vlissides. *Design Patterns-Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[6] D. Hou. *FCL: Automatically Detecting Structural Errors in Framework-Based Development*. PhD thesis, Department of Computing Science, University of Alberta, December 2003. Available as technical report TR04-01.

[7] D. Hou, E. Yin, and H. J. Hoover. Classifying Newsgroup Questions about JTree and JButton. Technical report, Department of Computing Science, University of Alberta, January 2004. TR05-07.

[8] I. R. Katz and J. R. Anderson. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction*, 3:351–399, 1987-1988.

[9] A. V. Mayrhauser and A. M. Vans. Program Understanding During Software Maintenance and Evolution. *IEEE Computer*, pages 44–55, August 1995.

[10] A. V. Mayrhauser and A. M. Vans. Program Understanding Behaviour During Adaptation of Large Scale Software. In *Proceedings of IEEE International Workshop on Program Comprehension*, 1998.

[11] M. P. Robillard. *Representing Concerns in Source Code*. PhD thesis, Department of Computer Science, University of British Columbia, November 2003.

[12] K. Rodden and A. Blackwell. Class Libraries: A Challenge for Programming Usability Research. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*, 2002. www.ppig.org.

[13] Sun Microsystems, Inc. Changes in Working with ContentPane. In *Core Java Technologies: Technical Tips*, November 2004. http://java.sun.com/developer/JDCTechTips/2004/#1102.html.

[14] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial (Second Edition)*. Addison Wesley, February 2004.