Documenting and Evaluating Scattered Concerns for Framework Usability: A Case Study

Daqing Hou and Chandan Raj Rupakheti
Electrical and Computer Engineering, Clarkson University, Potsdam, New York, USA 13699
{dhou, rupakhcr}@clarkson.edu

H. James Hoover
Computing Science, University of Alberta, Edmonton, Canada T6G 2E8
hoover@cs.ualberta.ca

Abstract

Scattered concerns, design features whose implementations span multiple program units*, can pose extra difficulty for developers to locate, understand, modify, and extend. In particular, since successful application frameworks tend to be widely used by many developers, the impact of scattered concerns on framework usability can be especially significant. Ideally, every scattered concern present at the framework interface should be carefully evaluated to justify its introduction. If it cannot be avoided, it should at least be well-documented to facilitate its use.*

To gain insights into how scattered concerns are actually *designed and documented in industrial frameworks, a method for documenting and evaluating scattered concerns is proposed, and a manual pilot study on the JFC Swing JTree is performed. In this method, concerns are identified and documented in a structured manner, listing not only those items which an application must directly depend on, but other methods and classes that indirectly contribute to the design of a concern. The documented concerns are compared with the framework architecture in order to justify their existence. Concerns that map nicely to the architecture are deemed to be acceptable as they fit within the design and were explicitly traded off for other quality attributes. Those that show a mismatch indicate a place where the concern could cause problems for application developers, and thus should be either refactored or well-documented. Concerns are also evaluated using design criteria like cohesion and coupling. While it is not always possible to identify design flaws in a scattered concern, at the least, documenting these concerns will make them easier to use. The JTree study results in 12 concerns, 7 of which are missing or only partially documented in the official Swing tutorial. 4 framework usability flaws are identified, which, if addressed, would lead to the removal of 2 of the 12 documented concerns and improvements to another 2. Thus, the proposed method has the potential to be useful to framework development teams.*

1. Introduction

Locating and understanding code relevant to conceptual considerations, or so-called *concerns*, is a fundamental task in software engineering, a prerequisite to both software modification (as in maintenance and evolution) and extension (as in reuse). [1] In particular, dealing with *scattered concerns*, conceptual units implemented across multiple program units, can be both time-consuming and error-prone [12, 14]. Scattered concerns may take extra time for developers to understand because they must locate and reason about code dispersed throughout the program, and untangle the concern code from code related to other concerns. Scattered concerns are also difficult to extend consistently because multiple locations in the code that may not be explicitly marked as related must be extended simultaneously.

The interfaces of object-oriented frameworks are often made of scattered concerns [1, 4, 5], which affect both programer productivity and product quality. In order to identify opportunities to improve framework usability (for developers), we propose a new development activity to document and evaluate the scattered concerns appearing at the framework interface. Our method can be summarized in three steps. First, "important" concerns are identified. This step can typically be driven by feedback from the actual use of a framework. A concern is considered important if it is used by multiple applications and is generating many questions but is not adequately documented. Second, the concern's design is then investigated and documented. Third, the documented concern design is compared against high-level design in a way similar to the reflexion model [11] in order to

1 We take the liberal view that anything of interest to a developer at some point in time can be a *concern*. Requirements are concerns.

conclude whether the concern has been introduced as a result of explicit design decision or not. The concern is also examined under other design criteria like cohesion and coupling to assess its quality.

The performance of this evaluation activity (by either the framework designers or third parties) may yield several outcomes and benefits. The resulting concern documentation will be useful to framework users. Since production frameworks are seldom sufficiently documented, this benefit cannot be undervalued. More importantly, the process of documenting and evaluating scattered concerns may draw attention to the usability of a framework. For example, a high number of scattered concerns would alert the framework team that application developers will have difficulties in using their framework, and they should consider either redesigning the framework or at least carefully documenting these concerns. This practice may also help identify design flaws, which can be valuable input for future releases.

To learn how to carry out such activity as well as assessing how scattered concerns are dealt with in practice, we performed a pilot study using the JTree component of the JFC Swing [15] as a subject framework. (Turning this approach into a general methodology requires future work involving more case studies and deeper analysis.) Specifically, we manually recovered twelve scattered concerns for JTree. We then documented them using our proposed notation. We characterized the concerns in terms of scattering and tangling effects, analyzed their potential impact on learning a framework, and evaluated them against the MVC architecture. We found that 9 out of the twelve scattered concerns are the direct consequence of MVC. We also examined the detailed designs of these concerns in terms of the design criteria of coupling and cohesion, and were able to uncover 4 design flaws that we believe have negatively affected the usability of JTree.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the pilot study. Section 4 presents the scattered concerns identified. Section 5 characterizes these concerns, compares them against the MVC architecture, and discusses the design flaws identified. Finally, Section 6 concludes the paper.

2. Related Work

Scattering and tangling effects. Soloway et al. empirically study the effect of delocalized plans (scattered concerns) when programmers perform program modification tasks and design review, and proposed design documentation to counter the negative effect of the delocalization [14]. The program modification task was done with a Fortran program of 250 lines with 14 modules. Our study is on a much larger scale. In addition to documentation, we also evaluate the quality of scattered concerns.

Documenting scattered concerns. Robillard's concern graphs [12] are a lightweight representation of concerns that use program elements and their relationships. The main goal is to enable a developer to effectively build up a concern representation as they explore code in an IDE. Concern graphs are conceived primarily in the context of program evolution. We focus on scattered concerns in the context of using a framework. Our notation is essentially a textual version of a concern graph with extra annotations. It supports free text as well as code snippets that show important control and data flow, and inheritance, which concern graphs do not currently support. We feel that these additional annotations are necessary for concern descriptions to be useful to a developer.

API usability. Ellis et al. empirically compare the use of the factory pattern versus constructors in API design, and conclude that the factory pattern is often detrimental to API usability [3]. Bloch presents a set of design guidelines for good APIs for Java [2]. Our work focuses on evaluating the quality of scattered concerns in a framework.

Empirical study of component based software engineering. Based on a case study, Morisio et al. report that both productivity and quality in framework-based development can be better than in traditional development, and that productivity increases massively when developers learn more about a framework over time [9]. Mohagheghi et al. report from an industrial setting that reused components have less bugs and are more stable than non-reused ones, and that bugs in reused components tend to receive high priority [8]. In a study of a reuse process in NASA, Morisio et al. report that familiarity with COTS is considered critical to COTS selection and project success [10]. Our study aims to help developers deal with framework usage problems.

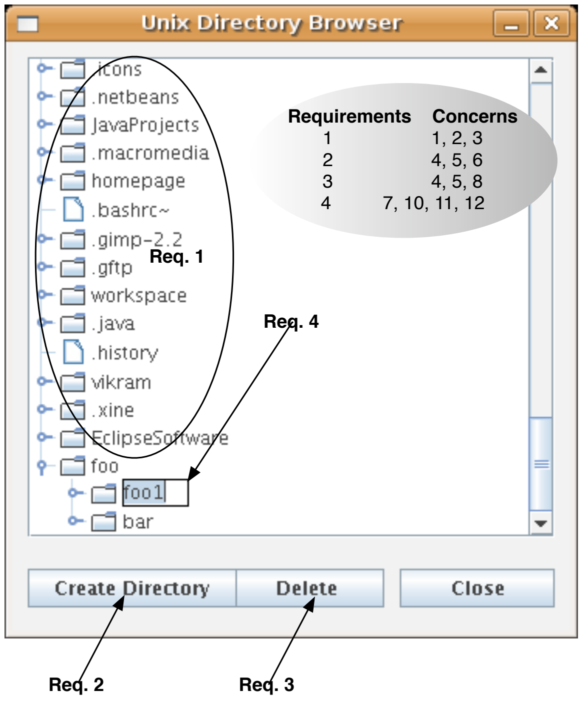
3. Description of Pilot Study

3.1. Identification of scattered concerns

The effectiveness of the proposed design evaluation relies on the quality of the recovered concerns. While it is reasonable to assume that the actual framework developers will have little problems in identifying scattered concerns in their own framework, for a third party like us to perform such evaluation, it is critical to have sufficient experience and knowledge about the subject system. In a previous study, we looked into the programming questions reported to a Swing news forum [7] in order to understand what causes difficulty in using a framework. This motivated us to study the design of JTree in detail. That study also revealed several scattered concerns not covered elsewhere. In this study, we thoroughly studied the JTree design by reading the source code provided as part of JDK. We also referenced the online JavaDocs and the Swing tutorial [15]. Furthermore, we wrote applications to confirm our understanding of JTree and implemented alternative designs to correct the identified design problems. Lastly, to further ensure the quality of the recovered concerns, we have mod-

Concern (Fig. #)	Description
1. tree model (2,Y)	Setting up tree model.
2. model-view (3,Y)	Connecting JTree with tree model to display it.
3. leaf node (4,N)	Displaying tree node without children as internal node.
4. notification (Y)	Notifying JTree of model changes to update its view.
5. changes (5,N)	Four kinds of changes to tree model.
6. adding node (6,Y)	Adding node to tree model.
7. renaming node (N)	Renaming a node.
8. removing node (N)	Removing a node.
9. drastic change (N)	Changing more than 2 levels of tree model at once.
10. create editor(7,Y)	Setting up an editor.
11. start editor (N)	Invoking an editor.
12. stop editor (N)	Stopping an editor.

Table 1. Overview of 12 JTree scattered concerns. Numbers in parentheses indicate figures where concern descriptions can be found. Concerns not covered by the official Swing tutorial are marked with N.



Requirements	Concerns
1	1, 2, 3
2	4, 5, 6
3	4, 5, 8
4	7, 10, 11, 12

Figure 1. Directory browser with 4 requirements mapped to 12 concerns.

eled all of them with the ConcernMapper tool [13]. As a result of all these efforts, we are reasonably confident that our understanding of JTree is sufficient to guarantee the validity of this study. The outcome of this process is a set of 12 scattered concerns listed in Table 1.

Classes & Interfaces	#method	#loc
JTree	135	4879
BasicTreeUI	126	4312
TreeModel	8	137
DefaultTreeModel	30	660
DefaultMutableTreeNode	54	1490
DefaultTreeCellEditor	26	737
TreeModelListener	4	81
TreeModelEvent	9	297
DefaultTreeCellRenderer	43	575
TreePath	15	318

Table 2. JTree related classes and interfaces.

Table 2 lists the major classes and interfaces investigated in this study. Several classes, like JTree and its look-and-feel peer BasicTreeUI, also contain non-trivial inner classes, which are not listed. The lines of code measure used includes comments and blank lines. These classes will be referenced in the discussion.

3.2. Illustrating scattered concerns with concrete JTree usage scenario

To help illustrate the 12 concerns, a familiar example of a directory browser is used, whose requirements are outlined as follows. A sample browser is shown in Figure 1, where the four requirements are marked up and mapped to the concerns of Table 1.

> Build a directory browser to support the exploration of a file system. Initially, the browser should display all the files under the current working directory as a tree (**Req. 1**). A client may select, expand, or collapse any directory node. Once a directory is selected, the client may create a new directory (with a button, **Req. 2**), delete a file (with a button, **Req. 3**), or rename a file (with a cell editor, **Req. 4**). To rename a file, the client can directly double-click the node to activate a cell editor. A new name can be entered into the editor and confirmed by a return key. If the name is not used by any sibling file, the selected node will be renamed. Otherwise, the client should be notified to provide a valid name or cancel the renaming.

3.3. Concern notation

The notation used in our concern descriptions (Section 4) supports elements like classes, interfaces, methods, and fields, as well as important relationships like inheritance (extends and implements), method overrides, and method calls (=>). A class is surrounded in a pair of angle brackets (<>), optionally followed by a list of relevant fields and methods, and an interface a pair of square brackets ([ ]).

Source code elements can precisely describe the API elements that application code will interact with. APIs that a client needs to call or override are marked up with * and **, respectively. Thus our concern descriptions not only cover specific ways of using a framework, but can also contain information to be used to explain why it should be used as such. Our concern descriptions also allow for the use of English to describe important conditions and events, and specify method behavior. Although informal, natural language can be expressive. Note that one may also use UML notation to represent concerns.

4. Documenting JTree Concerns

This section presents the 12 JTree related design concerns shown in Table 1. The presentation is organized around the concrete scenario of building the directory browser. In the interest of space, only a subset of the recovered concerns are covered in detail. Full details can be found in [6].

4.1. Setting up a directory browser

To set up a directory browser, a directory tree data structure is needed for representing the file system. To display the directory tree with JTree, the data structure must support the TreeModel interface. The directory tree can be built upon the java.io.File class.

```
<DefaultTreeModel> implements TreeModel
setRoot(TreeNode root)  *

<DefaultMutableTreeNode> implements MutableTreeNode
Object userObject;
setUserObject(Object userObject)  *
setParent(MutableTreeNode newParent) *
insert(MutableTreeNode child, int index)  *

[MutableTreeNode] extends TreeNode
setUserObject(Object userObject)
    Sets the user object for this node to userObject.
setParent(MutableTreeNode newParent)
insert(MutableTreeNode child, int index)

[TreeNode]
TreeNode getParent()
TreeNode getChildAt(int childIndex)
int getChildCount()
int getIndex(TreeNode node)
Enumeration children()
```

Figure 2. Tree model.

Figure 2 depicts how to build up a tree model. Swing provides a default implementation for TreeModel, DefaultTreeModel, which composes a root TreeNode. Tree nodes are implemented by two interfaces, TreeNode and MutableTreeNode, and one class, DefaultMutableTreeNode. TreeNode models a read-only node that contains positioned children and a parent, MutableTreeNode represents a tree node whose parent, children, and *user object* can be changed, and DefaultMutableTreeNode implements both interfaces. Note that the DefaultMutableTreeNode can compose a *user object*. Thus, each File can be wrapped up by a DefaultMutableTreeNode, which can be added as the child of another DefaultMutableTreeNode for directory. In this way, a tree model can be obtained.

```
[TreeModel]
Object getRoot()
Object getChild(Object parent, int index)
int getChildCount(Object parent)
      Returns the number of children of parent.
int getIndexOfChild(Object parent, Object child)
      Returns the index of child in parent.

<JTree>
JTree(TreeModel newModel) *
     Returns a JTree using the specified data model.
JTree()                                         *
     Returns a JTree with an internal DefaultTreeModel.
JTree(TreeNode root)             *
     Returns a JTree with the specified root node.

setModel(TreeModel newModel) *
TreeModel getModel()         *
```

Figure 3. Model-view.

The model-view concern in Figure 3 depicts how to set up a JTree. A JTree reads the data structure defined by the TreeModel interface. A tree model can be passed to a JTree when the JTree is constructed, or a root node can be passed to the corresponding JTree constructor, and the JTree will then create a DefaultTreeModel internally. The TreeModel for a JTree may be reset.

4.2. Showing empty directory as internal node

```
<DefaultTreeModel> implements TreeModel
protected boolean asksAllowsChildren
setAsksAllowsChildren(boolean asksAllowsChildren) *
boolean isLeaf(Object node)
=> ((TreeNode)node).isLeaf()? (asksAllowsChildren
     && ((TreeNode)node).getAllowsChildren(): false
[TreeModel]
boolean isLeaf(Object node)

<DefaultMutableTreeNode> implements TreeNode
boolean setAllowsChildren(boolean allows)   *
[TreeNode]
boolean getAllowsChildren()
boolean isLeaf()

<JTree>
displays icon according to TreeModel.isLeaf()
// undesired coupling with JTree!
JTree(TreeNode root, boolean asksAllowsChildren)
  A JTree with the TreeNode as  its root. A tree model
  will be created internally,  which will use
  asksAllowsChildren to test the leafness of a node.
```

Figure 4. Leaf node.

JTree displays internal nodes and leaf nodes with different icons. By default, a node without any child is regarded as a leaf. But this may not always be desirable. For example, an empty directory should still be treated as an internal node. The leaf nodes concern shown in Figure 4 describes how to specify a node without any child as an internal node. JTree learns whether a node is a leaf or not by invoking isLeaf(node) on its TreeModel, which will in turn ask the node. To support cases like an empty directory, DefaultTreeModel contains a field asksAllowsChildren that controls whether it should ask if an empty node is an internal node. If asksAllowsChildren for the DefaultTreeModel is set to true, an empty node can be viewed as an internal node by calling setAllowsChildren(true) on the DefaultMutableTreeNode.

4.3. Notifying changes in tree model

Swing separates a JTree from its tree model. The JTree displays the tree nodes contained in the tree model. When the tree model is changed, JTree needs to be notified about the change in order to adjust its view accordingly. The model notification concern is implemented as an instance of the Observer design pattern [6]. A change in a tree model is represented by a TreeModelEvent, which is posted to and processed by all the TreeModelListeners registered on the tree model.

```
[TreeModelListener]
treeNodesChanged(TreeModelEvent e)
   Invoked after a set of siblings change their names.
treeNodesInserted(TreeModelEvent e)
   Invoked after nodes have been inserted into parent.
treeNodesRemoved(TreeModelEvent e)
   Invoked after nodes have been removed from parent.
treeStructureChanged(TreeModelEvent e)
   Invoked after a subtree has drastically changed.

<TreeModelEvent>
TreeModelEvent(Object source, Object[] path,  int[]
       childIndices, Object[] children)
    Indicates nodes are changed, inserted, or removed.
    source: this tree model.
    children: nodes changed, inserted, or removed.
    childIndices: indices of modified items in parent.
    path: nodes from root to parent of modified nodes.

TreeModelEvent(Object source, Object[] path)
    Indicates subtree under path changed drastically.
```

Figure 5. 4 kinds of changes to tree model.

JTree distinguishes four kinds of changes to the tree model: three local changes that rename, insert, or remove, one or more sibling nodes under a parent node, and one global change that alters the sub-tree structure drastically. These changes require different responses from the view. Local changes will affect only a group of sibling nodes on the view, while a global change will force the view to collapse a whole subtree under a node. As shown in Figure 5, each change is represented by a TreeModelEvent and processed by a corresponding method in a TreeModelListener.

Figure 6 depicts the adding nodes concern. Nodes can be added by using either insertNodeInto() or nodesWereInserted() on the DefaultTreeModel. Both methods will call the fireTreeNodesInserted() method to create a TreeModelEvent object and fire the treeNodesInserted() method on all the TreeModelListener's currently registered to the DefaultTreeModel. In particular, since JTree registers a TreeModelListener to the DefaultTreeModel, it will re-

```
<DefaultTreeModel> implements TreeModel
insertNodeInto(MutableTreeNode newChild, MutableTreeNode
                          parent, int index) *
nodesWereInserted(TreeNode parent, int[] childIndices) *
=>protected fireTreeNodesInserted(Object source,
     Object[] path, int[] childIndices, Object[] children)
 => e=TreeModelEvent(source, path,
                     childIndices, children);
  ((TreeModelListener)listeners[i+1]).treeNodesInserted(e);
[MutableTreeNode]
insert()

[TreeModelListener]
treeNodesInserted(TreeModelEvent e)

<TreeModelEvent>
TreeModelEvent(Object source, Object[] path, int[]
           childIndices, Object[] children)
```

Figure 6. Adding tree node.

ceive the change notification and update the tree view. Concern descriptions for renaming and removing nodes as well as global change are similar, and can be found in [6].

4.4. Expanding and deleting nodes

Instead of loading a whole directory tree up front, a tree node can be expanded dynamically in response to a user's click on the user interface. This was done by registering a TreeExpansionListener to JTree, which, when a directory is expanded, will add all of its sub-directories to the tree model. To notify this change to JTree, the adding nodes concern in Figure 6 must be followed. Deleting a directory can be done similarly.

4.5. Renaming a directory node

A directory node can be renamed by using a tree cell editor. Since the tree cell editor design as a whole is too big to reason about, it is divided up into three sub-concerns: the setting up, starting, and stopping of a tree cell editor.

```
<DefaultTreeCellEditor>
// undesired coupling with TreeCellRenderer!
DefaultTreeCellEditor(JTree, TreeCellRenderer) *
   An editor that uses a text field as editor component.
boolean isCellEditable(EventObject) **

<JTree>
TreeUI ui;
setCellEditor(TreeCellEditor editor) *
=>ui.setCellEditor(editor)
 =>BasicTreeUI.updateEditor()
    => editor.addCellEditorListener(BasicTreeUI.handler)
setEditable(boolean) *

<BasicTreeUI>
handler that saves the editing value.
```

Figure 7. Setting up tree cell editor.

Figure 7 shows the setting up a tree cell editor concern. A key step is to create and pass a DefaultTreeCellEditor to JTree via JTree.setCellEditor(). JTree.setEditable(true) needs to be called to make the tree editable. The editability of individual nodes can be controlled by overriding DefaultTreeCellEditor.isCellEditable(). An important side effect of setCellEditor() is that a handler in BasicTreeUI is registered to the editor, which, when the editing session is stopped, will be activated to save the value in the editing component to the tree node object (part of the stop editor concern).

The starting a tree cell editor concern installs an editor component dynamically onto the JTree component. This complex interaction involves 6 methods from 3 classes (BasicTreeUI, JTree, and TreeCellEditor). The stopping an editor concern uninstalls the editor component and passes its result to the TreeModel, and involves 5 methods from 4 classes (TreeModel in addition to the above three). Complete concern descriptions can be found in [6].

5. Evaluating JTree Concerns

In this section, we first characterize the concerns developed in Section 4 in terms of the scattering and tangling effects of the concern code, and discuss their impact on learning a framework. The presence of a scattered concern in a system should be justified by a sound design decision. Thus we evaluate these concerns against the MVC architecture. Finally, examining these concerns in terms of the coupling and cohesion design criteria also reveals four potential design flaws in the framework.

5.1. Scattering and tangling

Table 4 summarizes the classes and interfaces that participate in each concern, which are surrounded in a pair of brackets in the concern descriptions. It is possible to include a subtype as well as its supertype, as a concern may be introduced in terms of the supertype. To highlight their architectural roles, the classes and interfaces are categorized as model, view, controller, or the data flow between model and view (m-v). The methods column contains, for each concern, the number of API methods versus the total number of methods mentioned in the concern description. A method that appears in both a supertype and a subtype is counted only once (e.g., the isLeaf(Object) method in Figure 4).

On average, each concern involves from two to five classes, and one to nine methods. Thus, the implementation for the concerns is *scattered*. At the same time, most classes participate in four or more concerns. Thus, multiple concerns are *tangled* within these classes.

The TreeModel interface shown in Table 3 further illustrates this tangling effect. This interface, with only eight methods, participates in four concerns (tree model notification, model-view, leaf nodes, and stopping editor). As a result, the eight methods serve four different purposes. Thus, discerning the individual concerns from such classes is a major task when learning the framework. A larger class, like JTree, with well over a hundred methods and perhaps dozens of concerns, will be much more difficult to understand than TreeModel.

Note that the majority of the multiple program elements involved in a concern do not interact directly with the client code. That is, most of them are not in the framework API that a client needs to call or override. Their presence in the concern descriptions is mainly to help explain how the framework design works.

Method	Concern
addTreeModelListener	model notification
removeTreeModelListener	model notification
getChild	model-view
getChildCount	model-view
getIndexOfChild	model-view
getRoot	model-view
isLeaf	leaf node
valueForPathChanged	stopping a cell editor

Table 3. TreeModel crosscut by 4 concerns.

These concerns also hint on the amount of effort needed to locate them. For example, the stopping editor concern (Table 4) involves four types and five methods, but only two of the methods interface directly with the client code. These two methods are only a very small portion of the total 295 methods of the four types (JTree, BasicTreeUI, TreeModel, and DefaultTreeCellEditor in Table 2). This illustrates that additional aids like documentation are needed to help developers discover a concern faster from a large number of classes and methods.

5.2. Mapping the scattered concerns to MVC

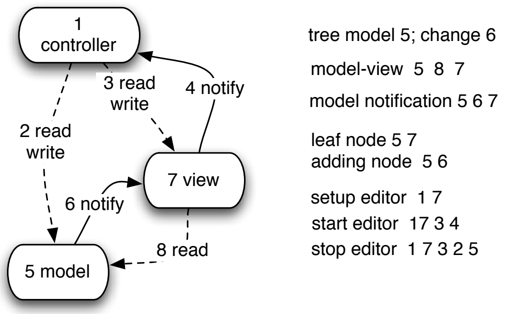


Figure 8. MVC (left) and the mapping from recovered JTree concerns to MVC (right).

MVC, as shown in Figure 8, structures an interactive system into model (5), view (7), and controller (1). Models hold application states. Views present the models to the user (8). Controllers process user input from devices such as keyboards and the mouse (4), and may read/write both models and views (2 and 3). When the state of the model is changed, the model will notify its views (6). When the state of a view changes (e.g., the selection in JTree), or an event happens, the view will notify its controllers (4).

The advantage of the MVC structure includes increased flexibility and reusability. Decoupling models from views

	Model					**View**		**Controller**		**m-v**	**methods**
concerns	TM	DTM	TN	MTN	DMTN	JTree	TCR	DTCE	BTUI	TML& TME	API/ total
tree model		X	X	X	X						4/9
leaf node	X	X	X		X	X					2/6
model-view	X					X					5/9
notification	X					X				X	0/1
changes										X	0/6
adding node		X	X	X						X	2/5
renaming		X	X							X	2/5
removing		X	X	X						X	2/5
drastic change		X	X							X	3/6
setup editor						X	X	X	X		4/4
start editor						X		X	X		2/6
stop editor	X					X		X	X		2/5

Table 4. Summary of JTree scattered concerns. (TM: TreeModel; DTM: DefaultTreeModel; TN: TreeNode; MTN: MutableTreeNode; DMTN: DefaultMutableTreeNode; TCR: TreeCellRenderer; DTCE: DefaultTreeCellEditor; BTUI: BasicTreeUI; TML: TreeModelListener; TME: TreeModelEvent)

makes it possible to reuse the same view to work with different models. Separating controllers from views makes it possible to provide custom controllers for the same view.

In light of MVC, the concerns in Table 4 can be divided into four groups. The first two concerns are about the tree model, the next two are about the model and view relation, the next five refine the model notification, and the last three provide the tree editor as an example of the controller.

In Figure 8, each concern is accompanied by the MVC elements it involves. For example, the model-view concern involves model (5), the read edge (8), and view (7), since it expresses the fact that JTree reads the data modeled by TreeModel. The Model notification concern, on the other hand, involves model (5), the notify edge (6), and view (7).

Two concerns, tree model and change, are local to the architectural elements 5 and 6, respectively. The other ten concerns all involve at least two MVC elements. Furthermore, all concerns except leaf node align well with the design intent of MVC. Thus we conclude that the MVC design produces most of the scattered concerns (9 out of 12). While MVC increases the reusability and flexibility of the framework, the price to pay is the introduction of scattered concerns and the associated usability problems.

Note that the twelve scattered concerns contain additional semantics beyond MVC. For example, the scattered concerns model four kinds of changes and notifications to the tree model (see column m-v in Table 4 for the concerns involved). The MVC design shown in Figure 8, however, abstracts away all these details into a single edge (6). This implies that in addition to the architectural comparison shown in this section, further evaluation of scattered concerns at the detailed design level is also needed .

5.3. Usability of scattered concerns

Examining these scattered concerns helps us identify four suspicious designs in the framework that affect its usability. These issues, if addressed by the framework team, would lead to the removal of two of the twelve concerns and improvements to another two.

The first two issues are of undesired coupling. Figure 7 shows that TreeCellRenderer is involved in the creating a cell editor concern. But intuitively, TreeCellRenderer and JTree should not belong to this concern. TreeCellRenderer is used by JTree for painting a tree node. It should not have anything to do with a tree editor. Figure 4 indicates that JTree participates in the leaf node concern. But how to determine a leaf node is a concern that should have been completely local to TreeModel. This decision should not be passed, via the JTree constructor, to TreeModel.

The other two design problems are related to model notification and the tree cell editor. When studying questions asked about the Swing framework [7], we observe that these two designs have produced many questions and defects, and thus are error-prone. For example, not knowing the design and APIs of model notification (Figures 5 and 6), many developers failed to initiate a proper notification for a change to a tree model. It is not unusual to see that after writing some code like the following, a developer complains that a new node does not show up in the tree view.

```
parentNode.insert(newNode, index);
```

Had they known the adding node concern in Figure 6, instead they would have written

```
DefaultMutableTreeNode parentNode, newNode;
File childDir;
          ...
newNode = new DefaultMutableTreeNode(childDir);
newNode.setAllowsChildren(true);
treeModel.insertNodeInto(newNode, parentNode, index);
```

or

```
parentNode.insert(newNode, index);
treeModel.nodesWereInserted(parentNode,new int[]{index});
```

Note that in the code snippets above, insertNodeInto() and nodesWereInserted() are invoked on a TreeModel to notify JTree about the node insertion.

It is clear to us that the unclean designs in the existing framework have somehow contributed to these user problems. In the model notification example above, the root cause is the separation between node insertion and notification. We have implemented an alternative design where the notification is done automatically for node insertion and deletion. In this new design, a developer does not have to remember to notify JTree about changes. Thus at least two concerns related to notification (adding node and removing node in Table 1) can be removed.

We have also redesigned the tree editor with a conceptually much cleaner interface. However, available space does not allow for the elaboration of the new design. We believe that our new design is less error prone and easier to use than the existing one, and we plan to test this hypothesis via formal experimentation as future work.

6. Conclusion

We investigated a method to evaluating the design and documentation of scattered concerns in frameworks to improve framework usability. Specifically, we performed a pilot study with Swing JTree and recovered twelve scattered concerns. Seven of these concerns were not properly documented in the official Swing tutorial. Thus, the concern descriptions developed as part of the activity can be used as additional documentation. We further performed two levels of design analyses to the recovered concerns, and found that although most concerns conform to the MVC architecture, the detailed design can be improved for better usability. In particular, four design flaws are identified as having negatively affected the usability of JTree. Fixing these usability flaws will lead to the removal of at least two of the twelve concerns and improvements to another two. We conclude that this activity has been effective in identifying opportunities for improving the usability of JTree.

Although a single study does not allow us to generalize this result immediately to other developers (for performing the design evaluation), frameworks, or components, we believe that the proposed approach has the potential to be actually used by framework developers to improve their frameworks. Given the importance of framework usability on software productivity and quality, this study justifies further refinement of the approach.

Acknowledgments

The authors are grateful to the three anonymous reviewers for their constructive comments on the initial paper. Daqing Hou wishes to thank Ciera Christopher and Jingyue Li for commenting on earlier drafts of this paper.

References

[1] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *Proceedings of MoDELS '06*, 2006.

[2] J. Bloch. How to Design a Good API and Why it Matters. `http://lcsd05.cs.tamu.edu/slides/keynote.pdf`. Last verified: September 10, 2008.

[3] B. Ellis, J. Stylos, and B. Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of ICSE '07*, pages 302–312.

[4] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *Proceedings of OOPSLA'06*, pages 75–88.

[5] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of ECOOP/OOPSLA 90*.

[6] D. Hou. Scattered Concerns from JFC Swing JTree. `http://www.clarkson.edu/~dhou/projects/JTree-extra.pdf`. Last verified: September 10, 2008.

[7] D. Hou, K. Wong, and H. J. Hoover. What can programmer questions tell us about frameworks? In *Proceedings of IWPC '05*, pages 87–96.

[8] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of ICSE '04*, pages 282–292.

[9] M. Morisio, D. Romano, and I. Stamelos. Quality, productivity, and learning in framework-based development: An exploratory case study. *IEEE Trans. Softw. Eng.*, 28(9):876–888, 2002.

[10] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft, and S. E. Condon. Investigating and improving a cots-based software development process. In *Proceedings of ICSE '00*, pages 32–41.

[11] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of FSE '95*, pages 18–28.

[12] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[13] M. P. Robillard and F. Weigand-Warr. ConcernMapper: Simple view-based separation of scattered concerns. In *eTX '05*, pages 65–69.

[14] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

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