Evaluating Forum Discussions to Inform the Design of an API Critic

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

***Abstract*—Learning to use a software framework and its API (Application Programming Interfaces) can be a major endeavor for novices. To help, we have built a critic to advise the use of an API based on the formal semantics of the API. Specifically, the critic offers advice when the symbolic state of the API client code triggers any API usage rules. To assess to what extent our critic can help solve practical API usage problems and what kinds of API usage rules can be formulated, we manually analyzed 150 discussion threads from the Java Swing forum. We categorize the discussion threads according to how they can be helped by the critic. We find that API problems of the same nature appear repeatedly in the forum, and that API problems of the same nature can be addressed by implementing a new API usage rule for the critic. We characterize the set of discovered API usage rules as a whole. Unlike past empirical studies that focus on answering why frameworks and APIs are hard to learn, ours is the first designed to produce systematic data that have been directly used to build an API support tool.**

***Keywords*-Software Frameworks; APIs; Critic; Symbolic Execution; Online Forum Discussions; AWT/Swing; Java**

I. Introduction

Efficient development of high-quality software is critical for business competitiveness. To this end, software reuse using frameworks and libraries has proven effective. Frameworks and libraries offer canned solutions for a set of common problems in some specific domain, for example, the signal processing libraries of Matlab or a GUI framework. They provide leverage in large part because they are used by many applications through a published API (Application Programming Interfaces). Unfortunately, frameworks, libraries, and APIs are hard to learn and use [7], [12], [16].

The challenge of learning and using APIs can be explained by the difference between an experienced and a novice user of a programming tool. Experienced programmers are able to use the context of the problem they are tasked to solve to narrow down the set of possible solutions they need to explore. Like experienced chess players, experienced programmers seem to rule out all but a handful of promising lines of development within the context of the tools at their disposal. Novices, in contrast, tend to lack the big picture that would otherwise help organize their attack and, thus, become overloaded.

Due to time pressure and an urge to solve problems quickly, many programmers learn APIs on demand and learn them by doing. That is, they try to learn just enough of an API so that they can solve the current task. Search tools can support this practice by helping locate relevant code examples. But programmers, especially novices, cannot always formulate good queries for what they are looking for [4], [11]. Furthermore, even if they find relevant code examples, they still face a significant challenge to understand and evaluate them for relevance [11]. Programmers can also seek help from online forums, but sometimes there can be significant response delay [13]. It would be ideal to engage a human expert for help, but experts are scarce. When popular third-party APIs are used, it may even be impossible to connect with the experts.

To help narrow down the performance gap between novices and experts, we are investigating a code analysis tool whose goal is to understand the problem-solving context like an expert, so that it can advise the novice in a timely and meaningful manner [1] [17]. In the literature, a computer program that critiques human-generated solutions is called *a critic* [18]. Critics have been successfully applied in clinical medicine management, engineering design, word processing, and software engineering [18].

In designing our API critic, we distinguish among three kinds of critiques: *criticisms* ("this code behavior is inappropriate"), *explanations* ("what have caused the code to behave this way"), and *recommendations* ("you may need this next"). These three kinds of critiques are broadly designed to address the respective well-known challenges in *debugging*, *understanding*, and *finding* relevant solutions [12]. In general, our API critic is expected to help bridge the long-standing information gap between API designers and application programmers, and thereby increase the quality of the novice's code, as well as move him or her toward being an expert.

We have built a first prototype critic for the AWT/Swing API. Specifically, we use symbolic execution [10] to trace API client code to create program states. The critic then examines the symbolic program states to recognize relevant behavioral features present in the API client code, which helps capture the problem-solving context. For example, a relevant behavioral feature could be that a widget has no size information set when it is made visible. Relevant features are defined as *API usage rules*, which are implemented in Java code that examines the symbolic program states. If the captured symbolic program state matches, or triggers, a specified *API usage rule*, our tool

[1]The **CriticAL** project (A **Critic** for **API**s and **L**ibraries) can be found at SourceForge (http://sf.net/p/critical). All URLs verified 2/14/2012.

will present context-appropriate documentation to critique the API client code.

The prototype critic implemented an initial set of API rules that was created based on our personal experience with the layout logic in the AWT/Swing API. While functional, we were not sure how *complete and general* our prototype was in terms of covering layout-related problems. Hence, to justify, and more importantly, to guide the further development of our critic, we would need solid empirical data about API use from the field. This motivated us to conduct a case study to analyze and collect such data from the programming discussions in the Swing Forum [2]. Specifically, we want to answer two questions:

- What are the characteristics of API usage problems? How are the needs for the three kinds of critiques grounded empirically?
- Do similar problems recur? To what extent can our critic help advise practical API usage problems?

The focus of this paper is to report on this case study, rather than the detailed design of the critic, but more details about the inner working of our critic infrastructure can be found elsewhere [17]. Although several studies have been directed toward answering the question why APIs are hard to learn and use [6]–[8], [12], [16], prior research has not produced concrete data that can be directly used to build an API support tool. The results of our study, including a spreadsheet that summarizes our analysis for each case of forum discussion, along with the source code that we collected from the forum (89 runnable programs), are made publicly available at

https://sourceforge.net/projects/critical/files/icpc2012/.

The remainder of this paper is organized as follows. Section II presents a survey of related work. Section III presents our research method. Results of our study are reported in Sections IV (criticisms), V (explanations), and VI (recommendations). We summarize our main findings in Section VII. Section VIII discusses the threats to study validity. Finally, Section IX concludes the paper.

II. Related Work

Ko, Myers, and Aung identify six learning barriers in a study of 40 novice programmers learning Visual Basic [12]. These barriers take into account several factors such as design, selection, coordination, use, understanding, and the availability of information about APIs. They make recommendations for improving end-user programming systems by providing more examples, improving the search experience, making the invisible system rules more visible through error messages from tools, making development environments more interactive, and developing tools that could explain some of the complex behavior of the APIs to the novices. These recommendations align closely to the goals for the API supporting systems that Fischer proposes [3]. Our critic is designed to meet these goals as well [17].

Several studies have pointed out the importance of design knowledge for the proper use of APIs. These include Robillard and DeLine's qualitative analysis of the API learning difficulties perceived by Microsoft developers [16]; Hou's quantitative analysis of framework learning difficulties for undergraduate students [6]; and Ko and Riche's qualitative study on the role of conceptual knowledge in using APIs [11]. Our critic operationalizes the API design knowledge to help with API use at the fine-grained level of API usage rules.

In addition to understanding framework design, it is also important to find the right API elements for a programming task. Code search tools such as Blueprint [1] bring code examples to the IDE by keyword-based searching. Other tools [5], [14], [19] help programmers locate documentation, examples, and related code on the web. However, lacking conceptual knowledge, novice programmers often find it hard to formulate a useful search query and to assess the relevance of the results [11]. Our critic can help grow the conceptual knowledge by critiquing code behavior, and by recommending precise suggestions directly within the programming context.

There are recommendation tools that extract and recommend common API sequences, e.g., [20], but such tools do not make design inferences based on program behavior and may suggest irrelevant API elements without proper explanations. It is not clear either how programmers would respond to such irrelevant recommendations. Moreover, program analysis tools provide only criticisms for generic design and implementation bugs, whereas the goal of this work is to discover API-specific rules. Tools for explaining program behavior through multiple views have also been researched previously [15]. None of these tools integrate the presentation of explanations, criticisms and recommendations for APIs in the same way as ours does.

Our past work on analyzing the Swing Forum have also assessed the challenges programmers face while using APIs [7]–[9]. Different from these, this work results in a set of framework rules and documentation related to GUI composition and layout, which have been directly used in building an API critic [17]. In particular, we show that the API-related problems recur, and we conclude that our critic is promising to become an effective tool for addressing these problems.

III. Research Method

In this case study, we are interested in answering two research questions:

- What are the characteristics of API usage problems? How are the needs for the three kinds of critiques grounded empirically?
- Do similar problems recur? To what extent can our critic help advise practical API usage problems?

To this end, we have conducted a case study of the online programming questions in the Java Swing Forum. To conduct an in-depth exploration, we have chosen to narrow down the scope of the analysis and to focus our study on problems related to *GUI composition and layout* in the Java Swing API. GUI composition and layout is an essential topic in GUI programming that is backed by a strong design, but which many novices have great difficulty with. Lessons learned from this study are likely to be generalizable to other similar APIs.

[2]https://forums.oracle.com/forums/forum.jspa?forumID=950

We employ Eisenhardt's methodology for case study research [2]. In our study, each forum discussion thread represents a real-world scenario (or a case, in terms of case study research) where somebody is having certain problems with the API. A typical discussion thread contains multiple posts with questions, answers, and code examples. Many posted code examples are self-contained, compilable programs so that forum members can run to assess the problems. Since there were too many discussion threads in the Swing Forum to go through manually (more than 46,000 threads and more than 211,000 messages), to expedite the process, we searched the forum with the keyword *layout* [3]. This query returned 264 threads. We sequentially analyzed 150 of the 264 threads returned, striving to understand each case thoroughly.

Eisenhardt's method dictates that the observer must be intimately familiar with the cases/subjects. To ensure that, we have paid close attention to the fine details in the cases. Specifically, in addition to reading the text throughout, we compiled and ran each code, sometimes with necessary modifications, in order to explore each case in detail. This process not only helped us get the full understanding of each case, but also resulted in a set of 89 test cases for testing our critic. In particular, we were able to clearly separate a problem about debugging or program behavior from those where the OP requests for additional information. During the process, we occasionally referred to the online tutorials [4] and the API reference manual for help.

As Eisenhardt describes, the process of identifying categories from case data and encoding them is highly iterative. In our study, the categories are the specific *API use rules* that can be used to trigger helpful advice based on the symbolic program states of the API client code. (For example rules, see Section IV Criticisms, Section V Explanations, and Section VI Recommendations). We label each thread with all rules that we conclude are useful for the thread. Disagreements between the two authors, in terms of both the interpretation and the classification of the discussion threads, were resolved through numerous discussions over the course of more than ten months. The results of our analysis, in the form of a spreadsheet along with the code collected from the forum, are available online.

Figure 1 shows the top-level categories that lead to a final categorization of the 150 threads according to how they can be supported by the three kinds of critiques:

- There were 9 threads for which we did not understand what was the OP's key question (Original Poster), due to either poor English or unclear presentation, e.g [5].
- Among the 141 clear cases, 7 were not related to the layout API that this study is focused on. For example, in one case [6], although the word "layout" appears in the messages, the OP was asking about how to change the layout of the keyboard from English to German.
- 17 of the 141 clear threads were about application specific requirements related to layout. For example, one OP posted a GUI design diagram and asked how to achieve it [7]. In such cases, there is not much that our critic can help other than consulting a human expert.
- The rest of 117 threads were related to *layout and GUI composition* that we conclude can be supported through the three forms of critiques, that is, *criticisms* (50 threads), *explanations* (21 threads), and *recommendations* (60 threads).

[3]Search URL: https://forums.oracle.com/forums/search.jspa?threadID=&q=layout&objID=f950&dateRange=all&userID=&numResults=15&rankBy=10001

[4]http://docs.oracle.com/javase/tutorial/uiswing/

[5]https://forums.oracle.com/forums/thread.jspa?messageID=5838236

[6]https://forums.oracle.com/forums/thread.jspa?messageID=5833268

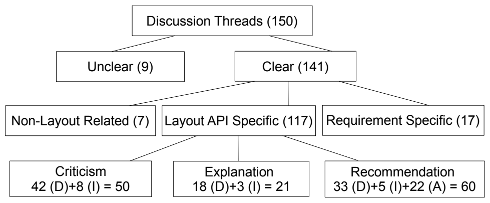


Fig. 1. Classification of 150 Swing Forum discussion threads. **D**: Directly helpful to an OP's core problems; **I**: Indirectly helpful; **A**: Anticipated by authors to be most likely helpful. A thread helped by multiple critiques of the same kind is counted only once. Since a thread may be helped by more than one kind of criticism, recommendation, and explanation, the total number for critiques (50+21+60) is more than 117.

The legends **D**, **I**, and **A** used in Figure 1 need to be defined. If a critique could directly help [8] the OP to solve the asked problem(s), we classified the thread under **D**. If a critique could point out other problems in the OP's code, but which were not directly asked by the OP, then we classified the thread under **I**. Based on the evidence collected from a thread, if we felt that a recommendation could most likely help OP but the thread did not have enough information for us to firmly classify it as either **D** or **I**, we classified the thread under **A** ('Anticipated'). As shown in Figure 1, for recommendations, there are 33 threads that can be directly helped by some recommendations, 5 indirectly helped, and 22 anticipated to be helped. Finally, if a thread was helped through more than one critique of the same kind, we only added one to the total count for that kind of critique (e.g., if a thread was helped by two criticisms, we would only add one to the total count for criticisms).

Since our critic analyzes source code, we have paid particular attention to how code was used in the forum. Our result shows that code is commonly used for communicating about API usage problems. Based on the analysis of the 141 clear threads, we found that 42.6% of the OPs (60 threads) posted code when asking about their problems, and that in 38.3%, or 54, of the 141 threads, some forum users replied with code. In this study, we have collected a total of 89 runnable Java programs and made them publicly available for other researchers.

[7]https://forums.oracle.com/forums/thread.jspa?messageID=5888922

[8]If the critique is a criticism, 'help' means that it finds a bug. If the critique is a recommendation, 'help' means that it provides information needed by the user. If the critique is an explanation, 'help' means it answers a user's question about the code.

In the next three sections, we discuss in detail the major findings of this study, that is, the critiquing rules that we have identified for criticisms, explanations, and recommendations.

IV. Criticisms

A criticism informs the programmer about some undesirable behavior in the API client code. Table I depicts the list of specific criticism rules that we have identified in this study, their current status of implementation, and how many times each rule has been found to be useful. The Return-on-Investment, which is defined as the ratio between the number of times all rules have been applied and the number of rules, gives a sense how general and useful a rule can be on average. Section IV-A introduces these rules and briefly discusses how they are checked by our critic. To provide more background to understand each rule in context, five cases are presented as examples in the follow-up subsections.

A. API Criticism Rules

To lay the necessary background for the remaining discussion, we start with a review of some fundamentals of GUI programming. A GUI programmed with the Swing API is essentially a tree data structure. The root of the tree is a special top-level widget such as a `JFrame` or a `JDialog`, leaves are made of basic GUI widgets such as `JLabel`, `JTextField`, and `JButton`, and internal nodes a container such as `JPanel`, which contains other widgets recursively. To be displayed, a widget must have a location and a size computed or explicitly set. A container may rely on a `LayoutManager`, which is essentially an algorithm, to automatically compute the size and location for each of its child widget, based on layout-specific constraints and strategies. Swing provides several built-in layout managers, each with a different layout strategy, such as `FlowLayout`, `BoxLayout`, `GridLayout`, `GridBagLayout`, and `SpringLayout`. Rather than using a layout manager, a programmer also has the option to manually specify the size and location for each widget, which is also known as *absolute positioning*.

The following criticism rules in Table I are found useful to enforce the internal consistencies of the GUI tree:

- *Orphan GUI Objects*: To be visible, all GUI objects must be part of a GUI tree rooted at a top-level component such as a `JFrame` or a `JDialog`.
- *Parent Switching*: When the containing GUI tree is invisible, moving a widget between two containers has no effect and, thus, should be avoided.
- *Missing Layout Constraints*: When using a `SpringLayout`, necessary constraints for the container as well as its widgets must be specified to get the desired effect.
- *Misplaced Layout Constraints*: Some layout managers, such as `BorderLayout` and `GridBagLayout`, require layout specific constraints to position the child components. When widgets are added to a container that uses a layout manager, only constraints specific to the layout manager should be used.
- *One Layout, One Container*: The relationship between layout managers and containers must be one-to-one. Each layout manager maintains the size and position information of the child widgets for a container. Sharing a layout manager may result in unpredictable GUI behavior. The following example shows three panels sharing the same `BorderLayout` [9]:

```
BorderLayout layout = new BorderLayout();
JPanel ui = new JPanel(layout);
JPanel preview = new JPanel(layout);
JPanel figures = new JPanel(layout);
```

- *Content Mismatch*: When a container is made visible, it must have the same set of child widgets with its layout manager.
- *Positioning and Sizing Constraints*: When a layout manager is used by a container, calling `setLocation()`, `setSize()`, and `setBounds()` methods on child components have no effect and should not be used. When `null` layout is used, the `setPreferredSize()`, `setMinimumSize()`, and `setMaximumSize()` methods have no effect and should not be used. The `JFrame.pack()` method should be used only when the content pane of the frame has a layout manager or when it has an explicitly set preferred size.
- *Dynamic GUIs*: When the content of a container is changed, it must be revalidated and repainted for the change to take effect.

Each API targets to solve a particular set of problems. APIs, thus, have some usage conventions. While it is not necessarily always wrong to use an API in a way deviating from conventions, such a use is nonetheless unusual, often showing signs of confusion or neglection from the programmer. Spotting such deviations can thus be useful. We have identified two common deviations from conventions, which can be detected by examining symbolic program states:

- *Components Resizing Behavior*: Not all components are meant to be resized in both directions. By convention, widgets such as `JButton` and `JLabel` should not be resized in either direction. Widgets such as `JTextField` and `JPasswordField` could grow horizontally but not vertically. A violation of such conventions is often undesirable [10]. When they are violated, it can be useful to teach the users how to prevent the widget from stretching.
- *Table Design*: A table-like GUI design is conventionally achieved using one of `GridLayout`, `GridbagLayout`, and `SpringLayout` with a single container. Two adjacent containers cannot be used to create a table-like design. This is because it would be hard, if not impossible, to align the GUI widgets contained in the two containers.

As shown in Table I, the criticism rules are enforced as preconditions, postconditions, and invariants for the GUI

[9]https://forums.oracle.com/forums/thread.jspa?messageID=5890601
[10]https://forums.oracle.com/forums/thread.jspa?messageID=5854070

TABLE I
LIST OF HELPFUL CRITICISMS DISCOVERED IN THE FORUM (D: DIRECTLY HELPFUL, I: INDIRECTLY HELPFUL, T: TOTAL). THE RATIO OF RETURN-ON-INVESTMENT EQUALS TO THE NUMBER OF TIMES RULES ARE HELPFUL (70), DIVIDED BY THE NUMBER OF RULES (11).

API Criticism Rules	**D / I / T**	**impl'ted**
Postconditions		
Orphan GUI Objects	6 / 0 / 6	Yes
Missing Layout Constraints	2 / 2 / 4	No
Parent Switching	2 / 0 / 2	Yes
Misplaced Layout Constraints	3 / 1 / 4	Partially
Invariants		
Content Mismatch	4 / 0 / 4	Yes
Dynamic GUIs	4 / 0 / 4	Yes
One Layout, One Container	3 / 0 / 3	Yes
Preconditions		
JFrame.pack() Constraints	5 / 4 / 9	Yes
Positioning and Sizing Constraints	8 / 0 / 8	Yes
Deviation from Usage Conventions		
Components Resizing Behavior	10 / 4 / 14	No
Table Design	10 / 2 / 12	Yes
Total	**58 / 12 / 70**	
Return on Investment	(70/11=) **6.36**	

layout API. Since none of these rules look overly complicated, it is probably safe to speculate that programmers stumble upon them mainly due to lack of awareness of these simple rules. Hence a tool like our critic can be very useful.

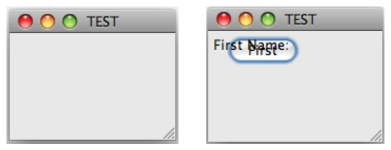
B. *Case 1 (Orphan Objects, Content Mismatch, Missing Constraints)*

The code for Case 1 is shown in Listing 1 [11], where the `CoordinateLayout` class extends Swing's layout manager `SpringLayout` to specify the position of a widget relative to its container. Our critic reveals three problems for the code.

Listing 1. Code for Case 1 (modified).

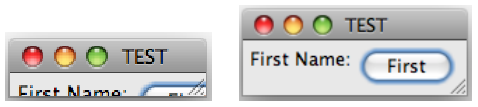
```
1  public class CoordinateLayout extends SpringLayout {
2   SpringLayout main; Container cont;
3   public CoordinateLayout(Container ct) {
4    main = new SpringLayout(); cont = ct; ...
5   }
6   public void addComponent(Component comp, int x, int y) {
7    main.putConstraint(WEST, comp, x, WEST, cont); // X−axis
8    main.putConstraint(NORTH, comp, y, NORTH, cont); // Y−axis
9   }}
10 public class CoordinateLayoutTest {
11  public static void main(String[] args) {
12   JFrame frame = new JFrame(``TEST'');
13   JPanel pane = new JPanel();
14   CoordinateLayout layout = new CoordinateLayout(pane);
15   pane.setLayout(layout);
16   JLabel aLabel = new JLabel(``First Name:'');
17   JButton aButton = new JButton(``First'');
18   // layout contains widgets but container does not
19   layout.addComponent(aLabel, 5, 5);
20   layout.addComponent(aButton, 15, 5);
21   frame.setSize(300,300);
22   frame.setContentPane(pane);
23   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24   frame.setVisible(true);
25  }}
```

[11] https://forums.oracle.com/forums/thread.jspa?messageID=5698221



(a) Widgets Invisible (b) Widgets Visible

Fig. 2. `JFrame` in Listing 1 before and after the fix for orphan widgets.



(a) Before (b) After

Fig. 3. Listing 1 before and after the patch for missing constraints.

First, the `SpringLayout` object created at line 4 and referenced by `main` is not used by any container. Hence it becomes an orphan GUI object. In fact, the `main` object is unnecessary, and the `main.putConstraint()` at lines 7 and 8 should be replaced by calls to `this.putConstraint()`. Although our critic does not directly give this advice, pointing out the orphan layout object should help guide the OP closer to the right solution.

Second, since `aLabel` and `aButton` are not added to the `pane`, they become orphan objects too, and, thus, are invisible when the frame is made visible, as shown in Figure 2(a). Related, our critic also reports a problem of *Container and Layout Content Mismatch* for `pane`. This is because when the pane is made visible, it contains no child widget but its layout contains both `aLabel` and `aButton`. These two objects can be added by calling `pane.add(aLabel); pane.add(aButton);` before line 22. As shown in Figure 2(b), the two widgets then become visible.

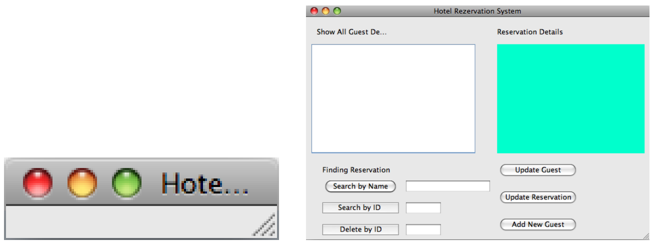
Third, our critic points out that the code fails to specify the constraints for the right (`EAST`) and bottom (`SOUTH`) edges of the content pane. As a result, the layout manager cannot correctly compute the size for the content pane. This problem is masked by the call to `setSize()` at line 21, but can be revealed by calling `frame.pack()`, which forces the layout manager to compute its size; as shown in Figure 3(a), the widgets are clipped. The code that follows can be applied to replace lines 19-20 in Listing 1, resulting in the view shown in Figure 3(b).

```
// Updated the coordinates for the button
layout.addComponent(aLabel, 5, 5);
layout.addComponent(aButton, 80, 5);
// Constraints for content pane with respect to aButton
layout.putConstraint(EAST, pane, 5, EAST, aButton);
layout.putConstraint(SOUTH, pane, 5, SOUTH, aButton);
```

C. *Case 2 (Parent Switching, Positioning and Sizing)*

Our critic reveals two problems for Case 2 [12] (Listing 2). The first problem is that `jLabel4` is first added to the content pane (line 7) but later to another container `jPanel1` (line 8).

[12] https://forums.oracle.com/forums/thread.jspa?messageID=5714432



(a) With `null` layout and `pack()`.
(b) Calling `frame.setSize())` to make other GUI widgets visible (scaled).

Fig. 4. The `JFrame` in Case 2 (Listing 2).

As a result, when the GUI tree is made visible, `jLabel4` is visible only under `jPanel1` but not under the content pane. In fact, the OP complained exactly about this. The critic advises the OP to create a new `JLabel`.

The second problem is calling `pack()` method on a `JFrame` whose layout manager is set to `null`. When the content pane's layout manager is set to `null`, and it does not have a *preferred size* set, the *pack()* method cannot compute the desired size of the window. As a result, the window becomes too small to show its title and content (Figure 4(a)). Instead of `pack()`, `setSize()` can be called to explicitly specify a size for the frame (Figure 4(b)).

There are cases where a call to `frame.setSize()` and `frame.pack()` appear together, e.g., this case [13]. These two methods cancel the effect of one another and should not be used together.

Listing 2. Case 2 (Parent Switching Positioning and Sizing).

```
JFrame frame = new JFrame();
frame.getContentPane().setLayout(null); ...
jPanel1 = new javax.swing.JPanel();
jPanel1.setBounds(700, 50, 270, 400);
jPanel1.setLayout(null)
frame.getContentPane().add(jPanel1); ...
frame.getContentPane().add(jLabel4);
jPanel1.add(jLabel4);
jLabel4.setBounds(700, 70, 180, 14); ...
frame.pack()
frame.setVisible(true);
```

D. Case 3 (Dynamic GUIs)

As shown in Listing 3 [14], the OP of Case 3 wants to switch widget `c` (line 1) and `lastSelectedLabel` in the container `puzzlePanel`. But the switching is not immediately visible but only after the frame is resized manually.

The process of adding and removing components in the GUI subtree of `puzzlePanel` makes the container invalid and the changes ineffective. Generally, such an issue arises from the dynamic construction of a GUI. The solution for the user is to explicitly tell the Swing framework to redo the layout. It can be done in two ways: by either calling `puzzlePanel.revalidate(); puzzlePanel.repaint();`, or by calling `pack();` on the root widget (`JFrame`) instead of just the `this.invalidate(); this.repaint();` methods in lines 9 and 10. The call to `revalidate()` first invalidates the previously computed size and position of the widgets and recomputes them by performing relayout of the changed container. The `pack()` method recomputes the layout of the whole GUI tree and not just the modified container. The modifications, however, becomes apparent when the frame resizes because the resizing event forces relayout.

Listing 3. Dynamic GUIs (modified).

```
1  puzzlePanel.remove(c);
2  puzzlePanel.remove(lastSelectedLabel);
3  gbc.gridx=cX;
4  gbc.gridy=cY;
5  puzzlePanel.add(lastSelectedLabel,gbc);
6  gbc.gridx=lX;
7  gbc.gridy=lY;
8  puzzlePanel.add(c,gbc);
9  this.invalidate();
10 this.repaint();
```

E. Case 4 (Content Mismatch, Positioning and Sizing)

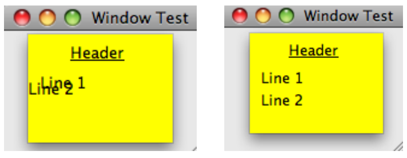
Listing 4. A `JWindow` that violates size constraints.

```
private void createAndShowWindow() {
 JWindow win= new JWindow(frame);
 win.setSize(120, 90);
 win.setLocation(90, 50);
 Container cp= win.getContentPane();
 cp.setBackground(Color.YELLOW);
 JLabel lb= new JLabel(‘‘<html><u>Header</u></html>’’);
 lb.setBounds(35,5, 80,20);
 cp.add(lb);
 for (int i=0; i<2; i++) {
  lb= new JLabel(‘‘Line ’’+(i+1));
  lb.setBounds(10,i∗20+30, 80,20);
  cp.add(lb);
 }}
 win.setVisible(true);
```

Our critic reveals two problems from the code in Listing 4 [15]. The content pane of the `JWindow` object has a `BorderLayout`. The content pane has all three labels but its `BorderLayout` contains only the last added label (`Line 2`). Hence the first two widgets are positioned using their specified sizes and locations and the last label positioned by the layout manager. As depicted in Figure 5(a), the label `Line 2` is positioned at the center location but the other two labels are located at their specified positions.

There can be two solutions to this problem. The first is to use `null` layout by calling `cp.setLayout(null)` and completely rely on absolute positioning. Figure 5(b) shows the effect of this solution. The second solution, and a better one, is to use only layout managers, without hard-coding sizes and positions.

[13]https://forums.oracle.com/forums/thread.jspa?messageID=5774019
[14]https://forums.oracle.com/forums/thread.jspa?messageID=5861121
[15]http://forums.oracle.com/forums/thread.jspa?messageID=9281019



(a) With layout (b) Without layout

Fig. 5. The view of `JWindow` before and after the fix.

F. Case 5 (Table Design, Resizing Conventions)

The code for Case 5 is shown in Listing 5 [16]. Although it gives an illusion of a table-like view as shown in Figure 6(a), where every widget seems to be positioned properly, this is only coincidental, and our critic gives two criticisms.

Listing 5. A table implemented using multiple containers (modified).

```
public static void main(String[] args) {
  JFrame frame = new JFrame(''DVD rental center'');
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel pane = (JPanel)frame.getContentPane();
  pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));

  JLabel rentFee = new JLabel(''Rent Fee '');
  JLabel lateFee = new JLabel(''Late Fee '');
  JTextField rentFeeField = new JTextField(10);
  JTextField lateFeeField = new JTextField(10);
  JPanel p1 = new JPanel(true);
  JPanel p2 = new JPanel(true);
  p1.setLayout(new BoxLayout(p1, BoxLayout.LINE_AXIS));
  p1.add(rentFee);
  p1.add(rentFeeField);
  p1.setAlignmentX(JPanel.LEFT_ALIGNMENT);
  pane.add(p1);
  p2.setLayout(new BoxLayout(p2, BoxLayout.LINE_AXIS));
  p2.add(lateFee);
  p2.add(lateFeeField);
  p2.setAlignmentX(JPanel.LEFT_ALIGNMENT);
  pane.add(p2); ...
}
```

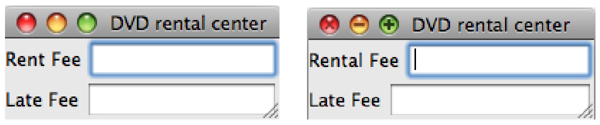
First, when the text in `rentFee` is changed from “Rent Fee” to “Rental Fee”, as shown in Figure 6(b), the two labels become different in length and the table columns are not properly aligned any more. In general, a table design that spans multiple containers often has alignment issues that are hard to solve. Instead of using multiple containers, the user is advised to use one of `GridLayout`, `GridBagLayout`, or `SpringLayout`.

Second, when the frame is resized, the text fields grow both horizontally and vertically, violating the resizing convention that a text field does not grow vertically. This is because `BoxLayout` is used and the text fields have a large default maximum size that causes them to grow.

V. Explanations

Our critic produces an explanation for those API elements that many users commonly find hard to work with, directly

[16]https://forums.oracle.com/forums/thread.jspa?messageID=5790663



(a) Labels in the same length. (b) Labels in different lengths.

Fig. 6. Table design achieved using three `BoxLayout`s.

TABLE II
List of helpful explanations discovered in the forum. (D: Directly, I: Indirectly, T: Total)

API Explanation Rules	**D / I / T**	**impl'ted**
Behavior of Null Layout	9 / 3 / 12	Yes
Behavior of `GridbagLayout`	5 / 0 / 5	No
Resizing Behavior of `BorderLayout`	4 / 0 / 4	Yes
API Specific Explanations	3 / 0 / 3	No
Total	**21 / 3 / 24**	
Return on Investment	(24/4=) **6**	

in the context where they are used. While explanations are also inherently part of criticisms and recommendations, we have found that explanations can help programmers just by themselves. Programmers often use an API without the full knowledge about its behavior and interaction with other code. They tend to be satisfied as long as the API elements appear to fulfill their needs, ignoring potential side-effects that often become visible later in the development. Explanations are useful in communicating such non-obvious, subtle behavior of the API. In this way, they facilitate the programmers in reasoning about their code.

In this section, we present some of the useful explanations that we have identified for GUI layout (Table II). Since programmers at different levels of expertise may need explanations with various levels of details, our study also demonstrates how opportunities for explaining API elements can be identified on an as-needed basis, by looking at actual forum discussions.

A. Behavior of Null Layout

As discussed in Section IV-A, when a container has a null layout, it must use *absolute positioning* to position its children. As a result, when the container is resized, its children will not resize automatically. Several OPs used `null` layout but still expected the child widgets to be resized automatically [17]. Our critic explains this implication of null layout to avoid the potential confusion. In addition, since the use of absolute positioning may adversely affect the appearance of the GUI when ported from one platform to another, the user should be advised about this behavior as well.

B. Behavior of `GridbagLayout`

`GridBagLayout` is a layout manager [18] that positions its child widgets inside a grid. It allows a child to span multiple rows and columns. The visual properties for each child, such as size and growth, are specified as parameters via

[17]https://forums.oracle.com/forums/thread.jspa?messageID=5849188
[18]http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html

a `GridbagConstraint` object. Some OPs were confused as to which constraint parameter causes what visual effect [19]. For example, in the absence of at least one non-zero value for `weightx` (`weighty`) for a column (row), that column (row) will not grow with the enclosing container. Moreover, in the absence of a fill property, a child component at each cell will not grow with the cell, leaving empty gap when the cell grows bigger. Explaining these can be very helpful for understanding the behavior of the API client code.

C. Resizing Behavior of `BorderLayout`

A `BorderLayout` positions its children in five pre-defined locations: center, north, south, east, and west. Some novices can be puzzled to by its exact behavior. For example, the element in the center takes all of the empty space and ignores the size property explicitly set by the user when resized. As the window becomes smaller, the center widget also grows smaller and eventually gets clipped. Only when there is no more space available for the center widget, do the widgets in south and then in north get clipped in the vertical axis and the widgets in west and then east get clipped in the horizontal axis. Such explanations can be helpful for the programmer to understand `BorderLayout` properly and also in deciding if the layout is the right choice [20].

D. API Specific Explanations

Some API elements need to be explained to help programmers understand certain behavior of the API client code. For instance, the call to `setMaximumSize()` on a basic widget such as `JButton` and `JLabel` can cause visual problems on a different platform where the maximum size set may be smaller than the required size [21]. Furthermore, setting the preferred size of one component has the surprising side effect of forcing all components in a `GridLayout` to have the same size [22]. Explaining these API elements can be useful for understanding the API client code.

VI. Recommendations

What differs an expert from a novice programmer is the amount of information they have about the framework and API. To help novices seek for API information, our critic recommends relevant API elements and documentation within the programming context. With enough of the programming context taken into account, the tool can make more precise recommendations on the use of API elements. It can also present competing solutions so that the user can choose the right one based on his or her needs. A key is the capability to infer the programmer's intent from the API client code. However, even with less knowledge of the programming context, our critic should still be able to make generic recommendations, just enough to push novices toward the right direction when they get stuck with their code. In both cases, recommendations bring information where the user needs the most. Table III shows a list of useful recommendations identified from the forum. We discuss them in order of ease of implementation.

TABLE III
List of helpful recommendations discovered in the forum. (D: Directly helpful, I: Indirectly helpful, A: Anticipated to be helpful, T: Total)

API Recommendation Rules	D / I / A / T	impl'ted
Confusing API Elements	4 / 1 / 0 / 5	Yes
Generic Recommendations	18 / 0 / 25 / 43	Partially
Usage Automata	4 / 3 / 0 / 7	Partially
Alternative Solutions	2 / 1 / 0 / 3	No
Layout Recommendations	5 / 0 / 1 / 6	No
Total	**33 / 5 / 26 / 64**	
Return on Investment	(64/5=) **12.8**	

A. Confusing API Elements

Sometimes, a group of API elements may share common terms in their names and perform related functions. This may confuse users in selecting the right one to use [23]. A group of such API elements from Swing are listed as follows:

- `setAlignmentX()` and `setAlignmentY()` of `JComponent`, which take parameters such as `TOP_ALIGNMENT` and `LEFT_ALIGNMENT`.
- `setHorizontalAlignment()` and `setVerticalAlignment()` of `JLabel` and `JButton`, with parameters such as `TOP` and `LEFT`.
- `setHorizontalTextPosition()` and `setVerticalTextPosition()` of `JLabel` and `JButton`, which takes its own unique set of parameters than the two groups above.

The first group above is designed to be used with `BoxLayout` to align its child widgets. The second is used by a component to align its content within itself. The third group is used for aligning the text within a `JLabel` and a `JButton` relative to its icon image. Whenever any one of the above confusing API elements is used with a `JLabel` or a `JButton`, a recommendation can be made to clarify the distinction among these three groups.

B. Generic Recommendations

Our critic recommends a list of *How-To* documents for the most frequently asked questions compiled from the discussion threads. Currently, the list includes the following topics: "How to use layout managers?", "How to do absolute positioning?", "How to dynamically change the location of a widget?", "How to add gap between components using a layout manager such as a `FlowLayout`?", and "How to combine multiple layout managers to achieve complex designs?". This list is presented to the user independent of what they are working on. In this way, the critic could still be helpful even when the programmer just begins to work on an API. This recommendation should be very useful for novices who do not have much conceptual knowledge of the API [11].

[19]https://forums.oracle.com/forums/thread.jspa?messageID=5743612
[20]https://forums.oracle.com/forums/thread.jspa?messageID=5849282
[21]http://forums.oracle.com/forums/thread.jspa?messageID=5698635
[22]http://forums.oracle.com/forums/thread.jspa?messageID=5828313
[23]http://forums.oracle.com/forums/thread.jspa?messageID=5827139

C. Usage Automata

Recommending future set of API elements can be helpful too. To improve the relevance of the recommended elements and avoid redundancy, our critic recommends only relevant elements that are not used in the current program. For example, recommending how to create a border for the `JPanel` in [24] would help solve the problem of the user. Similarly, a recommendation for the unused alignment properties of `FlowLayout` and `BoxLayout` as well as the missing constraints for `BorderLayout` and `GridBagLayout` could be helpful for novice programmers. However, further research is needed on how to strike a good balance between the quality and quantity of recommendations and the amount of technical sophistication.

D. Alternative Design

Listing 6. Using `GridBagLayout` where `FlowLayout` suited better.

```
1  protected void relayout() { // Complicated way
2    final GridBagConstraints gbc = new GridBagConstraints();
3    gbc.anchor = GridBagConstraints.LINE_START; // Start left
4    gbc.weighty = 0.0; // Grow with a factor of 0
5    gbc.fill = GridBagConstraints.NONE; // Do not fill extra space
6    gbc.insets = new Insets(0, 10, 0, 10); // External padding ...
7    for (int i = 0; i < panels.size();i++) {
8     gbc.gridx = i; // Single line, increase column when adding
9     rootPanel.add(panels.get(i), gbc); // Add element with constraint
10   }...}
11 protected void relayout() { // Simple alternative
12   rootPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
13   for (int i = 0; i < panels.size();i++) {
14    rootPanel.add(panels.get(i)); // No constraint needed
15   } ...}
```

The designer of an API usually has some intent as to what and how the API should be used for. It is almost always beneficial to use the API elements intended for the problem at hand rather than trying other API elements and complicating the code. Consider the code snippet in Listing 6 [25]. The `rootPanel` object (`JPanel`) uses `GridBagLayout`. Each cell of the layout can be configured to behave in a certain way using `GridBagConstraints`. The `GridBagConstraints` used in lines 2-6 and 8 is essentially trying to achieve the behavior of a `FlowLayout` with left alignment (lines 12-15) for `rootPanel`. Our critic can detect this use and suggest the much shorter, alternative solution that achieves the same effect as the existing, much more complicated one.

E. Layout Recommendations

Given multiple layout managers, a common problem is to select the right one to use [12]. A recommender can be created to help with this situation. For example, when a container has two components , one of which is a `JTextarea` and the other `JTabbedPane`, we can recommend `BorderLayout` on the basis that the text area is known to be growable [26]. Similarly, we can also recommend `BoxLayout` and `GridLayout` in case the user wants them to grow proportionally, but definitely not `FlowLayout`. Furthermore, if the user wants to partition the container disproportionately, then a `GridBagLayout` can be recommended.

VII. Discussion

In this section, we summarize our observations resulted from this study as answers to our research questions.

As shown in Figure 1, out of 134 layout-related API discussions, 117, or 87.3% can be helped by our critic; there are only 17 truly requirements specific problems that our critic cannot help, since it has no knowledge of the unique user requirements. In addition, the data in Figure 1 also indicate that all three forms of critiques are needed, and that recommendations and explanations are at least equally important as criticisms for supporting programmers.

To be useful, the critic must be equipped with high-quality API usage rules to accurately anticipate a user's goals and address his or her needs. This study supplies the specific API usage rules that can be used in the critic. Interestingly, we find that the discovered API usage rules are nothing more than the all too familiar preconditions, postconditions, and invariants (e.g., the the criticism rules shown in Table I) and, thus, can be enforced by tracing program states and actions. Furthermore, since in retrospect, none of these rules appear to be overly complicated, we conclude that raising programmers' awareness of these rules is the key for improving API usability, and our critic can have a very promising and useful role to play in doing so.

Our study also shows evidence that the discovered API rules have been applied to multiple instances of the same problem. Therefore, the proposed critic can be made general rather than only solve unique individual problems. To help measure how often a problem recurs and an API usage rule can be applied, we have calculated a rate of Return On Investment (ROI) for each of the three kinds of critiques in Tables I, II, and III. When resources are limited, the ROIs can also be used to prioritize the list of API usage rules to determine which subset should be implemented first.

We have found encouraging evidence that our critic can be a valuable complement for humans, despite the deployment of popular online forums and Q&A sites. In particular, our critic has been found to be helpful not only for cases that have been provided a solution but also for those without a solution. In particular, we have found that 44%, or 62, of the 141 clear threads did not contain a solution. Interestingly, 75.8%, or 47, of the 62 threads can be supported by the critiquing rules that we developed in this study, 16 through criticisms, 10 through explanations, and 21 through recommendations. Why were these threads not answered by forum participants? This is most likely because many of them (25 threads) contain a long piece of code and parsing through such long code to find

[24]http://forums.oracle.com/forums/thread.jspa?messageID=5716439

[25]https://forums.oracle.com/forums/thread.jspa?messageID=5729204

[26]http://forums.oracle.com/forums/thread.jspa?messageID=9201990

problems is a cumbersome task. Our critic can be particularly valuable for such cases, complementing human capabilities.

Finally, where do we get rules and what are the characteristics of these rules? At a level higher than the specific rules, we can categorize them into the following common sources:

- Internal consistency rules derived from pre-/post-conditions as well as invariants for an API;
- Common expectations on program behavior, such as the requirement that a text field and a button normally should not grow vertically;
- Requirements for some common user tasks, such as creating a m x n table, which can be inferred from program states;
- Additional solution procedures, or potentially surprising information, that are commonly used together with solutions already present in the API client code.

VIII. Threats to Validity

The results reported in this paper are based on classifying the layout-related discussions in the Swing Forum. Since the layout API has a strong flavor of a tree data structure, other APIs may exhibit different proportions for the three kinds of critiques. Nevertheless, we anticipate that our research method can be applied to study other frameworks and APIs.

In this study, we identified API usage rules and categorized the discussion threads according to whether they can be helped by these rules or not. These are done solely based on our own interpretation of the forum discussions. Since we cannot directly interview the original poster, we have to assume certain facts about the code and sometimes, an OP's intention. Hence, there is a danger that we may have misinterpreted some situations, or we may not have addressed all of the concerns of the original poster in our derived rule set. However, both of these concerns have been mitigated by the good amount of efforts that we have put into this study, by the use of two raters, and by achieving the final consensus between the two raters on the analyzed cases. Furthermore, we want to stress that our past experience with the Swing framework and its forums [7]–[9] should also have helped.

When calculating ROIs for API rules, we have merged a few related rules to simplify the presentation. Nevertheless, the presented ROI gives some insight as to the effectiveness and the applicability of the API rules. It should also be noted that the 150 discussions studied is only a very small subset of all the Swing Forum discussions. There are good reasons to believe that there are more cases in the forum that these rules can be applied to.

IX. Conclusion

To help with using APIs, we are building a critic to advise the usage of an API based on its formal semantics. Specifically, our critic offers advice when the symbolic state of the API client code triggers any API usage rules. To assess to what extent our critic can help solve practical API usage problems and what kinds of API usage rule can be formulated, we manually analyzed 150 discussion threads from the Java Swing forum, from which we created three sets of API usage rules related to the layout logic. We categorized the discussion threads according to how they can be helped by the critic into criticisms, explanations, and recommendations. We illustrate these API usage rules with concrete examples. We find that all three kinds of critiques are useful and justified, API problems of the same nature appear repeatedly in the forum, and that API problems of the same nature can be addressed by implementing a new API usage rule. We describe the nature of the API usage rules and how they can be checked. We also discuss the kind of code behavior inference that is needed in order to make the critic smarter and more powerful and the tradeoffs involved.

Acknowledgment

We thank Ahmed E. Hassan for encouraging us to write up this study, and the three reviewers whose feedback and suggestions help improve the presentation of this paper.

References

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *CHI*, 2010, pp. 513–522.

[2] K. M. Eisenhardt, "Building theories from case study research," *Academy of Management Review*, vol. 14, no. 4, pp. 532–550, 1989.

[3] G. Fischer, "Cognitive view of reuse and redesign," *IEEE Softw.*, vol. 4, pp. 60–72, July 1987.

[4] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Commun. ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[5] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: Finding and leveraging implicit references in a web search interface for programmers," in *UIST*, 2007, pp. 13–22.

[6] D. Hou, "Investigating the effects of framework design knowledge in example-based framework learning," in *ICSM*, 2008, pp. 37–46.

[7] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *ICPC*, 2011, pp. 91–100.

[8] D. Hou, C. Rupakheti, and H. Hoover, "Documenting and evaluating scattered concerns for framework usability: A case study," in *APSEC*, 2008, pp. 213 –220.

[9] D. Hou, K. Wong, and H. J. Hoover, "What can programmer questions tell us about frameworks?" in *IWPC*, 2005, pp. 87–96.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[11] A. J. Ko and Y. Riche, "The role of conceptual knowledge in API usability," in *VL/HCC*, 2011, pp. 173–176.

[12] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *VL/HCC*, 2004, pp. 199–206.

[13] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *CHI*, 2011, pp. 2857–2866.

[14] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *ICSE*, 2011, pp. 111–120.

[15] D. F. Redmiles, "Reducing the variability of programmers' performance through explained examples," in *CHI*, 1993, pp. 67–73.

[16] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Softw. Engg.*, vol. 16, pp. 703–732, 2011.

[17] C. R. Rupakheti and D. Hou, "CriticAL: A Critic for APIs and Libraries," in *ICPC*, 2012, 3 pp. Tool Demo.

[18] B. G. Silverman, "Survey of expert critiquing systems: Practical and theoretical frontiers," *Commun. ACM*, vol. 35, no. 4, pp. 106–127, 1992.

[19] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *VLHCC*, 2006, pp. 195–202.

[20] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *MSR*, 2006, pp. 54–57.