CriticAL: A Critic for APIs and Libraries

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

***Abstract*—It is well-known that APIs can be hard to learn and use. Although search tools can help find related code examples, API novices still face other significant challenges such as evaluating the relevance of the search results. To help address the broad problems of *finding*, *understanding*, and *debugging* API-based solutions, we have built a critic system that offers *recommendations*, *explanations*, and *criticisms* for API client code. Our critic takes API usage rules as input, performs symbolic execution to check that the client code has followed these rules properly, and generates advice as output to help improve the client code. We demonstrate our critic by applying it to a real-world example derived from the Java Swing Forum.**

***Index Terms*—API, Critic, Symbolic Execution, AWT/Swing.**

I. Introduction

Behind each API (Application Programming Interfaces) is a system that offers proven solutions for a set of common problems in some domain. The API facilitates the access to such a system so that new systems can be built on top of it. To be effective in using the API, one must learn enough of the domain, its problems and solutions, and how to map between them appropriately. For a system of rich functionalities with a large problem and solution space, learning its API can be a substantial endeavor [3], [6].

Due to time pressure and an urge to solve problems quickly, many programmers prefer to learn API's on demand and learn by doing. That is, they try to learn just enough of an API so that they can solve the current task. Search tools can partially support this practice by helping locate relevant code examples [1]. But programmers, especially novices, cannot always formulate good queries for what they are looking for. Furthermore, even if they find relevant code examples, they would still face the significant challenge to understand and evaluate them for relevance.

With only limited knowledge of the API, a novice's solution is often incomplete, incorrect or suboptimal. Programmers can seek help from online forums, but there can be some time lag before they can get one. It would be ideal to engage a human expert for help, but experts are scarce resources. To complement, we have developed a critic system [2], CriticAL [1], that can advise the novice online while the code is being written. More specifically, our critic is able to

1) *explain* the interactions of multiple API elements,
2) *criticize* the improper use of the API, and
3) *recommend* other relevant API elements for future use.

Our critic system requires a set of API usage rules and associated documentation to explain these rules. The API rules specify special conditions about program states for which advice should be produced. The critic symbolically executes API client code to obtain program states [4]. Based on the resulting states as well as knowledge of the API usage rules, our critic generates contextual advice. It is assumed that a human expert who has substantive experience with the API will design and implement the rules. [2] It is expected that with multiple interleaved rounds of coding and critiquing, the critic will help to incrementally improve the code. Hence, our critic has the potential to bridge the long-standing information gap between the API designers and the application programmers.

CriticAL is closely related to the Whyline tool [5] in that both tools reason about program states. However, while Whyline answers pre-defined *"why did"* and *"why didn't"* questions by tracking the concrete values of program variables and their causalities through concrete execution, the CriticAL approach offers greater flexibility in terms of defining API usage rules, hence stronger capabilities in criticizing and explaining API behavior. Furthermore, Whyline reasons about one execution trace at a time and, thus, requires multiple runs to have enough branch coverage, whereas CritcAL symbolically executes all possible execution traces in a single run. Although in theory the number of traces can be large and, thus, penalizes the performance of our critic, so far the client code for the studied AWT/Swing API has been simple enough that this potential problem has not become a performance bottleneck. Intuitively, this seems to be consistent with the goal that APIs are designed to simplify programming. Finally, unlike our critic, Whyline does not produce recommendations.

II. An Illustrative Example

To illustrate the three forms of advice (*explanation*, *recommendation*, and *criticism*) that our critic can produce, consider this message copied from the Swing Forum [3]:

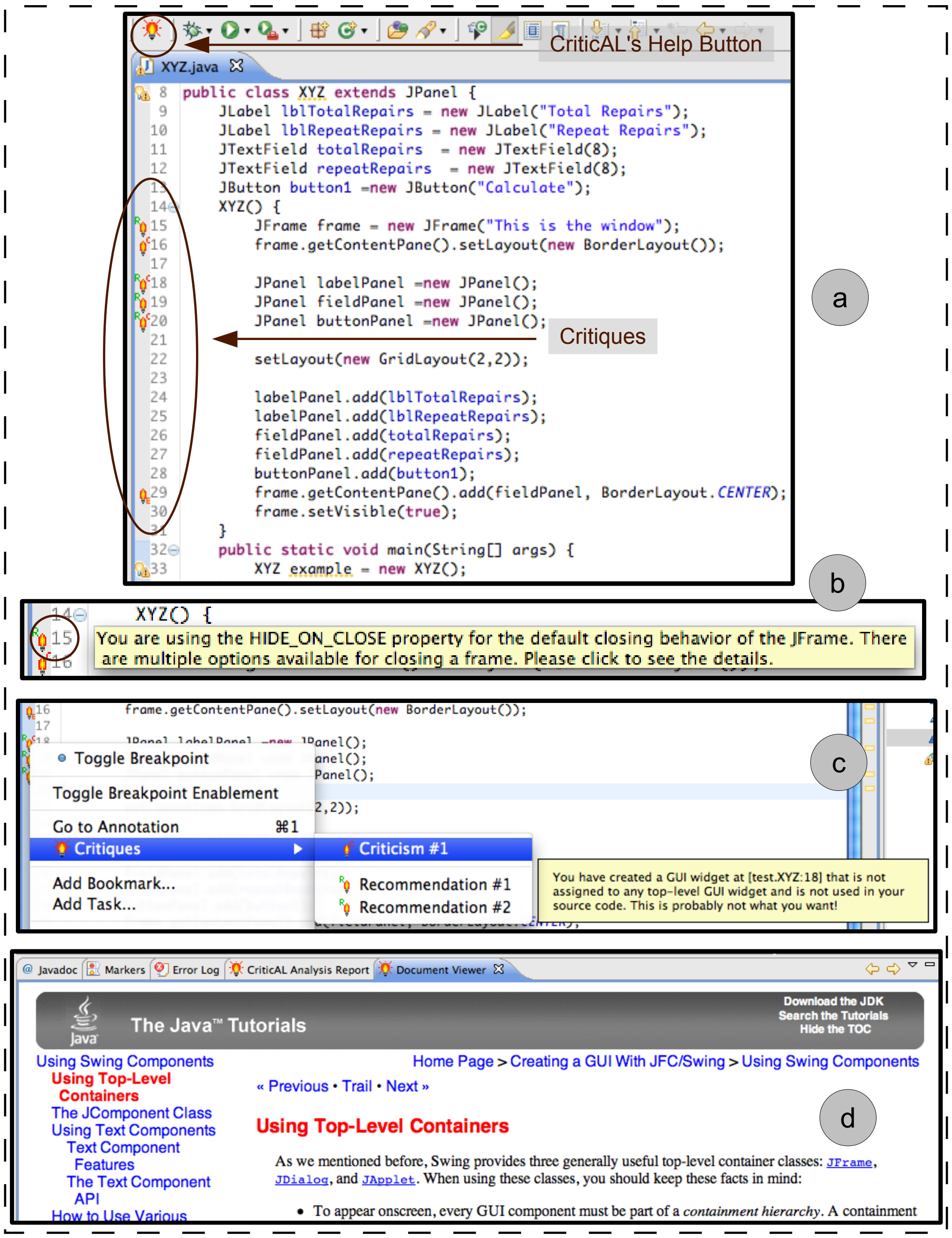
> *Please help me, how to use the Grid layout.*
> *In my code i have to use Grid layout*
> *I have to print the 4 labels in one row. and 4 textfields in the next row [to make] a table.*

Figure 1(a) shows the code mentioned in the above message (for presentation, simplified to contain only two labels and

[1]The **CriticAL** project (A **Critic** for **A**PIs and **L**ibraries) can be found at http://sf.net/p/critical. All URLs verified on 3/28/2012.

[2]Our experience with building a critic for the GUI layout logic is that the cost of building such a useful critic is reasonable [7].

[3]http://forums.oracle.com/forums/thread.jspa?messageID=5737802



```java
public class XYZ extends JPanel {
    JLabel lblTotalRepairs = new JLabel("Total Repairs");
    JLabel lblRepeatRepairs = new JLabel("Repeat Repairs");
    JTextField totalRepairs  = new JTextField(8);
    JTextField repeatRepairs  = new JTextField(8);
    JButton button1 =new JButton("Calculate");
    XYZ() {
        JFrame frame = new JFrame("This is the window");
        frame.getContentPane().setLayout(new BorderLayout());

        JPanel labelPanel =new JPanel();
        JPanel fieldPanel =new JPanel();
        JPanel buttonPanel =new JPanel();

        setLayout(new GridLayout(2,2));

        labelPanel.add(lblTotalRepairs);
        labelPanel.add(lblRepeatRepairs);
        fieldPanel.add(totalRepairs);
        fieldPanel.add(repeatRepairs);
        buttonPanel.add(button1);
        frame.getContentPane().add(fieldPanel, BorderLayout.CENTER);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        XYZ example = new XYZ();
```

You are using the HIDE_ON_CLOSE property for the default closing behavior of the JFrame. There are multiple options available for closing a frame. Please click to see the details.

You have created a GUI widget at [test.XYZ:18] that is not assigned to any top-level GUI widget and is not used in your source code. This is probably not what you want!

Using Top-Level Containers

As we mentioned before, Swing provides three generally useful top-level container classes: `JFrame`, `JDialog`, and `JApplet`. When using these classes, you should keep these facts in mind:

- To appear onscreen, every GUI component must be part of a *containment hierarchy*. A containment

Fig. 1. CriticAL helps a programmer build the Swing application shown in Figure 2.

two textfields). Figure 2 shows the GUI that the current code produces as well as the desired GUI. Using CriticAL for help, the programmer may press the CriticAL button that is shown in Figure 1(a). As a result, CriticAL symbolically executes the code in Figure 1(a) to create the states for the program, which are shown in Table I. Our critic checks the program states against API use rules to infer the current status of the program as well as the programmer's intent and goals for the program, and to offer advice. Generated advice are presented as markers on the ruler of the text editor (the left-hand side of Figure 1(a)), indicating that CriticAL has critiques (E: Explanation, R: Recommendation, and C: Criticism) for the code at the corresponding lines. Because the number of critiques at each line is small, we have not found it necessary to rank them.

A. *Explanation*

The facts holding in program states can be used to help the programmer understand why the program exhibits a certain behavior. For instance, CriticAL finds that a component is added to the center location of the `BorderLayout` that manages

TABLE I
SYMBOLIC STATES FOR THE PROGRAM OF FIGURE 1(A).

Line #	Facts that hold after line # for symbolic objects

14	example.parent = null
	example.layout = FlowLayout()
	example.lblTotalRepairs = JLabel(...)
	example.button1 = JButton(...)

15	frame.title = "This is the window"
	frame.visible = false
	frame.contentPane.children = []
	frame.contentPane.layout = BorderLayout()
	frame.contentPane.layout.properties = []

16	frame.contentPane.layout = BorderLayout()
22	example.layout = GridLayout(2,2)

28	labelPanel.children = [lblTotalRepairs, lblRepeatRepairs]
	fieldPanel.children = [totalRepairs, repeatRepairs]
	buttonPanel.children = [button1]

29	frame.contentPane.children = [fieldPanel]
	fieldPanel.parent = frame.contentPane
	frame.contentPane.layout.properties = [CENTER:fieldPanel]

30	frame.visible = true
	frame.defaultCloseOperation = HIDE_ON_CLOSE
	labelPanel.parent = null
	buttonPanel.parent = null
	example.parent = null

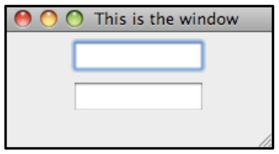
the content pane (line 29, Table I). On hovering over the explanation marker (line 29, Figure 1(a)), CriticAL presents a tool-tip description explaining that the added component will grow with the window as the window is resized. Although IDE's also explain individual API elements by showing Javadoc comments, they do not explain the interaction of multiple API elements. Note that a criticism and a recommendation may also contain explanations.

B. Recommendation

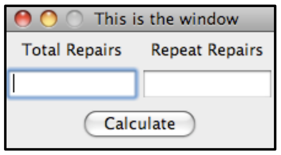
By inferring a programmer's intent from program states and anticipating his or her needs, our critic can recommend both alternative solutions and additional API elements that may be needed next. For instance, by default a `JFrame` has the `HIDE_ON_CLOSE` property set as the default close operation (line 30, Table I). When the user presses the close button of the frame, this property will only hide the frame without actually disposing the frame object. The recommendation at line 15 (Figure 1(b)) presents the user with other available options for the closing behavior, e.g. `EXIT_ON_CLOSE`. The three other recommendations at lines 18, 19, and 20 inform the user how to control the horizontal/vertical gaps and the alignments for the `JPanels` through their `FlowLayouts`.

C. Criticism

Criticisms are produced when the client code violates the pre-/post-conditions and the state invariants of the API objects. For example, to be visible, a non-top-level GUI widget must participate in a GUI hierarchy rooted at a top-level window. Our critic detects that this is not the case for `labelPanel` (created at line 18) and `buttonPanel` (line 20) because their parents are null at line 30 of Table I. Since there are more than one critique (R and C) at line 18, CriticAL offers a context menu for accessing these critiques (Figure 1(c)). On clicking the menu item in the figure, CriticAL presents a detailed



(a) Current GUI



(b) Desired GUI

Fig. 2. The GUI by the program in Figure 1(a) and the desired GUI.

document related to the problem, as shown in Figure 1(d). The document from the Java Swing tutorial is used in this case. In general, every critique has a short tool-tip description and a detailed explanation document, which can be stored either locally or remotely on the Internet.

Assume that to fix the problem, the programmer added the two orphan panels to the frame. Now, at line 30, where the GUI is made visible, the critic detects that `labelPanel` contains two `JLabel`'s as its children, and `fieldPanel` contains two `JTextField`'s (from facts at line 28 of Table I). By examining this symbolic GUI data structure, our critic infers that the programmer is creating a 2-by-2 table. The critic is also able to conclude that this way of making a table is problematic as it will be impossible to properly align a label and its corresponding text field. Instead, such a table can be made in a single container using `SpringLayout`, `GridLayout`, or `GridBagLayout`. These information have been added to the critic as a rule of criticism.

III. Conclusion and Status

We describe an API critic that is designed to significantly improve the experience of learning to use a new API. The key elements for our critic consist of symbolic program states, API use rules, and advising documentation. We have built a critic framework that is capable of symbolically executing API client code. Each API usage rule is implemented as a plugin for this framework. We have identified and implemented a set of API rules for the AWT/Swing layout logic [7]. Future work includes in-depth validation, building critics for other APIs to show the generality of the critic approach, and empirical evaluation of the critics with human subjects. Our working prototype CriticAL for the Java Swing API is open-source.

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] G. Fischer, A. C. Lemke, T. Mastaglio, and A. I. Morch, "Using critics to empower users," in *CHI*, 1990, pp. 337–347.

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

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

[5] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *ICSE*, 2008, pp. 301–310.

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

[7] C. R. Rupakheti and D. Hou, "Evaluating forum discussions to inform the design of an API critic," in *ICPC*, 2012, 10 pp.