Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior

Andrew J. Ko and Brad A. Myers
School of Computer Science, Carnegie Mellon University

Presenter: Shaosong Li
Instructor: Christoph Csallner

Agenda

- Motivation
- Prototype
- Design & implementation
- Evaluation
- Limitation
- Reference
- Q & A

Motivation

- Program understanding and debugging
- Most challenging and time consuming aspects of software development
- Takes up to 70% of the time [17]
- Little has changed in how people work

Suppose you see an inappropriate behavior

- Translate their question
- A series of queries about the program's code
- Guess what code is responsible
- Relationship between the symptoms of a problem and cause

- 90% Initial guesses were almost wrong! [7,8]

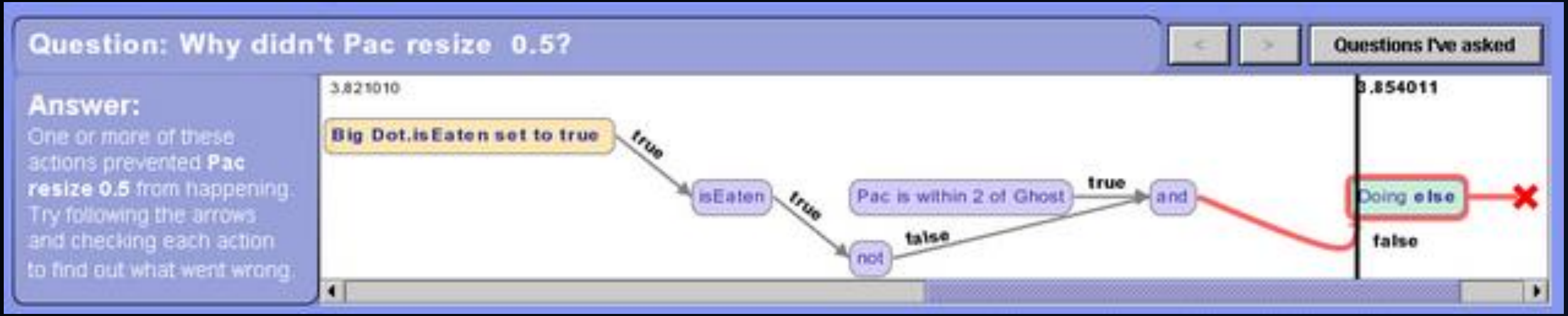
Current tools

- Breakpoint debuggers require people to choose a line of code
- Slicing tools require a choice of variable
- Querying tools require to write an executable expression about data

Limitation

- ‘garbage-in garbage-out’
- Not allow developers to ask why not questions

New Idea: Whyline


Question: Why didn't Pac resize 0.5?

Answer:
One or more of these actions prevented **Pac resize 0.5** from happening. Try following the arrows and checking each action to find out what went wrong.

- Allow developers to choose a why did or didn't question
- Generate answers to the question using a variety of program analyses

http://www.cs.cmu.edu/~NatProg/whyline.html

Earlier prototypes

- Alice Whyline: [8]

  Support similar interaction technique, but for an extremely simple language with little need for procedures and a rigid definition of output

- Crystal framework: [14]

  Support questions in end-user applications, applied same ideas, but limited the scope mostly to questions about commands

Extend and Implementation for Java

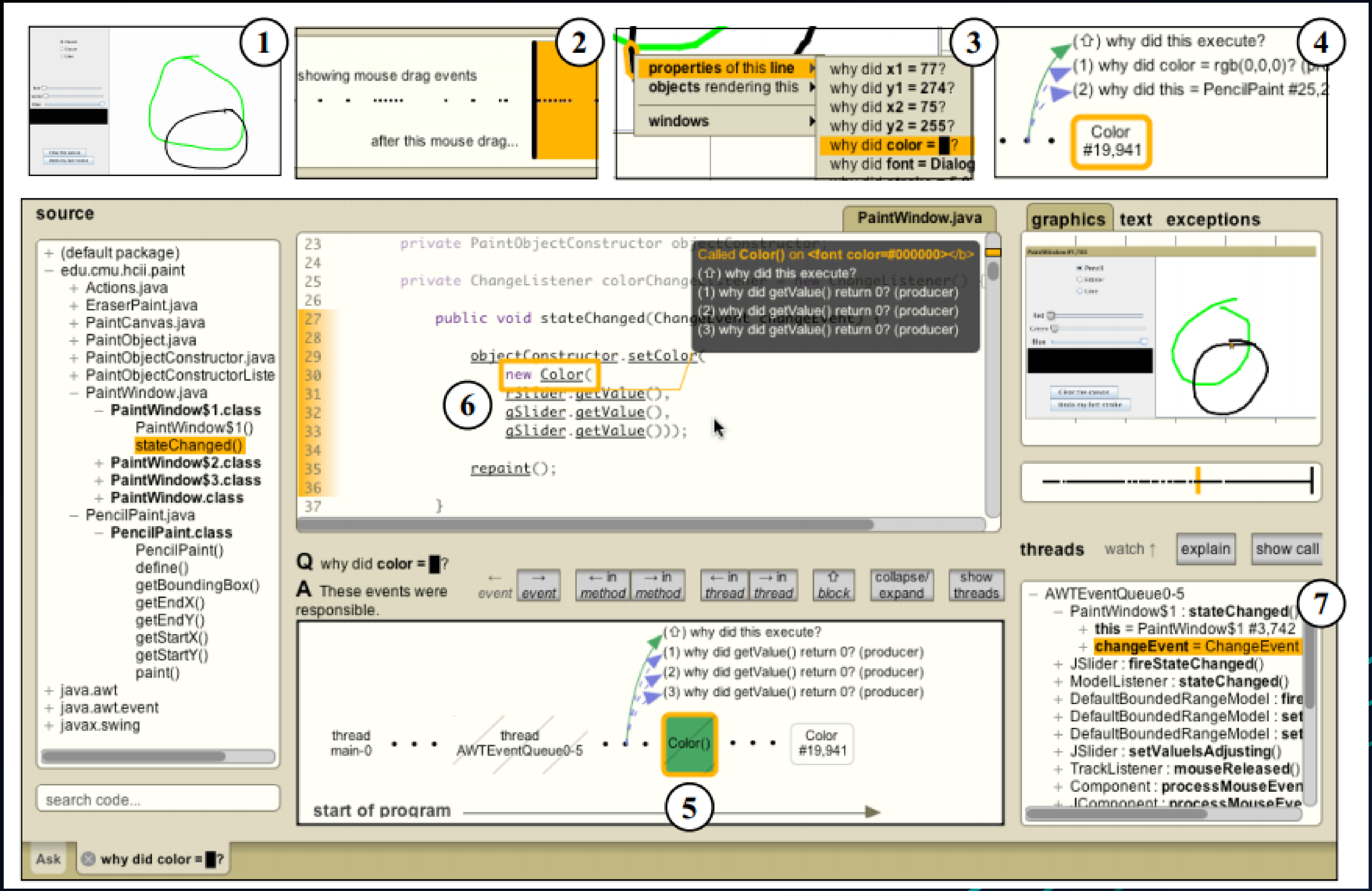
- Algorithms:
  - ➢ deriving questions from code
  - ➢ answering questions
  - ➢ visualization of answers
- Conditions:
  - ➢ standard Java I/O mechanisms
  - ➢ program not run too long

Experiment: A simple painting application

- 500 lines of code
- A few bugs
- One Problem:

RGB color slider did not create the right color

Text searches for “color”
Manually followed data dependencies
Using breakpoints



```
private PaintObjectConstructor objectConstructor;

private ChangeListener colorChangeListener = new ChangeListener() {

    public void stateChanged(ChangeEvent changeEvent) {

        objectConstructor.setColor(
            new Color(
                rSlider.getValue(),
                gSlider.getValue(),
                gSlider.getValue()));

        repaint();

    }
```

Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)

- (default package)
- edu.cmu.hcii.paint
  - Actions.java
  - EraserPaint.java
  - PaintCanvas.java
  - PaintObject.java
  - PaintObjectConstructor.java
  - PaintObjectConstructorListe
  - PaintWindow.java
    - **PaintWindow$1.class**
      - PaintWindow$1()
      - stateChanged()
    - **PaintWindow$2.class**
    - **PaintWindow$3.class**
    - **PaintWindow.class**
  - PencilPaint.java
    - **PencilPaint.class**
      - PencilPaint()
      - define()
      - getBoundingBox()
      - getEndX()
      - getEndY()
      - getStartX()
      - getStartY()
      - paint()
- java.awt
- java.awt.event
- javax.swing

Q why did color = █?

A These events were responsible.

(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)

thread main-0 . . . thread AWTEventQueue0-5 . . . Color() . . . Color #19,941

start of program

- AWTEventQueue0-5
  - PaintWindow$1 : **stateChanged()**
    - **this** = PaintWindow$1 #3,742
    - **changeEvent** = ChangeEvent
  - JSlider : **fireStateChanged()**
  - ModelListener : **stateChanged()**
  - DefaultBoundedRangeModel : **fire**
  - DefaultBoundedRangeModel : **set**
  - DefaultBoundedRangeModel : **set**
  - JSlider : **setValueIsAdjusting()**
  - TrackListener : **mouseReleased()**
  - Component : **processMouseEven**
  - JComponent : **processMouseEve**

1. Demonstrate the behavior user want to inquire about
2. Whyline loads the trace, then the user finds point in time they want to ask about by moving the time controller
3. Click on output and chooses a question
4. Whyline provides an answer, which user navigates
5. In order to understand the cause of the behavior
6. When user selects an event, the corresponding source file shown
7. Call stack and locals at the time of the selected execution event

Design and Implementation

- Intend to support interactive debugging
- Develop new incremental and cache reliant algorithm
- Take a post mortem approach to debugging, capturing a trace [19, 20] and then analyzing it after the program has stopped

Steps & processes

1. Recording an Execution Trace
2. Loading the Trace
3. Creating an I/O History
4. Deriving Questions
5. Answering ‘Why did…’ Questions
6. Answering ‘Why didn’t…’ Questions

Recording an Execution Trace

A Whyline trace of an execution contains:

- Sequences of events that occurred in each thread
- All class files executed
- Source files that represent them
- Other meta data recorded to interpret the data in the trace

Loading the Trace

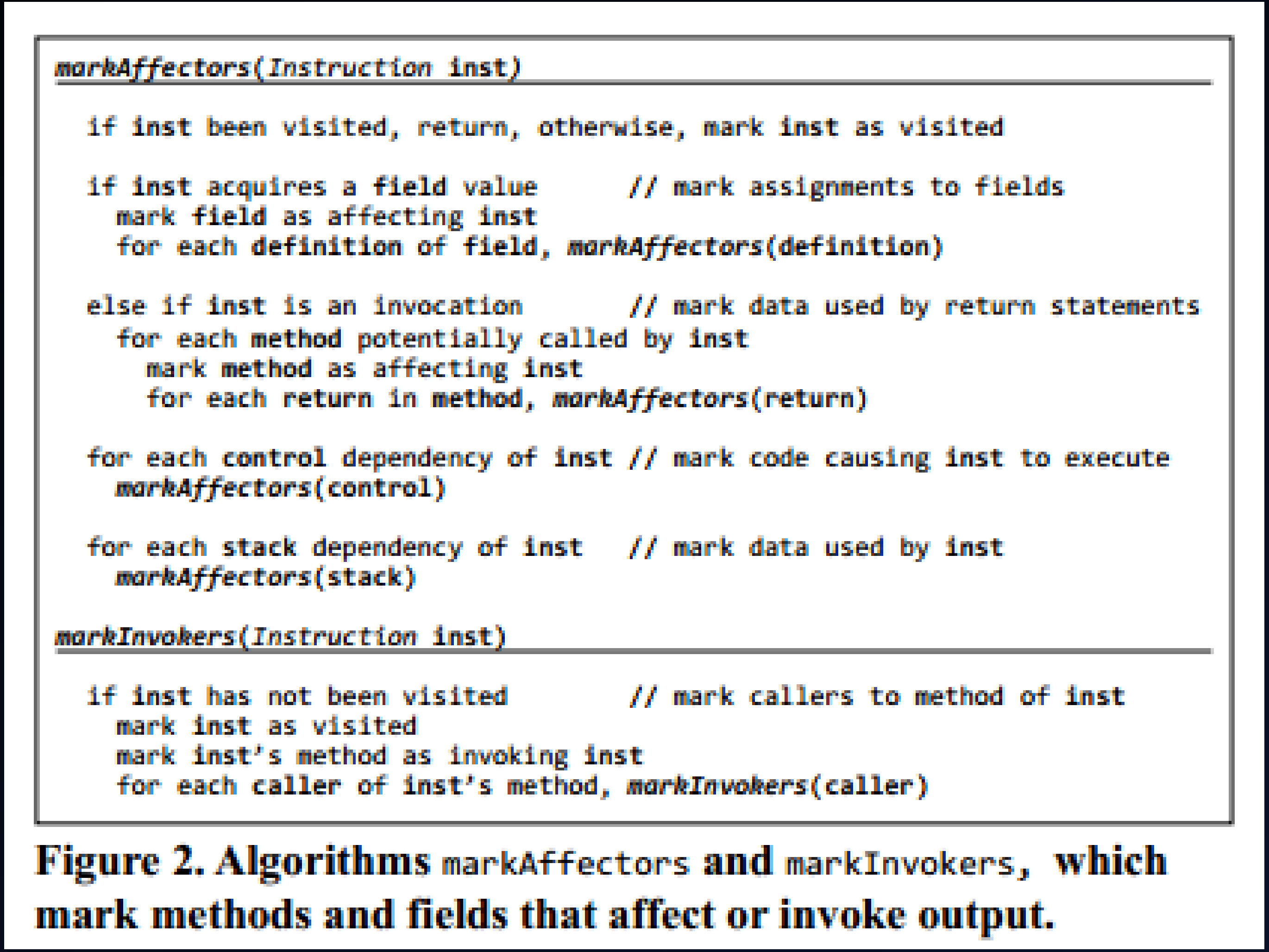
a. Source files and class files are loaded
b. Generate a precise call graph, using all of the invocations found in class files.
c. Loader reads the thread traces, loading events in order of their event IDs

Creating an I/O History

- Prototype assumption: Program uses standard Java I/O interfaces

Java I/O interface	Function
java. awt. Graphics2D	Graphical output
java. awt. Window	Represent windows
java. io. Writer, OutputStream, PrintStream, Reader, InputStream	Console and file I/O
java. lang. Throwable	Exception output

Whyline handles output instructions gathered during class loading. After parsing output, Whyline finds fields and invocations could have affected output



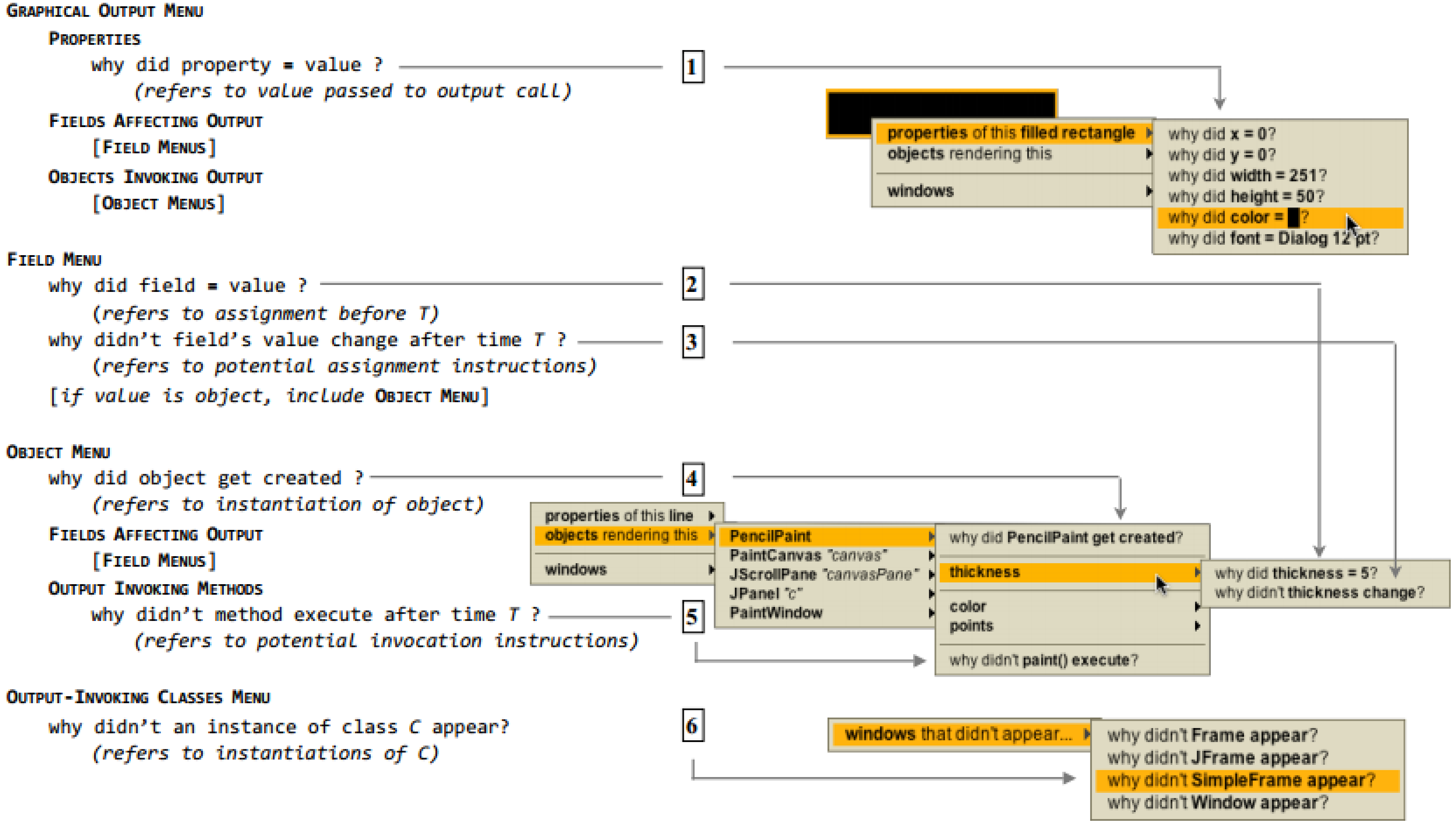
```
markAffectors(Instruction inst)

  if inst been visited, return, otherwise, mark inst as visited

  if inst acquires a field value      // mark assignments to fields
    mark field as affecting inst
    for each definition of field, markAffectors(definition)

  else if inst is an invocation       // mark data used by return statements
    for each method potentially called by inst
      mark method as affecting inst
      for each return in method, markAffectors(return)

  for each control dependency of inst // mark code causing inst to execute
    markAffectors(control)

  for each stack dependency of inst   // mark data used by inst
    markAffectors(stack)

markInvokers(Instruction inst)

  if inst has not been visited        // mark callers to method of inst
    mark inst as visited
    mark inst's method as invoking inst
    for each caller of inst's method, markInvokers(caller)
```

Figure 2. Algorithms `markAffectors` **and** `markInvokers`**, which mark methods and fields that affect or invoke output.**

Insight:
Particular code is responsible for particular output

Deriving Questions

- Why did questions refer to a specific event from a trace
  - Depend on the input time selected by the user(event)
  - Actual depends on the type of output
- Why didn’t questions refer to one or more instructions in the code
  - About variables (discrete-value, other types)
  - About output has no representative output to click on



- GRAPHICAL OUTPUT MENU
  - PROPERTIES
    - why did property = value ? — 1
      - *(refers to value passed to output call)*
  - FIELDS AFFECTING OUTPUT
    - [FIELD MENUS]
  - OBJECTS INVOKING OUTPUT
    - [OBJECT MENUS]
- FIELD MENU
  - why did field = value ? — 2
    - *(refers to assignment before T)*
  - why didn't field's value change after time *T* ? — 3
    - *(refers to potential assignment instructions)*
  - [*if value is object, include* OBJECT MENU]
- OBJECT MENU
  - why did object get created ? — 4
    - *(refers to instantiation of object)*
  - FIELDS AFFECTING OUTPUT
    - [FIELD MENUS]
  - OUTPUT INVOKING METHODS
    - why didn't method execute after time *T* ? — 5
      - *(refers to potential invocation instructions)*
- OUTPUT-INVOKING CLASSES MENU
  - why didn't an instance of class *C* appear? — 6
    - *(refers to instantiations of C)*

Figure 3. The question hierarchy for a graphical output event in the Java Whyline prototype, showing six types of questions currently supported by the prototype (numbered 1-6) and three types of menus. For each, the content on the left lists the meaning of the question (items in []'s represented menus of the specified type) and the content on the right gives an example screen shots.

1. Why did questions relate to the properties of the user selected
2. Why did questions about each of the familiar, output affecting field’s current values
3. Why these fields were not assigned after the selected time
4. Questions about objects that indirectly invoked the selected output primitive, including questions about the creation of the object
5. Questions about output-invoking methods that the user believes did not execute
6. Questions about output that has no representative output to click on

Answering ‘Why did…’ Questions

- Each question maps to an event that is analyzed using dynamic slicing techniques [2], including control [5] and data dependencies
- Produces a causal chain of events, this chain is the tree of events that are traversed in a typical dynamic slicing algorithm[2]
- Each event’s control and data dependencies are computed on demand when a user selects an event. Which means answers are almost produced immediately.

Answering ‘Why didn’t…’ Questions

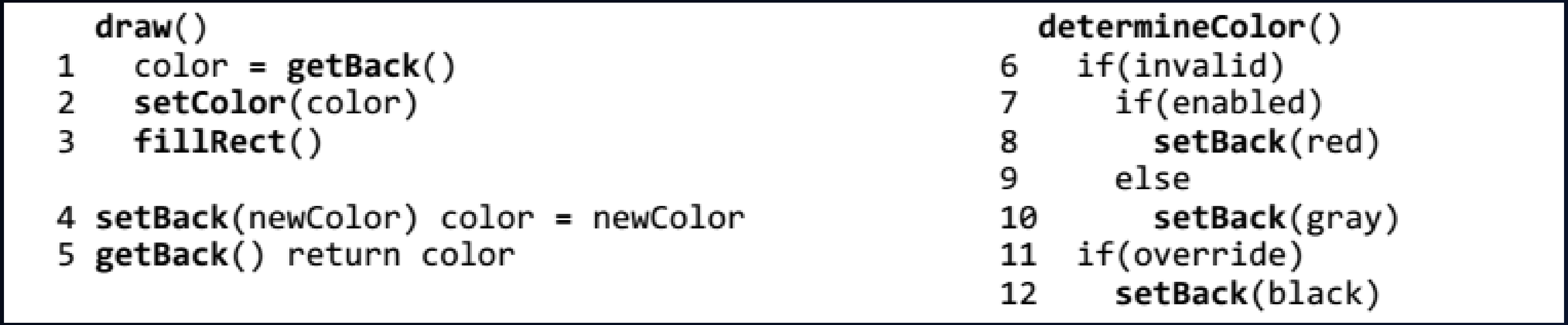
- Whyline handles each of the instructions referred to by the question individually
- Explain individual instruction --- two analyses
  - Determining why an instruction was not executed
  - Determining why a particular dynamic data dependency did not occur
- Constrained by two types of scope
  - Temporal scope: consider events
  - Identity scope : consider objects

Example 1 Why was this instruction not executed?

- First, check if it did execute
- If the instruction did not execute, the Whyline uses an algorithm(which be called whynotreached) to explain why
- Result of the algorithm is a directed graph, with nodes consisting of invocations and conditional instructions that were not reached
- Nodes also have a causal chain of event attached, explaining the source of the wrong object, or the values of the conditional’s expression

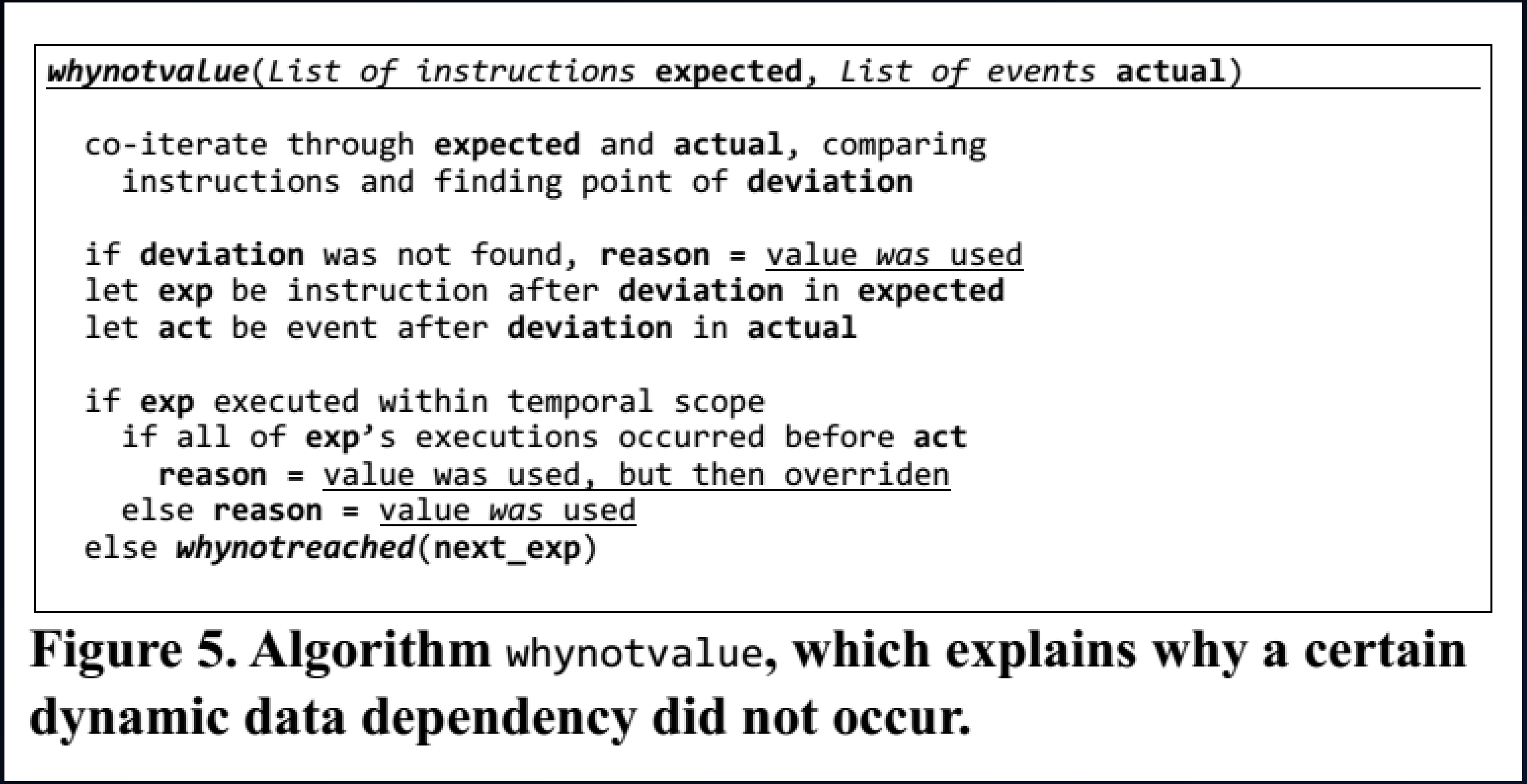
Example 2 Why was this value not used?

- Compare the expected dynamic dependency path to the actual dynamic dependency path at runtime
- Following code, which controls a text field’s background based on various state.



```
   draw()
1    color = getBack()
2    setColor(color)
3    fillRect()

4  setBack(newColor) color = newColor
5  getBack() return color
```

```
    determineColor()
6     if(invalid)
7       if(enabled)
8         setBack(red)
9       else
10        setBack(gray)
11    if(override)
12      setBack(black)
```



```
whynotvalue(List of instructions expected, List of events actual)

  co-iterate through expected and actual, comparing
    instructions and finding point of deviation

  if deviation was not found, reason = value was used
  let exp be instruction after deviation in expected
  let act be event after deviation in actual

  if exp executed within temporal scope
    if all of exp's executions occurred before act
      reason = value was used, but then overriden
    else reason = value was used
  else whynotreached(next_exp)
```

Figure 5. Algorithm `whynotvalue`**, which explains why a certain dynamic data dependency did not occur.**

Presenting answers

- As a navigational aid (help users understand the relationship between events that occurred at runtime and the code that caused them)

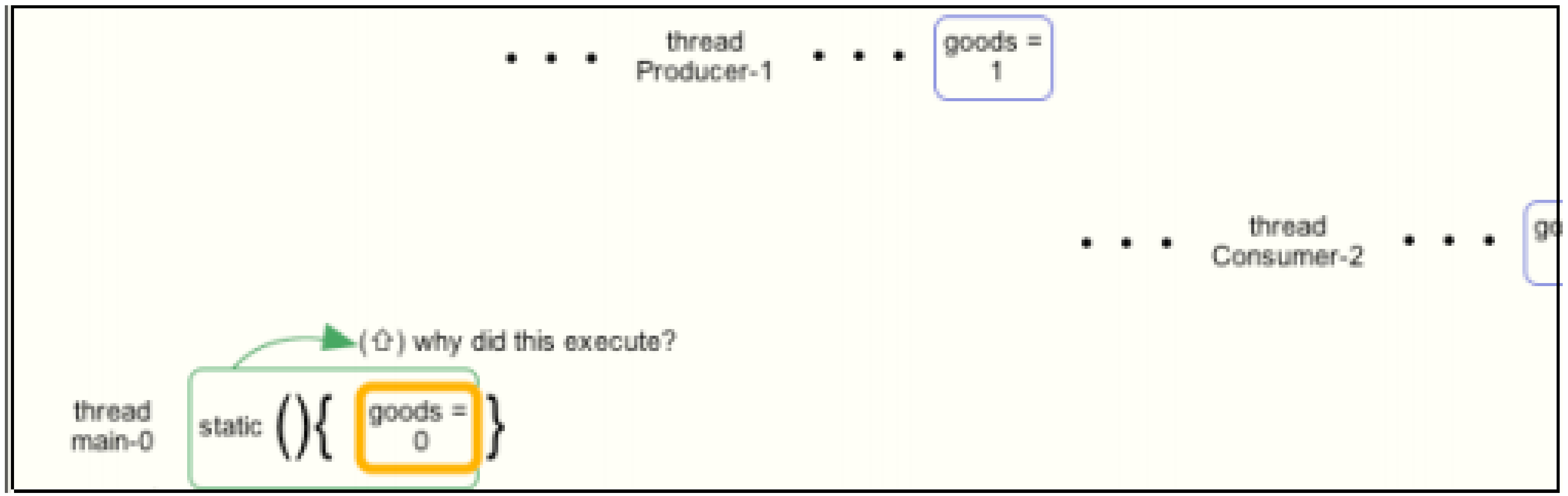
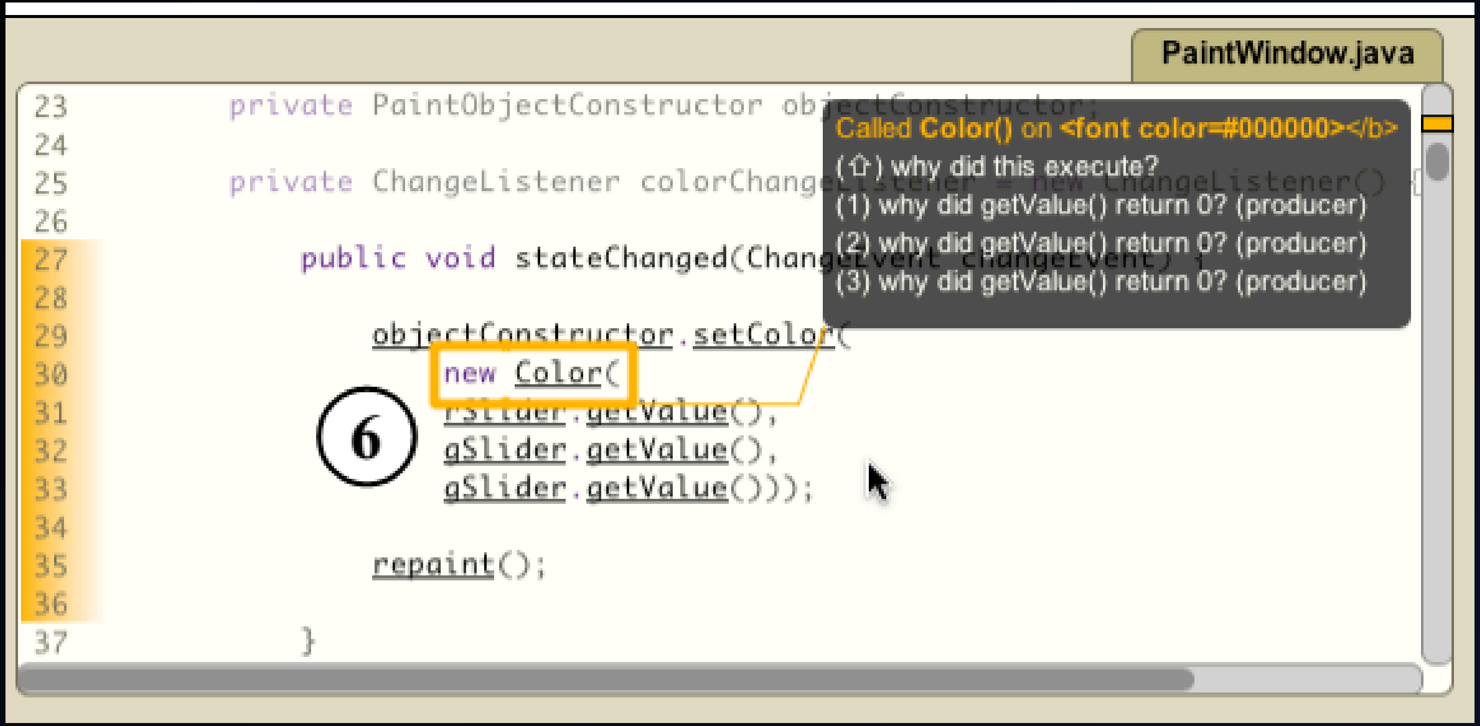

• • • thread Producer-1 • • •
goods = 1
• • • thread Consumer-2 • • •
go
(⇧) why did this execute?
thread main-0
static ( ){ goods = 0 }

Figure 6. Threads separated along the y-axis.



```
        private PaintObjectConstructor objectConstructor;

        private ChangeListener colorChangeListener = new ChangeListener() {

            public void stateChanged(ChangeEvent changeEvent) {

                objectConstructor.setColor(
                    new Color(
                    rSlider.getValue(),
                    gSlider.getValue(),
                    gSlider.getValue()));

                repaint();

            }
```

```
Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
```

Keep the visualization compact

Focus the developer on the code

The Whyline hides information that the user would find unfamiliar or irrelevant.

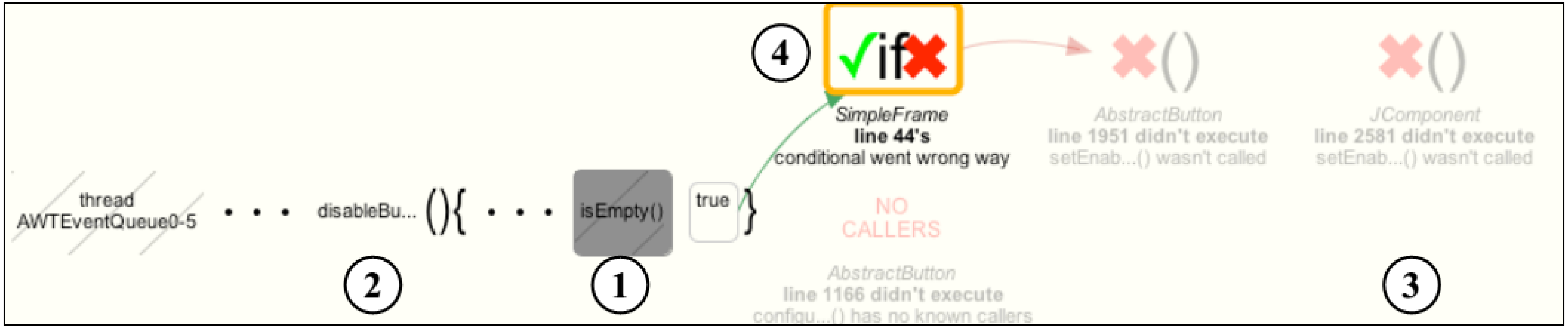
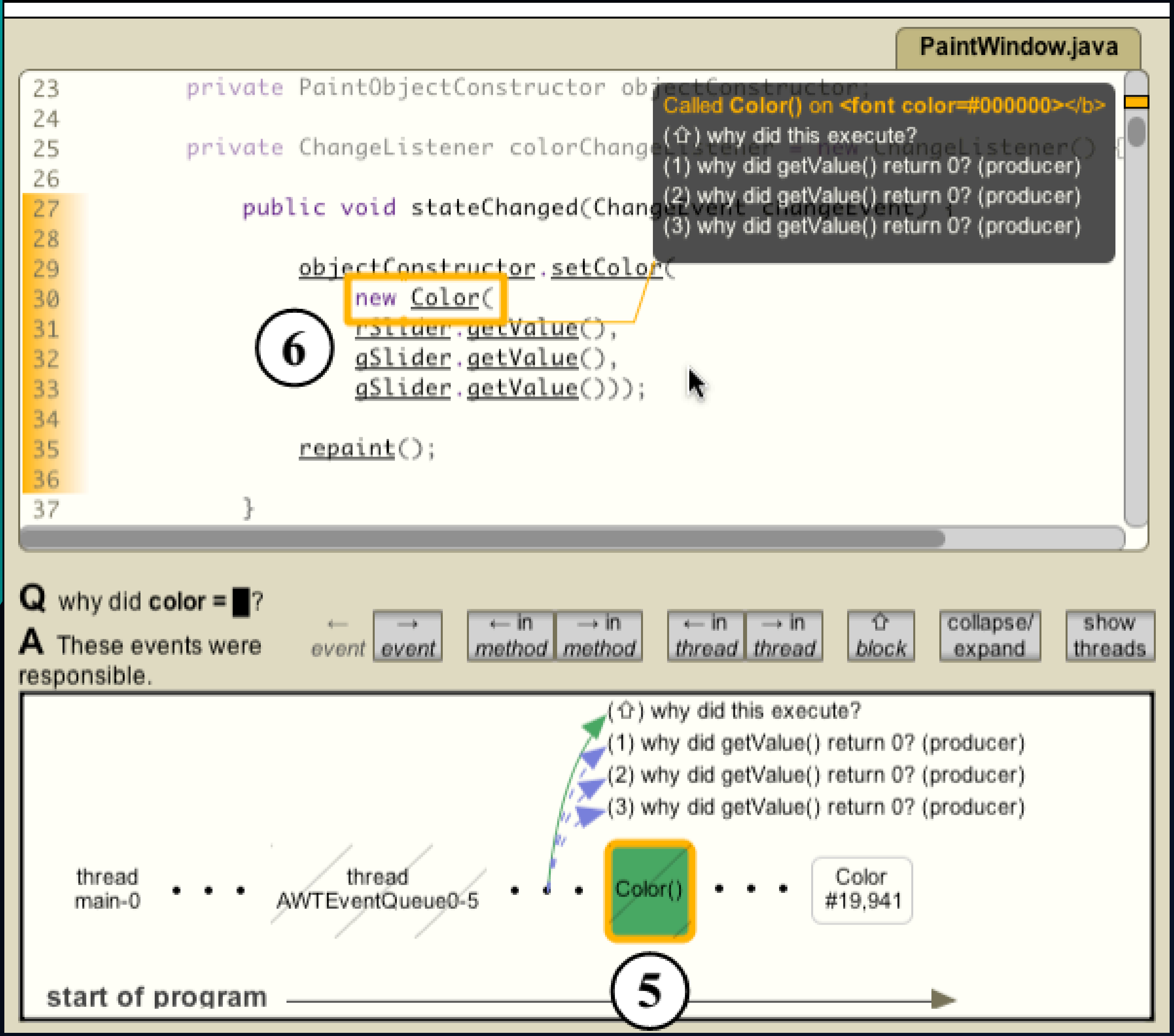


Figure 7. An answer showing (1) a collapsed invocation, (2) a hidden call context, (3) several instructions not executed and (4) a conditional that evaluated in the wrong direction, preventing the desired instruction from executing.



```
    private PaintObjectConstructor objectConstructor;

    private ChangeListener colorChangeListener = new ChangeListener() {

        public void stateChanged(ChangeEvent changeEvent) {

            objectConstructor.setColor(
                new Color(
                rSlider.getValue(),
                gSlider.getValue(),
                gSlider.getValue()));

            repaint();

        }
```

Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)

Q why did **color** = █?
A These events were responsible.

(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)

Provide simple navigational model for exploring an answer

Evaluation -- Performance Feasibility

1. Slowdown (comparing normal running time to tracing time)
2. Trace Size
3. Compressed Trace Size
4. Trace Loading Time

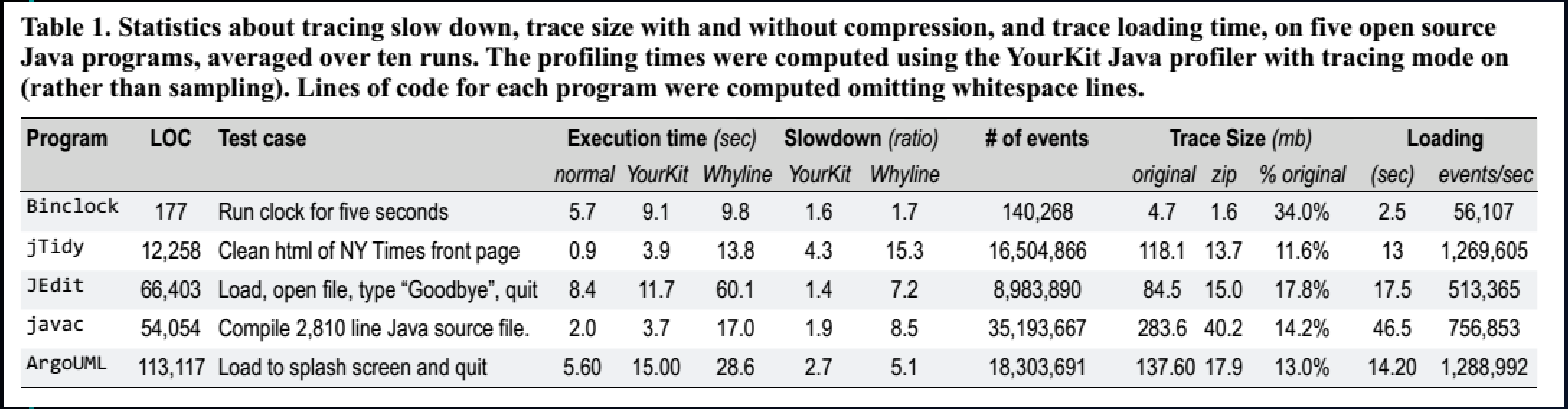


Table 1. Statistics about tracing slow down, trace size with and without compression, and trace loading time, on five open source Java programs, averaged over ten runs. The profiling times were computed using the YourKit Java profiler with tracing mode on (rather than sampling). Lines of code for each program were computed omitting whitespace lines.

Program	LOC	Test case	Execution time *(sec)*			Slowdown *(ratio)*		# of events	Trace Size *(mb)*			Loading	
			normal	*YourKit*	*Whyline*	*YourKit*	*Whyline*		*original*	*zip*	*% original*	*(sec)*	*events/sec*
Binclock	177	Run clock for five seconds	5.7	9.1	9.8	1.6	1.7	140,268	4.7	1.6	34.0%	2.5	56,107
jTidy	12,258	Clean html of NY Times front page	0.9	3.9	13.8	4.3	15.3	16,504,866	118.1	13.7	11.6%	13	1,269,605
JEdit	66,403	Load, open file, type "Goodbye", quit	8.4	11.7	60.1	1.4	7.2	8,983,890	84.5	15.0	17.8%	17.5	513,365
javac	54,054	Compile 2,810 line Java source file.	2.0	3.7	17.0	1.9	8.5	35,193,667	283.6	40.2	14.2%	46.5	756,853
ArgoUML	113,117	Load to splash screen and quit	5.60	15.00	28.6	2.7	5.1	18,303,691	137.60	17.9	13.0%	14.20	1,288,992

Subject programs

1. Binary clock (bincloce)
2. Command line HTML formatter (JTidy)
3. Java compiler (javac)
4. Text editor (jEdit)
5. Diagramming tool (ArgoUML)

Evaluation -- Question Coverage

The degree to which users would be able to find a question that matches the question they want to ask

Approach

1. Randomly sample bug reports on the five applications in table 1
2. Check to see whether any question seemed like a reasonable translation
3. If so, how much translation was required

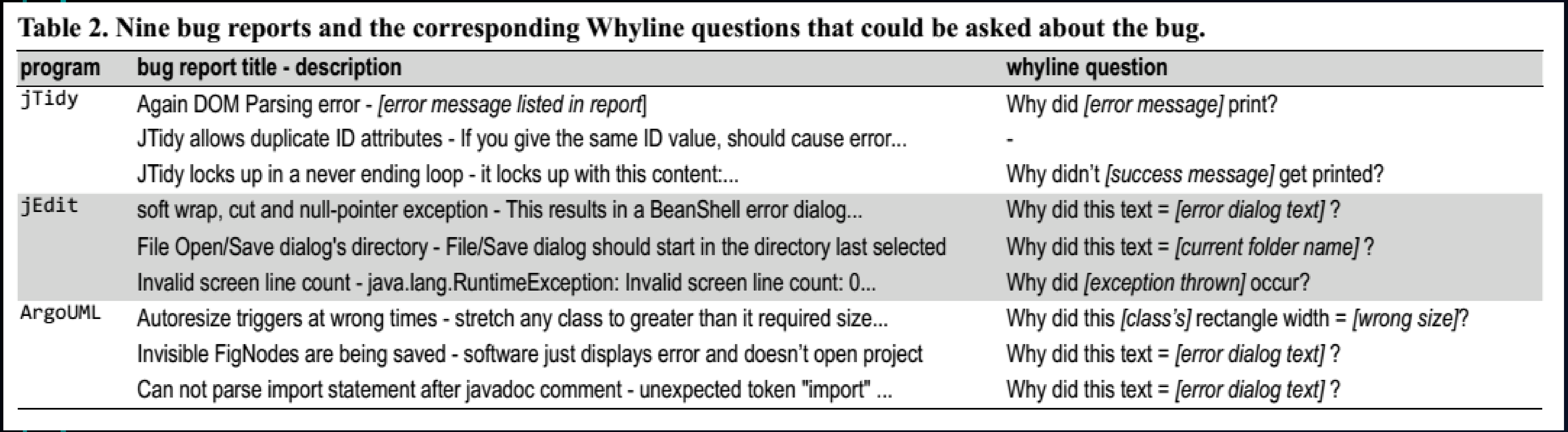


Table 2. Nine bug reports and the corresponding Whyline questions that could be asked about the bug.

program	bug report title - description	whyline question
jTidy	Again DOM Parsing error - *[error message listed in report]*	Why did *[error message]* print?
	JTidy allows duplicate ID attributes - If you give the same ID value, should cause error...	-
	JTidy locks up in a never ending loop - it locks up with this content:...	Why didn't *[success message]* get printed?
jEdit	soft wrap, cut and null-pointer exception - This results in a BeanShell error dialog...	Why did this text = *[error dialog text]* ?
	File Open/Save dialog's directory - File/Save dialog should start in the directory last selected	Why did this text = *[current folder name]* ?
	Invalid screen line count - java.lang.RuntimeException: Invalid screen line count: 0...	Why did *[exception thrown]* occur?
ArgoUML	Autoresize triggers at wrong times - stretch any class to greater than it required size...	Why did this *[class's]* rectangle width = *[wrong size]*?
	Invisible FigNodes are being saved - software just displays error and doesn't open project	Why did this text = *[error dialog text]* ?
	Can not parse import statement after javadoc comment - unexpected token "import" ...	Why did this text = *[error dialog text]* ?

Whyline questions

2 were about console output

5 were about primitive graphical output

1 was about an exception

1 could not find a suitable question

Subject

Evaluation – User Study

Experiment: A simple painting application

- 500 lines of code
- A few bugs
- One Problem:

RGB color slider did not create the right color

- Participants had a variety of backgrounds
- Compared with those people’s task performance with that of 18 self-described expert Java developers from a prior study [10]

Result

Time	Whyline Participants	Control Group
Median(min)	4	10
Range(min)	1--12	3--38

connection

Why is the line blue?

Speculate possible reasons

Limitations

1. Trace based approach
2. Knowledge about a program
3. Programming language
4. Only find code related to a behavior

Reference

[1] Andrew J. Ko & Brad A. Myers.(2008). Debugging reinvented: asking and answering why and why not questions about program behavior, ICSE 08, 301 – 310.

[2] http://www.cs.cmu.edu/~NatProg/whyline.html

QUESTIONS

&

ANSWERS