finding causes of program output with the **Java Whyline**

Andrew J. Ko

Brad Myers
finding causes of program output

one of the most difficult tasks in computer programming
**why** is the stroke *black*?

**why** was this exception thrown?

**why** is the fill rate so big?
why is the stroke black?

“debugging”

find the code responsible
why is the stroke *black*?

**breakpoints**

**reverse execution**

**visualizations**

**program slicing**

*guess* what line is relevant

*guess* where to pause execution

*guess* what to look for

*guess* what code to slice on

find the **code** responsible
what if instead of guessing ...
what if instead of guessing ...
whyline for Java
whyline for Java
**whyline for Alice**

2004

- **small** programs
- **limited** questions
- **static** visualization

**whyline for Java**

2008

- **large** programs
- **flexible** questions
- **dynamic** visualization
whyline for Java 2008

scaling the technology ICSE 2008

scaling the user interface this presentation
whyline for Java
2008

scaling the technology
ICSE 2008

scaling the user interface
this presentation
scaling the **user interface**

asking a why question

asking a why *not* question

evaluation
one bug, two symptoms

a painting program
one bug, two symptoms

Why is this stroke black?

Why didn’t this color panel change?

Why is this stroke black?
why was the line black?
why was the line **black**?
record the problem
record the problem

shareable recording supports collaborative nature of debugging
load the recording

minimal upfront processing since most of recording is irrelevant to question

Resolving classes (856 remaining)
why was the line color **black**?
questions about **output primitives** enable greater **range** of questions, but **sacrifice specificity**
why was the line color black?
why was the line color black?

selected event determines code shown, linking code and execution views
why was the line color black?
why was the line color black?

followup questions about selected event support reverse causal reasoning

(1) why did color = rgb(0,0,0)? (source)
(2) why did this = PencilPaint #25,299? (producer)
why was the line color **black**?

questions shown in both code and **timeline** since developers think in terms of code and execution at different times.
why was the line color **black**?

dependency implied in a hovered question is **highlighted** to reinforce causality and support “peeking”
why was the line color black?

source files layed out automatically based on selected event, avoiding manual opening, layout, and searching
why was the line color black?
why was the line color black?

```java
Stroke oldStroke = g.getStroke();
g.setStroke(new BasicStroke(thickness));
g.setColor(color);

for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex--)
    Point one = points[pointIndex];
    Point two = points[pointIndex];
g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(), (int)two.getY());
```

because gSlider was used twice, ignoring bSlider

```java
public void paintComponent(Graphics g) {
    objectConstructor.setColor(
        new Color(gSlider.getValue(),
                    gSlider.getValue(),
                    gSlider.getValue()));
}
```

```java
public void stateChanged(ChangeEvent changeEvent) {
    color = null;
    repaint();
    // other code...
}
```
why didn’t the panel repaint?
why didn’t the panel repaint?
find the appropriate time
find the appropriate time
find the appropriate time

time slider supports explicit selection of input and output events
find the appropriate time

filter buttons ease selection of specific types of input events
click on relevant output
click on relevant output

objects indirectly related to selected rectangle allow questions about higher-level visible entities

questions about data and behavior of selected objects sorted by type, filtered by familiarity
it did paint...

why didn’t questions look for executions immediately after selected time on selected object, since developers questions tend to refer to specific input events and objects in memory
it did paint...

this method did execute!

allows why didn’t questions about things that did happen in order to reveal assumptions
where did black come from?
where did **black** come from?

stepping commands allow precise, familiar control over execution history

**step forward to setColor() call**
where did **black** come from?

precise token-level highlighting allows explicit reference to runtime events
where did **black** come from?

timeline gathers “visited” events, acting as combined explanation, navigational aid, and bookmarking tool.
where did black come from?

notation mimics Java syntax to group events that occur within the same method

vertical rows indicate separate execution threads
where did **black** come from?

why did `getColor()` return **black**?
why did getColor() return black?

Q: why didn't paintComponent() execute?
A: Check the answer below.
found the **bug**!

```java
private JComponent currentColorComponent = new JComponent() {
    public void paintComponent(Graphics g) {
        Color oldColor = g.getColor();
        g.setColor(objectConstructor.getColor());
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(oldColor);
    }
};

public PaintWindow(int initialWidth, int initialHeight) {
    super("Paint");
    actions = new Actions(this);
```
it works well for these **simple bugs**...

does it work well for **trickier bugs**?
a comparison study

Whyline group vs control group

both groups had modern IDE features
show declaration, show callers, show references, etc.
the conventional debugger

simulated with a Whyline trace

**supported**
- breakpoints
- step **into/over/out**
- run to breakpoint/line
- pause at selected program output
- print variables value
- see call stacks

**unsupported**
- pausing live program
- editing live program
- arbitrary print statements

---
subject program

ArgoUML, an open source software design tool

~150,000 lines of code

22 external libraries

chose two bug reports from version 18.1

- one w/ simple fix
- one w/ complex fix
task one

ArgoUML bug 3121: “Remove ‘Report Usage Statistics’ since it does not do anything”
task two

ArgoUML bug 3128: “Problems with two classes with the same name in different packages”
Task 2

Sharon needs your help again, this time on a more complicated bug. In the screen shot below, there are three classes. Two have the same name, but are in different packages. A third class has an attribute. The user is trying to select the attribute's type, but notice that the list only includes "MyClass". It should include two classes named "MyClass," one from each package.

Again, you have two responsibilities.

1. Find out why this menu only has one item labeled "MyClass".
2. Write a change recommendation, detailed enough for Sharon to understand the cause of the problem. You should also include at least one idea for a solution to the problem.

You should be confident about the correctness of your recommendation (you don't want to waste Sharon's time), but you shouldn't spend too much time understanding the system, since you're on a tight schedule and this is not your code. You can afford to spend up to 30 minutes on this bug.

When 5 minutes remain, I will remind you about writing the recommendation. Tell the experimenter when you are done with your recommendation.
participants

20 masters students in software engineering

- all non-native English speakers
- 0-10 years in industry, median 1.5 years
- average self-rated Java expertise ("beginner" to "expert" scale)

groups did not significantly differ on any measures
task 1 results

whyline  control

# successful

minutes

more successful in half the time

p<.05
task 2 results

- whyline
- control

Number of successful:
- 4 of 10 gave up

Minutes:
- p<.05

more successful in the same time
subjective observations

users used the timeline to capture a distillation of relevant code for a problem

users preferred less-specific “why did” questions over more-specific “why didn’t” questions because of faster response

if the first few events in an answer didn’t seem relevant, users would ask a different question
subjective observations

Whyline users were **more thorough in reading code** presented by the Whyline

- control group participants deemed code irrelevant more quickly

Whyline **helped localize** bugs, but not in explaining **inherently complex code**

- finds the buggy code, but not the bug
unprompted opinions

“This is great, when can I get this for C?”

“My god, this is so cool.”

“This is really great!”

“This is really going to reduce the burden on programmers.”

“It's so nice and straight and simple...”
conclusions

today’s tools require guessing, costing time and money

the whyline limits guesswork by supporting questions about output

the whyline saves time, improves success rates
This work was supported by the National Science Foundation under NSF grant IIS-0329090 and the EUSES consortium under NSF grant ITR CCR-0324770. The author is also supported by an NDSEG fellowship and by a NSF Graduate Research Fellowship.

download the Java whyline at faculty.washington.edu/ajko/whyline-java.shtml or Google “whyline Java”

Andrew J. Ko
Brad Myers
This work was supported by the National Science Foundation under NSF grant IIS-0329090 and the EUSES consortium under NSF grant ITR CCR-0324770. The author is also supported by an NDSEG fellowship and by a NSF Graduate Research Fellowship.

download the Java whyline at faculty.washington.edu/ajko/whyline-java.shtml or Google “whyline Java”

Andrew J. Ko  Brad Myers