My name is Andrew Ko, and today I’ll be talking about some work that I’ve done on debugging as part of project Marmalade.
Interrogative Debugging

A new approach to debugging tools for programmers of all expertise.

The Whyline, an end-user programming interrogative debugging prototype.

- What is Interrogative Debugging?
- What is the Whyline?
  - Demonstration
  - User Study
- Conclusions
Among all programming activities, debugging is still the most common and costly.

- Software engineers spend 70-80% of their time testing and debugging, with the average bug taking 17.4 hours to fix.

- Aspiring end-user programmers often abandon programming systems because of difficulty with debugging.

But we’ve had the same debugging tools for over 30 years...

- Print statements are still the best.

- Several research prototypes have been proposed, but few have been shown to reduce debugging time or effort.

Why is debugging so hard? And why have so many tools proven ineffective?
Debugging

Understanding Debugging

- We asked 17 expert, novice, and non-programmers to think aloud while programming.

- We found several striking trends:
  - Programmers asked explicit questions about their program’s failures, immediately after failure.
  - All were in the form **why did** (32%) or **why didn’t** (68%).
  - Questions were always in terms of visual output, and never in terms of code and data.
  - i.e., “Why didn’t my object disappear?”
Assumptions

- Behind every question were implicit assumptions about what did or did not happen at runtime.

- Almost 80% of questions made false assumptions:

  Q: Why didn’t that disappear?

  A: It did disappear, but then immediately reappeared.

- 50% of all of programming errors were due to decisions programmers had made because of false assumptions.

Even more striking...

These observations led to approach to debugging interfaces, which we call...
Interrogative Debugging

1. Allow *why did* and *why didn’t* questions about a program’s behavior.

2. Reveal false assumptions about what did and did not occur at runtime.

3. Give answers in terms of the *code and data directly responsible* for the behavior in question.

...Interrogative Debugging.

It has three parts.

Will this work? To find out...
We prototyped an Interrogative Debugging tool called the Whyline, in the Alice 3D programming system...

Let me briefly describe the Alice programming system.
The Whyline

Alice

- A 3D animation end-user programming system.
- Event-based, imperative, and concurrent.
- Drag and drop interaction prevents all type and syntax errors.

A free gift from Randy Pausch, et al. at Carnegie Mellon's Entertainment Technology Center:

http://www.alice.org
What is the Whyline?

Question Menu

Textual Answer

Answer Visualization
The Whyline

What is the Whyline?

Question Menu

Textual Answer

Answer Visualization
The Whyline

The Question Menu

Contains *why did* and *why didn’t* questions about a program’s behavior.

Questions are organized by the *object* they relate to...

...then by *animation* and *variable*.
The **textual answer** reveals assumptions and summarizes what did or didn’t happen at runtime.

**Question:** Why didn't Ghost move 2 toward Pac did happen. Maybe it looked like it didn't for some reason?

**Answer:**
Actually, *Ghost move 2 toward Pac* did happen. Maybe it looked like it didn't for some reason?

**Question:** Why didn't Big Dot isShowing change to false from happening. Maybe one of the variables in your expressions had value you didn't expect?

**Answer:**
One of these conditions prevented *Big Dot isShowing change to false* from happening. Maybe one of the variables in your expressions had value you didn't expect?

**Question:** Why didn't Pac move forward 3?

**Answer:**
*Pac move forward 3* only happens when *Pac.movePac* happens, but there's nothing that makes *Pac.movePac* happen *Pac.movePac* in one of your events?
What is the Whyline?

- The **answer visualization** is an interactive timeline.
- Shows data and control flow directly related to a question.
We’ll recreate three scenarios directly from user studies.

In our studies, programmers created a simple Pac-Man game.

Pac tries to eat the dots and the Ghost tries to eat Pac.

We’ll watch a programmer:

1) Make Pac move.
2) Make Pac shrink if eaten by the Ghost.
3) Make Pac invincible if he eats the Big Dot.
Demonstration

Make Pac Move

Make Pac move forward 3 meters/second, as seen below.
The programmer creates the `movePac()` method.
The programmer creates a *move forward* animation and tests.

Before we see the use of the Whyline, how have users in prior studies reacted to this failure?
Programmers wondered, **why didn’t Pac move forward?**

- *Is there something wrong with my move command?*
- *Maybe forward doesn’t mean what I think?*
- *Was I supposed to save before I played the world?*

Experts, novices and non-programmers spent **2 - 3 minutes** diagnosing this failure.

Every programmer made this type of error at least **twice** in our observations.

With the Whyline, however...
Why didn’t Pac move forward 3?
The programmer clicks the question and the answer appears.
The programmer then inspects the answer.
The programmer makes an event that calls `movePac()`.

**Question:** Why didn't Pac move?

**Answer:**
Pac move forward 3 only happens when Pac in one of your events?
Invariant Answers

- An example of an **invariant** answer.
- Pac would **never** move because the programmer forgot to call `movePac()`.
- The Whyline revealed this problem, directly diagnosing the failure.

2-3 minute task ➔ 10 second task!

Reduced by a factor of 15!
Demonstration

Make the Ghost Eat Pac

If the Ghost touches Pac, make Pac shrink.
The programmer wraps a *DoTogether* around *move*.

The programmer wants to move pac and test the collision at the same time.

He does this with a do together, which executes statements concurrently.
He adds an *If-Else*, checking if *Ghost* is within 2 meters of *Pac*. 
The programmer adds a *Pac resize 0* animation and tests.

We want Pac to shrink, so we'll resize him to 0 if he touches the ghost.
Demonstration

Without the Whyline

- Programmers wondered, why didn’t Pac resize?
  - Maybe “is within threshold” doesn’t do what I thought...
  - Maybe I shouldn’t have used the resize animation...
  - Maybe I wasn’t supposed to put that in the DoTogether...

- Experts, novices and non-programmers alike spent 4 - 7 minutes diagnosing this failure.
Why didn’t Pac resize 0?

Again, the code in question becomes selected as the programmer hovers over the questions.
The answer visualization shows what caused the `resize`.

So the resize did happen...
He scrubs the history, verifying that `resize 0` had no effect.

The time cursor moves backward and forward through the execution history, changing the output window.

He scrubs the history, verifying that `resize 0` had no effect.
The programmer changes `resize 0` to `resize .5`, and tests.
False Proposition Answers

- An example of a *false proposition* answer.
- The programmer made the reasonable assumption that `resize 0` would have a visible effect.
- When it didn’t, he assumed that it did not occur.
- The Whyline revealed this assumption, allowing him to isolate the problem to the `resize` animation.

4 - 7 minute task ➞ 30 second task!

Reduced by a factor of 11!
Demonstration

Make Pac Invincible

If the Big Dot has been eaten, prevent Pac from shrinking.
The programmer makes *Big Dot* disappear if *Pac* touches it.
Pac should only resize when the Big Dot is visible.
Without the Whyline

- Programmers wondered, **why did Pac resize?**
  - I have no idea what just happened....
  - Maybe my If-Else is wrong...
  - Maybe Pac didn’t get to the dot in time...
  - Maybe isShowing doesn’t do what I think...

- Experts, novices and non-programmers alike spent **4 - 9 minutes** diagnosing this failure.

Don’t bother reading these.
Why did Pac resize .5?
Big Dot.isShowing was true when the Ghost touched Pac.
Why didn’t `Big Dot.isShowing` change to false?

Asks a question in context, which adds the answer to the Whyline.

You can see that the condition was tested before the value was changed.
The two conditionals were interleaved.
The Big Dot check should happen before the Ghost check.

Because the conditionals were in the same do together, they became interleaved. He fixes it by dragging the Big Dot check outside the do together.
Control & Dataflow Answers

- An example of a *control and dataflow answer*.
- The programmer assumed that the **Big Dot check** would finish executing before the **Ghost check**.
- The Whyline revealed this assumption, illustrating that the two conditionals were interleaved.

4 - 9 minute task ⇒ 30 second task!
Reduced by a factor of 13!
The Whyline records each execution of code and each assignment and use of data.

We use this information to construct the question menu...
The Question Menu

- A question for each unique execution of an output statement.
- Does not contain what didn’t happen.
  - Programmers rarely assumed something did happen when it actually didn’t.

- A question for each output statement that could have executed...
  - Only need to include what programmers expect to happen (what they wrote).

- A question for everything that did happen, allowing for false propositions.

Infinite number of things that could have happened, but you only have to include ones that programmers would expect (they expect what they wrote).

Why did doesn’t contain what didn’t happen for false propositions, because these were never asked in user tests.

To answer the questions...
Implementation

The Question Menu

- A question for each *unique execution* of an output statement.
- **Does not contain what didn’t happen.**
  - Programmers rarely assumed something did happen when it actually didn't.
- A question for each output statement that *could have* executed...
  - Only need to include what programmers expect to happen (what they wrote).
- A question for everything that *did* happen, allowing for false propositions.

Infinite number of things that could have happened, but you only have to include ones that programmers would expect (they expect what they wrote).

Why didn't contain what didn’t happen for false propositions, because these were never asked in user tests.

To answer the questions...
We do answer invariant.

Now let’s briefly discuss our user studies of the Whyline.
Expert, novice, and non-programmers were given 90 minutes to implement the Pac-Man game.

- The game required programmers to implement 6 distinct behaviors.

Programmers were asked to think aloud, and were videotaped over the shoulder.

Participants were all novice to Alice, and expressed interest in learning to program 3D simulations.

We saw 3 simplified versions of these behaviors in the demo.
User Study

How Often Was It Useful?

- Programmers used the Whyline for all 24 of their debugging questions.

- 19 of 24 answers (80%) led directly to valid repairs.
  - In each, programmers had made false assumptions about their program’s pattern of execution.

- 5 of 24 answers failed to help programmers with their debugging task.
  - In each, programmers already knew what had occurred, but didn’t know why it was wrong.
  - All concerned complex Boolean expressions.

Say 80% and 20% here.

So we’re thinking about how to address this with other stuff
User Study

Debugging Time

- Reduced debugging time by an average factor of 7.8 ($p < .02$).

In terms of task completion,...
User Study

Task Completion

- The **Whyline prevented further errors** by preventing programmers from editing code that was not at fault.

- As a result, the group of programmers with the Whyline **correctly completed 40% more tasks** ($p < .02$)

![Bar chart showing the average number of behaviors correctly implemented with and without the Whyline.](chart.png)
Conclusions

Discussion

- **Interrogative Debugging** was highly effective.

- When programmers made false assumptions about what occurred at runtime (80%), the Whyline revealed them.
  - Programmers were able to use the Whyline’s answer to quickly and successfully correct their code.

- Even when programmers knew what had occurred at runtime, but didn’t know why it was wrong (20%), the Whyline helped them focus on the code responsible.
Future Work

The Whyline raises several questions about Interrogative Debugging:

- Does it scale to larger, more complex programs?
- Would it help more experienced programmers?
- Would it help comprehend unfamiliar code?
- What types of questions are important in other domains?
- How should answers be visualized in other domains?
- Can questions involving “before” and “after” be answered effectively?

Look forward to working on several of these questions for the foreseeable future.
Q & A

Project Marmalade
http://www.cs.cmu.edu/~NatProg/marmalade.html

EUSES Consortium
http://eecs.oregonstate.edu/EUSES/

Project Marmalade is funded by NSF grant IIS-0329090 and in part by the EUSES consortium (End Users Shaping Effective Software) under NSF ITR CCR-0324770.

Thank you very much to NSF and EUSES and thank you for being here here.

Q: Barriers to Java implementation?
   A: What are the relevant questions? Efficiency.

Q: Does it scale?
   A: