Lecture 9

CSE 15L
Winter 2009
Scott B. Baden
UCSD

Questions about the course

- Difficulty of assignments
- Format
- Text

The Final Exam – Where and When

- Your final exam will be held in on Monday, March16th
  11:30 AM to 2:30 PM
  in SOLIS 109
- Closed book, closed notes
- IDs will be checked

The Final Exam - what

- You are responsible for all material covered in the course
- Assigned readings
- The labs
- Discussions in class
- Format
  - Matching
  - Short answers
  - Examine code, formulate hypotheses, predictions, identify bugs

The Scientific Debugging Process

0. **Characterize the intended behavior** of the code.

For each deviation from the intended behavior:

1. **Characterize a failure,** a deviation from intended behavior (*not a defect* you notice in the code). .. specify how the behavior deviated .. is it a compile-time, run-time, or semantic error?

How does a failure differ from a defect?

- A failure occurs when a program does not behave as intended
- A defect is a problem in the design or implementation that causes the failure

Scientific Debugging steps (II)

2. **Form an hypothesis**
   - Attempts to understand what is happening
   - Might speculate about conditions that lead to the failure
   - Identify an apparent defect in the code that caused the failure
   - Record each hypothesis even if you discard it later; invalidating hypotheses is an important part of the process.

3. **Make a prediction**
   - A scientific prediction is based on a hypothesis.
   - Hypotheses are based on what you've already observed
   - Predictions are what you expect to observe in an experiment

A prediction is distinct from a hypothesis

What is an hypothesis?

- Merriam-Webster says
  http://www.merriam-webster.com/dictionary/hypothesis
  - an interpretation of a practical situation or condition taken as the ground for action
  - a tentative assumption made in order to draw out and test its logical or empirical consequences

How does a prediction differ from an hypothesis?

- Hypotheses are based on what you've already observed
  - Might speculate about conditions that lead to a program failure
  - But more general: doesn't always lead to finding a bug, but rather, helps explain how the program works
- Predictions are what you expect to observe in an experiment
  - Attempts to understand what is happening
  - Predictions may not necessarily lead to a bug fix
- An hypothesis leads to experimentation, may result in a new hypothesis, etc.

Scientific Debugging steps (III)

4. **Perform an experiment**

   Change the conditions and compile or run again.

   Give the program different input,
   add debugging output, or alter the code

5. **Observe the results.** What is the behavior now?

6. **Reach a conclusion.**

   Did the results match your prediction?

   Do the results suggest a new hypothesis?

An example of the scientific process

```
Lab2.java:25: cannot find symbol
symbol  : constructor InputStreamReader(java.io.PrintStream)
location: class java.io.InputStreamReader
        InputStreamReader isr = new InputStreamReader(System.out);
```

`InputStreamReader(InputStream in)`

"Creates an `InputStreamReader` that uses the default charset"

Compile error at line 25. The code did not compile, and thus could not be run to achieve the desired output

After consulting the java api documentation, we **hypothesized** that the failure was caused by attempting to instantiate an InputStreamReader with System.out as a constructor parameter

We **predicted** that if we changed the System.out parameter to System.in, which makes more sense for an object that reads input, the failure would be fixed.

After changing System.out to System.in, we recompiled the program

We observed that the compile error did not reoccur

The results matched our prediction and validated our original hypothesis

A another example

We took a look at the code before attempting to compile and run the code.

The first thing we noticed was line 25, where the code initializes an InputStreamReader, using System.out as an argument.

This was a fairly clear **defect** in the code; the constructor for InputStreamReader takes an InputStream as an argument, and the code was giving it an OutputStream.

We made the **hypothesis** that the developer intended to use System.in as the input stream, as opposed to an arbitrary file on the system (which seemed pretty unlikely).

Changing the argument to System.in allowed the program to compile, as **predicted**

An third example

Characterize a Failure: When running the file with the input file of JD (utilizes the P function) The image is inverted and out of intended shape.

Hypothesis: The problem of inversion is usually an incorrect polarity of a variable. Instead of having -x you have x or vice versa. In this case however the image is inverted vertically. During the *while* loop of the P () method *xn* is declared as having the value *offs_x* + *amount2*. In all other parts of the program *amount2* is reserved for the y component of input.

Prediction: Before we mess with + and - of the P switch we should switch *amount* and *amount2*.

Result: Display is now valid but inverted with a tail on the starting line.

Continuing

Characterize a Failure: The display is still inverted but not as askew. Just upside down.

Hypothesis: The y component is inverted. This should be like stated above based upon the polarity of the input. The y component should be negative to flip the image to the desired direction. This is a result of y increasing as it moves down instead of up. This y inversion is valid in two different places the above changed code, *yn* and *yc*.

Prediction: Need to change to polarity of the numbers being input (amount2)

Experiment:

Changed Code:	to:
yc= offs_y + b_y + y0;	yc= offs_y + b_y - y0;
yn = offs_y + b_y+amount2;	yn = offs_y + b_y-amount2;

Result: Program now displays input as intended and is assumed to function as intended

Another characterization

- Failure #9.1:
  Characterization:JD is sideways.
- Hypothesis:This was likely due to moves in the point having their axes inverted.
- Prediction:By finding the axis inversion and correcting it, JD would stand.
- Experiment:We reversed the axes where the xn and yn are computed.
- Results:JD stood.
- Conclusion:He's still injured, but we're gettin' there.

Another characterization (II)

- Failure #9.2:
- Characterization:JD had an errant starting line.
- Hypothesis:The initial point was not being set correctly. This looked quite a bit like a y-axis sign inversion problem again.
- Prediction:By subtracting one of the variables setting the initial y point instead of adding, we would accomplish an accurate first line.
- Experiment:We changed y0 from being added to being subtracted from yn.
- Results:JD was drawn almost-correctly.
- Conclusion:One last bit to go...

Another characterization (III)

- Failure #9.3:
- Characterization:There was a line missing from JD, his side was missing.
- Hypothesis:This appeared to be the last line of the polygon, so it seemed that the issue was that the final line connecting finishing point to starting point was not being drawn.
- Prediction:By adding an implicit coordinate move from the finishing point to the starting point, the last draw() in the point code would draw the last line.
- Experiment:We added xn and yn computation to follow the while loop that consumed point input.
- Results:JD renders correctly.
- Conclusion: We pwn, and the rendering code operates correctly.

Debugging rules

1. UNDERSTAND THE SYSTEM
2. MAKE IT FAIL
3. QUIT THINKING AND LOOK
4. DIVIDE AND CONQUER
5. CHANGE ONE THING AT A TIME
6. KEEP AN AUDIT TRAIL
7. CHECK THE PLUG
8. GET A FRESH VIEW
9. IF YOU DIDN'T FIX IT, IT AIN'T FIXED

1. UNDERSTAND THE SYSTEM

- ".. you have to understand how things are supposed to work if you want to figure out why they don't" (p. 13)
- "When you do find the bugs, you'll need to fix them without breaking anything else. Understanding what the system is supposed to do is the first step toward not breaking it." (p. 15)
- Be careful about "lifting" another design, esp. if the context has changed
- Beware of implicit assumptions
- Look things up, you might have gotten some details wrong
- Know your tools

3. QUIT THINKING AND LOOK

- ".. the measure of a good debugger is not how soon you come up with a guess or how good your guesses are, but how few bad guesses you actually act on"
- Be sure to observe the failure!
- Dig deep to find the problem and sort out the details of the cause(s).
- "Keep looking until the failure you can see has a limited number of possible causes to examine"
- "As you make and chase bad guesses, you'll … know when the failure you see implicates a small enough piece of design."

QUIT THINKING AND LOOK - Cont'd

- Integrate debugging with the design process, add hooks, instrumentation, etc., that may be useful in debugging failures
- Make sure that your instrumentation isn't obscuring the bug or changing it! (Which principle?)
- Start with the simplest explanations
- Guess only if it allows you to focus the search
- Use your intuition about what is or is not likely

4. DIVIDE AND CONQUER

- Determine which side of the bug you're on, upstream or downstream.
- Begin with easy-to-spot test patterns
- Start where it's broken, don't start by testing things you believe are correct
- If you're aware of a bug, fix it. You might uncover something else, may avoid confusion when there are multiple bugs

5. CHANGE ONE THING AT A TIME

- Restore changes that didn't fix the problem!
- Don't fix more than you need to
- Isolate and control the variables- similar to using known inputs
- Compare non-working cases with working ones
- Can be painstaking if the output is bulky (maybe build a tool to help)

6. KEEP AN AUDIT TRAIL

- Keep track of even the smallest detail
- Note the effect of each thing that you did, and be **descriptive** and **circumspect**
- Don’t commit to memory, write it down!
- Correlate - discuss and describe the context

7. CHECK THE PLUG

- If your TV reception is snowy, should you immediately start to fix the TV?
- Question your assumptions
- If you encounter a problem that is “outworldly” check that you are on the “right planet”
- Start at the beginning
- Check for bugs or flaws in the tools, techniques, or assumptions used to troubleshoot the problem
- “Convictions are more dangerous enemies of truth than lies”

  - Nietzsche

8. GET A FRESH VIEW

- Know when to ask for help
- Get expert advice when you need it, could be a manual, FAQs website, wiki, message board, etc, groups.google.com
- Which is more useful: gaining insight or saving time?
- Report symptoms, not theories
- Describe what you didn't expect, what you didn't understand
- Sometimes explaining a problem is sufficient to resolve it on your own! Even when you are just writing it down!
- "When all else fails, read the manual again"
- There is a difference between debugging and troubleshooting
  - Debugging - figuring out why the design doesn't work as planned
  - Troubleshooting - find a problem in a particular instantiation, which more generally when the design is known to be "good" Doctors troubleshoot

Asking for help …. in the words of students who've taken this course

- "Asking for help grants you the opportunity to get a fresh view. You may be biased… Asking another person will allow them to explain in a way that you may never have thought of before due to your bias."
- "Ask for advice for new insight, experience… Asking for help does not make you weak or stupid. Everybody needs help once in a while."
- "…helps a person learn more from someone with experience.."
- "Ask for help because getting a fresh perspective, free from your own biases, can give you an extra clue in debugging. Asking people with different expertise, experience, and knowledge can go a long way to solving your problem. One myth that Agans dispels is that asking makes you look like a fool; on the contrary, it makes you look eager to solve the problem, and smart for using all resources on hand."

9. IF YOU DIDN'T FIX IT, IT AIN'T FIXED

- Convince yourself that ...
  - it's really fixed
  - your fix in fact fixed the problem
- Undo the fix, put it back in, etc., to see that the fix made the difference
- Is the fix intermittent?
- It might not be the right one, fixing things only under certain conditions.
- Figure out what is causing the failure
- If you get stumped, add diagnostic code, can be used to diagnose the failure the next time it occurs

If you didn't fix it, it ain't fixed, Cont'd

- Determine the cause of failure
  - Typo
  - programming error
  - conceptual error
- Fix the cause
- Fix the design process
  - There's oil the factory floor.
  - Do you just mop up the oil and forget about it?
  - Investigate a leaky pipe fitting.
  - Do you just tighten the fitting and that's it?
  - You discover that there is a design flaw. The machinery connected to fitting is fastened to the floor with only 2 bolts, causing the machine to vibrate, loosening the fitting.
  - Change the design to add 2 more bolts, and now the problem is fixed.

At the help desk

- Can you see what's going on?
- Do you understand how the program should behave?
- Do you need to change focus?
- Quit thinking and look (p. 163)
- Don't assume anything about how the software is used
- Maybe the software is old, and is now being used in ways that weren't anticipated when the software was designed
- Write a troubleshooting guide
- Is there anything you've overlooked?

At the help desk - Continued

- Imagine that you are at a help desk
  - Make sure the user follows the rules
  - "An ounce of patience is worth a pound of brains"
  - Verify all actions and results.
- Your users will misunderstand you and make mistakes (What you want them to look at, what you are describing, etc)
- They may not be able to describe what they see; they may give you an answer they assume to be true.
- Use automated tools, handling log files, etc.
- Verify even the simplest assumptions.
- Avail yourself of all troubleshooting information faqs, manuals, others' experiences
- Contribute to the body of knowledge (troubleshooting)