Debugging and Diagnostic Reasoning

PSU CS 300 Lecture 6-1

Bart Massey Assoc Prof Computer Science Portland State University <bart@cs.pdx.edu>

Debugging paradoxes

- Everyone knows how
- It's hard to find bugs
- It's easy to fix bugs
- It's hard to fix bugs right
- Debugging is unnecessary
- There is always debugging

Coding? Sure. Debugging?

- I know of no good debugging textbook
- I know of no good online tutorial on debugging
- There is little formal discipline of debugging in CS / SE

Think like a doctor

- Observe symptoms
- Generate diagnostic hypotheses, then observe further
- Select a treatment
- Check that the treatment solves the symptoms
- Consider consequences

Think like Dr. Frankenstein

- Kill patient as needed
- Start over anytime
- Take patient apart, patch in pieces
- Redesign or respecify patient as needed
- But don't work alone

Shorten the cycle

- Frankenstein → possibility of quick edit / fail /debug
- Thrashing ensues
- Don't proceed until you know what you're doing
- Check your work

Collecting symptoms

- Write things down!
- Have test cases (check 'em)
- Record not just function
- Form hypotheses in parallel
- Don't prematurely commit

Constructing hypotheses

- Write them down!
- Make sure that each is consistent with observation
- Only testable hypotheses
- Design experiments now
- Occam's Razor is sharp

Choosing a hypothesis

- Write things down!
- Run your distinguishing experiments
- May eliminate all
- What if there are two problems?

Confirming the hypothesis

- Run confirming experiments
- Write things down! (for future reference)
- Don't underdo this step!
- Look for root causes

Root cause analysis

- You aren't done until you can explain why the bug is there, in a way that anyone could understand
- Where is the root defect?
- How and when did the root defect get in?
- What needs to be done?

Experimental design

- Write designs down!
- What observations are needed? How can you make them?
- Instrumentation may be needed. Design it.
- Don't just poke at things

Debugging prototypes

- Often easier to build simple debugging prototype
 - understand system / language
 - understand algorithm
 - test experimental design
- Wrap stub prototype around real code as a test jig

Making a repair

- Root cause analysis guides repair strategy
- First do no harm
- The Kelly-Bootle Law
- Check the repair
- Document the repair

Pair debugging

- Better than pair programming
 - Common mistakes are humancaused
 - Fatigue is decreased
 - More knowledge is brought to bear
 - Sanity checks

Common pitfalls

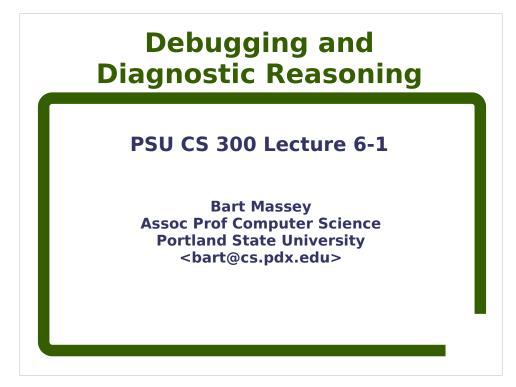
- Not understanding the defect (e.g. bad test case)
- Not understanding the error (e.g. code patches)
- Not understanding the repair (e.g. edit wrong file)
- Not checking the repair

Minimizing debugging

- If your design is right, and if you pseudocode, you will spend less time debugging
- If your V&V is good, you will do debugging in a less tight loop (is this a good thing?)

Get expert help

- Debugging is not for novices: seek expert debugging help
- Can learn a lot about debugging this way
- Experts will help you feel better about it all, too



Debugging paradoxes

- Everyone knows how
- It's hard to find bugs
- It's easy to fix bugs
- It's hard to fix bugs right
- Debugging is unnecessary
- There is always debugging

Coding? Sure. Debugging?

- I know of no good debugging textbook
- I know of no good online tutorial on debugging
- There is little formal discipline of debugging in CS / SE

Think like a doctor

- Observe symptoms
- Generate diagnostic hypotheses, then observe further
- Select a treatment
- Check that the treatment solves the symptoms
- Consider consequences

Think like Dr. Frankenstein

- Kill patient as needed
- Start over anytime
- Take patient apart, patch in pieces
- Redesign or respecify patient as needed
- But don't work alone



- Frankenstein → possibility of quick edit / fail /debug
- Thrashing ensues
- Don't proceed until you know what you're doing
- Check your work

Collecting symptoms

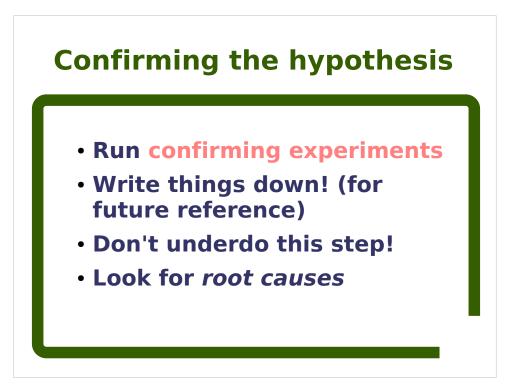
- Write things down!
- Have test cases (check 'em)
- Record not just function
- Form hypotheses in parallel
- Don't prematurely commit

Constructing hypotheses

- Write them down!
- Make sure that each is consistent with observation
- Only testable hypotheses
- Design experiments now
- Occam's Razor is sharp



- Write things down!
- Run your distinguishing experiments
- May eliminate all
- What if there are *two* problems?



Root cause analysis

- You aren't done until you can explain why the bug is there, in a way that anyone could understand
- Where is the root defect?
- *How and when* did the root defect get in?
- What needs to be done?



- Write designs down!
- What observations are needed? How can you make them?
- Instrumentation may be needed. Design it.
- Don't just poke at things

Debugging prototypes

- Often easier to build simple debugging prototype
 - understand system / language
 - understand algorithm
 - test experimental design
- Wrap stub prototype around real code as a test jig





Common pitfalls

- Not understanding the defect (e.g. bad test case)
- Not understanding the error (e.g. code patches)
- Not understanding the repair (e.g. edit wrong file)
- Not checking the repair

Minimizing debugging

- If your design is right, and if you pseudocode, you will spend less time debugging
- If your V&V is good, you will do debugging in a less tight loop (is this a good thing?)

Get expert help

- Debugging is not for novices: seek expert debugging help
- Can learn a lot about debugging this way
- Experts will help you feel better about it all, too