Lecture 11: Debugging & Defensive Programming

Terminology

- Bugs vs. Defects

The scientific approach to debugging

- hypothesis refutation
- occam's razor

Debugging tips

Designing for fewer defects

- firewalls
- instrumentation
- exceptions

Debugging basics

Debugging follows…

…"successful" testing
- the input is a list of tests that failed

…an inspection meeting
- the input is a list of issues raised

…failure to prove correctness (?)

The Debugging Process:

requires: a symptom of a problem

effects: defect corrected; symptom has disappeared; process has been fixed so mistake won't be repeated

Difficulties:

The symptom and the defect may be geographically remote
- especially in a closely coupled program

The symptom might change if another defect is corrected

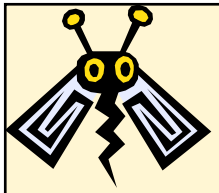
The defect may be in the verification procedure

Some symptoms are hard to reproduce
- especially timing problems

The symptom might not be the result of an defect
- But assume it is until you can prove otherwise

Bugs or Defects?

Source: Adapted from Liskov & Guttag, 2000, section 10.9



Bugs

crawl around

breed

eat things

can be detected because of chewed holes

crawl in when you are not looking

attack anyone indiscriminately

Defects

stay where they are

do not breed

do not eat things
(but may corrupt your data!)

can be detected because of failed tests

are inserted by people making mistakes

happen to careless programmers

you can't learn much (about software) from bugs,
but you *can* learn from your MISTAKES!

Source: Adapted from Liskov & Guttag, 2000, section 10.9

Debugging Approaches

Strategies

Brute force and ignorance:

- take memory dumps, do run-time tracing, put print statements everywhere
- You will be swamped with data!!

Backtracking:

- Begin at the point where the symptom occurs…
- … trace backwards step by step, by hand
- Only feasible for small programs

Cause elimination:

- Use deduction to list all possible causes
- devise tests to eliminate them one by one
- (try to find the simplest input that shows the symptom)

The Scientific Method

1) Study the available data
   - which tests worked?
   - which tests did not work?
2) Form a hypothesis
   - …that is consistent with (all) the data
3) Design an experiment to refute the hypothesis
   - the experiment must be repeatable
   - don't try to prove your hypothesis, try to disprove it

Occam's Razor

The simplest hypothesis is usually the correct one

Source: *Adapted from Liskov & Guttag, 2000, section a10.9*

Example

```
boolean palindrome (char *s)
/* effects: returns true if s reads the
   same reversed as it does forward */
```

Test cases:

s="able was I ere I saw elba" returns false

s="deed" returns true

Hypothesis 1:

maybe it fails on odd length strings?

simple refutation case: s="r" returns true

Hypothesis 2:

maybe it fails on strings with spaces in them?

simple refutation case: s=" " returns true

Hypothesis 3:

maybe it fails on odd length strings longer than 1?

simple refutation case: s="ere" returns false

The hypothesis was not refuted, but that doesn't mean it it true!

Note:
At each step we have more data.

Each hypothesis must be consistent with *all* the data

Source: Adapted from Liskov & Guttag, 2000, section 10.9

Debugging Tips

Examine intermediate results

- add in diagnostic code if necessary
- use binary chop to locate defects

The defect is probably not where you think it is

- keep an open mind
- occasionally review your reasoning

Ask yourself where the defect is not

- sometime proving the things you think you know will demonstrate that you were wrong

Check your input as well as your code

- test drivers, stubs, test cases, are just as likely to contain defects

Think carefully about what you can take for granted

- a fully tested procedure can still contain defects

Try simple things first

- reversed order of parameters
- failure to initialize a variable
- failure to re-initialize a variable
- failure to parenthesize an expression
- missing close comment bracket

Get someone to help

- Two heads are better than one

Make sure you have the right source code

Take a break

Source: Adapted from Liskov & Guttag, 2000, section 10.9

Before you repair

Don't be in a hurry to fix it

It's worthwhile finding as many defects as possible, and fixing them all together (because regression testing is expensive)

Is the defect repeated elsewhere?

You may be able to track down other instances of the same problem

What 'next defect' will be introduced by your fix?

Think about how the fix degrades your original design

especially look at its effect on coupling and cohesion

What could you have done to avoid the defect in the first place?

This is the first step towards process improvement

Correct the process as well as the product

LEARN FROM YOUR MISTAKES!
THEY ARE NOT 'BUGS', THEY ARE DEFECTS!!

Change Management

Keep a record of all changes

Comment each procedure / module with

- author
- date
- list of tests performed
- list of modifications (date, person, reason for change)

Design a change process

Ensure all changes are agreed by the team

Use a "request for change" form

Distribute the completed forms to clients, subcontractors, etc.

Don't skip the regression testing!

Should run *all* tests again after making any changes.

- Don't just run the ones that failed…

Use a configuration management tool

…if you are modifying different parts of the software in parallel

Firewalls & instrumentation

Design to make debugging easy

Firewalls: these check preconditions for each piece of code

Instrumentation: print statements that provide diagnostic information

(on a separate output channel)

Q: Should they be removed before delivery?

A: No!! Never!!! Don't even think about it!!!!

Removing them may introduce new defects

You have modified the code after it was tested!!!!!!!!!!!!!!

Accept some small loss of performance in return for:

Ability to diagnose non-software failures (hardware, system, etc)

Ability to diagnose latent defects during operations

Protection from defects introduced by future enhancement

Testing future changes is much easier

Removing firewalls and instrumentation…

…is like disconnecting the warning lights on an plane!

…is like flying untested software!

Exception Handling

If the programming language supports exceptions

use the full power of the language

e.g. in Java, making all exceptions "checked" leads to safer programs

but you'll still need to document them

e.g. Java doesn't force you to say why a method might throw an exception

Otherwise:

1) declare an enumerated type for exception names

```
enum str_exceptions {okay, null_pointer, empty_string, not_null_terminated};
```

2) have the procedure return an extra return value

either: `str_exceptions palindrome(char *s, boolean *result);`

or: `boolean palindrome(char *s, str_exceptions *e);`

(be consistent about which pattern you use)

3) test for exceptions each time you call the procedure

e.g.
```
if (palindrome(my_string, &result)==okay) { … }
else /*handle exception*/
```

4) write exception handlers

procedures that can be called to patch things up when an error occurs.

Writing Exception Handlers

The calling procedure is responsible for:

checking that an exception did not occur

handling it if it did

Could handle the exception by:

reflecting it up to the next level

i.e. the caller also throws an exception (up to the next level of the program)

Can throw the same exception (automatic propogation)…

…or a different exception (more context info available!)

masking it

i.e. the caller fixes the problem and carries on (or repeats the procedure call)

halt the program immediately

equivalent to passing it all the way up to the top level

When to use exceptions

Partial procedures vs. Exceptions

In general it is better to eliminate partial procedures

- unless checking for the exception is very expensive
- … or the exception can never occur (be careful!)

Normal behavior vs. Exceptional Behaviour

In general, exceptions should be kept separate from normal return values

- e.g. avoid using special values of the normal return value to signal exceptions

The exception result could get used as real data!

Exceptions *can be* used for "normal behaviours"

- E.g. Can use Java exception mechanism for alternative control flows
- But this makes the program harder to understand, so don't overuse them

Exceptions are for communication…

…between program units only (i.e. internally)

Users should never see exceptions, nor error codes!

References

Liskov, B. and Guttag, J., "Program Development in Java: Abstraction, Specification and Object-Oriented Design", 2000, Addison-Wesley.

Liskov and Guttag's section 10.9 includes one of the best treatments of debugging I have come across. Chapter 4 is a thorough treatment of Java exceptions, with lots of tips on how to use exceptions sensibly

Blum, B. "Software Engineering: A Holistic View". Oxford University Press, 1992

p379 for the history of the term 'bug', and a picture of the first 'bug'