it4490 – Software DESIGN AND CONSTRUCTION

11. Programming

For Your Amusement

- "Any fool can write code that a computer can understand. Good programmers write code that humans can understand" -- Martin Fowler
- "Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' " -- Steve McConnell
- "Programs must be written for people to read, and only incidentally for machines to execute." -- Abelson / Sussman
- "Everything should be built top-down, except the first time." -- Alan Perlis

Construction

- Coding
  - Program style ← we are here
  - Data structures and algorithms
  - Code tuning
- Debug
  - Debugging techniques & tools
- Test
  - Testing techniques

Ways to get your code right

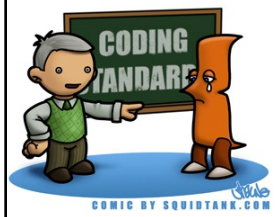
- Verification/quality assurance
  - Purpose is to uncover problems and increase confidence
  - Combination of reasoning and test
- Debugging
  - Finding out why a program is not functioning as intended
- Defensive programming
  - Programming with validation and debugging in mind
- Testing ≠ debugging
  - test: reveals existence of problem; test suite can also increase overall confidence
  - debug: pinpoint location + cause of problem

Outline

1. **Programming style**
2. Code tuning
3. Debugging

Programming style

- Coding convention/standard



Proper programming styles

- Use constants for all constant values (e.g. tax rate).
- Use variables whenever possible
- Always declare constants before variables at the beginning of a procedure
- Always comment code (especially ambiguous or misleading code), but do not over comment.
- Use appropriate descriptive names (with/without prefixes) for identifiers/objects (e.g. btndone).

Proper programming styles (2)

- Use brackets for mathematical expressions even if not required so that it is clear what was intended. E.g. (3*2) + (4*2)
- Always write code as efficiently as possible
- Make user friendly input and output forms

9

Proper programming styles (3)

- Include a header at the top of your code (using ' so that it does not get compiled):
  - Programmer's name
  - Date
  - Name of saved project
  - Teacher's name
  - Class name
  - Names of anyone who helped you
  - Brief description of what the program does

10

Outline

1. Programming style
2. **Code tuning**
3. Debugging

11

Code tuning

- Modifying correct code to make it run more efficiently
- Not the most effective/cheapest way to improve performance
- 20% of a program's methods consume 80% of its execution time.

12

Code Tuning Myths

- Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code – false!

```
for i = 1 to 10
   a[ i ] = i
end for
```

VS

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

13

Code Tuning Myths (2)

- *A fast program is just as important as a correct one* – false!

14

Code Tuning Myths (3)

- *Certain operations are probably faster or smaller than others* – false!
  - Always measure performance!

15

Code Tuning Myths (4)

- *You should optimize as you go* – false!
  - It is hard to identify bottlenecks before a program is completely working
  - Focus on optimization detracts from other program objectives

16

When to tune

- Use a high-quality design.
  - Make the program right.
  - Make it modular and easily modifiable
  - When it's complete and correct, check the performance.
- Consider compiler optimizations
- Measure
- Write clean code that's easy to understand and modify.

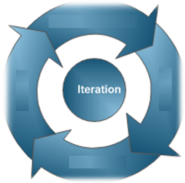
17

Measurement

- Measure to find bottlenecks
- Measurements need to be precise
- Measurements need to be repeatable

Optimize in iterations

- Measure improvement after each optimization
- If optimization does not improve performance – revert it



18

19

Code Tuning Techniques

- Stop Testing When You Know the Answer

```
if ( 5 < x ) and ( y < 10 ) then ...
```

```
?
```

```
negativeInputFound = False;
for ( i = 0; i < iCount; i++ ) {
  if ( input[ i ] < 0 ) {
    negativeInputFound = True;
  }
}
```

?

20

Code Tuning Techniques

- Order Tests by Frequency

```
Select char
 Case "+", "="
   ProcessMathSymbol(char)
 Case "0" To "9"
   ProcessDigit(char)
 Case ",", ".", "!", "?"
   ProcessPunctuation(char)
 Case " "
   ProcessSpace(char)
 Case "A" To "Z", "a" To "z"
   ProcessAlpha(char)
 Case Else
   ProcessError(char)
End Select
```

```
Select char
 Case "A" To "Z", "a" To "z"
   ProcessAlpha(char)
 Case " "
   ProcessSpace(char)
 Case ",", ".", "!", "?"
   ProcessPunctuation(char)
 Case "0" To "9"
   ProcessDigit(char)
 Case "+", "="
   ProcessMathSymbol(char)
 Case Else
   ProcessError(char)
End Select
```

21

Code Tuning Techniques

- Unswitching loops

```
for ( i = 0; i < count; i++ ) {
  if ( sumType == SUMTYPE_NET ) {
    netSum = netSum + amount[ i ];
  }
  else { grossSum = grossSum + amount[ i ]; }
}
```

```
?
```

22

Code Tuning Techniques

- Minimizing the work inside loops

```
for (i = 0; i < rateCount; i++) {
  netRate[i] = baseRate[i] * rates->discounts->factors->net;
}
```

```
?
for (i = 0; i < rateCount; i++) {
  netRate[i] = baseRate[i] * ?
}
```

23

Code Tuning Techniques

- Initialize at Compile Time

```
const double Log2 = 0.69314718055994529;
```

Code Tuning Techniques

- Use Lazy Evaluation

```
public int getSize() {
  if(size == null) {
    size = the_series.size();
  }
  return size;
}
```

24

Code Tuning Techniques

```
var myClass = function() {
  this.array_one = [1,2,3,4,5];
  this.array_two = [1,2,3,4,5];
  this.total = 0;
}

var my_instance = new myClass();

for (var i=0; i < 4; i++) {
   my_instance.total +=
                          (my_instance.array_one[i] +

                          my_instance.array_two[i]);
}
```


When iterating through data, keep memory references sequential

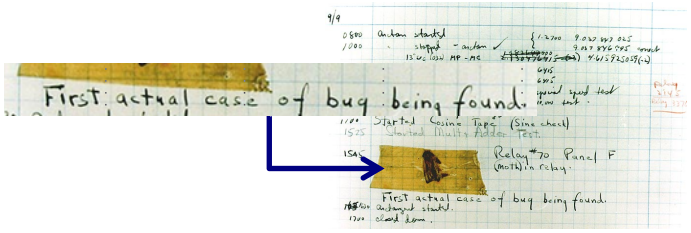
```
var myClass = function() {
  this.array_one = [1,2,3,4,5];
  this.array_two = [1,2,3,4,5];
  this.total = 0;
}

var my_instance = new myClass();

?
```


Outline

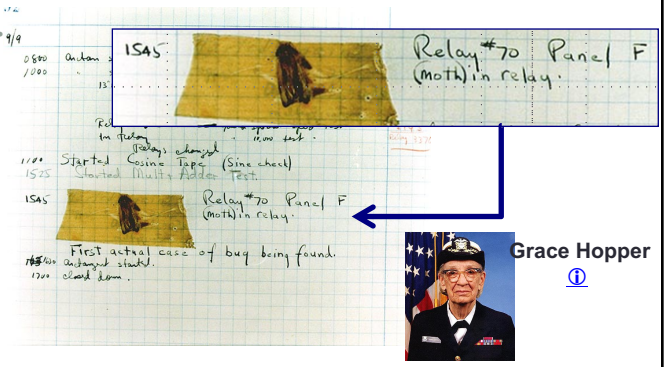
1. Programming style
2. Code tuning
3. **Debugging**

3.1. Overview

- Error
- Bug
- Fault
- Defect
- Failure



Whence "bug"



Grace Hopper

3.2. Defense in depth

- Make errors impossible
  - Java makes memory overwrite errors impossible
- Don't introduce defects
  - Correctness: get things right the first time
- Make errors immediately visible
  - Local visibility of errors: best to fail immediately
  - Example: assertions
- Last resort is debugging
  - Needed when failure (effect) is distant from cause (defect)
  - Scientific method: Design experiments to gain information about the defect
    - Fairly easy in a program with good modularity, representation hiding, specs, unit tests etc.
    - Much harder and more painstaking with a poor design, e.g., with rampant rep exposure

3.2.1. First defense: Impossible by design

- In the language
  - Java makes memory overwrite errors impossible
- In the protocols/libraries/modules
  - TCP/IP guarantees that data is not reordered
  - `BigInteger` guarantees that there is no overflow
- In self-imposed conventions
  - Banning recursion prevents infinite recursion/insufficient stack – although it may push the problem elsewhere
  - Immutable data structure guarantees behavioral equality
  - Caution: You must maintain the discipline

3.2.2. Second defense: Correctness

- Get things right the first time
  - Think before you code. Don't code before you think!
  - If you're making lots of easy-to-find defects, you're also making hard-to-find defects – don't use the compiler as crutch
- Especially true, when debugging is going to be hard
  - Concurrency, real-time environment, no access to customer environment, etc.
- Simplicity is key
  - Modularity
    - Divide program into chunks that are easy to understand
    - Use abstract data types with well-defined interfaces
    - Use defensive programming; avoid rep exposure
  - Specification
    - Write specs for all modules, so that an explicit, well-defined contract exists between each module and its clients

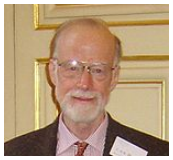
Strive for simplicity

"There are two ways of constructing a software design:

- One way is to make it so simple that there are obviously no deficiencies, and
- the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Sir Anthony Hoare

Brian Kernighan

3.2.3. Third defense: Immediate visibility

- If we can't prevent errors, we can try to localize them to a small part of the program
  - Assertions: catch errors early, before they contaminate and are perhaps masked by further computation
  - Unit testing: when you test a module in isolation, you can be confident that any error you find is due to a defect in that unit (unless it's in the test driver)
  - Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed
- When localized to a single method or small module, defects can usually be found simply by studying the program text

Benefits of immediate visibility

- The key difficulty of debugging is to find the defect: the code fragment responsible for an observed problem
  - A method may return an erroneous result, but be itself error-free, if there is prior corruption of representation
- The earlier a problem is observed, the easier it is to fix
  - Frequently checking the rep invariant helps
- General approach: fail-fast
  - Check invariants, don't just assume them
  - Don't (usually) try to recover from errors – it may just mask them

Don't hide errors

```
// k is guaranteed to be present in a
int i = 0;
while (true) {
  if (a[i]==k) break;
  i++;
}
```

- This code fragment searches an array a for a value k.
  - Value is guaranteed to be in the array
  - What if that guarantee is broken (by a defect)?
- Temptation: make code more "robust" by not failing

37

Don't hide errors

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
  if (a[i]==k) break;
  i++;
}
```

- Now at least the loop will always terminate
  - But it is no longer guaranteed that `a[i]==k`
  - If other code relies on this, then problems arise later
  - This makes it harder to see the link between the defect and the failure

38

Don't hide errors

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
  if (a[i]==k) break;
  i++;
}
assert (i=a.length) : "key not found";
```

- Assertions let us document and check invariants
  - Abort/debug program as soon as problem is detected: turn an error into a failure
- But the assertion is not checked until we use the data, which might be a long time after the original error
  - "*why* isn't the key in the array?"

39

Checks In Production Code

- Should you include assertions and checks in production code?
  - Yes: stop program if check fails - don't want to take chance program will do something wrong
  - No: may need program to keep going, maybe defect does not have such bad consequences (the failure is acceptable)
  - Correct answer depends on context!

Ariane 5 – program halted because of overflow in unused value, exception thrown but not handled until top level, rocket crashes… [although the full story is more complicated]

40

3.2.3. Debugging: Last resort

- Defects happen – people are imperfect
  - Industry average: 10 defects per 1000 lines of code ("kloc")
- Defects that are not immediately localizable happen
  - Found during integration testing
  - Or reported by user
- The cost of finding and fixing an error usually goes up by an order of magnitude for each lifecycle phase it passes through

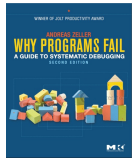
- step 1 – Clarify symptom (simplify input), create test
- step 2 – Find and understand cause, create better test
- step 3 – Fix
- step 4 – Rerun all tests

3.3. Debugging

- Step 1 – find a small, repeatable test case that produces the failure (may take effort, but helps clarify the defect, and also gives you something for regression)
  - Don't move on to next step until you have a repeatable test
- Step 2 – narrow down location and proximate cause
  - Study the data / hypothesize / experiment / repeat
  - May change the code to get more information
  - Don't move on to next step until you understand the cause
- Step 3 – fix the defect
  - Is it a simple typo, or design flaw? Does it occur elsewhere?
- Step 4 – add test case to regression suite
  - Is this failure fixed? Are any other new failures introduced?

Debugging and the scientific method

- Debugging should be systematic
  - Carefully decide what to do – flailing can be an instance of an epic fail
  - Keep a record of everything that you do
  - Don't get sucked into fruitless avenues
- Formulate a hypothesis
- Design an experiment
- Perform the experiment
- Adjust your hypothesis and continue



Reducing input size example

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A,B s.t.full=A+sub+B)
boolean contains(String full, String sub);
```

- User bug report
  - It can't find the string `"very happy"` within:
    `"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`
- Less than ideal responses
  - See accented characters, about not having thought about unicode, and go diving for your Java texts to see how that is handled
  - Try to trace the execution of this example
- Better response: simplify/clarify the symptom

Reducing absolute input size

- Find a simple test case by divide-and-conquer
- Pare test down – can't find `"very happy"` within
  - `"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`
  - `"I am very very happy to see you all."`
  - `"very very happy"`
- Can find `"very happy"` within
  - `"very happy"`
- Can't find `"ab"` within `"aab"`

45

Reducing relative input size

- Sometimes it is helpful to find two almost identical test cases where one gives the correct answer and the other does not
  - Can't find "very happy" within
    - `"I am very very happy to see you all."`
  - Can find "very happy" within
    - `"I am very happy to see you all."`

46

General strategy: simplify

- In general: find simplest input that will provoke failure
  - Usually not the input that revealed existence of the defect
- Start with data that revealed defect
  - Keep paring it down (binary search "by you" can help)
  - Often leads directly to an understanding of the cause
- When not dealing with simple method calls
  - The "test input" is the set of steps that reliably trigger the failure
  - Same basic idea

47

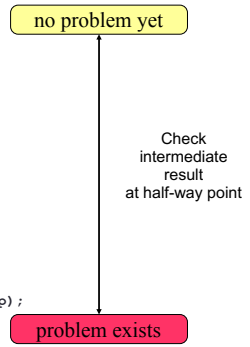
Localizing a defect

- Take advantage of modularity
  - Start with everything, take away pieces until failure goes
  - Start with nothing, add pieces back in until failure appears
- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results
- Binary search speeds up the process
  - Error happens somewhere between first and last statement
  - Do binary search on that ordered set of statements

48

binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
```



binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
```

no problem yet

Check intermediate result at half-way point

problem exists

Quickly home in on defect in O(log n) time by repeated subdivision

Detecting Bugs in the Real World

- Real Systems
  - Large and complex (duh!)
  - Collection of modules, written by multiple people
  - Complex input
  - Many external interactions
  - Non-deterministic
- Replication can be an issue
  - Infrequent failure
  - Instrumentation eliminates the failure
- Defects cross abstraction barriers
- Large time lag from corruption (defect) to detection (failure)

Heisenbugs

- Sequential, deterministic program – failure is repeatable
- But the real world is not that nice…
  - Continuous input/environment changes
  - Timing dependencies
  - Concurrency and parallelism
- Failure occurs randomly
- Hard to reproduce
  - Use of debugger or assertions → failure goes away
  - Only happens when under heavy load
  - Only happens once in a while

Logging Events

- Build an event log (circular buffer) and log events during execution of program as it runs at speed
- When detect error, stop program and examine logs to help you reconstruct the past
- The log may be all you know about a customer's environment – helps you to reproduce the failure

53

Tricks for Hard Bugs

- Rebuild system from scratch, or restart/reboot
  - Find the bug in your build system or persistent data structures
- Explain the problem to a friend
- Make sure it is a bug – program may be working correctly and you don't realize it!
- Minimize input required to exercise bug (exhibit failure)
- Add checks to the program
  - Minimize distance between error and detection/failure
  - Use binary search to narrow down possible locations
- Use logs to record events in history

54

Where is the bug?

- If the bugis not where you think it is, ask yourself where it cannot be; explain why
- Look for stupid mistakes first, e.g.,
  - Reversed order of arguments: `Collections.copy(src, dest)`
  - Spelling of identifiers: `int hashcode()`
    - `@Override` can help catch method name typos
  - Same object vs. equal: `a == b` versus `a.equals(b)`
  - Failure to reinitialize a variable
  - Deep vs. shallow copy
- Make sure that you have correct source code
  - Recompile everything

55

When the going gets tough

- Reconsider assumptions
  - E.g., has the OS changed? Is there room on the hard drive?
  - Debug the code, not the comments – ensure the comments and specs describe the code
- Start documenting your system
  - Gives a fresh angle, and highlights area of confusion
- Get help
  - We all develop blind spots
  - Explaining the problem often helps
- Walk away
  - Trade latency for efficiency – sleep!
  - One good reason to start early

56

Key Concepts in Review

- Testing and debugging are different
  - Testing reveals **existence of failures**
  - Debugging pinpoints **location of defects**
- Goal is to get program right
- Debugging should be a **systematic process**
  - Use the scientific method
- Understand the source of defects
  - To find similar ones and prevent them in the future

57

Supporting Tools: Eclipse plug-in

- Checkstyle: Help programmers write Java code that adheres to a coding standard
- FindBugs: Uses static analysis to look for bugs in Java code
  - Standalone Swing application
  - Eclipse plug-in
  - Integrated into the build process (Ant or Maven)

58

Findbugs features

- Not concerned by formatting or coding standards
- Detecting potential bugs and performance issues
  - Can detect many types of common, hard-to-find bugs
  - Use "bug patterns"

NullPointerException

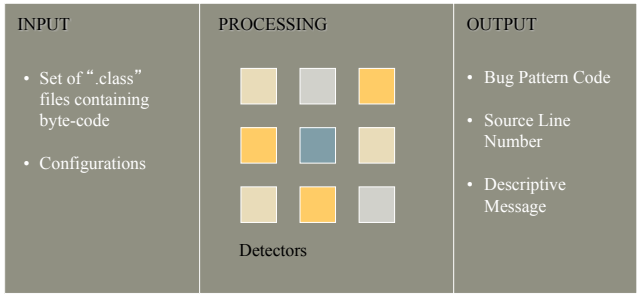
```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
 ...
}
```

Uninitialized field

```
public class ShoppingCart {
   private List items;
   public addItem(Item item) {
      items.add(item);
   }
}
```

59

How Findbug works?



60

What Finbugs can do?

- FindBugs comes with over 200 rules divided into different categories:
  - Correctness
    - E.g. infinite recursive loop, reads a field that is never written
  - Bad practice
    - E.g. code that drops exceptions or fails to close file
  - Performance
    - Multithreaded correctness
  - Dodgy
    - E.g. unused local variables or unchecked casts


61
?