

9. Programming



1

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

2

2

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 \neq debugging
 - test: reveals existence of problem; test suite can also increase overall confidence
 - debug: pinpoint location + cause of problem

3

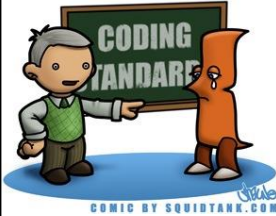
Outline

1. **Programming style**
2. Code tuning / optimization
3. Code refactoring
4. Debugging

4

Programming style

- Coding convention/standard



COMIC BY SQUIDTANK.COM

Java Coding Conventions on One Page

William C. Wake (William.Wake@acm.org), 2-17-2000

General conventions see <http://java.sun.com/docs/codeconv/index.html>

Specify a package (not default)

```
package com.mine.display;
```

May use ""

```
import java.util.*;
```

Each word capitalized

```
public class MineFrame implements Interfacable {
```

Each word capitalized

```
public static final int MIN_SIZE = 22;
```

All caps with "_" separator

Constant

```
static int height;
```

No special prefix for "static" variables

Noun phrase. Fields not "public"

```
String lastName;
```

JavaDoc conventions see <http://java.sun.com/products/jdk/1.4/docs/writingdoccomments/index.html>

One liners are OK

```
/** Represents the getLastName() and getAge() */
public String getName() { return lastName; }
```

JavaBeans naming conventions see <http://java.sun.com/beans/docs/beans.101.pdf>

Braces here and here

```
public int getAge() {
    if (age == MIN_SIZE)
        return MIN_SIZE;
    return height;
}
```

4-space indent throughout

```
protected void paint(Graphics g, int x, int y) throws IOException {
    Writer out;
    try {
        out = new FileWriter(filename);
        appendName(out);
        out.println("Age: " + age);
    } catch (FileNotFoundException e) {
        log.info(e);
    }
    finally {
        if (out != null) try {out.close();} catch (IOException ignored) {}
    }
}
```

Exceptions either re-thrown or handled

"finally" clause to release resources

Catch any exceptions in "finally" clause so original exception is reported

5

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).

6

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

7

Proper programming styles (3)

- Include a header at the top of your code:
 - 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

8

Check style tool

- Demo video

9

Outline

1. Programming style
2. **Code tuning / optimization**
3. Code refactoring
4. Debugging

10

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.

11

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
```

12

Code Tuning Myths (2)

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



13

Code Tuning Myths (3)

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



14

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

15

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.

16

Measurement

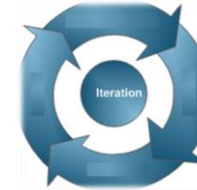
- Measure to find bottlenecks
- Measurements need to be precise
- Measurements need to be repeatable



17

Optimize in iterations

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



18

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;
    }
}
```

19

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
```

20

Code Tuning Techniques

- Unswitching loops

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

?

21

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] * ?
}
```

22

Code Tuning Techniques

- Initialize at Compile Time

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



23

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]);
}
```

25

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();

?
```

26

Outline

1. Programming style
2. Code tuning / optimization
3. **Code refactoring**
4. Debugging

27

What is Refactoring?

Refactoring means "to improve the design and quality of existing source code without changing its external behavior".

Martin Fowler



- A step by step process that turns the bad code into good code
- Based on "refactoring patterns" → well-known recipes for improving the code

28

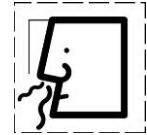
Code Refactoring

- What is **refactoring** of the source code?
 - Improving the design and quality of existing source code without changing its behavior
 - Step by step process that turns the bad code into good code (if possible)
- **Why** we need refactoring?
 - Code constantly changes and its quality constantly degrades (unless refactored)
 - Requirements often change and code needs to be changed to follow them

29

When to Refactor?

- **Bad smells in the code** indicate need of refactoring
- Refactor:
 - To make adding a new function easier
 - As part of the process of fixing bugs
 - When reviewing someone else's code
 - Have technical debt (or any problematic code)
 - When doing test-driven development
- **Unit tests** guarantee that refactoring does not change the behavior
 - If there are no unit tests, write them



30

Refactoring: Main Principles

- Keep it simple (**KISS** principle)
- Avoid duplication (**DRY** principle)
- Make it expressive (self-documenting, comments, etc.)
- Reduce overall code (**KISS** principle)
- Separate concerns (decoupling)
- Appropriate level of abstraction (work through abstractions)
- **Boy scout rule**
 - Leave your code better than you found it

31

Refactoring: The Typical Process

1. Save the code you start with
 - Check-in or backup the current code
2. Prepare tests to assure the behavior after the code is refactored
 - Unit tests / characterization tests
3. Do refactoring one at a time
 - Keep refactoring small
 - Don't underestimate small changes
4. Run the tests and they should pass / else revert
5. Check-in (into the source control system)



32

Refactoring Tips

- Keep refactoring small
- One at a time
- Make a checklist
- Make a "later" / TODO list
- Check-in / commit frequently
- Add tests cases
- Review the results
 - Pair programming
- Use tools (Visual Studio + add-ins / Eclipse + plugins / others)



33



CODE REFACTORING

Live Demo

34

Code Smells

- **Code smells** == certain structures in the code that suggest the possibility of refactoring
- Types of code smells:
 - The bloaters
 - The obfuscators
 - Object-oriented abusers
 - Change preventers
 - Dispensables
 - The couplers



35

Code Smells: The Bloaters

- **Long method**
 - Small methods are always better (easy naming, understanding, less duplicate code)
- **Large class**
 - Too many instance variables or methods
 - Violating "Single Responsibility" principle
- **Primitive obsession (overused primitives)**
 - Over-use of primitive values, instead of better abstraction
 - Can be extracted in separate class with encapsulated validation



36

Code Smells: The Bloaters (2)

- Long parameter list (**in** / **out** / **ref** parameters)
 - May indicate procedural rather than OO style
 - May be the method is doing too much things
- Data clumps
 - A set of data are always used together, but not organized together
 - E.g. credit card fields in the **Order** class
- Combinatorial explosion
 - Ex. **ListCars()**, **ListByRegion()**, **ListByManufacturer()**, **ListByManufacturerAndRegion()**, etc.
 - Solution may be the **Interpreter** pattern (LINQ)

37

Code Smells: The Bloaters (3)

- Oddball solution
 - A different way of solving a common problem
 - Not using consistency
 - Solution: Substitute algorithm or use an **Adapter**
- Class doesn't do much
 - Solution: Merge with another class or remove
- Required setup / teardown code
 - Requires several lines of code before its use
 - Solution: use parameter object, factory method, **IDisposable**

38

Code Smells: The Obfuscators

- Regions
 - The intent of the code is unclear and needs commenting (smell)
 - The code is too long to understand (smell)
 - Solution: partial class, a new class, organize code
- Comments
 - Should be used to tell **WHY**, not **WHAT** or **HOW**
 - Good comments: provide additional information, link to issues, explain an algorithm, explain reasons, give context
 - Link: [Funny comments](#)

39

Code Smells: The Obfuscators (2)

- Poor / improper names
 - Should be proper, descriptive and consistent
- Vertical separation
 - You should define variables just before first use to avoid scrolling
 - In JS variables are defined at the function start → use small functions
- Inconsistency
 - Follow the POLA (Principle of Least Astonishment)
 - Inconsistency is confusing and distracting
- Obscured intent
 - Code should be as expressive as possible

40

Code Smells: OO Abusers

- **Switch statement**
 - Can be replaced with polymorphism
- **Temporary field**
 - When passing data between methods
- **Class depends on subclass**
 - The classes cannot be separated (circular dependency)
 - May break the Liskov substitution principle
- **Inappropriate static field**
 - Strong coupling between **static** and callers
 - Static things cannot be replaced or reused

41

Code Smells: Change Preventers

- **Divergent change**
 - A class is commonly changed in different ways / different reasons
 - Violates SRP (single responsibility principle)
 - Solution: extract class
- **Shotgun surgery**
 - One change requires changes in many classes
 - Hard to find them, easy to miss some
 - Solution: move methods, move fields, reorganize the code

42

Code Smells: Change Preventers (2)

- **Conditional complexity**
 - Cyclomatic complexity (number of unique paths that the code can be evaluated)
 - Symptoms: deep nesting (arrow code) and buggy **if**-s
 - Solutions: extract method, "Strategy" pattern, "State" pattern, "Decorator"
- **Poorly written tests**
 - Badly written tests can prevent change
 - Tight coupling

43

Code Smells: Dispensables

- **Lazy class**
 - Classes that don't do enough to justify their existence should be removed
 - Every class costs something to be understood and maintained
- **Data class**
 - Some classes with only fields and properties
 - Missing validation? Class logic split into other classes?
 - Solution: move related logic into the class

44

Code Smells: Dispensables (2)

- **Duplicated code**
 - Violates the DRY principle
 - Result of copy-pasted code
 - Solutions: extract method, extract class, pull-up method, **Template Method** pattern
- **Dead code** (code that is never used)
 - Usually detected by static analysis tools
- **Speculative generality**
 - "Some day we might need this ..."
 - The "YAGNI" principle

45

Code Smells: The Couplers

- **Feature envy**
 - Method that seems more interested in a class other than the one it actually is in
 - Keep together things that change together
- **Inappropriate intimacy**
 - Classes that know too much about one another
 - Smells: inheritance, bidirectional relationships
 - Solutions: move method / field, extract class, change bidirectional to unidirectional association, replace inheritance with delegation

46

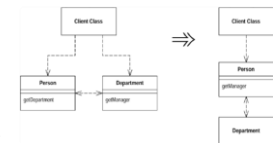
Code Smells: The Couplers (2)

- **The Law of Demeter (LoD)**
 - A given object should assume as little as possible about the structure or properties of anything else
 - Bad e.g.: **customer.Wallet.RemoveMoney()**
- **Indecent exposure**
 - Some classes or members are public but shouldn't be
 - Violates encapsulation
 - Can lead to inappropriate intimacy

47

Code Smells: The Couplers (3)

- **Message chains**
 - **Something.Another.SomeOther.Other.YetAnother**
 - Tight coupling between client and the structure of the navigation
- **Middle man**
 - Sometimes delegation goes too far
 - Sometimes we can remove it or inline it
- **Tramp data**
 - Pass data only because something else needs it
 - Solutions: Remove middle-man data, extract class



48

Code Smells: The Couplers (4)

- **Artificial coupling**
 - Things that don't depend upon each other should not be artificially coupled
- **Hidden temporal coupling**
 - Consecutively performed operations should not be guessed
 - E.g. pizza class should not know the steps of making pizza -> **Template Method** pattern
- **Hidden dependencies**
 - Classes should declare their dependencies in their constructor
 - **new** is glue / **Dependency Inversion** principle

49



50

Refactoring Patterns

- **When** should we perform refactoring of the code?
 - **Bad smells** in the code indicate **need of refactoring**
- **Unit tests** guarantee that refactoring preserves the behavior
- Refactoring patterns
 - **Large repeating code** fragments → extract duplicated code in separate method
 - **Large methods** → split them logically
 - **Large loop** body or **deep nesting** → extract method

51

Refactoring Patterns (2)

- Class or method has **weak cohesion** → split into several classes / methods
- Single change carry out changes in several classes → classes have tight coupling → consider redesign
- Related data are always used together but are not part of a single class → group them in a class
- A method has **too many parameters** → create a class to groups parameters together
- A method calls more methods from another class than from its own class → move it

52

Rafactoring Patterns (3)

- Two classes are tightly coupled → merge them or redesign them to separate their responsibilities
- Public non-constant fields → make them private and define accessing properties
- Magic numbers in the code → consider extracting constants
- Bad named class / method / variable → rename it
- Complex boolean condition → split it to several expressions or method calls

53

Rafactoring Patterns (4)

- Complex expression → split it into few simple parts
- A set of constants is used as enumeration → convert it to enumeration
- Too complex method logic → extract several more simple methods or even create a new class
- Unused classes, methods, parameters, variables → remove them
- Large data is passed by value without a good reason → pass it by reference

54

Rafactoring Patterns (5)

- Few classes share repeating functionality → extract base class and reuse the common code
- Different classes need to be instantiated depending on configuration setting → use factory
- Code is not well formatted → reformat it
- Too many classes in a single namespace → split classes logically into more namespaces
- Unused using definitions → remove them
- Non-descriptive error messages → improve them
- Absence of defensive programming → add it

55

Refactoring Levels



56

Data-Level Refactoring

- Replace a magic number with a named constant
- Rename a variable with more informative name
- Replace an expression with a method
 - To simplify it or avoid code duplication
- Move an expression inline
- Introduce an intermediate variable
 - Introduce explaining variable
- Convert a multi-use variable to a multiple single-use variables
 - Create separate variable for each usage



57

Data-Level Refactoring (2)

- Create a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
 - Additional behavior / validation logic (money)
- Convert a set of type codes (constants) to **enum**
- Convert a set of type codes to a class with subclasses with different behavior
- Change an array to an object
 - When you use an array with different types in it
- Encapsulate a collection



58

Statement-Level Refactoring

- Decompose a boolean expression
- Move a complex boolean expression into a well-named boolean function
- Use **break** or **return** instead of a loop control variable
- Return as soon as you know the answer instead of assigning a return value
- Consolidate duplicated code in conditionals
- Replace conditionals with polymorphism
- Use null-object design pattern instead of checking for **null**



59

Method-Level Refactoring

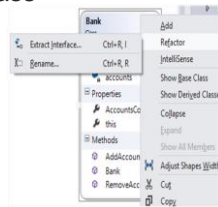
- Extract method / inline method
- Rename a method
- Convert a long routine to a class
- Add / remove parameter
- Combine similar methods by parameterizing them
- Substitute a complex algorithm with simpler
- Separate methods whose behavior depends on parameters passed in (create new ones)
- Pass a whole object rather than specific fields
- Encapsulate downcast / return interface types



60

Class-Level Refactoring

- Change a structure to class and vice versa
- Pull members up / push members down the hierarchy
- Extract specialized code into a subclass
- Combine similar code into a superclass
- Collapse hierarchy
- Replace inheritance with delegation
- Replace delegation with inheritance



61

Class Interface Refactorings

- Extract interface(s) / keep interface segregation
- Move a method to another class
- Split a class / merge classes / delete a class
- Hide a delegating class
 - A calls B and C when A should call B and B call C
- Remove the man in the middle
- Introduce (use) an extension class
 - When you have no access to the original class
 - Alternatively use the **Decorator** pattern

62

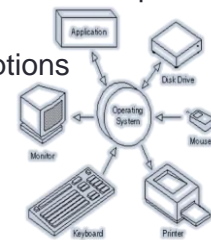
Class Interface Refactoring (2)

- Encapsulate an exposed member variable
 - Always use properties
 - Define proper access to getters and setters
 - Remove setters to read-only data
- Hide data and routines that are not intended to be used outside of the class / hierarchy
 - private -> protected -> internal -> public
- Use strategy to avoid big class hierarchies
- Apply other design patterns to solve common class and class hierarchy problems (**Façade**, **Adapter**, etc.)

63

System-Level Refactoring

- Move class (set of classes) to another namespace / assembly
- Provide a factory method instead of a simple constructor / use fluent API
- Replace error codes with exceptions
- Extract strings to resource files
- Use dependency injection
- Apply architecture patterns



64

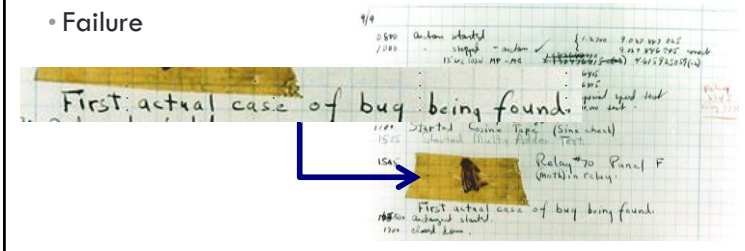
Outline

1. Programming style
2. Code tuning / optimization
3. Code refactoring
4. **Debugging**

65

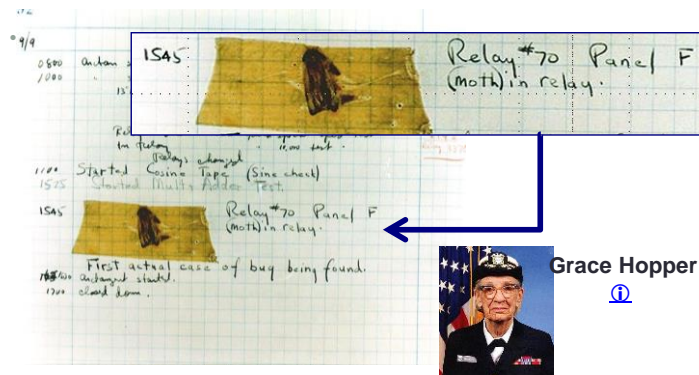
4.1. Overview

- Error
- Bug
- Fault
- Defect
- Failure



66

Whence “bug”



Grace Hopper [i](#)

67

4.2. 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

68

4.2.1. 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

69

4.2.2. 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

70

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."



Sir Anthony Hoare



Brian Kernighan



"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."

71

4.2.3. 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

72

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

73

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

74

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

75

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?”

76

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]



77

4.2.3. 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

78

4.3. 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?

79

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



80

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

81

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"

82

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."

83

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

84

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

85

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

86

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

87

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)

88

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

89

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

90

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

91

Where is the bug?

- ☐ If the bug is 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

92

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

93

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

94

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)

95

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

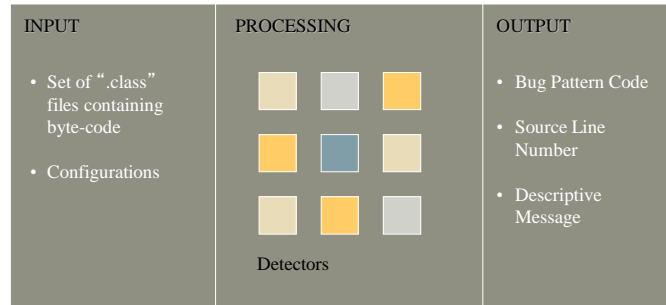
```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
    ...
}
```

Uninitialized field

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

96

How Findbug works?



97

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

98



99