

Imperative Programming 3

Errors

Peter Jeavons

Trinity Term 2019



Software Bugs


A software bug is an error, flaw, mistake, failure, fault or “undocumented feature” in a computer program that prevents it from behaving as intended [Wikipedia].

A moth stuck in
a relay of Mark II
computer (1947)

9/9

0800 Antam started
1000 " stopped - antam ✓
1300 (032) MP-MC { 1.2700 9.037 847 025
2.130476415 (033) 9.037 846 995 correct
2.130476415 (033) 4.615925059 (-2)
2.130676415 correct
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test.

Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.
1700 Antam started.
1700 closed down.

Relay 3145
Relay 3370

First actual case of bug being found

Eliminating Software Bugs?

- Bugs **will happen** regardless of your experience and effort
- Industry-average bug density
 - 10 - 100 bugs / KLOC after coding
 - 0.5 - 5 bugs / KLOC not detected before delivery

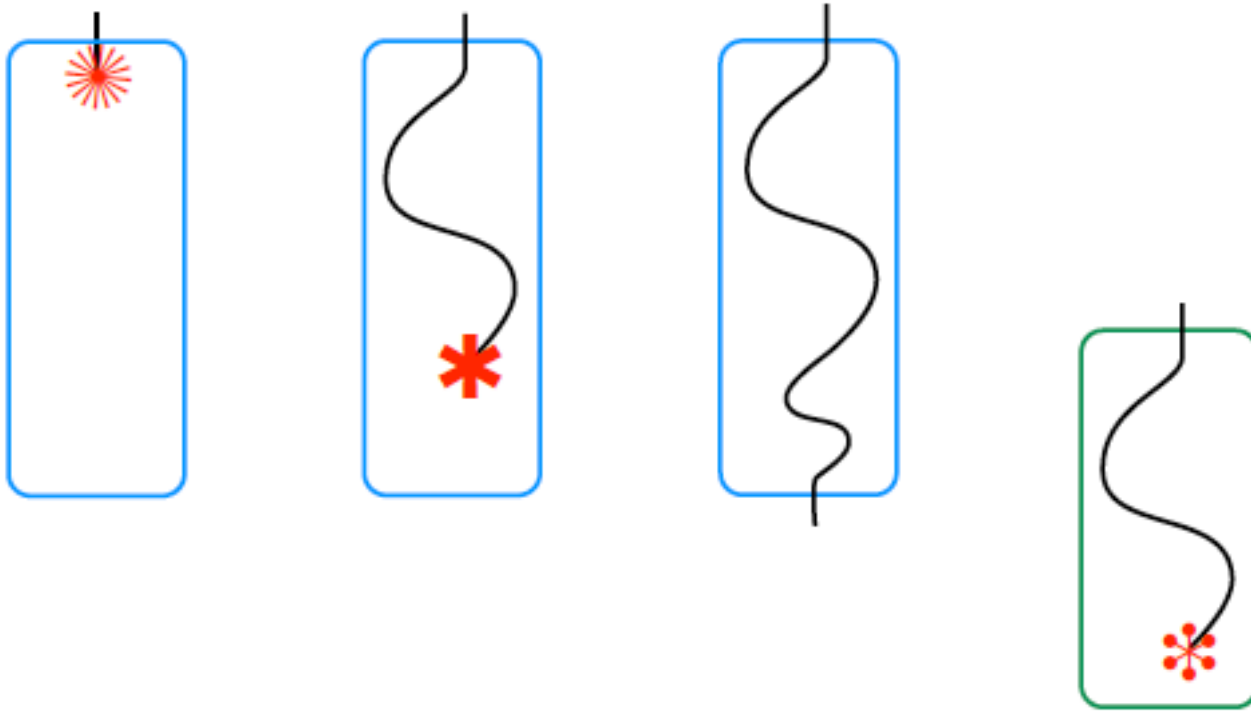
Question: What kind of errors have you experienced in software?

Why Do Errors Happen?

- Humans have a limited ability to...
 - anticipate all possible **inputs** and **states**
 - **specify** the desired response to each set of valid inputs
 - certain inputs are valid, but not anticipated \Rightarrow desired output to these inputs is not specified
 - understand all possible **execution paths**
 - decision points and loops increase the number of possible executions

Unanticipated Input: “Garbage In”

- No checks \Rightarrow garbage out
 - fail with confusing exception later
 - silently compute the wrong value
 - return normally but compromise some other object



Defensive Programming

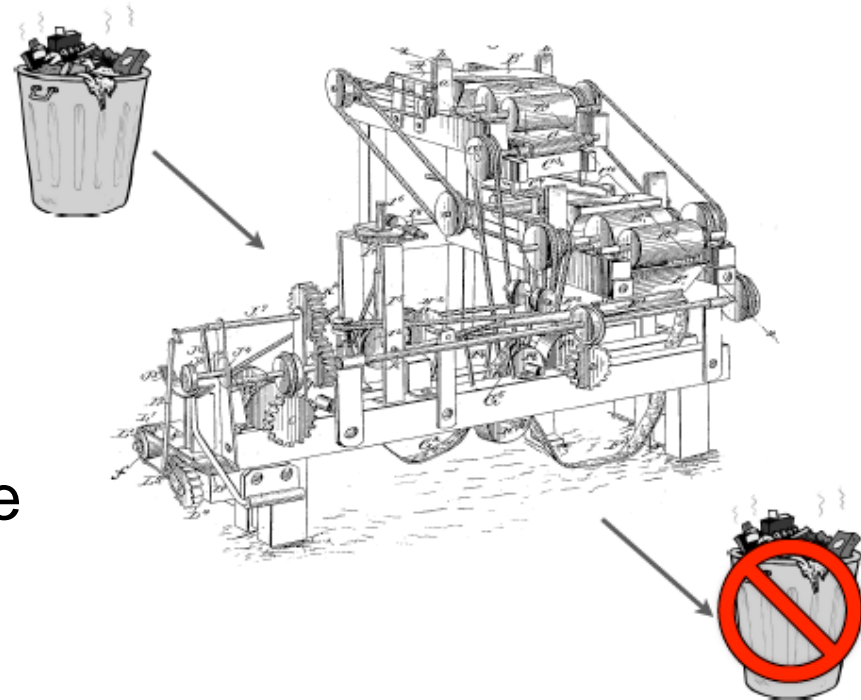
- Goal: Garbage In, Non-garbage Out

- Sources of garbage

- uncontrollable external sources
- method parameters
- corrupt state

- Options for dealing with garbage

- garbage in, nothing out
- garbage in, error message out
- turn garbage input into clean input



Rejecting Invalid Input

- Document pre-conditions of a method: the “contract”
- Check inputs for validity
 - reference is not null
 - input param values are within valid range
 - stream status
 - file access type: read, write, both
- Throw exception if bad input

The key trade-off ...
cost of throwing exception
vs.
cost of wrong input
x probability it is wrong

```
/**
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * nonnegative BigInteger.
 *
 * @param m the modulus, which must be positive.
 * @return this mod m.
 * @throws IllegalArgumentException if m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0) {
        throw new IllegalArgumentException("Modulus not positive");
    }

    // ...
}
```


Rejecting Invalid Input

Input to a method doesn't just come via parameters:

```
def register(facebookAccessToken: String) = {  
    response: FacebookResponse =  
        _facebookSdkClient.GetUser(facebookAccessToken);  
  
    if (string.IsNullOrEmpty(response.Email))  
        throw new InvalidOperationException(  
            "Invalid response from Facebook");  
  
    /* Register the user */  
    . . .  
}
```

Fixing Invalid Input

```
import java.util.Scanner
val input = new Scanner(System.in)

println("Enter a lowercase vowel")
while (!input.hasNext("[aeiou]")) {
    println("Not a vowel; skipping")
    input.next()
}
processVowel(input.next())
```

- Ways to replace/fix invalid data
 - use the previously used value
 - use a neutral value
 - use the next valid entry/element
 - find closest legal value

Avoiding Unchecked Input/Changes

```
class Period(_start: Date, _end: Date) {  
    private val _start = new Date(_start.getTime)  
    private val _end = new Date(_end.getTime)  
    if (start.compareTo(end) > 0)  
        throw new IllegalArgumentException(...)  
  
    def start = _start.clone().asInstanceOf[Date]  
    def end = _end.clone().asInstanceOf[Date]  
}
```

```
class Period(val start: Date, val end: Date) {  
    if (start.compareTo(end) > 0)  
        throw new IllegalArgumentException(...)  
}
```

This is known as
“defensive copying”

```
val s = new Date  
val e = new Date  
val p = new Period(s, e)  
e.setYear(78) // Modifies internals of p  
p.end.setYear(78)
```

Assertions Check Assumptions

`assert`(invariant, details)

- Checks for “impossible” conditions
- Catch bugs during development
 - mismatched interface assumptions
 - errors caused by modified code
- Assertions serve as documentation
 - insurance against future code evolution
- Sanity checks for your program
 - check parameters of non-public methods
 - verify code invariants
 - fulfills a subset of the audit methods’ role

```
class HashMap[K,V] extends AbstractMap[K, V] {  
  // ...  
  this(initCapacity: Int, loadFactor: Float) = {  
    this()  
    if (initCapacity < 0)  
      throw new IllegalArgumentException(...)  
    else if (loadFactor < 0.0)  
      throw new IllegalArgumentException(...)  
    // ...  
  }  
  
  private def resize(newCapacity: Int) = {  
    assert(newCapacity > table.length ||  
           table.length == MAXIMUM_CAPACITY)  
  }  
}
```

Code Invariants

- "Invariant" means "always true"
 - property that is purported to always hold
 - generally restricted to a certain portion of code
 - examples: loop invariant, datatype invariant
- Use asserts to enforce invariants
- Use asserts to catch the impossible
 - e.g., empty default statements

```
object Suit extends Enumeration {  
  CLUBS, DIAMONDS, HEARTS, SPADES  
}  
// ...  
  
suit match {  
  case CLUBS:  
    // ...  
  case DIAMONDS:  
    // ...  
  case HEARTS:  
    // ...  
  case SPADES:  
    // ...  
  case _:  
    throw new AssertionError(suit)  
}
```

Declarations & Initialization: The Dangers

- Improper initialization → fertile source of bugs
 - result: variable has unexpected value
- Know your programming language's semantics !
- Sources of bugs
 - not initialized at all (default in Java, random in C/C++)
 - inconsistent values (e.g., constructor initializes only part of the class)

Declarations & Initialization: What You Can Do

- Move initialization close to declaration

```
val area = 3.14 * radius * radius
val myBoard = new Board(8,8)
```

- Move declaration/initialization close to point of first use

```
var accountIndex = 0
// code using accountIndex
// ...
var done = false
while (!done) {
    // ...
}
```

Exception to the rule...

```
var dist = 0.0
try {
    // lots of code
    dist = distance()
}
catch {
    // handle ...
}
retVal = 2 * dist
```

Why Do Errors Happen?

- Humans have a limited ability to...
 - anticipate all possible **inputs** and **states**
 - **specify** the desired response to each set of valid inputs
 - certain inputs are valid, but not anticipated \Rightarrow desired output to these inputs is not specified
 - understand all possible **execution paths**
 - decision points and loops increase the number of possible executions

So, **check** inputs and states have the properties we need – preconditions and invariants

Understanding the Requirements

“There were so many assumptions in the requirements that were easy to miss or were deemed too trivial by the requirements gatherers. These began showing up when **writing tests**.”

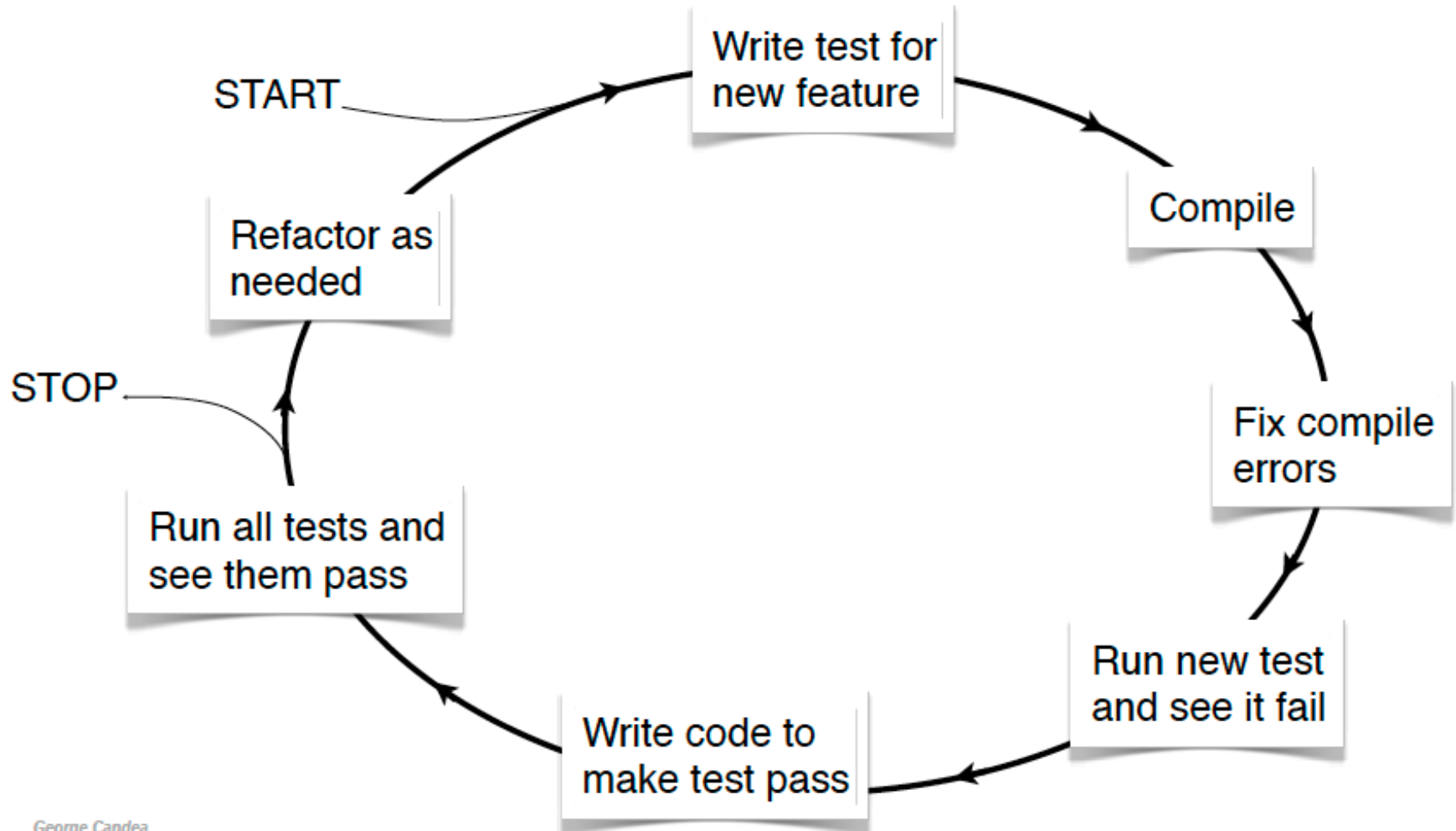
Gaurav Sood, “*Scala Test-Driven Development*”

Test-Driven Development

- **No production code** until you have a failing test case
- You write code to make tests pass
 - you don't write tests to check code you wrote
 - tests drive the design and implementation
- Invest time early in writing tests → save time later
- Reduces bug density

Metric description	IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Defect density of comparable team in organization but not using TDD	W	X	Y	Z
Defect density of team using TDD	0.61W	0.38X	0.24Y	0.09Z
Increase in time taken to code the feature because of TDD (%) [Management estimates]	15–20%	25–35%	15%	25–20%

How To Do TDD?



Writing Good Unit Tests

- Small and check only one thing
 - each unit test focused on at most one functionality
- Independent of each other
- Independent (as much as possible) of interface and implementation
 - especially when it comes to object creation
- Tests should be **automated** → use a test framework
 - e.g., JUnit, ScalaTest, ScalaCheck
 - simplifies and speeds up test development and management
 - integrated with popular IDEs

ScalaCheck (<https://www.scalacheck.org>)

```
import org.scalacheck._
import Prop.forAll

object Q_Text extends org.scalacheck.Properties("Text") {

  property("insert at start") =
    forAll { (s: String) =>
      val t = new Text(); t.insert(0, s)
      t.toString() == s }

  property("insert at end") =
    forAll { (s1: String, s2: String) =>
      val t = new Text(s1); t.insert(t.length, s2)
      t.toString() == s1 + s2 }

}
```

+ Text.insert at start: OK, passed 100 tests.

+ Text.insert at end: OK, passed 100 tests.

When To Use TDD?

- Good candidates
 - user interface behavior (button enabling, button logic, models, etc.)
 - business logic
 - pretty much any Java/Scala class or method
- Bad candidates
 - user interface appearance (layout, colours, etc.)
 - client/server interactions (will need to do mock testing)
 - large code bases, legacy code
- Start TDD from the beginning of the project
 - code coverage >90%
 - add a test for every problem found
 - be disciplined, test continuously

Testing Larger Programs

- Decomposition: test at different levels
 - **unit testing**: test individual pieces of the system
 - **integration testing**: test how pieces are assembled into more complex subsystems
 - **system testing**: test the entire application (end-to-end)

Types of Testing



Black Box

```
trait Map[K,V] {  
  def size: Int  
  def isEmpty: Boolean  
  def contains(key: K): Boolean  
  def apply(key: K): V  
  def iterator: Iterator[(K,V)]  
  def +(kv: (K,V)): Map[K,V]  
  def -(k: K): Map[K,V]  
}
```

- no knowledge about internal organization of a component
- influenced by knowledge of components' interfaces



White Box

```
class HashMap[K,V](val initCapacity: Int,  
  val loadFactor: Float) extends Map[K,V] {  
  // ...  
  if (initCapacity < 0 || loadFactor <= 0)  
    throw new IllegalArgumentException(...)  
  var capacity = 1  
  while (capacity < initCapacity)  
    capacity <<= 1  
  table = new Entry(capacity)  
  init()  
}  
// ...
```

- influenced by knowledge of components' internals

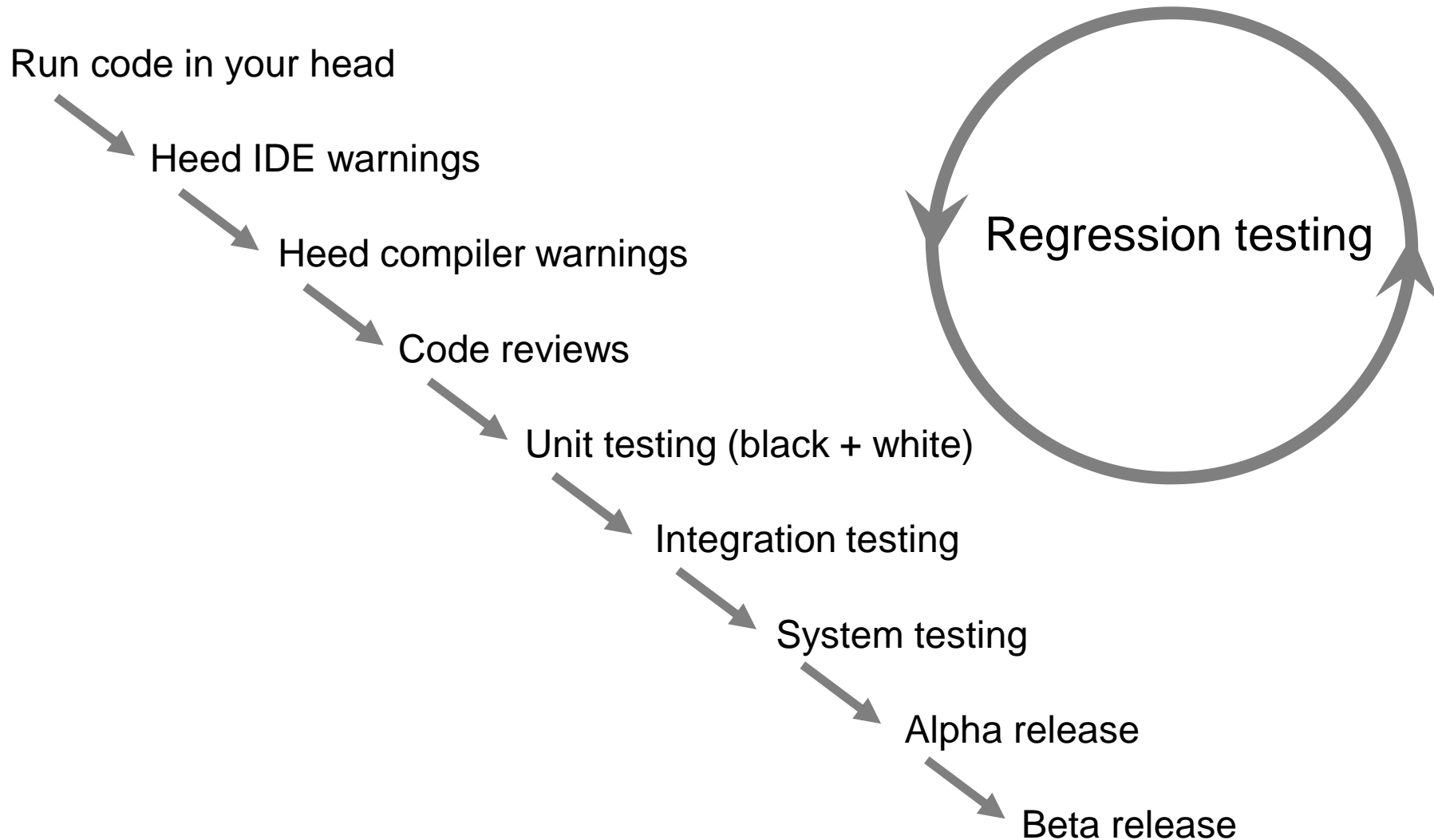
Equivalence Classes

- One check \rightarrow two classes
 - e.g. input x must be < 10
 - equiv. class #1: $x < 10$
 - equiv. class #2: $x \geq 10$
- Two checks \rightarrow three classes
 - e.g., valid x is in range $(0, 100)$
 - equiv. class #1: $0 < x < 100$
 - equiv. class #2: $x \leq 0$
 - equiv. class #3: $x \geq 100$
- Test at least one value in each class

Boundary Testing

- Most programs fail at input boundaries
- If valid input $\in [min \dots max]$, test
 - $x = min$ and $x = max$
 - $x < min$ and $x > max$
- Same for boundaries of data structures (e.g., arrays)

Traditional Quality Assurance



Regression Testing

- A change in a system can break seemingly unrelated parts
 - this is called a **regression**
- Preventing regressions:
 - change the system (e.g., fix the bug)
 - add tests that demonstrates correctness of the change
 - **run the entire test suite** in order to discover possible regressions
 - fix possible regressions
- If you do not run tests immediately, it will be hard to discover what caused the regression

Other Kinds of Testing

- Acceptance testing
 - performed by user upon receiving product
- Smoke/sanity testing
 - quick test to check for serious errors
 - e.g., does it compile? Does it do the basic stuff?
- Compatibility testing
 - does app work with other hw / sw?
 - e.g., web app with smartphone
- Fault injection
 - does app work in the presence of bad inputs, bad returns from libraries, etc.
 - can inject exceptions, simulate failures
- Performance testing
 - goal is to check performance characteristics
 - load/stress/scalability testing
- Usability testing
 - can users accomplish their objectives with the software as designed?

Why do Errors Happen?

- Humans have a limited ability to...
 - anticipate all possible **inputs** and **states**
 - **specify** the desired response to each set of valid inputs
 - certain inputs are valid, but not anticipated \Rightarrow desired output to these inputs is not specified
 - understand all possible **execution paths**
 - decision points and loops increase the number of possible executions

So, **check** inputs and states have the properties we need – preconditions and invariants

So, write **tests** before code to clarify specification
- postconditions

...and keep running them (automatically) every time something **changes**

Metrics

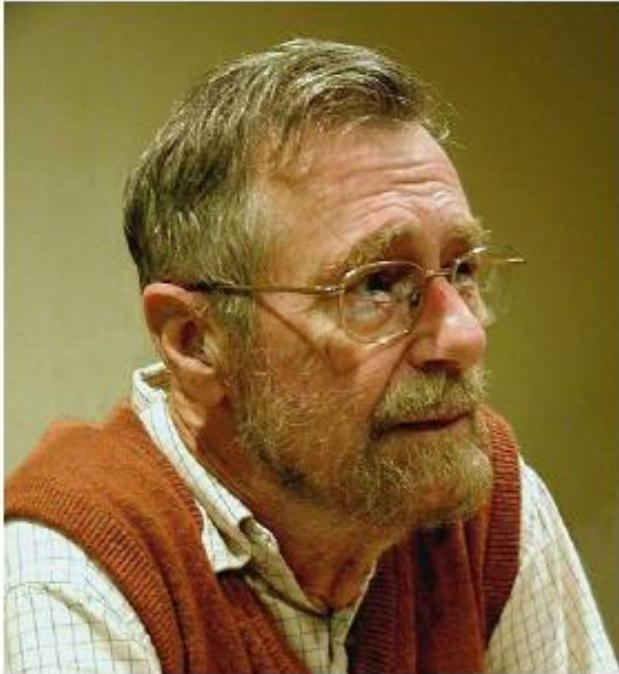
X is **covered** if it is executed at least once by at least one test

Coverage = % covered of total available

Tension between quality vs. cost

- X = methods → method / function coverage
- X = statements → statement / line / basic-block coverage
- X = branches → branch coverage
- X = paths → path coverage

Limitations of Testing



“Program testing can be used to show the presence of bugs, but never to show their absence.”

— Edsger Dijkstra

“Code Smells”

A software bug is an error, flaw, mistake, failure, fault or “undocumented feature” in a computer program that prevents it from behaving as intended [Wikipedia].

Code smells are usually not bugs—they are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. [Wikipedia]

“Code Smells”

- Bloaters
 - **Long method:** any method longer than 10 lines (**decompose!**)
 - **Large class:** any class larger than 200 lines (**decompose!**)
 - **Long parameter lists:** more than 3 or 4 parameters (**abstract!**)
- Expendables
 - **Code duplication:** same functionality in many places (**abstract!**)
 - **Excessive comments:** because code too complex (**decompose!**)
- Couplers
 - **Inappropriate intimacy:** violating encapsulation (**abstract!**)
 - **Feature envy:** methods that over-use another class (**refactor!**)

REFACTORING

IMPROVING THE DESIGN
OF EXISTING CODE

MARTIN FOWLER

With contributions by **Kent Beck, John Brant,
William Opdyke, and Don Roberts**

Foreword by **Erich Gamma**
Object Technology International, Inc.



"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

– Martin Fowler

Conclusion – how to reduce errors

- Defensive programming

- Check inputs
 - can use exceptions for public methods, assertions for non-public ones
 - discard bad inputs, repair bad inputs
- Document assumptions explicitly
- Check code invariants



See also *Programming in Scala*: Chapter 14

- Test-driven development

- Write tests before production code
- It's not a silver bullet



- Avoiding code smells

- **Next:** Reducing errors by polymorphism...

