EN.601.422 / EN.601.622

Software Testing & Debugging

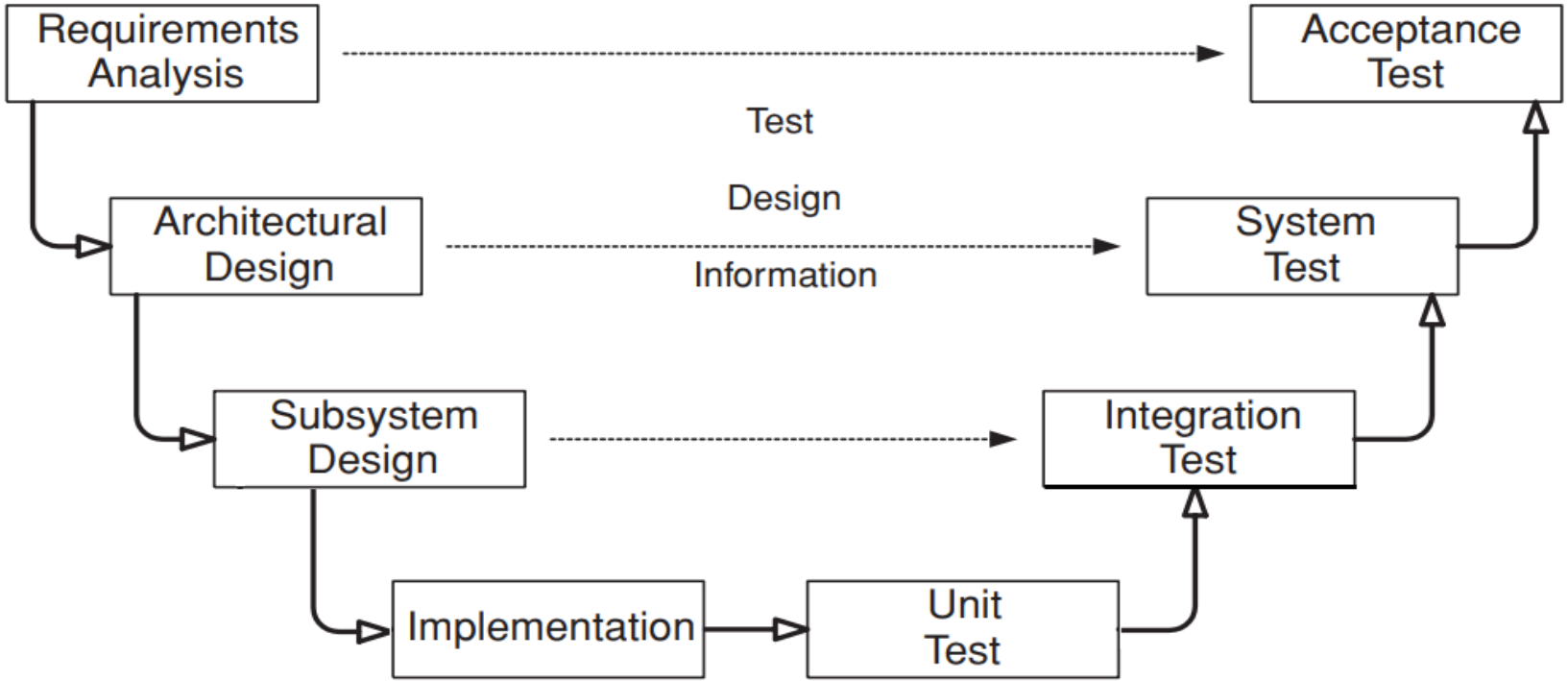
The Goal of Software Testing

- In the last class we discussed:
  - Why software testing matters
  - “the goal of software testing”
    - show presence of bugs, not their absence
  - Fault vs error vs failure
  - Testing is a destructive process!
  - Verification vs. Validation

Plan for Today

- The **V** model, Test early rather than late
- RIPR Fault/Failure Model
- Oracle problem
- Review Few (More) Testing principles

The V Model

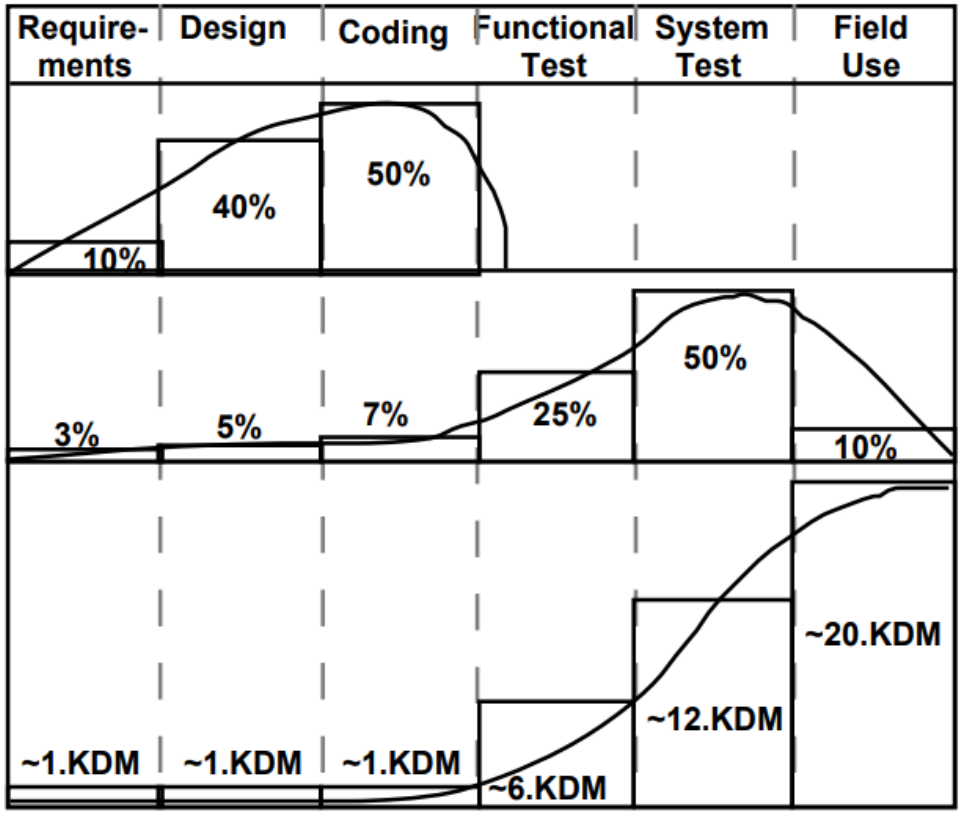


Read more at https://en.wikipedia.org/wiki/V-Model_(software_development)

Testing Levels

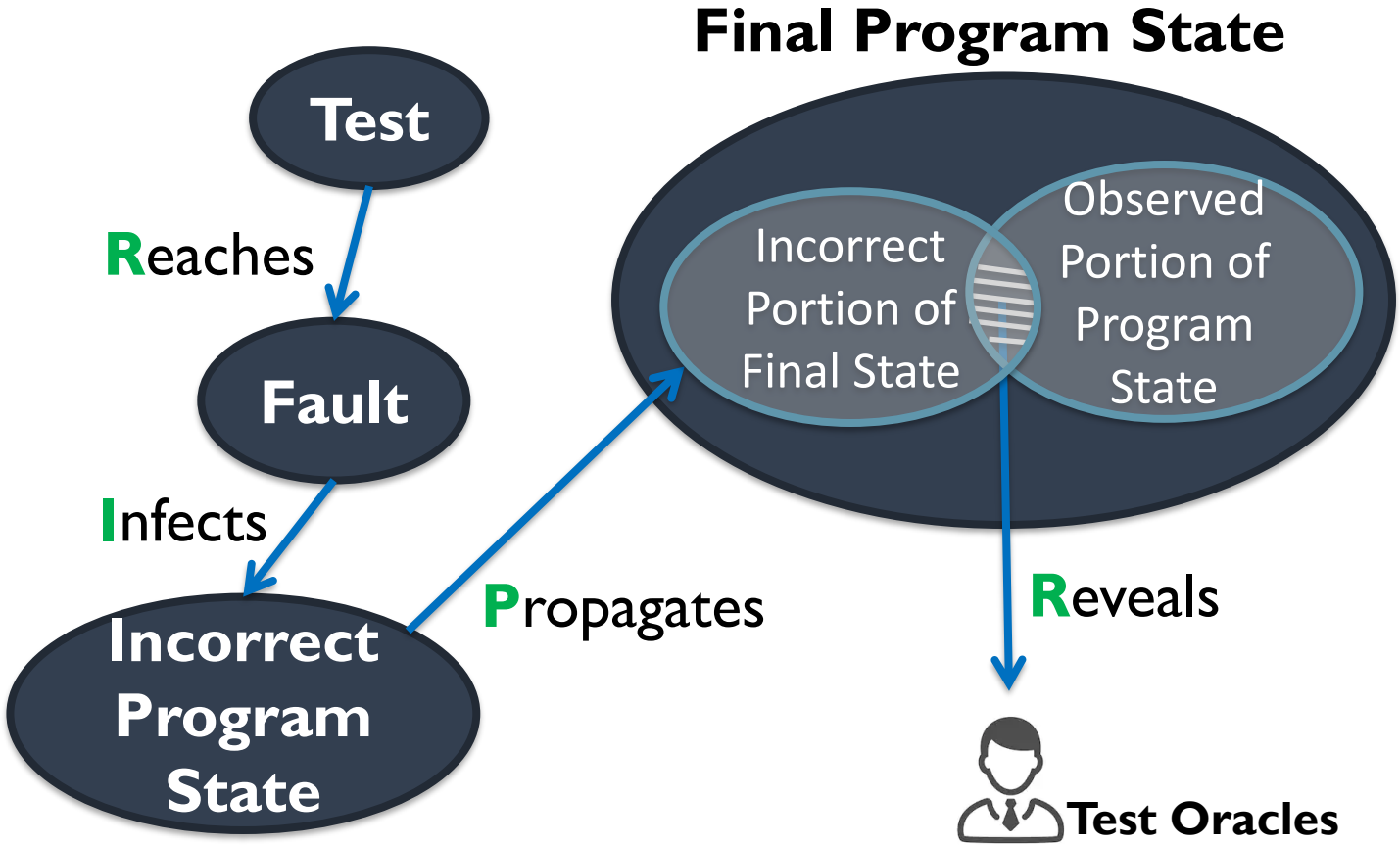
- **Acceptance Testing:** assess software with respect to user's needs
  - Alpha & Beta Testing
- **System Testing:** assess software with respect to architectural design and overall expected behavior
- **Integration Testing:** assess software with respect to sub-system design
- **Unit Testing:** assess software with respect to implementation of isolated modules/units

Faults as a Cost Driver



Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002

RIPR or Fault/Failure Model



RIPR Model

- **Reachability:** The location or locations in the program that contain the fault must be reached
- **Infection:** The state of the program must be incorrect
- **Propagation:** The infected state must cause some output or final state of the program to be incorrect
- **Reveal:** The tester must observe part of the incorrect portion of the program state

Test Automation

- **Two Types of Testing:**
  - Manual
  - Automated
- **Test Automation:** automation of testing-related activities
  - **Generation:** generate test cases automatically
  - **Execution:** run tests on the software under test (SUT)
  - **Evaluation:** evaluate test results i.e., does the test case pass or fail

Test Evaluation

- **Test Oracle:** a mechanism for determining whether a test has passed or failed. An oracle can be:
  - Expected output value
  - A program
  - Documentation that gives specific correct outputs for specific given inputs
  - A (human) domain expert who can tell whether test output is correct or not
  - Any other way or combination of the above that can tell that output is correct or not

Examples

- Unit Testing:
  - E.g., assertEquals(4, sum(2, 2)): 4 is the oracle and hard coded!
  - Is the above oracle complete?

Examples

- Unit Testing:
  - E.g., assertEquals(4, sum(2, 2)): 4 is the oracle and hard coded!
  - Is the above oracle complete?

```
public class MyClass {
   int c;

   public int sum(int a, int b) {
      c  = 10;
      return a + b;
   }
   ...
}
```

More Examples (Other Types of Testing)

- System Testing:
  - E.g., Testing Google search engine with a query
  - What set of results should it return exactly for the query?
- Security Testing:
  - E.g., a test case that simulates a sql injection attack
  - How and what test oracle would you write? i.e., what is exactly the expected behavior?
- Usability testing:
  - E.g., testing the Graphical User Interface of a web app for user friendliness
  - A test case would be accomplishing a task using the GUI. Was it user friendly enough?

Test Oracles

- A ***complete*** Oracle would be based on the entire final state after running a test case
  - Impractical/impossible
- Weak (partial) oracles:
  - Usually, good enough in practice
  - Examples:
    - check for expected output
    - check for software crashes
    - Etc.

Test Evaluation

- One of the most challenging problems in software testing
- Much harder than it might seem; it might not be straightforward what the correct/expected output is/should be
- Requires knowledge of domain, user interfaces, psychology etc.
- This is known as:

The Oracle Problem!

https://en.wikipedia.org/wiki/Test_oracle

Testing Principle 1

A necessary part of any test case is a definition of the expected output/behavior

Testing Principle 2

A test case must not have any logic in it (e.g., must not calculate anything); A test case must merely set things up, make calls, and verify the results.

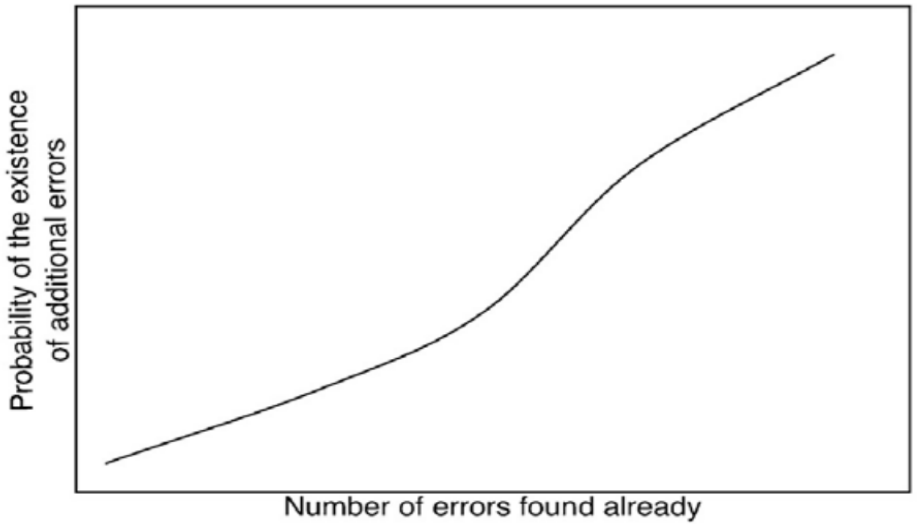
Testing Principle 3

(Ideally) a programmer should avoid attempting to test her own program

Debugging is best done by the original developer though

Testing Principle 4

- Faults are not uniformly distributed
  - The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.



Testing Principle 5

- Examining a program to see if it does what it is supposed to do is only half the battle; the other half is seeing whether the program also does not what it is not supposed to do

More Testing Principles

- Discussed in the last class:
  - Test early
  - Pesticide Paradox
  - Absence of Error Fallacy
  - Testing is context dependent

What is the fault?

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
```

What is the fault?

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[2, 3, 5]; y = 2
   //        Expected = 0
```

What is the fault?

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in x

   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}
```

What is the fault?

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in x

   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[0, 1, 0]
   //        Expected = 2
```

What is the fault?

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in x.
   int count = 0;
   for (int i=0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}
```

What is the fault?

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in x.
   int count = 0;
   for (int i=0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}
   // test:  x=[-4, 2, 0, 2]
   //        Expected = 2
```

What is the fault?

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//      are either odd or positive (or both)
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i]% 2 == 1 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}
```

What is the fault?

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//      are either odd or positive (or both)
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i]% 2 == 1 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}
   // test:  x=[-3, -2, 0, 1, 4]
   //        Expected = 3
```

Can you see a fault?

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Point))
      return false;
    Point p = (Point)o;
    return p.x == x && p.y == y;
  }
  // remainder omitted
}
```

```
public class ColorPoint extends Point {
  private final Color color;
  public ColorPoint(int x, int y, Color color) {
    super(x, y);
    this.color = color;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
      return false;
    return super.equals(o) &&
           ((ColorPoint) o).color == color;
}
```

Taken from Effective Java, 3rd Edition by Joshua Bloch

Can you see a fault?

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Point))
      return false;
    Point p = (Point)o;
    return p.x == x && p.y == y;
  }
  ... // remainder omitted
}
```

```
public class ColorPoint extends Point {
  private final Color color;
  public ColorPoint(int x, int y, Color color) {
    super(x, y);
    this.color = color;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
      return false;
    return super.equals(o) &&
           ((ColorPoint) o).color == color;
}
```

```
// Tests
Point p = new Point(1, 2);
ColorPoint cp1;
cp1 = new ColorPoint(1, 2, Color.RED);
ColorPoint cp2;
cp2 = new ColorPoint(1, 2, Color.BLUE);
p.equals(cp1); // Test 1: result true
cp1.equals(p1); // Test 2: result false
```

equals should be symmetric!

Taken from Effective Java, 3rd Edition by Joshua Bloch

Let’s try to fix

```
@Override
public boolean equals(Object o) {
  if (!(o instanceof Point))
    return false;
  // If o is a normal Point, do a color-blind comparison
  if (!(o instanceof ColorPoint))
    return o.equals(this);
  // o is a ColorPoint, do a full comparison
  return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Let's try to fix

```
@Override
public boolean equals(Object o) {
  if (!(o instanceof Point))
    return false;
  // If o is a normal Point, do a color-blind comparison
  if (!(o instanceof ColorPoint))
    return o.equals(this);
  // o is a ColorPoint, do a full comparison
  return super.equals(o) && ((ColorPoint) o).color == color;
}

// Test
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
p1.equals(p2); // returns true
p2.equals(p3); // returns true
p1.equals(p3); // returns false
```

Violation of transitivity

What is a solution then?

- Turns out this is a fundamental problem of equivalence relations in object-oriented languages
  - there is no way to extend an instantiable class and add a value component while preserving the equals contract
- A reasonable workaround is to use composition in place of inheritance

```
// Adds a value component without
// violating the equals contract
public class ColorPoint {
  private final Point point;
  private final Color color;
  …
```