

Faults, Errors, Failures

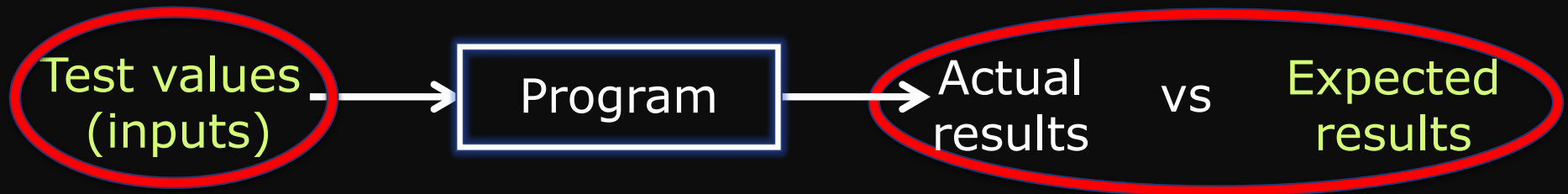
RIPR Model

CS 3250

Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch.1, Ch. 2.1]

Software Testing



Testing can only reveal the presence of faults;
Not showing the absence of faults

Test case consists of

- test input value(s)
- expected result(s)

Will all inputs trigger a problem in software? Will an input that triggers a problem always lead to a software failure?



[Ref: emoji by Ekarin Apirakthanakorn]

Today's Objectives

- Understand the differences between faults, errors, and failures
- Understand how faults, errors, and failures affect the program
- Understand the four conditions that must be satisfied when designing tests

- Reachability
- Infection
- Propagation
- Reveability

“**RIPR** model”

Bug?

"'Bug' – as such little faults and difficulties are called – show themselves, and months of anxious watching, study, and labor are requisite before commercial success – or failure – is certainly reached." [Thomas Edison, 1878]

[Ref: Did You Know? Edison Coined the Term "Bug", <http://theinstitute.ieee.org/tech-history/technology-history/did-you-know-edison-coined-the-term-bug>, IEEE 2013]



"A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways."

[Ref: https://en.wikipedia.org/wiki/Software_bug]

- **"Bug"** is used informally.
- Fault? Error? Or failure?
- This course will try to use words that have **precise**, **defined**, and **unambiguous** meaning – and avoid using the term "bug"



Fault, Error, and Failure in Action

Consider the following pseudo. Trace through & execute it

Test 1: doSomething(5,3)

```
doSomething (int x, int y)
  if (x >= y)
    clap 3 times
  else
    stomp 5 times
```

Correct
code

```
doSomething (int x, int y)
  if (x > y)
    clap 3 times
  else
    stomp 5 times
```

Faulty
code

Test 2: doSomething(5,5)

What is the **expected** output?

What is the **actual** output?

Fault, Error, and Failure

- **Fault**: a **static defect** in the software's source code
 - Cause of a problem – “fault location”
- **Error**: An **incorrect internal state** that is the manifestation of some fault
 - Erroneous/infected program state caused by execution of the defect
- **Failure**: **External, incorrect behavior** with respect to the requirements or other descriptions of the expected behavior
 - Propagation of erroneous state to the program outputs



Let's pause and think!!

**Why do we care about
“fault, error, and failure”?**

**How can we benefit from
analyzing them?**

Example

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
    if (arr[i] == 0)
      count++;
  return count;
}
```

- There is a simple fault in `numZero`
- Where is the fault **location** in the source code?
- How would you fix it?
- Can the fault location be **reached**? How does it corrupt program **state**? Does it always corrupt the program state?
- If the program state is corrupted, does `numZero` **fail**? How?

Example – Let's Analyze

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
    if (arr[i] == 0)
      count++;
  return count;
}
```

- **Fault:** a defect in source code
i = 1 [should start searching at 0, not 1]
- **Error:** erroneous program state caused by execution of the defect
i becomes 1 [array entry 0 is not ever read]
- **Failure:** propagation of erroneous state to the program outputs
Happens as long as arr.length > 0 and arr[0] = 0

Example – Test Cases

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
    if (arr[i] == 0)
      count++;
  return count;
}
```

Fault: i = 1 [should start searching at 0, not 1]

- Test 1: [4, 6, 0], expected 1

Error: i is 1, not 0, on the first iteration

Failure: none

- Test 2: [0, 4, 6], expected 1

Error: i is 1, not 0, error propagates to the variable count

Failure: count is 0 at the return statement

Example – State Representation

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
L1  ↓ int count = 0;
L2   for ↓ int i = 1; ↓ i < arr.length; i++)
L3       if (arr[i] == 0)
L4         count++;
L5   return count;
}
```

- Assume that we want to represent program states using the notation $\langle \text{var}_1 = v_1, \dots, \text{var}_n = v_n, \text{PC} = \text{program counter} \rangle$
- Sequence of states in the execution of `numZero({0, 4, 6})`
 - 1: $\langle \text{arr}=\{0, 4, 6\}, \text{PC}=[\text{int count}=0 \text{ (L1)}] \rangle$
 - 2: $\langle \text{arr}=\{0, 4, 6\}, \text{count}=0, \text{PC}=[i=1 \text{ (L2)}] \rangle$
 - 3: $\langle \text{arr}=\{0, 4, 6\}, \text{count}=0, i=1, \text{PC}=[i<\text{arr.length} \text{ (L2)}] \rangle$
 - ...
 - $\langle \text{arr}=\{0, 4, 6\}, \text{count}=0, i=3, \text{PC}=[\text{return count (L5)}] \rangle$

Example – Error State

- Error state

- The **first different** state in execution in comparison to an execution to the state sequence of what would be the correct program

- If the code had $i=0$ (correct program), the execution of `numZero({0, 4, 6})` would be

1: < arr={0, 4, 6}, PC=[int count=0 (L1)] >

2: < arr={0, 4, 6}, count=0, PC=[i=0 (L2)] >

3: < arr={0, 4, 6}, count=0, i=0, PC=[i<arr.length (L2)] >

...

- Instead, we have

1: < arr={0, 4, 6}, PC=[int count=0 (L1)] >

2: < arr={0, 4, 6}, count=0, PC=[i=1 (L2)] >

3: < arr={0, 4, 6}, count=0, i=1, PC=[i<arr.length (L2)] >

...

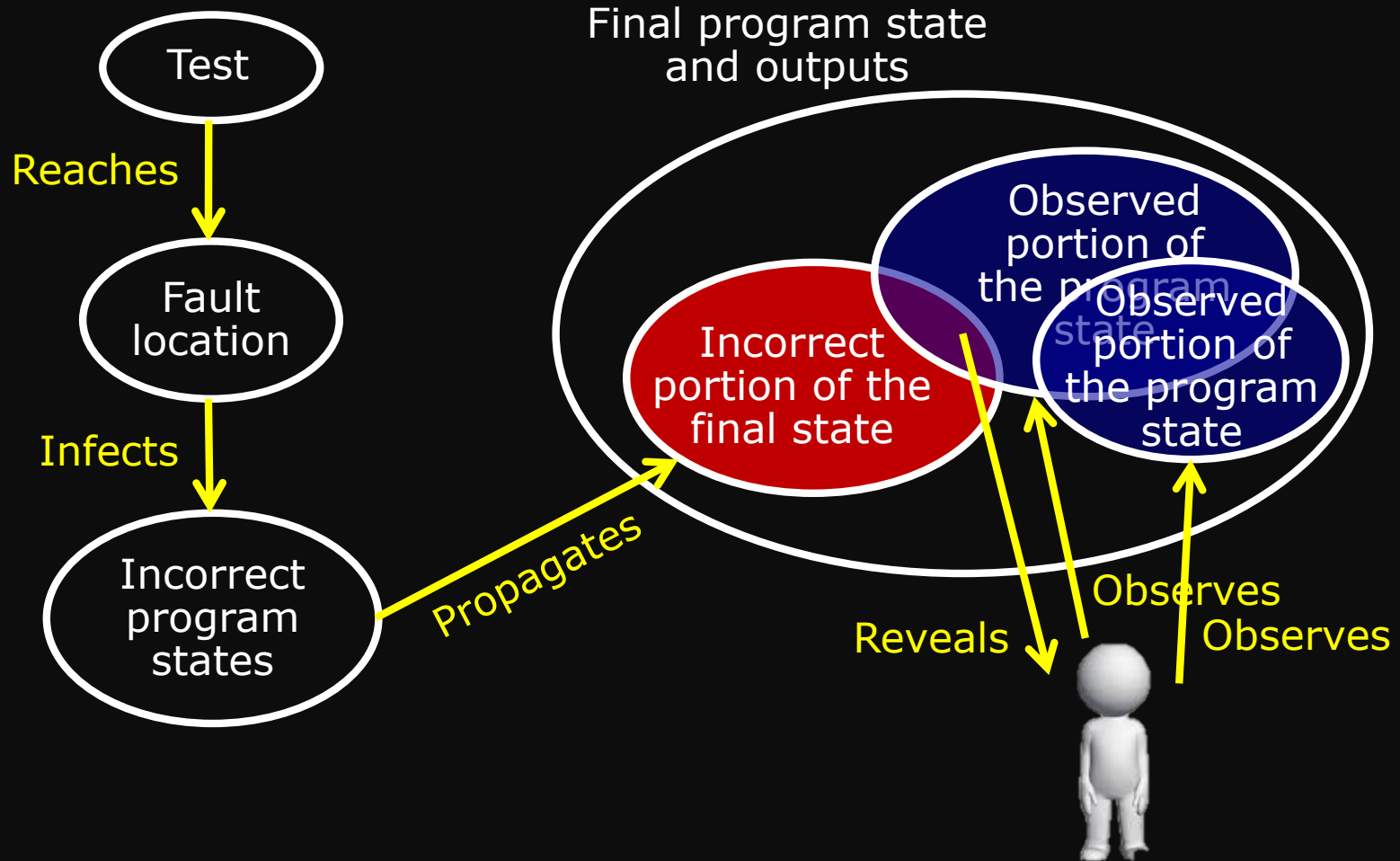
The first error state is immediately after executing $i=1$ (line L2)

RIPR Model

Four conditions necessary for a failure to be observed

- **R**eachability
 - The fault is reached
- **I**nfection
 - Execution of the fault leads to an incorrect program state (error)
- **P**ropagation
 - The infected state must cause the program output or final state to be incorrect (failure)
- **R**evealability
 - The tester must observe part of the incorrect portion of the program state

RIPR Model



[AO, p.21]

Example – Applying RIPR

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
L1   int count = 0;
L2   for (int i = 1; i < arr.length; i++)
L3       if (arr[i] == 0)
L4           count++;
L5   return count;
}
```

Revisit the example, what characteristics (or constraints) the inputs should have (or satisfy)?

- **Reach** a fault (i.e., execute the fault)
- Cause the program **state to be incorrect** (i.e., error)
- Cause the infected state to be **propagated** (i.e., failure)

Did you consider “happy paths” or “non happy paths” ?

Example – RIPR (Error, No Failure)

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
L1   int count = 0;
L2   for (int i = 1; i < arr.length; i++)
L3       if (arr[i] == 0)
L4           count++;
L5   return count;
}
```

Revisit the example, apply RIPR to design tests that

- **Reach** a fault (i.e., execute the fault)
- Cause the program **state to be incorrect** (i.e., error)
- Does **not propagate** (i.e., no failure)
 - One possible test is [4, 6, 0] – now, design some more
- How does RIPR model help designing tests?

Example – RIPR (Error, Failure)

```
public static int numZero (int[] arr)
{ // If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
L1   int count = 0;
L2   for (int i = 1; i < arr.length; i++)
L3       if (arr[i] == 0)
L4           count++;
L5   return count;
}
```

Revisit the example, apply RIPR to design tests that

- **Reach** a fault (i.e., execute the fault)
- Cause the program **state to be incorrect** (i.e., error)
- **Propagate** (i.e., failure)
 - One possible test is [0, 4, 6] – now, design some more
- How does RIPR model help designing tests?

Wrap-up

- Faults, errors, failures
- Fault location
- Infected state
- RIPR model
- Observability and revealability

What's Next?

- Model-Driven Test Design (MDTD)