Faults, Errors, Failures
RIPR Model

CS 3250
Software Testing

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

- **Testing** = process of finding input values to check against a software *(focus of this course)*

  Test case consists of test values and expected results

- **Debugging** = process of finding a fault given a failure

1. Testing is fundamentally about choosing finite sets of values from the input domain of the software being tested

2. Given the test inputs, compare the actual results with the expected results
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
  - Revealability

  “RIPR model”
Fault, Error, and Failure

• **Fault**: a static defect in the software’s source code
  - Cause of a problem

• **Error**: An incorrect internal state that is the manifestation of some fault
  - Erroneous 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
Example

```java
public static int numZero (int[] arr)
{
    // Effects: 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?
public static int numZero (int[] arr) {
    // Effects: 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

```java
public static int numZero (int[] arr)
{
    // Effects: 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;
}
```

- 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

*Fault:* i = 1 [should start searching at 0, not 1]*
Example – State Representation

```java
public static int numZero (int[] arr) {
    int count = 0;
    for (int i = 1; i < arr.length; i++)
        if (arr[i] == 0)
            count++;
    return count;
}
```

- Assume that we want to represent program states using the notation `<var_1 = v_1, ..., var_n = v_n, PC = program counter>`

- Sequence of states in the execution of `numZero({0, 4, 6})`

  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)]>`
  ...
  `<arr={0, 4, 6}, count=0, PC=[return count; (L5)]>`
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 i=1 in line L2
RIPR Model

Four conditions necessary for a failure to be observed

• **Reachability**
  • The fault is reached

• **Infection**
  • Execution of the fault leads to an incorrect program state (error)

• **Propagation**
  • The infected state must cause the program output or final state to be incorrect (failure)

• **Revealability**
  • The tester must observe part of the incorrect portion of the program state
RIPR Model

Test → Reaches → Fault location → Infects → Incorrect program states

Final program state and outputs

Incorrect portion of the final state

Observed portion of the program state

Observes → Reveals → Observes

Reaches 

[AO, p.21]
Example – RIPR (Error, No Failure)

public static int numZero (int[] arr) {
    // Effects: 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;
}

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)

```java
public static int numZero (int[] arr)
{
    // Effects: 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;
}
```

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 revealibility

What’s Next?

- Model-Driven Test Design (MDTD)