

# 软件分析与验证前沿

苏亭

软件科学与技术系

# Software Specifications (and Testing)

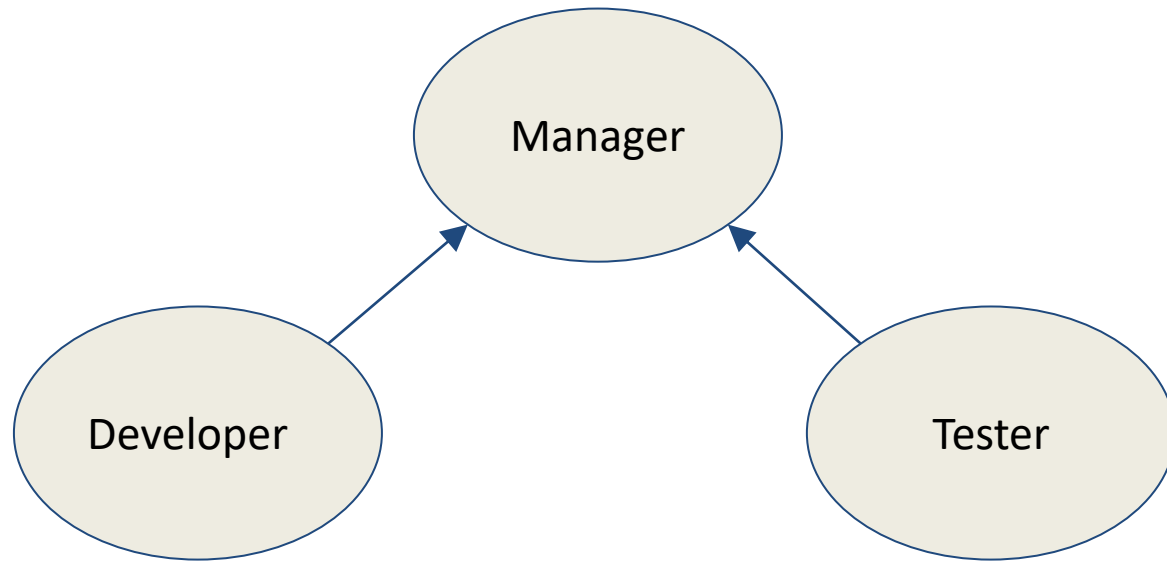
# SEGMENT

---

Software Development Scenario

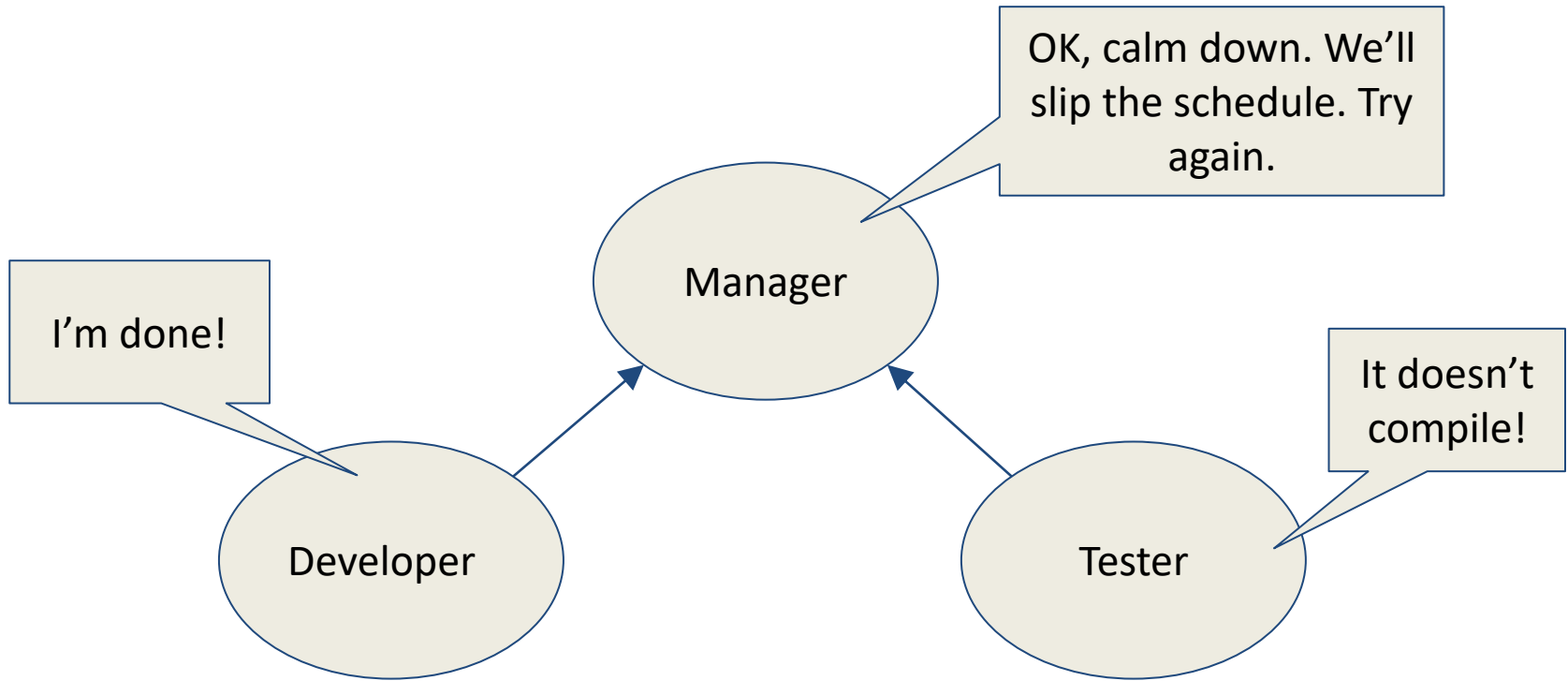
# Software Development Today

---



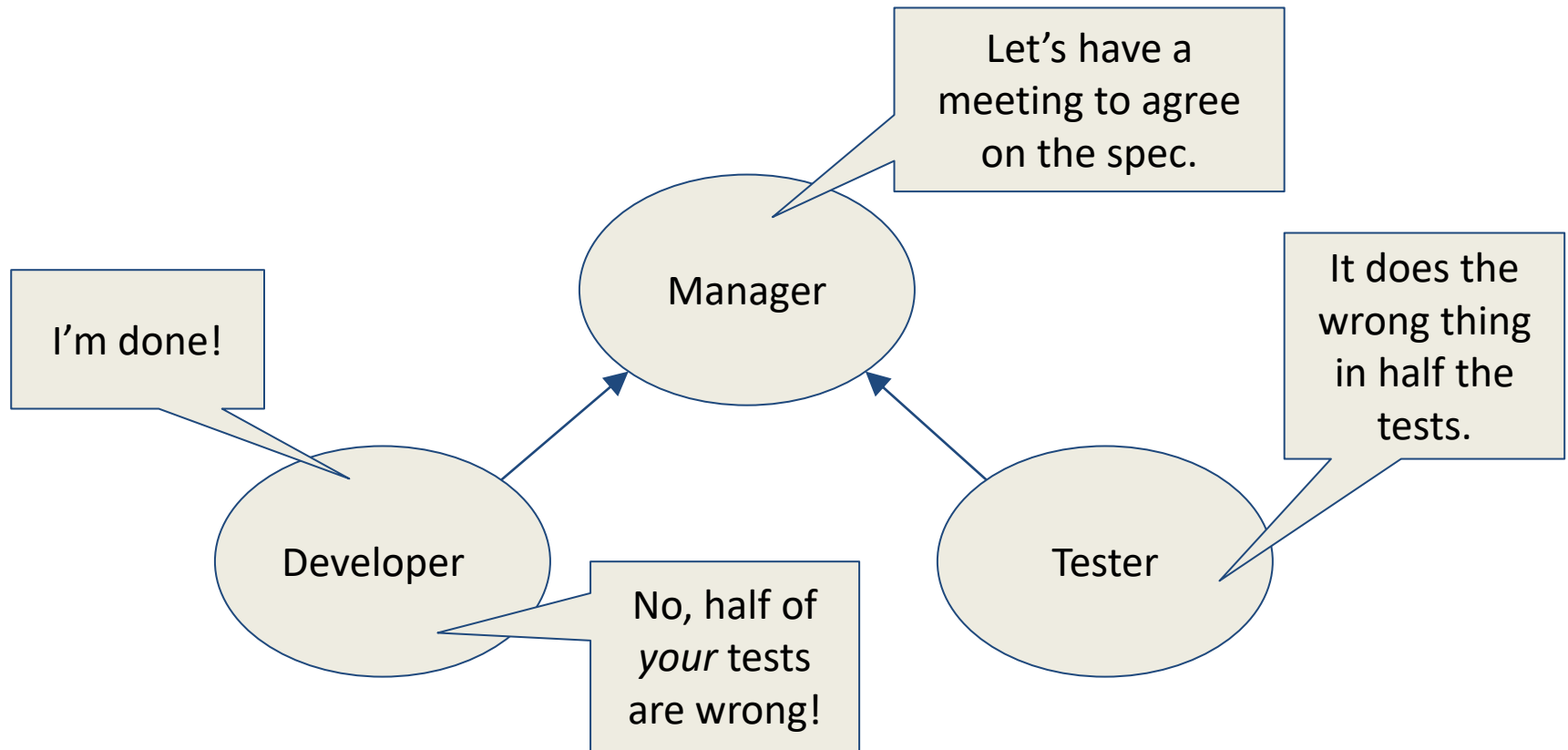
# A Typical Scenario

---



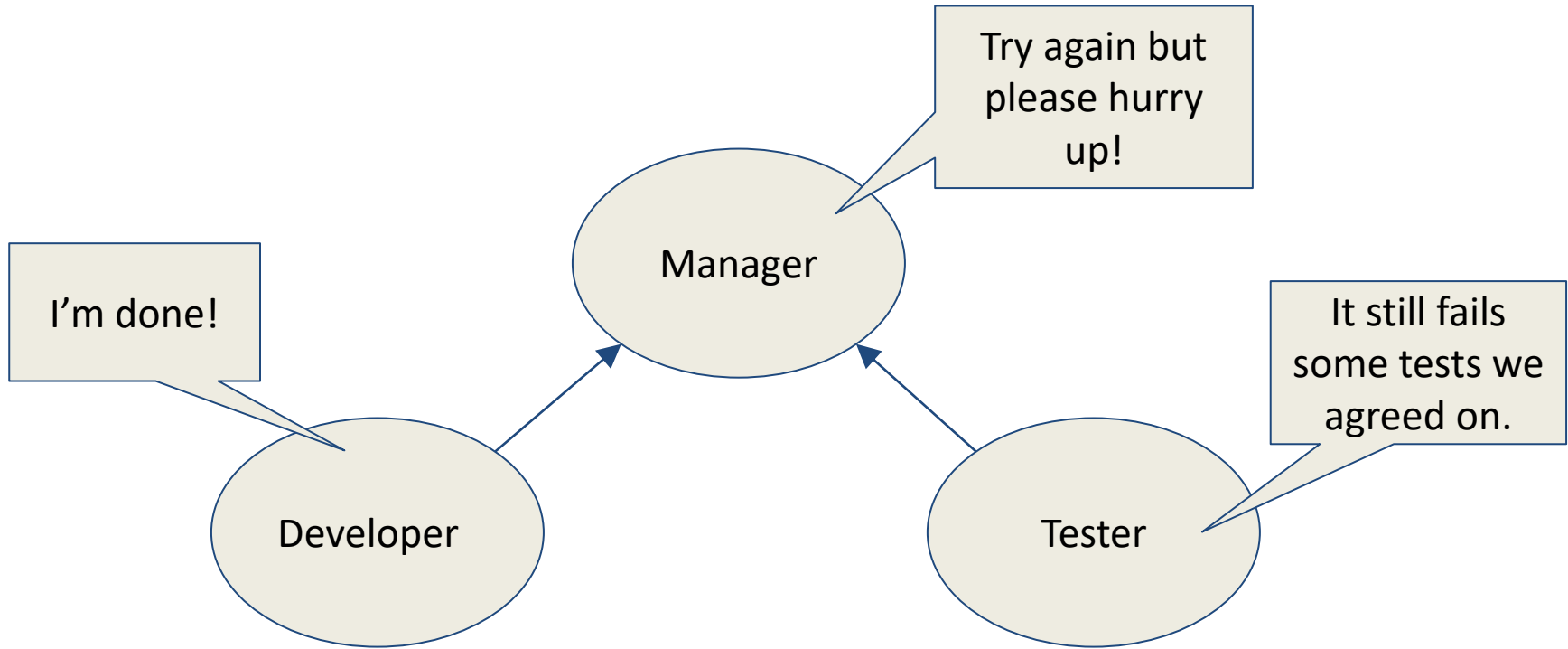
# A Typical Scenario

---



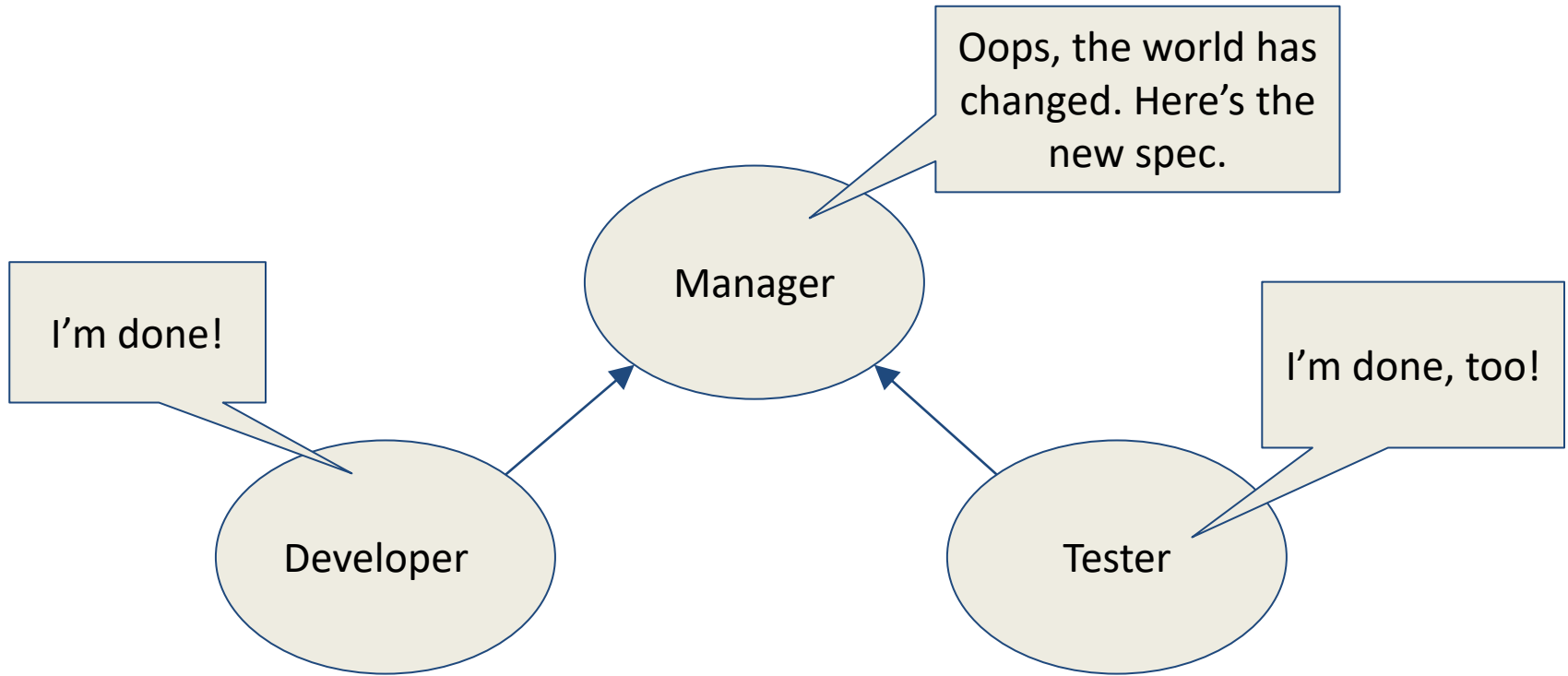
# A Typical Scenario

---



# A Typical Scenario

---



# SEGMENT

---

## The Role of Specifications

# Key Observations

---

- Specifications must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time

# The Need for Specifications

---

- Testing checks whether program implementation agrees with program specification
- Without a specification, there is nothing to test!
- Testing is a form of consistency checking between implementation and specification
  - Recurring theme for software quality checking approaches
  - What if both implementation and specification are wrong?

# Developer != Tester

---

- Developer writes **implementation**, tester writes **specification**
- Unlikely that both will independently make the same mistake
- Specifications useful even if written by developer itself
  - Much simpler than implementation
  - => specification unlikely to have same mistake as implementation

# Other Observations

---

- Resources are finite
  - => Limit how many tests are written
- Specifications evolve over time
  - => Tests must be updated over time
- An Idea: Automated Testing
  - => No need for testers!?

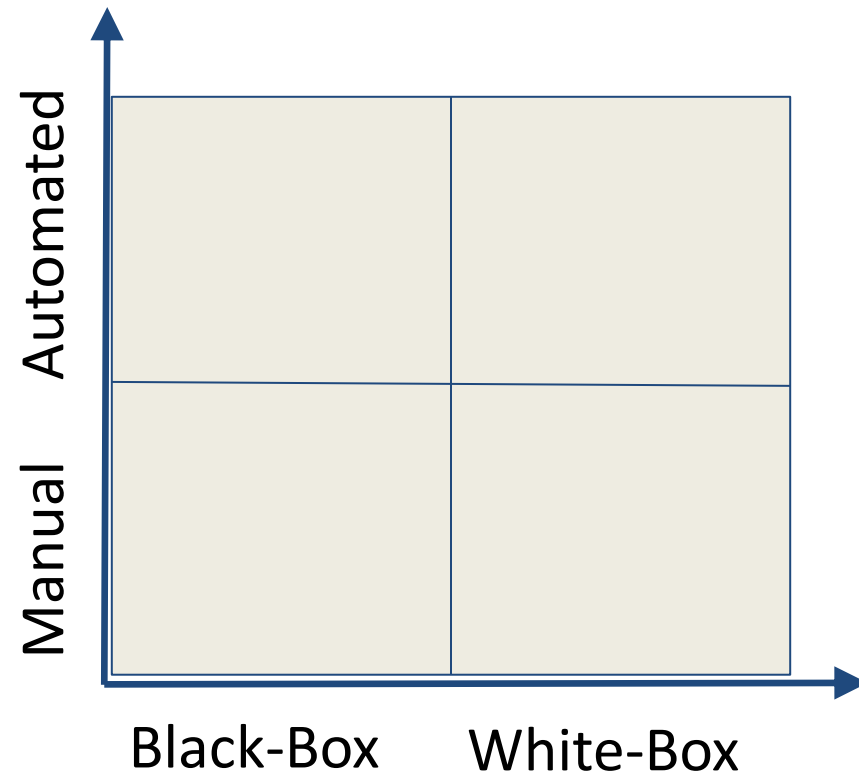
# Outline

---

- Landscape of Testing
- Specifications
  - Pre- and Post- Conditions and Invariants
- Measuring Test Suite Quality
  - Coverage Metrics
  - Mutation Analysis
- Classification of Testing Techniques

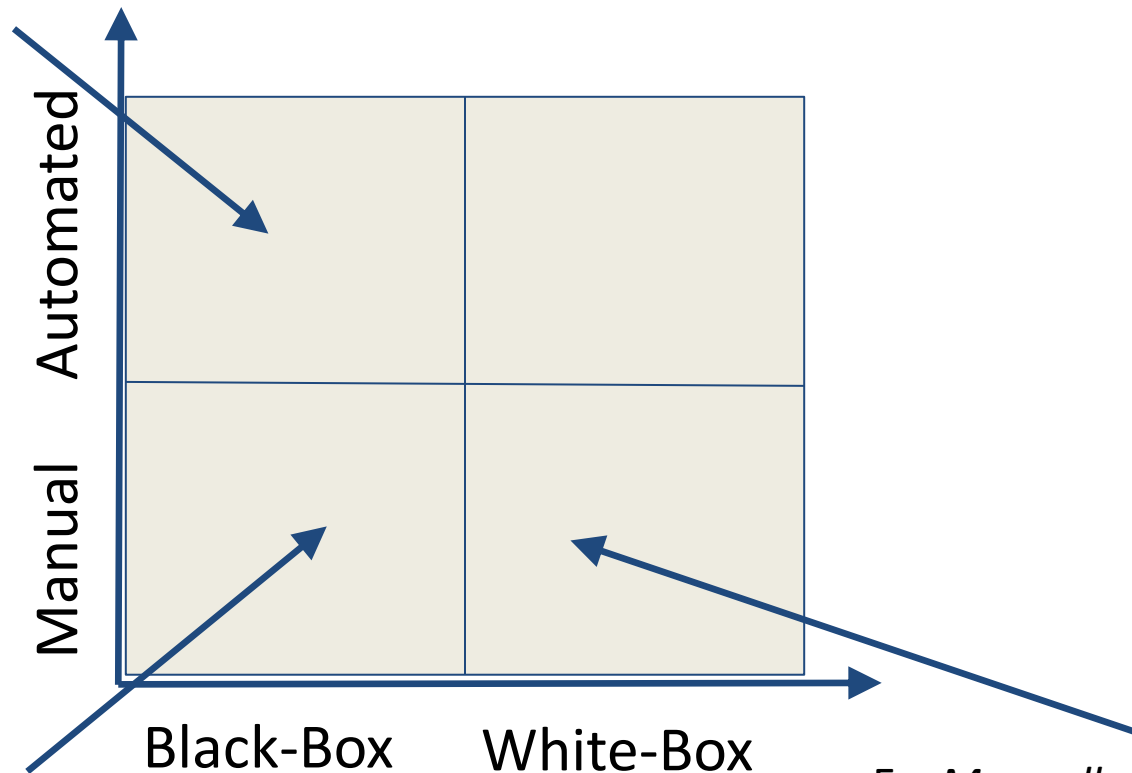
# Landscape of Testing

---



# Landscape of Testing

Ex: *Use Monkey to test an app*

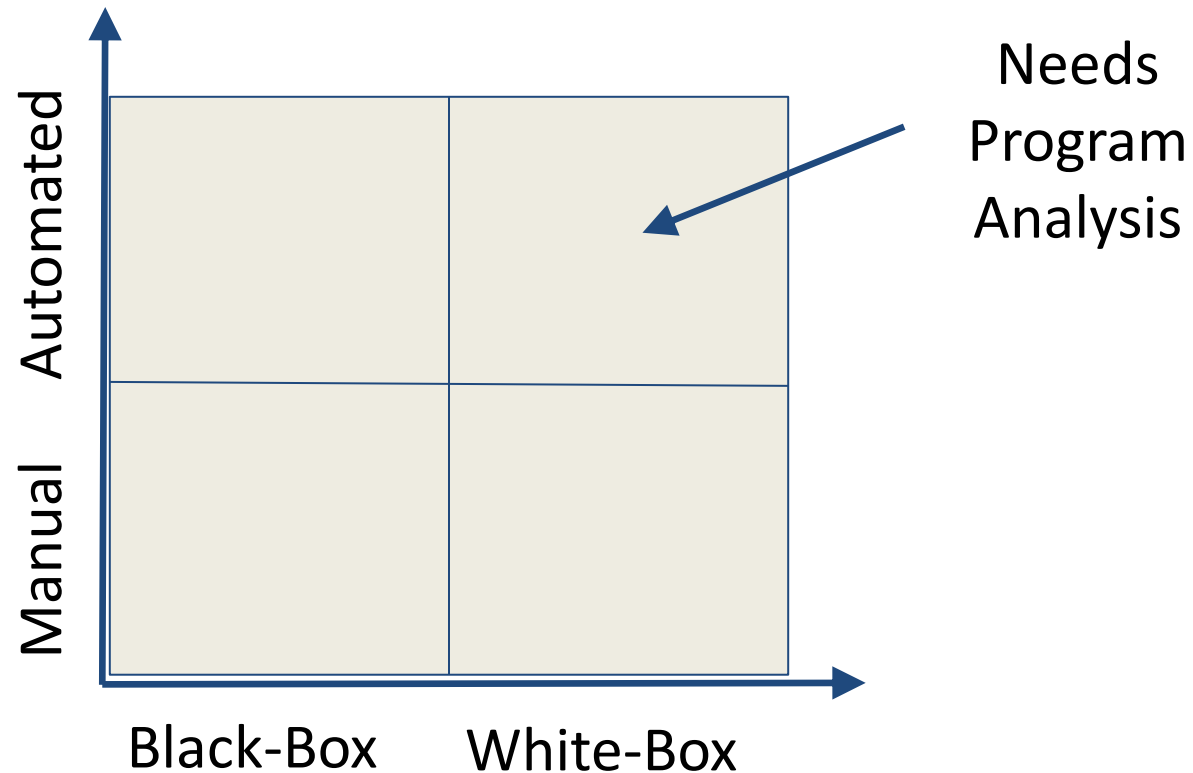


Ex: *Manually testing an app*

Ex: *Manually testing an app by inspecting uncovered code*

# Landscape of Testing

---



# Automated vs. Manual Testing

---

- Automated Testing:
  - Find bugs more quickly
  - No need to write tests
  - If software changes, no need to maintain tests
- Manual Testing:
  - Efficient test suite
  - Potentially better coverage

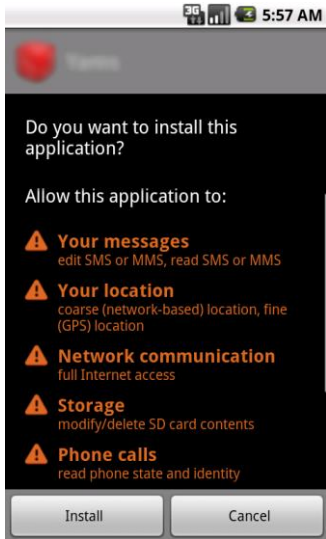
# Black-Box vs. White-Box Testing

---

- Black-Box Testing:
  - Can work with code that cannot be modified
  - Does not need to analyze or study code
  - Code can be in any format (managed, binary, obfuscated)
- White-Box Testing:
  - Efficient test suite
  - Potentially better coverage

# An Example: Mobile App Security

---



```
HttpPost localHttpPost = new HttpPost(...);  
(new DefaultHttpClient()).execute(localHttpPost);
```

[http://\[...\]search.gongfu-android.com:8511/\[...\]](http://[...]search.gongfu-android.com:8511/[...])



# Software Fault, Error and Failures

```
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:** Should start searching at 0, not 1

Test 1  
[ 2, 7, 0 ]  
Expected: 1  
Actual: 1

**Error:** i is 1, not 0, on the first iteration  
**Failure:** none

Test 2  
[ 0, 2, 7 ]  
Expected: 1  
Actual: 0

**Error:** i is 1, not 0  
Error propagates to the variable count  
**Failure:** count is 0 at the return statement

# Software Fault and Failure Model

---

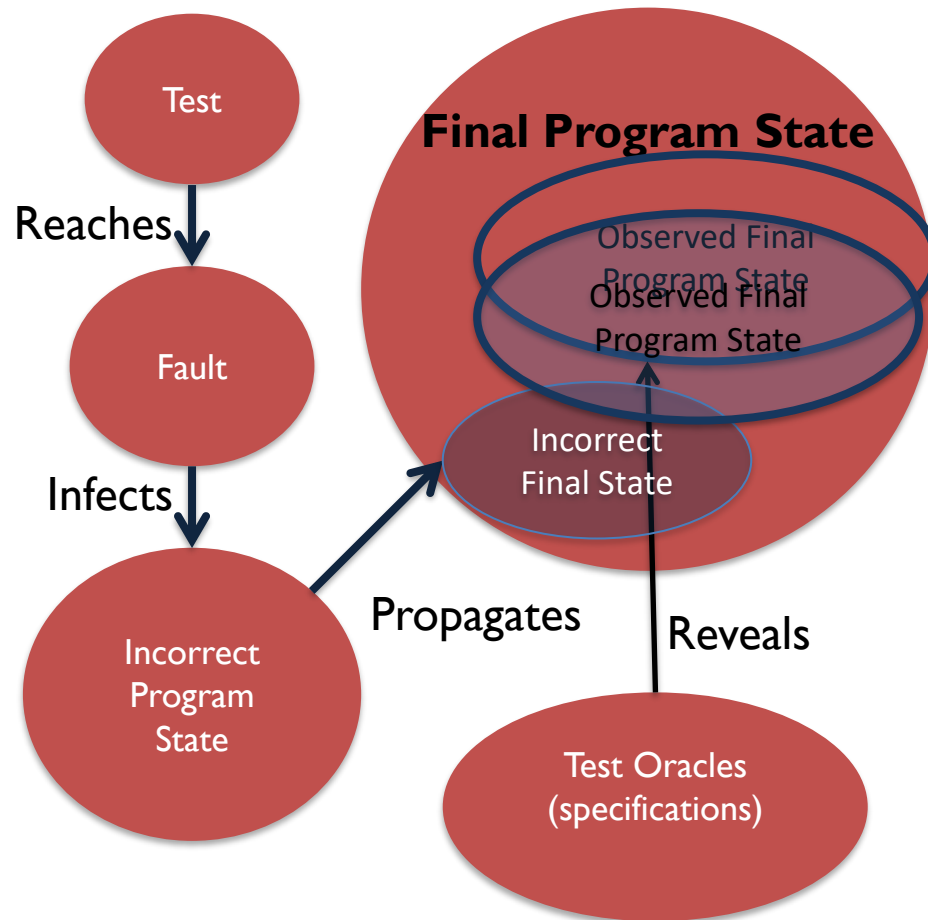
## Four conditions necessary for a failure to be observed

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

# Software Fault and Failure Model

---

- Reachability
- Infection
- Propagation
- Revealability



# The Automated Testing Problem

---

- Automated testing is hard to do
- Probably impossible for entire systems
- Certainly impossible without specifications

# LESSON

---

## Kinds of Specifications

# SEGMENT

---

Safety and Liveness

# Classification of Specifications

---

## Safety Properties

Program will never reach a  
**bad** state

**Examples:** assertions, types,  
type-state properties

## Liveness Properties

Program will eventually reach a  
**good** state

**Examples:** program termination,  
starvation freedom

# Common Forms of Safety Properties

---

- Types

- `int x, java.net.Socket s, ...`

- Type-state Properties

- Program must not read data from `java.net.socket` until socket is connected

**Precondition:** `x > 0`

**Postcondition:** the return value 'ret' satisfies  
`ret * ret == x`

- Assertions

- Implicit, .e.g, `p != null` before each dereference of `p`
- Explicit, e.g., `assert(y == 42)`

- Pre- and Post- Conditions

- `double sqrt(double x) {...}`

- Loop and Class Invariants

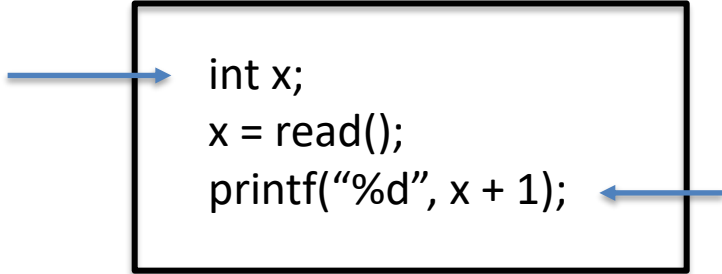
# Types

---

Adding the **specification** to the **implementation** allows to check the implementation against the specification

**Specification:**

x is an integer



```
int x;  
x = read();  
printf("%d", x + 1);
```

**Implementation:**

x is an integer because of  
being added to an integer

# Types

---

Explicit type specification is missing in dynamically-typed or untyped programming languages

```
function getPass(clearTextPass) {  
  if (clearTextPass) return 'PASS';  
  return '****';  
}  
let pass = getpass('false' ); // "PASS"
```

JavaScript:  
Dynamically-typed



```
function getPass(c1earTextPass : boolean) : string {  
  if (clearTextPass) return 'PASS';  
  return '****';  
}  
let pass = getpass('false' ); // error
```

TypeScript:  
Statically-typed



# The Checker Framework

---

- The **Checker Framework** enhances Java's type system  
Goal: statically eliminate entire classes of runtime errors via annotations
- Example: Nullness Checker

```
public class Example {  
    void sample() {  
        @NonNull Object ref = null;  
    }  
}
```



Error:

incompatible types  
found: @Nullable <nulltype>  
required: @NonNull Object

# Type-State Properties

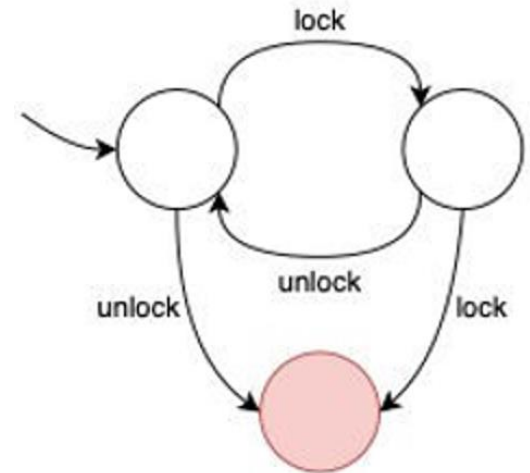
---

- Type-state refines types with (finite) state information  
e.g. File in OPEN state, Socket in CONNECTED state, etc.
- Enables to specify which operations are valid in each state, and how operations affect the state  
e.g. fread() may only be called on File in OPEN state,  
and fclose() changes File state from OPEN to CLOSED
- Also called **temporal safety properties**

# Example Property 1: Locking

---

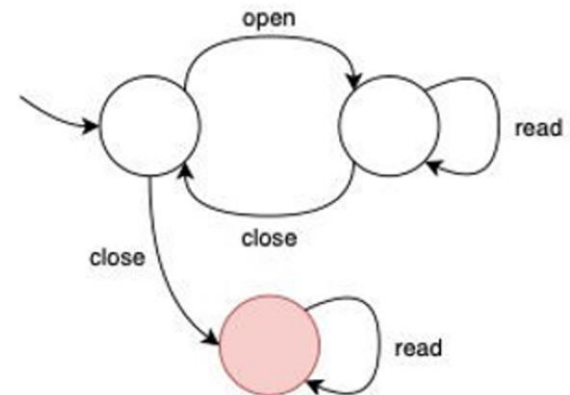
- Calls to lock and unlock must alternate
- Attempting to re-acquire an acquired lock or release a released lock causes an error
- Models behavior of `pthread_mutex_lock()` and in `pthread_mutex_unlock()` in pthreads



# Example Property 2: File Management

---

- A file must be opened before reading
- A file may be read an arbitrary number of times before it is closed
- A file must not be closed twice



# SEGMENT

---

Pre- and Post- Conditions

# Pre- and Post-Conditions

---

- A **pre-condition** is a predicate
  - Assumed to hold before a function executes
- A **post-condition** is a predicate
  - Expected to hold after a function executes, whenever the pre-condition also holds

# Example

---

```
class Stack<T> {  
    T[] array;  
    int size;
```

```
    Pre: s.size() > 0
```

```
    T pop() { return array[--size]; }
```

```
    Post: s'.size() == s.size() - 1
```

```
    int size() { return size; }  
}
```

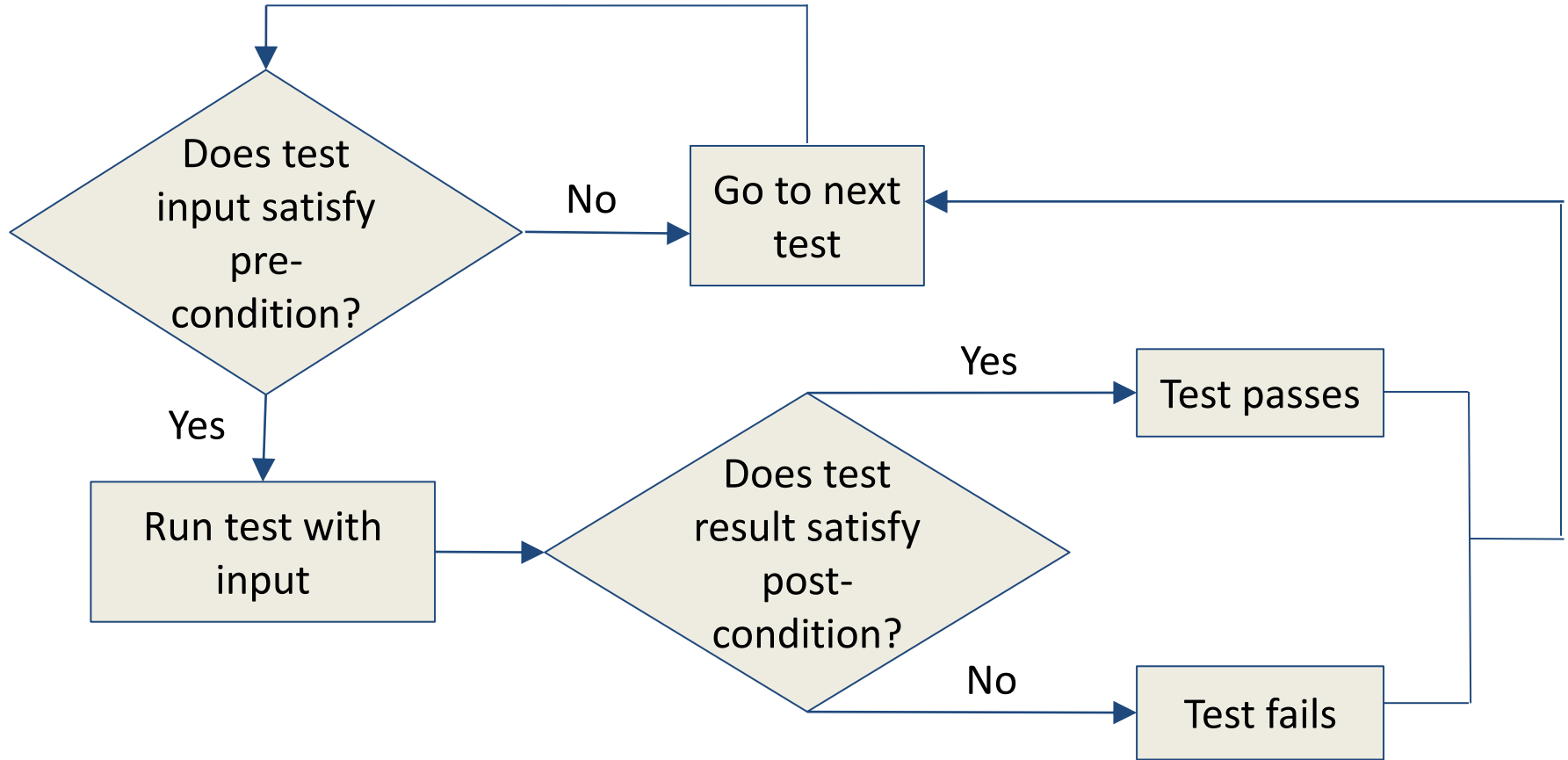
# More on Pre- and Post-Conditions

---

- Most useful if they are executable
  - Written in the programming language itself
  - A special case of assertions
- Need not be precise
  - May become more complex than the code!
  - But useful even if they do not cover every situation

# Using Pre- and Post-Conditions

---



Doesn't help write tests, but helps run them

# QUIZ: Pre-Conditions

---

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

Pre:

```
int foo(int[] A, int[] B) {  
    int r = 0;  
    for (int i = 0; i < A.length; i++) {  
        r += A[i] * B[i];  
    }  
    return r;  
}
```

# QUIZ: Pre-Conditions

---

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

Pre: `A != null && B != null && A.length <= B.length`

```
int foo(int[] A, int[] B) {  
    int r = 0;  
    for (int i = 0; i < A.length; i++) {  
        r += A[i] * B[i];  
    }  
    return r;  
}
```

# QUIZ: Post-Conditions

---

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the **strongest possible post-condition**.

- ☐ B is non-null
- ☐ B has the same length as A
- ☐ The elements of B do not contain any duplicates
- ☐ The elements of B are a permutation of the elements of A
- ☐ The elements of B are in sorted order
- ☐ The elements of A are in sorted order
- ☐ The elements of A do not contain any duplicates

# QUIZ: Post-Conditions

---

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the **strongest possible post-condition**.

- ☒ B is non-null
- ☒ B has the same length as A
- ☐ The elements of B do not contain any duplicates
- ☒ The elements of B are a permutation of the elements of A
- ☒ The elements of B are in sorted order
- ☐ The elements of A are in sorted order
- ☐ The elements of A do not contain any duplicates

# Executable Post-Condition

---

- B is non-null

```
B != null;
```

- B has the same length as A

```
B.length == A.length;
```

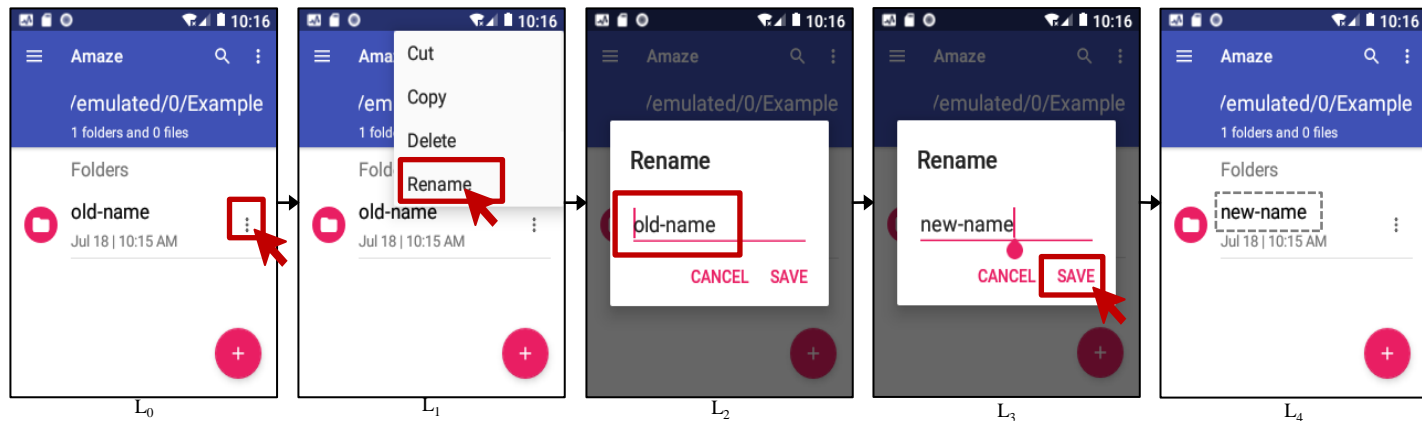
- The elements of B are in sorted order

```
for (int i = 0; i < B.length-1; i++)  
    B[i] <= B[i+1];
```

- The elements of B are a permutation of the elements of A

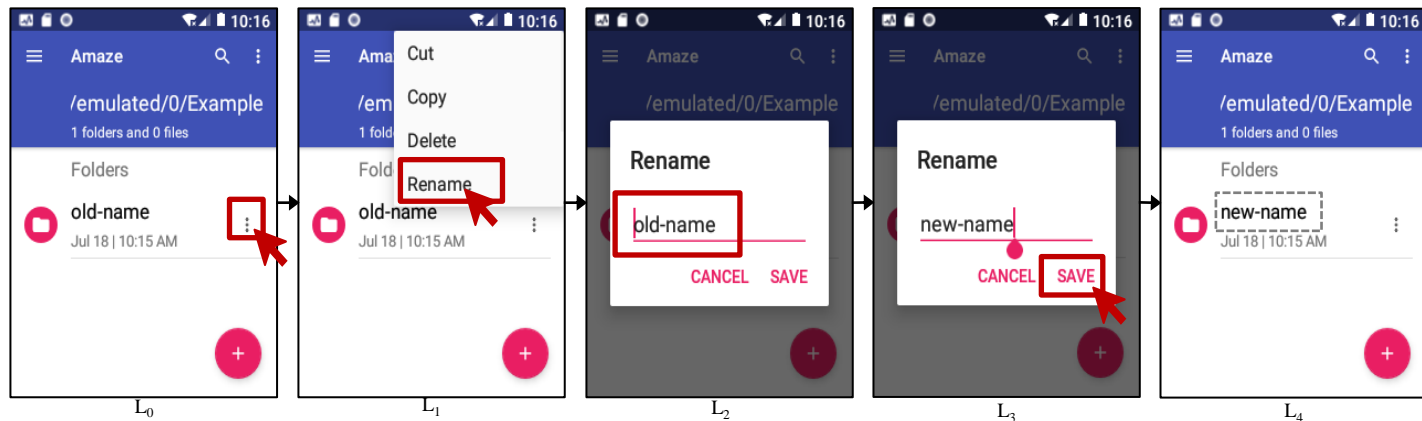
```
// count number of occurrences of  
// each number in each array and  
// then compare these counts
```

# Example: Data Manipulation Functionalities



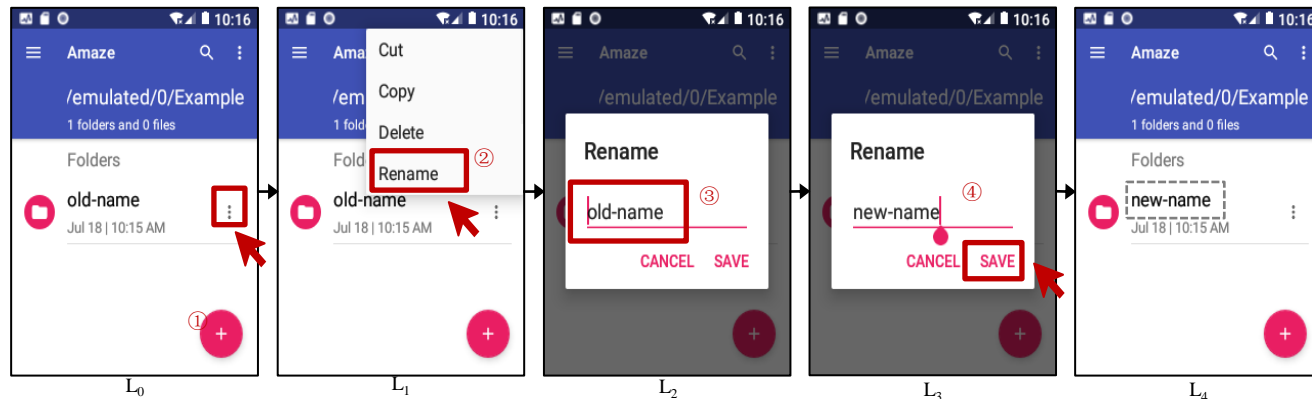
A "Rename" Function for app data "Folder"


# What are the Pre- and Post-conditions?


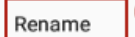
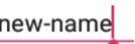



A “Rename” Function for app data “Folder”

# What are the Pre- and Post-conditions?



**P** : a menu icon of folder (  ) exists on the current UI page


**I** : (*steps of “rename a folder”*): ① open the menu icon of any folder (  ) ,  
② select “Rename” (  ), ③ input any text (e.g.,  ), ④ click “SAVE” (  )

**Q**: the folder name (  ) should be updated to the inputted text (e.g.,  )

# 广告： Call For Contributions!

README

## Kea



Kea is a general and practical testing tool based on the idea of property-based testing for finding functional bugs in Android apps.

[Documentation](#)

The apk file used in our evaluation can be downloaded from [here](#)

### Setup

Requirements:

- Python 3.8+
- Android SDK

You can input following commands to grep and install the required packages.

```
git clone https://github.com/ecnusse/Kea.git
cd home
pip install -e .
```

Kea 0.1.0

Search docs

- Installation
- How to setup Android SDK Environment
- Setup
- Quick example
- How to write properties
- User API
- Bug List

Develop and launch modern apps with MongoDB Atlas, a resilient data platform.

Ad by EthicalAds

Kea: a property-based testing tool for mobile apps [View page source](#)

## Kea: a property-based testing tool for mobile apps

Kea is a testing tool based on the idea of property-based testing for finding functional bugs in Android apps. Given a set of properties, Kea automatically generates test cases to check the properties. When a property is violated, Kea generates a bug report that shows the bug's behavior. Users can use Kea in the following steps:

1. Specify properties for the app under test. A property is the expected behavior of the app under certain conditions. Users can specify properties based on the app's specification, documentation, test cases, or bug reports. Currently, Kea supports specifying properties in Python with multiple APIs.
2. Run Kea with the properties and the app under test. Kea will automatically generate test cases to explore the app. When the precondition of a property is satisfied, Kea will execute the interaction scenario of the property and check the postcondition. If the postcondition is violated, Kea will generate a bug report.

### Contents

- [Installation](#)
- [How to setup Android SDK Environment](#)
- [Setup](#)
- [Quick example](#)
  - [Bug report](#)
- [How to write properties](#)
  - [Example 1](#)

[Read the Docs](#) [latest](#)

<https://github.com/ecnusse/Kea>

# SEGMENT

---

Invariants

# Invariants

---

- Loop invariants

A property of a loop that holds before (and after) each iteration  
Captures the essence of the loop's correctness, and by extension  
of algorithms that employ loops

- Class invariants

A property that holds for all objects of a class  
Established upon the construction of the object and constantly  
maintained between calls to public methods

# Invariants in Code

---

```
procedure divide( n : int, d : int )
    returns( q : int, r : int)
requires n >= 0 && d > 0;
ensures q * d <= n && 0 <= r && r <= n;
{
    q := 0;
    r := 0;
    while (r >= d)
        invariant n == q * d + r;
        {
            q := q + 1;
            R := r - d;
        }
}
```

# Example: Loop Invariant

```
m = 0; k = 0;
while (m != N)
  @invariant: a[0..k-1] is all RED &&
              a[k..m-1] is all BLUE
{
  if (a[m] is RED) {
    swap(a, k, m);
    k = k + 1;
  } else {
    // a[m] is BLUE
  }
  m = m + 1;
}
```



# Example: Class Invariant

class invariant: all  
methods must preserve  
 $\text{denom} \neq 0$

to preserve the class  
invariant, must check  
 $d \neq 0$

```
class Rational {  
    //@ invariant denom != 0;  
    int num, denom;  
  
    //@ requires d != 0;  
    Rational (int n, int d){ num = n, denom = d; }  
  
    double getDouble() { return ((double) num) / denom; }  
  
    public static void main(String[] a) {  
        int n = readInt(), d = readInt();  
        if (d == 0) return;  
        Rational r = new Rational(n, d);  
        print (r.getDouble());  
    }  
}
```

divide-by-zero bug  
is averted

# READING

---

Hardening C/C++ Code with Clang Sanitizers

# LESSON

---

## Measuring Test Suite Quality

# How Good Is Your Test Suite?

---

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain

# How Good Is Your Test Suite?

---

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

# Code Coverage

---

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., inaccessible code
  - Often required for safety-critical applications

# Types of Code Coverage

---

- Function coverage: which functions were called?
- Statement coverage: which statements were executed?
- Branch coverage: which branches were taken?
- Many others: line coverage, condition coverage, basic block coverage, *data-flow coverage*, prime path coverage, ...

# QUIZ: Code Coverage Metrics

---

Test Suite:

foo(1, 0)

Statement Coverage:  %

Branch Coverage:  %

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x =       y =

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

# QUIZ: Code Coverage Metrics

---

Test Suite:

foo(1, 0)

Statement Coverage:

Branch Coverage:

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x =       y =

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

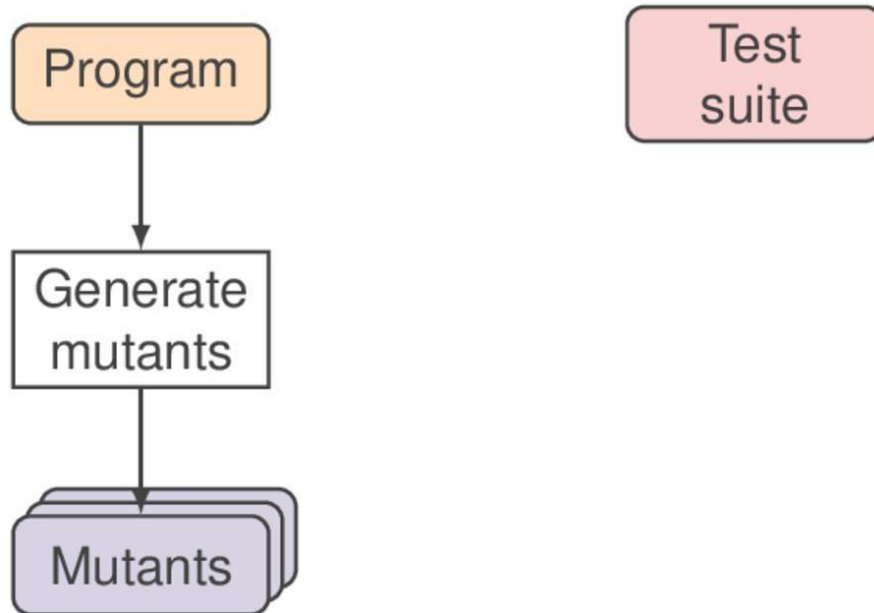
# Mutation Analysis (Mutation Testing)

---

- Founded on “competent programmer assumption”:  
*The program is close to right to begin with*
- Key idea: Test variations (mutants) of the program
  - Replace  $x > 0$  by  $x < 0$
  - Replace  $w$  by  $w + 1$ ,  $w - 1$
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

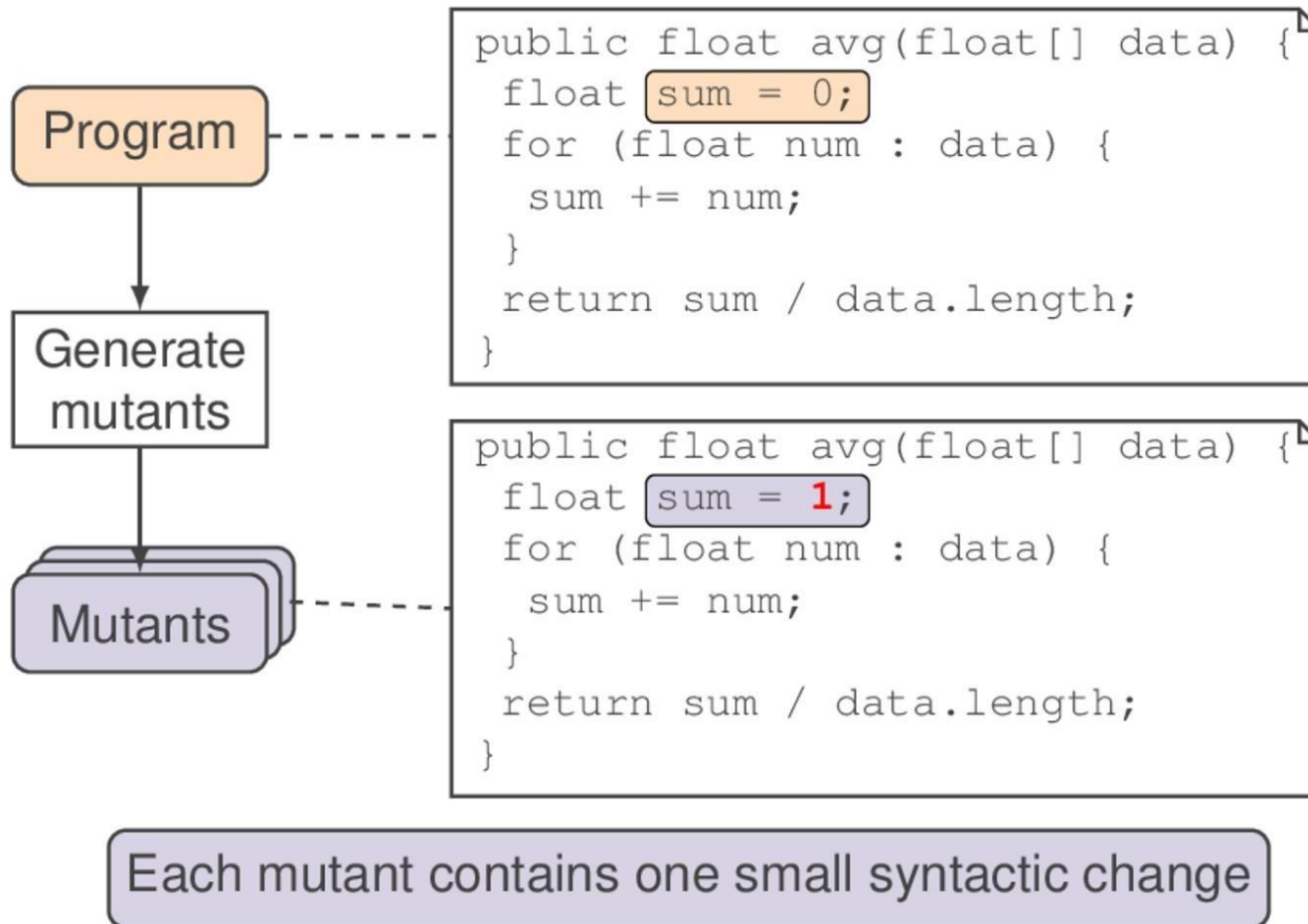
# Mutation Analysis

---



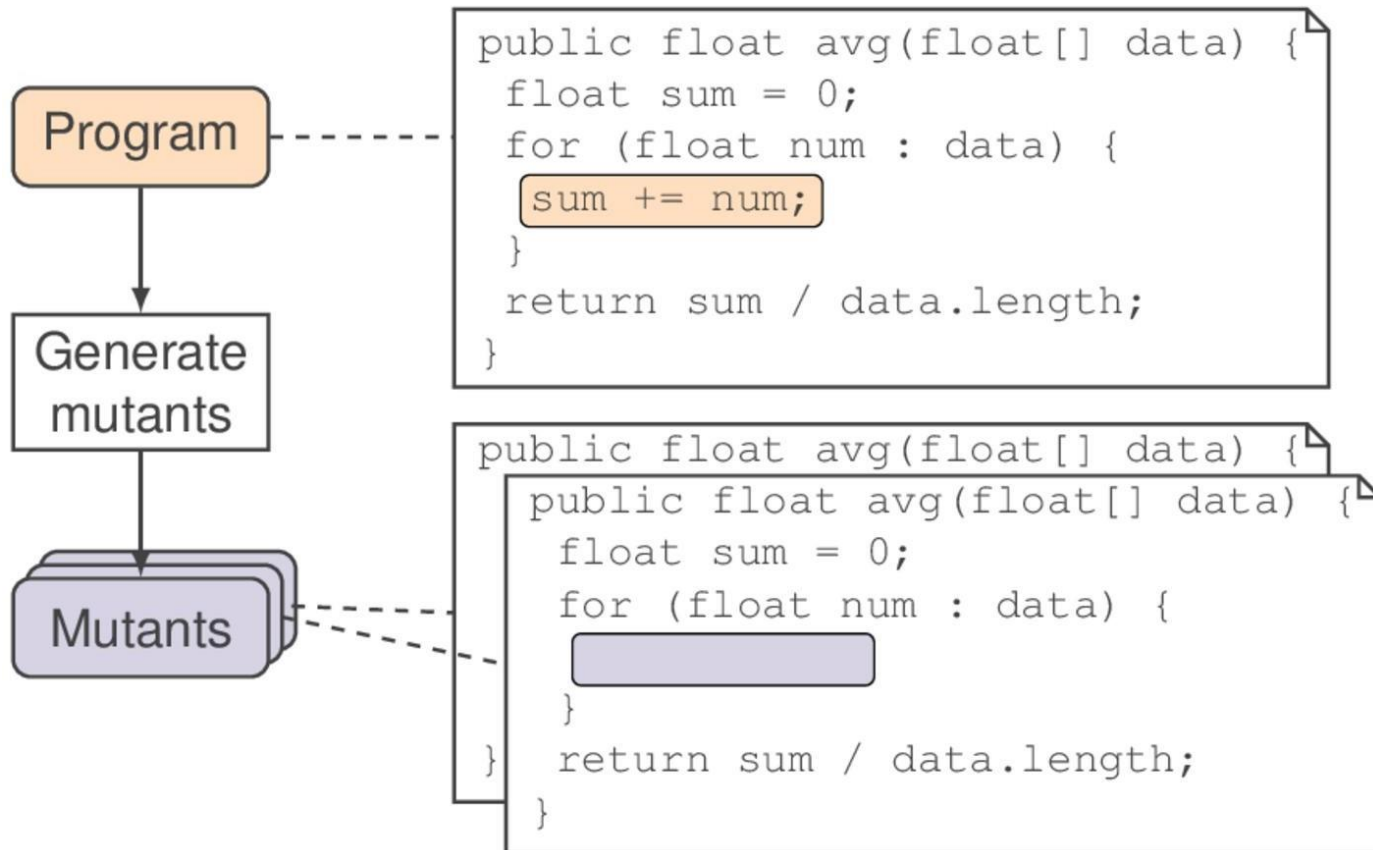
# Mutation Analysis

---



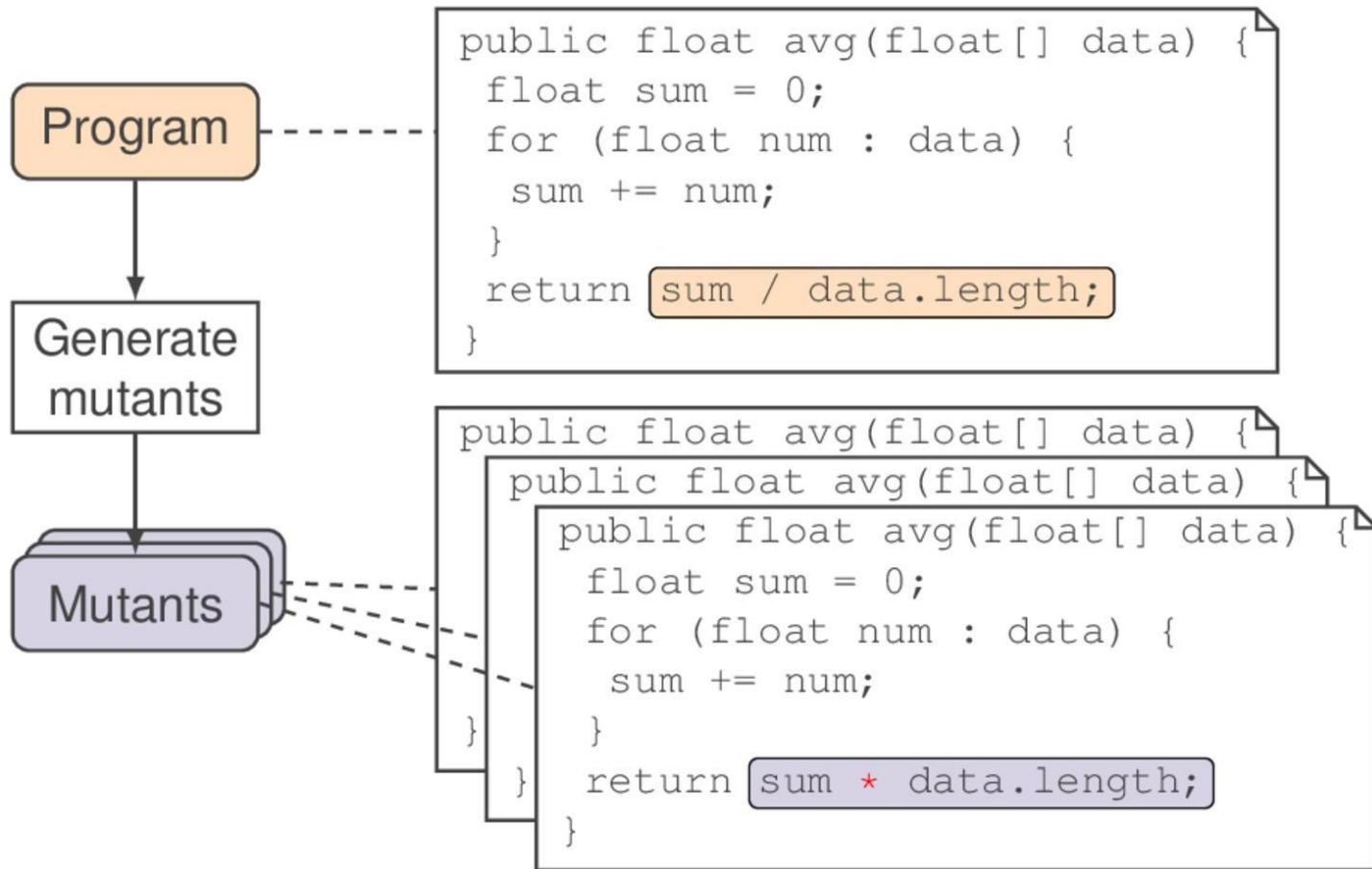
# Mutation Analysis

---



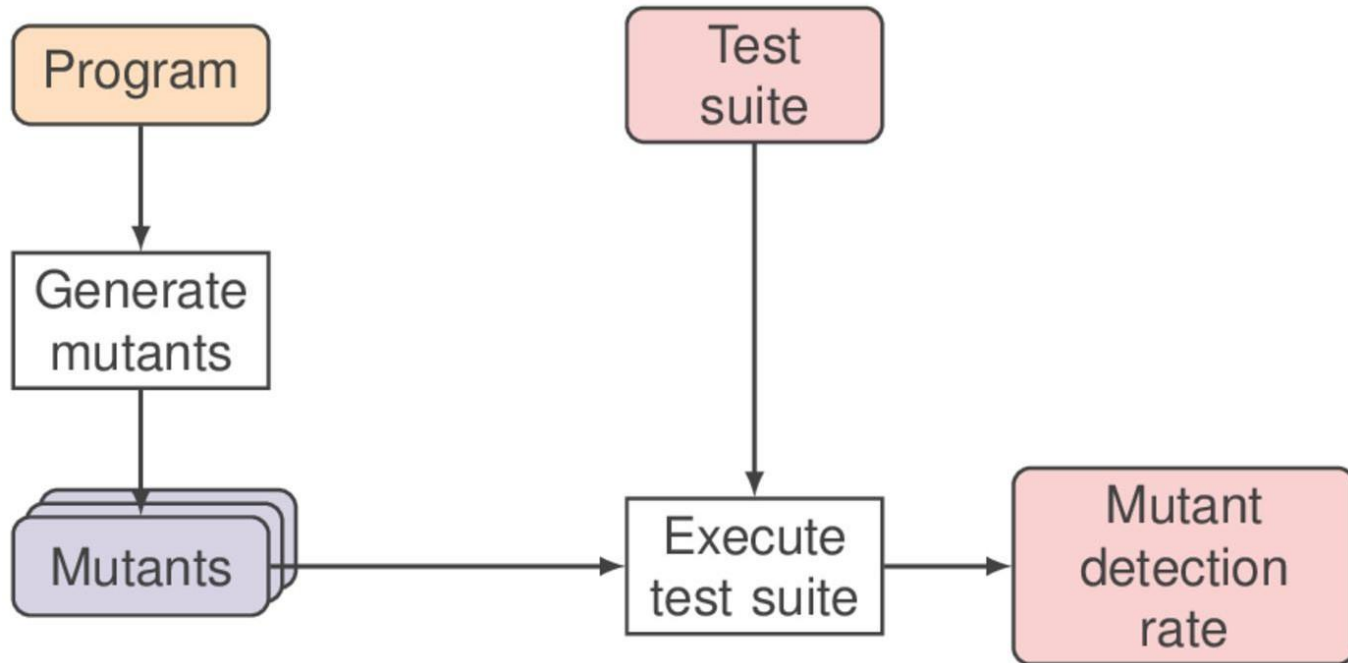
# Mutation Analysis

---



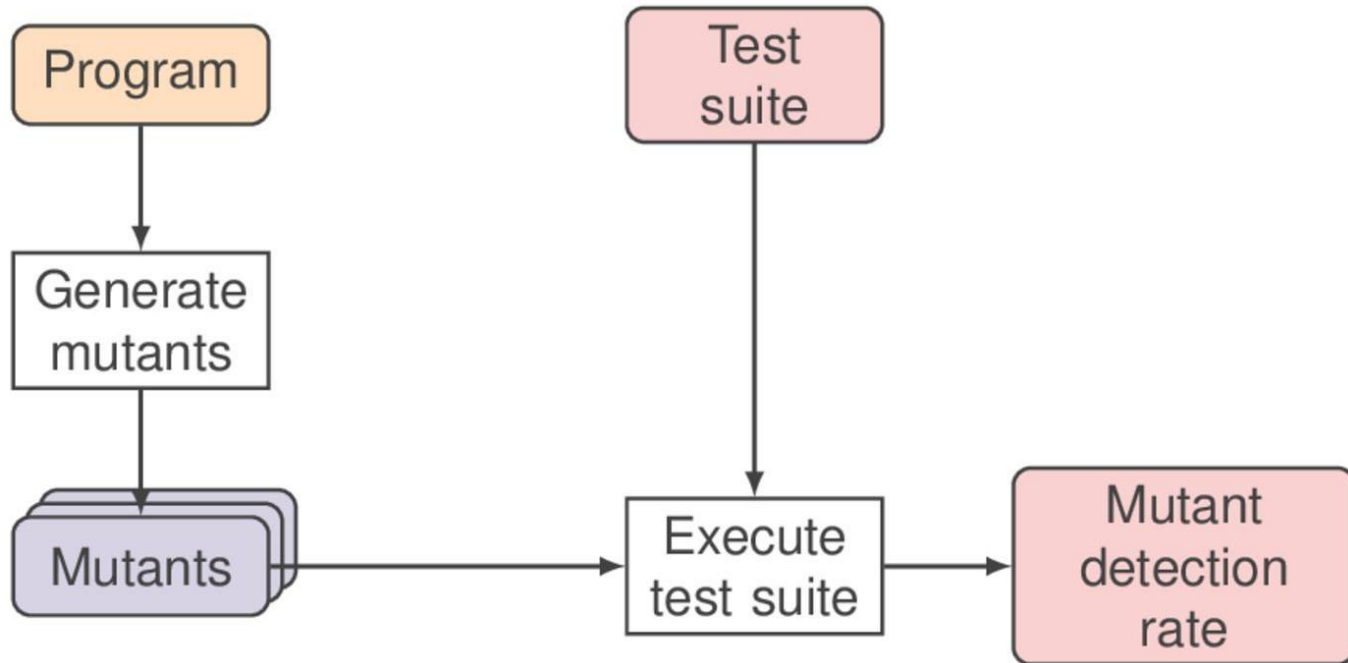
# Mutation Analysis

---



# Mutation Analysis

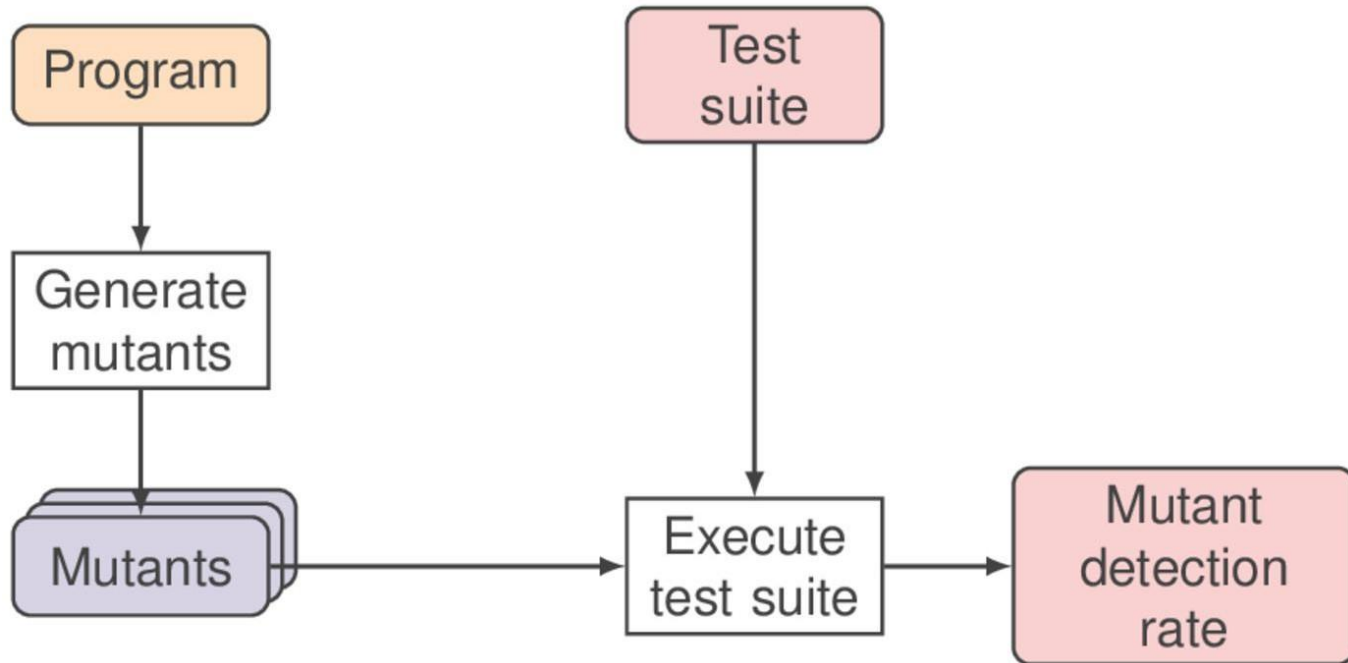
---



- Given Program  $P$  and its Mutant  $P'$ , if  $\exists$  Test  $t \in T$ , s.t.  $P(t) \neq P'(t)$ ,  $P'$  is killed or detected.
- Mutant detection rate =  $\frac{\text{\#killed mutants}}{\text{\#Mutants}}$

# Mutation Analysis

---



**Assumption:** Mutant detection rate is a useful proxy for software testing.  
([https://homes.cs.washington.edu/~rjust/publ/mutants\\_real\\_faults\\_fse\\_2014.pdf](https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf))

# QUIZ: Mutation Analysis - Part 1

---

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input type="checkbox"/>	<input type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Is the test suite adequate with respect to both mutants?

☐ Yes      No ☐

# QUIZ: Mutation Analysis - Part 1

---

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Is the test suite adequate with respect to both mutants?

☐ Yes

No ☒

# QUIZ: Mutation Analysis - Part 2

---

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Give a test case which Mutant 2 fails  
but the original code passes.

assert:  
foo(, ) ==

# QUIZ: Mutation Analysis - Part 2

---

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Give a test case which Mutant 2 fails  
but the original code passes.

assert:  
foo(, ) ==

# A Problem

---

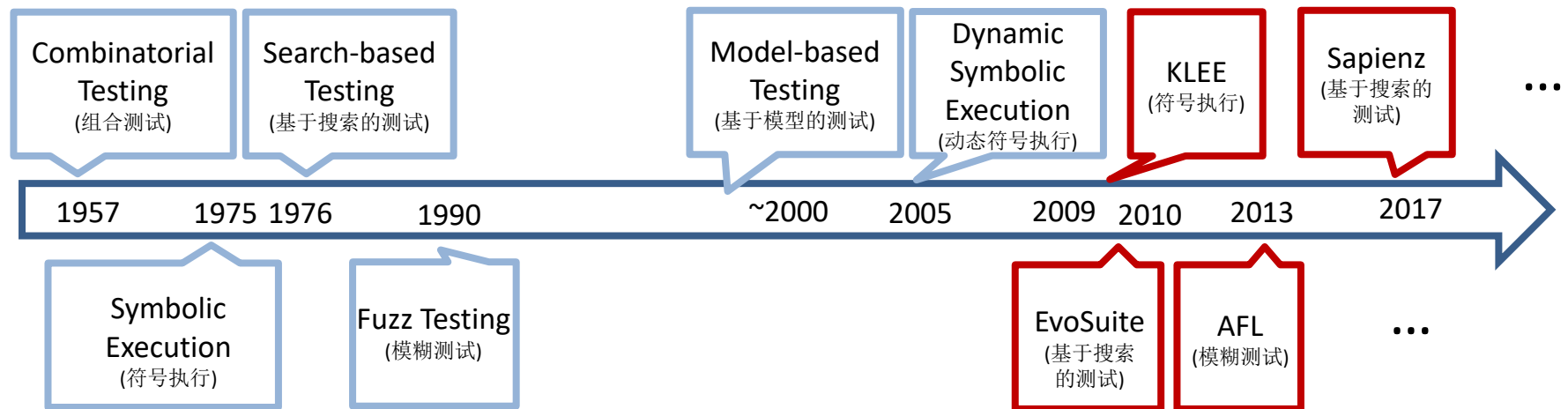
- What if a **mutant** is equivalent to the **original**?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
  - Try to prove **program equivalence** automatically
  - Often requires manual intervention

# Reality

---

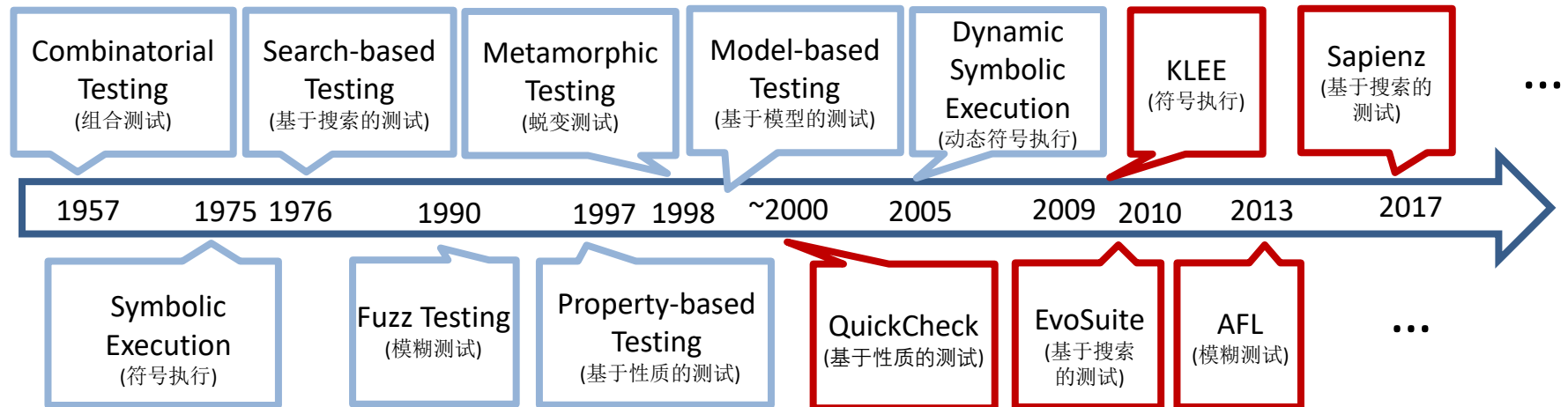
- Many proposals for improving software quality
- But the world tests
  - > 50% of the cost of software development
- Conclusion: Testing is important

# Classification of Testing Techniques



\* First proposed, Became popular.

# Classification of Testing Techniques



\* First proposed, Became popular.

# What Have We Learned?

---

- Landscape of Testing
  - Automated vs. Manual
  - Black-Box vs. White-Box
- Specifications: Pre- and Post- Conditions
- Measuring Test Suite Quality
  - Coverage Metrics
  - Mutation Analysis
- Classification of Testing Techniques

# Paper Readings

---

- *The Oracle Problem in Software Testing: A Survey.*  
IEEE Trans. Software Eng. 41(5): 507-525 (2015)
- *Programs, tests, and oracles: the foundations of testing revisited. ICSE 2011: 391-400*
- *Are mutants a valid substitute for real faults in software testing? SIGSOFT FSE 2014: 654-665*
- *Coverage is not strongly correlated with test suite effectiveness. ICSE 2014: 435-445*
- *An orchestrated survey of methodologies for automated software test case generation.*  
J. Syst. Softw. 86(8): 1978-2001 (2013)