Test Automation

CS 3250
Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 3]

© Praphamontripong

Manual Testing

Av

Name:

Email:

Phone:

What would you like to drink?

Hot or cold?

Hot

Cold

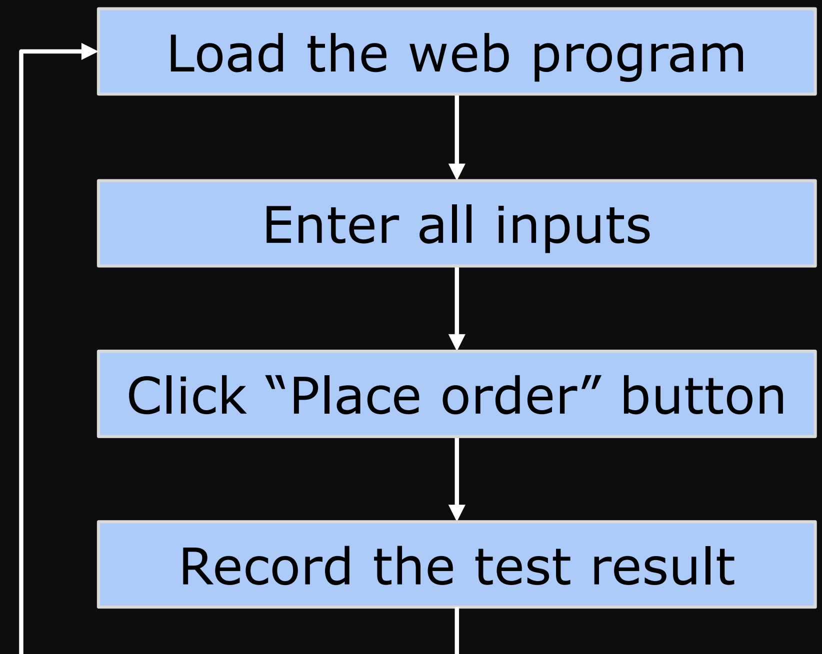
Send me a text message when my order is ready

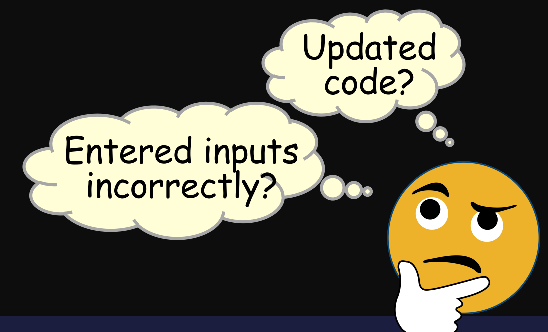
Place order

Name [valid, invalid]
Email [valid, invalid]
Phone [valid, invalid]
Drink [select, not select]
Hot or cold [hot, cold, not select]
Send text [send, not send]

How many tests?
How many clicks, keystrokes?

Imagine you are testing a web program that takes a drink order





[Ref: emoji by Ekarin Apirakthanakorn]

© Praphamontripong

Manual Testing

Benefits

- Simple and straightforward
- No up-front cost
- Easy to set up
- No additional software to learn, purchase, write
- Flexible
- More likely to test things that users care about
- Possible to catch issues that automated testing does not

Drawbacks

- (Extremely) boring
- Often not repeatable
- Some tasks are difficult or impossible to test manually
- Human errors
- Extremely time- and resource-intensive
- Limit to black-box and gray-box testing

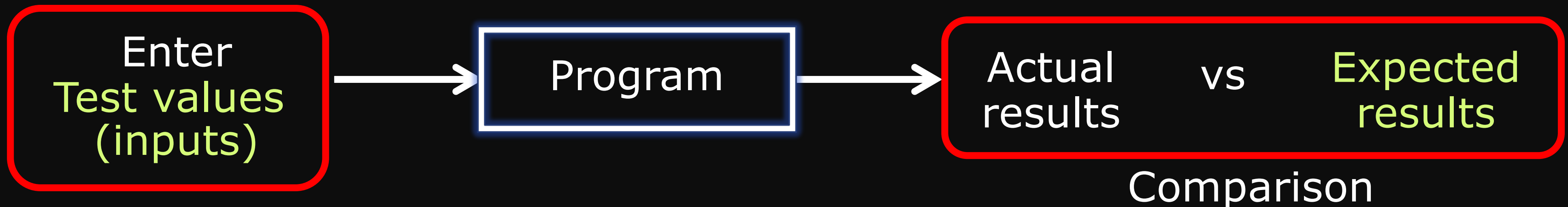
Tasks in Testing Process

Design tests

Revenue task

Enter inputs, run program, record results

Excise task



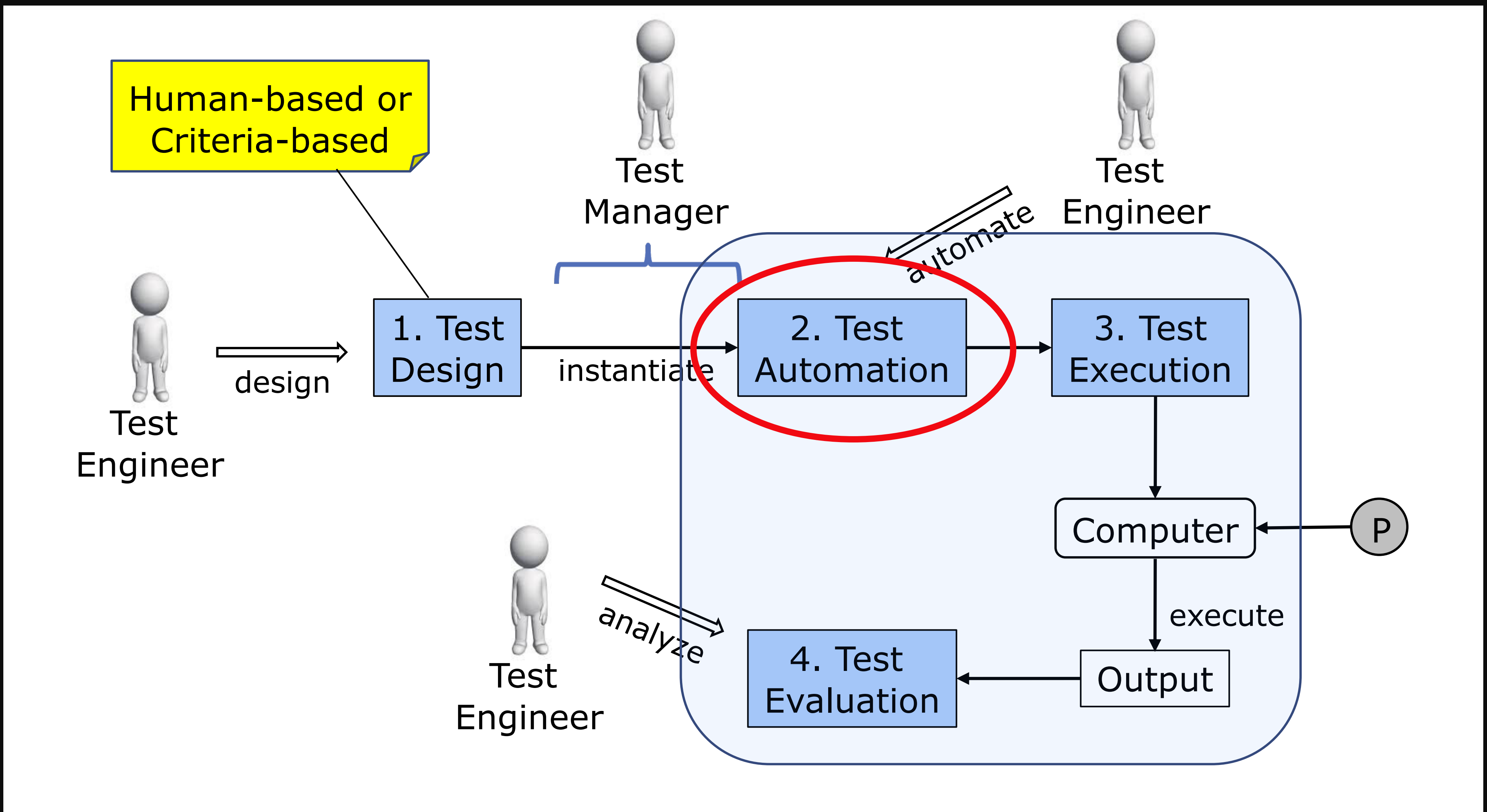
Execute tests against the program under test (PUT)
or sometimes referred to as a software under test (SUT)

Analyze results

Revenue task

Revisit

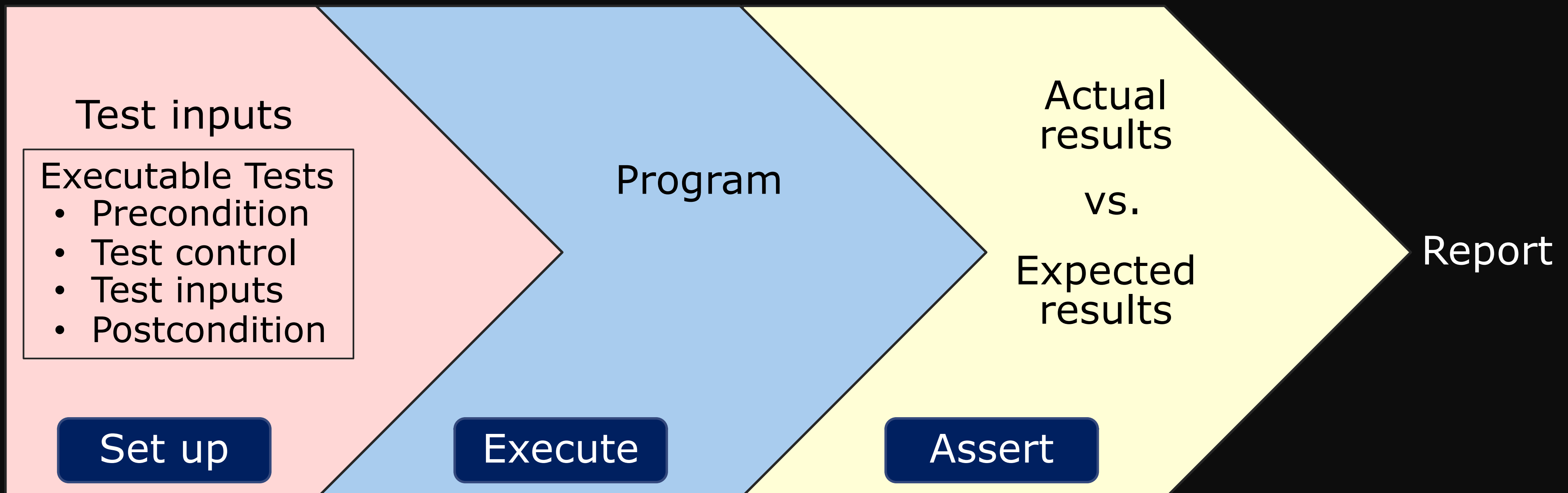
Software Testing Activities



[AO, p.22]

Test Automation

- Use of software to control the execution of tests
- Comparison of actual outcomes to predicted outputs
- Setting up of test preconditions, and other test control and test reporting functions



Automated Testing

Benefits

- Reduce human errors
- Reduce excise tasks
- Free up time to focus on revenue tasks
- Reduce cost, less resource-intensive
- Reduce variance in test quality from different individuals
- Easily repeatable
- Significantly reduce the cost of regression testing
- Cover aspects that manual testing is impossible
- Scalable

Drawbacks

- Setup time up-front
- May miss some user-facing defects
- Require training (programming languages, testing frameworks, tools)

Automate as much as possible

© Praphamontripong

Flaky (Nondeterministic) Tests

"Tests that exhibits both a passing and a failing results with the same code"
– Google

Run #1
Test A on
program P

Run #2
Test A on
program P

Run #3
Test A on
program P

Google says 16% of their tests are flaky

Controllability

Potential causes

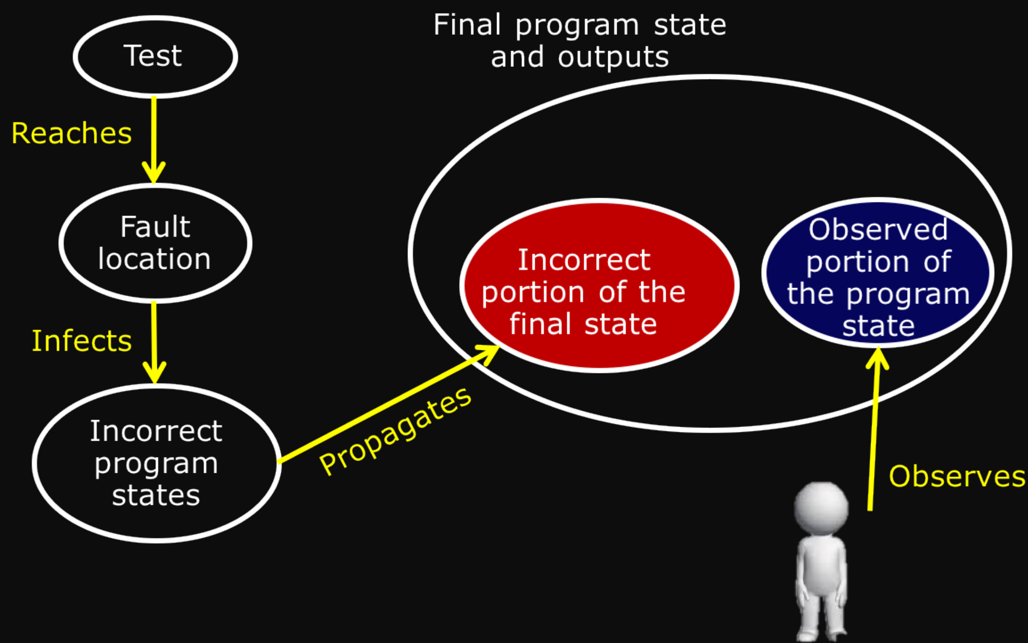
- Concurrency
- Asynchronous behavior
- Random inputs
- Resource leaks
- Test order dependency
- Collection class assumptions
- Relying on external systems
- Not checking emails

[Thanks to Professor Jeff Offutt, CS 3250 Guest speaker, Fall 2019]

Blind Tests

Tests that do not notice failures

"Almost half the tests written by professional software engineers had broken test oracles" – Jeff Offutt



Observability

Potential causes

- Unintended ignorance
- Observe the wrong place
- Overlooked information

[Thanks to Professor Jeff Offutt, CS 3250 Guest speaker, Fall 2019]

What Makes Test Automation Hard

Imagine you are tasked to test a company's software.

You need to **set up** three different web services, create five different files in different folders of the system, and put SQL and NoSQL databases in specific states.

After all the setup, you **execute** the feature under test.

To **assert** the behavior of the software under test, you need to check if the three web services were invoked, the five files were manipulated correctly, and the databases were now in proper state.

Doable? Can this be simpler?

What Makes Test Automation Hard

Testability

- *"The degree to which a system or component facilitates that establishment of test criteria and the performance of tests to determine whether those criteria have been met"*

How hard it is to detect faults in the software

Two practical problems:

- Controllability: how to provide the test values to the software
- Observability: how to observe the results of test execution

Aspects that Impact Testability

Controllability

- *"How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors"*

- Example of software that often have low controllability:
  - Embedded software often gets its inputs from hardware sensors
    - Difficult to control
    - Some inputs may be difficult, dangerous, or impossible to supply
  - Component-based software
  - Distributed software
  - Web applications
  - Mobile applications

(Some) **Controllability Problems**

1. State hidden in method
2. Difficult setup
3. Incomplete shutdown
4. State-leaks across tests
5. Framework frustration
6. Difficult mocking
7. Hidden effects
8. Hidden inputs

Aspects that Impact Testability

Observability

- *"How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components"*

- Example of software that often have low observability:
  - Embedded software often does not produce output for human consumption
  - Component-based software
  - Distributed software
  - Web applications
  - Mobile applications

Rules of Thumb

1. Design and implement your software for testability
2. Create good test code

If it is difficult to test your code, there may be problems with your design and implementation.

Writing testable code is harder than writing untestable code.

Solving design problems will solve testing problems.

Making code testable does not necessarily make the design better.

Design for testability is fundamental for systematic testing

(Some) **Design Ideas for Testability**

1. Separate the infrastructure from the domain code
2. If a class depends on another class, make it in a way that the dependency can easily be replaced by a mock, fake, or stub
3. Make your classes and methods observable
4. Passing values via method parameters to simplify the callers, instead of passing dependency via a constructor
5. Focus on a single responsibility
6. Try to minimize coupling
7. Keep Boolean expressions small and simple
8. If you need to test a private method, this private method may not belong in its current place → refactor your code

Good Test Code

1. Test one thing (purpose, traceable) – one assertion per test
2. Clear, easy to understand, precise, concise
3. Keep it small (more small tests are better than few large tests)
4. Move repeated code to fixtures or shared methods
5. Independent (test frameworks don’t guarantee order of execution)
6. Repeatable
7. Avoid complicated control flow in tests
8. Remember to refactor test code
9. Test oracles should check the right place (no need to check the entire output state)
10. Use the right type of assertion

Components of a Test Case

Test case	A multipart artifact with a definite structure Test case values, expected results, prefix and postfix values necessary for a complete execution and evaluation of the software under test	
Test case values	Inputs needed to complete an execution of the software under test Inputs can be input values or series of actions	Determine the testing quality
Expected results	The result that will be produced by the test if the software behaves as expected	Test oracle (pass or fail)

Components of a Test Case

Prefix values: Inputs needed to put the software into the appropriate state to receive the test case values

Postfix values: Inputs needed to be sent to the software after the test case values are sent

- **Verification values**: values needed to see the results of the test case values
- **Exit values**: values or commands needed to terminate the program or otherwise return it to a stable state

Affecting controllability and observability

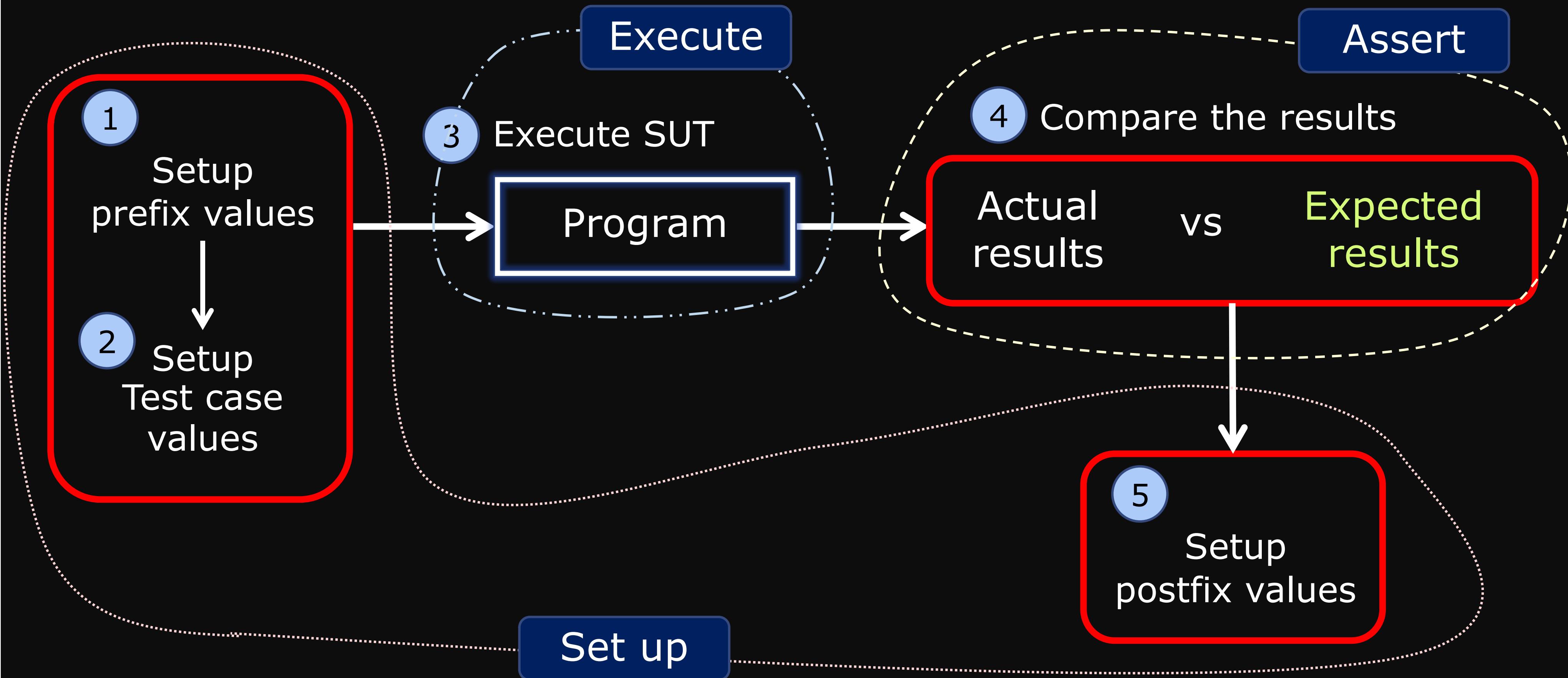
Inputs can be input values or series of actions

Additional components: test case ID, purpose of the test

Putting Tests Together

- Test case
  - The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test
- Test set
  - A set of test cases
- Executable test script
  - A test case that is prepared in a form to be executed automatically on the test software and produce a report

Test Case Execution



Test Automation Framework

- A set of assumptions, concepts, and tools that support test automation
- Provides a standard design for test scripts and support for the test driver
- Test driver
  - Runs a test set by executing the software repeatedly on each test
  - Supplies the "main" method to run the software if it is not standalone (i.e., a method, class, or other component)
  - Compares the results of execution with the expected results
  - Reports the results to the tester

© Praphamontripong

Test Automation Frameworks (2)

- Most test automation frameworks support
  - Assertions to evaluate expected results
  - The ability to share common test data among tests
  - Test sets to easily organize and run tests
  - The ability to run tests from either a command line or a GUI

- Most test automation frameworks are designed for unit and integration testing; some specifically support system testing; some are built to support testing over the web

- Example test automation frameworks
  - JUnit, HttpUnit, HtmlUnit, JWebUnit, Selenium, unittest, Jasmine, Jest, Cucumber, PHPUnit, Robotium

Wrap-up

- Test automation
- Testability, Observability, Controllability
- Design software for testability
- Write good test code
- Components of a test case

What's Next?

- Test automation framework – JUnit