Intro to Software Testing
Chapter 2

Model-driven Test Design

Dr. Brittany Johnson-Matthews

(Dr. B for short)

https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

But first…

Quiz time!

Login to Blackboard

Under ‘Assessments’ select ‘Week 1 Quiz – Why Test’

You have 15 minutes

Intro to Software Testing

Chapter 2

Model-driven Test Design

Dr. Brittany Johnson-Matthews

(Dr. B for short)

https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

Complexity of testing software

No other engineering field builds products as **complicated** as software

The term **correctness** has no meaning

- Is a building correct?
- Is a car correct?
- Is a subway system correct?

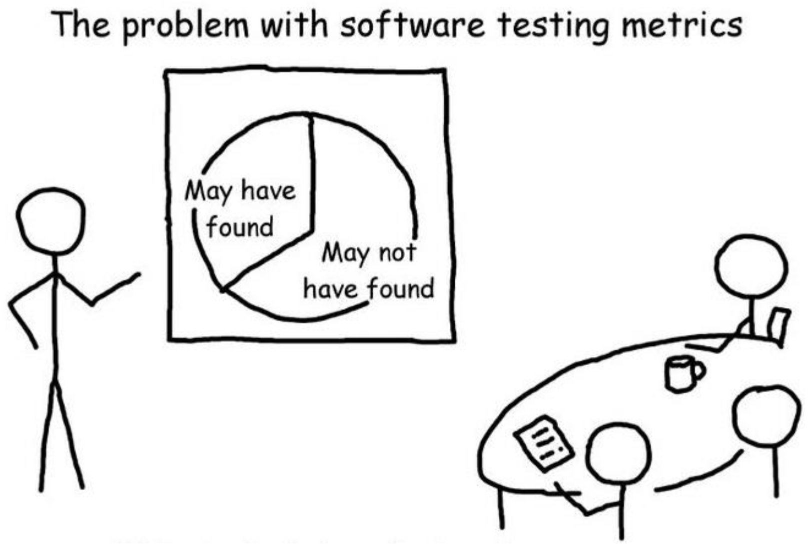
Unlike other engineers, we must use **abstraction to manage complexity**

- This is the purpose of the **model-driven test design** process
- The "model" is an abstract structure

Software Testing Foundations (2.1)

Testing can only show the presence of failures

<u>Not their absence</u>



Testing and Debugging

Testing: evaluating software by observing its execution

Test failure: execution of test that results in software failure

Debugging: The process of finding a fault given a failure

Not all inputs will "trigger" a fault into causing a failure!

Driving Testing Behavior

Fault & Failure Model (RIPR)

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. **Revealability**: The tester must observe part of the incorrect portion of the program state.

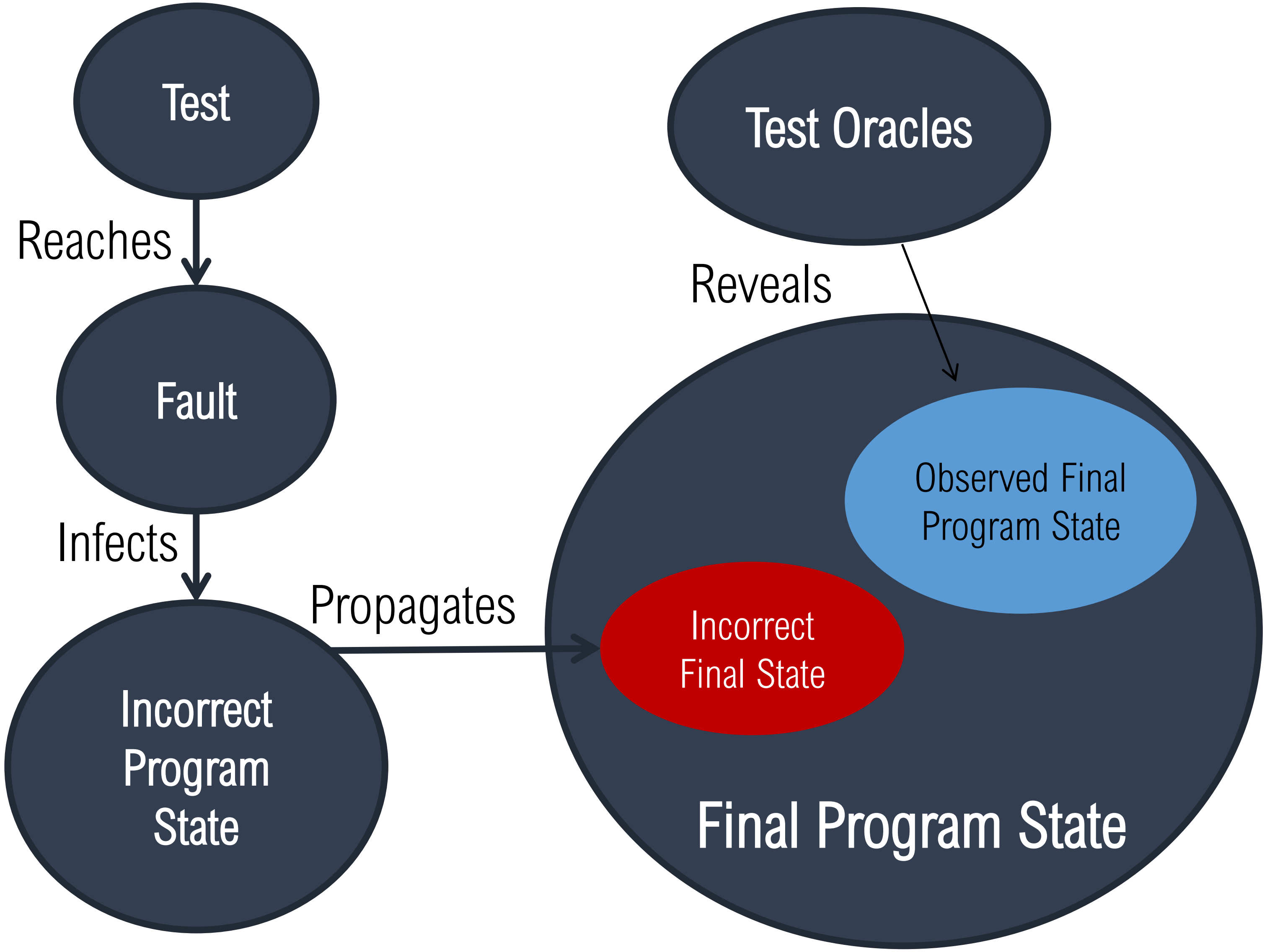
RIPR Model

Reachability

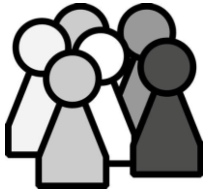
Infection

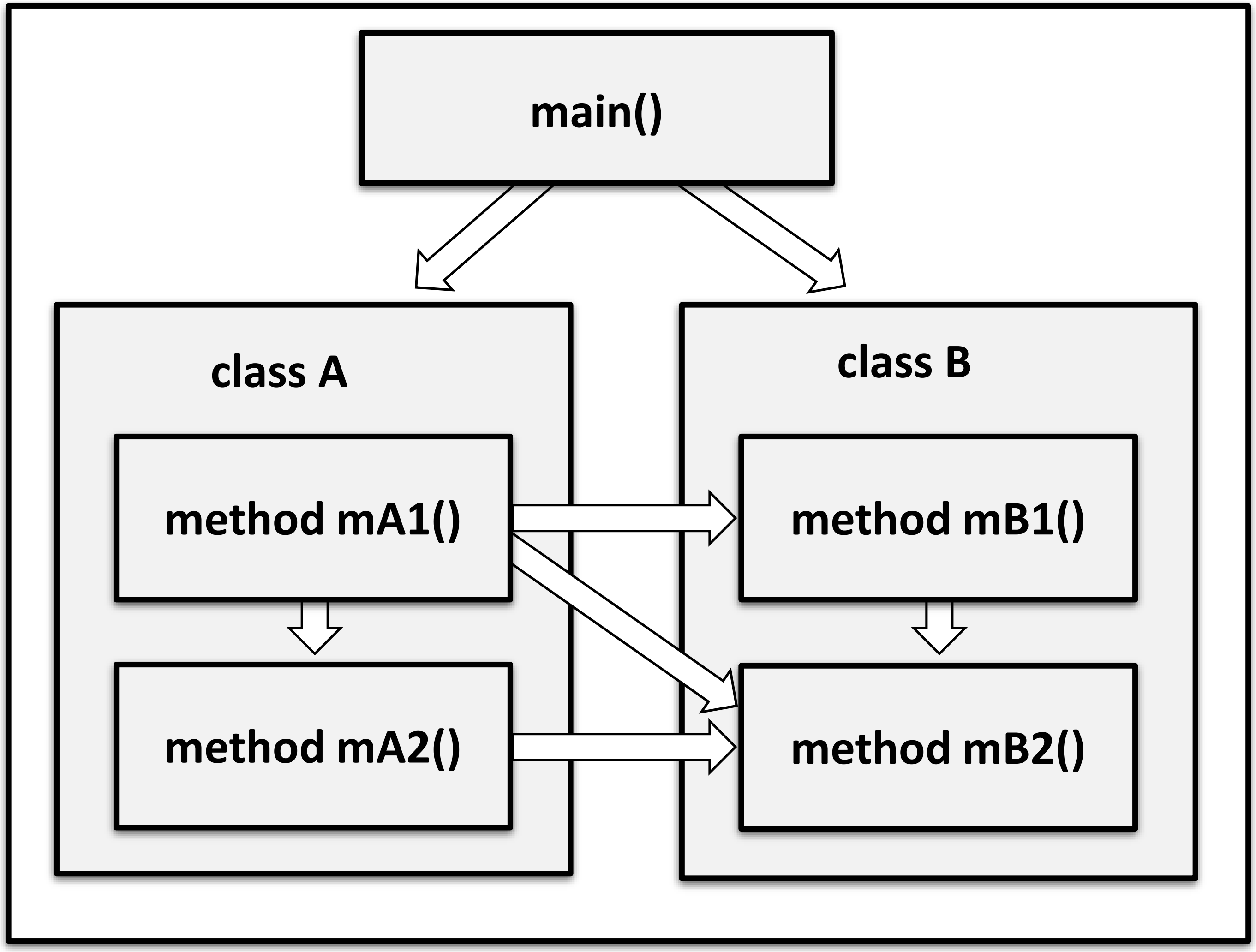
Propagation

Revealability

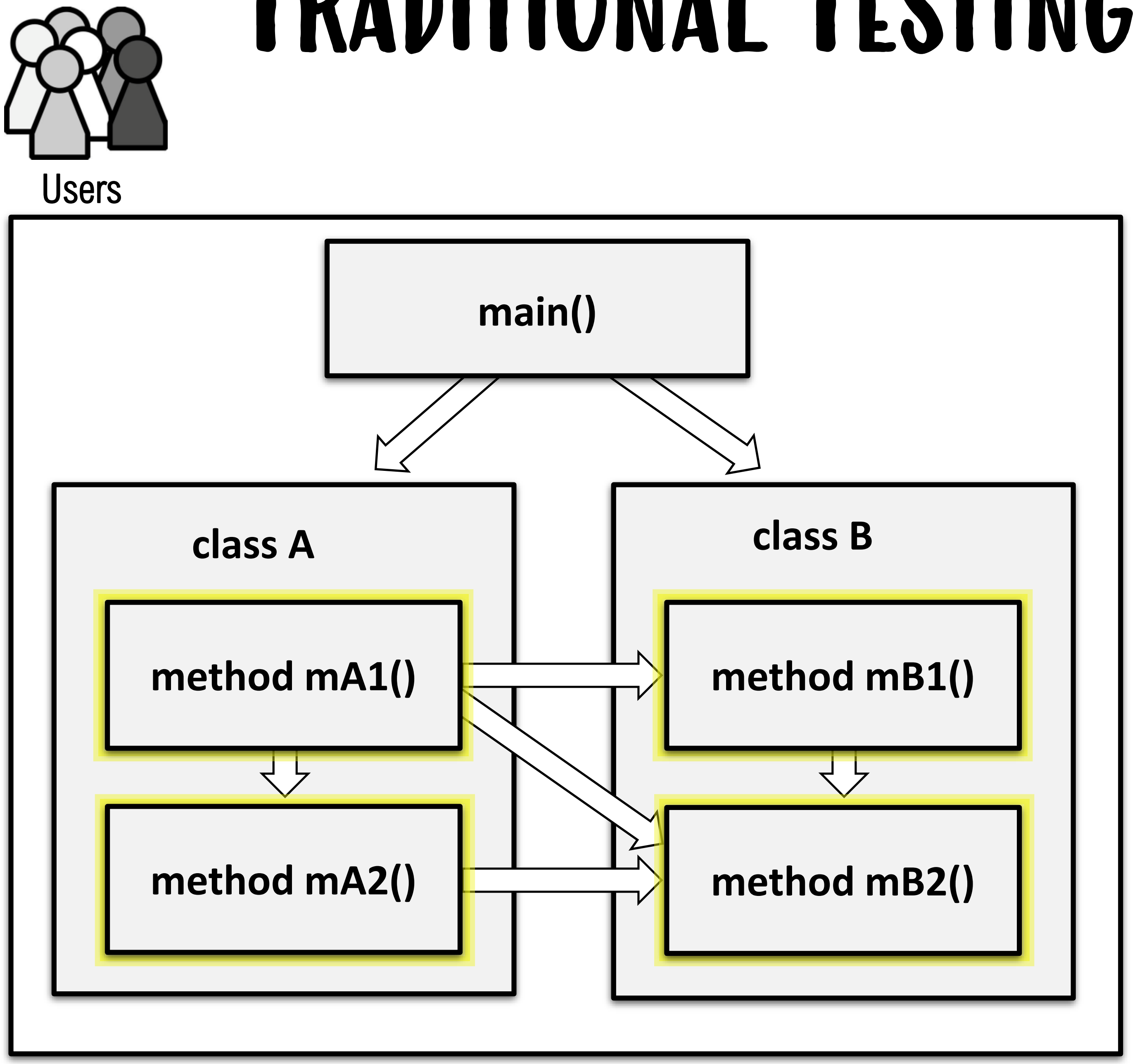


Traditional Testing Levels (2,3)





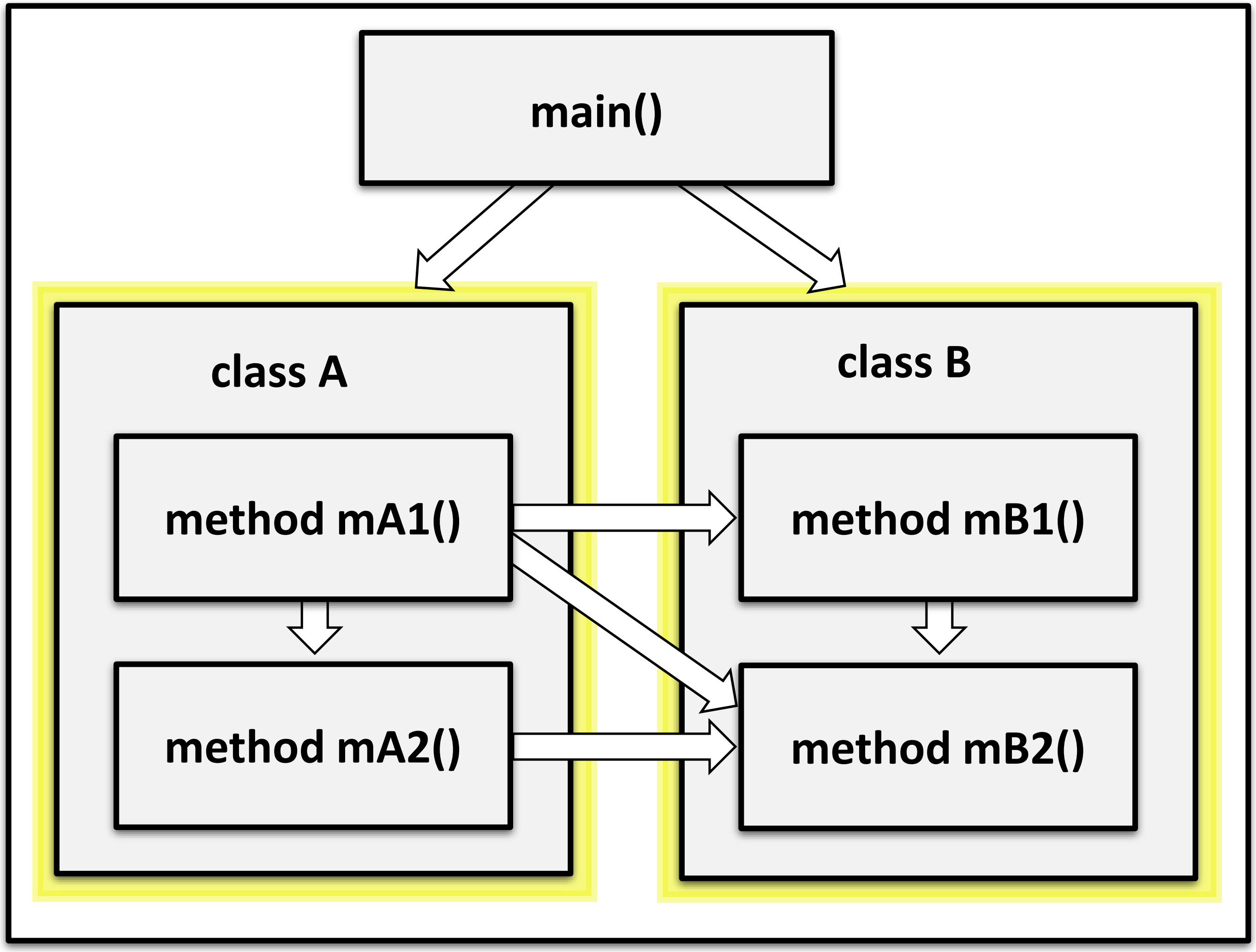
Traditional Testing Levels (2,3)



Unit testing: test each unit (method) individually

Traditional Testing Levels (2.3)

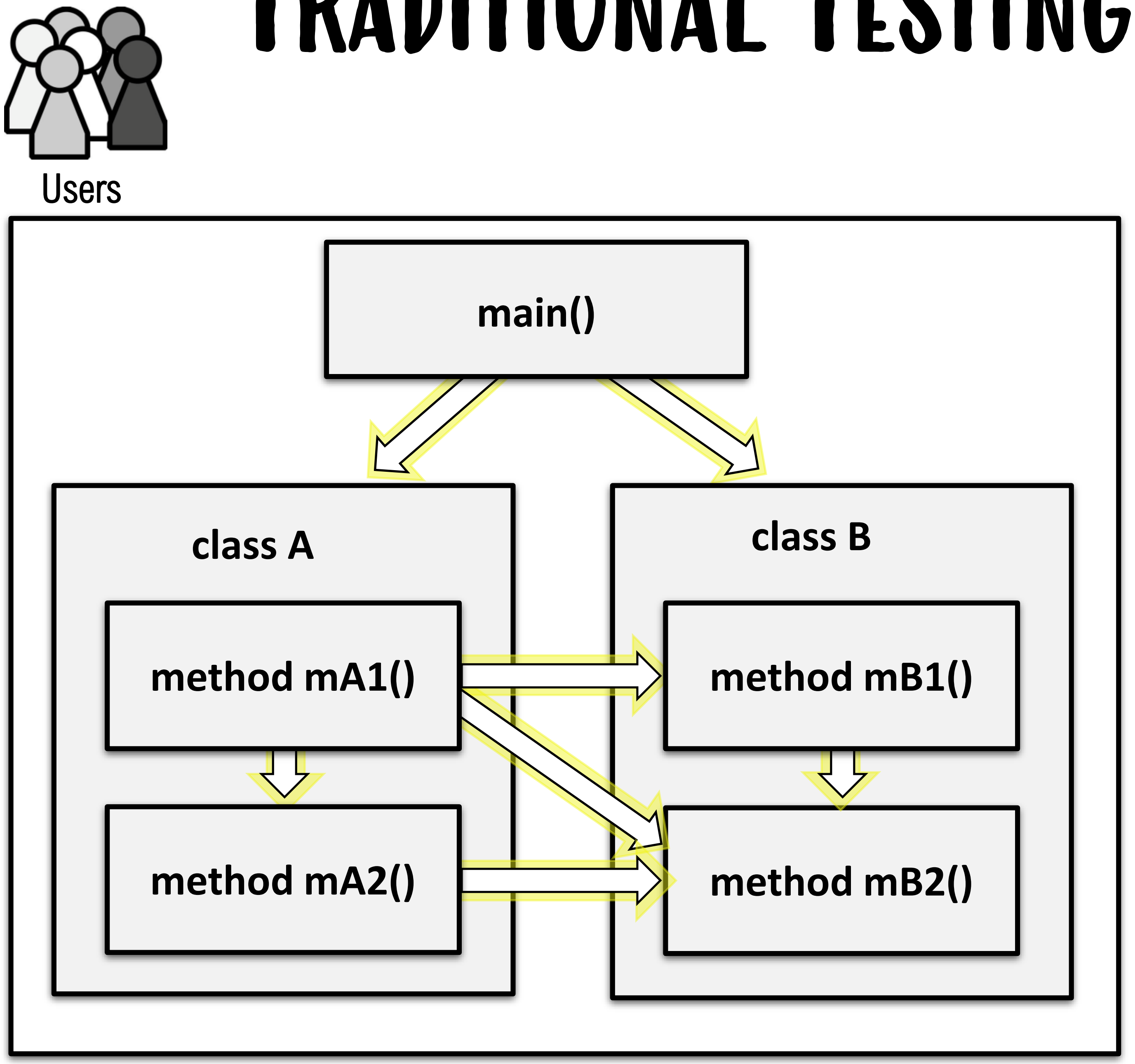
Users



Module testing: test each class, file, module, component

Unit testing: test each unit (method) individually

Traditional Testing Levels (2.3)

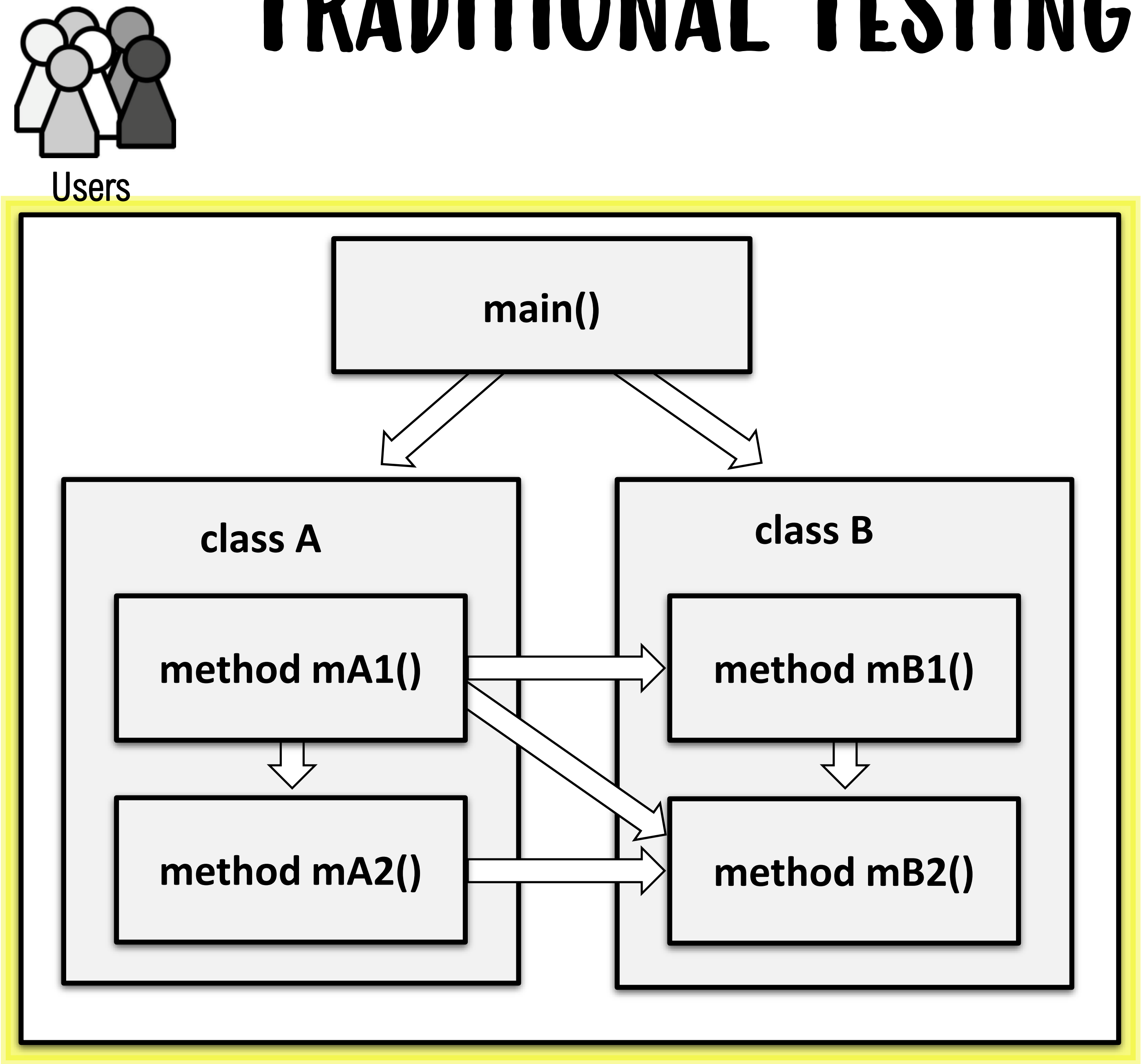


Integration testing: test how modules interact

Module testing: test each class, file, module, component

Unit testing: test each unit (method) individually

Traditional Testing Levels (2.3)



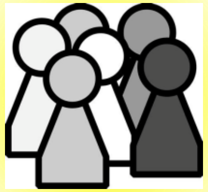
System testing: test the overall functionality of the system

Integration testing: test how modules interact

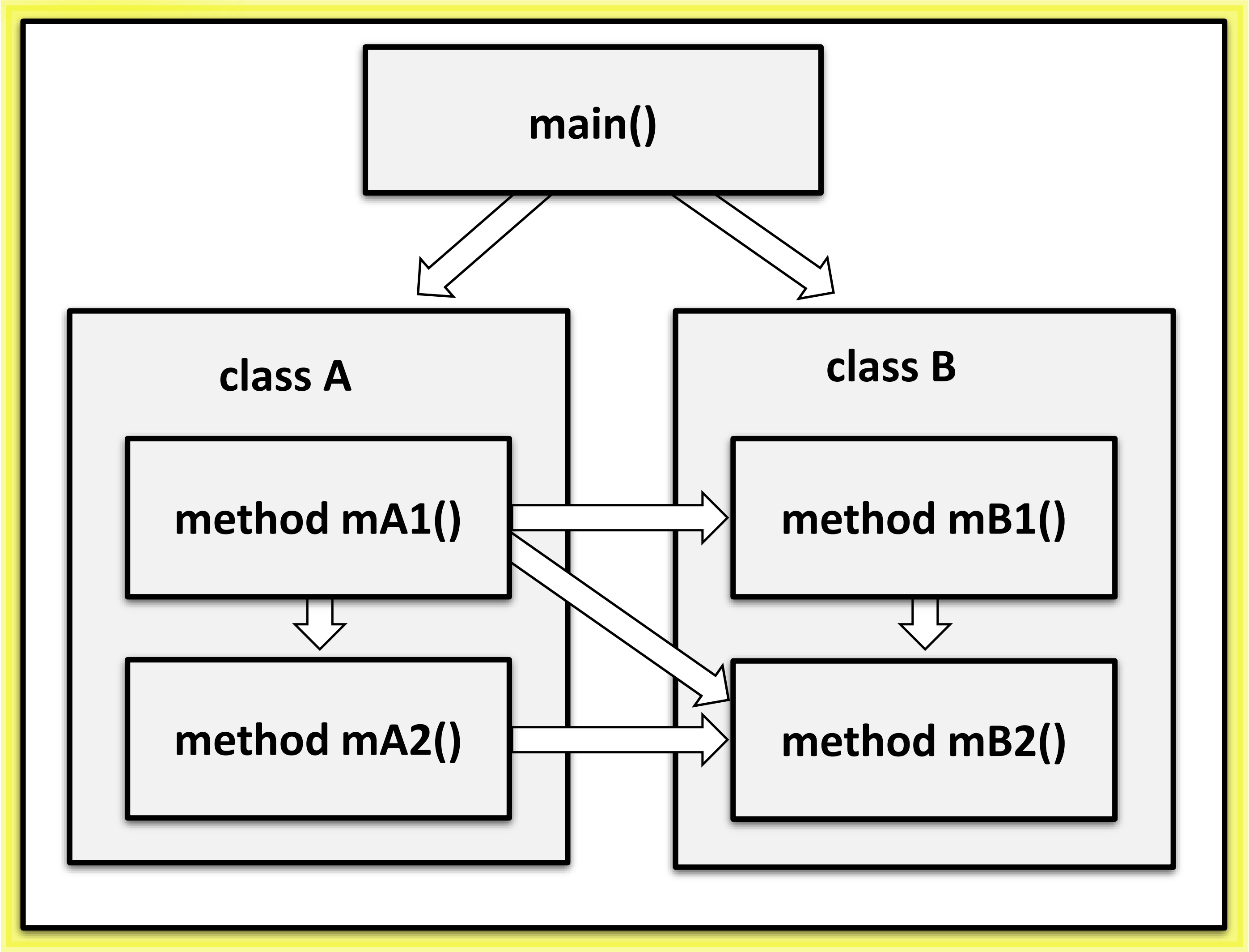
Module testing: test each class, file, module, component

Unit testing: test each unit (method) individually

Traditional Testing Levels (2.3)



Users



Acceptance testing: is software acceptable to the user?

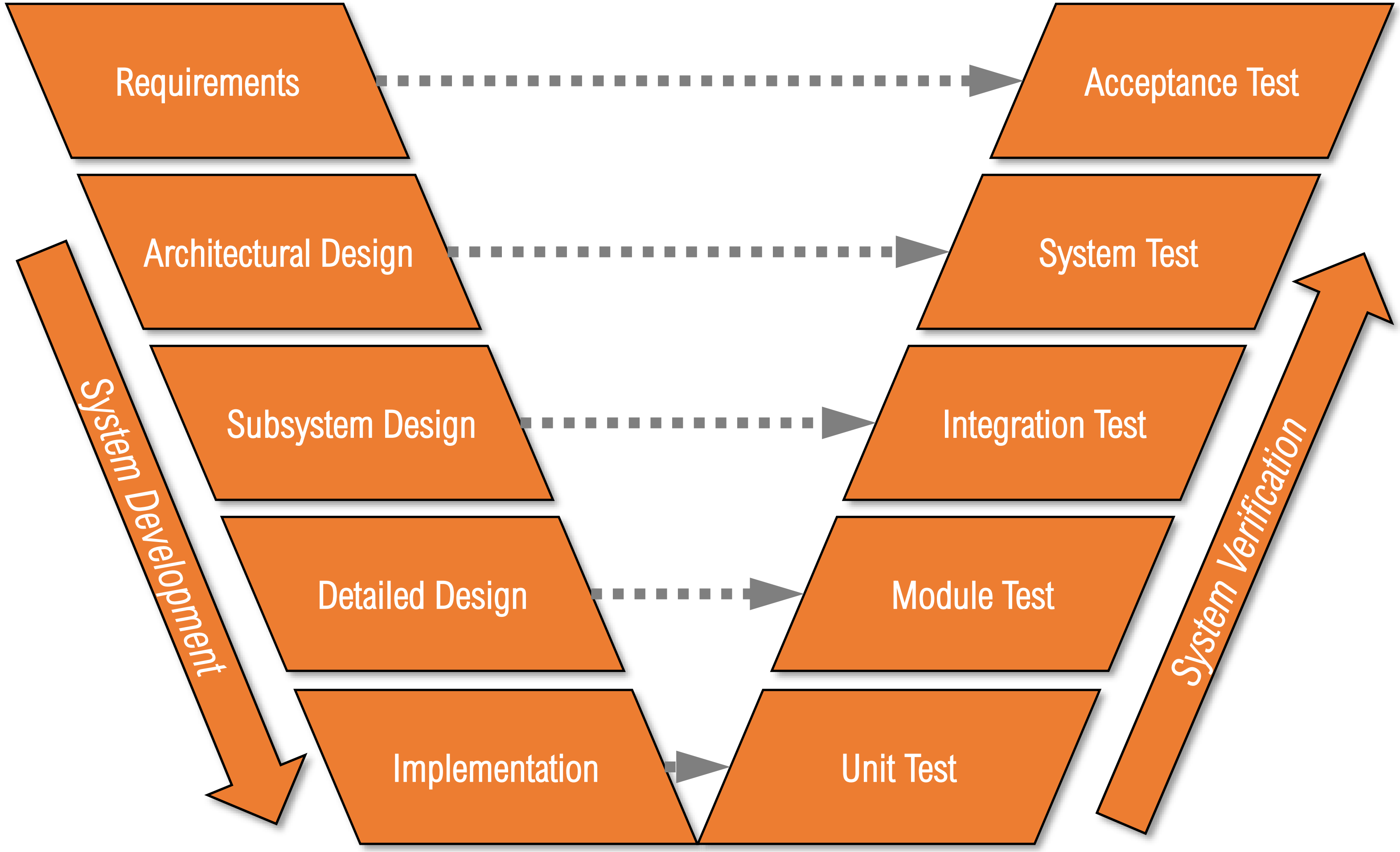
System testing: test the overall functionality of the system

Integration testing: test how modules interact

Module testing: test each class, file, module, component

Unit testing: test each unit (method) individually

V Model of System Development



Coverage Criteria (2.4)

Even small programs have **too many inputs** to fully test them all

- private static double computeAverage (int A, int B, int C)
- On a 32-bit machine, each variable has over **4 billion** possible values
- Over **80 octillion possible tests**!!
- Input space might as well be infinite

Testers **search** a huge input space for **fewest inputs** that will find the **most problems**

Coverage criteria give structured, practical ways to search the input space

- **search** the input space thoroughly
- not much **overlap** in the tests

Advantages of Coverage Criteria

Maximize the “**bang for the buck**”

Provide **traceability** from software artifacts to tests

- source, requirements, design models,…

Make **regression testing** easier

Gives testers a **“stopping rule”** … when testing is finished

Can be well supported with powerful **tools**

Test requirements and criteria

Test criterion: A collection of rules and a process that defines test requirements

- Cover every statement
- Cover every functional requirement

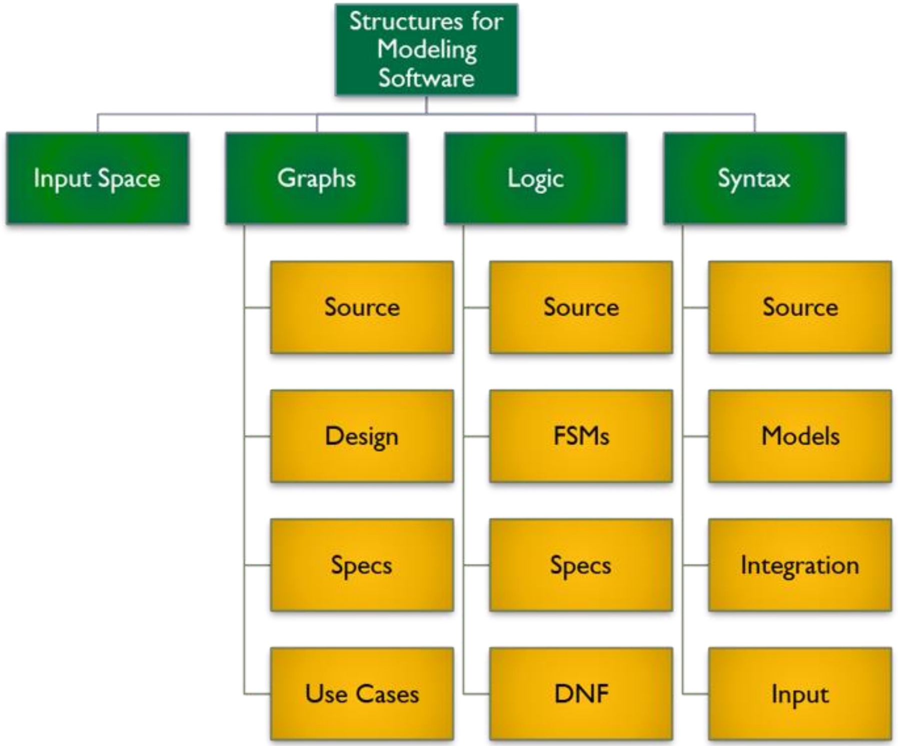
Test requirements: specific things that must be satisfied or covered during testing

- each statement might be a test requirement
- each functional requirement might be a test requirement

Test requirements and criteria

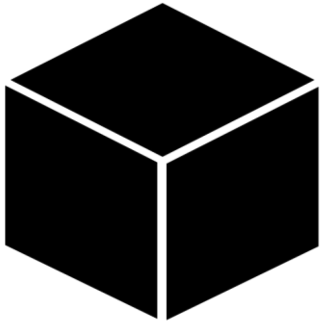
Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures:

1. Input domains
2. Graphs
3. Logic expressions
4. Syntax descriptions

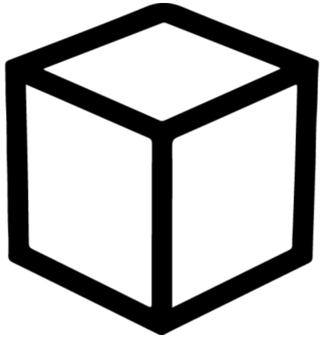


Old view: color boxes

Black box testing: derive tests from external descriptions of the software, including specifications, requirements, and design



White box testing: derive tests from the source code internals of the software, specifically including branches, individual conditions, and statements



Model-based testing: derive tests from a model of the software (such as a UML diagram)

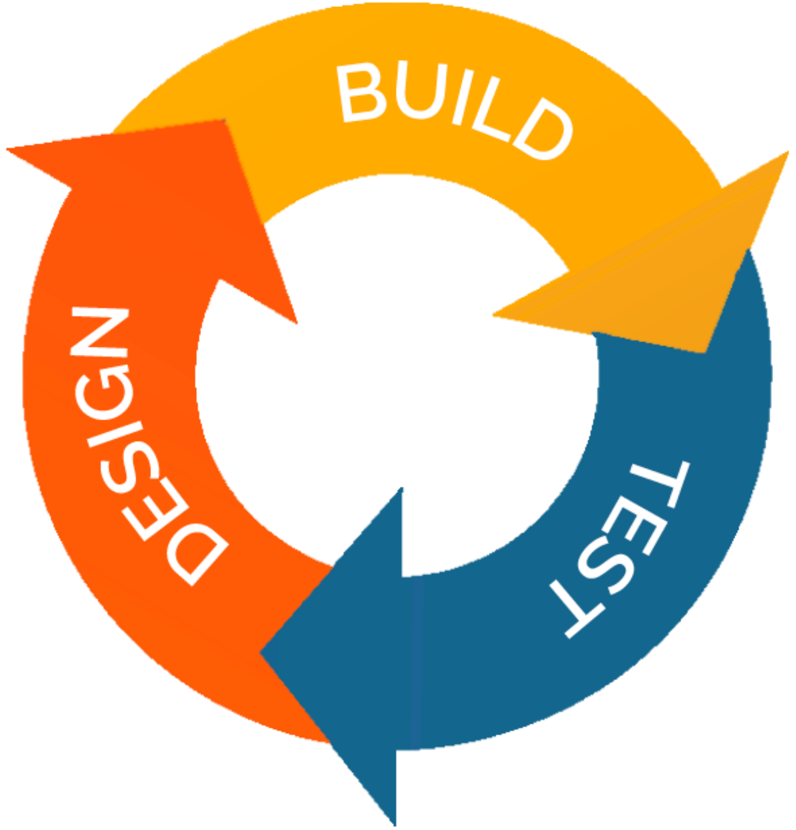
MDTD makes these distinctions less important.
The more general question is:
from what abstraction level do we derive tests?

Model-driven test design (2.5)

Test design is the process of designing input values that will effectively test software

Test design is one of the **several activities** for testing software

- Most **mathematical**
- Most **technically** challenging



Types of test activities

Testing can be broken up into **four** general types of activities

1. **Test design**
    1.a. **Criteria based**
    1.b. **Human-based**
2. **Test automation**
3. **Test execution**
4. **Test evaluation**

Each type of activity requires different **skills**, background **knowledge**, **education**, and **training**

1(A) TEST DESIGN – CRITERIA-BASED

Design test values to satisfy coverage criteria or other engineering goal

This is the **most technical** job in software testing

Requires **knowledge** of:

- discrete math
- programming
- testing

Requires much of a **traditional CS** degree

This is **intellectually** stimulating, rewarding, and challenging

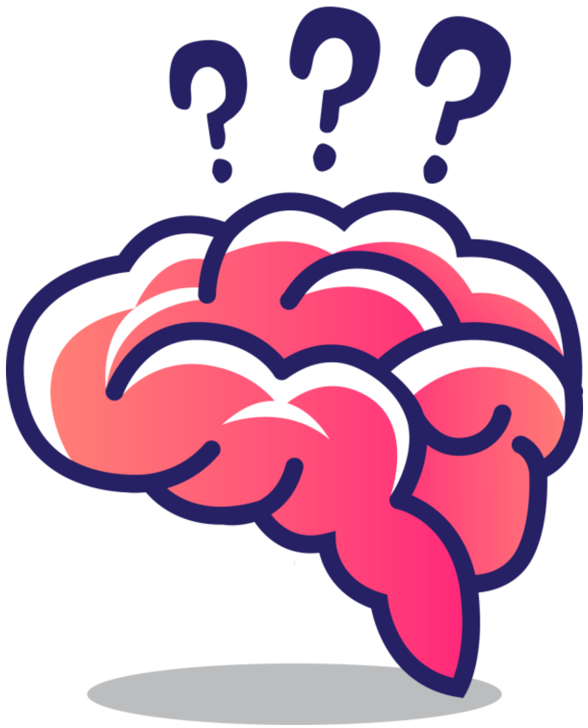
Test design is analogous to **software architecture** on the development side

Using people who are not qualified to design tests is a sure way to get **ineffective tests**

This class is primarily about criteria-based test design

1. Test Design – (b) Human-Based

Design test values based on domain knowledge of the program and human knowledge of testing



This is much **harder** than it may seem to developers

Criteria-based approaches can be blind to special situations

Requires **knowledge** of:

- domain, testing, and user interfaces

Requires almost **no traditional CS**

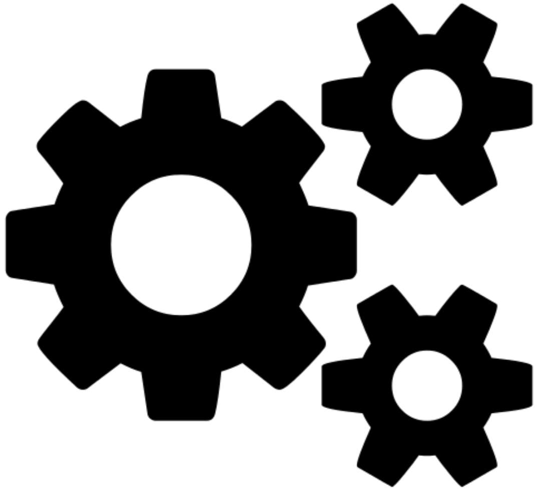
- a background in the **domain** of the software is essential
- an **empirical background** is very helpful (biology, psychology…)
- a **logic background** is very helpful (law, philosophy, math…)

This is **intellectually** stimulating, rewarding, and challenging

- But not to typical CS majors – they want to solve problems and build things

2. Test automation

Embed test values into executable scripts



This is slightly **less technical**

Requires knowledge of **programming**

Requires very **little theory**

Often requires solutions to difficult problems related to **observability** and **controllability**

Can be **boring** for test designers

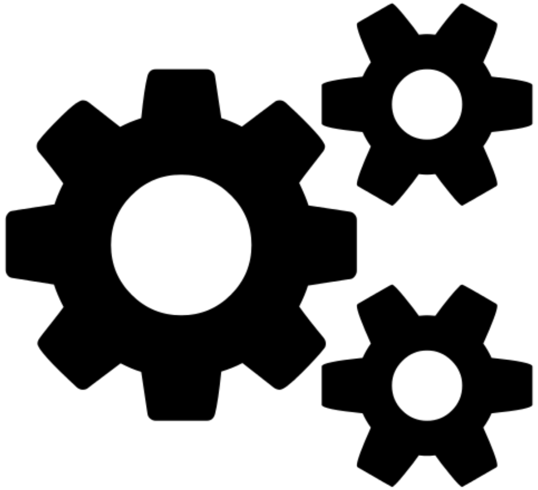
Programming is out of reach for many **domain experts**

Who is responsible for determining and embedding the **expected outputs**?

- **Test designers** may not always know the expected outputs
- **Test evaluators** need to get involved early to help with this

3. Test Execution

Run tests on the software and record the results



This is easy if the tests are well automated

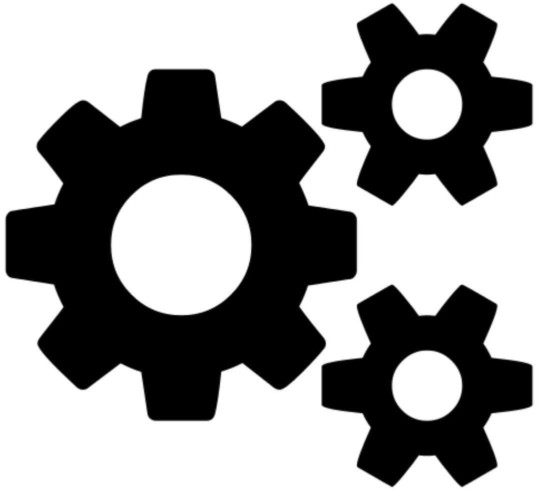
- Asking qualified test *designers* to execute tests is a sure way to convince them to look for a development job

If tests are not automated, this requires a lot of manual labor

Test executors have to be very careful and meticulous with bookkeeping

4. Test Evaluation

Evaluate results of testing, report to developers



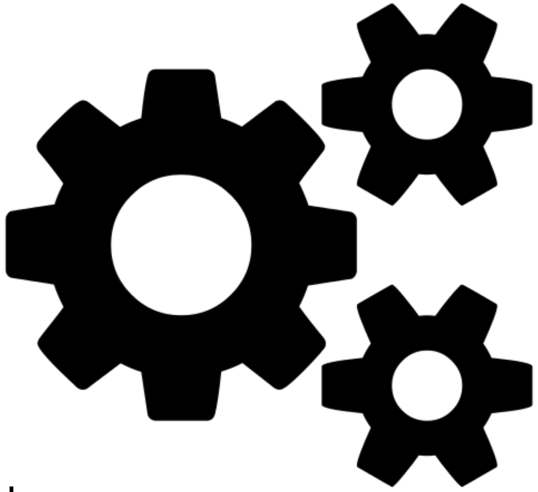
This is much harder than it may seem

Requires extensive domain knowledge

This is intellectually stimulating, rewarding, and challenging

- But not to typical software developers – they want to solve problems and build things

Other testing activities



Test management: Set policy, organize teams, interface with development, choose criteria, decide how much automation needed…

Test maintenance: Save test for reuse as software evolves

- requires cooperation of test designers and automators
- Deciding when to trim the test suite is partly policy, partly technical – and in general, very hard!
- Test should be put in configuration control

Test documentation: all parties participate

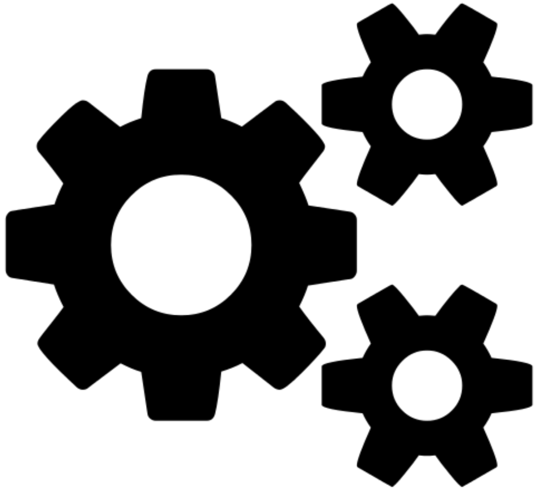
- Each test must document "why" -- criterion and test requirement satisfied or rational for human-designed tests
- Ensure traceability
- Keep documentation in automated tests

Using MDTD in Practice

This approach lets one test designer do theory

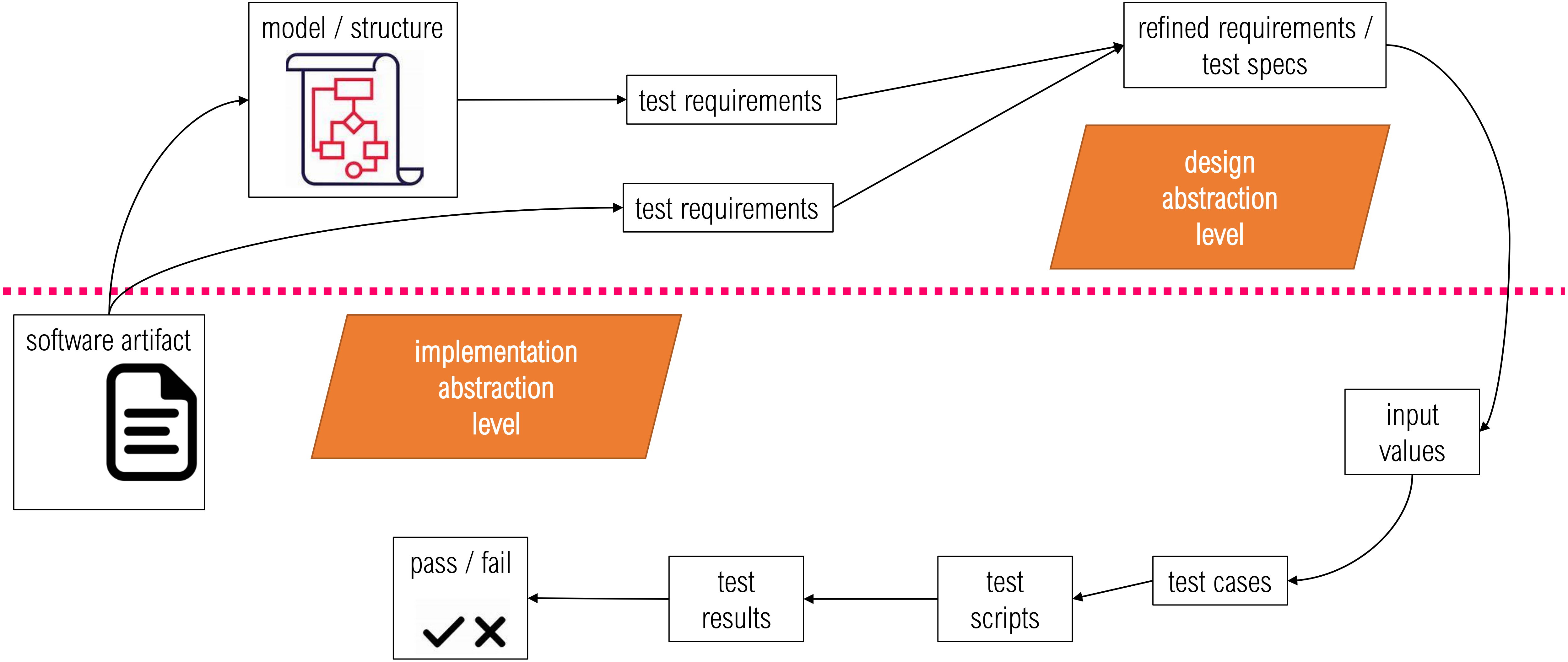
Then traditional testers and programmers can do their parts

- Find values
- Automate tests
- Run tests
- Evaluate tests

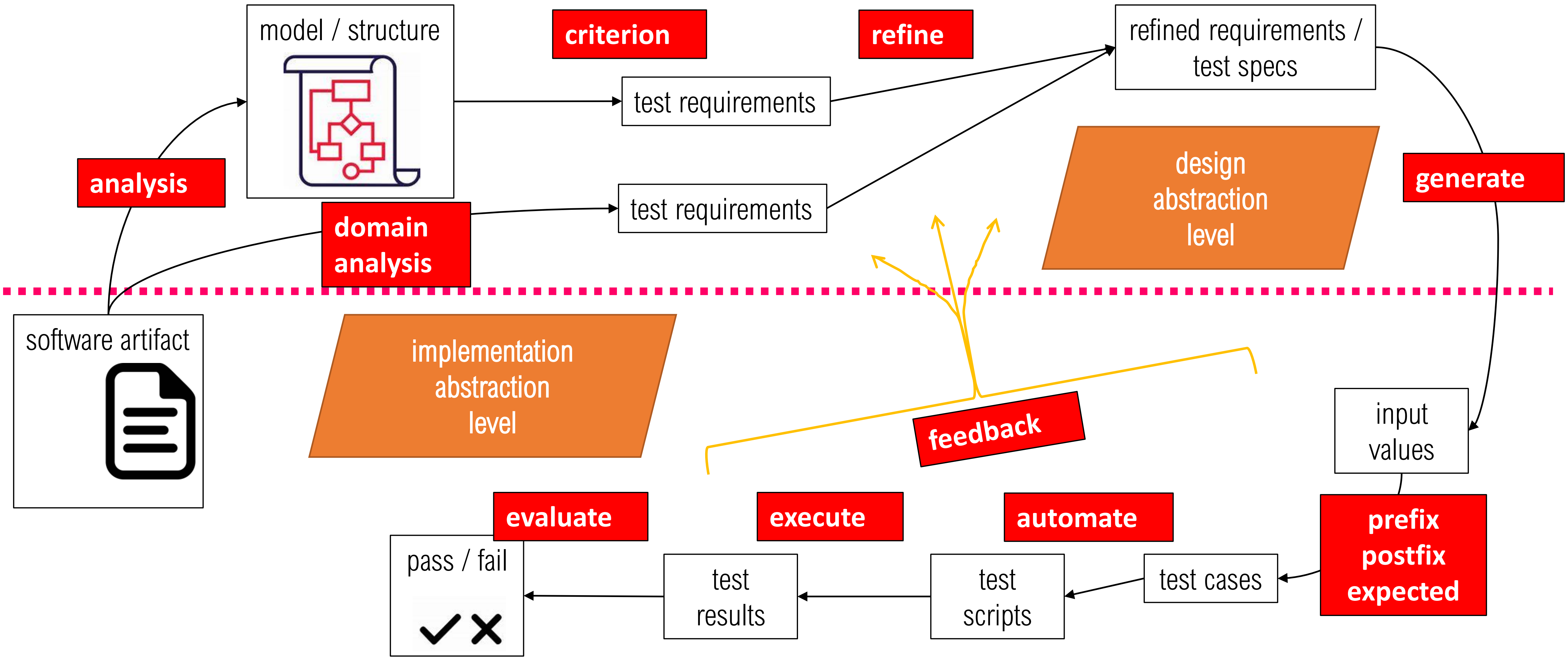


Just like *traditional engineering*...an engineer constructs models calculus, then gives directions to carpenters, electricians, etc...

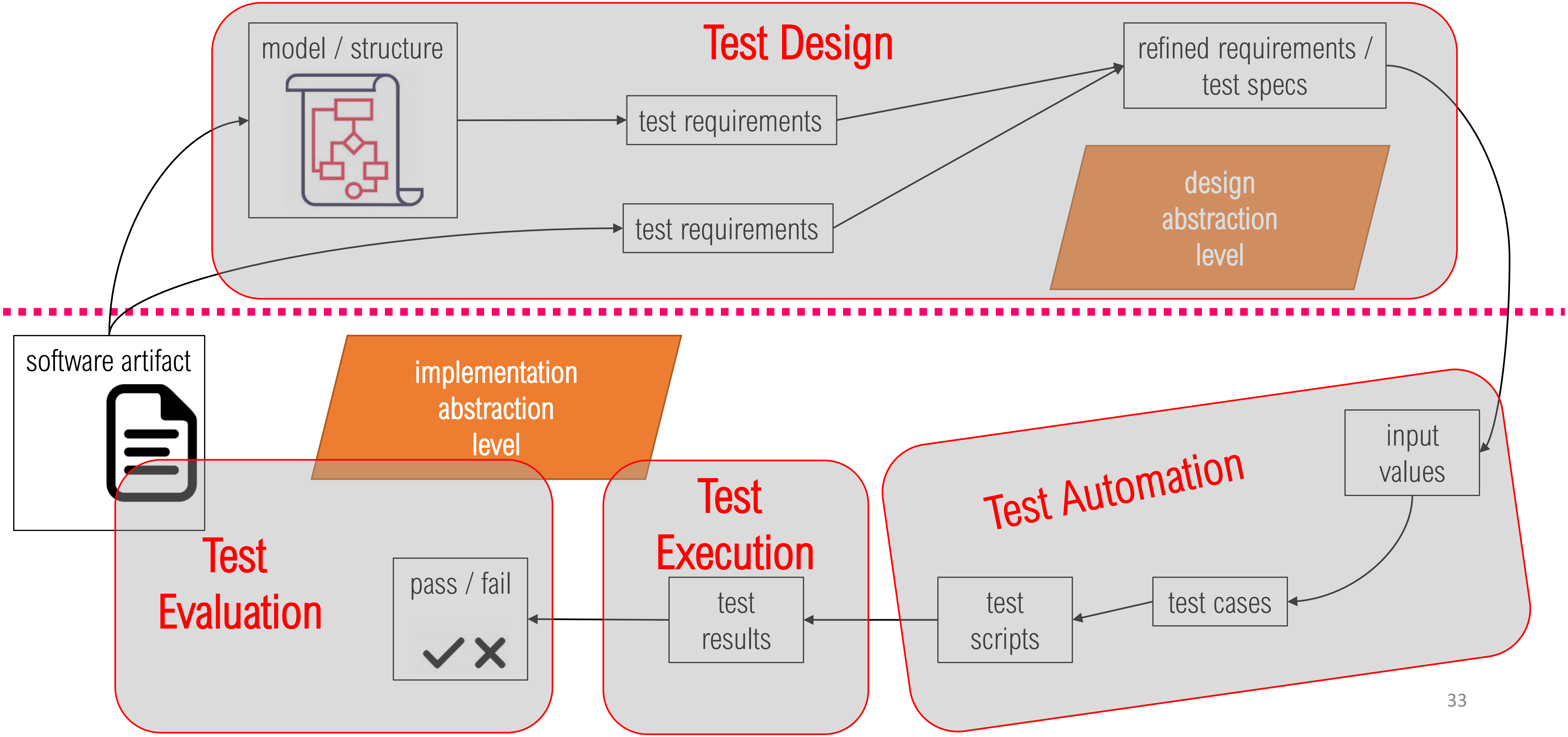
Model-Driven Test Design



Model-Driven Test Design Steps



Model-Driven Test Design Activities

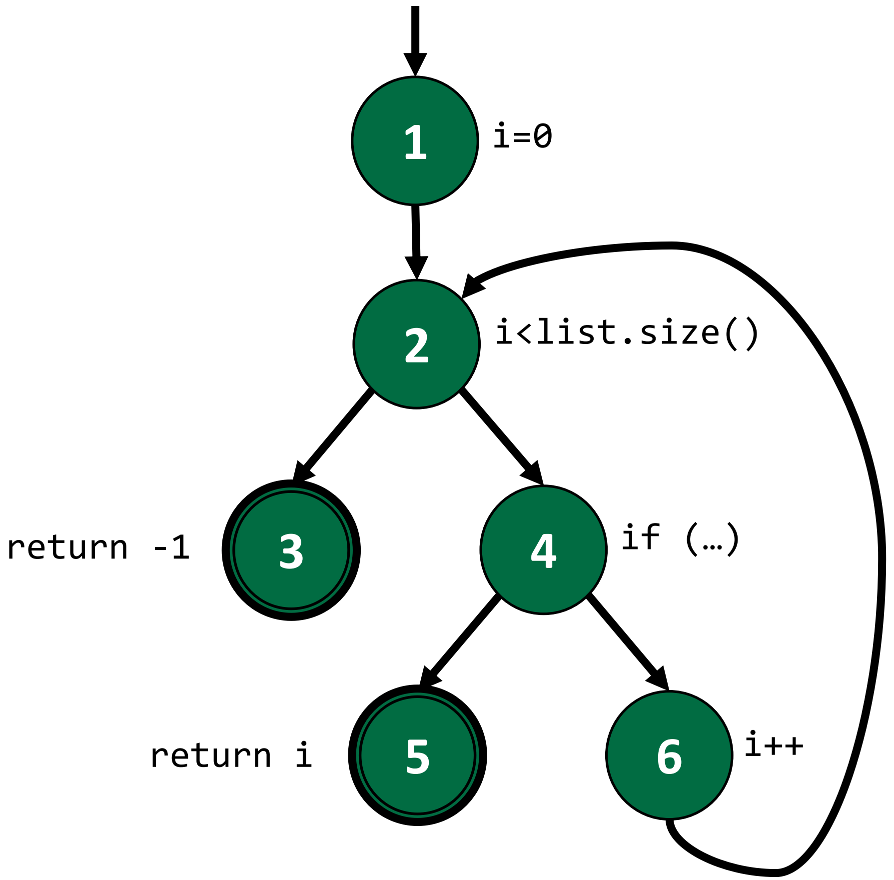


Small example

Software Artifact : Java Method

```
    * Return index of node n at the
    * first position it appears,
    * -1 if it is not present
*/
public int indexOf (Node n)
{
    for (int i=0; i < path.size(); i++)
        if (path.get(i).equals(n))
            return i;
    return -1;
}
```

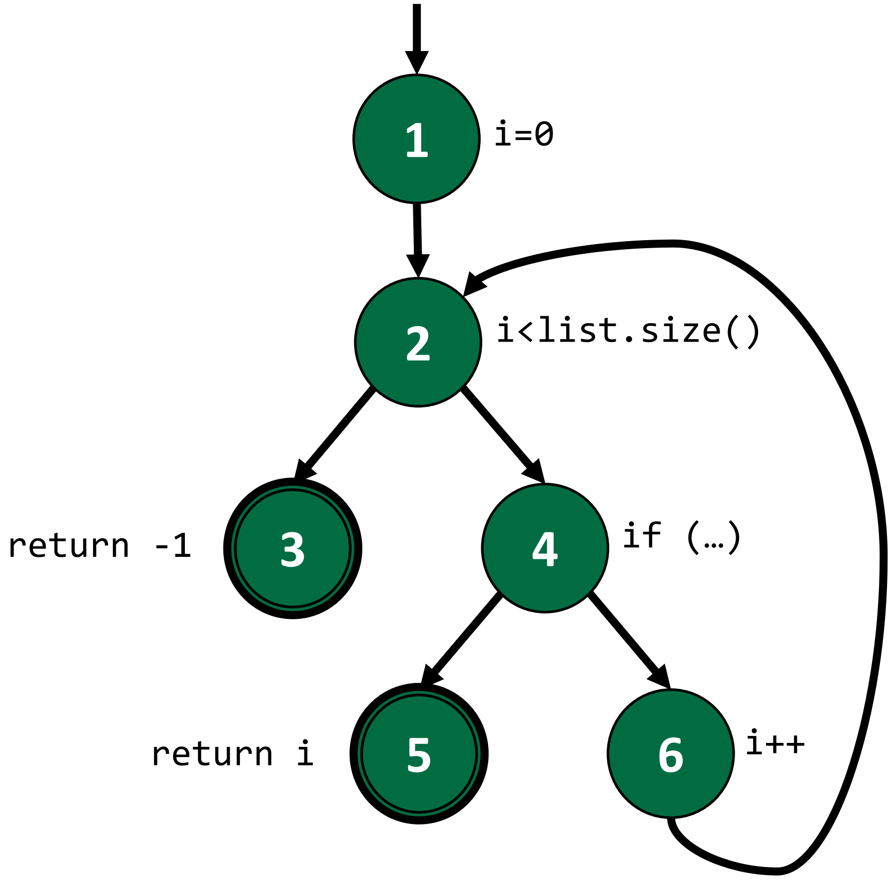
Control Flow Graph



Example (continued)

Support tool for graph coverage

http://www.cs.gmu.edu/~offutt/softwaretest/



Initial Node: 1

Final Nodes: 3, 5

Edges:

(1, 2)

(2, 3)

(2, 4)

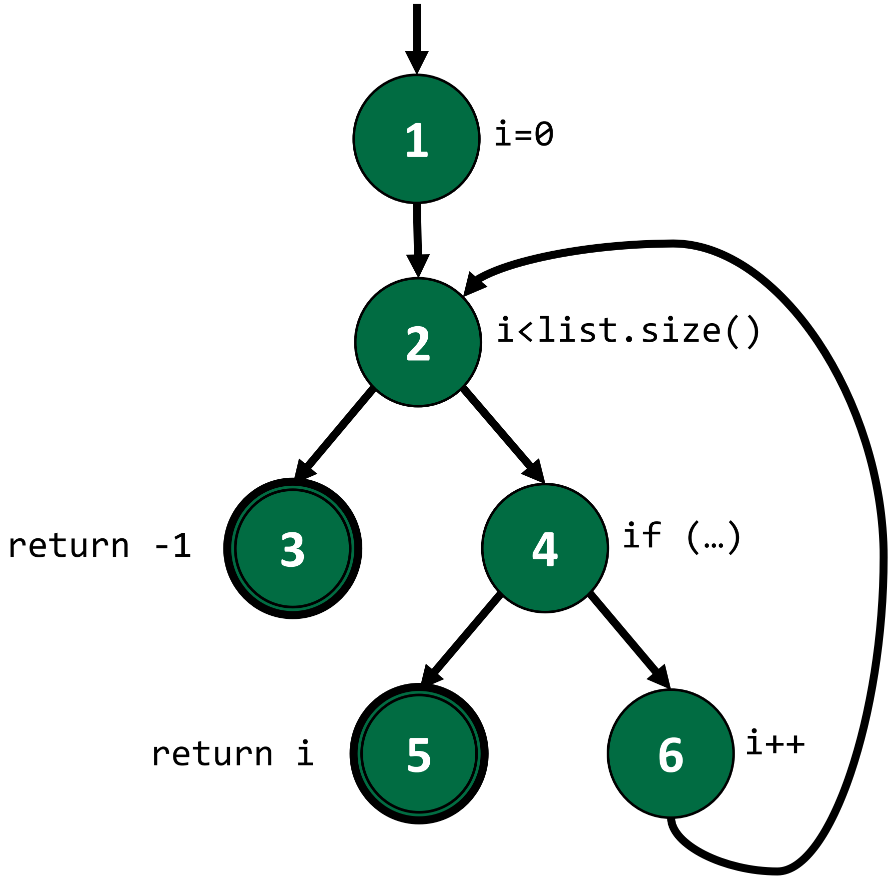
(4, 5)

(4, 6)

(6, 2)

Example (continued)

Support tool for graph coverage
http://www.cs.gmu.edu/~offutt/softwaretest/



6 requirements for Edge Coverage

1. [1, 2]
2. [2, 3]
3. [2, 4]
4. [4, 5]
5. [4, 6]
6. [6, 2]

Test Paths

[1, 2, 3]

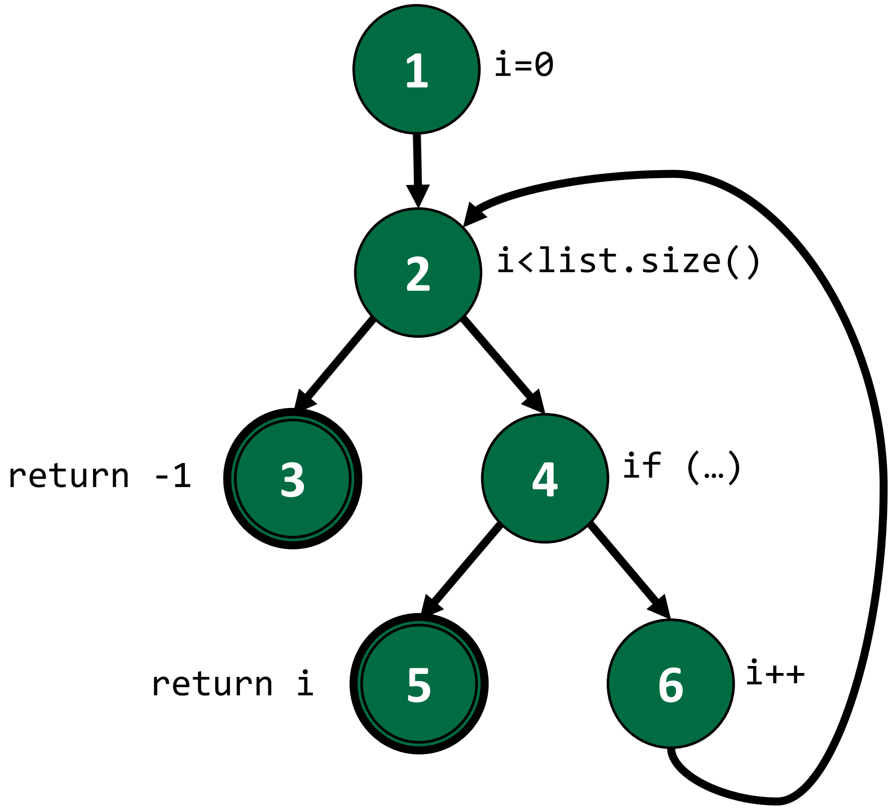
[1, 2, 4, 6, 2, 4, 5]

Next we need to find values to execute those test paths

Example (continued)

Support tool for graph coverage

http://www.cs.gmu.edu/~offutt/softwaretest/



Test Path [1, 2, 3]

list = {}

o = null

Test Path [1, 2, 4, 6, 2, 4, 5]

list = {1, 2}

o = 2

We'll talk about implementation in future classes