Chapter 2
Model-Driven Test Design

Brittany Johnson
SWE 437

Adapted from slides by Paul Ammann & Jeff Offutt

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

In-Class Exercise

Discuss

software correctness

Have you thought of correctness in software as possible or impossible?

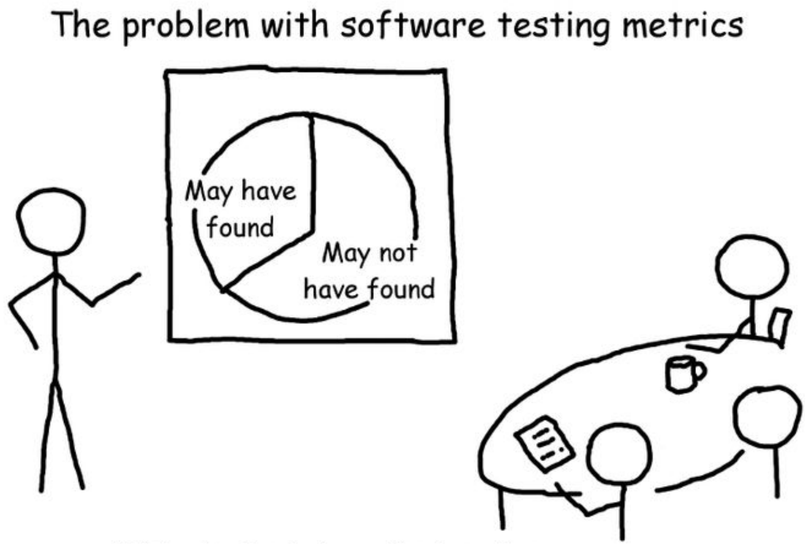
Do you agree with the claim in the book, or is it hard to accept?

You have five minutes.

Software Testing Foundations (2.1)

Testing can only show the presence of failures

Not their absence



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

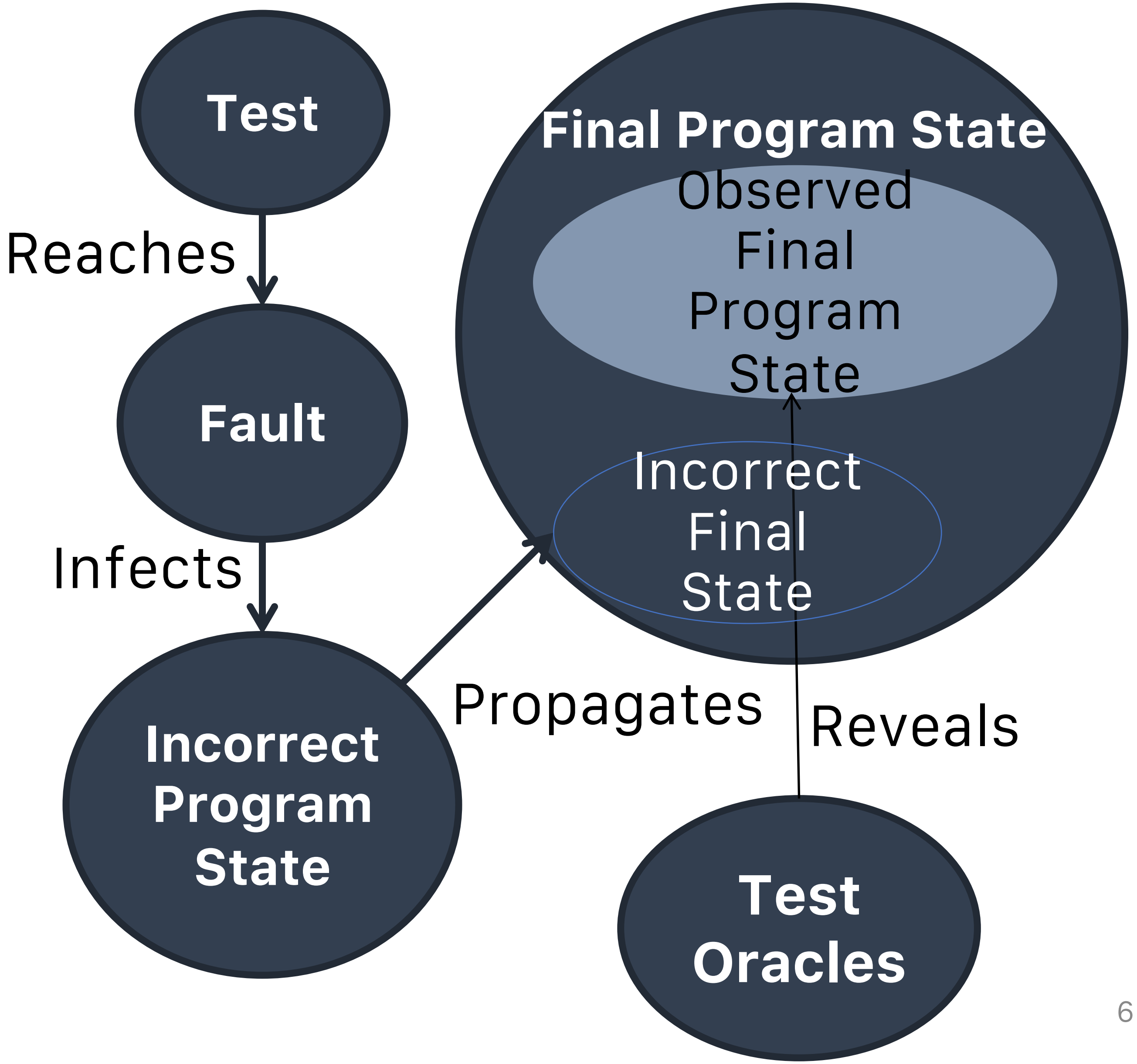
Fault & Failure Model (RIPR)

<u>Four conditions necessary for a failure to be observed</u>

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.

RIPR Model

- Reachability
- Infection
- Propagation
- Revealability



In-Class Exercise

Discuss

test oracles

Have you written any automated tests?

How did you decide what assertions to write?

Do you think you every checked the wrong part of the state?

You have five minutes.

Traditional Testing Levels (2.3)

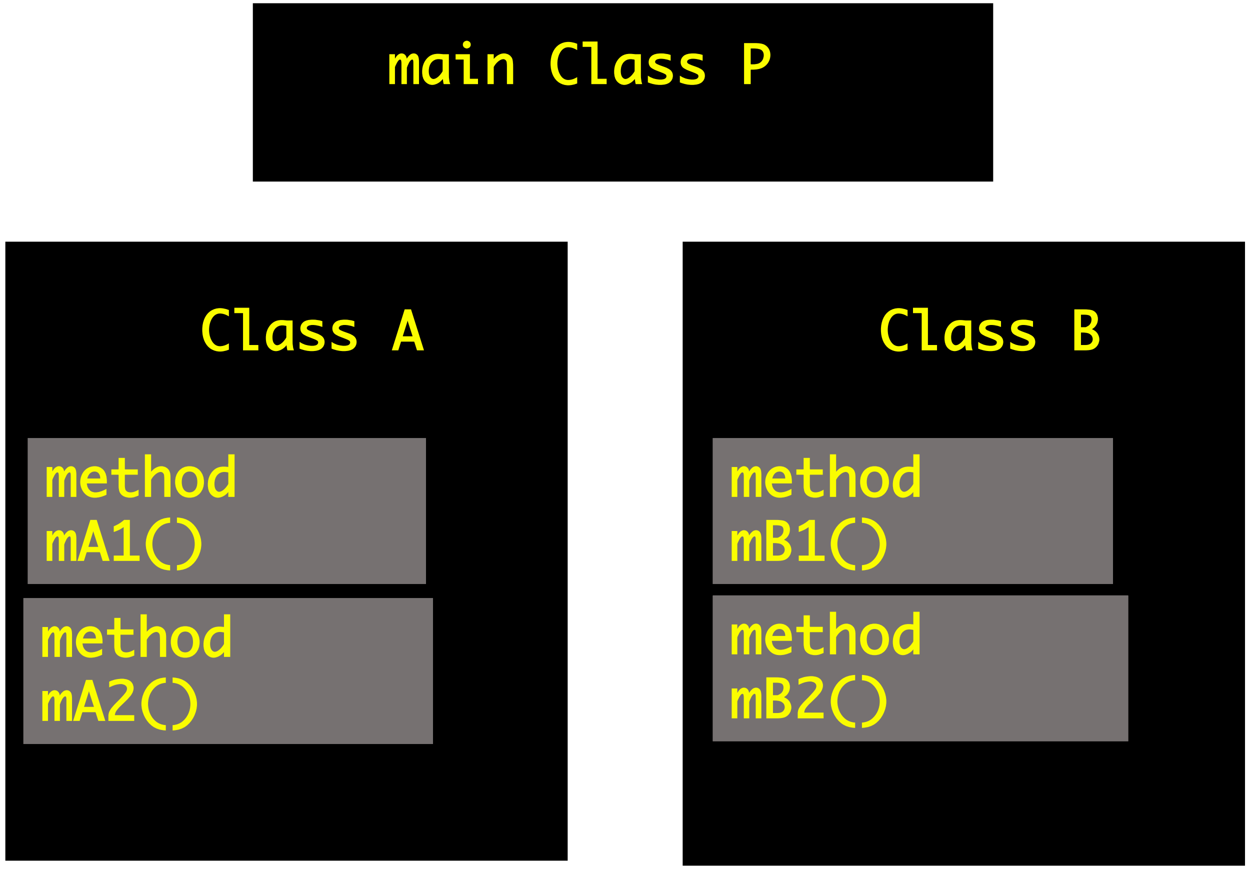
Acceptance testing

Systems testing

Integration testing

Module testing (developer testing)

Unit testing (developer testing)



Traditional Testing Levels (2.3)

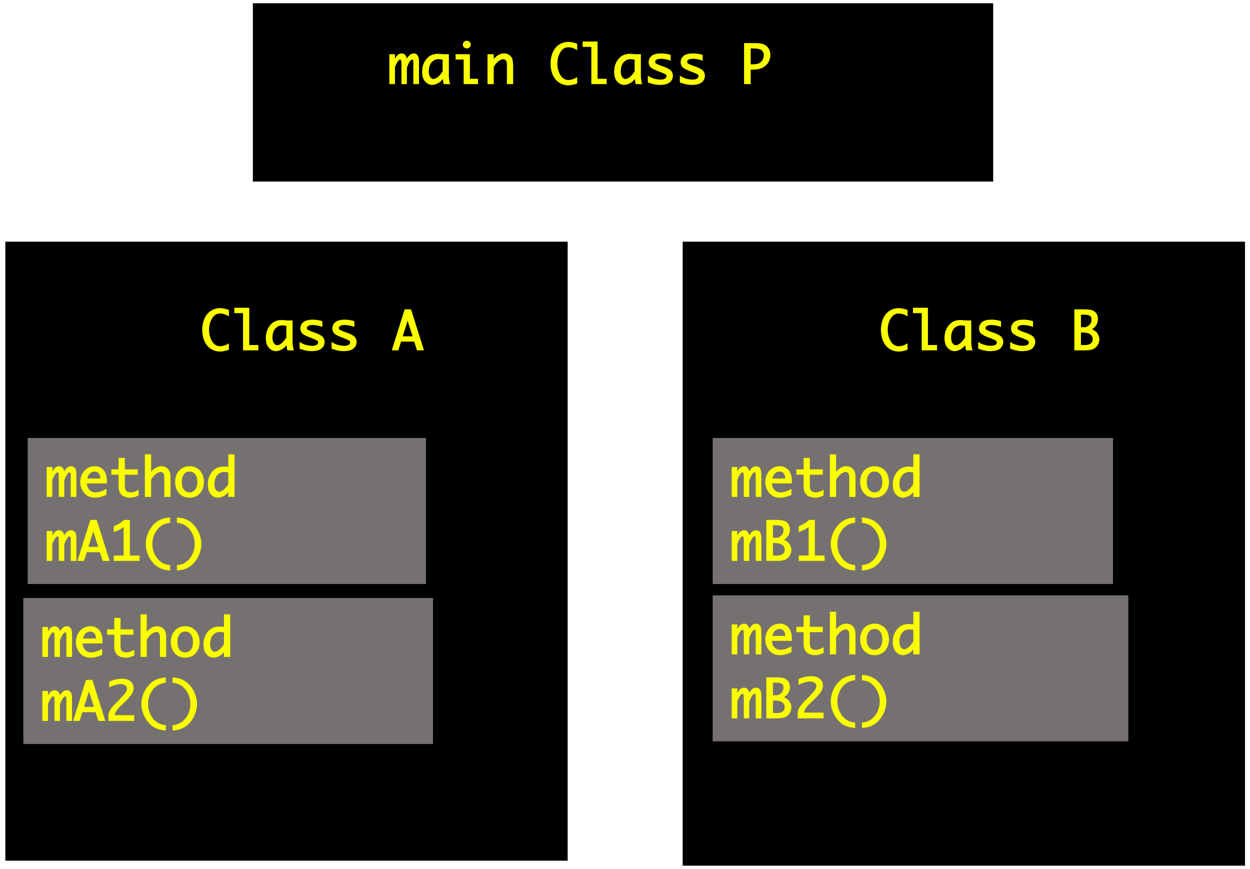
Acceptance testing: Is the software acceptable to the user?

Systems testing

Integration testing

Module testing (developer testing)

Unit testing (developer testing)



Traditional Testing Levels (2.3)

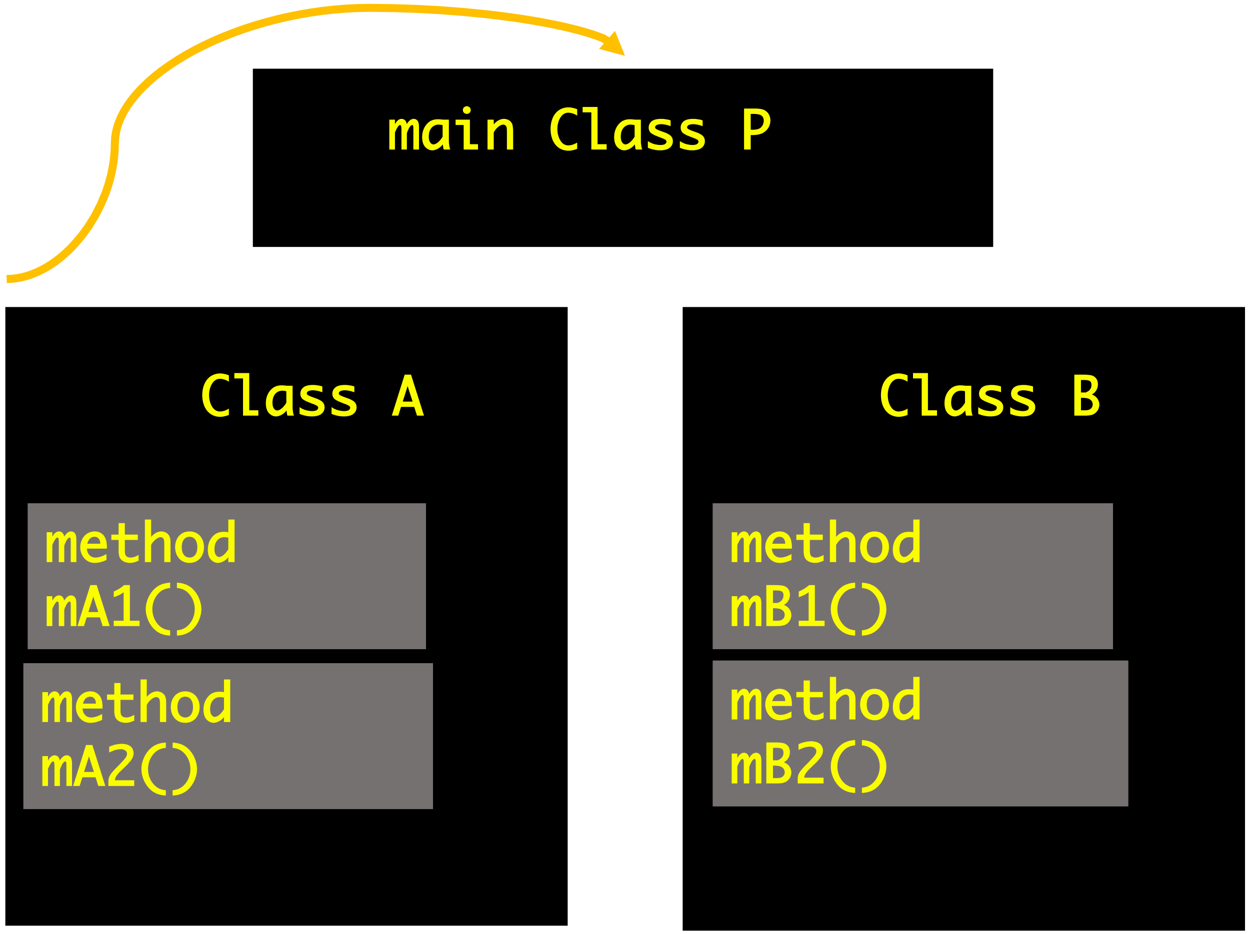
Acceptance testing: Is the software acceptable to the user?

Systems testing: Test the overall functionality of the system

Integration testing

Module testing (developer testing)

Unit testing (developer testing)



Traditional Testing Levels (2.3)

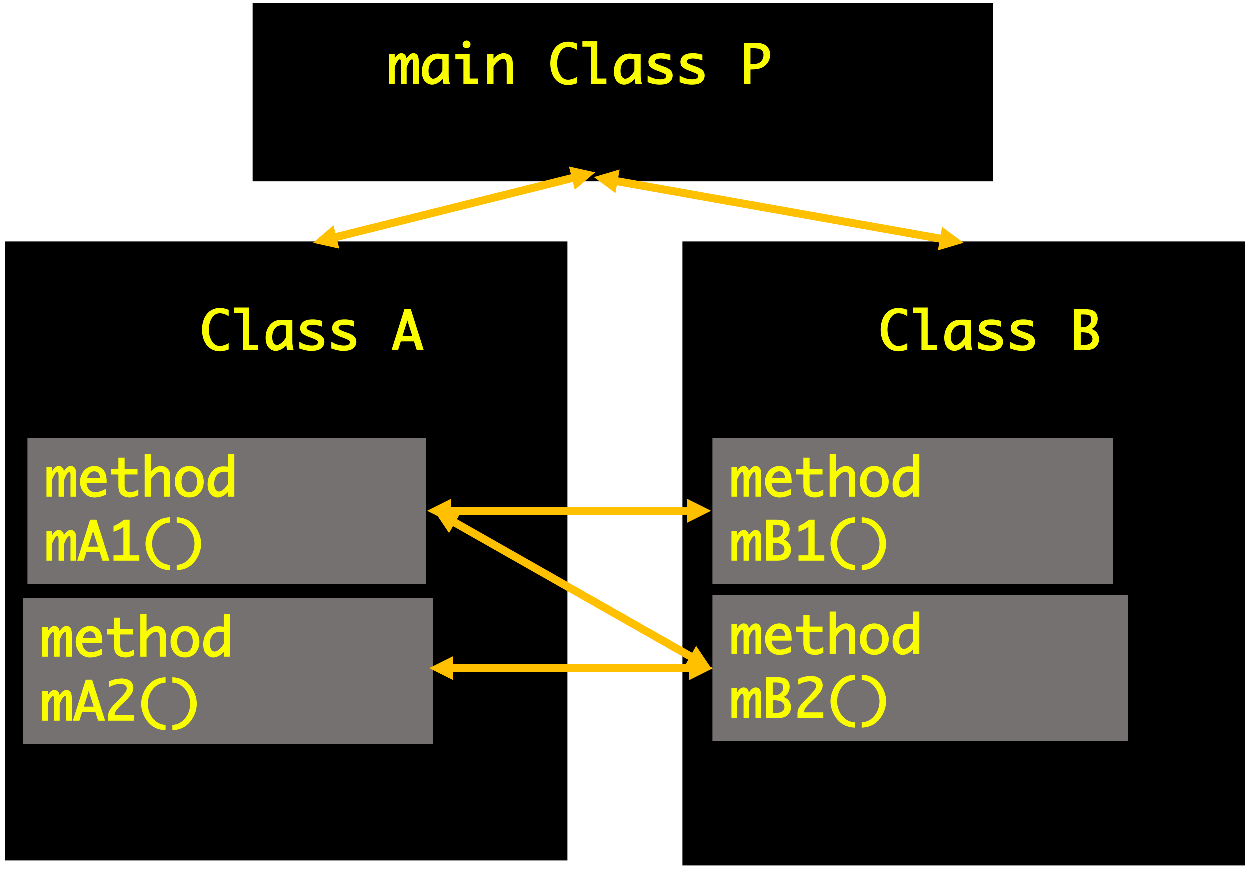
Acceptance testing: Is the software acceptable to the user?

Systems testing: Test the overall functionality of the system

Integration testing: Test how modules interact with each other

Module testing (developer testing)

Unit testing (developer testing)



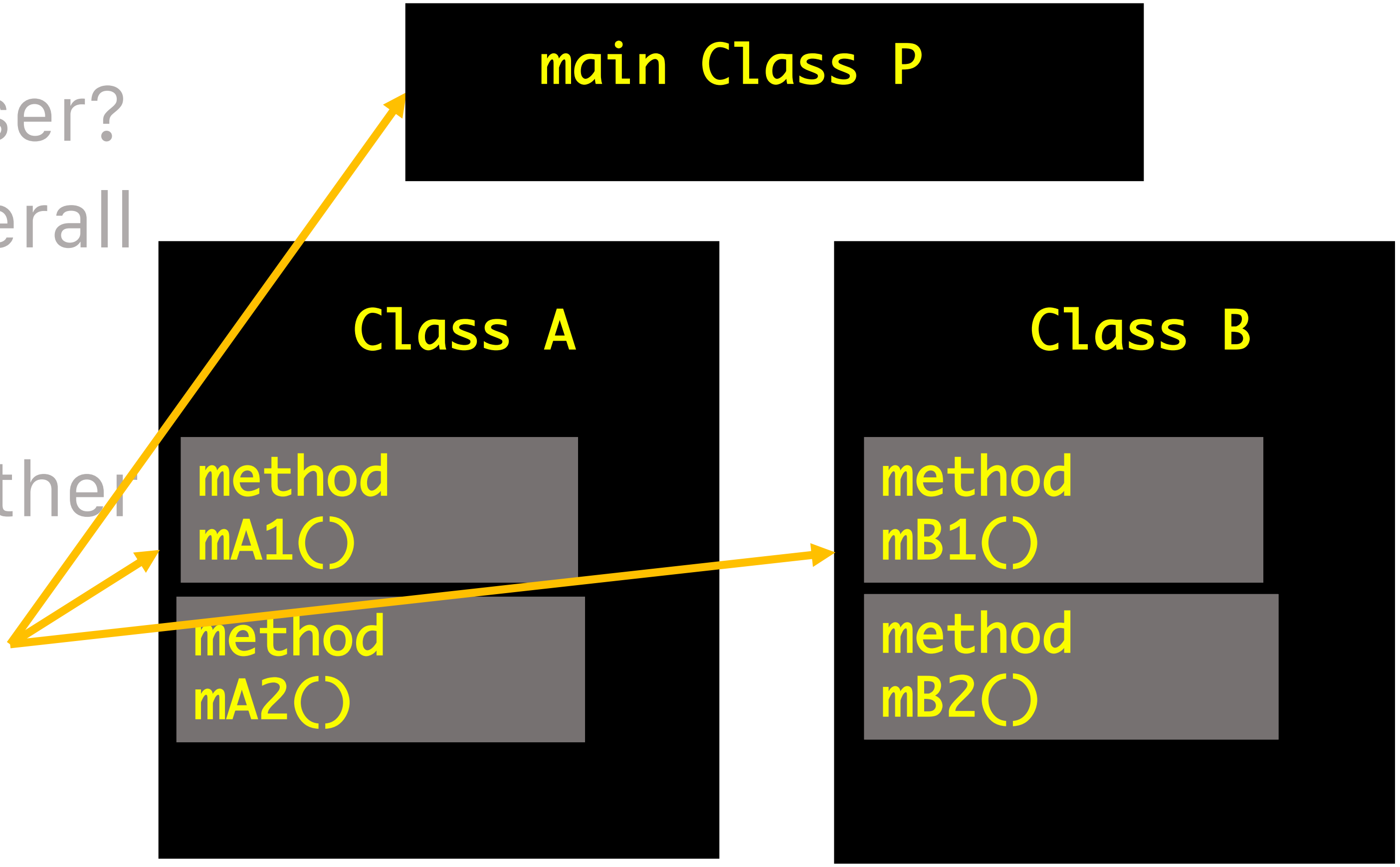
Traditional Testing Levels (2.3)

Acceptance testing: Is the software acceptable to the user?

Systems testing: Test the overall functionality of the system

Integration testing: Test how modules interact with each other

Module testing (developer testing): Test each class, file, module, component



Unit testing (developer testing)

Traditional Testing Levels (2.3)

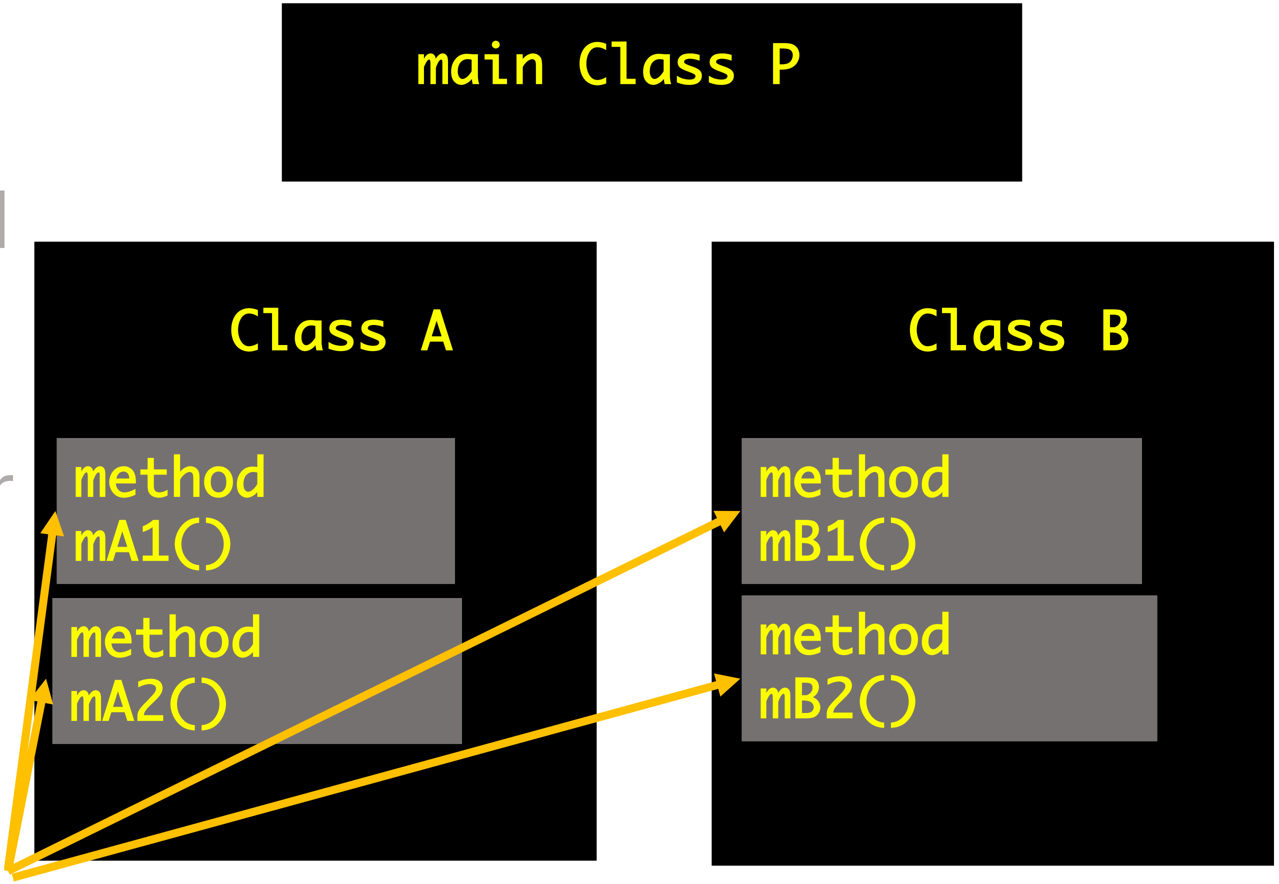
Acceptance testing: Is the software acceptable to the user?

Systems testing: Test the overall functionality of the system

Integration testing: Test how modules interact with each other

Module testing (developer testing): Test each class, file, module, component

Unit testing (developer testing): Test each unit (method) individually



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 ahs over **4 billion** possible values
- Over **80 octillion possible tests**!!
- Input space might as well be infinite

Testers **search** a huge input space

- Trying to find the **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**

JACOCO
Java Code Coverage

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

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**

No reasonable software dev organization uses the same people for requirements, design, implementation, integration, and configuration control

Why do test organizations still use the same people for all four test activities??

This clearly wastes resources.

1. Test design – (a) 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**

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**

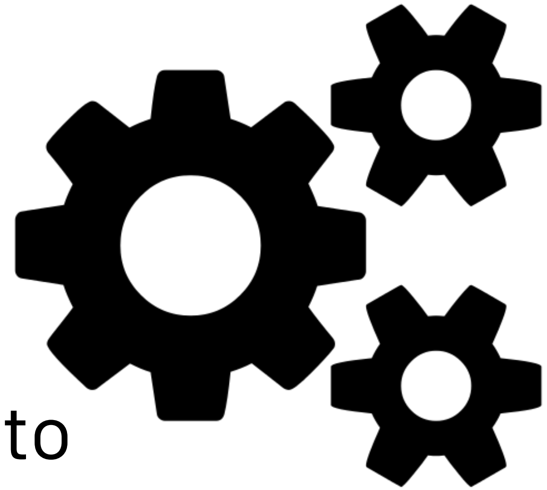
- a background in the **domain** of the software is essential
- an **empirical background** is very helpful (biology, psychology…)
- a **logic background** is very helpful (law, philosophy, math…)

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**

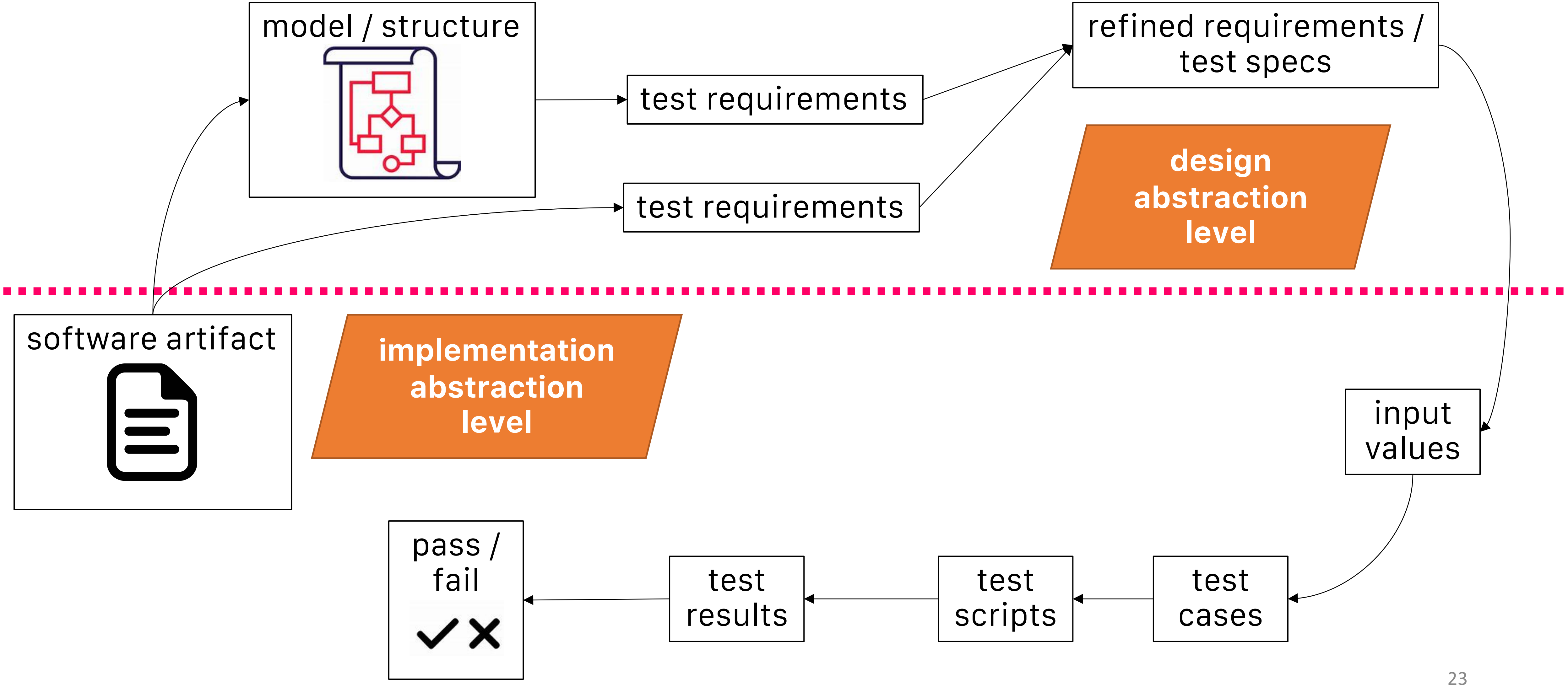
Can be **boring** for test designers

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

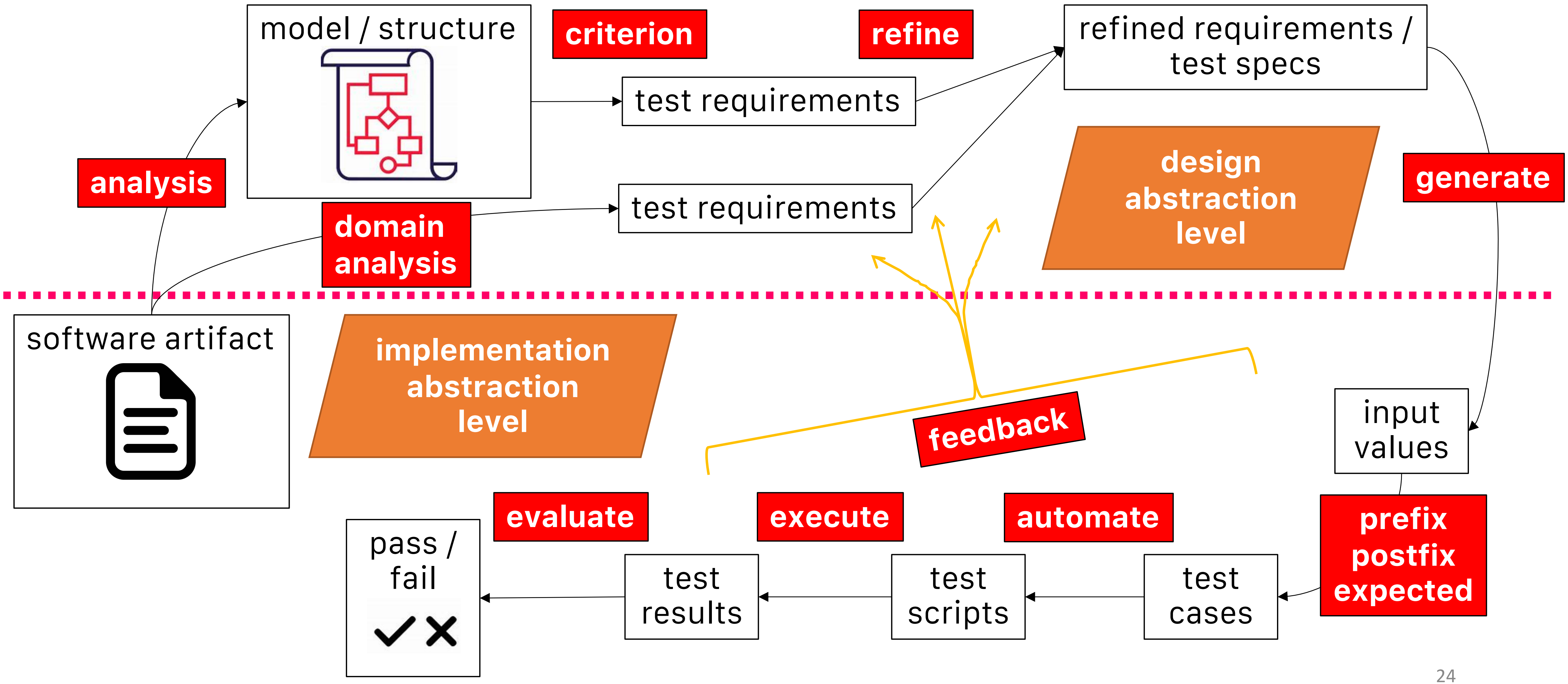
Who is responsible for determining and embedding the **expected outputs**?

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

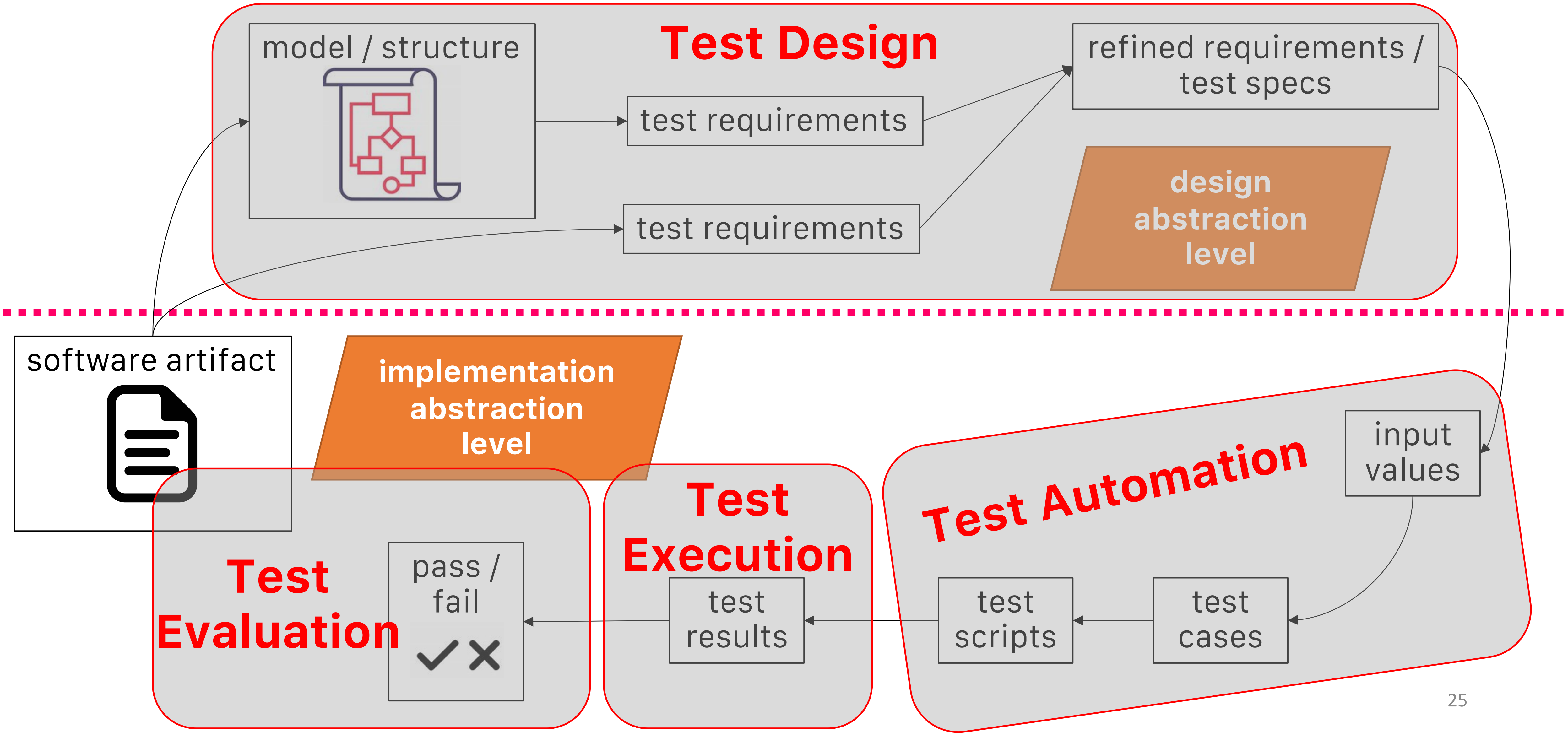
Model-driven test design



Model-driven test design - steps



Model-driven test design - activities

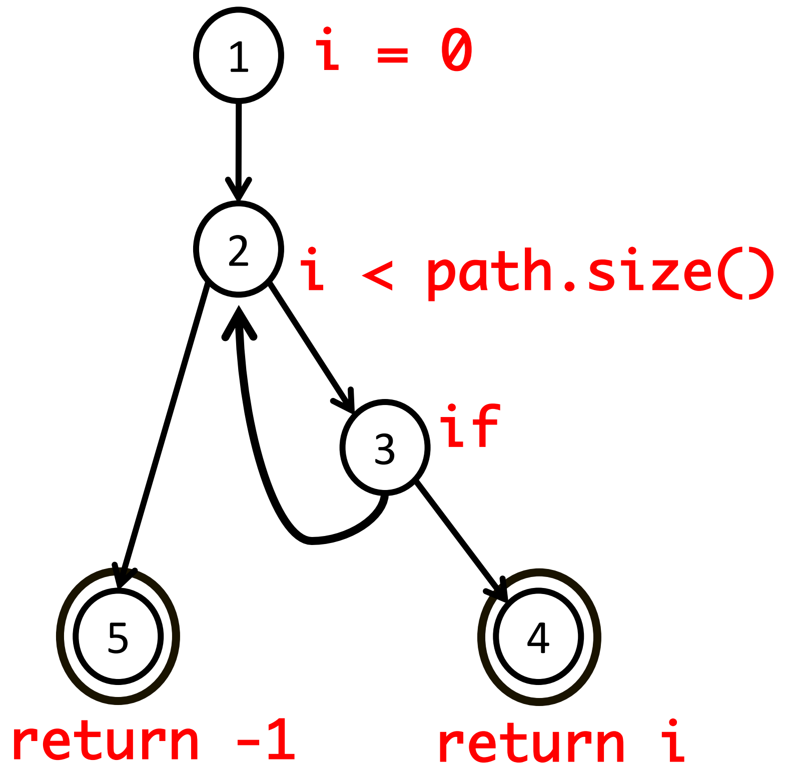


Small example

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;
}
```

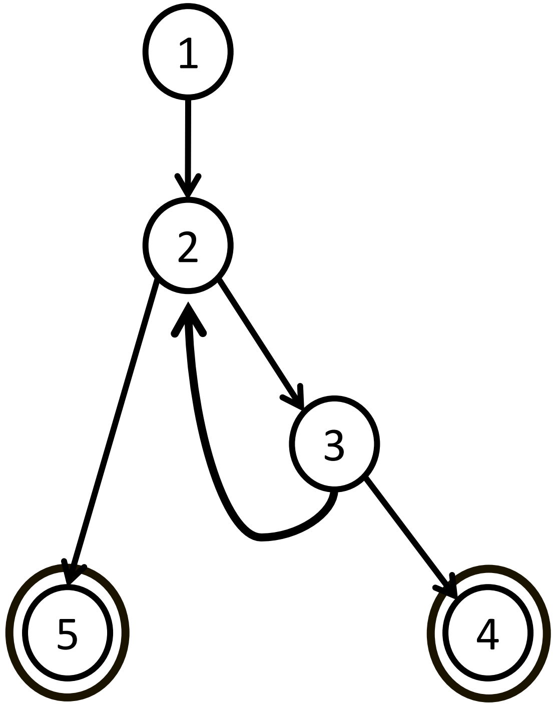
Control Flow Graph



Example (continued)

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

Abstract graph version



Edges
1 2
2 3
3 2
3 4
2 5
Initial Node: 1
Final Nodes: 4, 5

6 requirements for Edge-Pair Coverage
1. [1, 2, 3]
2. [1, 2, 5]
3. [2, 3, 4]
4. [2, 3, 2]
5. [3, 2, 3]
6. [3, 2, 5]

Test Paths
[1, 2, 5]
[1, 2, 3, 2, 5]
[1, 2, 3, 2, 3, 4]

Types of activities in the book

Most of this book is about test design

Other activities are well covered elsewhere

In-Class Exercise

Discuss

coverage criteria

Why do software orgs use coverage criteria?

Why don’t more software orgs use coverage criteria?

You have five minutes.