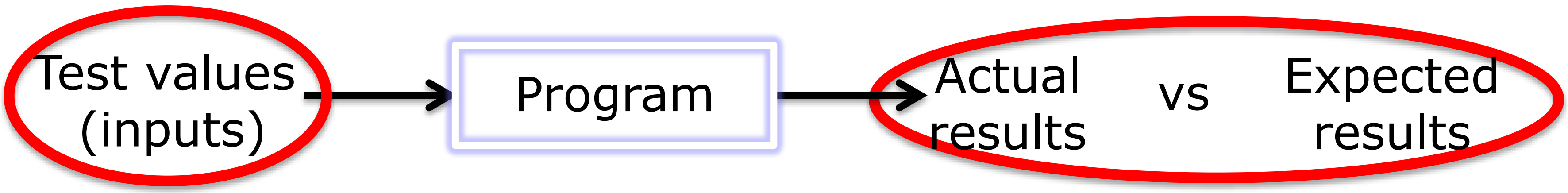
Model-Driven Test Design

[Ammann and Offutt, "Introduction to Software Testing"]
Model-Driven Test Design (Chapter 2)
Extra Slides are from CSC 179/234 Fall 2017

CSc Dept, CSUS

Recap: What is Software Testing?

- Testing = process of finding input values to **check** against a software

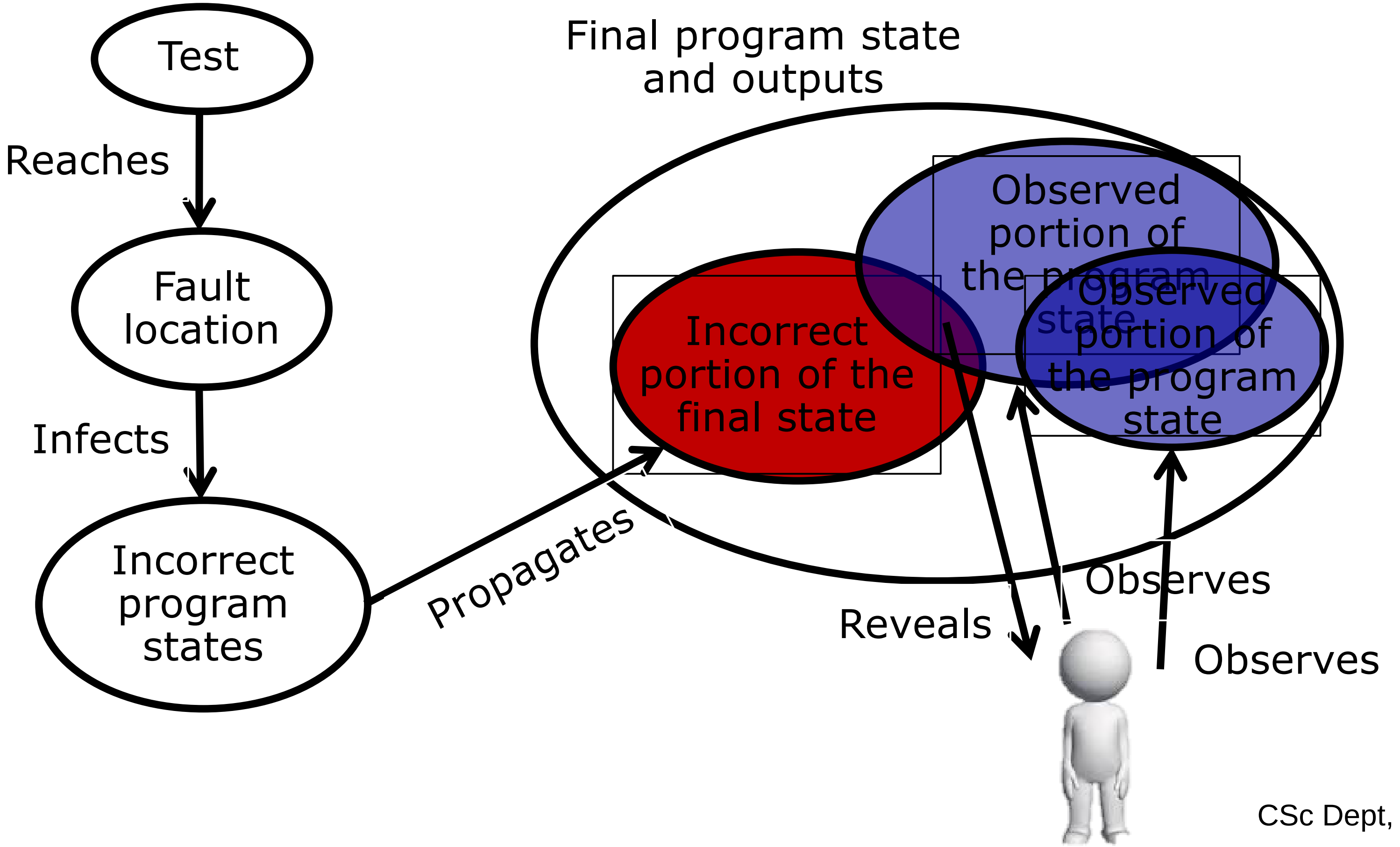


Test failure: actual results ≠ expected results
(execution of a test that results in a software failure)

Testing can only show the presence of failure, not their absence

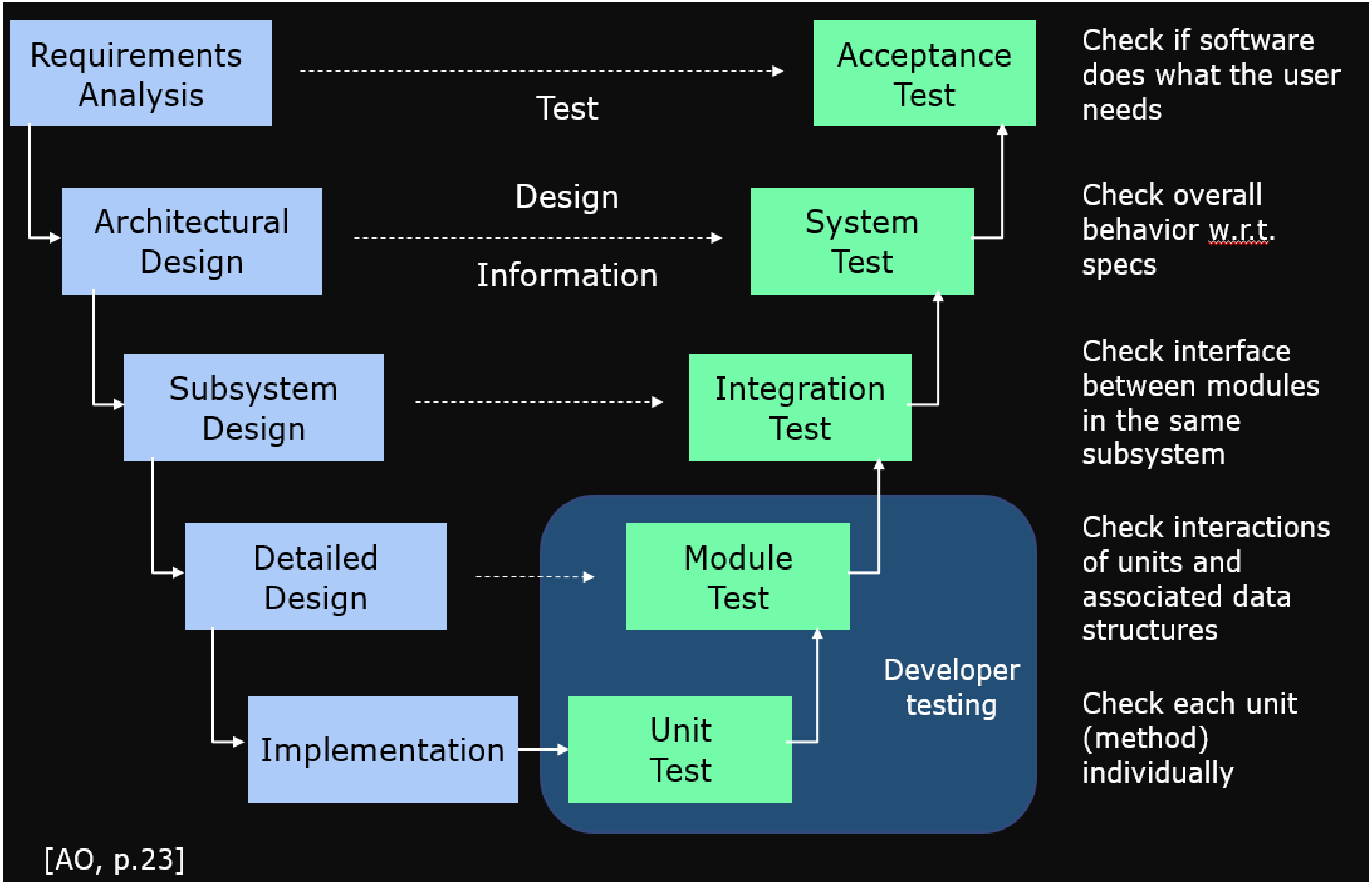
Recap: RIPR Model

- Sometimes refer to as Fault, Error, Failure model
- Not all inputs will “trigger” a fault into causing a failure

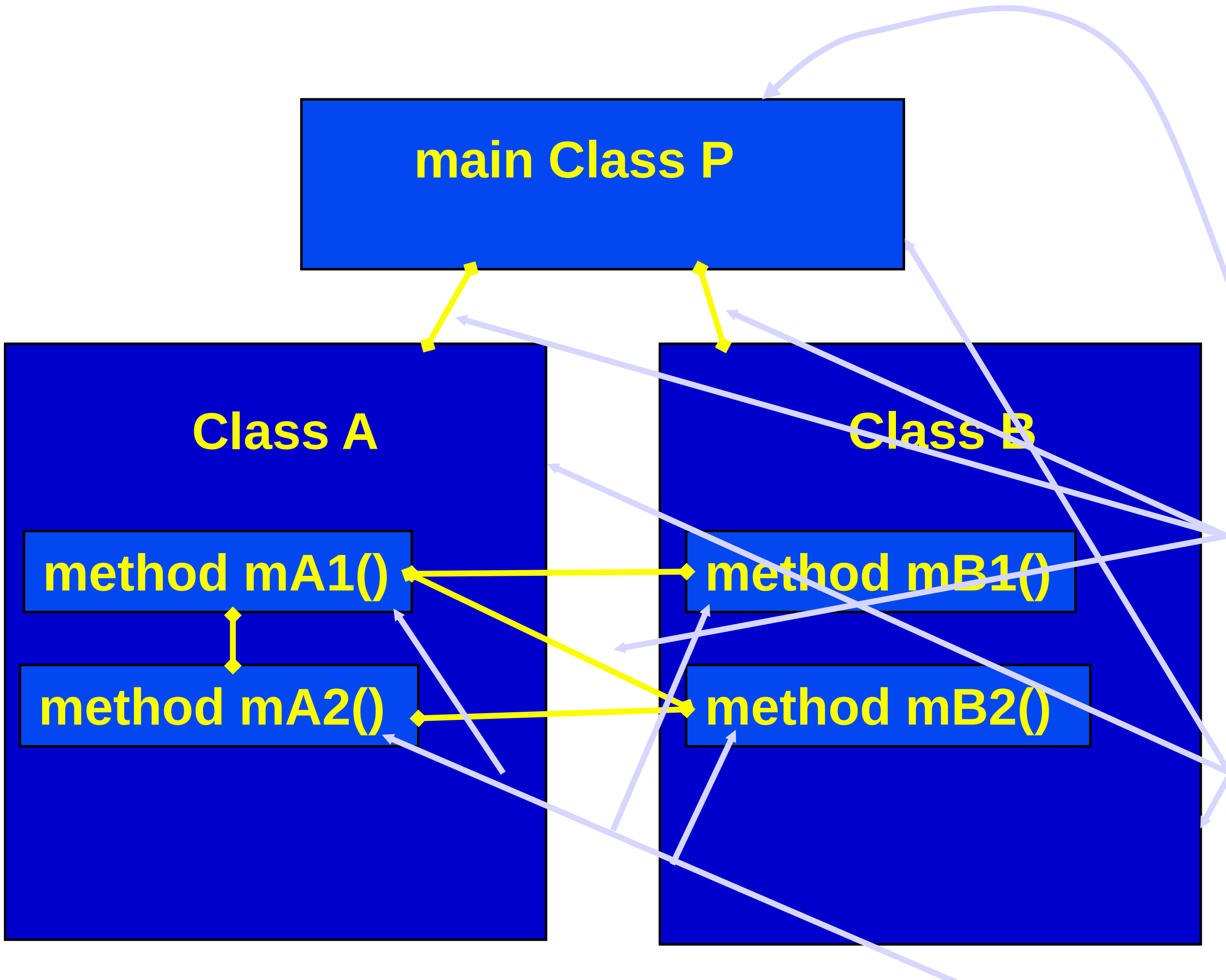


[AO, p.21]

Testing Levels and Types of Faults



Traditional Testing Levels

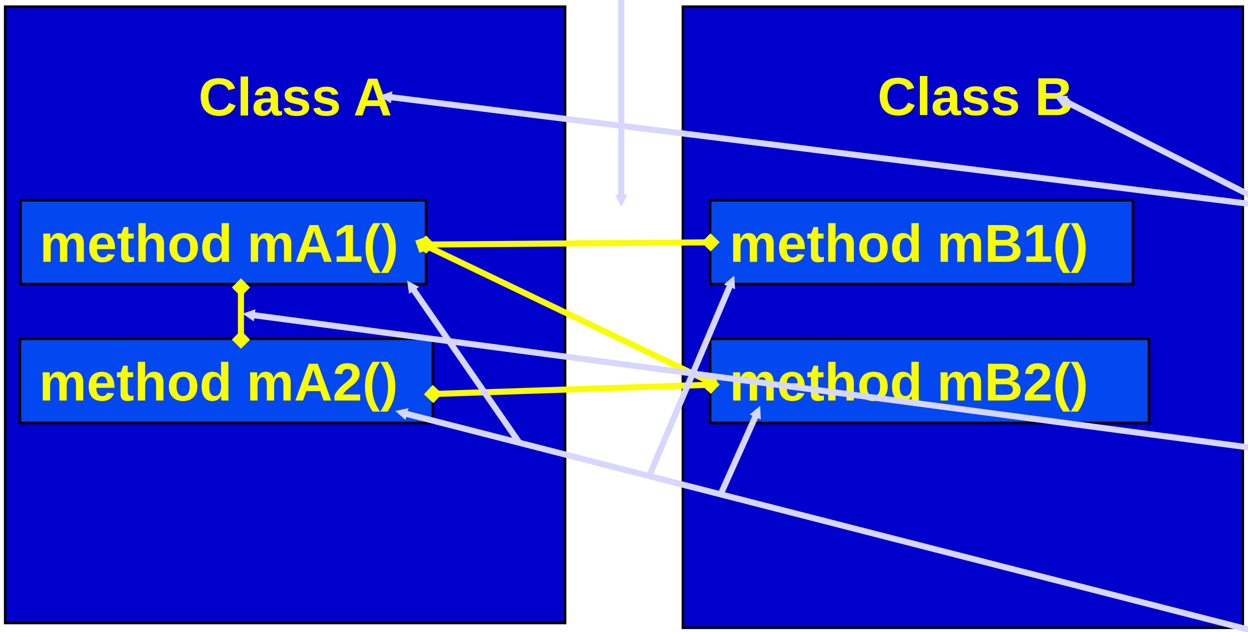


- **Acceptance testing** : Is the software acceptable to the user?
- **System testing** : Test the overall functionality of the system
- **Integration testing** : Test how modules interact with each other
- **Module testing (developer testing)** : Test each class, file, module, component
- **Unit testing (developer testing)** : Test each unit (method) individually

Object-Oriented Changes Testing Levels

- **Inter-class testing** : Test multiple classes together

integration testing!

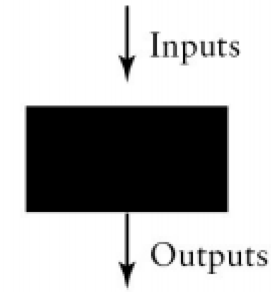


- **Intra-class testing** : Test an entire class as sequences of calls
- **Inter-method testing** : Test pairs of methods in the same class
- **Intra-method testing** : Test each method individually

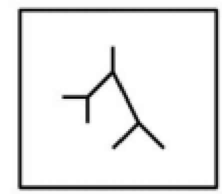
The first three are variations of unit and module testing!

Old View: Colored Boxes

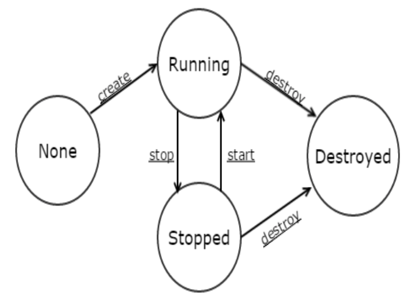
- Black-box testing
  - Derive tests from external descriptions of the software, including specifications, requirements, and design



- 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, FSM based testing)

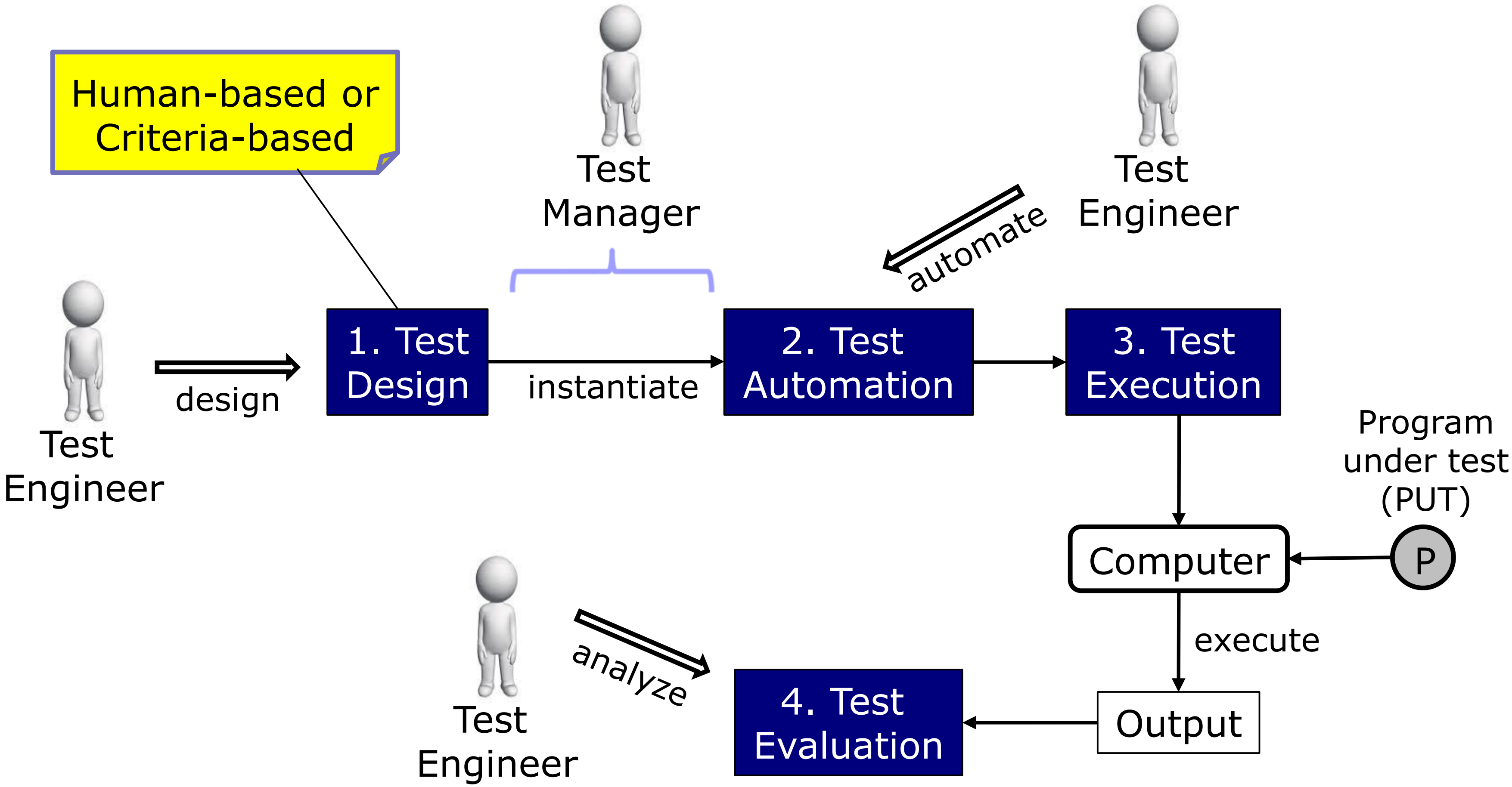


- Model-Driven Test Design
  - Makes the distinctions less important by focusing on “from what abstraction level do we derive tests?”

Model-Driven Test Design

- Breaks testing into a series of **small tasks** that simplify test generation
- **Isolate** each task
- Work at a higher level of **abstraction**
  - Use mathematical engineering structures to design test values independently of the details of software or design artifacts, test automation, and test execution
- Key intellectual step: **test case design**
- Test case design can be the primary factor determining whether tests successfully find failures in software

Software Testing Activities



activity requires different skills, background knowledge, education, and training

1. Test Design

Human-based approach

- Design test values based on
  - Domain knowledge of the program
  - Human knowledge of testing
  - Knowledge of user interface
- Require almost no traditional CS degree
  - Background in the software domain is essential
  - Empirical background is very helpful
  - Logic background is very helpful

Criteria-based approach

- Design test values to satisfy coverage criteria
- Much of the work involves creating abstract models and manipulating them to design high-quality tests.
- The most technical job in software testing
- CS Degree, require knowledge of: Discrete math, Programming, and Testing
- Using people who are not qualified to design tests will result in ineffective tests

Often combined

Why Test Case Design Techniques?

Exhaustive testing (use of all possible inputs and conditions) is impractical

- Must use a subset of all possible test cases
- Must have high probability of detecting faults

Need processes that help us selecting test cases

- Different people – equal probability to detect faults

Effective testing – detect more faults

- Focus attention on specific types of faults
- Know you're testing the right thing

Efficient testing – detect faults with less effort

- Avoid duplication
- Systematic techniques are measurable and repeatable

A Challenge

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

- Which values for *a*, *b*, *c* should we test?
  assuming a, b, c, were 32-bit integers, we'd have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

Coverage Criteria

- Testers search a huge input space -- to find the fewest inputs that will reveal the most problems

How to search, when to stop

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

- Advantages of coverage criteria
  - Search the input space thoroughly
  - Not much overlap in the tests
  - Maximize the "bang for the buck"
  - Provide traceability from software artifacts to tests
  - Make regression testing easier
  - Provide a "stopping rule"
  - Can be well supported with **tools**

Test Criteria and Requirements

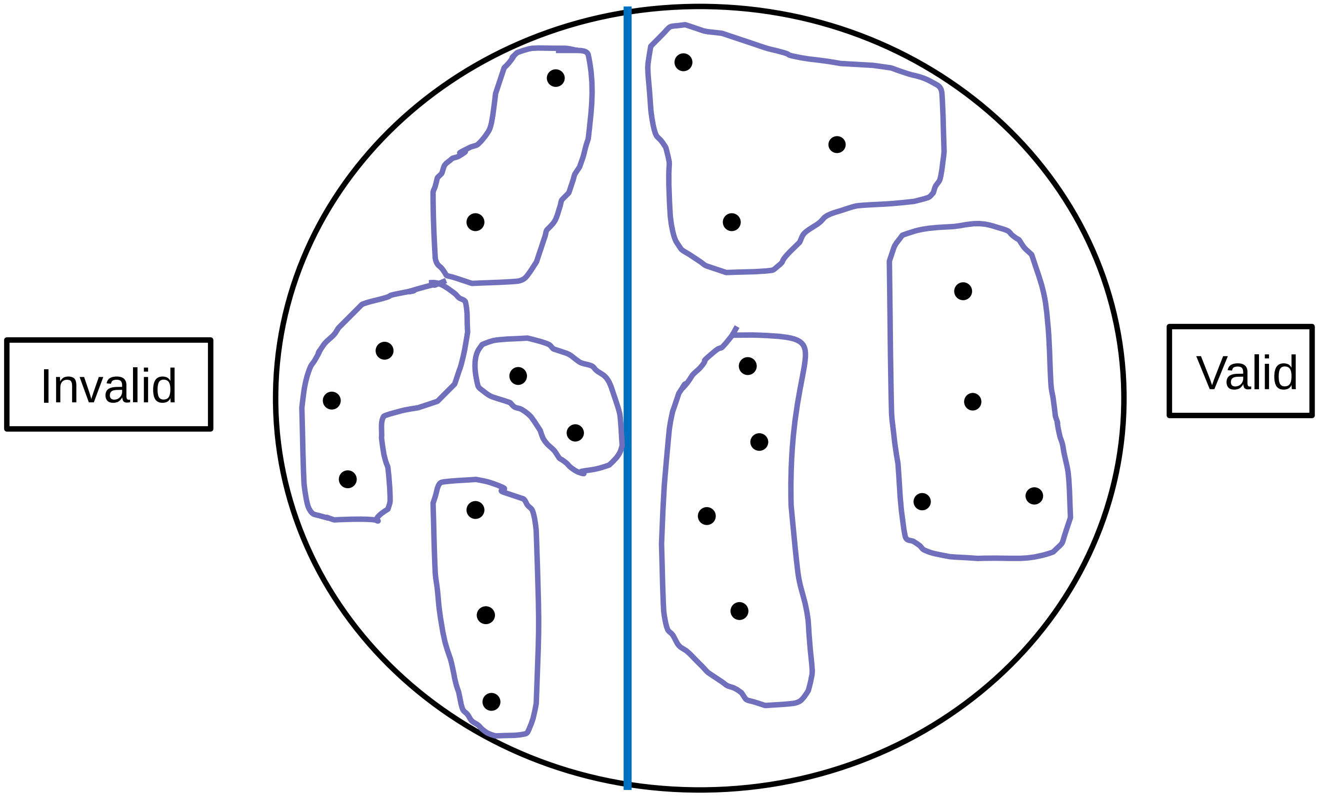
- Test criterion: A collection of rules and a process that define test requirements
  - Cover every statement
  - Cover every functional requirement
- Test requirements: Specific things that must be satisfied or covered during testing
  - Each statement might be a test requirement
  - Each functional requirement might be a test requirement

Many criteria have been defined. They can be categorized into 4 types of structures

1. Input domains (Input Space Partitioning)
2. Graphs
3. Logic expressions
4. Syntax descriptions (Grammar based)

1. Input Domain: Equivalence Partitioning (Example only)

- Partition valid and invalid test data into equivalence classes



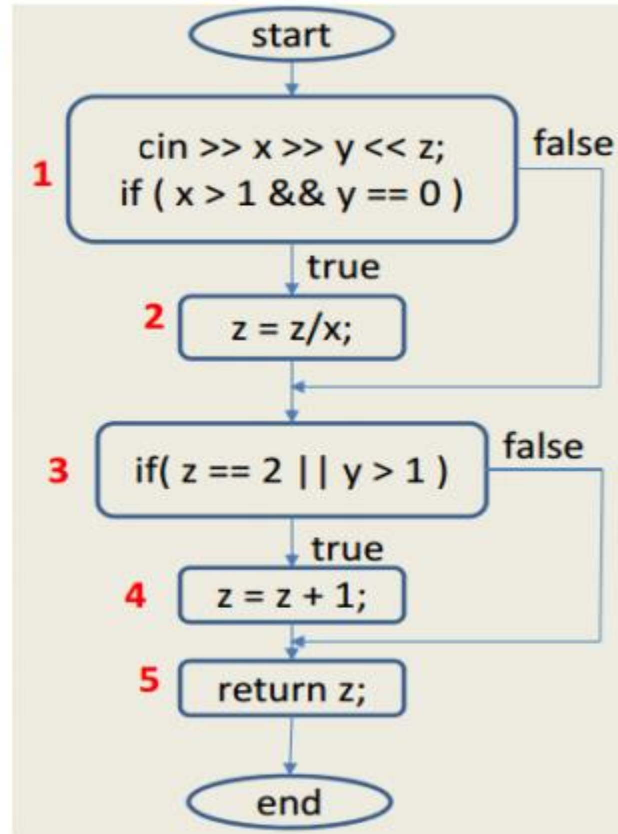
2. Graph (Example only)

- Control Flow Graph (CFG): Converted from a program

1) Draw CFG

Program

```
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;
```



3. Logic Expression (Example only)

- Logic Expression: Derived from a criteria i.e Statement Coverage

```
cin >> x >> y >> z;
if ( x > 1 && y == 0 )
    z = z / x;
if ( z == 2 || y > 1 )
    z = z + 1;
return z;
```

1) Draw CFG

2) Criteria: Do statement coverage

3) Path: 1, 2, 3, 4, and 5 (covered all statements)

Logical Expression

4) Predicate expression derived from path above:
(x>1&&y==0) && (z/x==2||y>1)

5) Generate test inputs T = { (x = 2, y = 0, z = 4) }

Characteristics of Good Tests

- Test one thing
  - Have accurate purpose
  - Traceable to requirement or design
- Clear and easy to understand
- Relatively small
- Independent
- Precise and concise
- Repeatable

Sample Test case

Test Cast ID

Purpose

Pre-conditions

Inputs

Expected Outputs

Post-conditions

Execution History

Date	Result	Version	Run By

Test cases: example only

<table>
<tr><td>TC#</td><td>Proj.Fun.-010</td><td>UC flow</td><td>2.2.2 main success scenario (Basic, alternative, exception flow name or function under test)</td></tr>
<tr><td>Objectives</td><td colspan="3">Try to use:
-Verify that (for TC with valid data)
-Attempt to (for TC with invalid data)</td></tr>
<tr><td>Preconditions</td><td colspan="2">Input</td><td>Expected Results</td></tr>
<tr><td>-The system displays…
-User has successfully…
-The system allows…
-The user has been authenticated…</td><td colspan="2">(For different conditions where applicable)
-The user selects…
-The user enters…</td><td>-Expected result may be copy-paste from Use Case but it depends on how the Use Case is written.</td></tr>
</table>

Source: http://extremesoftwaretesting.com/Testing/TCtemplate.html

2. Test Automation

- Embed test values into executable scripts
- Slightly less technical
- Require knowledge of programming
- Require very little theory
- Often involve observability and controllability issues
- 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
- Easy and trivial if the tests are well automated
- Requires basic computer skills
  - Interns
  - Employees with no technical background
- Can be boring for test designers
  - Asking qualified test designers to execute tests is a sure way to convince them to look for a development job
- 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 knowledge of
  - **Domain**
  - **Testing**
  - **User interfaces and psychology**
- Usually requires almost no traditional CS
  - Background in the **software domain** is essential
  - **Empirical background** is very helpful (biology, psychology, …)
  - **Logic background** is very helpful (law, philosophy, math, ...)

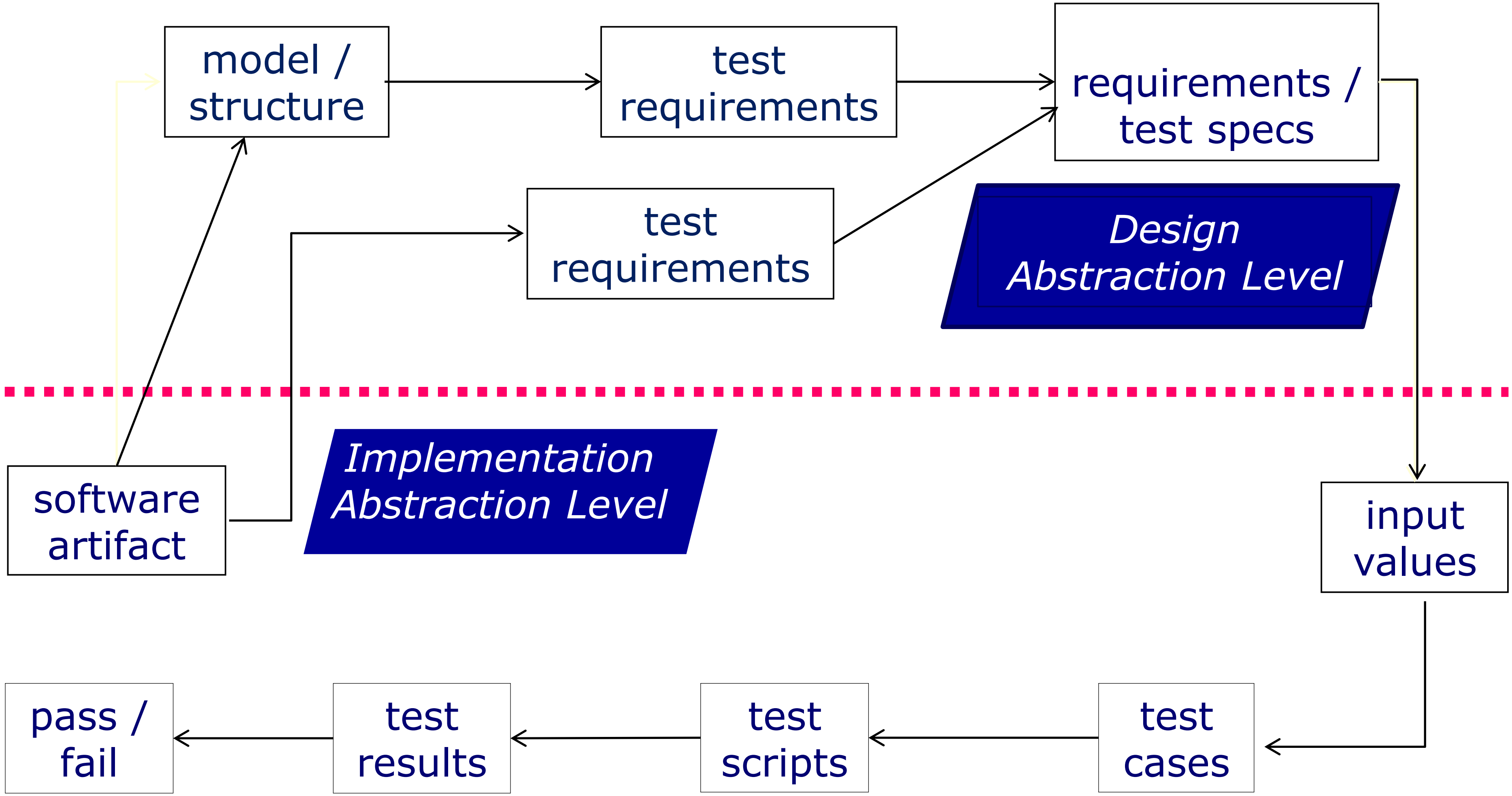
Other Activities

- **Test management**
  - Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed (i.e when to stop), …
- **Test maintenance**
  - Save tests for result as solve evolve (for metrics, auditing functions)
  - Requires cooperation of test designers and test automators
  - Partly policy and partly technical
- **Test documentation**
  - All parties participate
  - Each test must document “why” – criterion and test requirement satisfied or a rationale for human-designed test
  - Ensure traceability throughout the process
  - Keep documentation in the automated tests

Using MDTD in Practice

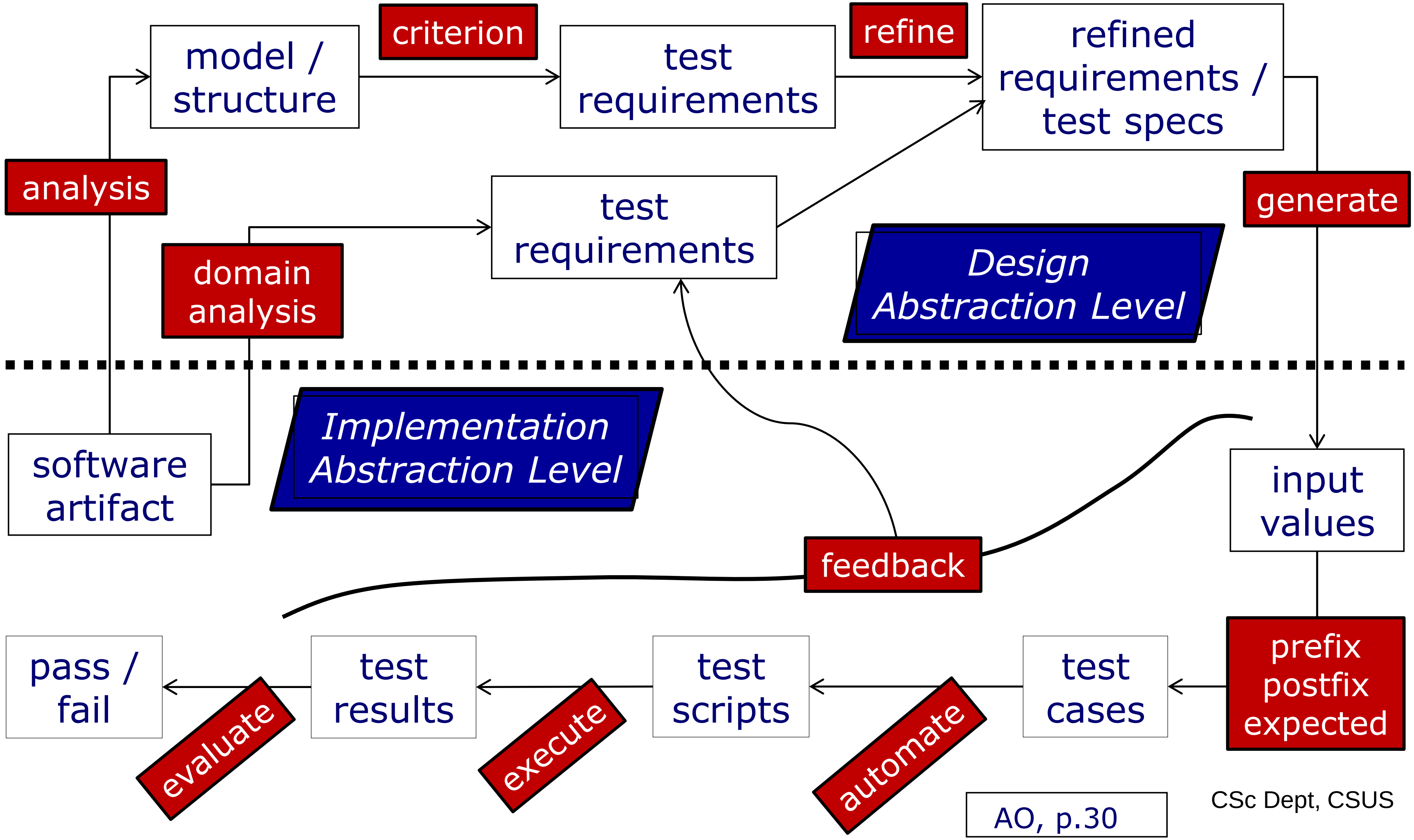
- This approach lets one test designer do the math
- Then traditional testers and programmers can do their part
  - Find values
  - Automate the tests
  - Run the tests
  - Evaluate the tests
- Test designers become technical experts
- Many test designers get involved in crowd testing

Model-Driven Test Design

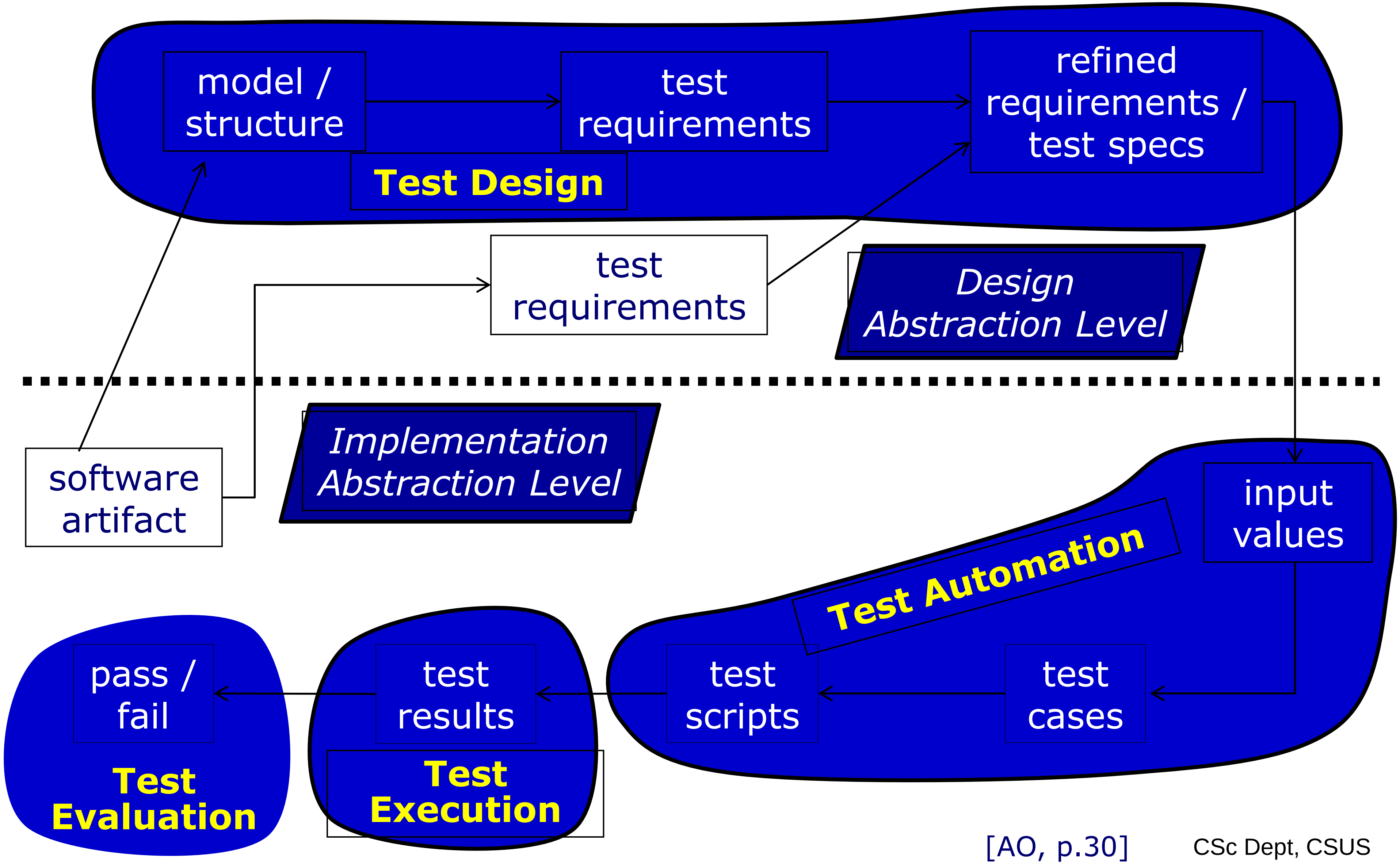


[AO, p.30]

Model-Driven Test Design - Steps



Model-Driven Test Design - Activities



[AO, p.30]

Wrap-up

- Discussing test design with criteria-based approach (focus)
- Testing activities
  - Design tests: model software + apply test coverage criteria
  - Automate tests
  - Execute tests
  - Evaluate tests
- Characteristics of good test cases
  - Sample a test case
- MDTD (Model-Driven Test Design)
  - Break testing into smaller tasks.
  - Two level of abstraction: Test Design and Implementation

What's Next?

Wrap-up (Cont)

What's Next?

- Putting testing first
- Black box testing
- Test Automation (Junit)