Introduction to Software Testing

(*2nd edition*)

Chapter 2

Model-Driven Test Design

Paul Ammann & Jeff Offutt

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

Updated August 2014

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?

- Like 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

Not their absence

Testing & Debugging

- Testing : Evaluating software by observing its execution
- Test Failure : Execution of a test that results in a software failure
- Debugging : The process of finding a fault given a failure

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

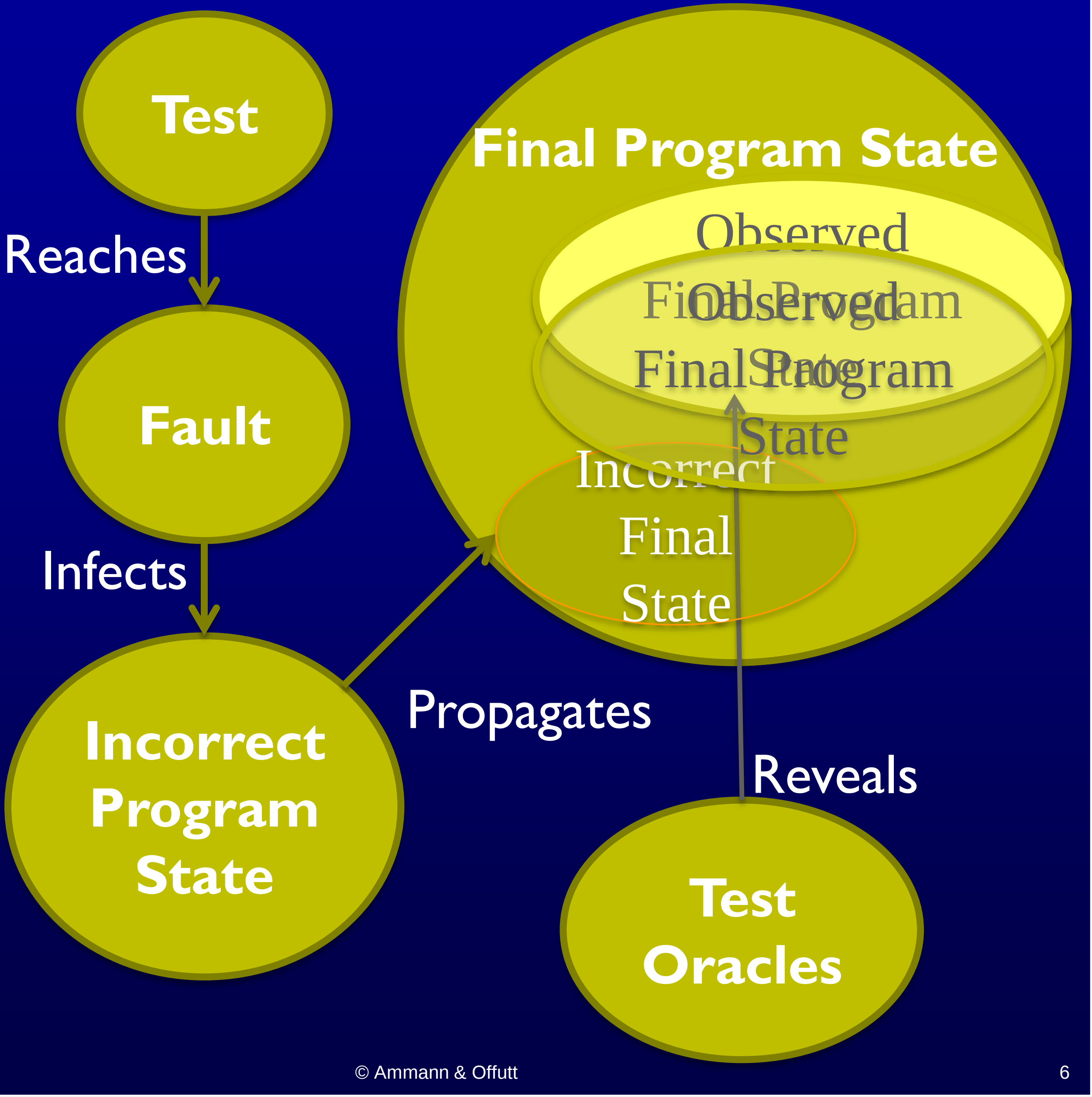
Fault & Failure Model (RIPR)

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

RIPR Model

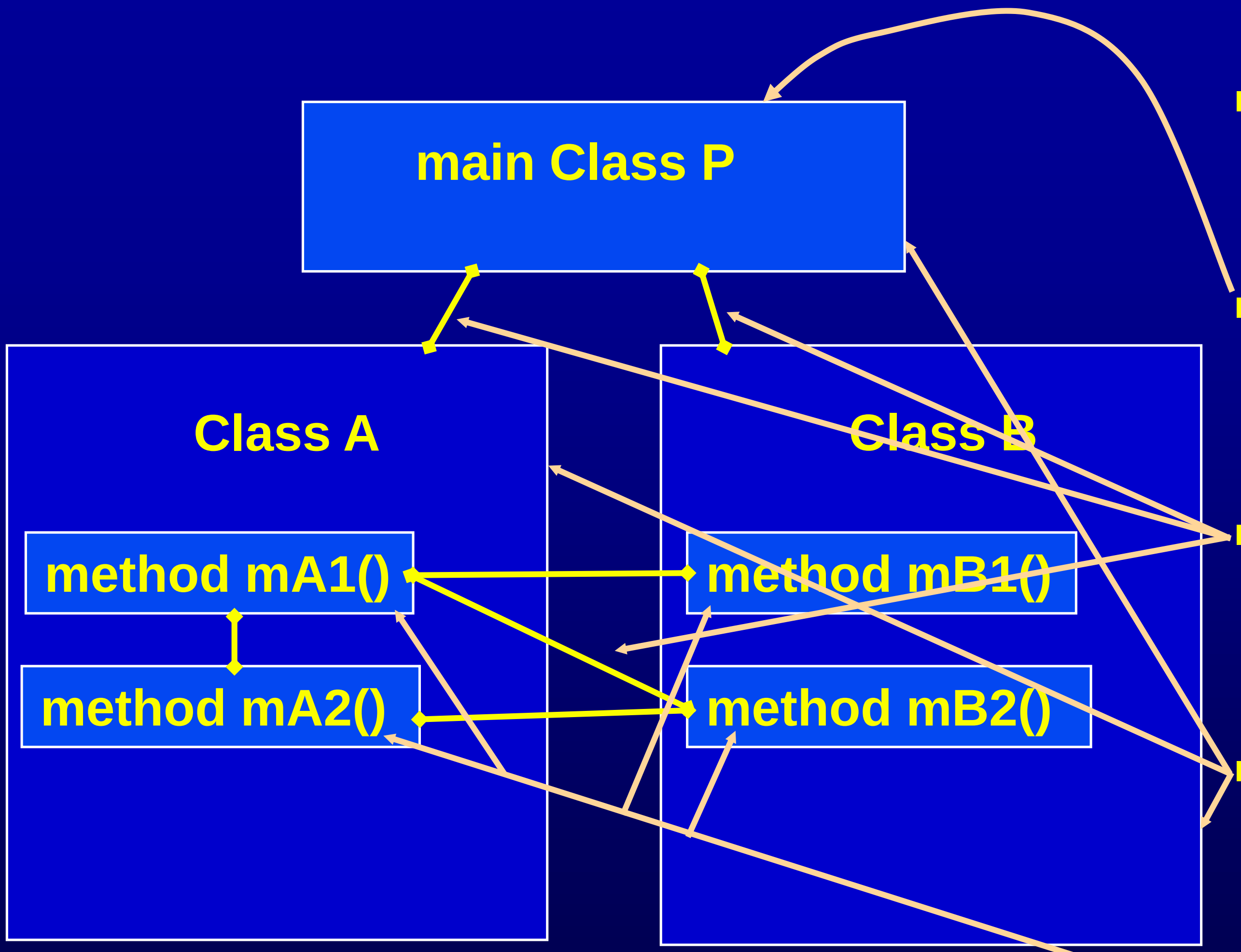
- Reachability
- Infection
- Propagation
- Revealability



Software Testing Activities (2.2)

- Test Engineer : An IT professional who is in charge of one or more technical test activities
  - Designing test inputs
  - Producing test values
  - Running test scripts
  - Analyzing results
  - Reporting results to developers and managers
- Test Manager : In charge of one or more test engineers
  - Sets test policies and processes
  - Interacts with other managers on the project
  - Otherwise supports the engineers

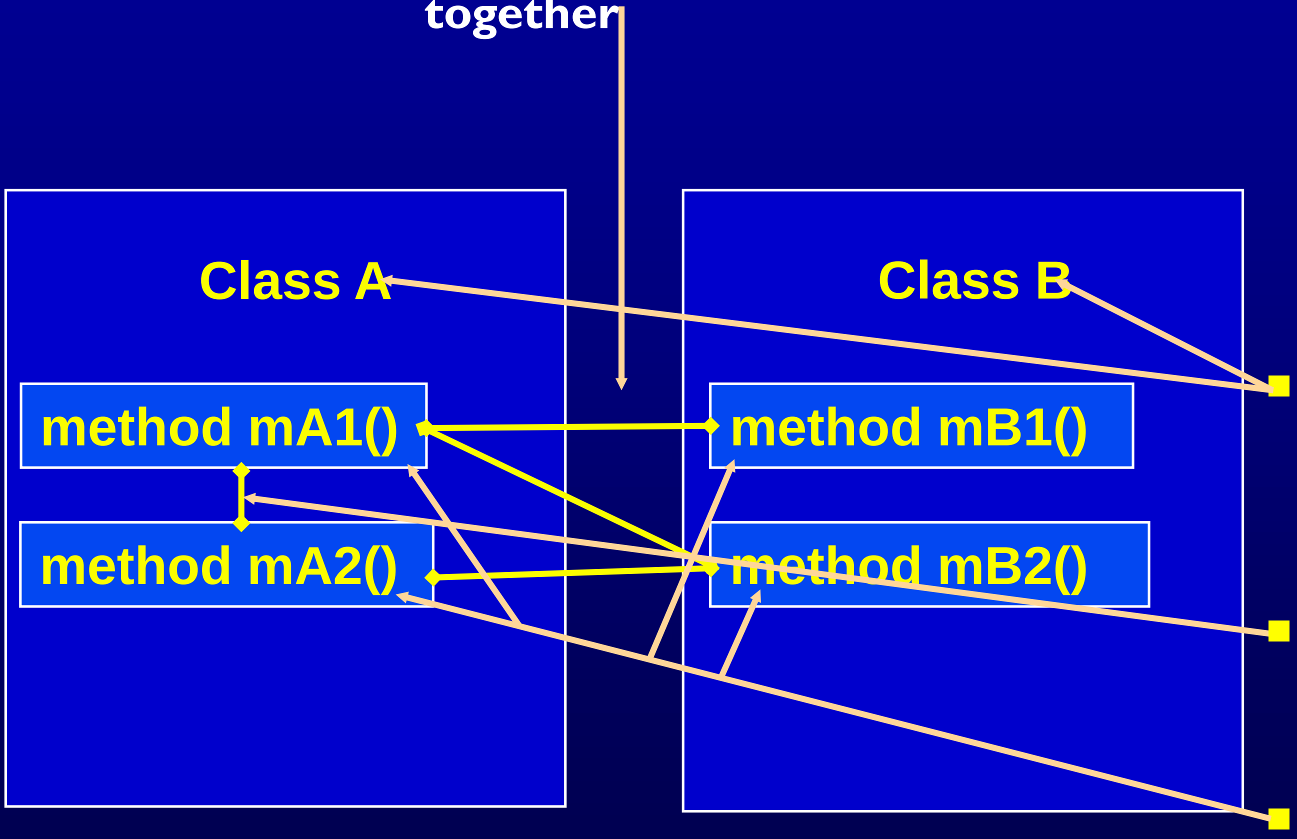
Traditional Testing Levels (2.3)



- **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

This view obscures underlying similarities

Object-Oriented Testing Levels



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

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 (10^9) possible values
  - Over 80 octillion (10^{27}) 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" (cost-benefit)
- 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 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

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 : 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)

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 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 development 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 <u>wastes</u> 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

3. Test Execution

Run tests on the software and record the results

- This is easy – and trivial if the tests are well automated
- Requires basic computer skills
  - Interns
  - Employees with no technical background
- Asking qualified test designers to execute tests is a sure way to convince them to look for a development job
- If, for example, GUI tests are not well 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 knowledge of :
  - Domain
  - Testing
  - User interfaces and psychology
- Usually 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

Other Activities

- Test management : Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed, …
- Test maintenance : Save tests for reuse as software evolves
  - Requires cooperation of test designers and automators
  - Deciding when to trim the test suite is partly policy and partly technical – and in general, very hard !
  - Tests should be put in configuration control
- Test documentation : All parties participate
  - Each test must document "why" – criterion and test requirement satisfied or a rationale for human-designed tests
  - Ensure traceability throughout the process
  - Keep documentation in the automated tests

Organizing the Team

- A mature test organization needs only one test designer to work with several test automators, executors and evaluators
- Improved automation will reduce the number of test executors
  - Theoretically to zero … but not in practice
- Putting the wrong people on the wrong tasks leads to inefficiency, low job satisfaction and low job performance
  - A qualified test designer will be bored with other tasks and look for a job in development
  - A qualified test evaluator will not understand the benefits of test criteria
- Test evaluators have the domain knowledge, so they must be free to add tests that "blind" engineering processes will not think of
- The four test activities are quite different

Many test teams use the same people for ALL FOUR activities !!

Applying Test Activities

To use our people effectively

and to test efficiently

we need a process that

lets test designers

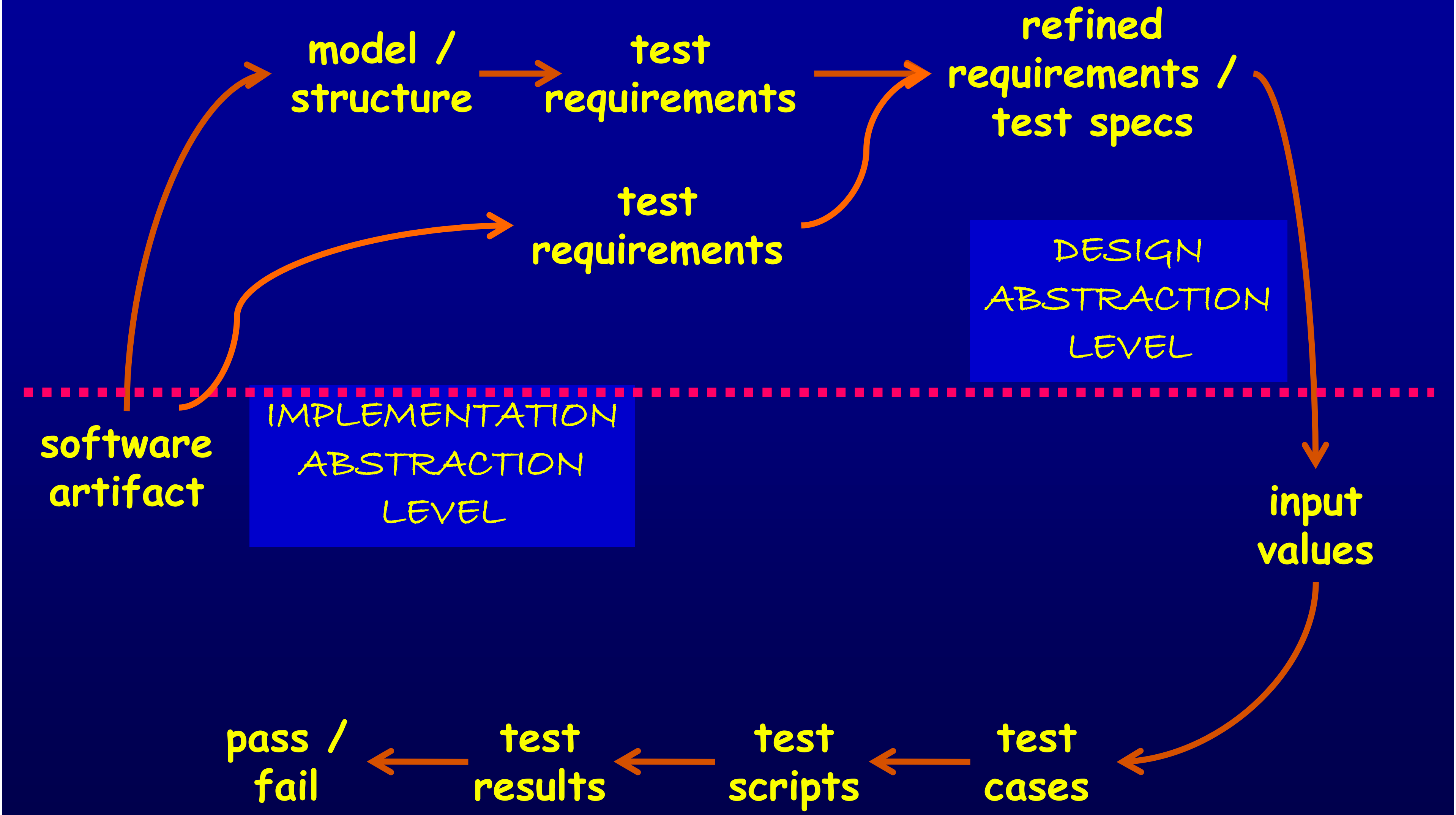
raise their level of abstraction

Using MDTD in Practice

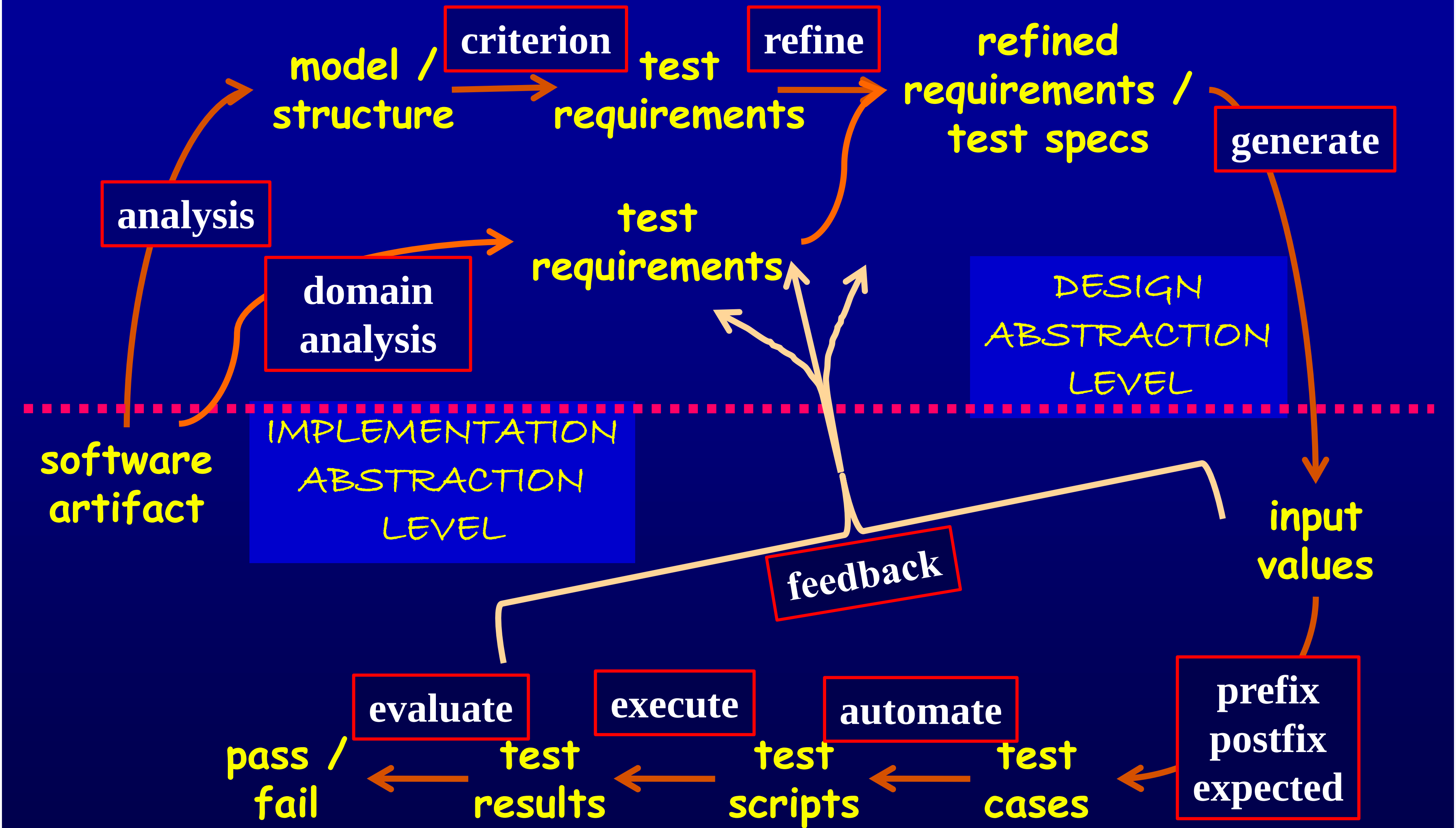
- This approach lets one test designer do the math
- Then traditional testers and programmers can do their parts
  - Find values
  - Automate the tests
  - Run the tests
  - Evaluate the tests
- Just like in traditional engineering … an engineer constructs models with calculus, then gives direction to carpenters, electricians, technicians, …

Test designers become technical experts

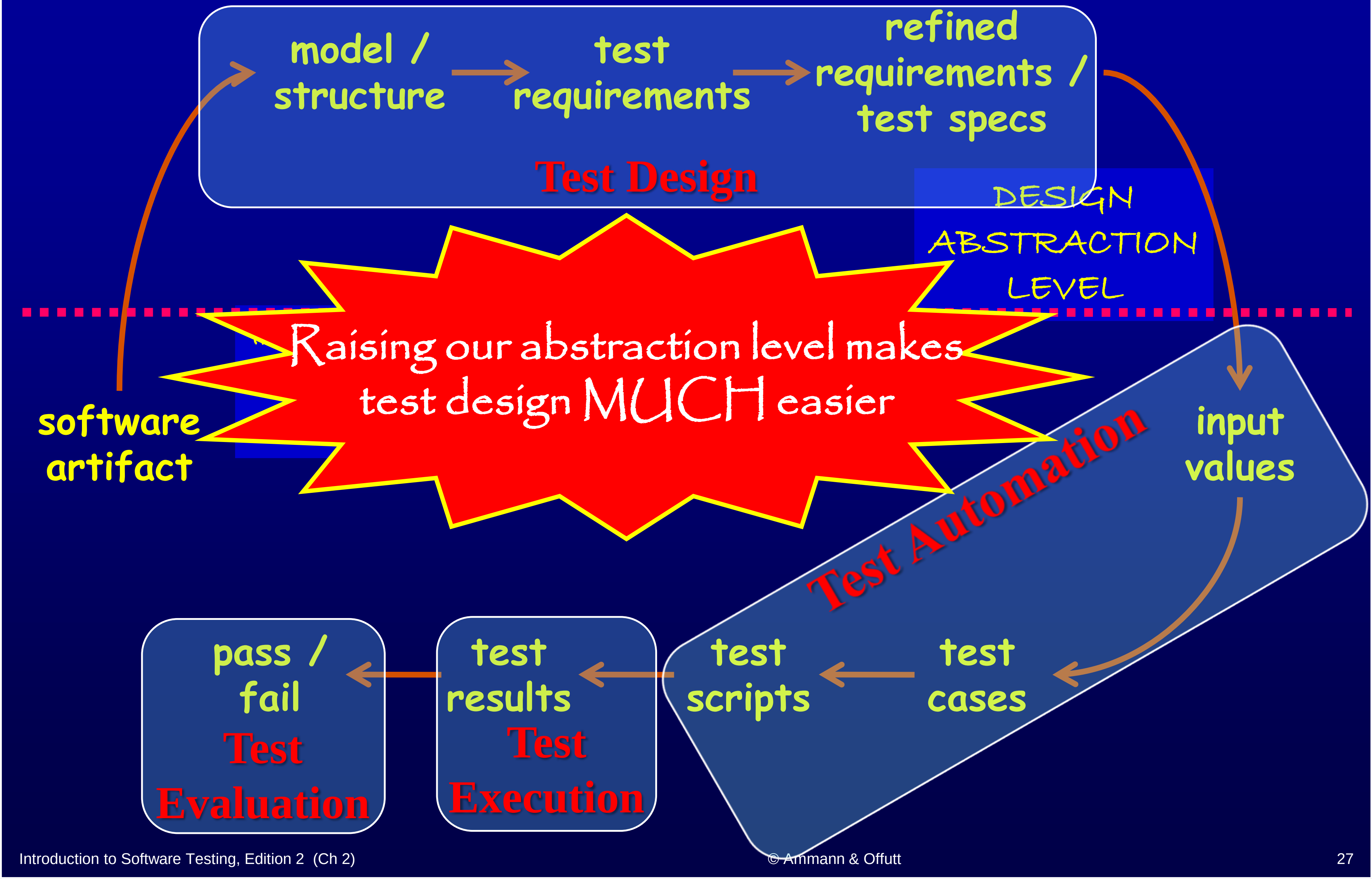
Model-Driven Test Design



Model-Driven Test Design – Steps



Model-Driven Test Design–Activities



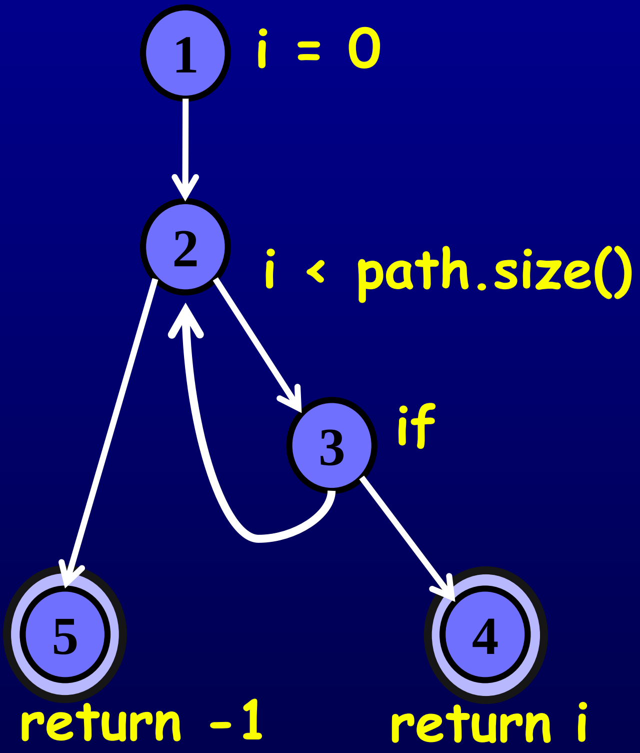
Raising our abstraction level makes test design MUCH easier

Small Illustrative 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;
}
```

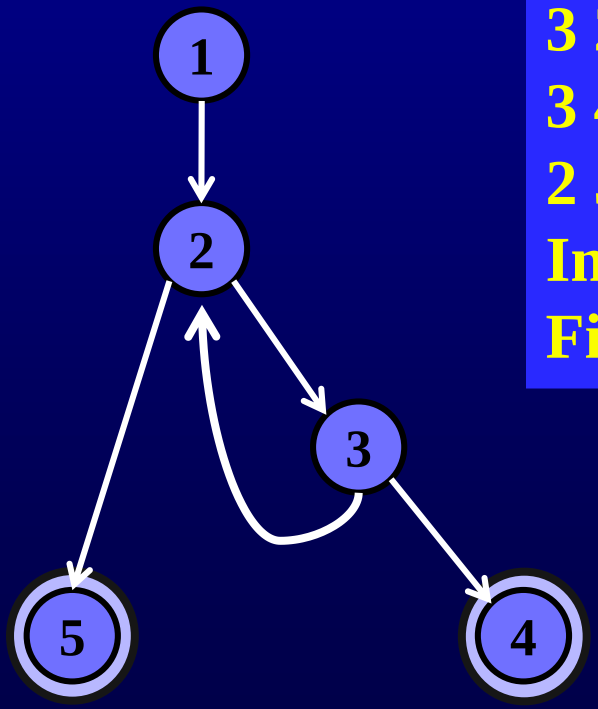
Control Flow Graph



Example (2)

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

Graph
Abstract 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]

Find values …

Types of Activities in the Book

Most of this book is about test design

Other activities are well covered elsewhere