CS 5154

Criteria-Based Test Design

Owolabi Legunsen

The following are modified versions of the publicly-available slides for Chapters 2 and 5 in the Ammann and Offutt Book, “Introduction to Software Testing”
(http://www.cs.gmu.edu/~offutt/softwaretest)

In Today's Class (Hopefully…)

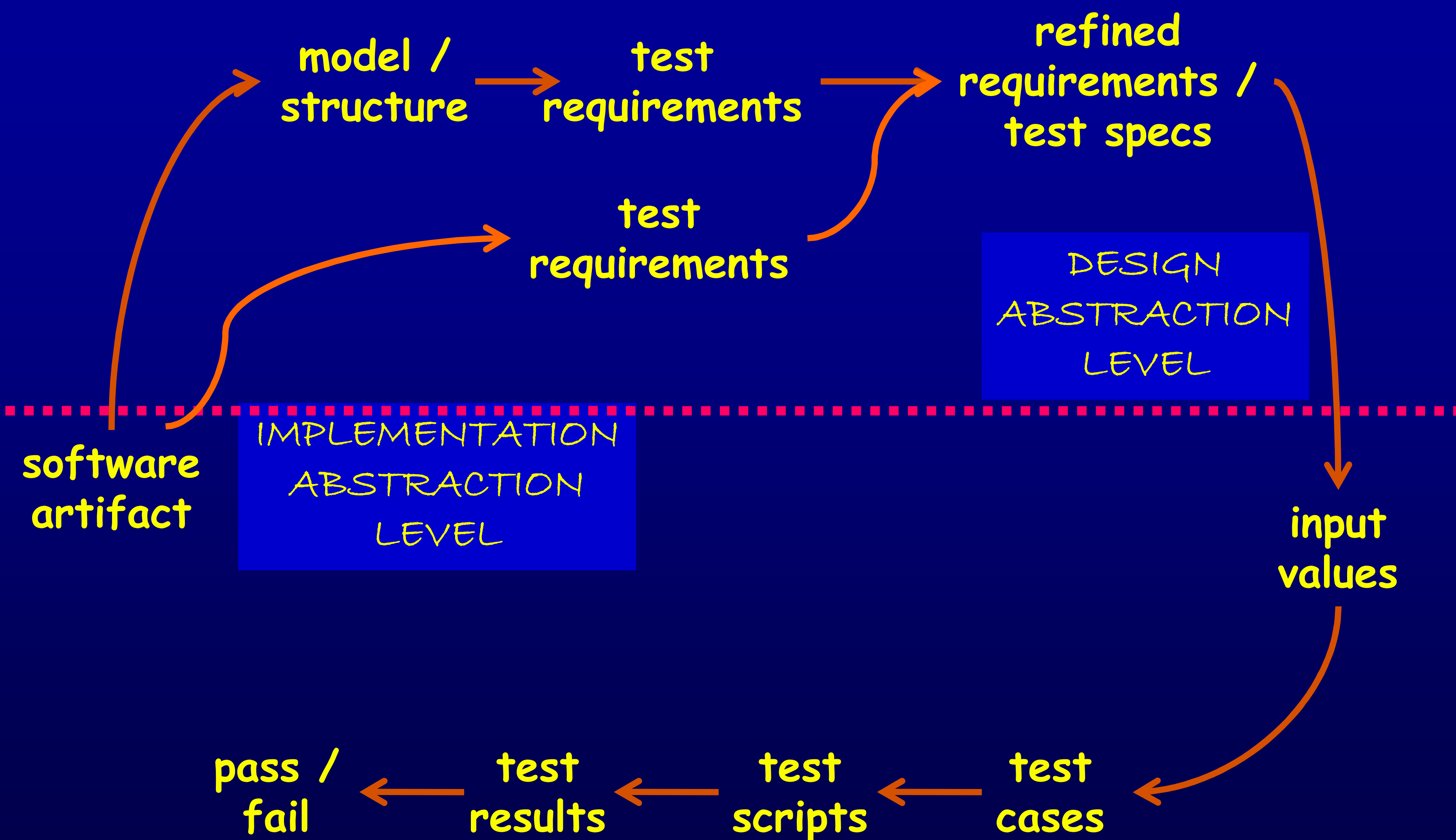
- Introduction to Model-Driven Test Design

- Hands-on Demo (if we have time)
  - The Maven Build System
  - Measuring Coverage

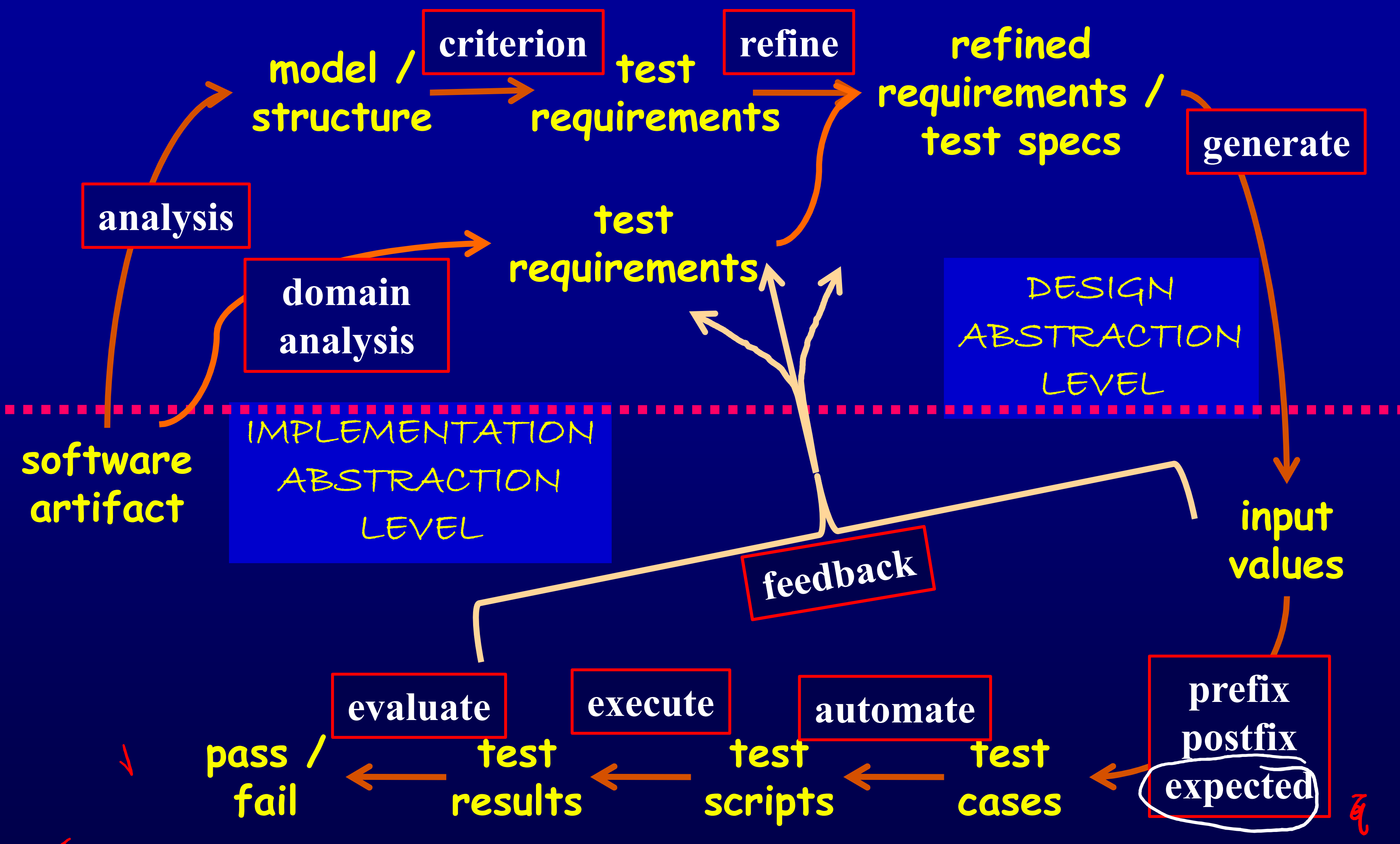
Changing Notions of Testing

- Old view: focus on testing at each software development phase as being very different from other phases
  - Unit, module, integration, system, …
- This class: think in terms of structures and criteria
  - input space, graphs, logical expressions, syntax
- Test design is largely the same at each phase
  - Creating the model is different
  - Choosing values and automating the tests is different

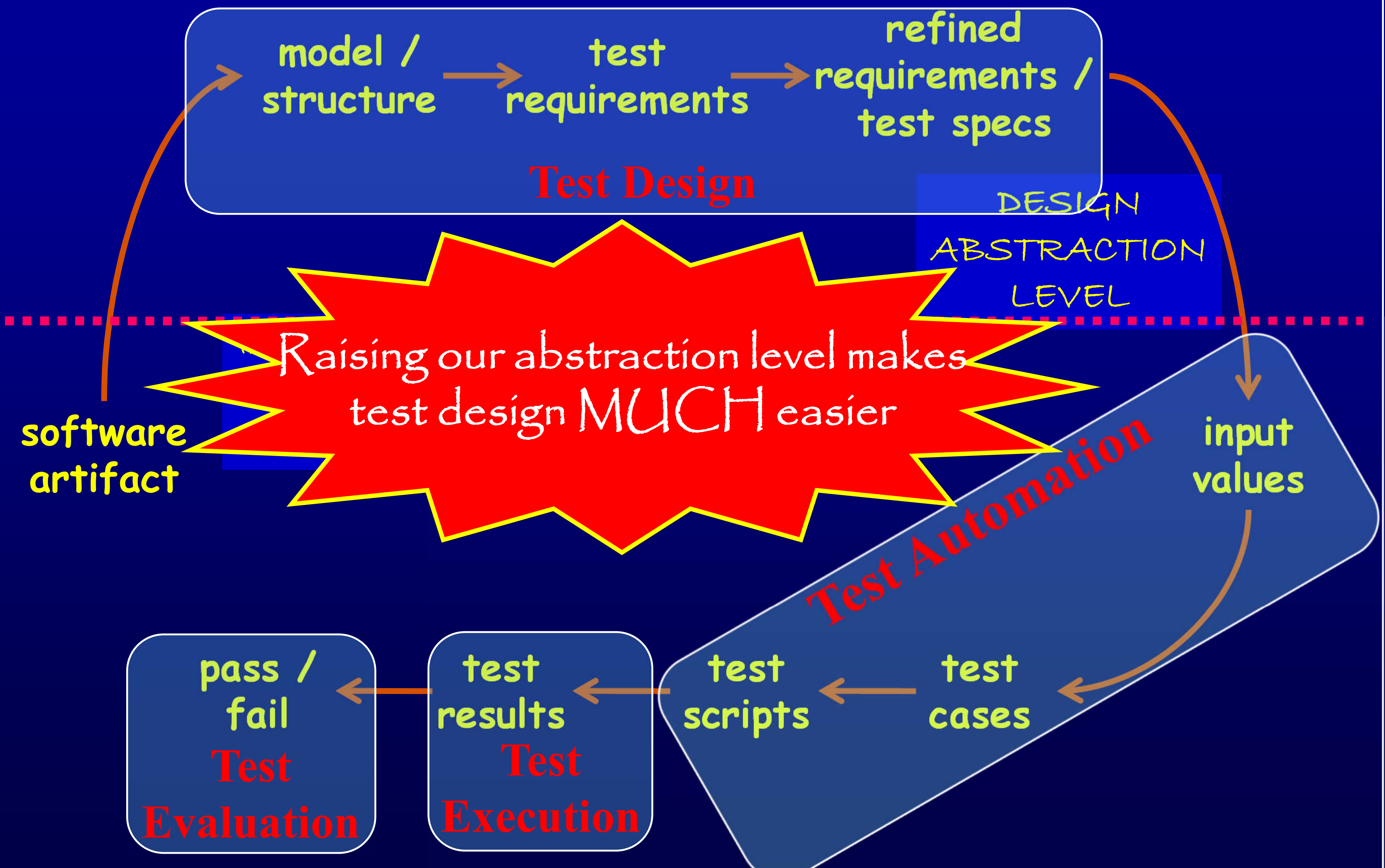
Model-Driven Test Design



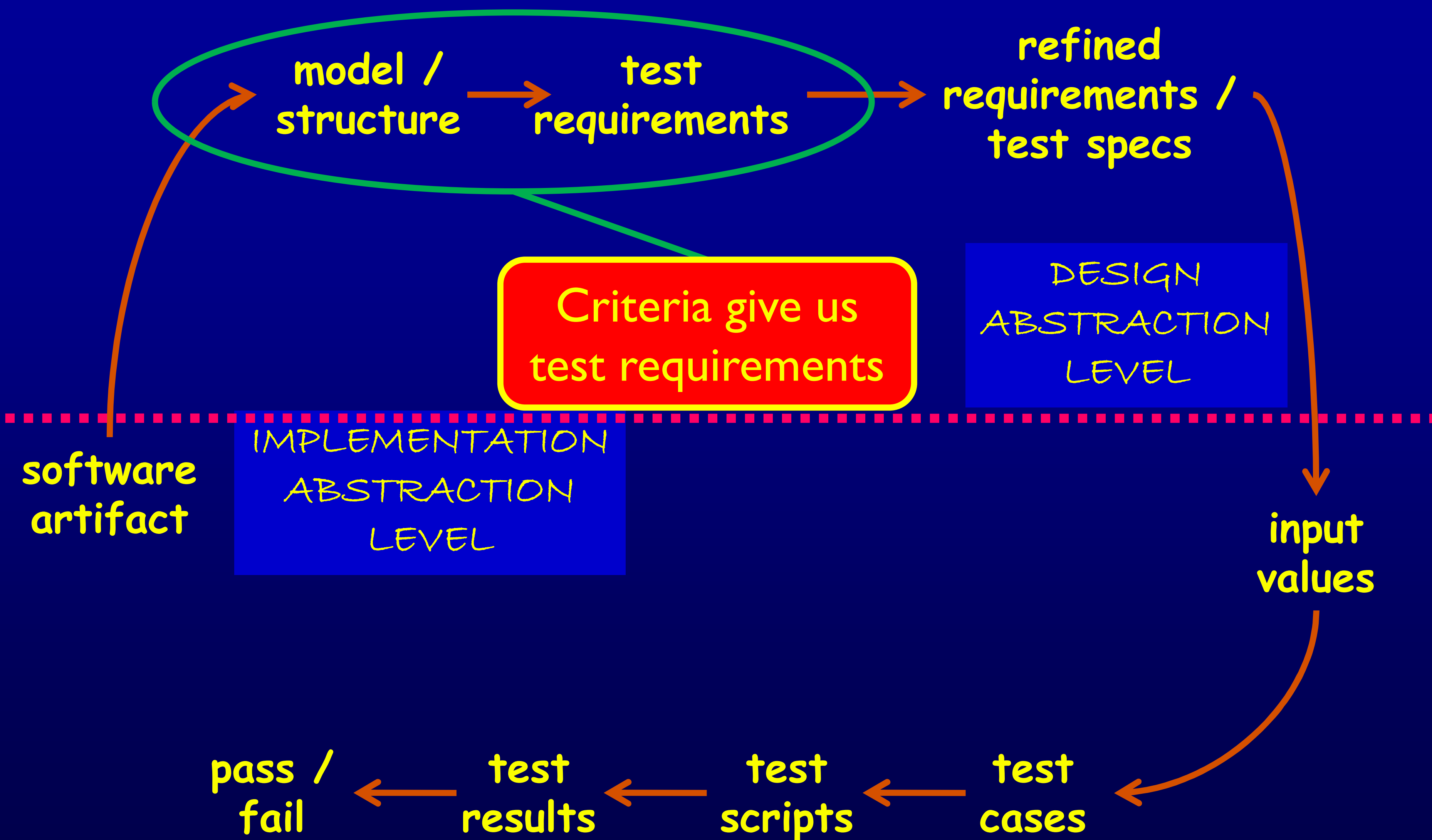
Model-Driven Test Design – Steps



Model-Driven Test Design–Activities



Criteria-Based Test Design



Test design concepts

A tester's job is simple : Define a model of the software, then find ways to cover it

- Test Requirements : A specific element of a software artifact that a test case must satisfy or cover

o ✓ statement coverage : lines of code
× branch coverage : condition

- Coverage Criterion : A rule or collection of rules that impose test requirements on a test set

But, many coverage criteria exist

All Combinations Coverage
Each choice Coverage
Pair-Wise Coverage
T-Wise Coverage
Base Choice Coverage
Multiple Base Choice Coverage
Node Coverage
Edge Coverage
Edge-pair Coverage
Prime Path Coverage
Simple Round Trip Co
Complete Round T erage
Complete Pa age
Specified verage
All- erage
All-U Coverage

All-du-Paths Coverage
Predicate Coverage
Combinatorial Coverage
General Active Claus ge
Correlated Acti Coverage
Restricted A use Coverage
Gener Clause Coverage
R nactive Clause Coverage
nt Coverage
rminal Symbol Coverage
Production Coverage
Mutation Coverage
Mutation Operator Coverage
Mutation Production Coverage
Strong Mutation Coverage
Weak Mutation Coverage

Multiple Unique True Points Coverage
Corresponding Unique True Points and Near False Point Pair Coverage
Multiple Near False Point Coverage

These are only a subset of those in the Book!

Organized approach to criteria

- Researchers defined many more criteria
- Some criteria in the literature are redundant with respect to one another
- The view in this book (and in this course): all criteria are defined on just four types of structures
  - Input Domain
  - Graph Representations of Software
  - Logic expressions in Software
  - Syntax

How to obtain these structures?

- The structures can be extracted from lots of artifacts
  - Graphs can be extracted from UML use cases, finite state machines, source code, …
  - Logical expressions can be extracted from decisions in program source, guards on transitions, conditionals in use cases, …

- MDTD ≠ "*model-based testing*," (MBT) which derives tests from formal models of the system under test
  - MBT models usually describe part of the behavior
  - The source code is explicitly *not* considered a model in MBT

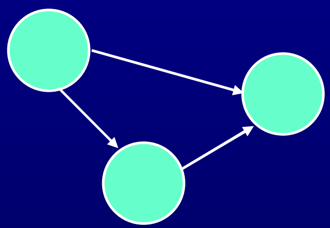
Criteria Based on Structures

Structures : Four ways to model software

1. Input Domain Characterization (sets)

   A: {0, 1, >1}
   B: {600, 700, 800}
   C: {cs, ece, is, sds}

2. Graphs



3. Logical Expressions

   (not X or not Y) and A and B

4. Syntactic Structures (grammars)

```
if (x > y)
   z = x - y;
else
  z = 2 * x;
```

Example : Jellybean Coverage

Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot

Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

- Quiz: What coverage criteria would be appropriate?

Example : Jellybean Coverage

Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot

Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

- Possible coverage criteria :
  1. Taste one jellybean of **each flavor**
     - Deciding if yellow is Lemon or Apricot is a controllability problem
  2. Taste one jellybean of **each color**

Coverage

Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

- Infeasible test requirements : test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Example: Dead code
  - Detecting infeasible test requirements is undecidable for most test criteria
- Thus, 100% coverage is impossible in practice

More Jellybeans

T1 = { three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots }

- Does test set T1 satisfy the flavor criterion ?

T2 = { One Lemon, two Pistachios, one Pear, three Tangerines }

- Does test set T2 satisfy the flavor criterion ?
- Does test set T2 satisfy the color criterion ?

Coverage Level

The ratio of the number of test requirements satisfied by *T* to the size of *TR*

- T2 on the previous slide satisfies 4 of 6 test requirements

Two Ways to Use Test Criteria

1. Directly generate test values to satisfy the criterion
   - Often assumed by the research community
   - Most obvious way to use criteria
   - Very hard without automated tools

2. Generate test values and measure against the criterion
   - Usually favored by industry
   - Sometimes misleading
   - If tests have <100% coverage, what does that mean?

Test criteria are sometimes called metrics

Generators and Recognizers

- Generator : A procedure that automatically generates values to satisfy a criterion
- Recognizer : A procedure that decides whether a given set of test values satisfies a criterion

- Both problems are undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion

- Coverage analysis tools are quite plentiful

All criteria are NOT born equal

- Quiz: What is the smallest code and its test values that you can come up with such that the test values satisfy 100% statement coverage but miss a bug in the code?

All criteria are NOT born equal

- Code and test values that satisfy 100% statement coverage but miss a bug in the code?

short-circuit

```
int stringFactor(String i, int n) {
if (i != null || n !=0)
  return i.length()/n;
else
  return -1; }
```

√ ["", 0] 0/0

["a", -1]

√ [null, 1]

√ ["a", 0]

- Test values: ["happy", 2], [null, 0]
- "stronger" criteria that can reveal this bug

Comparing criteria: subsumption

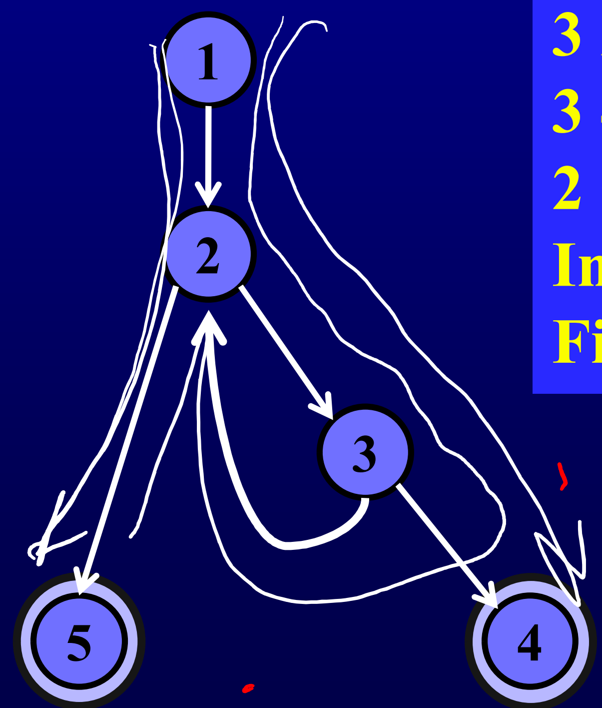
- Criteria Subsumption : Test criterion *C1* subsumes *C2* iff every set of test cases that satisfies *C1* ? also satisfies *C2* ?
- Must be true for every set of test cases
- *Examples* :
  - The flavor criterion on jellybeans subsumes the color criterion … if we taste every flavor, we taste one of every color
  - If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement

We've seen subsumption before

- Many more
- multiple ending point
- easier

[1,2] + [2,3] ⇒ [1,2,3]

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]

Criteria-Based Test Design: Pros

- Criteria maximize the "bang for the buck"
  - Fewer tests that are more effective at finding faults
  - Comprehensive test set with minimal overlap
- Traceability from software artifacts to tests
  - The "why" for each test is answered
  - Built-in support for regression testing
- A "stopping rule" for testing—advance knowledge of how many tests are needed
- Natural to automate

Criteria-Based Test Design: Cons

- Blindly aiming to satisfy coverage criteria can make it easy to ignore domain knowledge
  - Domain knowledge: very useful for deriving tests that find bugs
  - Criteria-Based Test Design should be complemented with human-based test design

Characteristics of a Good Coverage Criterion

1. It should be fairly easy to compute test requirements automatically
2. It should be efficient to generate test values
3. The resulting tests should reveal as many faults as possible

- Subsumption is only a rough approximation of fault revealing capability
- Researchers still need to gives us more data on how to compare coverage criteria

Test Coverage Criteria

- Software testing is expensive and labor-intensive
- Coverage criteria help choose which test inputs to use
- More likely that the tester will find problems
- More assurance that software has high quality & reliability
- A goal or stopping rule for testing
- Criteria makes testing more efficient and effective

How do we start applying these ideas in practice?

Steps to improving adoption?

- Testers need more and better software tools
- Testers need to adopt practices and techniques that lead to more efficient and effective testing
  – More education
- Testing & QA teams need more technical expertise
  – Developer expertise has been increasing dramatically
- CS5154 will help you to start taking these steps

Four Roadblocks to Adoption

1. Lack of test education
   - Microsoft and Google say half their engineers are testers, programmers test half the time
   - Number of UG CS programs in US that require testing ? **0**
   - Number of MS CS programs in US that require testing ? **0**
   - Number of UG testing classes in the US ? **~50**
2. Necessity to change process
   - Adoption of many test techniques and tools require changes in development process
   - This is expensive for most software companies
3. Usability of tools
   - Many testing tools require the user to know the underlying theory to use them
   - Do we need to know how an internal combustion engine works to drive ?
   - Do we need to understand parsing and code generation to use a compiler ?
4. Need for better tools
   - Most test tools don't do much – but most users do not realize they could be better
   - Few tools solve the key technical problem – generating test values automatically

Summary: criteria-based design

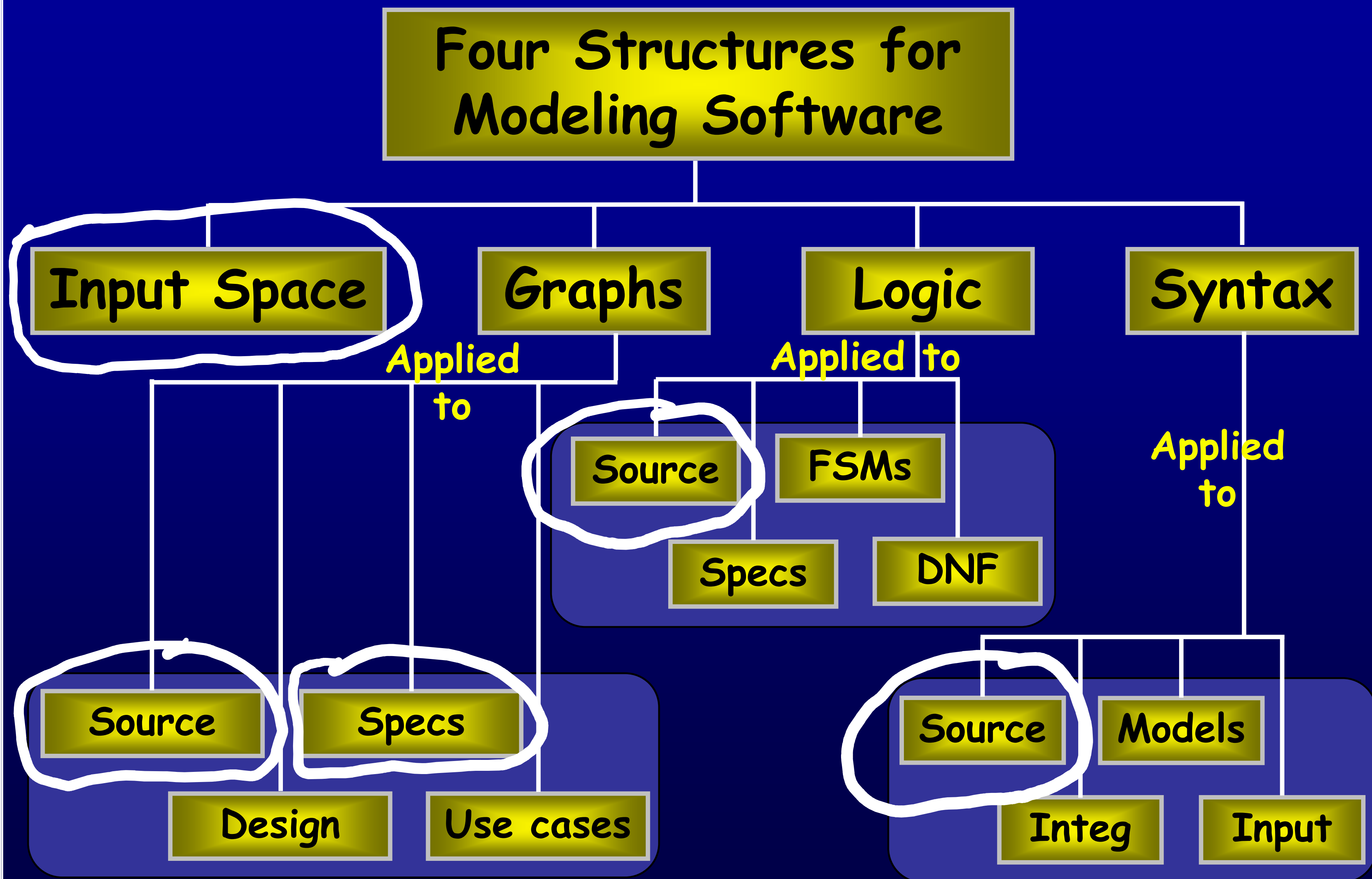
- Many companies still use "monkey testing"
  - A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  - No automation
  - Minimal training required
- Some companies automate human-designed tests
- But companies that use both automation and criteria-based testing

Save money

Find more faults

Build better software

Structures for Criteria-Based Testing



Ideas that we learned so far…

1. Why do we test – to reduce the risk of using software
   - Faults, failures, the RIPR model
   - Test process maturity levels – level 4 is a mental discipline that improves the quality of the software
2. Model-Driven Test Design
   - Four types of test activities – test design, automation, execution and evaluation
3. Test Automation
   - Testability, observability and controllability, test automation frameworks
4. Criteria-based test design
   - Four structures – test requirements and criteria

Earlier and better testing empowers test managers

Next Class

- Get started on Input Space Partitioning
- (Maybe) Hands-on demo on measuring coverage
  - command line
  - Maven