# Model-Driven Test Design

Isaac Griffith

CS 4422 and CS 5599
Department of Computer Science
Idaho State University

# Outcomes

At the end of Today's Lecture you will be able to:

- Understand the complexity of software testing
- Understand the notion of software correctness
- Understand traditional testing models

# Inspiration

"More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded – indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest." – Boris Beizer

# 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

# In-Class Exercise

## Group Discussion: **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

**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)

## Four conditions necessary for a failure to be observed

❶ **Reachability**: The location or locations in the program that contain the fault must be reached.

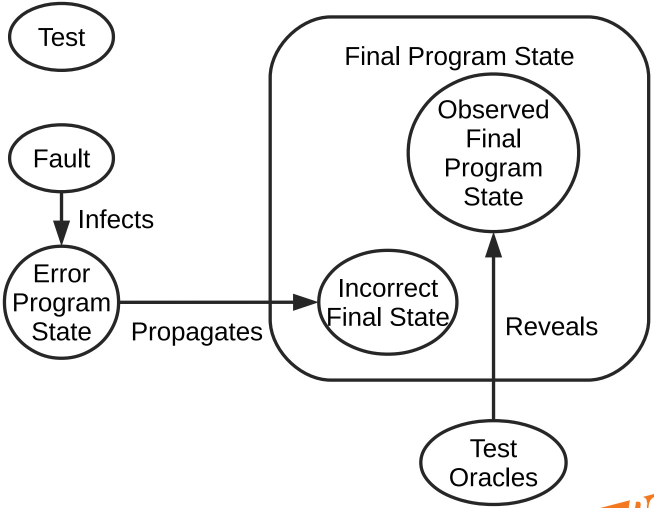
❷ **Infection**: The state of the program must be incorrect.

❸ **Propagation**: The infected state must cause some output or final state of the program to be incorrect.

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

# RIPR Model

- Reachability
- Infection
- Propagation
- Revelability



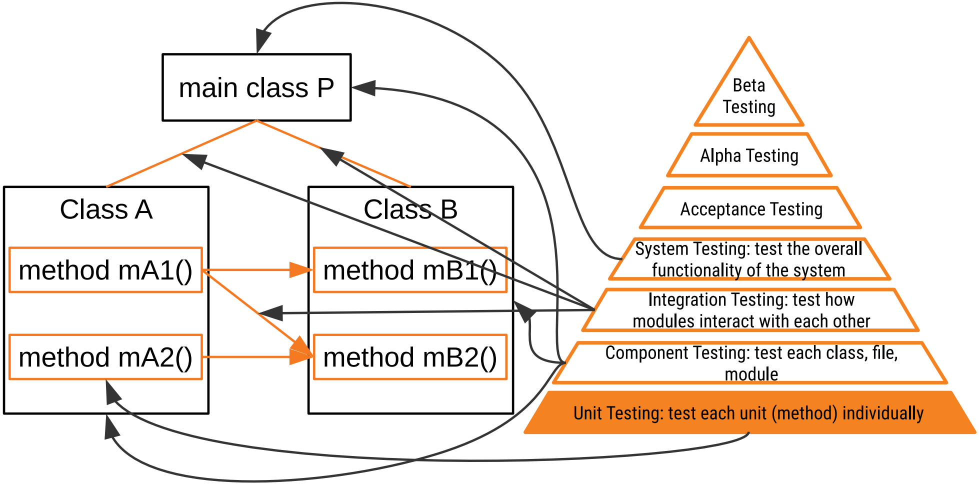
# In-Class Exercise

## Group Discussion: **Test Oracles**

- Have you written many automated tests?
- How did you decide what assertions to write?
- Do you think you ever checked the wrong part of the state?

You have five minutes

# Traditional Testing Levels


main class P
Class A
method mA1()
method mA2()
Class B
method mB1()
method mB2()
Beta Testing
Alpha Testing
Acceptance Testing
System Testing: test the overall functionality of the system
Integration Testing: test how modules interact with each other
Component Testing: test each class, file, module
Unit Testing: test each unit (method) individually

# Coverage Criteria

- 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
  - 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**

# 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 …**

❶ Input domains

❷ Graphs

❸ Logic expressions

❹ 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 questions is:**
**from what abstraction level do we derive tests?**

# Model-Driven Test Design

- **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**
     a. **Criteria-based**
     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 all aspects of SE

**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.

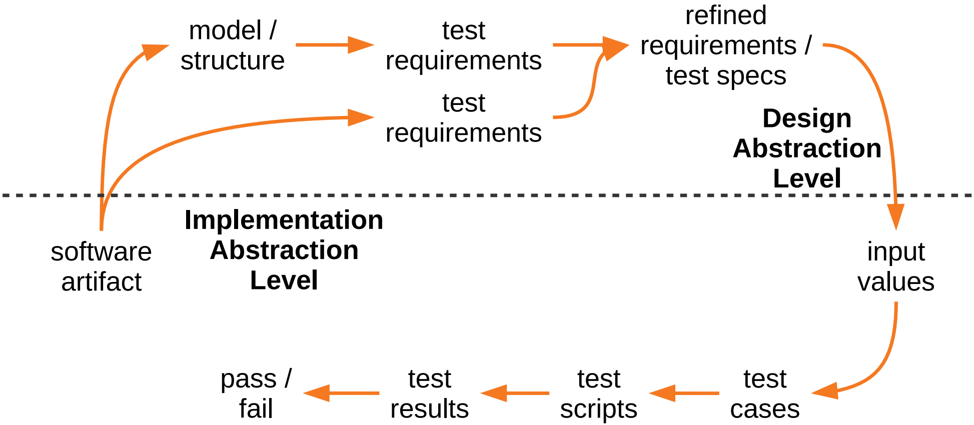
- This is much **harder** than it may seem to developers
- Criteria-based approaches can be blind to special situations
- Requires **knowledge** of:
  - Domain
  - Testing
  - User Interfaces
- Requires almost **no traditional CS**
  - A background in the **domain** of the software is essential
  - An **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 / structure

test requirements

test requirements

refined requirements / test specs

**Design Abstraction Level**

software artifact

**Implementation Abstraction Level**

input values

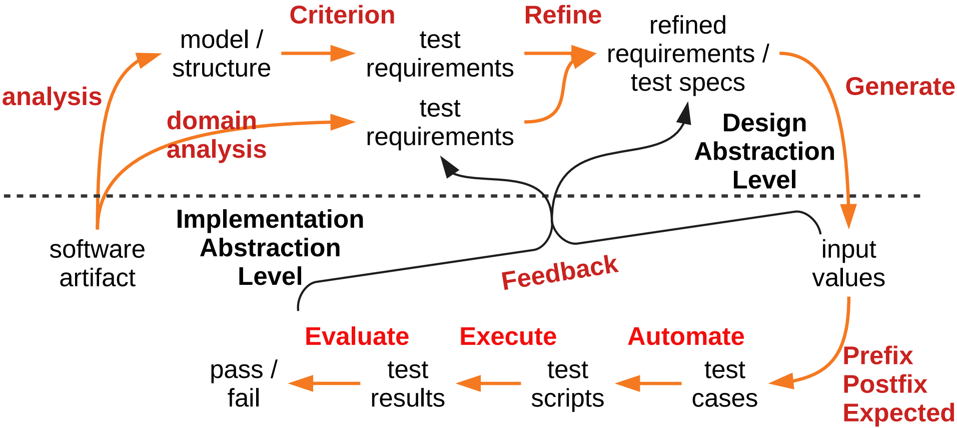
test cases

test scripts

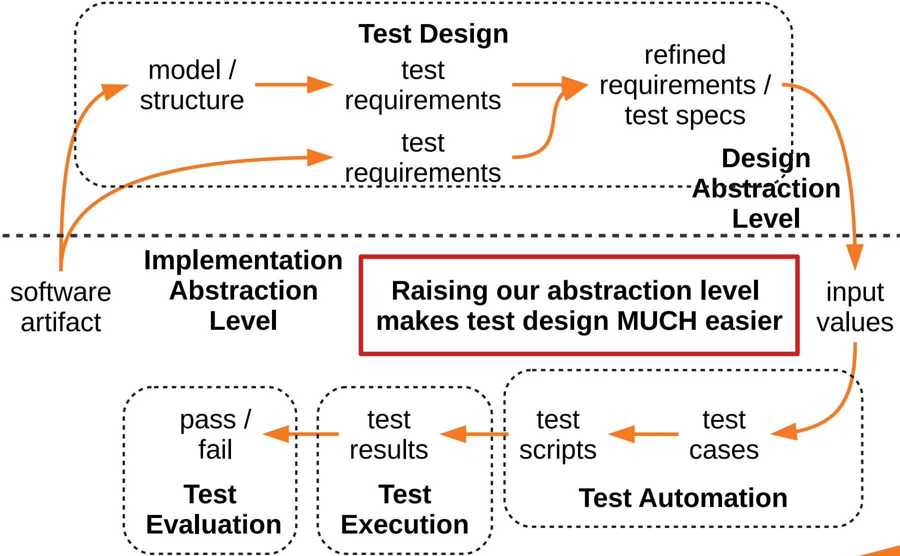
test results

pass / fail

# MDTD - Steps


Criterion
Refine
model / structure
test requirements
refined requirements / test specs
analysis
domain analysis
test requirements
Generate
Design Abstraction Level
software artifact
Implementation Abstraction Level
Feedback
input values
Evaluate
Execute
Automate
Prefix
Postfix
Expected
pass / fail
test results
test scripts
test cases

# MDTD - Activities



Test Design

model / structure

test requirements

test requirements

refined requirements / test specs

Design Abstraction Level

software artifact

Implementation Abstraction Level

**Raising our abstraction level makes test design MUCH easier**

input values

test cases

test scripts

Test Automation

test results

Test Execution

pass / fail

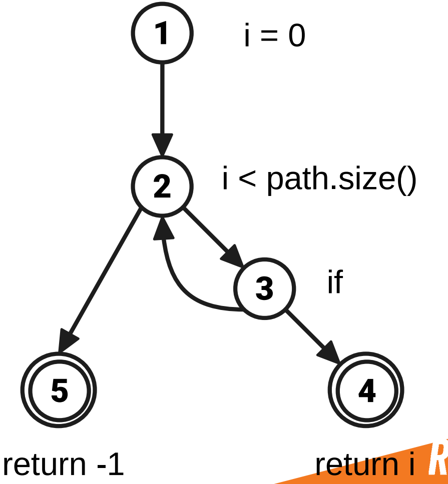
Test Evaluation

# Small Example

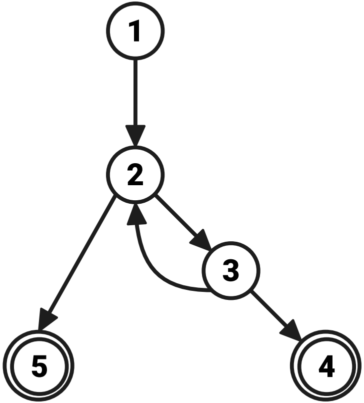
## Software Artifact: Java Method

```java
/**
 * Return index of node n at the
 * first position it appears,
 * -2 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)



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 the book is about test design

Other activities are well covered elsewhere

# In-Class Exercise

## Group Discussion: **Coverage Criteria**

- Why do software organizations use coverage criteria?
- Why don’t more software organizations use coverage criteria?

You have five minutes.

# Are there any questions?