Introduction to Software Testing
Model-driven Test Design

Software Testing & Maintenance
SWE 437
http://go.gmu.edu/swe437

Dr. Brittany Johnson-Matthews
(Dr. B for short)
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 (2.0)

Testing can only show the presence of failures, not their absence!

Remember: 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

Test
Reaches
Fault
Infected
Propagate
Reveal
Incorrect
Final State

Observed Final Program State

Test Oracles
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)

Class A
- method mA1()
- method mA2()

Class B
- method mB1()
- method mB2()

main Class P
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 one another

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 one another

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

**Unit testing (developer testing):**

- Class A:
  - method mA1()
  - method mA2()

- Class B:
  - method mB1()
  - method mB2()

- main Class P
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 & 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: testing transparency

**Opaque** (or black box) **testing**: derive tests from external descriptions of the software, including specifications, requirements, and design.

**Transparent** (or 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).

Model Driven test design 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
Testing 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**

*Using the same people for all four test activities clearly wastes resources.*
Testing 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**

*Using the same people for all four test activities clearly wastes resources.*
Criteria-based test design

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
Human-based test design

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...)

Can be intellectually stimulating, typically not preferred by CS majors.
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

- software artifact
- model / structure
- implementation abstraction level
- test requirements
- test requirements
- design abstraction level
- refined requirements / test specs
- input values
- pass / fail
- test results
- test scripts
- test cases
Model-driven test design

- Analysis
- Domain analysis
- Software artifact
- Criterion
- Test requirements
- Refine
- Refined requirements / test specs
- Design abstraction level
- Implementation abstraction level
- Evaluate
- Test results
- Automate
- Test cases
- Execute
- Test scripts
- Feedback
- Input values
- Prefix
- Postfix
- Expected
- Pass / fail
- Model / structure
- Generate
Model-driven test design

- Test Design
  - model / structure
  - test requirements
  - refined requirements / test specs

- Test Evaluation
  - pass / fail

- Test Execution
  - test results

- Test Automation
  - input values
    - test cases
    - test scripts
**Software Artifact : Java Method**

* Return index of node n at the first position it appears,
* -1 if it is not present

```java
public int indexOf(Node n) {
    for (int i=0; i < path.size(); i++)
        if (path.get(i).equals(n))
            return i;
    return -1;
}
```
Support tool for graph coverage

http://www.cs.gmu.edu/~offutt/softwaretest/
In this textbook...

Most of the content 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.