Model-Driven Test Design

CS 3250 Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 2]
Software Testing

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

Test case consists of test values and expected results

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**
RIPR Model

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

[AO, p.21]
Complexity of Testing Software

- $11 \times 11 = 121 \quad \checkmark$
- $1111 \times 1111 = 1234321 \quad \checkmark$
- $111111 \times 111111 = 1245654321 \quad \times$

NetBadge Web Login
NetBadge is a digital "badge" allowing you to access protected resources on the UVa network

Got a UVa Digital Certificate?
Log in with your Digital Certificate.
(What’s this? | Get one now!)

No Certificate?
Log in with your UVa computing ID and a password you use for one of the compatible systems.

Applicant for admission or SCPS student? Use this option.

UVa computing ID
Password
Log In

Protect Your Privacy!
To log out, completely exit your Web browser when you are finished.
- Windows users: Close all Web browser windows.
- Mac users: Quit your Web browser.

Otherwise, your NetBadge access will last for 9 hours if you're on Grounds (1 hour if you're off Grounds)—and someone else can use your browser to log in as you.
Testing Levels and Types of Faults

Requirements Analysis → Architectural Design

Test Design Information → System Test

Subsystem Design → Integration Test

Detailed Design → Module Test

Implementation → Unit Test

Acceptance Test

Check if software does what the user needs

Check overall behavior w.r.t. specs

Check interface between modules in the same subsystem

Check interactions of units and associated data structures

Check each unit (method) individually

[based in part on AO, p.23]
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)

- **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
Each activity requires different skills, background knowledge, education, and training

[AO, p.22]
# 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 (biology, psychology, ...)
  - Logic background is very helpful (law, philosophy, math, ...)

## Criteria-based approach

- Design test values to satisfy coverage criteria
- The most technical job in software testing
- Require knowledge of
  - Discrete math
  - Programming
  - Testing
- Require much of a traditional CS degree
- Using people who are not qualified to design tests will result in ineffective tests
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
  
  Examples: Cover every statement
  Cover every functional requirement

- **Test requirements**: Specific things that must be satisfied or covered during testing
  
  Examples: 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 four types of structures.

1. Input domains
2. Graphs
3. Logic expressions
4. Syntax descriptions
Characteristics of Good Tests

Each test case:

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

© Praphamontripong
Other Activities

- **Test management**
  - Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed, ...

- **Test maintenance**
  - Save tests for result as solve evolve
  - 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
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

• 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
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

• Test designers become technical experts

• Many test designers get involved in crowd testing
Model-Driven Test Design

- model / structure
- test requirements
- refined requirements / test specs
- test requirements
- Design Abstraction Level
- Implementation Abstraction Level
- software artifact
- input values
- pass / fail
- test results
- test scripts
- test cases

[AO, p.30]
Model-Driven Test Design - Steps

1. Analysis
   - Model/structure
   - Criterion
   - Test requirements
   - Refine
   - Refined requirements/test specs

2. Design Abstraction Level
   - Analysis
   - Domain analysis
   - Test requirements
   - Generate
   - Input values
   - Feedback

3. Implementation Abstraction Level
   - Software artifact
   - Test cases
   - Test scripts
   - Evaluate
   - Test results
   - Execute
   - Automate
   - Prefix/postfix expected

[AO, p.30]
Model-Driven Test Design - Activities

**Test Design**
- model / structure → test requirements → refined requirements / test specs
- design abstraction level
- implementation abstraction level
- software artifact

**Test Automation**
- input values → test cases → test scripts → test results → pass / fail

**Test Evaluation**
- test automation

[AO, p.30]
Wrap-up

• This course focuses on test design with criteria-based approach

• Testing activities
  • Design tests: model software + apply test coverage criteria
  • Automate tests
  • Execute tests
  • Evaluate tests

• Characteristics of good test cases

What’s Next?

• Test automation