Testing, TDD, and R

And A Little on [S]OLID

Donald Sawyer (dsawyer@phdata.io)

Presentation & Source Code on GitHub:

https://github.com/donaldsawyer/phdata-tdd-r

Essentials of Software Testing

1. Quality of process determines success of test effort
2. Use techniques to test early in lifecycle
3. Use tools
4. Somebody MUST take responsibility for improving testing
5. Testing requires trained, skilled people
6. Embrace a creative destruction culture

Principles of Testing

1. Shows presence of bugs, never their absence
2. Hard to know when to stop
3. Impossible to test your own program
4. Where there’s smoke, there’s fire
5. Creative and intellectually challenging

What is Unit Testing?

- Types of testing
  - Unit
  - Integration
  - System
  - Data Quality, Functional, Acceptance, UX, and many more…
- Unit Testing
  - Testing of an individual code module, function, or “unit”
  - Done by a developer before deployment
  - Verifies input/output of a “unit”
- Why?
  - Verify the individual components perform as expected
  - Gain confidence that “done” code is still working as the program changes

Concepts of Automated Testing

- Coded tests
    - Your tests are code
    - They need code review too
- SOLID is the key
- They should be run by your CI tools (Jenkins)
- Try to achieve high coverage
- You’ll use a framework
- Mocking is often required (more in detailed slides)

Quick Tips

1 Test > No Tests. Always.

Coder:
I’ve only got legacy code and it’s really hard to start integrating automated unit tests.

Me:
So? Be creative. Refactor.

Coder:
There's no framework for the language I'm using.

Me:
You're a developer, build what you need.

You’ll need to learn a lot of new things.

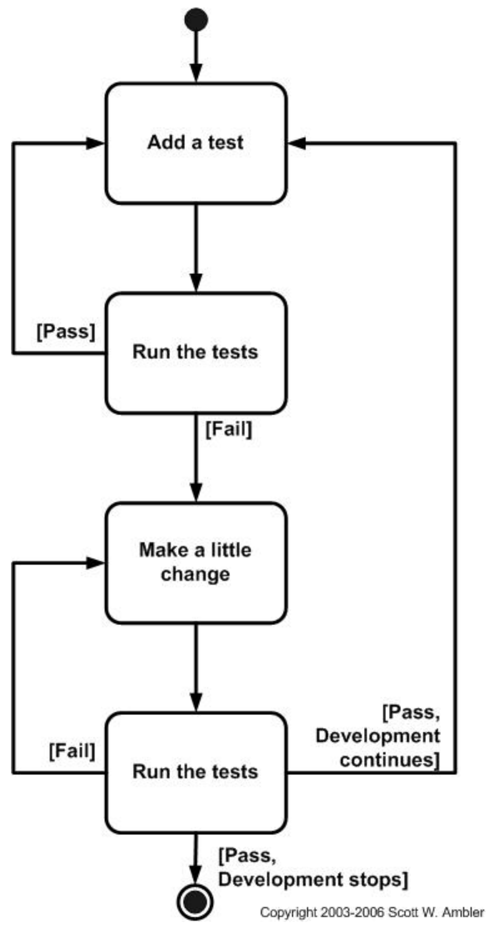
If you can “find time” to support defects and product issues, you can make time to start testing.

Let’s Get Started!

An Introduction to SOLID

- SOLID Principles of Object-Oriented Design
    - S – Single Responsibility
    - O – Open/Closed Principle
    - L – Liskov Substitution
    - I – Interface Segregation
    - D – Dependency Inversion
- Single Responsibility
    - A module should have a single responsibility and that responsibility should be entirely encapsulated in the module
    - A module should have one, and only one, reason to change
    - ***THIS is what we'll unit test in this talk***

Test-Driven Development (TDD)



Add a test
[Pass]
Run the tests
[Fail]
Make a little change
[Fail]
Run the tests
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

- Helps you understand your requirements
- Forces you to test (YAY!)
- If your test doesn’t fail first, it’s a bad test
  - Avoid false positives
  - Ensure better tests
- When you have tests, you can CONFIDENTLY change your code in the future
- Incorporate SOLID to make it easier
- It takes practice!

Whole Lotta Definitions

You first need to know what testing is.

Statement Coverage

- Every statement executed once

Snippet →

- Only one test needed to get 100% statement coverage

```
are_both_positive(1, 1)
```

```
def are_both_positive(x, y):
    rval = false

    if x > 0 and y > 0:
        rval = true

    return rval
```

Multiple Condition Coverage

- All possible Boolean expression outcomes are tested (all combinations)

Snippet →

- If x is positive, the 2nd condition is never tested
- Need 4 cases
    1. (-1, -1)
    2. (-1, 1)
    3. (1, -1)
    4. (1, 1)

```python
def are_any_positive(x, y):
    rval = False

    if x > 0 or y > 0:
        rval = True

    return rval
```

Boundary Value Analysis

Boundary Testing

- Test values near design limits
  - Value limits
  - First/last rows in a table or data frame
  - Null, empty, single-character strings
- Errors tend to occur near extremes
- Robustness: reaction when boundaries are exceeded

Boundary Testing (cont’d)

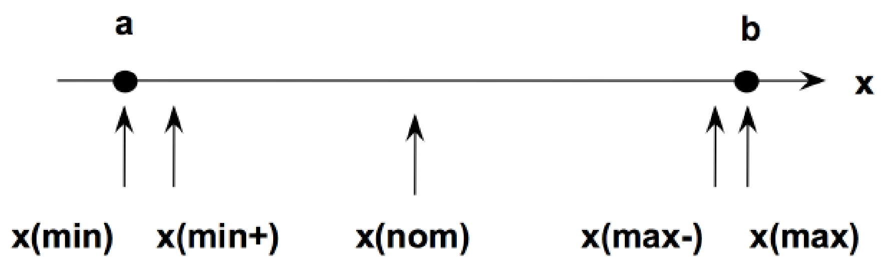
Single Fault Assumption (SFA):

Assume that a failure is NOT the result of 2+ simultaneous faults.

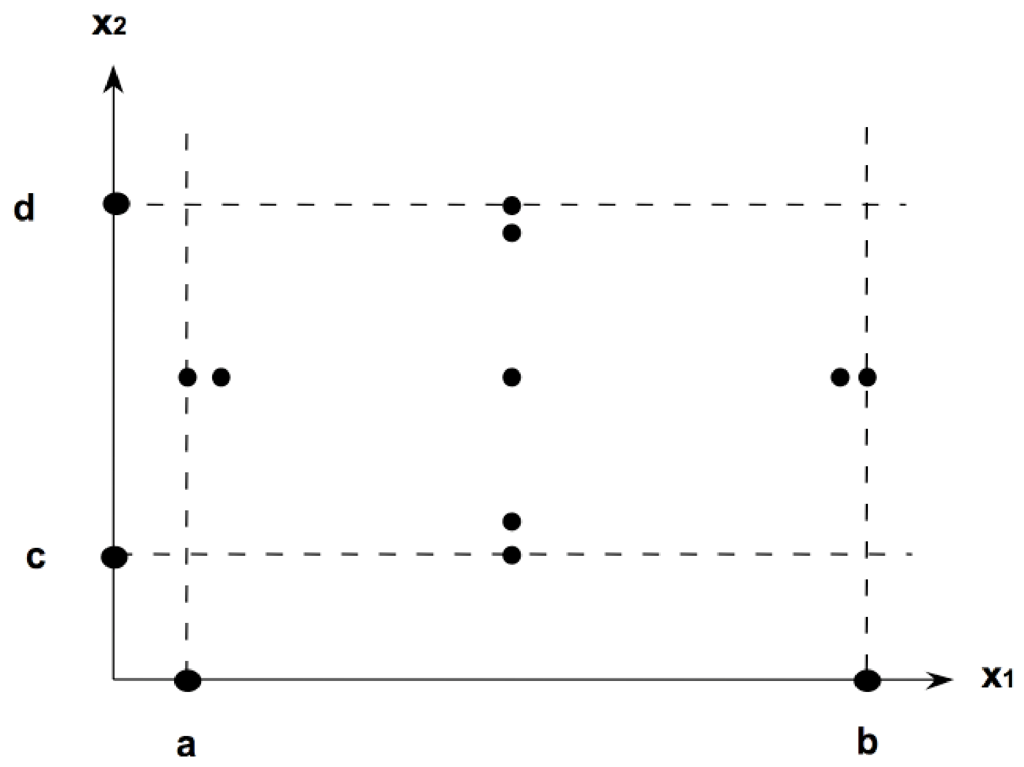
SFA Boundary Value Testing	• Stay within the design limits
SFA Robustness Boundary Value Testing	• Exceed the limits, if possible
Worst-case Boundary Value Testing	• Two or more variables • Discard SFA • Stay within limits
Worst-case Robustness Boundary Value Testing	• Two or more variables • Discard SFA • Exceed the limits

Boundary Values

Single Input Variable



Two Input Variables

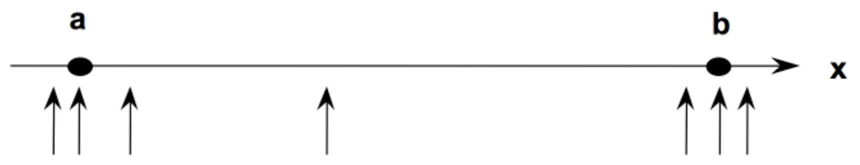


Best Practice:
Use input variables at and just above minimum, and at and just below maximum.

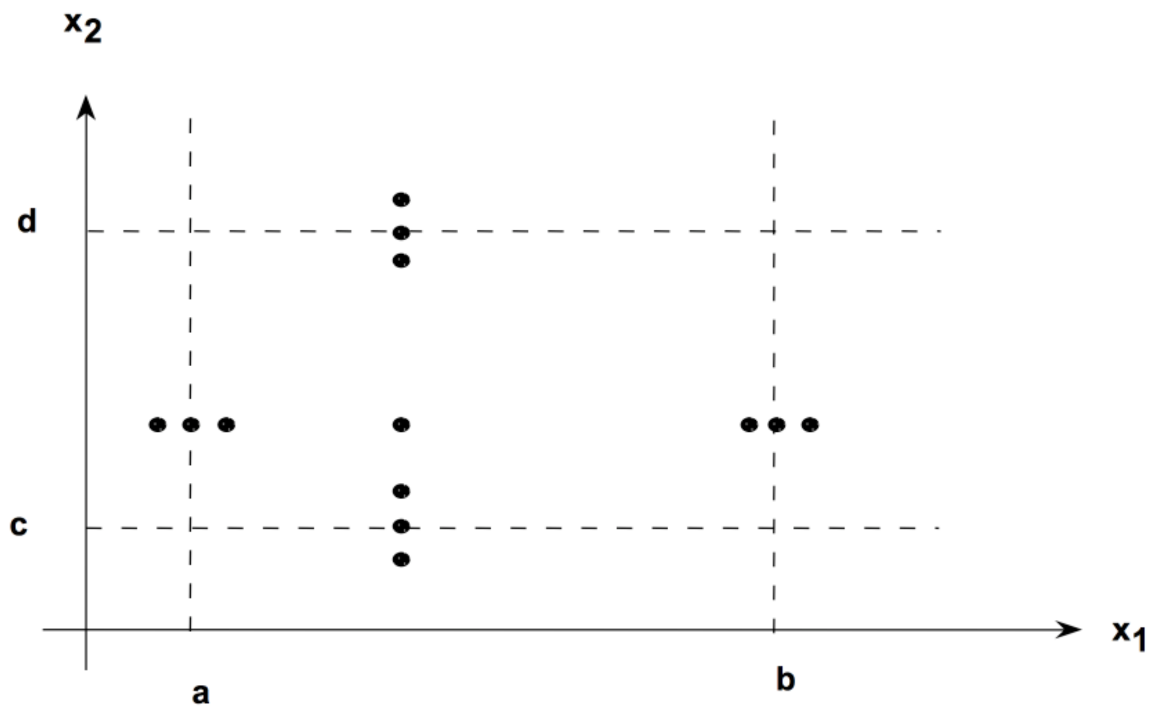
Robustness Boundary Value Testing

Include values below the minimum and above the maximum.

Single Input Variable



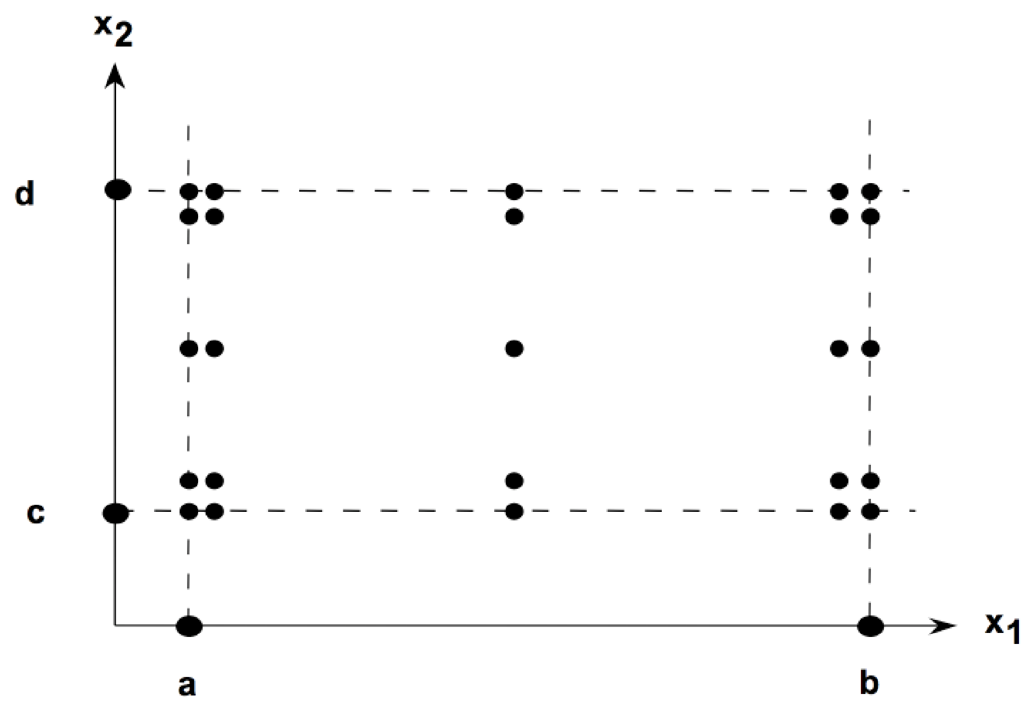
Two Input Variables



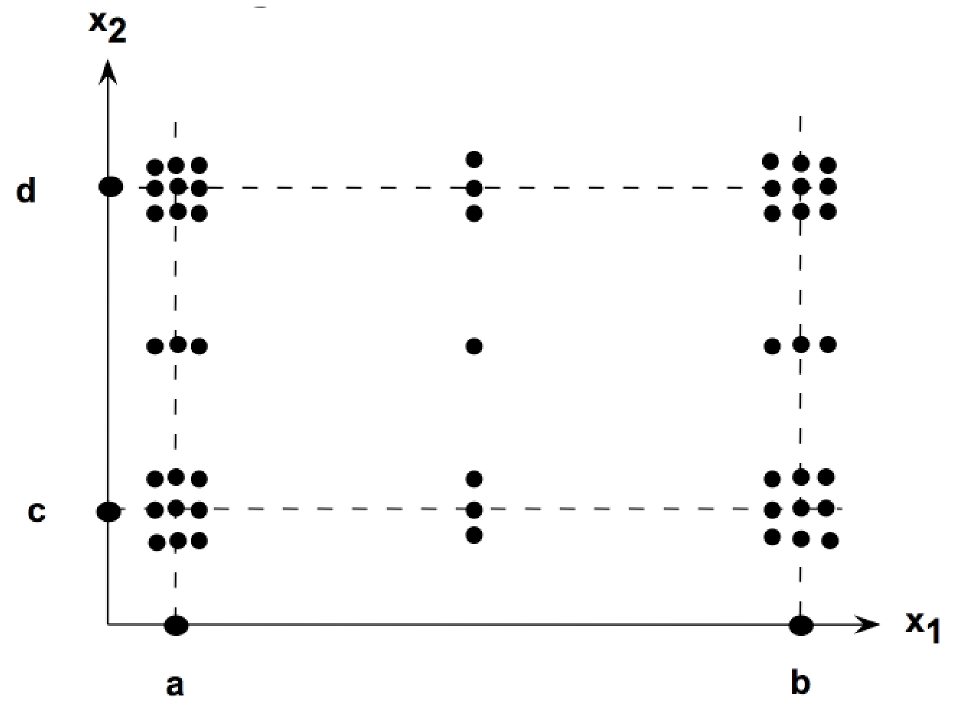
Worst-Case BV Testing

Allow both variables to approach (and maybe exceed) boundaries.

Worst-Case Boundary Value



Worst-Case Robustness BV (paranoid)



Choosing Your Inputs

Using equivalence classes to choose inputs

Equivalence Classes

Mathematical Definition

    The sets in a partition of the domain (input or output)

Motivation

    Gain a sense of complete testing without redundancy

Create Equivalence Classes

    Determine the boundaries → Determine the equivalencies

If a set of values all cause the program to behave in exactly the same way, then that might be a candidate for an Input Equivalence Class.

Demo App in R

Coded Unit Testing with RUnit

- Why?
  - Every time you change your code, you want to re-test it, ENTIRELY
  - Without coded testing, we skip testing pieces we've already tested
  - Gain confidence in testing changes to large data sets
  - What if we broke something we previously built?
    - We won't know until a defect is reported
  - Integrate testing within the deployment process
- Package: Runit
  - Allows automated verification of code units
  - Allows a test suites to be defined and tested together
  - Allows for test results to be reported
- Steps
  - Create a test driver
  - Create test scripts
  - Add test scripts to test suites in driver
  - Run test driver
  - View results

A Doorbuster Metric Program

Requirements

1. Shall read two product csv files
    1. Doorbuster1.csv
    2. Doorbuster2.csv
2. Shall add two metrics to the product data
    1. Products that are doorbusters but have no price assigned
    2. Products that are doorbusters and online, but are out of stock
3. Shall write out a csv with the following:
    1. Product id
    2. No price metric
    3. Online but out of stock metric

The R Example

- 0_monolith.R
  - A typical R Script
  - Contains a lot of code that all runs in sequence
- What the script does
  - Reads 2 data sets from csv (doorbuster data)
    - doorbuster1.csv & doorbuster2.csv
  - Combines the datasets
  - Adds metrics for
    - Doorbuster items missing a price
    - Doorbuster items that are online, but out of stock
  - Writes metric data to csv called doorbuster_metrics.csv

Refactor: Extract Method

- Purpose
  - Take a chunk of code and create a method/function so it can be tested.
  - Make a small function to follow the Single Responsibility principle
- In 0_monolith.R
  - Create a method for reading a single doorbuster csv file
  - Create a method that reads both doorbuster csv files
  - Create a method that adds metric for doorbusters with no price
  - Create a method that adds metric for doorbusters that are online but out of stock

To Refactor (the TDD Way)

1. Stub out a function for the code being refactored
2. Write the appropriate unit tests
3. RUN the tests → They BETTER fail!
4. Fill the code into the method
5. RUN the tests & fix function until the tests pass
6. Move on
7. Every change to the program you make, RUN ALL THE TESTS
8. Every defect that gets reported → CREATE NEW TESTS

Helpful Resources

- TDD
  - https://en.wikipedia.org/wiki/Test-driven_development
  - http://www.agiledata.org/essays/tdd.html
- Refactoring
  - http://refactoring.com/
  - Code Smells: https://en.wikipedia.org/wiki/Code_smell

Detailed Slides

Slides that were removed from the main presentation.

Verification or Validation?

Verification

Check a program against a "design" or specification.

Evaluate artifacts to ensure their correctness.

Did you build the THING RIGHT?

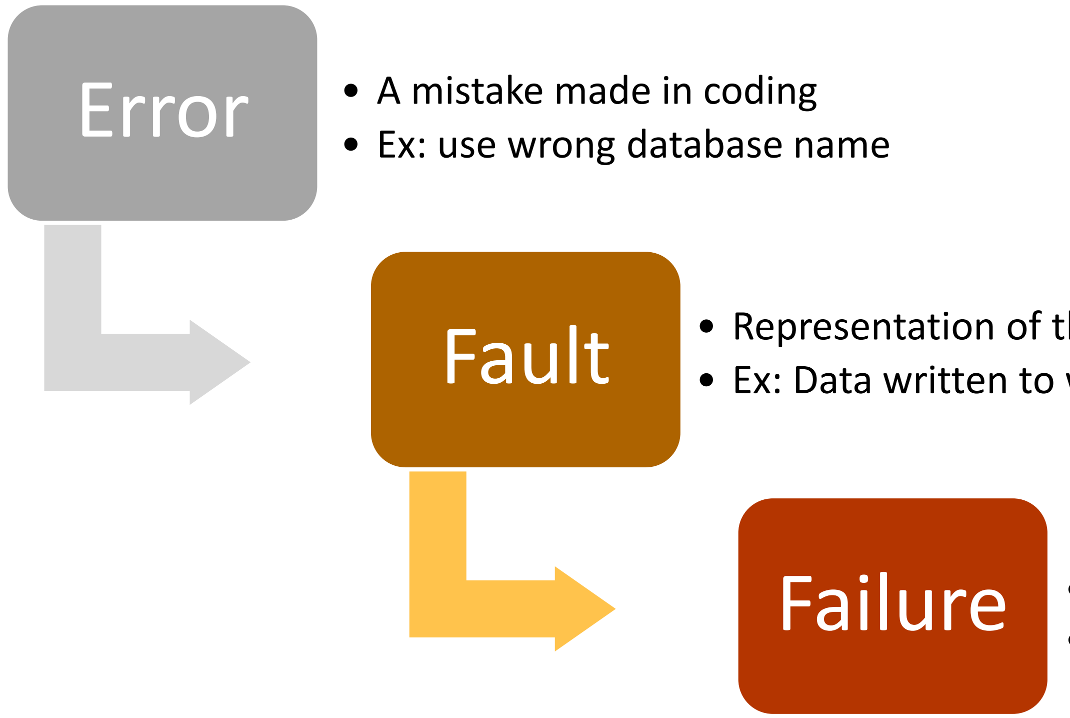
Validation

Check a system against an expectation/requirement.

Focus on product –related activities by executing something!

Did you build the RIGHT THING?

Error, Fault, and Failure



- A mistake made in coding
- Ex: use wrong database name

- Representation of the error
- Ex: Data written to wrong database

- User-visible manifestation of the failure
- Ex: User doesn't see new data

The tester is tasked with writing cases to cause failures!

Example: Error vs. Fault vs. Failure

```
def count_postive_transactions(transList):
    positive_tran_count = 0
    for t in transList:
        if t.saleAmount >= 0:
            positive_tran_count += 1
    return positive_tran_count
```

If saleAmount is 0, it's not a positive transaction. The logical operator is wrong.

Fault: sales transactions are counted incorrectly if their saleAmount is 0.

- What is the error made?
- What is the fault?
- Give a test case that will cause a failure.
- Give a test case that will *not* cause a failure (fault still there).

Failed Test Case: call function with at least one transaction where saleAmount = 0.

This is how most people test, at BEST… and why our [automated] tests stink.

Happy Path

No Failure Case: call function where all transactions have saleAmount greater than 0.

Black vs. White Box Testing

Black Box Testing	White Box Testing
• No knowledge of internal structure/design • Based on functional spec • AKA Functional Testing	• Examine internal design of program • Requires knowledge of program structure • AKA Structural Testing

Static vs. Dynamic Testing

Static Testing

- Test w/o executing software
- Structural testing
- Walkthroughs, reviews, compiling

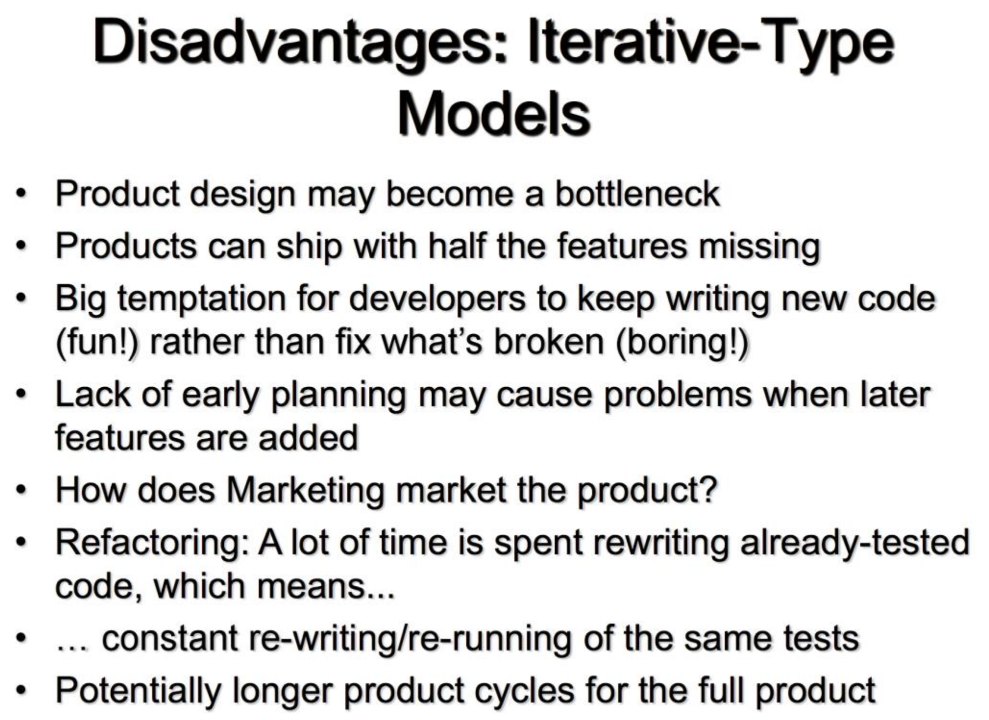
Dynamic Testing

- Test by executing software
- Structural & functional testing

Main Types of Testing

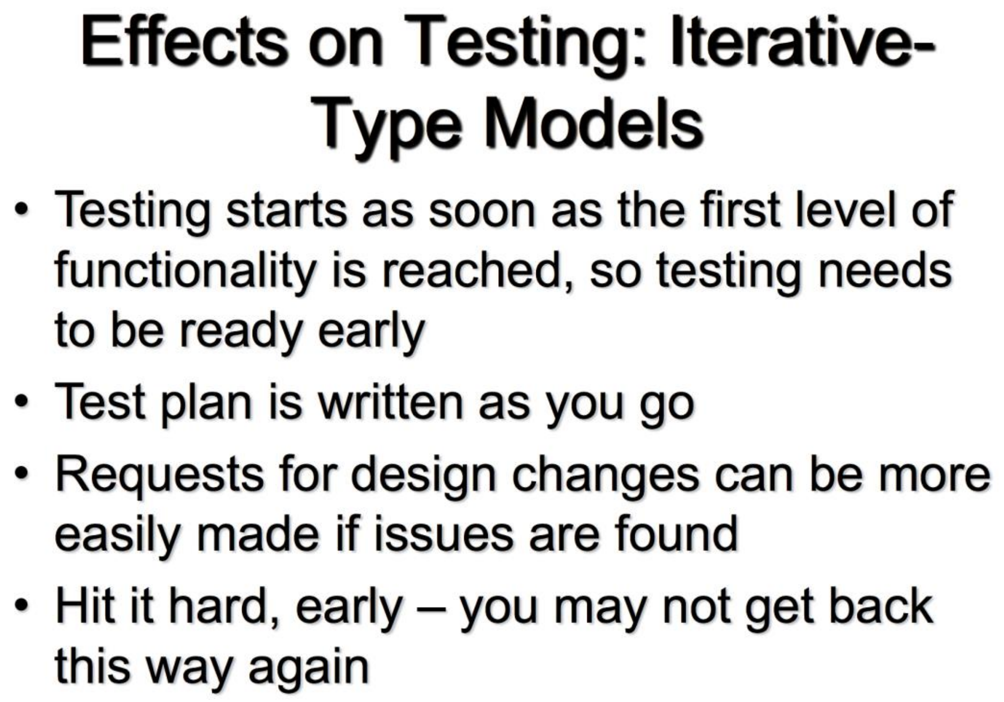
Testing Type	Typically by Who?	White/Black	Automate?
Unit	Programmer	White	YES
Integration	Programmer	Mostly white	Desired
Functional/System	Debatable, historically not the programmer	Mostly black	Desired
Acceptance	Customer/Product Owner	Black	Not likely, but doable
Regression	It depends	All	Good automation of unit, integration, and system testing can cover this.
Design Verification	Team	White	No. Verifying design artifacts.

Testing in an Agile Model



Disadvantages: Iterative-Type Models

- Product design may become a bottleneck
- Products can ship with half the features missing
- Big temptation for developers to keep writing new code (fun!) rather than fix what's broken (boring!)
- Lack of early planning may cause problems when later features are added
- How does Marketing market the product?
- Refactoring: A lot of time is spent rewriting already-tested code, which means...
- … constant re-writing/re-running of the same tests
- Potentially longer product cycles for the full product



Effects on Testing: Iterative-Type Models

- Testing starts as soon as the first level of functionality is reached, so testing needs to be ready early
- Test plan is written as you go
- Requests for design changes can be more easily made if issues are found
- Hit it hard, early – you may not get back this way again

Code Coverage

- Statement
- Branch/Decision
- Condition
- Multiple Condition
- Modified Condition/Decision Coverage (MCDC)
- Path

Statement Coverage

- Every statement executed once

Snippet →

- Only one test needed to get 100% statement coverage

```
are_both_positive(1, 1)
```

```
def are_both_positive(x, y):
    rval = false

    if x > 0 and y > 0:
        rval = true

    return rval
```

Branch/Decision Coverage

- Every decision branch gets executed at least once

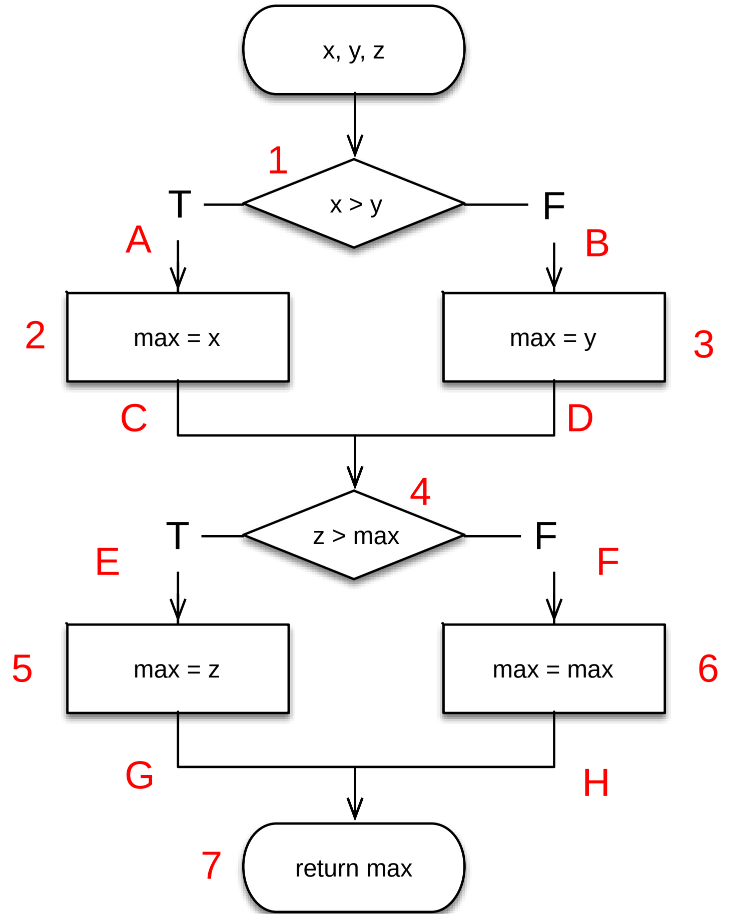
Program Snippet/Path →

- Two tests needed
  1. 1A-2C-4E-5G-7
  2. 1B-3D-4F-6H-7
- Could different test paths be used?

```
def get_max(x, y, z):

    max = x if x > y else y
    max = z if z > max else max

    return max
```



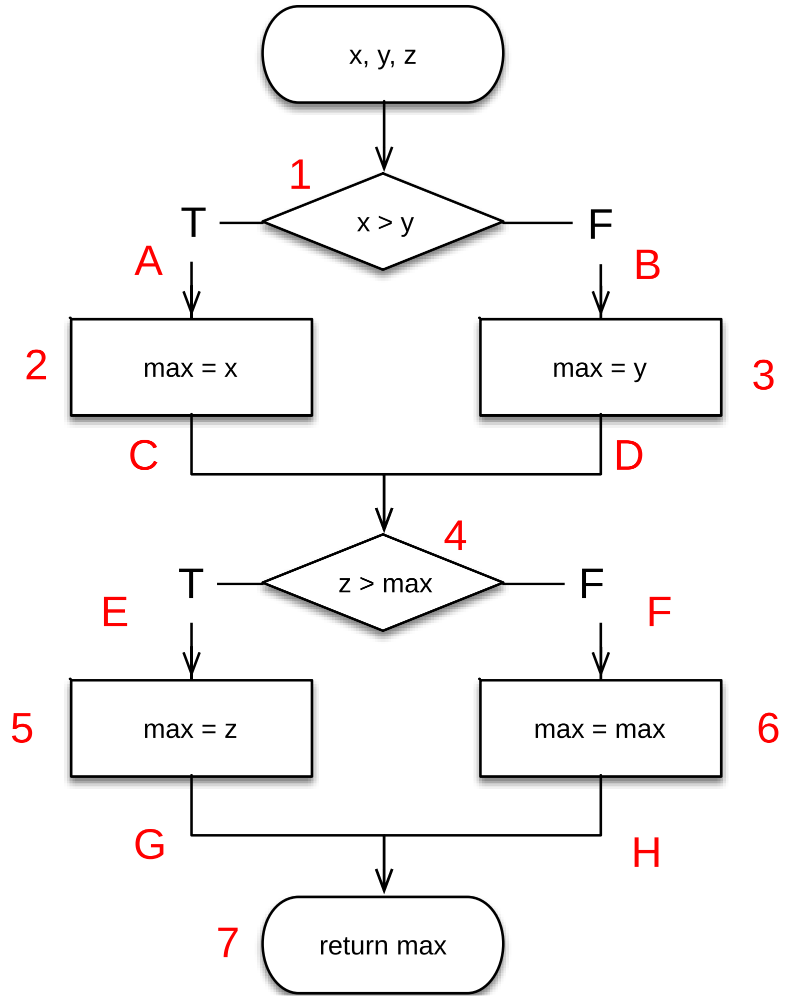
Condition Coverage

- Every Boolean sub-expression has been evaluated to both True and False at least once.

```
def get_max(x, y, z):

    max = x if x > y else y
    max = z if z > max else max

    return max
```



Program Snippet/Path →

- Two tests needed
  1. 1A-2C-4E-5G-7
  2. 1B-3D-4F-6H-7
- Could different test paths be used?

Multiple Condition Coverage

- All possible Boolean expression outcomes are tested (all combinations)

Snippet →

- If x is positive, the 2nd condition is never tested
- Need 4 cases
  1. (-1, -1)
  2. (-1, 1)
  3. (1, -1)
  4. (1, 1)

```python
def are_any_positive(x, y):
    rval = False

    if x > 0 or y > 0:
        rval = True

    return rval
```

Boundary Value Analysis

Boundary Testing

- Test values near design limits
  - Value limits
  - First/last rows in a table or data frame
  - Null, empty, single-character strings
- Errors tend to occur near extremes
- Robustness: reaction when boundaries are exceeded

Boundary Testing (cont'd)

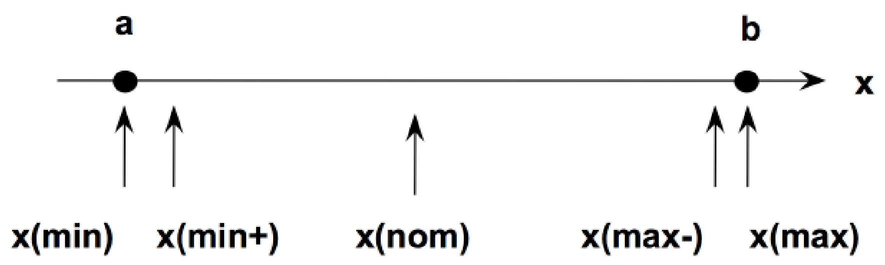
Single Fault Assumption (SFA):

Assume that a failure is NOT the result of 2+ simultaneous faults.

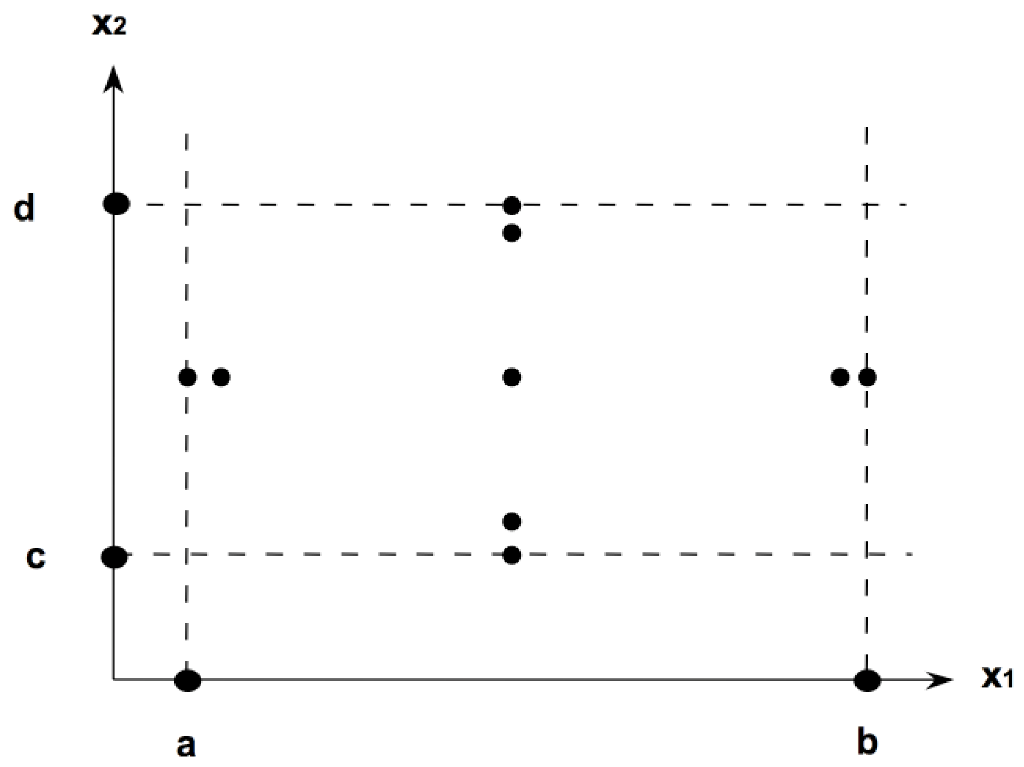
SFA Boundary Value Testing	• Stay within the design limits
SFA Robustness Boundary Value Testing	• Exceed the limits, if possible
Worst-case Boundary Value Testing	• Two or more variables • Discard SFA • Stay within limits
Worst-case Robustness Boundary Value Testing	• Two or more variables • Discard SFA • Exceed the limits

Boundary Values

Single Input Variable



Two Input Variables

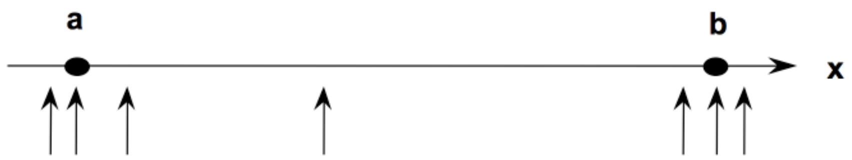


Best Practice:
Use input variables at and just above minimum, and at and just below maximum.

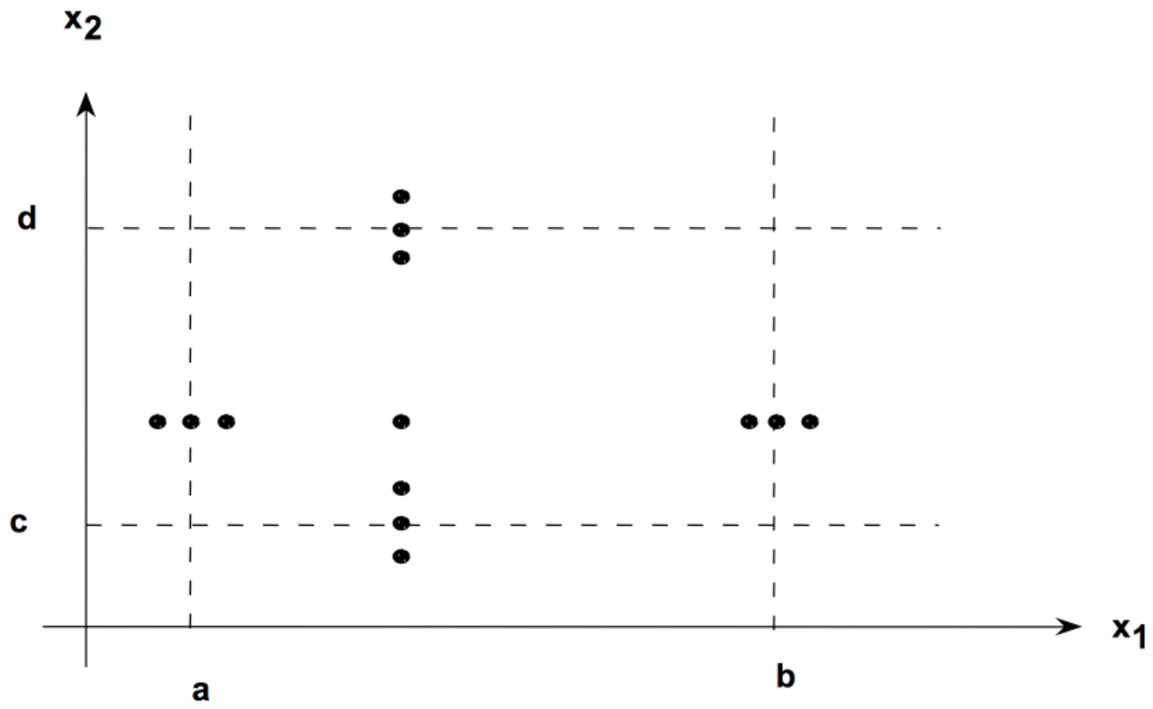
Robustness Boundary Value Testing

Include values below the minimum and above the maximum.

Single Input Variable



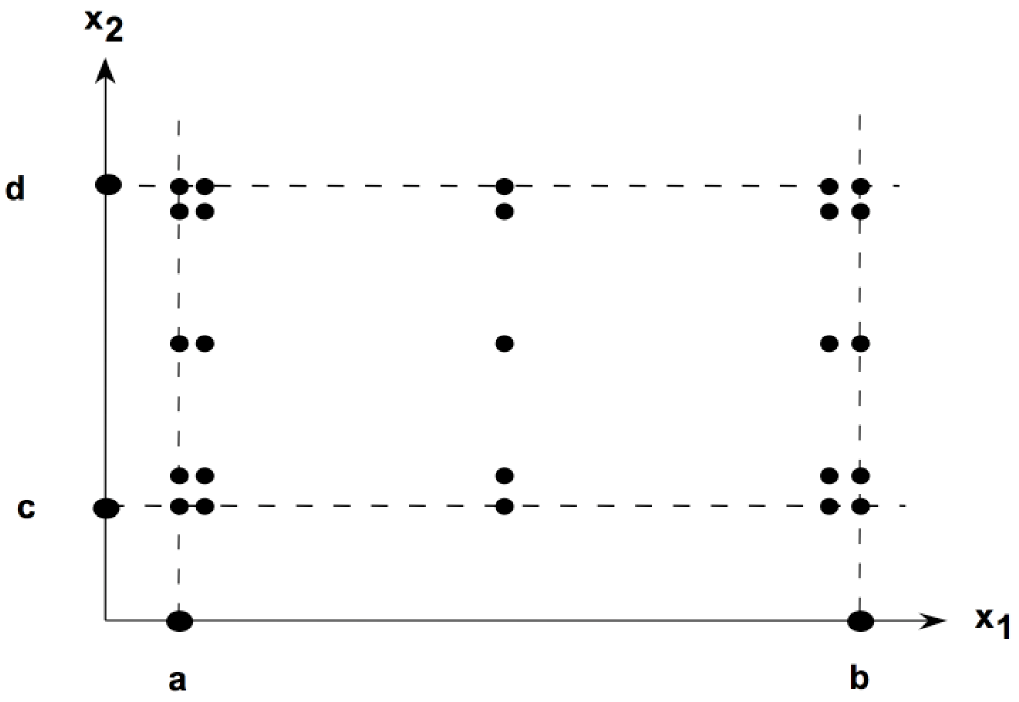
Two Input Variables



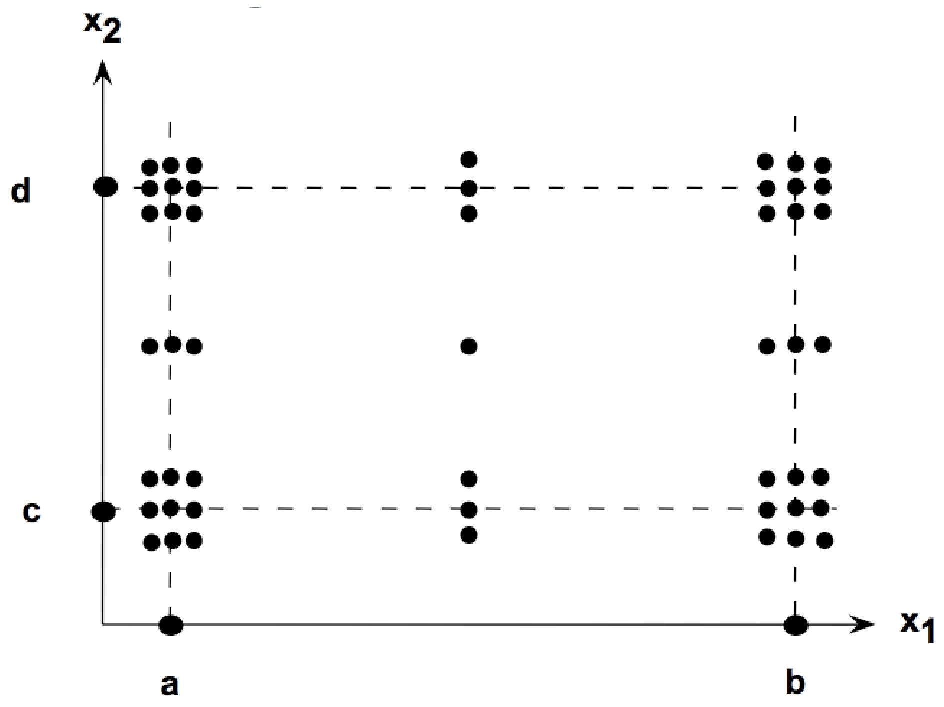
Worst-Case BV Testing

Allow both variables to approach (and maybe exceed) boundaries.

Worst-Case Boundary Value



Worst-Case Robustness BV (paranoid)



Boundary Testing: How Many Tests?

- SFA BV: $4n + 1$
- SFA Robust BV: $6n +1$
- Worst-Case BV: 5^n
- Paranoid: 7^n

Testing Automation

A Few Notes on Automation

- It’s hard
- You’ll hit roadblocks
- You MUST be creative
- Apply all the test theory to develop good tests
- It is VERY expensive up front
- The quality is absolutely worth the investment
- Work toward eliminating the need for formal regression testing

Concepts of Automated Testing

- Coded tests
  - Your tests are code
  - They need code review too
- SOLID is the key
- They should be run by your CI tools (Jenkins)
- Try to achieve high coverage
- You'll use a framework
- Mocking is often required (more in two slides)

Automation Tools & Resources (Some)

- Unit Testing Frameworks
- Static Code Analysis
- Javascript tools
- GUI Testing Tools
- Apache Pig
- Big Data
- In EDABI (BI-System-Automation)

Concept: Mocking

- Sometimes your program calls other classes, methods, or external modules.
- Mocking simulates these complex integrations.
- You can mock exceptions being thrown.

- Examples:
  - Calling legacy libraries
  - Calling other classes
  - API Calls
  - DB Calls