Com S 417
Software Testing

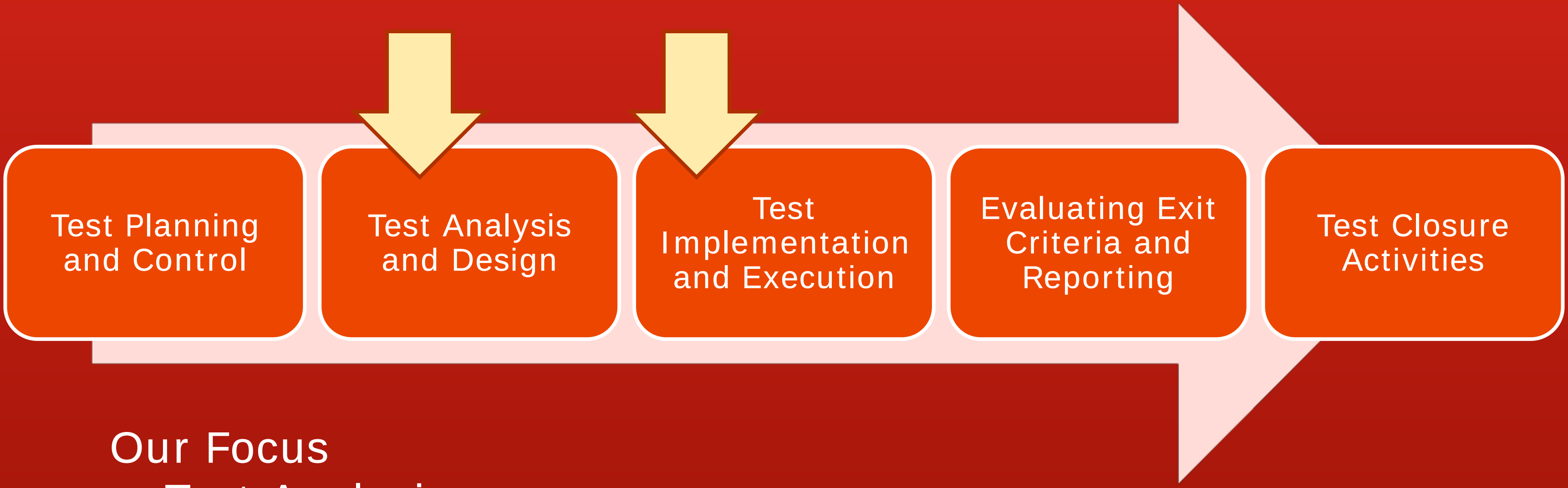
Fall 2017 – Week 2, Lecture 3

Knowing What To Test

Requirements; Black Box vs. White Box; Input Partitioning

Test Activities

ISTQB Test Process


Test Planning and Control
Test Analysis and Design
Test Implementation and Execution
Evaluating Exit Criteria and Reporting
Test Closure Activities

Our Focus

- Test Analysis
  - includes studying/reviewing requirements
  - Identifying Test conditions, Test Cases, Test Steps
  - Designing and prioritizing test cases.
  - Choosing coverage criteria/strategy
- Test implementation and execution
  - Writing the tests (junit for us)
  - Running tests
- Wash and repeat

Test Case

A specification of the specific input values, expected output, preconditions, and execution instructions for a single test.

For the next few chapters, we will not treat preconditions as separate, but capture them as additional state input to the test.

Note that each @Test in a Junit TestCase Class is a separate Test Case under this definition.

Requirements

Definition

Requirements

Our understanding of product correctness is based on our perception of product requierments. Thus we create tests based on requirements.

Definition (three versions from different sources)

- Requirements are specifications of what functions the product is expected to perform.
- Requirements describe what the product is intended to accomplish for its users.
- What the product is meant to do and be.

Requirement Sources

Requirements are determined by customer/owner needs. Multiple sources and stake holders may be consulted when writing requirements.

Even though testers are sometimes consulted (especially about testability) during the requirements writing process, we consider the requirements process outside the scope of this class.

Different organizations produce requirements differently. These differences effect scope, completeness, specificity, clarity, and when the requirements are available.

The less complete and rigourous the requirements, the more interpretation required during test generation.

A central agile principle is to defer finalizing requirements until the last responsible moment.

Scope of requirements

Requirements address both behavior and quality attributes, such as speed, security, capacity (load), reliability, and usability.

Requirements can also capture information about constraints and the use context.

> During periods of heavy load (more than 2000 new session requests per second), the product shall generate a page response to each login request in less than 500 ms.

Black Box vs. White Box

Black Box vs. White Box Tests

- A distinction based on whether information about the specific implementation is used in test design.
- Black Box Tests: tests designed without reference to software design documents (such as high-level and low-level design documents) or source code.
  - Example: the SUT is available only as a packaged .jar file. No other information, other than the requirements, is available.
- White Box Tests: test designed with information about the actual implementation of the software. E.g., source code, class and interface definitions, design drawings, etc.

Black Box Tests

Advantages:

- Can be designed before the implementation design is started.
- Can be used to test proprietary, NIH products.
- Use simple optimization techniques.

Disadvantages:

- Tests tend to be less productive because they are selected without knowledge of how the product is implemented.

Input Domains
& Input Partitioning

The Input Domain

- The input domain (or input space) is ‘the set of all possible inputs to the program’.
  - Where “an input” is the collection of all n-tuples of possible combinations of values for all n inputs.
- A typical input domain is effectively infinite.
  - consider, how many different unsorted sets integers exist? Each set is an element in the input domain for a simple sort program.

NOTE: Invalid inputs are still part of the input domain. (Should the program just blow up? or should it give an error message? What should be in the error message?)

Focusing our test effort.

Are all members of the Input Domain equally likely to be associated with a fault?

Thought Exercise:

If you were testing a calculator’s multiply function and you could on afford to run 5 tests, what tests would you choose?

Note: each test input is a 5-tuple of the form:

```
(<last operation result>,
  <left operand>,
  <operator>,
  <right operand>,
  <expected result>
)
```

Would you use these five:

- (0,1,*,1,1)
- (0,1,*,2,2)
- (0,1,*,3,3)
- (0,1,*,4,4)
- (0,1,*,5,5)

Why or why not?

Equivalence Partitions

- If 1*1 and 1*2 work, how likely is it that 1*3 or 4 or 5 will fail?
- Wouldn’t it be better to use your limited test budget to find out what happens
  - if the prior result isn’t zero? or
  - if one of the operands is zero? or
  - if one of the operands is negative? or
  - if one of the operands is ‘*’?
- An equivalence class is a set of inputs for which we believe the Software under Test will generate the same behavior (correct or incorrect).
- An equivalence partition is a partition of an input set into a set of equivalence classes.

Equivalence Classes for operands

Example partitioning

- negative 1
- all other negative numbers
- zero
- positive one
- all other positive numbers
- improperly formed integer (ex: ‘0-’)
- illegal character (ex: ‘*’)
- Memory reference (ex: ‘m’)

Equivalence classes for prior result?

Equivalence classes for operands?

Tests for Partitions

D = union(p1, p2, p3 ...) and

empty = intersection(p1, p2, p3 ...)

- Partition must be complete.
- Partition must be disjoint.

Easiest to accomplish if you choose a single characteristic or rule to determine the partitioning:

- non-negative (divides integers into neg and [0→])
- zero/non-zero.
- legal/illegal
- We will refer to each equivalence class determined by a characteristic as a ‘block’.

We will look at how to deal with multiple characterists within the same domain later.

How should we design test cases?

We will start with three basic strategies:

All Combinations -- all combinations (representative from each equivalence class) for all inputs.

Intuitive. Relatively comprehensive. What is the cost?

Each Choice – one choice from each equivalence class used in at least one test.

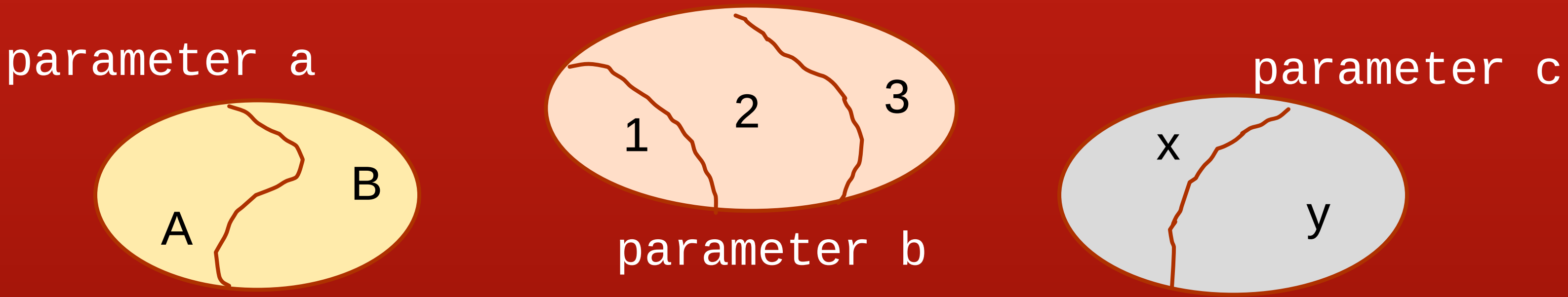
Opposite extreme from above.

Pairwise Combinations – *all pair wise combinations (by input) of representatives from each equivalence class.*

A middle ground.

All Combinations Example

Assume three parameters a,b,c (inputs), with spaces partitioned as below:



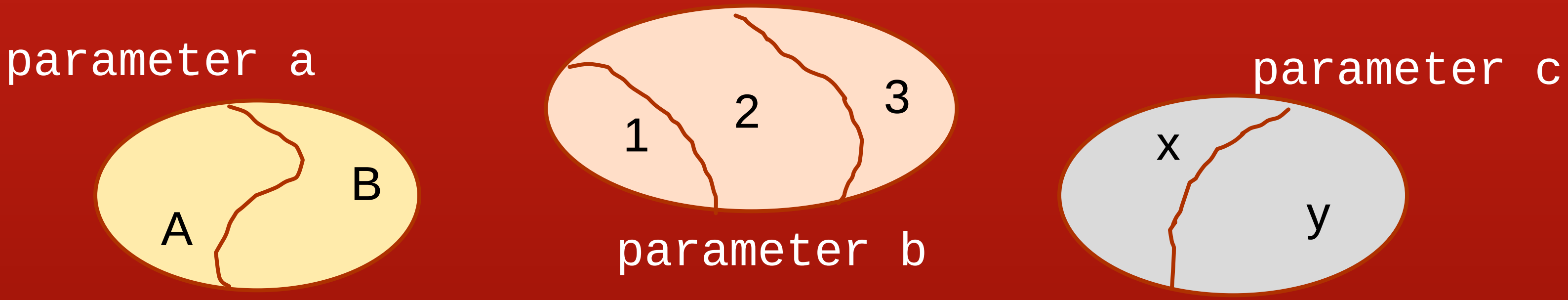
All Combinations	
(A, 1, x)	(B, 1, x)
(A, 1, y)	(B, 1, y)
(A, 2, x)	(B, 2, x)
(A, 2, y)	(B, 2, y)
(A, 3, x)	(B, 3, x)
(A, 3, y)	(B, 3, y)

We can call each equivalence class a 'block.' Here, parameter a has two blocks named A and B.

The block name in the test case represents a single (arbitrary) value chosen from that block.

Each Choice Example

For “Each choice” a value chosen from each block must appear in at least one test:

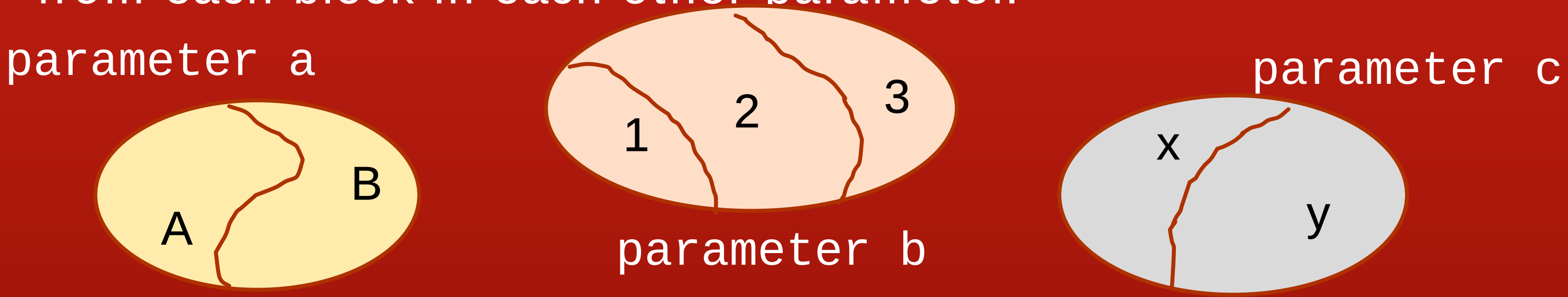


Each Choice
(A, 1, x) (B, 2, y)
(A, 3, x)

Many fewer tests, but many more opportunities for a bug to go undiscovered.

Pair-Wise Example

For “Pair wise” a value for each block in each parameter partition (characteristic) must be combined with a value from each block in each other parameter.



Pairs to Include

(A,1)	(B,1)	(1,x)
(A,2)	(B,2)	(1,y)
(A,3)	(B,3)	(2,x)
(A,x)	(B,x)	(2,y)
(A,y)	(B,y)	(3,x)
		(3,y)

Candidate Tests

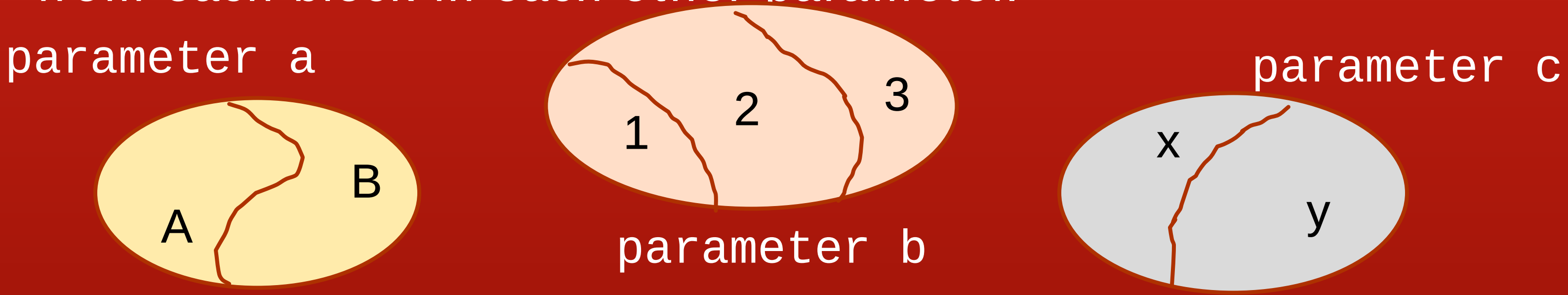
(A, 1, x)	(B, 1, x)
(A, 1, y)	(B, 1, y)
(A, 2, x)	(B, 2, x)
(A, 2, y)	(B, 2, y)
(A, 3, x)	(B, 3, x)
(A, 3, y)	(B, 3, y)

Choose a combination from the candidates and mark the embedded pairs out of the list.

Then choose some remaining candidate that does not contain any of the previously used pairs.

Pair-Wise Example

For “Pair wise” a value for each block in each parameter partition (characteristic) must be combined with a value from each block in each other parameter.



Pairs to Include

(A,1)	(B,1)	(1,x)
(A,2)	(B,2)	(1,y)
(A,3)	(B,3)	(2,x)
(A,x)	(B,x)	(2,y)
(A,y)	(B,y)	(3,x)
		(3,y)

Pairwise Tests

(A, 1, x)	~~(B, 1, x)~~
~~(A, 1, y)~~	(B, 1, y)
(A, 2, x)	~~(B, 2, x)~~
~~(A, 2, y)~~	(B, 2, y)
(A, 3, x)	~~(B, 3, x)~~
(A, ~~3~~, y)	(B, ~~3~~, y)

Many fewer tests, but many more opportunities for a bug to go undiscovered.

Mathur introduces pairwise design in sections 4.6 and 4.7

What have we accomplished

- Test Design Strategies that work without access to the code or other implementation details.
- By identifying equivalence sets, we significantly reduce the input values we must test with little effect on bug discovery.
- We have shown three alternate test selection strategies that can be used to further reduce the number of tests, but with more significant impact on bug discovery.
  - All combinations
  - Each Choice
  - Pair-wise combinations

Coming Up

- The meaning of coverage
- Counting the cost of selection rules
- More test vocabulary
- Basic White Box Techniques
- Code Coverage Tools

Reading Assignment

- Chapter 1: Sections 1.13, 1.14.
- Chapter 2: From beginning through 2.3.4. In 2.3.4, skip the discussion of multidimensional partitioning.

Lab 1 Questions