LECTURE 10:
TESTING, DEBUGGING, VERIFICATION
BLACK-BOX TESTING

Outline

- Testing Fundamentals
- Testing vs. Verification
- Testing Phases
- Unit Testing
  - Black-Box Testing
House Keeping

- Design document
  - 1st draft due on Sunday 04/08/2018 by 11:59pm
  - Final draft due on Sunday 04/15/2018 by 11:59pm
- 2nd Exam
  - Tuesday 04/24/2018
- Final presentations
  - Thursday (05/03) & Friday (05/04)
  - 8:00-9:00, 9:30-10:30, 11:00-12:00, 1:30-2:30, & 3:00-4:00
- Project Demo (source code)
  - Week of finals

Tuesday’s Quote

“Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often”

Steve McConnell; Author, Code Complete
Software Testing is a critical element of producing quality software. It represents the ultimate review of the requirements specification, the design, and the code. It is the most widely used method to insure software quality. Many organizations spend 40-50% of development time in testing.

Testing is the process of determining if a program has any errors. Debugging is concerned with finding where defects occur (in code, design or requirements) and removing them. (fault identification and removal)

Even if the best review methods are used (throughout the entire development of software), testing is necessary.
Testing Fundamentals - 3

- Testing is the one step in software engineering process that could be viewed as destructive rather than constructive.

- A successful test is one that uncovers an as yet undiscovered defect.

Program Behavior

![Program Behavior Diagram]

- **Specification (expected)** vs **Program (observed)**
- **Missing Functionality (sins of omission)**
- **Extra Functionality (sins of commission)**
- "Correct" Portion
**Correctness**

- Relative term: Program P is correct with respect to specification S
- Do specification and program meet customer expectations?
- Is it possible to demonstrate correctness?
  - Impossible to demonstrate
  - Or is it?

**Testing Steps**
Unit Testing

- **Unit Testing** checks that an individual program unit (subprogram, object, package, module) behaves correctly.

- **Static Testing**
  - testing a unit without executing the unit code

- **Dynamic Testing**
  - testing a unit by executing a program unit using test data

Black vs. White-box Testing

- **Specification**
  - Functional (Black Box) establishes confidence

- **Program**
  - Structural (White Box) seeks faults
A **test case** is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A **test set** is a collection of zero or more test cases.

**Sample test case # 1 for Triangle:**
Test data: <2, 3, 4>
Expected output: Scalene

**Sample test case # 2 for Triangle:**
Test data: <5, 15, 25>
Expected output: Not a Triangle
White-Box Testing

Objective: to ensure that code implemented the design properly.

Unit & Integration Testing

- Objective: to ensure that code implemented the design properly.
Unit and Integration Testing

Test Oracles

- Two types of oracles
  - Human: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.
  - Automated: a system capable of performing the above task.
Testing can be viewed as selecting different colored balls from an urn where:
- Black ball = input on which program fails.
- White ball = input on which program succeeds.
- Only when testing is exhaustive is there an “empty” urn.

Balls and Urn - 2

A program that always fails
A correct program
A typical program
Black-box/Closed-box Testing

- Incorrect or missing functions
- Interface errors
- Performance error

Black-Box Testing Techniques

- Definition: a strategy in which testing is based on requirements and specifications.
- Applicability: all levels of system development
  - Unit
  - Integration
  - System
  - Acceptance
- Disadvantages: never be sure of how much of the system under test (SUT) has been tested.
- Advantages: directs tester to choose subsets to tests that are both efficient and effective in finding defects.
Black-Box Testing

- Exhaustive testing (Typically not possible)
- Equivalence class testing
- Boundary value analysis
- Decision table testing
- Pairwise testing
- State-Transition testing
- Domain analysis testing
- Use case testing

Correctness

- Relative term: Program P is correct with respect to specification S

- Do specification and program meet customer expectations?

- Is it possible to demonstrate correctness?
  - Impossible to demonstrate
  - Or is it?
Goal of Testing

Can we prove a program is correct by testing?

Exhaustive Testing

- Definition: testing with every member of the input value space.
- Input value space: the set of all possible input values to the program.
- How Feasible?

```c
int increment (int a) {
    ..........
}
```

# of possible inputs: From -2,147,483,648 to 2,147,483,647, from \((2^{31})\) to \(2^{31} - 1 = 2^{32} - 1\) possible inputs

If one test takes 1 ms, how much time will it take to exhaustively test the function above?

Time to test \(2^{32} - 1\) ms = \((2^{32} - 1)/1000\) sec = 1.6 months

```c
long increment (long a) {
    ..........
}
```

Time to test \(2^{64} - 1\) ms = \((2^{64} - 1)/1000\) sec = 5,845,420,461 years
Goal of Testing 2

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence” [Dijkstra 1972]

Equivalence Class Testing

- Testing technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.

- Each EC consists of a set of data that is treated the same by the module or that should produce the same result.

- Any data value within a class is equivalent, in terms of testing, to any other value.
Black-box Testing Techniques

Equivalence Class Testing Technique

Partition test cases into classes such that:
1. Every possible input belongs to one of the classes
2. No input belongs to two different classes
3. If we demonstrate a fault in the code for a given input, we should demonstrate a fault with any other input from the same class (with a high probability)
4. Classes are identified by looking at boundary values for the variables of the application
Example-1

- If \((x > y)\) then \(S_1\) else \(S_2\)

Equivalence classes for values of \(x\) and \(y\):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &gt; y)</td>
<td>(x \leq y)</td>
</tr>
</tbody>
</table>

Example - 2

- Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

  - Input condition
    - range of values \(x\): 1-50
  
  - Valid equivalence class
    - \(? < x < ?\)
  
  - Invalid equivalence classes
    - \(x < ?\)
    - \(x > ?\)
Example - 2

- Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values x: 1-50

- Valid equivalence class
  - $0 < x < 51$

- Invalid equivalence classes
  - $x < 1$
  - $x > 50$

Example 3 (Group Exercise)

- The specification states that the program accepts 4 to 10 inputs which are five-digit integers greater than 10,000.

  What are the partitions?
Example 3-Partitions

The program specification states that the program accepts 4 to 10 inputs which are five-digit integers greater than 10,000.

Number of input values

- $x < 4$
- $4 \leq x \leq 10$
- $x > 10$

Input values

- $y < 10,000$
- $10,000 \leq y \leq 99,999$
- $y > 99,999$

Discussion Point

- Consider a module for human resources system that decides how to handle employment process based on the applicant's age. The rules are:
  - 0-16: Don't Hire
  - 16-18: Can hire only as part-time
  - 18-55: Can hire as a full-time
  - 55-99: Don't hire

- Typical test would be (15, Don't Hire)

- How about test cases like (969, ...), (FRED, ...), (&#$@, ...)?
Testing based on design-by-contract relies on the module's pre and post conditions.

- Pre condition: defines what the module requires for it to accomplish its task
  - Example: for a module that reads a file, an appropriate pre condition would be that the file is readable

- Post condition: defines the conditions that must be met if the module completes its task(s) successfully
  - Example: file is open and ready for operations

In testing based on design by contract, you have to worry only about testing the conditions satisfied by the pre condition.

In defensive design, the module is designed to accept any input. If pre conditions are met, the module will achieve its post conditions. Otherwise, the module will notify the caller by returning an error code or throwing an exception.

As a tester, you have to consult with designers in order to understand which approach is being used. Even though they might not be even aware of it.
Equivalence Class Testing: Guidelines

1. If an input condition specifies a range of values; identify one valid EC and two invalid EC.

2. If an input condition specifies the number (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).

3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.

4. If an input condition specifies a “must be” situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter)

5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

Identifying Test Cases

- Assign a unique number to each EC.

- Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.

- Until all invalid ECs have been covered by test cases, write a test case that covers one, and only one, of the uncovered invalid ECs.
Group Exercise

Problem statements: NextDate is a function of three variables: month, day and year. It returns the date of the day after the input date. The month, day and year variables have integer values subject to these conditions:

C1: 1 <= month <= 12
C2: 1 <= day <= 31
C3: 1812 <= year <= 2012

Valid ECs:

Invalid Ecs:

The craft of testing is to combine test cases to test as many ECs as possible within a single test.

But this might cause a problem. Any ideas why?

Group Exercise - Answers

Valid ECs:

M1 {Month: 1 <= Month <= 12}
D1 {Day: 1 <= Day <= 31}
Y1 {Year: 1812 <= Year <= 2012}

Invalid Ecs:

M2 {Month: 1 > Month },
M3 {Month: Month > 12},
D2 {Day: 1 > Day },
D3 {Day: Day > 31},
Y2 {Year: 1812 > Year },
Y3 {Year: Year > 2012}

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected Output</th>
<th>Ecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/15/2010</td>
<td>10/16/2010</td>
<td>M1, D1, Y1</td>
</tr>
<tr>
<td>0/15/2010</td>
<td>Invalid</td>
<td>M2</td>
</tr>
<tr>
<td>14/15/2010</td>
<td>Invalid</td>
<td>M3</td>
</tr>
<tr>
<td>10/0/2010</td>
<td>Invalid</td>
<td>D2</td>
</tr>
<tr>
<td>10/35/2010</td>
<td>Invalid</td>
<td>D3</td>
</tr>
<tr>
<td>10/10/1800</td>
<td>Invalid</td>
<td>Y2</td>
</tr>
<tr>
<td>10/10/2020</td>
<td>Invalid</td>
<td>Y3</td>
</tr>
</tbody>
</table>
Group Exercise

Consider the example of the Goofy Mortgage Company (GMC).

- Mortgages are allowed for:
  - A person with income between $1,000 and 83,333
  - Condominiums, Townhouses, Single family dwellings
  - 1 to 5 dwellings in a single mortgage

- Mortgages are not allowed for:
  - Corporations, Trusts, or Partnerships
  - Duplex, Mobile Home, or Tree houses
  - Less than one dwelling or more than 5

<table>
<thead>
<tr>
<th>Monthly Income</th>
<th># Dwellings</th>
<th>Applicant</th>
<th>Dwelling Type</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>2</td>
<td>person</td>
<td>townhouse</td>
<td>Grant Mortgage</td>
</tr>
<tr>
<td>500</td>
<td>2</td>
<td>person</td>
<td>townhouse</td>
<td>Deny Mortgage</td>
</tr>
<tr>
<td>85,000</td>
<td>2</td>
<td>person</td>
<td>townhouse</td>
<td>Deny Mortgage</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>person</td>
<td>townhouse</td>
<td>Deny Mortgage</td>
</tr>
<tr>
<td>2500</td>
<td>7</td>
<td>person</td>
<td>townhouse</td>
<td>Deny Mortgage</td>
</tr>
<tr>
<td>2500</td>
<td>2</td>
<td>Corporation</td>
<td>townhouse</td>
<td>Deny Mortgage</td>
</tr>
<tr>
<td>2500</td>
<td>2</td>
<td>Person</td>
<td>Mobile Home</td>
<td>Deny Mortgage</td>
</tr>
</tbody>
</table>

Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.

- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system.

- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system’s requirements.
Boundary Value Testing

- Equivalence class testing has the advantage of reducing the number of test case.
- But it also has the other advantage of alerting us to the notion of boundary testing.
- Boundary value testing focuses on the boundaries simply because that is where so many defects hide.
  - The defects can be in the requirements or in the code.
- The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham’s book).

Going back to the HR system example:
- 0-16: Don’t Hire
- 17-19: Can hire only as part-time
- 20-55: Can hire as a full-time
- 56 -99: Don’t hire

Boundary Value Testing - Technique

- Identify the ECs.
- Identify the boundaries of each EC.
- Create test cases for each boundary value by choosing:
  - one point on the boundary,
  - one point just below the boundary,
  - and one point just above the boundary.
What are the boundary values to test?

Less than 10000 | Between 10000 and 99999 | More than 99999
Boundary Value Analysis Test Cases for Triangle program (upper bound for any size is 200)

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>1</td>
<td>Isosceles</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>100</td>
<td>2</td>
<td>Isosceles</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
<td>199</td>
<td>Isosceles</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>Not a Triangle</td>
</tr>
</tbody>
</table>

Quiz

Given the following method specification:

```java
/** Returns true iff key is contained in values. */
//@ requires values != null;
//@ ensures \result == (\exist int i; i >= 0 & i < values.length;  
//@ values[i] == key);
public static boolean contains(int[] values, int key)
```

Perform boundary analysis to select a reasonable set of test cases.
Answer

Test suite:
(({}, 2), false),
({2,33,24,5,66,7,22,6}, 9), false),
({2,33,24,5,66,7,22,6}, 2), true),
({2,33,24,5,66,7,22,6}, 33), true),
({2,33,24,5,66,7,22,6}, 22), true),
({2,33,24,5,66,7,22,6}, 6), true))

Applicability and Limitations

- Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels.
- All it requires are inputs that can be partitioned and boundaries that can be identified based on the system’s requirements.
Decision Table Testing

- Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design.

- They are used to record complex business rules that a system must implement.

- In addition, they can serve as a guide to creating test cases.

Technique: General Format of a Decision Table
Don’t Care Entries

The don’t care entry has two interpretations:
- the condition is irrelevant, or
- the condition does not apply. Sometimes the “n/a” symbol for this latter interpretation.

Types of Decision Tables

- Limited entry decision tables:
  - all the conditions are binary

- Extended entry decision tables:
  - conditions are allowed to have several values.

- Decision tables are deliberately declarative (as opposed to imperative):
  - no particular order is implied by the conditions, and selected actions do not occur in any particular order.
Decision Table-Based Testing

- Applicable for requirements in *if-then* form
  - if $C_1$ and $C_2$ and ... and $C_n$ then $A_k$

- Create a decision table comprised of conditions and actions.

- Number of columns: $2^n$ ($n$ is number of conditions)

- Number of rows: $n + m$, where $m$ is number of actions

- For each set of conditions, there is a corresponding action

---

Decision Table to Test Cases

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Test Case 1</th>
<th>Test Case 2</th>
<th>...</th>
<th>Test Case P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Text Editor

Functions: copy, paste, boldface, underline, and select.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy selected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bold selected</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Text Highlighted</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Text copied</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Text bolded</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Develop Decision Table for the Triangle Problem

- What are the conditions?
  - C1: a, b, c form a triangle?
  - C2: a = b?
  - C3: a = c?
  - C4: b = c?
- What are the actions?
  - Not a triangle, scalene, isosceles, equilateral, impossible
- How many rows?
  - # of rows = #conditions + #actions
- How many columns?
  - 2 # of conditions
  - Or is it?
Group Exercise: Triangle program

The program accepts three integers, \( a \), \( b \), and \( c \) as input. The three values are interpreted as representing the lengths of sides of a triangle. The integers \( a \), \( b \), and \( c \) must satisfy the following conditions:

\[
\begin{align*}
\text{a} &< \text{b} + \text{c} \\
\text{b} &< \text{a} + \text{c} \\
\text{c} &< \text{a} + \text{b} \\
1 &\leq \text{a} \leq 200 \\
1 &\leq \text{b} \leq 200 \\
1 &\leq \text{b} \leq 200
\end{align*}
\]

- If the integers \( a \), \( b \), and \( c \) do not constitute a triangle, we do not even care about possible equalities (rule 1)
- If two pairs of integers are equal, by transitivity, the third pair must be equal; thus the negative entry (N) makes these rules impossible (rule 3).

Complete the Worksheet

<table>
<thead>
<tr>
<th>C1: a, b, c form a triangle?</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>C2: a = b?</td>
<td>DC</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>C3: a = c?</td>
<td>DC</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>C4: b = c?</td>
<td>DC</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

A1: Not a Triangle X
A2: Scalene X
A3: Isosceles X X X
A4: Equilateral X
A5: Impossible X X X
Let's Expand it

What if we expand the first condition to:

- C1: \( a < b + c \) ?
- C2: \( b < a + c \) ?
- C3: \( c < a + b \) ?

What will the decision table look like?

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C2</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C3</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>a=b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>a=c</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>b=c</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

- Not Triangle: X X X
- Scalene: X
- Isosceles: X X X
- Equilateral: X
- Impossible: X X X

Quiz - Decision-Table Testing
### Quiz- Decision-Table Testing

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Weekday</td>
<td>1. $65</td>
</tr>
<tr>
<td>2. Weekend</td>
<td>2. $80</td>
</tr>
<tr>
<td>3. El Paso Resident</td>
<td>3. $45</td>
</tr>
<tr>
<td>4. None resident</td>
<td>4. $60</td>
</tr>
<tr>
<td>5. Senior</td>
<td>5. $20</td>
</tr>
<tr>
<td>6. Junior</td>
<td>6. $30</td>
</tr>
<tr>
<td>7. Neither</td>
<td>7. $40</td>
</tr>
<tr>
<td></td>
<td>8. $50</td>
</tr>
</tbody>
</table>

Number of columns = $2^{\text{# of conditions}} = 2^7 = 128$

Number of rows = # of conditions + # of actions = 15

---

### Quiz- Decision-Table Testing

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Weekday</td>
<td>1. $65</td>
</tr>
<tr>
<td>2. Weekend</td>
<td>2. $80</td>
</tr>
<tr>
<td>3. El Paso Resident</td>
<td>3. $45</td>
</tr>
<tr>
<td>4. None resident</td>
<td>4. $60</td>
</tr>
<tr>
<td>5. Senior</td>
<td>5. $20</td>
</tr>
<tr>
<td>6. Junior</td>
<td>6. $30</td>
</tr>
<tr>
<td>7. Neither</td>
<td>7. $40</td>
</tr>
<tr>
<td></td>
<td>8. $50</td>
</tr>
</tbody>
</table>

Number of columns = $2^{\text{# of conditions}} = 2^4 = 16$

Number of rows = # of conditions + # of actions = 12
<table>
<thead>
<tr>
<th>Weekday</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resident</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Senior</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Junior</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>60</th>
<th>65</th>
<th>80</th>
</tr>
</thead>
</table>
Something Totally Different

- Example Handout
  - How would you test this?

Possible Strategies

- Don’t test at all
- Test all combinations
- Chose one or two tests and hope for the best
- Chose tests you have already run
- Chose the easy tests
- Make a list of all combinations and chose first few
- Make a list of all combinations and choose a random subset
- By magic, chose a specially selected and fairly small subset that can find great many defects
Pairwise Testing

- Test all pairs of variables when combinations to test is very large
- Most defects are either single-mode or double-mode defects
- # of tests needed to trigger a defect

![Pie chart](image)

Pairwise testing defines a minimal subset that guides us to test for all single-mode and double-mode defects.

- Four parameters, with three possible values each
  - $3^4 = 81$ possibilities
  - Can be done in 9 tests

- 13 parameters, with three possible values each
  - $3^{13} = 1,594,323$ possibilities
  - Can be done in 15 tests

- 20 parameters, with 10 possible values each
  - $10^{20} = MANY$ possibilities
  - Can be done in 180 tests
Orthogonal Arrays

Two-dimensional array of numbers that has this property:
- Choose any two columns in the array, all the pairwise combinations of values will occur in within the two columns.
- Three variables, with two possible values each:
  - $2^3 = 8$ Possibilities

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Note that we are not looking for triplets such as (1, 2, 1). We only care about pairs.
Orthogonal Array Example

- An orthogonal array with 4 columns (attributes), maximum value 3 (1-3)
- $L_9(3^4)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Orthogonal Arrays

- Not all columns must have the same range of values
  - $L_{18}(2^{13})$: 18 rows, 1 column of 1s and 2s, 7 columns of 1s, 2s, and 3s
  - Good news about pairwise testing is that we do not actually have to come up with the OA.
  - Tools will assist in doing this task
    - [http://www.phadkeassociates.com/index_rdexperttestplanning.htm](http://www.phadkeassociates.com/index_rdexperttestplanning.htm)
    - [http://designcomputing.net/gendex/noa/](http://designcomputing.net/gendex/noa/)
    - [https://app.hexawise.com](https://app.hexawise.com)
Using Orthogonal Arrays

- a) Identify the variables
- b) Determine the number of choices for each variable
- c) Locate an orthogonal array which has a column for each variable and values within the columns that correspond to the choices for each variable
- d) Map the test problem onto the orthogonal array
- e) Construct the test cases

A library of over 200 orthogonal arrays:
http://www.research.att.com/~njas/oadir/
Maintained by N.J. A. Sloane.

Handout Example

- a) Identify the variables:
  - Browser
  - Plug-in
  - Client operating system
  - Server
  - Server operating system

Number of variables = 5
b) Determine the number of choices for each variable

- Browser: Internet Explorer 5.0, 5.5, and 6.0, Netscape 6.0, 6.1, and 7.0, Mozilla 1.1, and Opera 7 (8 choices).
- Plug-in: None, RealPlayer, and MediaPlayer (3 choices)
- Client operating system: Windows 95, 98, ME, NT, 2000, and XP (6 choices).
- Servers: IIS, Apache, and WebLogic (3 choices)
- Server operating system: Windows NT, 2000, and Linux (3 choices)

Number of combinations: 1,296 (8*6*3*3*3).

c) Locate an orthogonal array which has a column for each variable and values within the columns that correspond to the choices for each variable

- 5 columns-array (5 variables):
  - 1 column for supporting 8 different levels.
  - 1 column for supporting 6 different levels.
  - 3 column for supporting 3 different levels.

As exact size for orthogonal array does not exist,
- pick up the next larger array; which is an array with 5 column, 64 rows,
- Two columns of 8 and 3 columns of 4
d) Map the problem into the orthogonal array: Use a tool
Handout Example (cont.)

e) Construct the test cases
  □ Construct a test case for each row in the orthogonal array
  □ (64 test cases)

Applicability and Limitations

□ It is equally applicable at the unit, integration, system and acceptance test levels

□ Reduces the number of test combination significantly

□ Issues with OA use