"Combinatorial Test Design: Beyond the Gee-wiz Numbers"

Presented by:

Karen Rosengren
IBM
Karen Rosengren

IBM

As a Senior Technical Staff Member and chief technologist for IBM's Quality Software Engineering team, Karen Rosengren is responsible for setting the direction on how the more than 20,000 IBM testers approach their work. With more than thirty years of experience in software engineering, Karen has held positions in development, architecture, design, and testing in both operating system and middleware. She has spent the majority of her career in various testing roles and held management positions as well. A passionate advocate for advancing testing as a career, Karen looks for every opportunity to mentor and guide testers.
Combinatorial Test Design: Beyond the Gee-Whiz Numbers

Karen Rosengren
Senior Technical Staff Member

IBM Corporate Headquarters
Office of Innovation and Technology

krosengr@us.ibm.com

Agenda

Introduction to Combinatorial Test Design

Our experience in IBM
- A good model of the test space is key
- Our existing tests have redundancy and gaps
- Generated test cases enable an automated test framework
- A good deployment model results in successful pilots and quicker time to value
Introduction to Combinatorial Test Design

- Combinatorial Test Design (CTD)
  - CTD is a structured, tool based design methodology where the test scenario is defined by the set of variables and their values, and tests are generated to cover all input values and combinations of input values.
  - Also called “Combinatorial Testing” or “Pairwise Testing” (Pairwise is a subset of CTD where the interaction is two-way)
  - Creates a model of the points of variability in a system.

- Cartesian products
  - The Cartesian product of two sets X and Y, denoted $X \times Y$, is the set of all possible ordered pairs whose first component is a member of X and whose second component is a member of Y.

- Levels of interaction (LOI)
  - LOI refers to the number of combinations a single attribute will be tested against. For example, a two-way LOI means that each value will be tested in conjunction with every other single value. While a three-way means every combination of three will be tested.

Standard playing cards have:
- Ranks (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
- Suits (Clubs, Spades, Hearts, Diamonds)

The Cartesian product has:
- $13 \times 4 = 52$ elements
  - ((Ace, Clubs), (Two, Clubs), ..., (King, Diamonds))


Combinations in an ordering system

- This specification refers to the following attributes:
  - Item validity (two values)
  - In-stock status (two values)
  - Pricing schemes (three values)
  - Delivery timeframe (three values)
  - Shipping (three values)
  - Customer credit status ("known")
  - Export control status (two values)

- This specification implies
  - Customer credit status ("unknown")
  - Results of customer credit check (approved, denied)
  - Shipping destination (foreign, domestic)

- One estimate of the number of tests needed for all combinations of these points of variation is
  - $2 \times 2 \times 3 \times 3 \times 2 \times 2 = 1,728$ potential tests
Introduction to Combinatorial Test Design

- The root cause analysis of many bugs shows they depend on a value of one variable (20%-68%). Most defects can be discovered in tests of the interactions between the values of two variables (65%-97%). Therefore, the majority of defects can be triggered by a single value or a combination of two values.

- To balance cost and risk, Combinatorial Test Design (CTD) selects a subset of tests that covers all the interactions of variables at some level of interaction (pairs, three-way, etc.)

Table 1. Number of variables involved in triggering software faults

<table>
<thead>
<tr>
<th>Vars</th>
<th>Medical Devices</th>
<th>Browser</th>
<th>Server</th>
<th>NASA GSC</th>
<th>Network Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>66</td>
<td>29</td>
<td>42</td>
<td>58</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>76</td>
<td>70</td>
<td>93</td>
<td>85</td>
</tr>
<tr>
<td>3</td>
<td>99</td>
<td>95</td>
<td>89</td>
<td>98</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>97</td>
<td>96</td>
<td>100</td>
<td>98</td>
</tr>
<tr>
<td>5</td>
<td>99</td>
<td>99</td>
<td>98</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [http://cs.cse.lsu.edu/SNS/acts/ffi.html](http://cs.cse.lsu.edu/SNS/acts/ffi.html)

- The CTD algorithm finds a small test plan that covers 100% of a requested interaction level.

---

Combinatorial Test Design at IBM

- Problems being addressed
  - I don't have enough resource to execute everything, how do I select a good set of tests?
  - We run hundreds of tests but still have escapes.
  - We don’t know how to measure the effectiveness of our tests.
  - Our regression buckets have grown huge, we run them constantly, have millions of results.

- And what did we learn??
A good model of the test space is key

- Defining the goal of *this* set of tests is absolutely necessary
  - When it was unclear what attributes and values were necessary it was typically because we lost sight of the goal

- Example scopes
  - Verify that create contract creates valid contracts.
  - Verify that search results properly reflect the search rules that are in place.
  - Verify that the workload driving the product executes properly in supported environments.
  - Verify that the product installs correctly.

---

**Coverage Example**

<table>
<thead>
<tr>
<th>Before CTD</th>
<th>After CTD</th>
</tr>
</thead>
<tbody>
<tr>
<td># Test Cases</td>
<td>97</td>
</tr>
<tr>
<td>% Coverage</td>
<td>20%</td>
</tr>
</tbody>
</table>

**Efficiency Example**

<table>
<thead>
<tr>
<th>Before CTD</th>
<th>After CTD</th>
</tr>
</thead>
<tbody>
<tr>
<td># Test Cases</td>
<td>45</td>
</tr>
<tr>
<td>Defects</td>
<td>19%</td>
</tr>
<tr>
<td>Productivity (defects/cases)</td>
<td>10%</td>
</tr>
</tbody>
</table>

**Effectiveness and Efficiency Examples**
A good model of the test space is key

- Defining the test space is an iterative process where you build and refine the model by reading available documentation and talking with subject matter experts. Following this process
  - resulted in a better understanding of the product by the test teams
  - drove better low level design because the structured discussions led to many “What if...” questions
- Reviewing the models instead of long lists of tests resulted in shorter, more effective reviews. Teams reported
  - review times shortened by as much as 50%
  - entire team was more engaged in the discussion
  - models were updated as a result of the review, both adding and removing attributes and values
- Using the model decreased the learning curve for new additions to the team
- Updating the model for a new release easier than updating traditional test scripts
- Modeling functional testing was interesting and exciting work

A good model of the test space is key

- Defining the test space showed the complexity of feature creep
  - Baseline is the combinations that result from the selected level of interaction
  - Can show increase in numbers of combinations in order to cover new functions or configuration

Adding Two Carriers added 6 Tests (12 to 18)
Our existing tests had significant redundancy and gaps

- Existing test suites represented high level of investment
  - had a great deal of pride of authorship
- Redundancy are tests that have all combinations covered by other tests
- Gaps are missing combinations
- In existing test suite represented by chart, approx. 40% of tests were redundant, removing redundancy allowed adding tests to fill gaps while actually reducing the overall number of tests

Example of Redundancy and Gaps

Features:
- X with values (1,2,3), Y with values (4,5,6), Z with values (7,8,9)

Coverage requirement is 2-way LOI

If this is the existing test suite:

a) 1 4 7
b) 2 4 7
c) 2 4 9
d) 1 4 9 Redundant (All pairs covered)
e) 1 5 9

Gaps (Missing pairs)

(1,6) (1,8) (3,4) (3,9) ...

The generated test cases enable an automated test framework

- Modeling the test space in terms of attributes and values separated WHAT to test from HOW to test it, i.e. requirements versus implementation

<table>
<thead>
<tr>
<th>Availability</th>
<th>Payment</th>
<th>Carrier</th>
<th>Delivery Schedule</th>
<th>ExpControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>Credit</td>
<td>FedEx</td>
<td>TotalShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueDate</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>FedEx</td>
<td>ExpDate</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>Credit</td>
<td>FedEx</td>
<td>TotalShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueDate</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>Credit</td>
<td>FedEx</td>
<td>TotalShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueShipment</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>PayLess</td>
<td>UPS</td>
<td>DueDate</td>
<td>True</td>
</tr>
</tbody>
</table>

Test Case Layer
- workflow oriented
  - reusable with modification
- Read rows of inputs from table to pass to task layer

Task Layer
- reflect business logic
  - application specific
- Implements the specific interface to the application

appObjects Layer
- highly reusable
- Wrapper classes for the GUI widgets
A good deployment model results in successful pilots and quicker time to value

- **Pattern**
  - Start with a small pilot; allowed the team to learn and improve
  - Gain management support; kept focus when problems arose
  - Establish pilot goals; agreement on what the pilot included allowed teams to achieve the full value
  - Establish support mechanism, workshops to learn the basics, weekly meetings to stay on track, forums to ask questions; kept teams moving forward and focused

- **Anti-Pattern**
  - Attempting to start too big causes too much change at once
  - Lack of management support results in redirect when problems arise
  - Starting without a defined end goal results in teams that stop early and do not achieve the full value
  - Giving teams a link to the tool leads to misuse and incorrect results, frustration, negative feedback

---

Summary of our experience with CTD

- **Combinatorial Test Design provided great expected results:**
  - Reduced number of tests
  - Reduced number of configurations
  - Higher number of defects per test cases
  - More even coverage of the test space (no gaps or duplication)

- **Unexpected benefits clearly made CTD a winner:**
  - Better coverage metrics
  - More thorough understanding of the test space
  - More effective test case reviews
  - Better understanding of our existing test cases
  - Test cases that feed maintainable automation
Questions?