Specification-based and boundary testing

Maurício F. Aniche

M.FinavaroAniche@tudelft.nl

We need more **systematic** and **rigorous** ways to test software!

SPECIFICATION

Requirements
Models

STRUCTURAL

Structure
(e.g., source code)

SPECIFICATION

Requirements
Models

STRUCTURAL

Structure
(e.g., source code)

A package should store a total number of kilos. There are small bars (1 kilo each) and big bars (5 kilos each). We should calculate the number of small bars to use, assuming we always use big bars before small bars.

Return -1 if it can't be done.

Input: small bars, big bars, total.

What tests would you design?

Inspired by https://codingbat.com/prob/p191363

Examples

Small	Big	Total	Small bars to use (output)
1	3	11	1
7	3	20	5
2	0	1	1
10	2	10	0

Partitions based on the requirements

A package should store a **total number of kilos**. There are **small bars** (1 kilo each) and **big bars** (5 kilos each). We should calculate the number of small bars to use, assuming **we always use big bars before small bars**.
Return -1 if it can't be done.

- Identify representative classes
  - Only small bars
  - Only big bars
  - Small + big bars
  - Not enough bars
  - Not from the specs: invalid number
- Choose representative values
- Exploit the knowledge to identify trouble-prone regions of the input space.

1) The total is higher than the amount of small and big bars.

Ex: small = 1, big = 1, total = 10

3) Need for big and small bars.

Ex: small = 5, big = 3, total = 17

2) Only big bars.

Ex: small = 5, big = 3, total = 10

4) Only small bars.

Ex: small = 4, big = 2, total = 3

5) Invalid input.

Ex: small = 4, big = 2, total = -1

The category-partition method
(in a nutshell)

- Identify the parameters
- The characteristics of each parameter
  - From the specs
  - Not from the specs
- Add constraints (minimize)
  - Remove invalid combinations
  - Reduce number of exceptional behaviors
- Generate combinations

SPECIAL ARTICLE

THE CATEGORY-PARTITION METHOD FOR SPECIFYING AND GENERATING FUNCTIONAL TESTS

A method for creating functional test suites has been developed in which a test engineer analyzes the system specification, writes a series of formal test specifications, and then uses a generator tool to produce test descriptions from which test scripts are written. The advantages of this method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of the tests by annotating the tests specification with constraints.

THOMAS J. OSTRAND and MARC J. BALCER

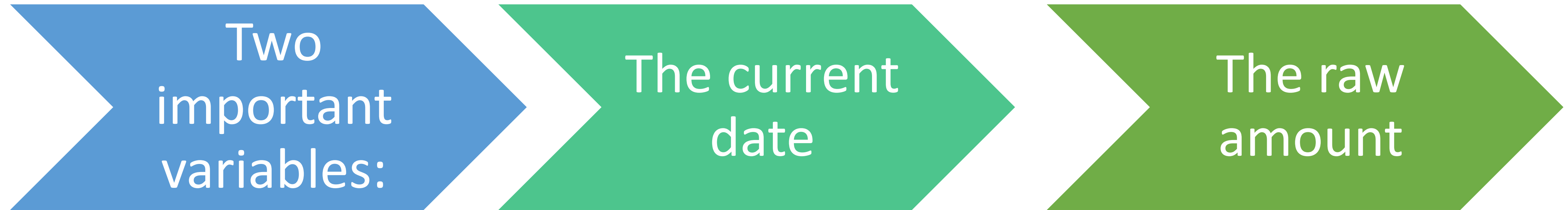
The goal of functional testing of a software system is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification. To achieve this goal requires first, that tests be executed for all of the system's functions, and second, that the tests be designed to maximize the chances of finding errors in the software. Although a particular method or testing group may emphasize one or the other, these two aspects of testing are mutually complementary, and both are necessary for maximally productive testing. It is not enough merely to "cover all the functionality"; the tests must be aimed at the most vulnerable parts of the implementation. For functional testing, this implies testing boundary conditions, special

Functional tests can be derived from the software's specifications, from design information, or from the code itself. All three test sources provide useful information, and none of them should be ignored. Code-based tests relate to the modular structure, logic, control flow, and data flow of the software. They have the particular advantage that a program is a formal object and it is therefore easy to make precise statements about the adequacy or thoroughness of code-based tests. Design-based tests relate to the programming ab[stractions], data structures, and algorithms used to co[n]struct the software. Specification-based tests relate d[i]rectly to what the software is supposed to do, and therefore are probably the most intuitively appealin[g] type of functional tests.

We offer a discount during Christmas.
If it’s Christmas, we give a 15% discount in the total amount of the order.
If it’s not Christmas, no discount.

We offer a discount during Christmas.
If it's Christmas, we give a 15% discount in the **total amount** of the order.
If it's not **Christmas**, no discount.

Category Partition



- The current date
- The raw amount

- Christmas
- Not Christmas

- Positive number
- Zero
- Negative number

Constraints

- The current date
- The raw amount

- Christmas
- Not Christmas

- Positive number
- Zero
- Negative number [exceptional]

Combinations / Tests

- Christmas
  - Positive number
  - Zero
  - Negative number
- Not Christmas
  - Positive number
  - Zero

Partitions are representative classes of our program, and they guide me throughout the testing phase.

Partitions

2) Only big bars.

Ex: small = 5, big = 3, total = 10
Ex: small = 5, big = 4, total = 15
Ex: small = 5, big = 5, total = 20
Ex: small = 5, big = 6, total = 25
…

Which one should I pick? All?

Equivalent partitions

- If the case is really representative and independent, any instance should do.
  - *ISTQB definition: "A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification:".*
- We should try to reduce the human cost.
  - Having lots of (repeated) tests increase the cost.

```java
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total)
        return -1;
    return total;
}
```

Need for big and small bars

Try this input: (5, 3, 17).

Output should be: 2

Your tests were not enough!
If I provide
small = 2
big = 3
total = 17

It returns -1,
but it should be 2!

```java
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total)
        return -1;
    return total;
}
```

Can you find the bug?

Try the input: (2, 3, 17).

```java
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small <= total)
        return -1;
    return total;
}
```

```java
public int calculate(int small, int big, int total) {
    int maxBigBoxes = total / 5;
    int bigBoxesWeCanUse =
        maxBigBoxes < big ? maxBigBoxes : big;

    total -= (bigBoxesWeCanUse * 5);

    if(small < total)
        return -1;
    return total;
}
```

1) The total is higher than the amount of small and big bars.

Ex: small = 1, big = 1, total = 10

2) Only big bars.

Ex: small = 5, big = 3, total = 10

4) Only small bars.

Ex: small = 4, big = 2, total = 3

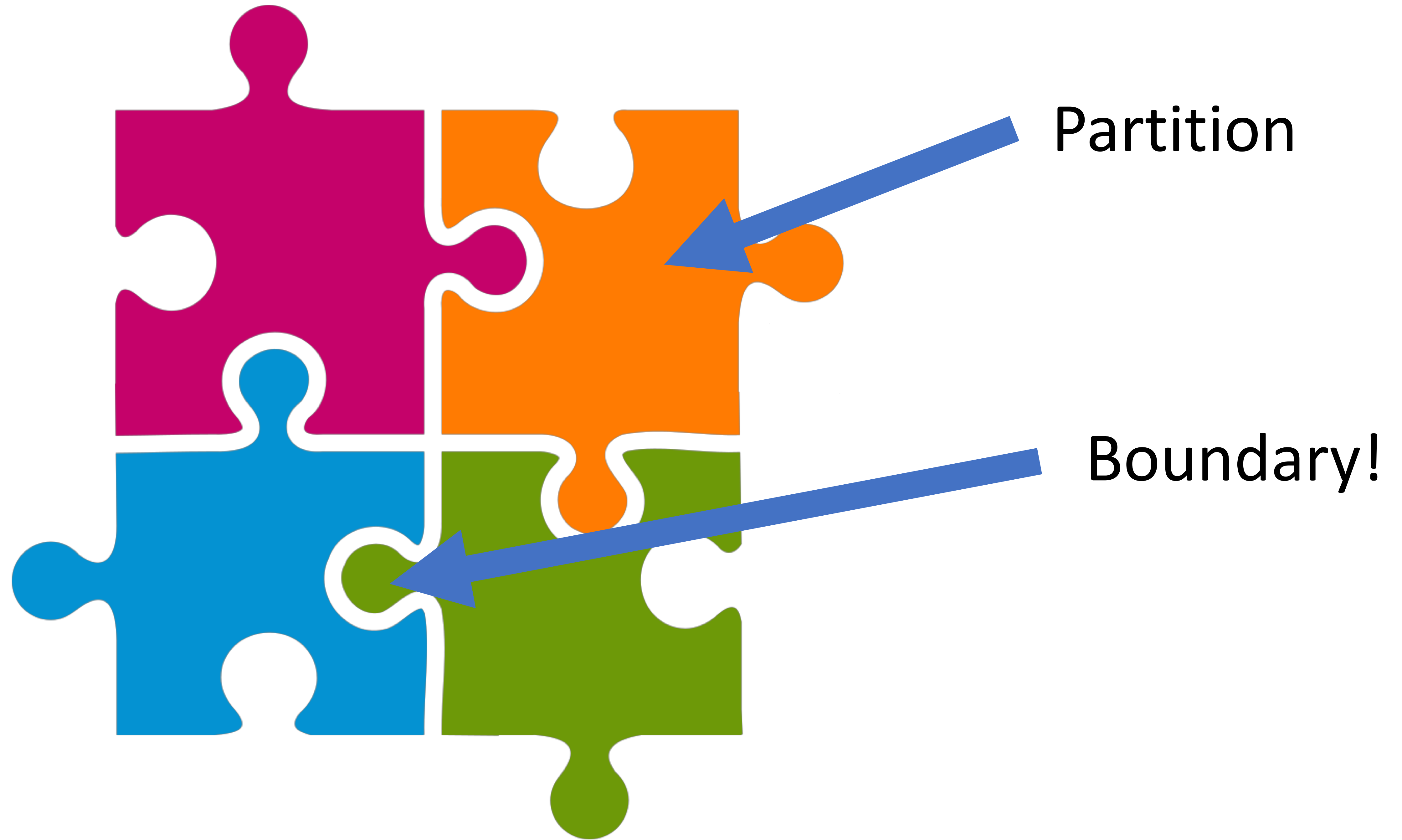
3) Need for big and small bars.

Ex: small = 5, big = 3, total = 17

5) Invalid input.

Ex: small = 4, big = 2, total = -1

But.. But… Does this mean that thinking about partitions is not enough? :(


Partition
Boundary!

Hmm, with these inputs,
small = 2 is on the boundary
of the required number of
small bars!

small = 2
big = 3
total = 17

Small = 1, not possible
Small = 2, possible
Small = 3, possible

Hmm, ok, let me think about the boundaries for each of these partitions, and do some **boundary testing**.

1) The total is higher than the amount of small and big bars.
2) Only big bars.
3) Need for big and small bars.
4) Only small bars.

The total is higher than the amount of small and big bars.

Ex: small = 1, big = 1, total = 10

small = 1, big = 1, **total = 5**, = 0
small = 1, big = 1, **total = 6**, = 1

small = 1, big = 1, **total = 7**, = -1
small = 1, big = 1, **total = 8**, = -1

Only big bars.

Ex: small = 5, big = 3, total = 10

small = 5, **big = 0**, total = 10, = -1

small = 5, **big = 1**, total = 10, = 5

small = 5, **big = 2**, total = 10, = 0

small = 5, **big = 3**, total = 10, = 0

Need for big and small bars.

Ex: small = 5, big = 3, total = 17

small = 0, big = 3, total = 17, = -1

small = 1, big = 3, total = 17, = -1

small = 2, big = 3, total = 17, = 2

small = 3, big = 3, total = 17, = 2

All big bars

---

small = 2, big = 3, total = 14, = -1

small = 3, big = 3, total = 14, = -1

small = 4, big = 3, total = 14, = 4

small = 5, big = 3, total = 14, = 4

Not all big bars

Only small bars.

Ex: small = 4, big = 2, total = 3

small = 4, big = 2, total = 3, = 3

small = 3, big = 2, total = 3, = 3

small = 2, big = 2, total = 3, = -1

small = 1, big = 2, total = 3, = -1

- If the score is between 100 and 200, the player gets 50 bonus points.
- If the total ordering is above $100.00, shipping costs is $5.00.
- …

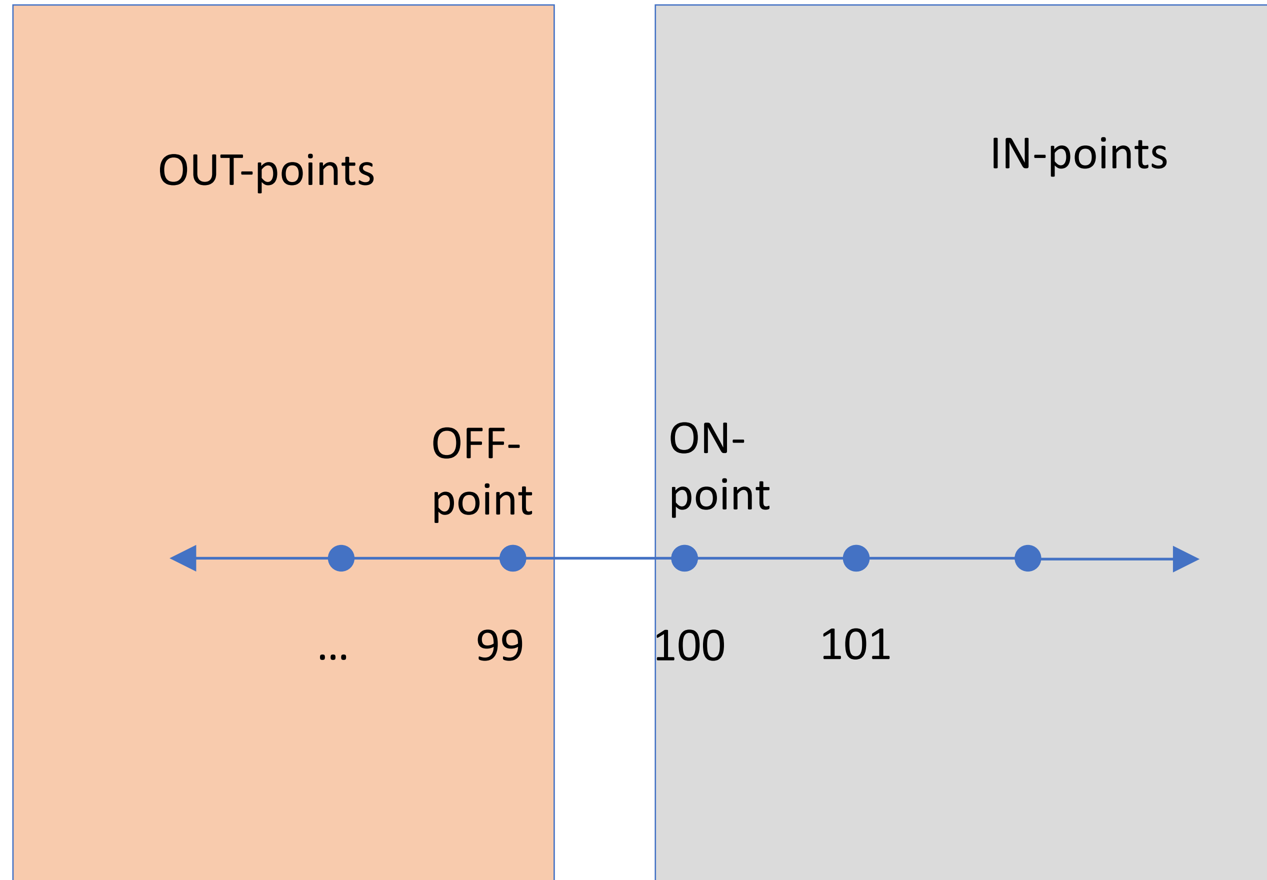
Let me test it!
I do **“off-by-one”** mistakes all the time!

Boundary analysis

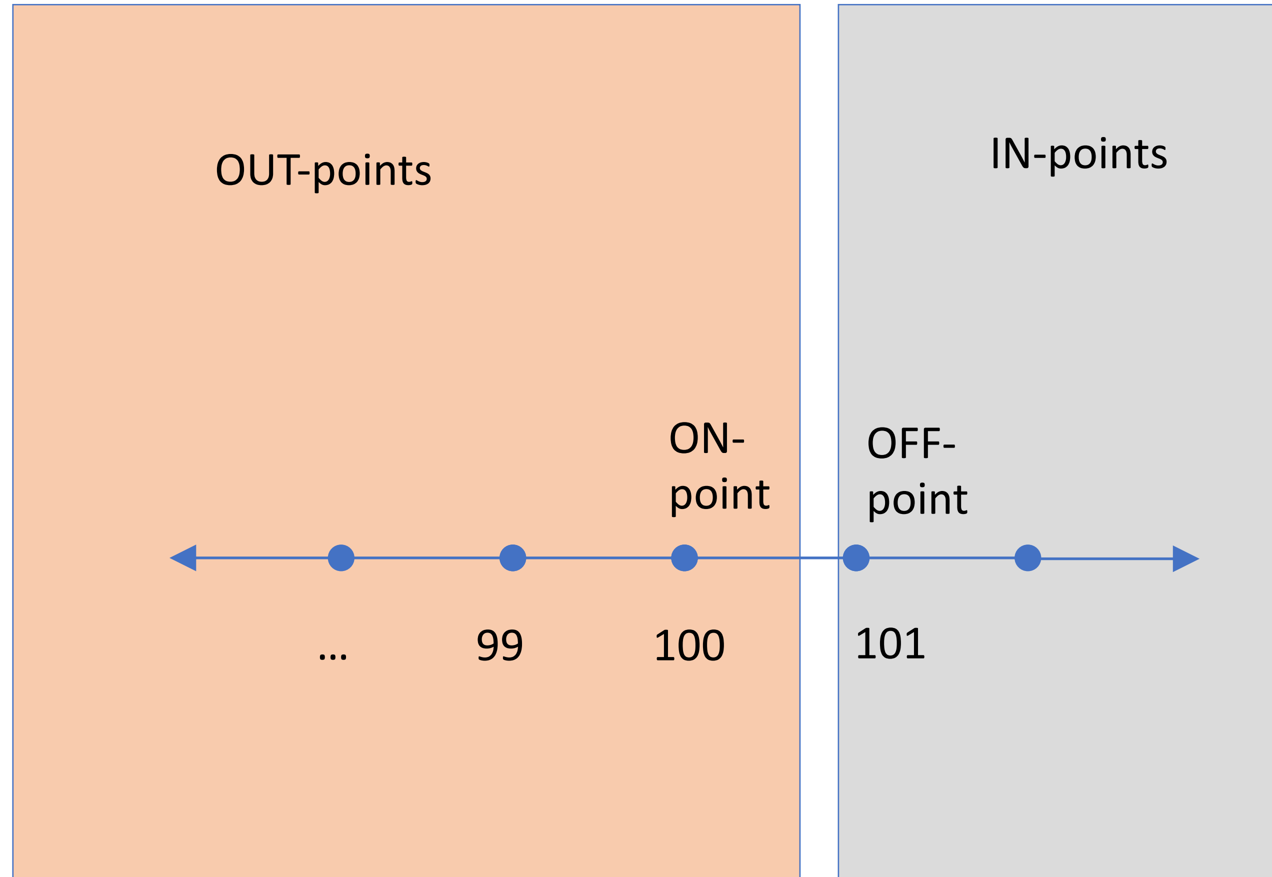
Let me test it!
I do **“off-by-one”**
mistakes a

- If the score i ween
  100 an yer
  ing is
  .00,
  ng costs is $5.00.

X >= 100



X > 100



score >= 100

- **On point:** *Exactly on boundary*
- **In point:** *Makes the condition true*
- **Out point:** *Makes the condition false*
- **Off point:**
  - *Flips the outcome for on point* **and**
  - *Is as close to boundary as possible*

On is 100; In is e.g. 200;
Out is e.g. 50; Off is 99.

Multiple boundaries?

A Simplified Domain-Testing Strategy

BINGCHIANG JENG
Sun Yat-Sen University
and
ELAINE J. WEYUKER
New York University

A simplified form of domain testing is proposed that is substantially cheaper than previously proposed versions, and is applicable to a much larger class of programs. In particular, the traditional restrictions to programs containing only linear predicates and variables defined over continuous domains are removed. In addition, an approach to path selection is proposed to be used in conjunction with the new strategy.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Domain testing, software testing

1. INTRODUCTION

Domain testing is a fault-based software-testing strategy proposed by White and Cohen [1978]. Testers have frequently observed that subdomain boundaries are particularly fault-prone and should therefore be carefully checked. Domain testing was proposed as a relatively sophisticated form of boundary value testing, and is applicable whenever the input domain is subdivided into subdomains by the program's decision statements.

In this paper, a simplified version of domain testing is described which removes several limitations associated with earlier domain-testing strategies. In particular, our strategy is applicable to arbitrary types of predicates, detects both linear and nonlinear errors, for both discrete and continuous variable spaces. In addition, we will show that our new technique requires much smaller test suites than earlier versions, and will argue that its effectiveness is comparable to, and in some cases superior to, the others.

... and by NASA grant NAG-1-1238

Simplified domain-testing strategy

- Handle boundaries independently
- For each boundary, *pick on and off point*
- While testing one boundary, use varying *in points* for the remaining boundaries.
- Use *domain matrix.*

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on						
		off						
	<= 10	on						
		off						
	typical	in						
y	>= 1.0	on						
		off						
	typical	in						

Boundary conditions for ' x > 0 && x <= 10 && y >= 1.0'

			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on						
		off						
	<= 10	on						
		off						
	typical	in						
y	>= 1.0	on						
		off						
	typical	in						

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	2390.2		

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	2390.2		

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"

			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	2390.2		

JUnit for multiple data points?

- 6 test cases, each with three values

  < x-value, y-value, outcome >

- For each test case:
  - Check that with given inputs method produces desired output.

- Hand-code in loop?

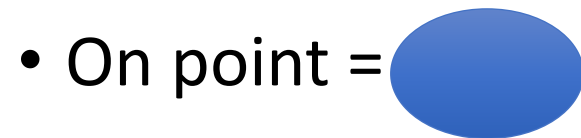
Use JUnit 5
@ParameterizedTest

```java
/**
 * Verify that squares at key positions are properly set.
 * @param x Horizontal coordinate of relevant cell.
 * @param y Vertical coordinate of relevant cell.
 */
@ParameterizedTest
@CsvSource({
    "0, 0",
    "1, 2",
    "0, 1"
})
void testSquareAt(int x, int y) {
    assertThat(board.squareAt(x, y)).isEqualTo(grid[x][y]);
}
```

Open & Closed Boundaries

Closed boundary

- Score >= 200

- On point = 

- Off point =

Open boundary

- Score > 200

- On point =

- Off point =

Multiple choice

Which of the following statements is true

A. An out point cannot also be an on point
B. The in point is included in the set of on points
C. If the on point is an in point, the off point is an out point.
D. An out point can never be an off point

Multiple choice

Which of the following statements is true

A. An out point cannot also be an on point
B. The in point is included in the set of on points
C. **If the on point is an in point, the off point is an out point.**
D. An out point can never be an off point

Chapter 7

Boundary conditions: the Correct way

[C]orrect: Conformance

- Many data elements must conform to a specific format.
  - Example: e-mail (always name@domain).
- Test when your input is not in conformance with what is expected.

C[o]rrect: ordering

- The order of the data might influence the output.
- What happens if the list is ordered? Unordered?

Co[r]rect: range

- Inputs should usually be within a certain range.
  - Example: Age should always be greater than 0 and smaller than 120.
  - In most programming languages, basic types give you more than you need, e.g., int when you just need a number between 1-100.

Cor[r]ect: reference

- When testing a method, consider:
  - What it references outside its scope
  - What external dependencies it has
  - Whether it depends on the object being in a certain state
  - Any other conditions that must exist

Corr[e]ct: existence

- Does something really exist? What if it doesn’t?

Corre[c]t: cardinality

- Off-by-one errors
- Loops:
  - Zero
  - One
  - Many

Correc[t]: time

- Ordering in time
  - What happens if I forget to invoke a() before b()?
- Timeouts
- Date/Time operations
  - Should we use UTC? GMT?
- Concurrency

Random vs Partition testing

- Would it be better to simply test random inputs?
- Would it be more effective or less effective?

Random testing

- If generating random inputs is cheap, then even with a small budget (e.g., 1 day), we'd generate millions of tests. A human would only generate a few.
- Random testing is an ineffective way to find singularities in the large input space.

Partition testing

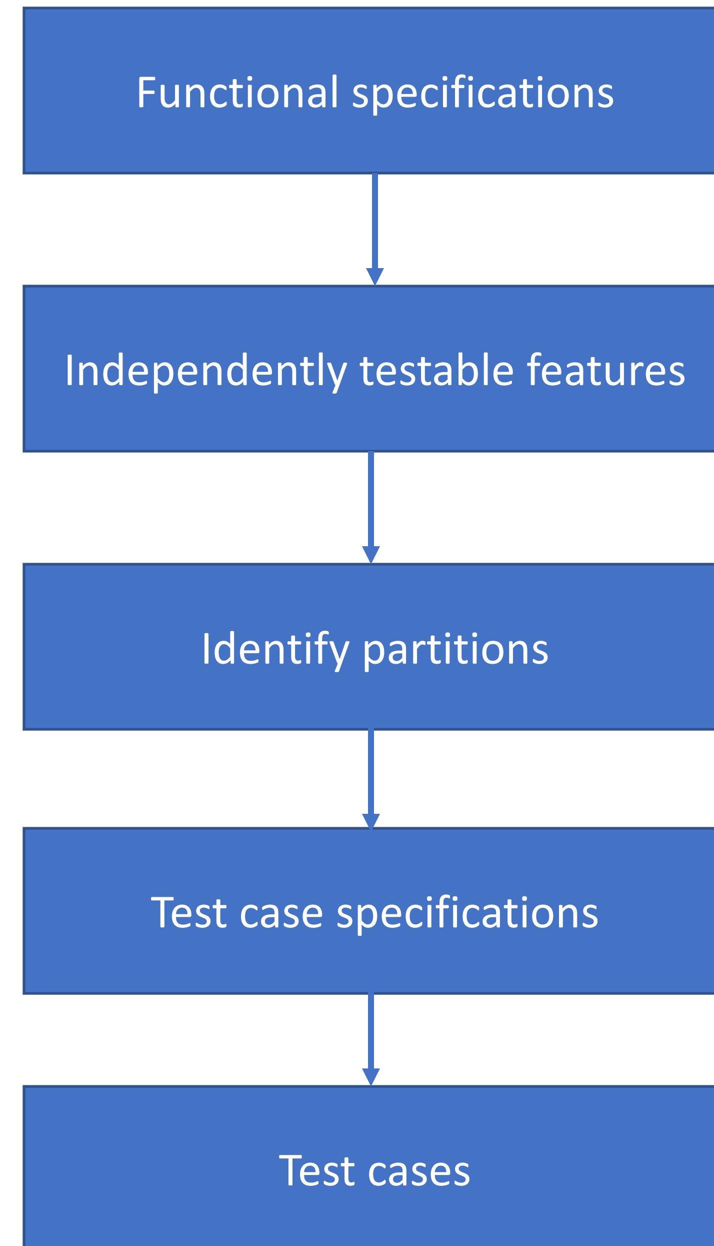
- Test designers usually exploit some knowledge of application semantics to choose samples that are more likely to include "special" or trouble-prone regions of the input space.
- Partition testing is more expensive than random testing.

Given a fixed budget, the optimum may not lie in only partition testing or only random testing, but in some mix that makes use of available knowledge.

Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.

Functional testing in large systems

- Functional specifications can be large and complex. Partition the specifications into features that can be tested separately.
- An ITF is a feature that can be tested independently of other functionalities of the software.
- Given an ITF, apply partition testing.
- Instantiate (concrete and executable) test cases.



Adapted from Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.

Summary

- Functional (specification-based) tests
- Partition testing
- The Category-Partition method
- Equivalence class
- Boundary tests and boundary analysis
- Multiple boundary tests
- The CORRECT way
- Random testing vs Partition tests

References

- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Chapter 7 of Pragmatic Unit Testing in Java 8 with Junit. Langr, Hunt, and Thomas. Pragmatic Programmers, 2015.
- Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.