

# Input space partitioning

- The ISP approach
- Identifying input parameters
- Modelling the input domain
- Deriving test inputs
- Coverage criteria

# Input State Partitioning

- Testing is about choosing elements from the input space of the software being tested
- The input domain is defined in terms of the possible values that the input parameters can have.
- The input parameters can be method parameters and global variables, objects representing current state, or user-level inputs to a program.
- The input domain is partitioned into regions assumed to contain equally useful values from a testing perspective, and values are selected from each region.

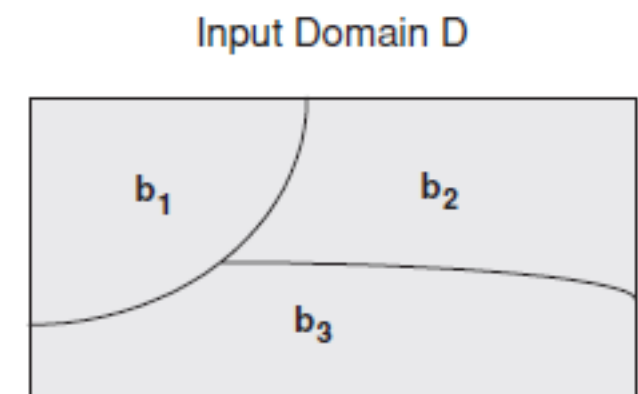


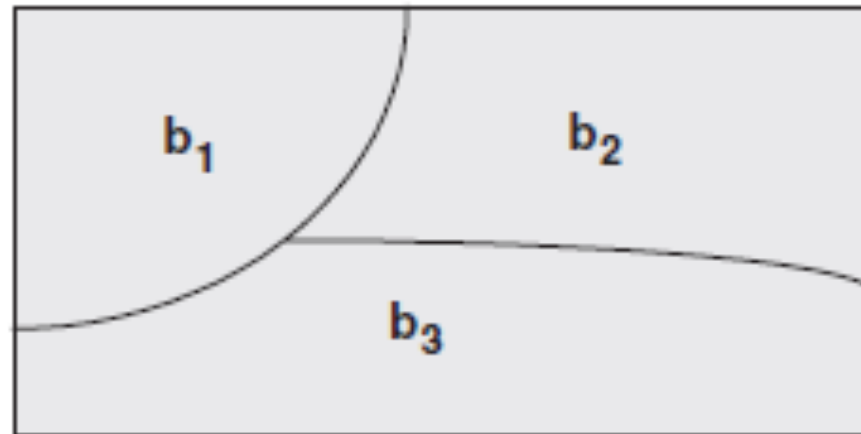
Figure 4.1. Partitioning of input domain  $D$  into three blocks.

# Advantages

- It is fairly easy to get started because it can be applied with no automation and very little training.
- The tester does not need to understand the implementation; everything is based on a description of the inputs.
- It is simple to "tune" the technique to get more or fewer tests.

# Partitioning a given domain

Input Domain D



- A partition  $q$  defines a set of equivalence classes, which we simply call **blocks**.
- The blocks are *pairwise disjoint*, that is

$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

and together the *blocks cover the domain D*, that is  $\bigcup_{b \in B_q} b = D$

# Input space partitioning (ISP)

- **Step 1. Identify the input parameters** for the SUT.
  - **Step 2. Model the input domain** by identifying one or more characteristics for the input domain. Each characteristic defines blocks that *must* partition the input space.
  - **Step 3. Apply some criterion** over the characteristics, defining a set of test requirements.
  - **Step 4.** Derive test inputs (test cases).
- 
- Eg, characteristic 'order of file', i.e., is it sorted, inverse sorted, or arbitrary?
    - This is not a valid partitioning: an empty file would belong to all partitions
  - Better: define a characteristic pair (is sorted ascending?, is sorted descending?)
    - Now an empty file only belongs to partition (true,true)

# Identifying input parameters

- Mechanic analysis and automation can help here
- To test a method, input parameters may include:
  - Method parameters
  - Instance fields
  - Global class variables.
  - User-level or system-level I/O (e.g., user actions, input files)

# Example \_ class Time

```
public final class Time {
    private int hours;
    private int minutes;
    /**
     * Constructor.
     * @param h hours (from 0 to 23)
     * @param m minutes (from 0 to 59)
     * @throws IllegalArgumentException if arguments are invalid
     */
    public Time(int h, int m) throws IllegalArgumentException { ... }
    public int getHours() { ... }
    public int getMinutes() { ... }
    /**
     * Advance one minute to the current time.
     */
    public void tick() { ... }
    public boolean equals(Object obj) { ... }
    public String toString() { ... }
}
```

Input domain modelling:

Time()	h m (method parameters)
toString(), tick()	hours minutes (instance fields)
equals()	hours minutes obj (instance fields + parameters)

# Identifying characteristics and blocks

- Each characteristic should represent a "meaningful" feature of the input domain.
- Blocks of a characteristic should have distinctive values.
- **Guidelines** - look for:
  - ✱ "common use" values
  - ✱ boundary values
  - ✱ "invalid use" values
  - ✱ relations between input parameters
- **... and always ensure** that blocks
  - ✱ do not overlap
  - ✱ effectively define a partition



# Characteristics and blocks for class Time

- Suppose that `toString()` must return a string in a 12-hour AM/PM form: "hh:mm [AM|PM]", e.g., "01:05 PM", "12:10 AM".
- Possible characteristics:
  - ✱ **AM/PM** - Earlier or later than midday. Blocks:
    - $AM = 0 \leq h < 12$
    - $PM = 12 \leq h < 24$
  - ✱ **hz** - Need '0' prefix for hour representation. Blocks:
    - $yes = h \in \{1, 2, \dots, 9, 13, 14, \dots, 21\}$
    - $no = h \in \{0, 10, 11, 12, 22, 23\}$
  - ✱ **mz** - Need a '0' prefix for minute representation. Blocks:
    - $yes = 0 \leq m < 10$
    - $no = 10 \leq m < 60$

# Input domain modelling

1. Identification of testable functions
2. Identification all of the parameters that can affect the behavior of a given testable function
3. Modelling the input domain (the key creative engineering step):  
characteristics → blocks → values
4. Built test inputs: a tuple of values, one for each parameter

# Interface-based vs functionality-based IDM

- The interface-based approach develops characteristics directly from input parameters to the program under test.
- The functionality-based approach develops characteristics from a functional or behavioural view of the program under test.

# Interface-Based Input Domain Modelling

- Considers each particular parameter in isolation
- Pros:
  - Easy to identify characteristics
  - Easy to translate the abstract tests into executable test cases.
- Cons
  - Not all the information available to the test engineer is reflected in the interface domain model
  - Some parts of the functionality may depend on combinations of specific values of several interface parameters

# Functionality-Based Input Domain Modelling

- Identify characteristics that correspond to the intended functionality of the SUT
- Allows the tester to incorporate some semantics or domain knowledge into the modelling.
- Preconditions and postconditions are excellent sources
- It is preferable for the test engineer to use specifications or other documentation instead of program code to develop characteristics
- Use *domain knowledge* about the problem, not the implementation

# Example \_ findElement()

```
/**
 * @return true if element is in the list, false otherwise
 * @throws NullPointerException if list or element is null
 */
public boolean findElement (List<E> list, Object element)
```

- Interface-based approach. Characteristics:
- **list** is null. Blocks:
  - $b1 = \text{True}$
  - $b2 = \text{False}$
- **list** is empty. Blocks:
  - $b1 = \text{True}$
  - $b2 = \text{False}$

# Example \_ continued

- Functional-based approach
  - Requires more thinking on the part of the test engineer, but can result in better tests, since the model includes domain knowledge
- Characteristics:
- number of occurrences of **element** in **list**. Blocks:
  - $b1 = 0$
  - $b2 = 1$
  - $b3 = \text{More than 1}$
- **element** occurs **list** in list. Blocks:
  - $b1 = \text{True}$
  - $b2 = \text{False}$

# Example

```
public static Type triangleType(int[] sides) {  
    ...  
    throw new IllegalArgumentException();  
    ...  
    return Type.SCALENE; ...  
    return Type.ISOSCELES; ...  
    return Type.EQUILATERAL;  
}
```

What is more appropriate, an interface or functional approach?



# Example

1. Test case for a valid scalene triangle
2. Test case for a valid equilateral triangle
3. Three test cases for valid isosceles triangles ( $a=b$ ,  $b=c$ ,  $a=c$ )
4. One, two or three sides has zero value (five cases)
5. One side has a negative value
6. Sum of two numbers equals the third (e.g. 1,2,3) is invalid b/c not a triangle (tried with three permutations  $a+b=c$ ,  $a+c=b$ ,  $b+c=a$ )
7. Sum of two numbers is less than the third (e.g. 1,2,4) (three permutations)
8. Wrong number of values (too many, too few)

# Example

Specific values per case

1. {5,6,7}
2. {15,15,15}
3. {3,3,4; 5,6,6; 7,8,7}
4. {0,1,1; 2,0,2; 3,2,0; 0,0,9; 0,8,0; 11,0,0; 0,0,0}
5. {3,4,-6}
6. {1,2,3; 2,5,3; 7,4,3}
7. {1,2,4; 2,6,2; 8,4,2}
8. {2,4; 4,5,5,6}

# Exercise I

```
/**
 * @param s1 One set; null treated as an empty set.
 * @param s2 Another set; null treated as an empty set.
 * @return A (non null) Set equal to the intersection of s1 and s2
 */
public Set intersection (Set s1, Set s2) {
    ...
}
```

- Characteristic 1: Type of **s1**
  - **s1** = null
  - **s1** = {}
  - **s1** has at least 1 elem

1. Interface/Functionality?
2. Complete?
3. Disjoint?

- Characteristic 2: Relation between **s1** and **s2**
  - **s1** and **s2** represent the same set
  - **s1** is a subset of **s2**
  - **s2** is a subset of **s1**
  - **s1** and **s2** do not have any elements in common

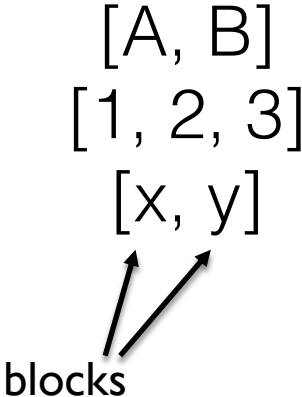
# Combination Strategies Criteria

- How should we consider multiple partitions at the same time?
- What combination of blocks should we choose values from?
- The most obvious choice is to choose all combinations
- Using all combinations will be impractical when more than two or three partitions are defined

# All Combinations Coverage (ACoC)

- All combinations of blocks from all characteristics must be used.
- Yields a unique test for each combination of blocks for each partition
- The number of tests is the product of the number of blocks for each partition

Partitions	Tests
[A, B]	(A, 1, x) (A, 1, y)
[1, 2, 3]	(A, 2, x) (A, 2, y)
[x, y]	(A, 3, x) (A, 3, y)
	(B, 1, x) (B, 1, y)
	(B, 2, x) (B, 2, y)
	(B, 3, x) (B, 3, y)



# Each Choice Coverage (ECC)

- One value from each block for each characteristic must be used in at least one test case
- Not very effective tests

Partitions	Tests
[A, B]	(A, 1, x)
[1, 2, 3]	(B, 2, y)
[x, y]	(A, 3, x)

# Pair-Wise Coverage (PWC)

- A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- I.e., all pairs of blocks must be tested
- The tests with ‘—’ mean that any block can be used.

Partitions	Tests
[A, B]	(A, 1, x) (A, 2, x)
[1, 2, 3]	(A, 3, x) (A, —, y)
[x, y]	(B, 1, y) (B, 2, y)
	(B, 3, y) (B, —, x)

# T-Wise Coverage (TWC)

- A value from each block for each group of  $t$  characteristics must be combined.
- $t = 1$  — equivalent to ECC
- $t = 2$  — equivalent to PWC
- $t = \text{number of characteristics}$  — equivalent to ACoC



# ACoC coverage for toString()

test	hours	minutes	expected result	(am/pm, hz, mz) coverage
1	10	30	"10:30 AM"	(am, n, n)
2	10	0	"10:00 AM"	(am, n, y)
3	9	30	"09:30 AM"	(am, y, n)
4	9	0	"09:00 AM"	(am, y, y)
5	22	30	"10:30 PM"	(pm, n, n)
6	22	0	"10:00 PM"	(pm, n, y)
7	21	30	"09:30 PM"	(pm, y, n)
8	21	0	"09:00 PM"	(pm, y, y)

**PWC:** Tests 2, 3, 5 and 8 would suffice (or 1,4,6 and 7).

**ECC:** Tests 1 and 8 would suffice (or 2+7, 3+6, 4+5).

# Exercise 2

```
public class Time {
    /**
     * Constructor.
     * @param h hours (from 0 to 23)
     * @param m minutes (from 0 to 59)
     * @throws IllegalArgumentException if arguments are invalid
     */
    public Time (int h, int m) throws IllegalArgumentException {...}
    public int getHours () {...}
    public int getMinutes () {...}
    /**
     * Advance one minute to the current time.
     */
    public void tick () {...}
    public boolean equals (Object o) {...}
    public String toString () {...}
}
```

1. Define 2 characteristics for the input domain of method **tick()**.
2. Derive tests that satisfy **a) ECC b), PWC, c) ACoC**  
Do the same for **equals()**.

# Exercise 3

```
public class FixedCapacityStack {
    public FixedCapacityStack(int capacity) { ... }
    public int capacity() { .... }
    public int size() { .... }
    public boolean isEmpty() { ... }
    public boolean isFull() { ... }
    public void push (Object o)
        throws IllegalStateException { ... }
    public Object pop ()
        throws IllegalStateException { ... }
    public Object peek ()
        throws IllegalStateException { ... }
}
```

- Identify meaningful blocks for the following characteristics of input domain:
  - ✱ CAP: capacity of the stack
  - ✱ SIZE: size of the stack
- Identify test requirements for ECC, PWC and method call sequences required for the corresponding test cases in order to test methods pop() and push().
- Write a JUnit test class for the test cases.

# ISP - "base choices"

- PWC and TWC combine blocks "blindly".
- ... some combinations may be more relevant than others.
- A "base choice" reflects the most "important" block for each characteristic.
- The base choice can be the simplest, the smallest, the first in some ordering, or the most likely from an end-user point of view.

# Base Choice Coverage (BCC)

- What is the most important block for each partition?
- Define a base choice by selecting a single block for each characteristic.
- Define remaining test requirements by varying one block for each characteristic at a time.

Partitions	Base choice	Tests
[A, B]		(A, 1, x)
[1, 2, 3]		( <b>B</b> , 1, x)
[x, y]	(A, 1, x)	(A, <b>2</b> , x)
		(A, <b>3</b> , x)
		(A, 1, <b>y</b> )

# Time.toString() example

- Recall our **Time.toString()** example and the characteristics we used for IDM:
  - **am/pm** Earlier or later than midday.
  - **hz** Need of '0' prefix for hour representation.
  - **mz** Need of a '0' prefix for minute representation.
- Let us consider the base choice (**am, n, n**).

# Time.toString() example

test	hours	minutes	exp. result	(am/pm, hz, mz) coverage
<b>1</b>	<b>10</b>	<b>30</b>	<b>"10:30 AM"</b>	<b>(am, n, n)</b>
2	22	30	"10:30 PM"	( <b>pm</b> , n, n)
3	9	30	"09:30 AM"	(am, <b>y</b> , n)
4	10	0	"10:00 AM"	(am, n, <b>y</b> )

For a base choice of **(am, n, n)**, the remaining BCC requirements are:

(pm, n, n)

(am, y, n)

(am, n, y)

# Exercise 4

```
public class FixedCapacityStack {  
    public FixedCapacityStack(int capacity) { ... }  
    public int capacity() { .... }  
    public int size() { .... }  
    public boolean isEmpty() { ... }  
    public boolean isFull() { ... }  
    public void push (Object o)  
        throws IllegalStateException { ... }  
    public Object pop ()  
        throws IllegalStateException { ... }  
    public Object peek ()  
        throws IllegalStateException { ... }  
}
```

- Consider the characteristics of Exercise 3 and apply the BCC criterion.
- Subjective question: what is the base choice that seems to be more "meaningful"?

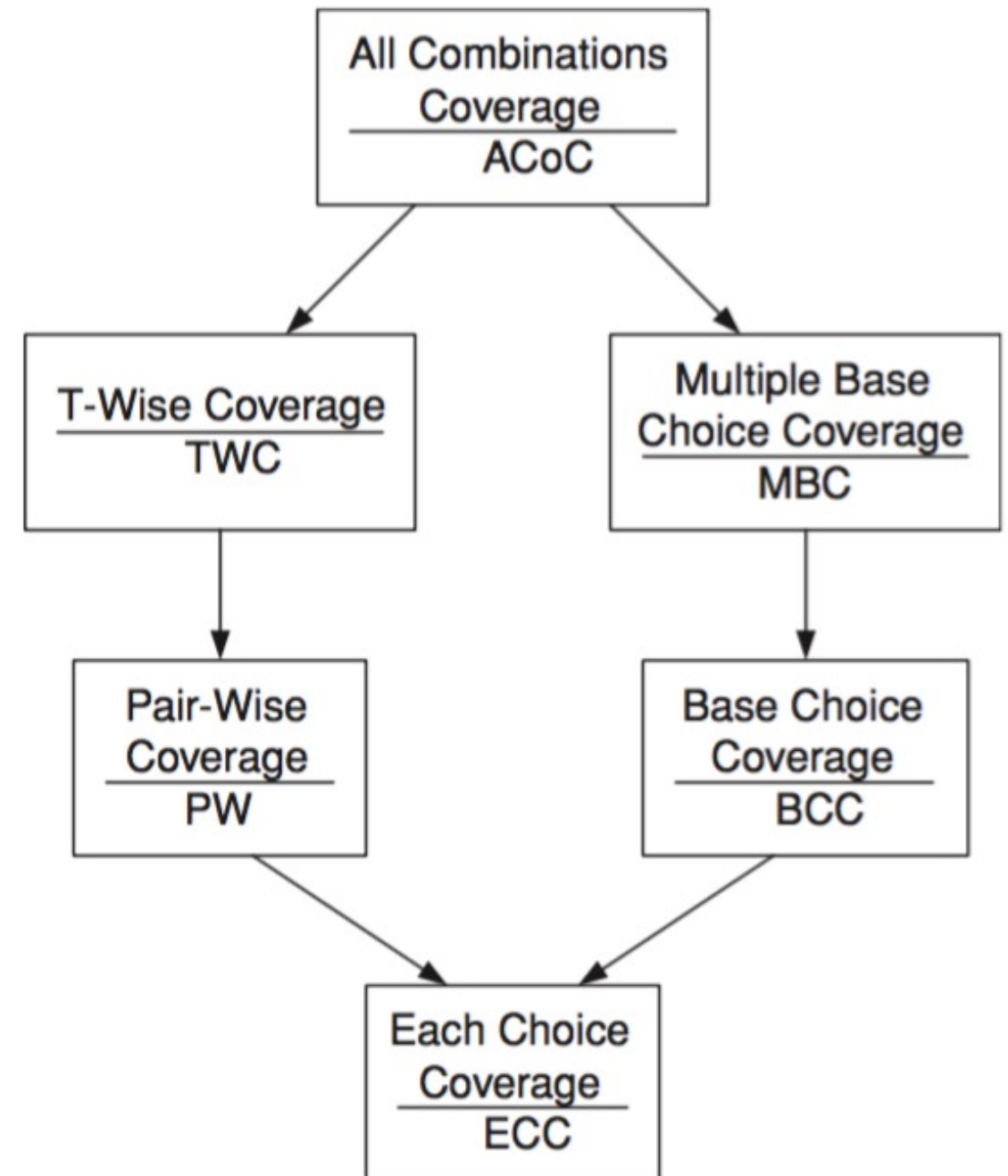


# Multiple Base Choices (MBCC)

- Use  $n > 1$  base choices.
- Note: select the base choices such that there is little overlapping between requirements
- Test requirements are defined for each base choice as in BCC.

# Subsumption relations among Input Space Partitioning criteria

- ACoC may not be practical: leads to many test requirements; possibility of also many infeasible requirements
- On the other extreme, ECC may be too simplistic
- PWC, TWC combine characteristics blindly
- BCC/MBCC try to be "smart" in picking meaningful blocks



# Exercise 5

```
enum Type {SCALENE, ISOSCELES, EQUILATERAL}

/**
 * @param a The length of one side
 * @param b The length of another side
 * @param c The length of the third side
 * @throws IllegalArgumentException when one of the sides
 *         is non-positive or the three sides do not form a
 *         triangle
 */
static Type triangleType (int a, int b, int c) {...}
```

Identify characteristics and blocks

1. Define a base choice and apply BCC to derive tests.
2. Select one more base choice and apply MBCC to derive additional tests.

# Case study: Metro de Lisboa

- Let us apply the ISP criteria for finding paths between metro stations.
  - Source code in maven project (package `sut.metro`)
- First model:
  - Two simple and exhaustive characteristics ...
  - Too many test requirements ...
- Second model:
  - Let us think about meaningful characteristics ...

There are 4 lines  
and 49 stations.



# Code (MetroLine)

```
public enum MetroLine {  
    AMARELA( "Odivelas", "Senhor Roubado", "Ameixoeira", "Lumiar",  
        "Quinta das Conchas", "Campo Grande", "Cidade Universitária",  
        "Entre Campos", "Campo Pequeno", "Saldanha", "Picoas",  
        "Marquês de Pombal", "Rato"),  
  
    AZUL( "Amadora Este", "Alfornelos", "Pontinha", "Carnide",  
        "Colégio Militar/Luz", "Alto dos Moinhos", "Laranjeiras",  
        "Jardim Zoológico", "Praça de Espanha", "São Sebastião", "Parque",  
        "Marquês de Pombal", "Avenida", "Restauradores", "Baixa-Chiado",  
        "Terreiro do Paço", "Santa Apolónia"),  
  
    VERMELHA( "São Sebastião",  
        "Saldanha", "Alameda", "Olaias", "Bela Vista", "Chelas", "Olivais",  
        "Cabo Ruivo", "Oriente", "Moscavide", "Encarnação", "Aeroporto"),  
  
    VERDE(  
        "Telheiras", "Campo Grande", "Alvalade", "Roma", "Areeiro", "Alameda",  
        "Arroios", "Anjos", "Intendente", "Martim Moniz", "Rossio",  
        "Baixa-Chiado", "Cais do Sodré")  
;  
...
```

# Code (MetroDB)

```
public class MetroDB {  
  
    public List<String> findPath(String src, String dst) {  
        ...  
    }  
}
```

Campo Grande -> Baixa-Chiado :

[Campo Grande, Cidade Universitária, Entre Campos, Campo Pequeno, Saldanha, Picoas, Marquês de Pombal, Avenida, Restauradores, Baixa-Chiado]

- `findPath` finds the shortest path between two stations
- Example above: notice there is a line change at Marquês de Pombal (implicit: take linha AMARELA at Campo Grande until Marquês de Pombal, then take linha AZUL until Baixa-Chiado).

# Model 1 \_ Characteristics and Criteria

- **SRC** \_ Source, 49 stations/blocks
  - ✱ Aeroporto
  - ✱ Alameda
  - ✱ ...
  - ✱ Terreiro do Paço
- **DST** \_ Destination, 49 stations / blocks
  - ✱ ... (same blocks as above)
- **ECC**: 49 + 49 test requirement. Tests fail to capture some meaningful aspects of the metro line (**what aspects?**)
- **PWC** (=ACoC in this case): exhaustive but lead to too many requirements ( $49 \times 49 = 2401$ )



# A bug in MetroLine ...

```
public enum MetroLine {  
    ...  
    VERDE(  
        "Telheiras", "Campo Grande", "Alvalade", "Roma", "Areeiro",  
        /* "Alameda", */  
        "Arroios", "Anjos", "Intendente", "Martim Moniz", "Rossio",  
        "Baixa-Chiado", "Cais do Sodré")  
    ;  
    ...  
}
```

Olaias -> Anjos : [Olaias, Alameda, Arroios, Anjos] correct path

Olaias -> Anjos : [Olaias, Alameda, Saldanha, Campo Pequeno, Entre Campos, Cidade Universitária, Campo Grande, Alvalade, Roma, Areeiro, Arroios, Anjos]

!!!!!!

- Alameda is a “hub” station that connects the VERDE and the VERMELHA lines.
- This simple bug leads to **1733 incorrect paths out of 2401!**
- It is significant to consider one or more characteristics that take “hub stations” in consideration (line change?)

# A bug in MetroDB...

```
List<String> l = line.getStations();
String stationA = l.get(0);
for (int i = 1; i < l.size(); i++) {
    String stationB = l.get(i);
    addConnection(stationA, stationB);
    addConnection(stationB, stationA);
    stationA = stationB;
}
```

```
List<String> l = line.getStations();
String stationA = l.get(0);
for (int i = 1; i < l.size(); i++) {
    String stationB = l.get(i-1);
    addConnection(stationA, stationB);
    addConnection(stationB, stationA);
    stationA = stationB;
}
```

São Sebastião -> Rato : [São Sebastião, Parque, Marquês de Pombal, Rato] correct path

São Sebastião -> Rato : [] no path found

- Stations at the end of the line are significant!
- This simple bug leads to **736 / 2401 incorrect results!**

# Model 2 \_ Characteristics and Criteria

- LINE\_CHANGES (account for hub stations)
  - 0: Path requires no line changes
  - 1: Path requires one line change
  - > 1: Path requires more than one line change
- TERMINAL
  - BOTH: SRC and DST are terminal
  - SRC: SRC is terminal but DST is not
  - DST: DST is terminal but SRC is not
  - NONE: Neither SRC nor DST are terminal
- What else is relevant?

# Model 2 (cont.)

- Let us also consider the significant aspects of path length.
- LENGTH (account for hub stations)
  - ✱ **1**: Path has length 1 (i.e. [SRC,DST])
  - ✱ **2-8**: "average" trips
  - ✱ **> 8**: "long" trips
- Additionally we could (should) consider the type of LINE for SRC and DST, but what we have provides us with enough material for analysis.

# Summary

Characteristic	Blocks (partition)
line changes	0, 1, >1
terminal	both, src only, dst only, none
path length	1, 2-8, >8

# Analysis

- ECC :  $3 + 4 + 3 = 10$  requirements
- PWC:  $(3 \times 4) + (3 \times 3) + (4 \times 3) = 33$  requirements
- ACoC:  $3 \times 4 \times 3 = 36$  requirements
  - ECC leads to a simplistic coverage
  - PWC is better but will not be as accurate as ACoC (# reqs. are more or less the same, but they combine only 2 characteristics, not 3)
  - ACoC is "doable" in this case
- What about MBCC?
  - **Exercise:** Let us derive test cases for BCC and MBCC (2 base choices).
- **Homework:** Solve PWC and ACoC ...

# ECC \_ Test inputs for findPath()

src	dst	covered requirements
Campo Grande	Cidade Universitária	<b>LINE_CHANGES = 0</b> <b>TERMINAL = NONE</b> <b>LENGTH = 1</b>
Rato	Parque	<b>LINE_CHANGES = 1</b> <b>TERMINAL = SRC</b> <b>LENGTH = 2 to 8</b>
Avenida	Cais do Sodré	<b>LINE_CHANGES = 1</b> <b>TERMINAL = DST</b> <b>LENGTH = 2 to 8</b>
Amadora Este	Odivelas	<b>LINE_CHANGES = 2</b> <b>TERMINAL = BOTH</b> <b>LENGTH &gt; 8</b>

# BCC \_ Deriving test requirements

- If we take the base choice to be  
(LINE\_CHANGES=1, TERMINAL=NONE, LENGTH=2-8)

the test requirements are:

- ✱ (1, NONE, 2-8)
- ✱ (0, NONE, 2-8)
- ✱ (>1, NONE, 2-8)
- ✱ (1, SRC, 2-8)
- ✱ (1, DST, 2-8)
- ✱ (1, BOTH, 2-8)
- ✱ (1, NONE, 1)
- ✱ (1, NONE, >8)



# BCC \_ Test inputs for findPath()

src	dst	covered requirements
Rossio	Avenida	(1, NONE, 2-8)
Campo Grande	Entre Campos	(0, NONE, 2-8)
Rossio	Picoas	(>1, NONE, 2-8)
Rato	Parque	(1, SRC, 2-8)
Avenida	Cais do Sodré	(1, DST, 2-8)
Rato	Telheiras	(1, BOTH, 2-8)
Infeasible		(1, NONE, 1)
Sete Rios	Oriente	(1, NONE, >8)

# MBCC \_ Deriving test requirements

- Consider in addition to **(1, NONE, 2-8)** the following requirement **(LINE\_CHANGES=0, TERMINAL=SRC, LENGTH=2-8)**. Test requirements: those for BCC, plus:
  - \* **(0, SRC, 2-8)**
  - \* (1, SRC, 2-8)                      Duplicated!
  - \* (>1, SRC, 2-8)
  - \* (0, **NONE**, 2-8)                      Duplicated!
  - \* (0, **DST**, 2-8)
  - \* (0, **BOTH**, 2-8)
  - \* (0, SRC, 1)
  - \* (0, SRC, >8)

# MBCC \_ Test inputs for findPath()

src	dst	covered requirements
Rossio	Avenida	(1, NONE, 2-8)
Campo Grande	Entre Campos	(0, NONE, 2-8)
Rossio	Picoas	(>1, NONE, 2-8)
Rato	Parque	(1, SRC, 2-8)
Avenida	Cais do Sodré	(1, DST, 2-8)
Rato	Telheiras	(1, BOTH, 2-8)
Infeasible		(1, NONE, 1)
Sete Rios	Oriente	(1, NONE, >8)
Aeroporto	Oriente	(0, SRC, 2-8)
Cais do Sodré	Picoas	(>1, SRC, 2-8)
Oriente	Aeroporto	(0, DST, 2-8)
Infeasible (every line has > 8 stations)		(0, BOTH, 2-8)
Cais do Sodré	Baixa-Chiado	(0, SRC, 1)
Odivelas	Avenida	(0, SRC, >8)