Combinatorial Testing Part A

Dr Fani Deligianni,

Fani.Deligianni@glasgow.ac.uk

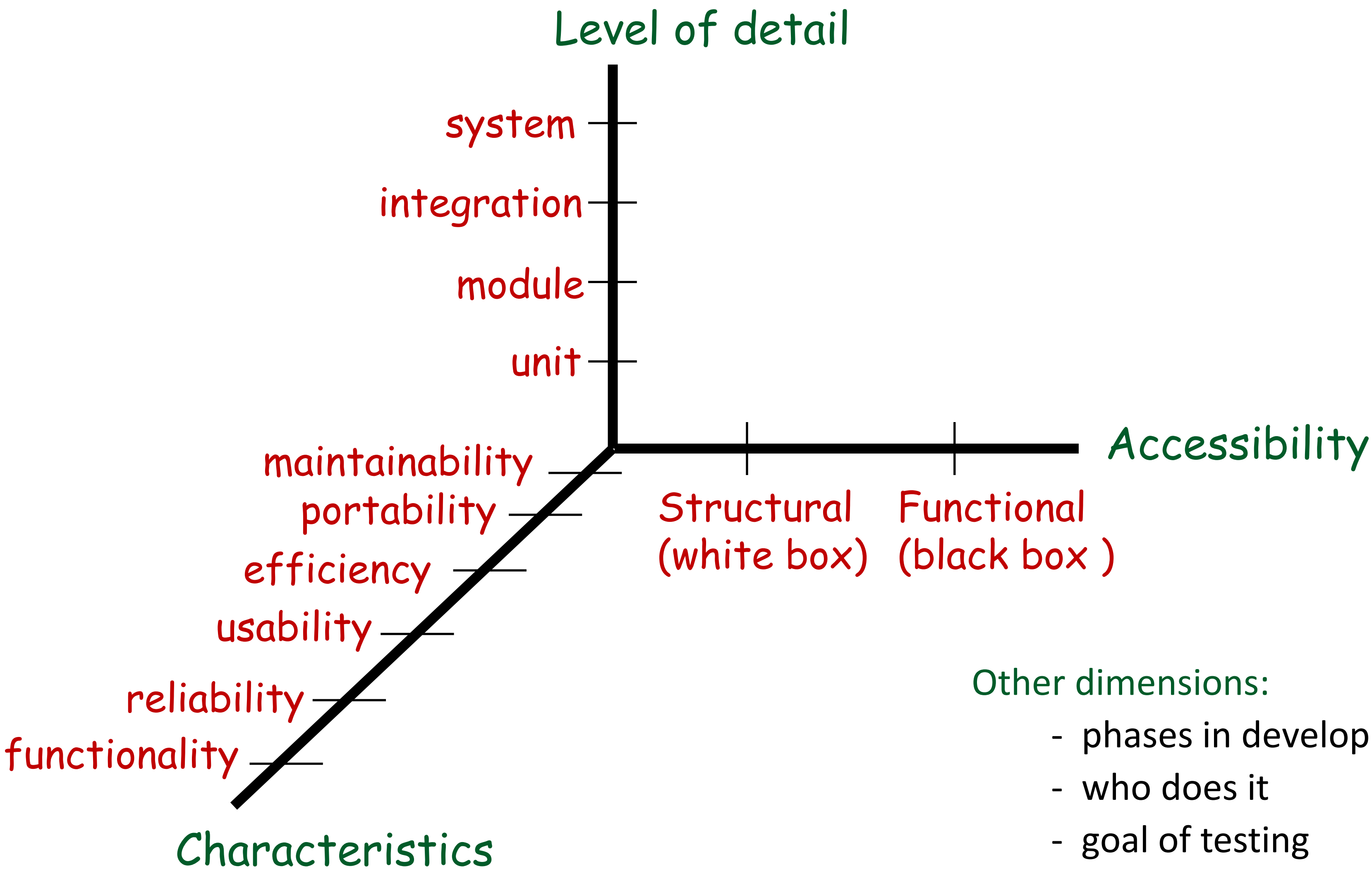
https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Learning Objectives

1. Understand and apply different software testing techniques:
   - Exhaustive Testing
   - Random Testing
   - Partition Testing
   - Boundary Value Testing

2. Understand the two main techniques in Input Domain Modelling
   - Learn and apply a systematic approach to functional testing
   - Apply the category-partition method for generation of test cases.

Dimensions of Testing – Previous Lecture



Other dimensions:

- phases in development
- who does it
- goal of testing
- . . . . .

Functional Specification

- Statement of the application functional and operational requirements.
- It serves as a contract between the developer and the customer for whom the system is being developed.

Functional (Black-Box) Testing

- Advantages:
  - Focus on the domain
  - No need for the code - Makes early test design possible
  - Catches logic effects
  - Applicable at all granularity levels

Functional Specification

NAME

grep - search a file for a pattern

SYNOPSIS

grep <pattern> <filename>

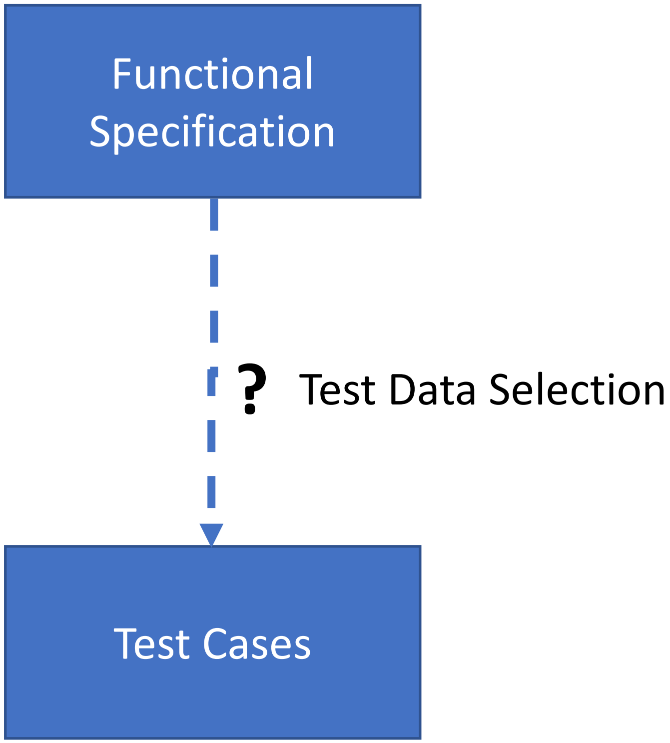
DESCRIPTION

The grep utility searches files for a pattern and prints all lines that contain that pattern on the standard output. A line that contains multiple occurrences of the pattern is printed only once.

The pattern is any sequence of characters. To include a blank in the pattern, the entire pattern must be enclosed in single quotes (‘). To include a quote sign in the pattern, the quote sign must be escaped (\’). In general, It is safest to enclose the entire pattern in single quotes ’…'.

Functional (Black-Box) Testing

- From Functional Specification to Test cases



Test Data Selection

- Addresses the problem of identifying relevant inputs for a software feature.

Test Data Selection

- Addresses the problem of identifying relevant inputs for a software feature.

Challenge:
How can we select meaningful inputs for the software

- Exhaustive Testing (brute force)
- Random Testing
- Systematic Approach (Combinational Testing)

Exhaustive Testing is Hard

```
int max(int x, int y) {
  if (x > y)
    return x;
  else
    return x;
}
```

- How long will it take to exhaustively test max (assuming 32 bit integers)
  - Total number of test cases: $2^{32} * 2^{32} = 2^{64} \simeq 10^{19}$
  - Assume that you can run 1 test per nanosecond, then we have 10^9 Tests/Sec
  - Total Time: 10^{10} seconds overall $\simeq$ **600 years**

Exhaustive Testing is Hard

```
int max(int x, int y) {
  if (x > y)
    return x;
  else
    return x;
}
```

- **Do bigger test sets help?**

  Test set {(x=3,y=2), (x=2,y=3)} will detect the error

Exhaustive Testing is Hard

```
int max(int x, int y) {
 if (x > y)
  return x;
 else
  return x;
}
```

- **Do bigger test sets help?**
  Test set {(x=3,y=2),(x=4,y=3),(x=5,y=1)} will not detect the error although it has more test cases

Exhaustive Testing is Hard

- Assume that the input for the max method was an integer array of size *n*
  - Then number of test cases: $2^{32 * n}$

- Assume that the size of the input array is not bounded
  - then number of test cases: ∞

Random Testing

- Use a random number generator to generate test cases
- Derive estimates for the reliability of the software using some probabilistic analysis
- Coverage is a problem

Random Testing

```
boolean isEqual(int x, int y) {
  if (x == y)
    return false;
  else
    return false;
}
```

- If we pick test cases randomly it is unlikely that we will pick a case where *x* and *y* have the same value

Random Testing

```
boolean isEqual(int x, int y) {
  if (x == y)
    return false;
  else
    return false;
}
```

- If *x* and *y* can take 2^{32} different values, there are 2^{64} possible test cases, and in 2^{32} of them x and y are equal
- Thus: The probability of picking a case where *x* is equal to *y* is 2^{-32}

Random Testing

```
boolean isEqual(int x, int y) {
   if (x == y)
      return false;
   else
      return false;
}
```

- It is not a good idea to pick the test cases randomly (with uniform distribution) in this case.
- Naive random testing will not be effective

Summary

- Exhaustive testing is impractical even for small functions
- Random testing may seems to have some advantages
- However random testing is also impractical

Combinatorial Testing Part B

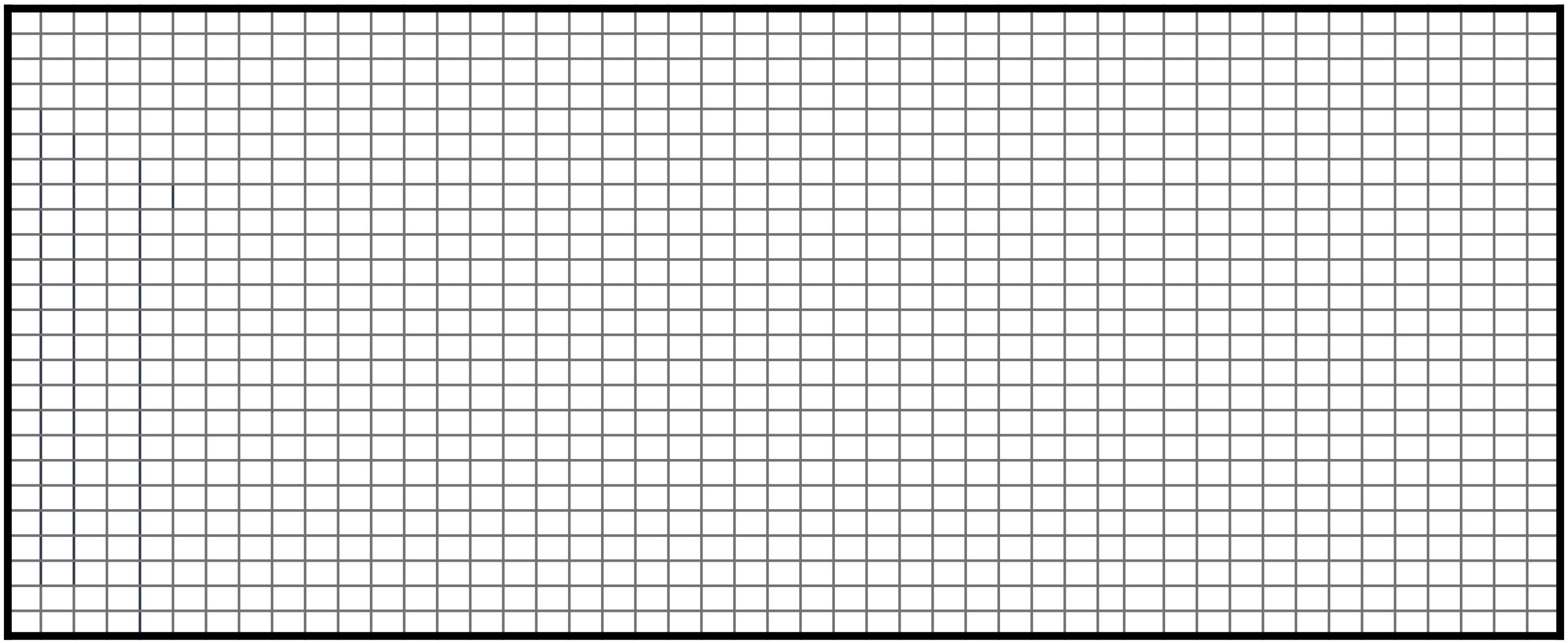
Dr Fani Deligianni,

Fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

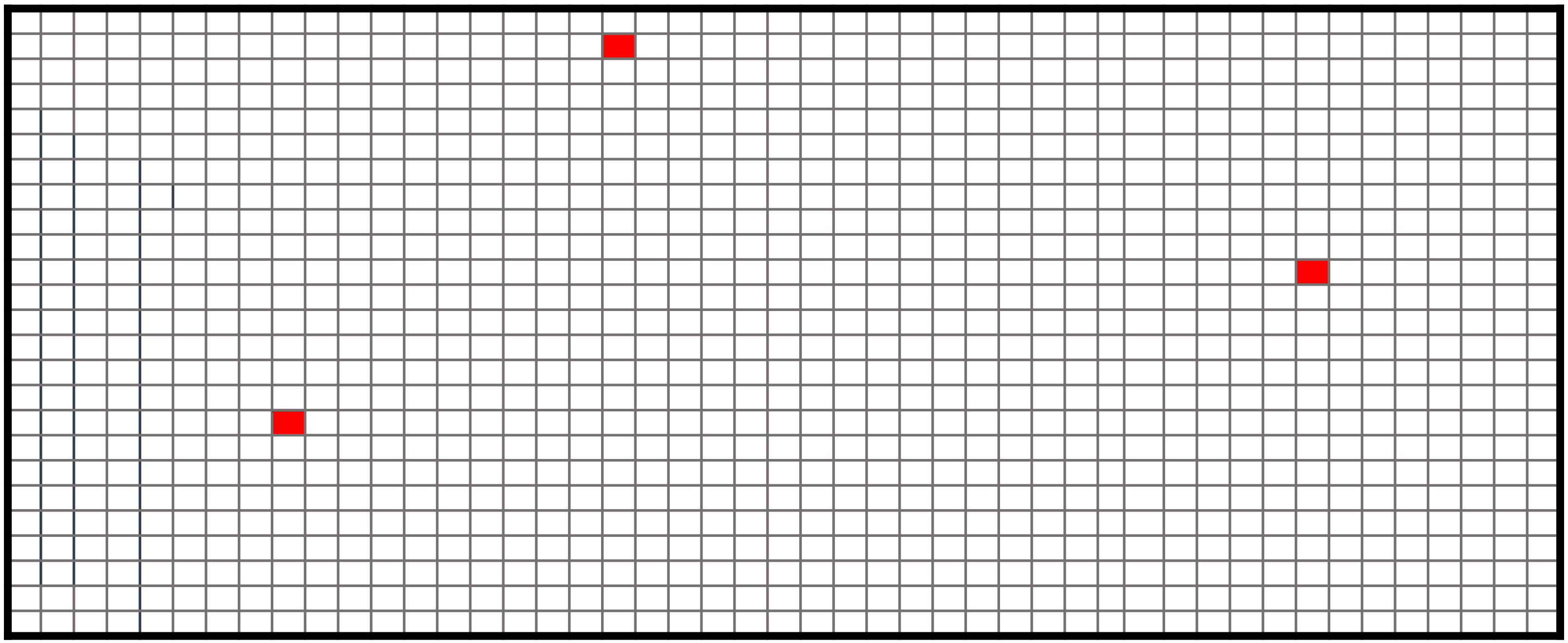
Partition Testing

Input domain for a software program



- Each square represents an input value

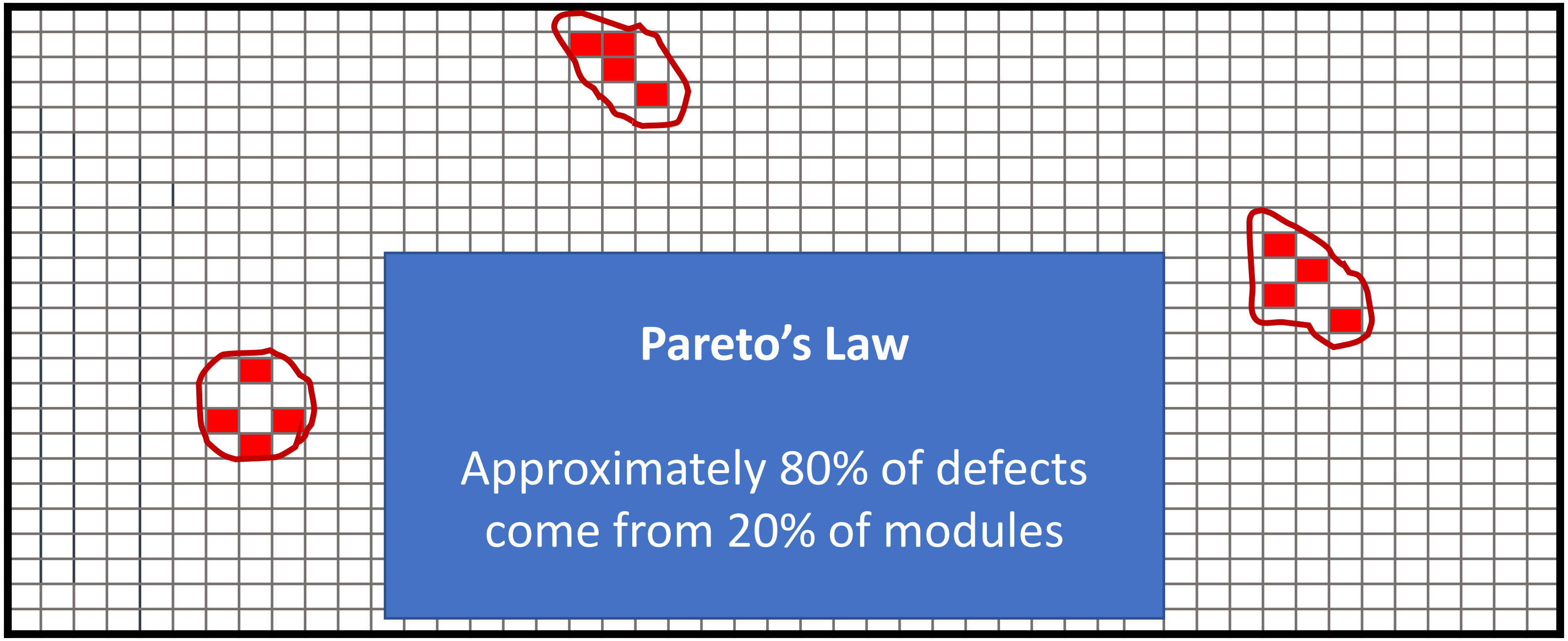
Partition Testing



- Failing inputs are generally sparse
- Randomly finding bug is like looking for a needle in a haystack

Partition Testing

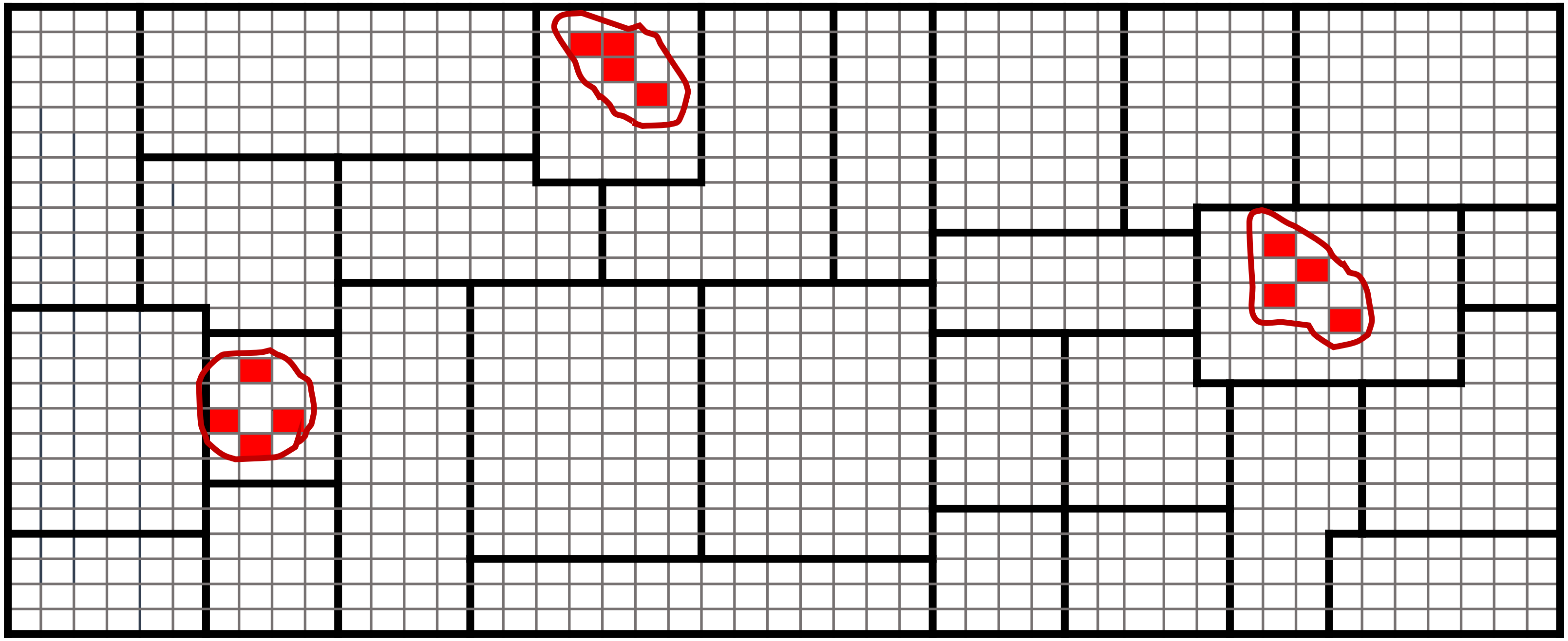
Input domain for a software program:



- However, failing inputs tend to be dense in some parts of the domain

Partition Testing

Input domain for a software program:



- Input domain tends to be naturally split into partitions
- Each partition is a block that defines a set of **equivalent subdomains**

Partition Testing

Equivalent Sub-domains

Calendar example:

- A function that takes any month of the year and returns the number of days.

Sub-domains:

- Month less than 1
- Month between 1-12
- Month greater than 12

Month Value **characteristics** used to define equivalent sub-domains

Partition Testing

Equivalent Sub-domains

Bus ticketing price example:

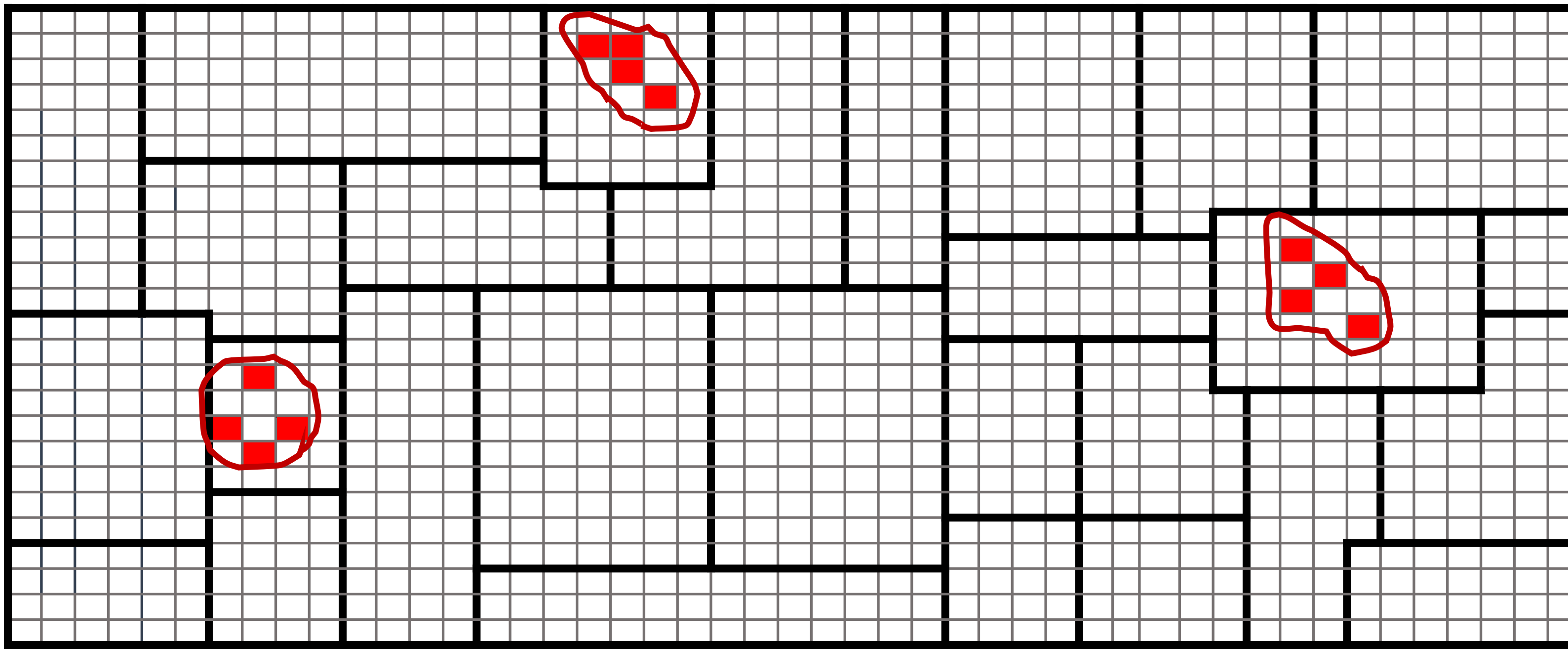
- Children under 6 years go free. Young people pay £2.50, Adults £5.00 and Senior Citizen pay £1.00.

Classes:
Age:0-5 ⟶ Price:0
Age:6-18 ⟶ Price:2.50
Age:19-64 ⟶ Price:5.00
Age:65-infinity ⟶ Price:1

age value **characteristic** used to define equivalent classes

Partition Testing

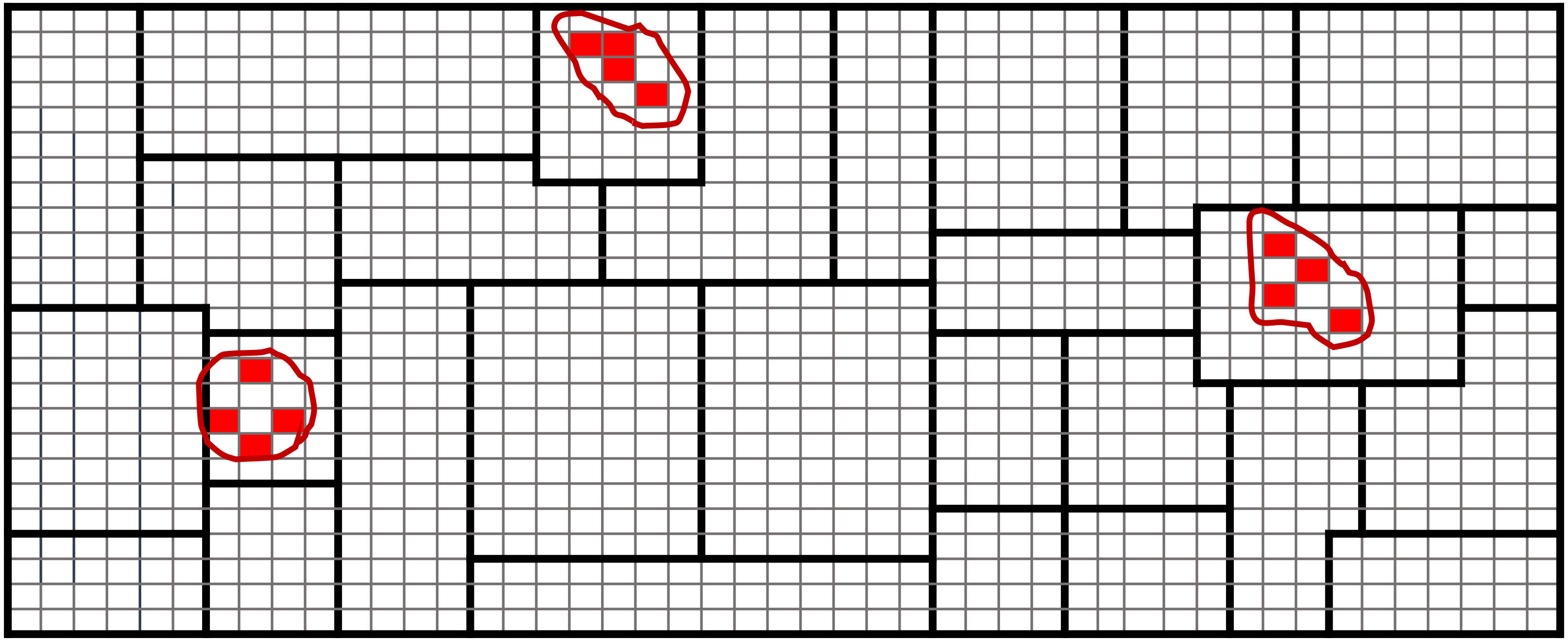
Input domain for a software program:



- So a partition is usually based on some **characteristic** of the program and the program's inputs

Partition Testing

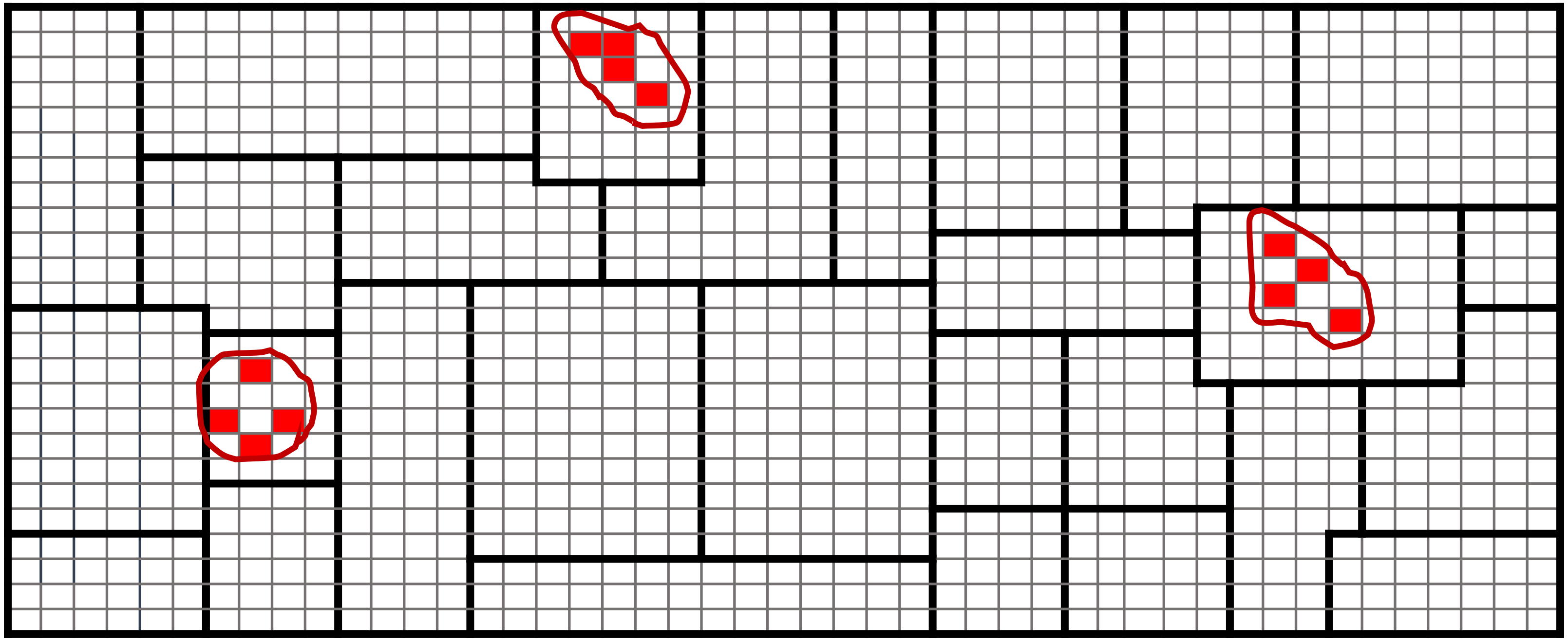
Input domain for a software program:



- Other possible **characteristic** examples:
  - Input *X* is (null, not null)
  - Order of file *F* (sorted, inverse sorted, arbitrary)
  - Min separation distance of two aircraft (>10 km, >20 km)

Partition Testing

Input domain for a software program:



- Formally, a partition must satisfy two properties:
  1. **The partition must cover the entire domain (completeness)**
  2. **The blocks must not overlap (disjoint)**

Combinatorial Testing
Part C

Dr Fani Deligianni,

Fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Input Domain Model

- Example:
  1. consider the characteristic "order of file F ". This could be used to create the following (defective) partitioning:
     - Order of file *F*
       – *b*1 = Sorted in ascending order
       – *b*2 = Sorted in descending order
       – *b*3 = Arbitrary order

This is a defective and invalid partitioning because:

- If the file is of length 0 or 1, then the file will belong in all three blocks. That is, the blocks are **not disjoint**.

Input Domain Model

- Example:
  - Order of file F (defective)
    – b_1 = Sorted in ascending order
    – b_2 = Sorted in descending order
    – b_3 = Arbitrary order

1. File F sorted ascending (valid)
   – b_1 = True
   – b_2 = False

2. File F sorted descending (valid)
   – b_1 = True
   – b_2 = False

files of length 0 or 1 are only in the True block for both characteristics.

Input Domain Model

- Approaches:
  1. **Interface-based input domain modelling:**
     - Develops ***characteristics*** directly from input parameters to the program under test
     - Typically each input parameter is considered in isolation.
     - **Advantage**: Relatively easy to identify characteristics
     - **Disadvantage**: Not all information is reflected in the interface, and testing some functionality may require parameters in combination

Input Domain Model

Interface-based input domain modelling

- *Example*:

Q: Define two interface-based characteristics for list, including blocks and values in the function below:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
// else returns true if element is in the list, false otherwise
```

Input Domain Model

Interface-based input domain modelling

- *Example*:

Q: Define two interface-based characteristics for list, including blocks and values in the function below:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
// else returns true if element is in the list, false otherwise
```

A:

1. list is null
   - b_1 = True
   - b_2 = False
2. list is empty
   - b_1 = True
   - b_2 = False

Input Domain Model

- Approaches:
  2. ***functionality-based* input domain modelling:**
     - Develops **characteristics** from a functional or behavioral view of the program under test.

Input Domain Model

Functionality-based input domain modelling

- *Example*:

Q: Define two functionality-based characteristics for list, including blocks and values in the function below:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
// else returns true if element is in the list, false otherwise
```

A:

1. Number of occurrences of element in list
   - $b_1 = 0$
   - $b_2 = 1$
   - $b_3 = >1$
2. Element occurs first in list
   - b_1 = True
   - b_2 = False

Partition Testing

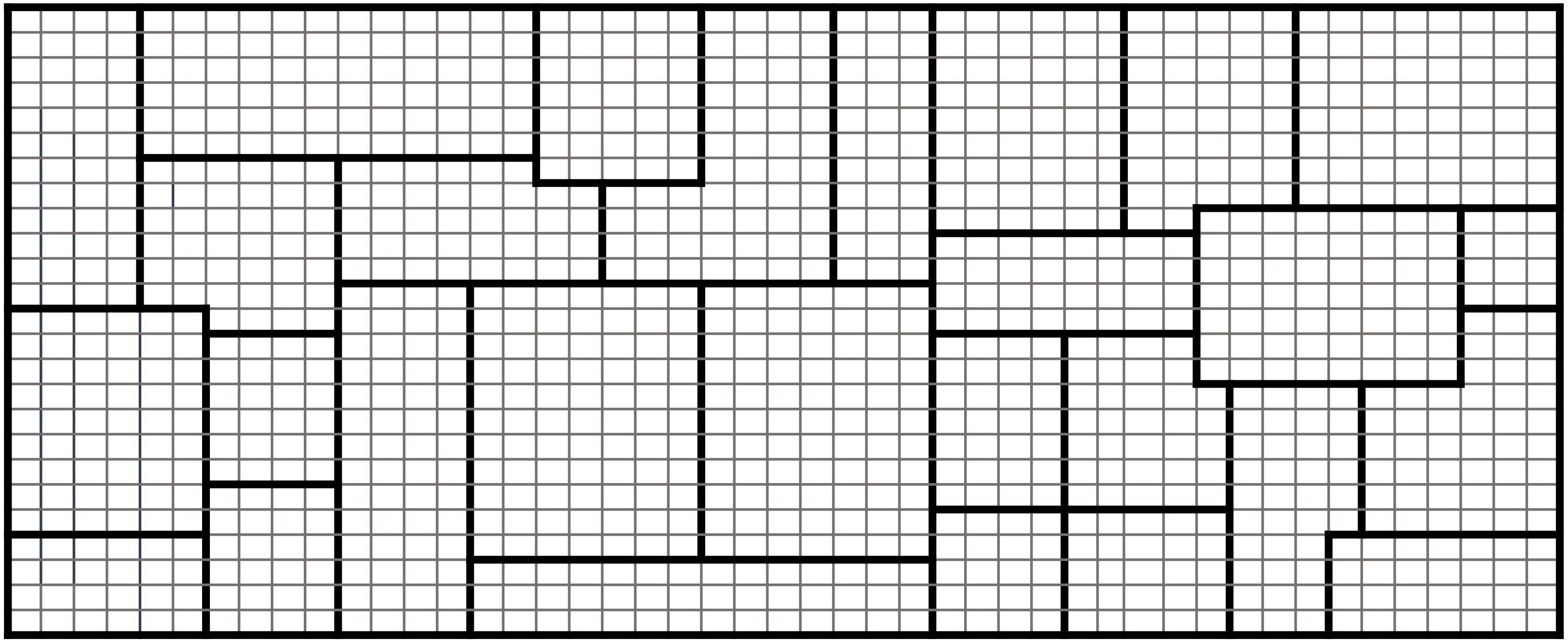
- Choosing blocks and values:
  - More blocks will result in more tests, requiring more resources but possibly finding more faults.
  - Fewer blocks will result in fewer tests, saving resources but possibly reducing test effectiveness.

Partition Testing

- Strategies for choosing blocks and values:
  - **Valid values**: Include at least one group of valid values.
  - **Invalid values**: Include at least one group of invalid values.
  - **Boundaries**: Values at or close to boundaries often cause problems.
  - **Missing partitions:** Check that the union of all blocks of a characteristic completely covers the input space of that characteristic.
  - **Overlapping partitions:** Check that no value belongs to more than one block.

Boundary Values

- Errors tend to occur at the boundary of a partition



  - The objective is to select input at this boundaries
  - Boundary testing (i.e. selection of boundary inputs) is complementary to partition testing

Boundary Value

- Example:
  - Define the partitions, blocks, values and possible boundary inputs for the split function

String[] split(String str, int size)

1. *Partition*: size value
   - b_1 = <0
   - b_2 = 0
   - b_3= >0

   Possible inputs:
   - b_1 : size = -1
   - b_2 : size = 0
   - b_3 : size = 1
   - b_3 : size = MAXINT

2. *Partition*: str with length
   - b_1 = < size
   - b_2 = in[size,size*2]
   - b_3 = > size*2

   Possible inputs:
   - b_1 : str with length = size−1
   - b_2 : str with length = size
   - b_3 : str with length = size*2+1

   etc…

Summary

- Partition Testing is a non-naïve random testing approach
- It leverages the observation that software defects are sparse and clustered in small regions
- Split the input domain into disjoint and non-overlapping regions based on appropriately picked characteristics
- Boundary values are complementary to partition testing

A Systematic Functional Testing Approach Part A

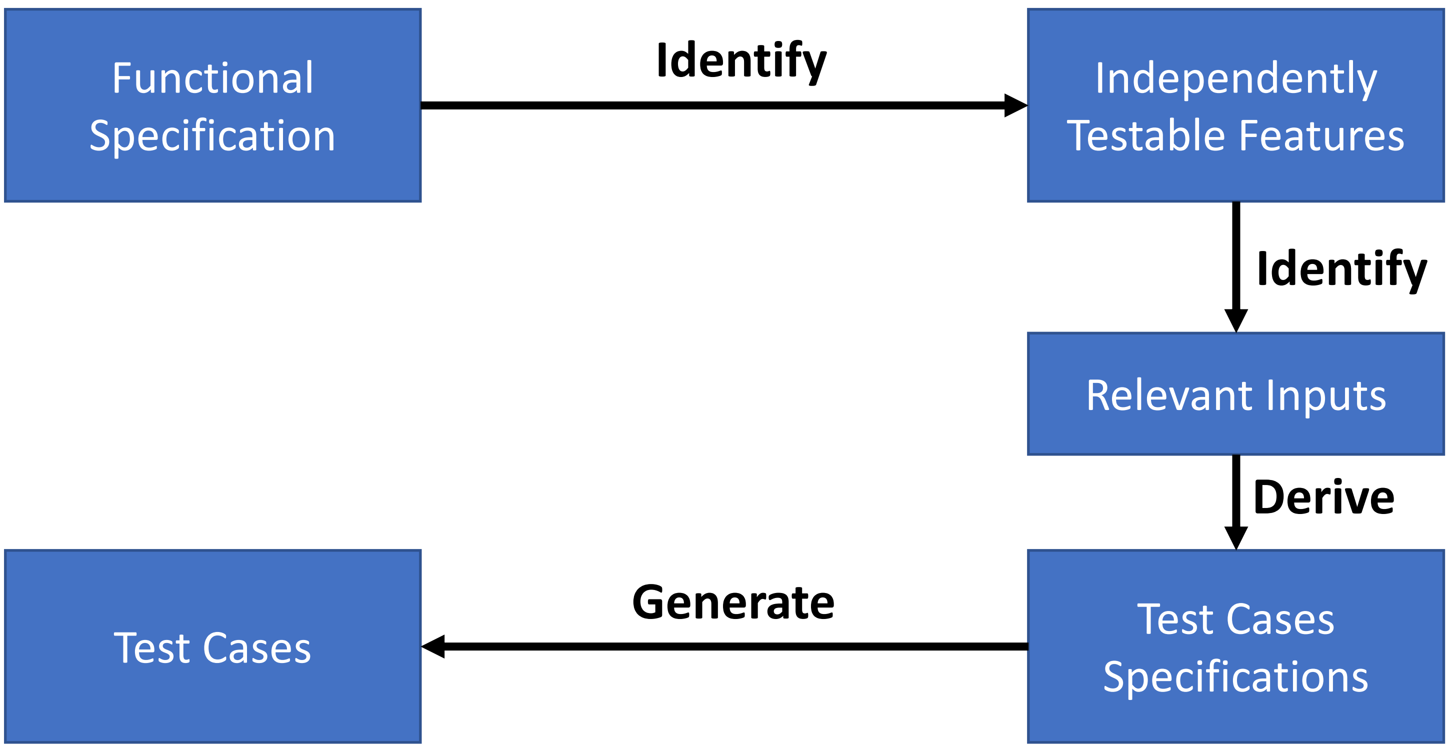
Dr Fani Deligianni,

Fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Systematic Approach

- From Functional Specification to Test cases



Systematic Approach

- Identifying Independently testable features

Functional Specification $\xrightarrow{\textbf{Identify}}$ Independently Testable Features

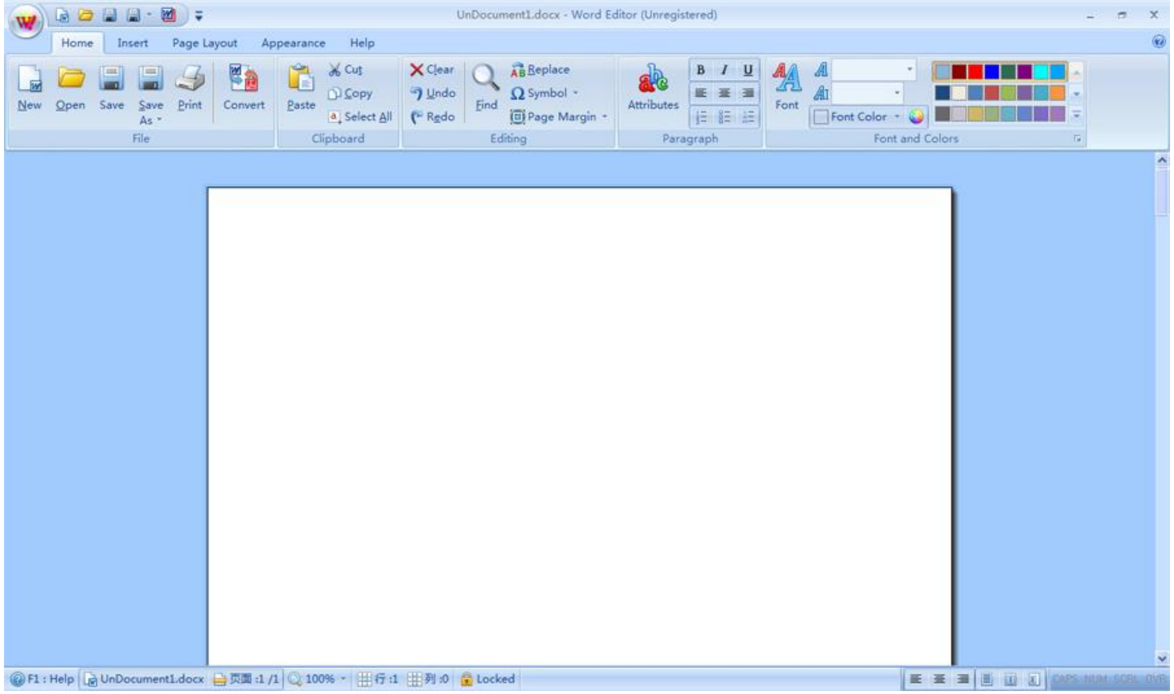
```
getSum(int a int b)
```

Q: How many independently testable features do we have in getSum?

A: [ ] 1
[ ] 2
[ ] 3
[ ] >3

Systematic Approach

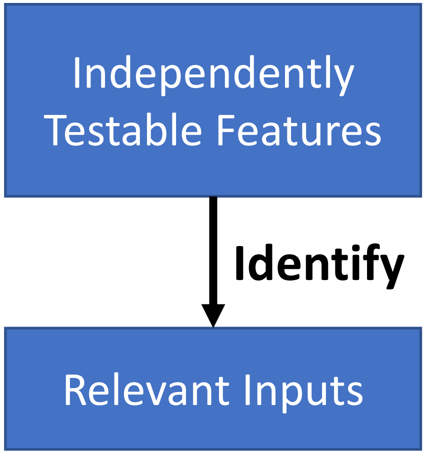
- Identifying Independently testable features



Q: List three independently testable feature of a word processor?
A: (1) ________ (2)___________ (3) _____________

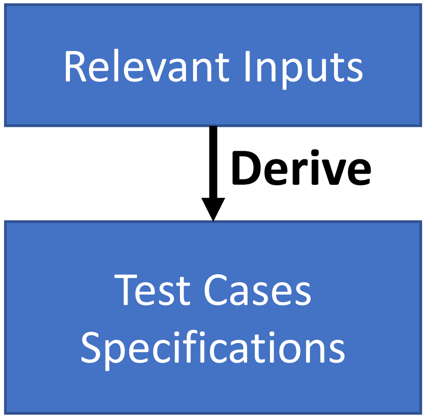
Systematic Approach

- Identifying relevant inputs for each features
  - Using Partition and Boundary Value Testing



Systematic Approach

- Deriving test case specifications:
  A *test case specification* defines how value of relevant inputs should be put together when testing the system.



Test Case Specification

- Example:
  - Write the test case specification for the split function

String[] split(String str, int size)

Possible inputs:	Possible inputs:
– b_1 : size = -1	– b_1 : str with length = size−1
– b_2 : size = 0	– b_2 : str with length = size
– b_3 : size = 1	– b_3 : str with length = size*2+1
– b_3 : size = MAXINT	etc…

Test Case Specifications

Test Case Specification

- Example:
  - Write the test case specification for the split function

String[] split(String str, int size)

Possible inputs:		Possible inputs:
$-b_1$: size = -1		$-b_1$: str with length = size−1
$-b_2$: size = 0	×	$-b_2$: str with length = size
$-b_3$: size = 1		$-b_3$: str with length = size*2+1
$-b_3$: size = MAXINT		etc…

- Test Case Specifications – **Combinatorial Approach**

size = -1 , str with length = -2

size = -1 , str with length = -1

size = -1 , str with length = 0

Test Case Specification

- Example:
  - Write the test case specification for the split function

String[] split(String str, int size)

- Test Case Specifications – Combinatorial Approach

size = -1 , str with length = -2

size = -1 , str with length = -1

size = -1 , str with length = 0

size = 0 , str with length = -1

size = 0 , str with length = 0

size = 0 , str with length = 1

etc…

Combinatorial Explosion

Test Case Specification

- Example:
  - Write the test case specification for the split function

String[] split(String str, int size)

- Test Case Specifications – Combinatorial Approach

size = -1 , str with length = -2

size = -1 , str with length = -1

size = -1 , str with length = 0

size = 0 , str with length = -1

size = 0 , str with length = 0

size = 0 , str with length = 1

etc…

Combinations that don’t make much sense

A Systematic Functional Testing Approach Part B

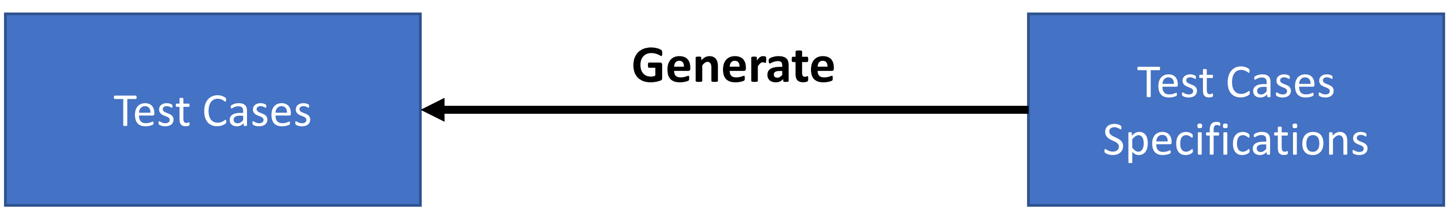
Dr Fani Deligianni,

Fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Systematic Approach

- The last step of generating test cases from test case specification is a fairly mechanical process



The Category-Partition Method

- Ostrand & Balcer , CACM, June 1988

T. J. Ostrand and M. J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* 31, 6 (June 1988), 676-686.

The Category-Partition Method

Specification

1. Identify independently testable features
2. Identify **categories**
3. **Partition** categories into choices
4. Identify constraints among choices
5. Produce/evaluate test case specification
6. Generate test case from test case specifications

Test Cases

The Category-Partition Method

1. Identify independently testable features
   - We are already familiar with this step in our systematic approach

The Category-Partition Method

2. Identify Categories
   - Where each category is the ***characteristics*** of each input element

String[] split(String str, int size)

Input str	Input size
–length	–value
–content	

The Category-Partition Method

3. Partition categories into choices

String[] split(String str, int size)

Input str

–length

- 0
- size -1
- …

–content

- spaces
- special characters
- …

Input size

–value

- 0
- >0
- <0
- MAXINT
- …

The Category-Partition Method

4. Identify constraints among choices
   - To eliminate meaningless combinations
   - To reduce the number of test cases
   - Three types of constraint properties:
     - PROPERTY… IF,
     - ERROR,
     - SINGLE

The Category-Partition Method

4. Identify constraints among choices

PROPERTY… IF

String[] split(String str, int size)

Input str

–length

- 0 PROPERTY zerovalue

PROPERTY… IF

- We can define zero value as a special property to highlight when length is 0
- We can then use this property to exclude some meaningless combinations (for instance a string of length =0 cannot be a special character).

The Category-Partition Method

4. Identify constraints among choices

ERROR

String[] split(String str, int size)

Input size

–value

- <0 ERROR
- MAXINT

- The choice <0 for input size is marked as an error property.
- This means that when generating possible combination of test case specifications, the choice <0 will be considered only once.

The Category-Partition Method

4. Identify constraints among choices

SINGLE

String[] split(String str, int size)

Input size
–value
- <0
- MAXINT SINGLE

- Used when the objective is to limit the number of test cases.
- The choice is used only once, and not combined multiple times.
- A SINGLE property for MAXINT means that we will only have one test case for which the size equals MAXINT

The Category-Partition Method

5. Produce/evaluate test case specification
   - Can be completely automated
   - The result is a set of **test frames**

Specification of a test

Example:

Test frame #34
    Input str
        Length: size -1
        content: special characters
    Input size
        value: >1

The Category-Partition Method

6. Generate test case from test case specifications
   - Involves simple instantiation of test frames.
   - **Final Result:** Set of concrete test cases

Example:

Test frame #34
- Input str
  - Length: size -1
  - content: special characters
- Input size
  - value: >1

Test case #34
- Str =“ABCC!\n\t**β**”
- Size=11