CMPT 473
Software Quality Assurance

Input Space Partitioning

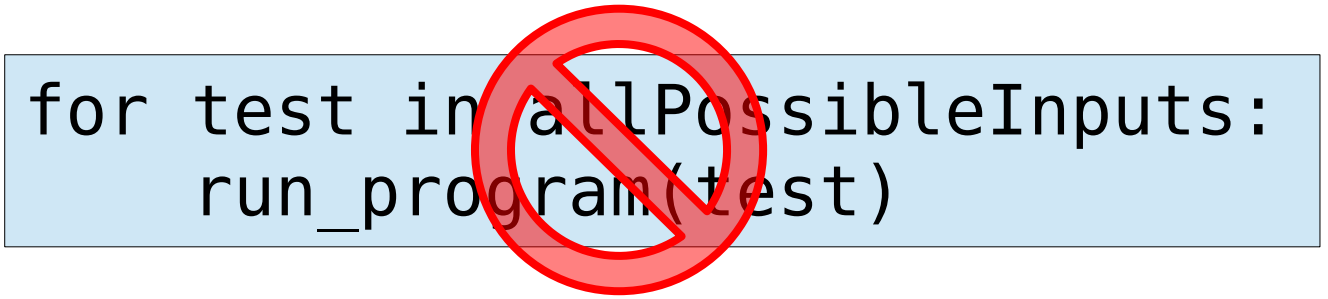
Nick Sumner

Recall

- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output

Recall

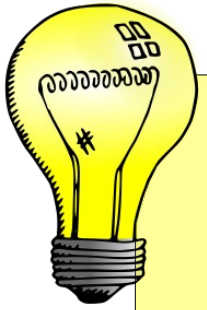
- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output
- Problem: The *input domain* is infinite or pragmatically infinite.



Recall

- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output
- Problem: The *input domain* is infinite or pragmatically infinite.
- Testing is about selecting a finite subset of inputs that can help measure quality

Input Space Partitioning



Take the direct approach:
Focus on the input!

Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes

Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

e.g. `abs(x)`

Input Domain: ..., -3, -2, -1, 0, 1, 2, 3, ...

How many tests if done exhaustively?

Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

e.g. `abs(x)`

Input Domain: ..., -3, -2, -1, 0, 1, 2, 3, ...

Partitions: ..., -3, -2, -1, 0, 1, 2, 3, ...

What might reasonable partitions be?

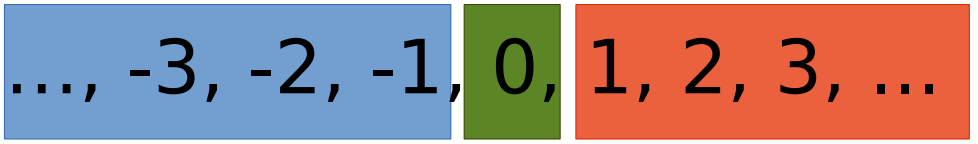
Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

e.g. `abs(x)`

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

Partitions:



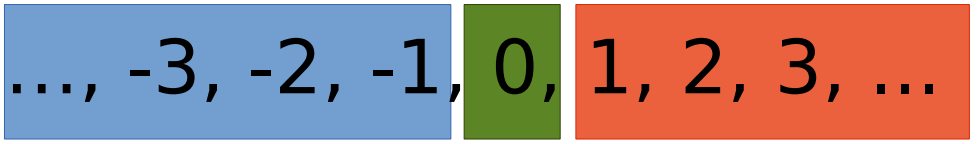
Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

e.g. `abs(x)`

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

Partitions:



How many tests for the partitions?

Input Space Partitioning

- *Input Space Partitioning*
  - Divide (*partition*) the set of possible inputs into equivalence classes
  - Test one input from each class

e.g. `abs(x)`

Input Domain: ..., -3, -2, -1, 0, 1, 2, 3, ...

Partitions:

Impressive! How do we do it?

Input Space Partitioning

1) Identify the component

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs

What might the inputs be?

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters
   - Object state

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters
   - Object state
   - Global variables

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters
   - Object state
   - Global variables
   - File contents

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters
   - Object state
   - Global variables
   - File contents
   - User provided inputs

Input Space Partitioning

1) Identify the component
   - Whole program
   - Module
   - Class
   - Function

2) Identify the inputs
   - Function/method parameters
   - Object state
   - Global variables
   - File contents
   - User provided inputs
   - …

Input Space Partitioning

3) Develop an *Input Domain Model*

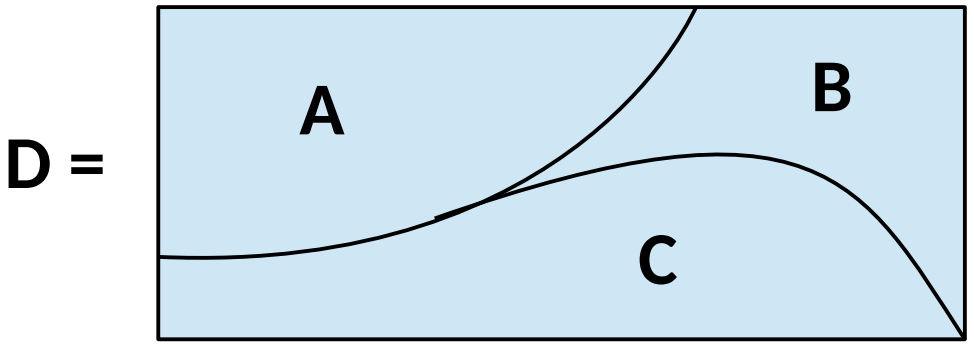
Input Space Partitioning

3) Develop an *Input Domain Model*

- A way of *describing* the possible inputs
- Partitioned by characteristics

Partitioned Input Domain

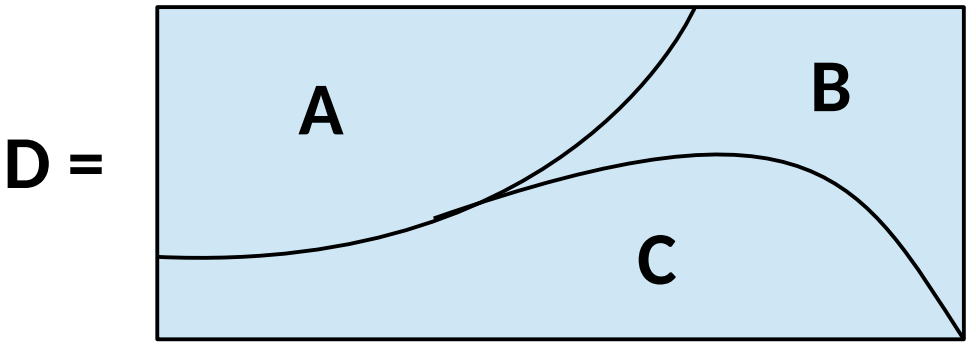
- ***Partition*** the domain D *on characteristics*



Partitioned Input Domain

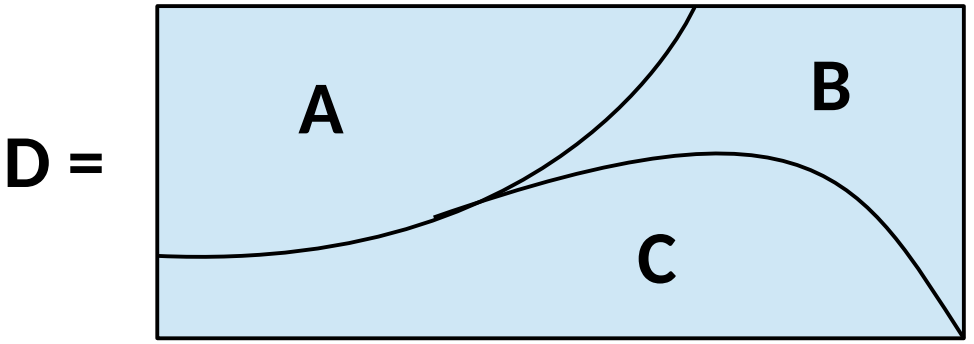
- ***Partition*** the domain D *on characteristics*

What are ***characteristics***?



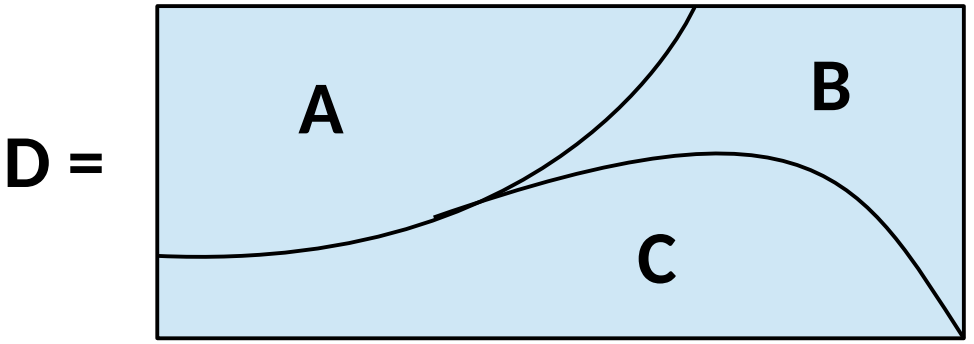
Partitioned Input Domain

- ***Partition*** the domain D on characteristics
- Must satisfy 2 criteria:
  - Disjoint: $A \cap B \cap C = \varnothing$
  - Cover: $A \cup B \cup C = D$



Partitioned Input Domain

- ***Partition*** the domain D on characteristics
- Must satisfy 2 criteria:
  - Disjoint: $A \cap B \cap C = \varnothing$
  - Cover: $A \cup B \cup C = D$



What do these criteria intuitively provide?

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs
    (from requirements, types, etc.)

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs (from requirements, types, etc.)
  - Partition into ***blocks*** based on each characteristic

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs (from requirements, types, etc.)
  - Partition into ***blocks*** based on each characteristic
  - Create tests by ***selecting values*** for each block

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs (from requirements, types, etc.)
  - Partition into ***blocks*** based on each characteristic
  - Create tests by ***selecting values*** for each block

How many tests might this imply?
Might there be more? Fewer?

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs (from requirements, types, etc.)
  - Partition into ***blocks*** based on each characteristic
  - Create tests by ***selecting values*** for each block

How many tests might this imply?
Might there be more? Fewer?

We're hiding some details in this last step.
It's not quite right yet.

Using Partitions

- Select one input from each block
- Each input in a block is assumed equally useful
- How?
  - Identify ***characteristics*** of the possible inputs (from requirements, types, etc.)
  - Partition into ***blocks*** based on each characteristic
  - Create tests by ***selecting values*** for each block
- Characteristics:
  - List s is sorted ascending
  - X is null
  - String length
  - …

Partitioning is Subtle

- Suppose we have:

```
classifyParallelogram(p1)
```

(Informal) Characteristic: “The subtype of parallelogram”

Partitioning is Subtle

- Suppose we have:

```
classifyParallelogram(p1)
```

Characteristic: “The subtype of parallelogram”

  - How can we partition based on this characteristic?
  - What problems might arise?

Partitioning is Subtle

- Suppose we have:

```
classifyParallelogram(p1)
```

Characteristic: “The subtype of parallelogram”

  - How can we partition based on this characteristic?
  - What problems might arise?

- In class exercise:
  Partitioning a triangle classifying program

```
triType(int s1, int s2, int s3)
```

Partitioning is Subtle

- Suppose we have:

```
classifyParallelogram(p1)
```

Characteristic: “The subtype of parallelogram”

  - How can we partition based on this characteristic?
  - What problems might arise?

- In class exercise:
  Partitioning a triangle classifying program

- It is easy to create overlapping partitions.
  - Care and design required to avoid it.

Partitioning is Subtle

- Suppose we have:

```
classifyParallelogram(p1)
```

Characteristic: "The subtype of parallelogram"

  - How can we partition based on this characteristic?
  - What problems might arise?

- In class exercise:
  Partitioning a triangle classifying program

- It is easy to create overlapping partitions.
  - Care and design required to avoid it.

Why do disjoint partitions matter?

Process (Reiterated)

3 step process (for now):

1) Find the component / function to test

   methods, classes, programs, functions

Process (Reiterated)

3 step process (for now):

1) Find the component / function to test

   methods, classes, programs, functions

2) Find all test parameters

   - Must identify ***everything***

     locals, globals, files, databases, schedules, servers, …

Process (Reiterated)

3 step process (for now):

1) Find the component / function to test

   methods, classes, programs, functions

2) Find all test parameters

   - Must identify ***everything***

     locals, globals, files, databases, schedules, servers, …

3) Model the input domain

   - Identify characteristics
   - Partition the input domain
   - Select values for each region

Process (Reiterated)

3 step process (for now):

1) Find the component / function to test

   methods, classes, programs, functions

2) Find all test parameters

Domain knowledge, tactics, and creativity apply here.

3) Model the input domain
   - Identify characteristics
   - Partition the input domain
   - Select values for each region

Approaches to Input Modeling

We still haven't talked about ***how*** to model input!

Approaches to Input Modeling

2 Main approaches:

Approaches to Input Modeling

2 Main approaches:

1) Interface based
   - Guided directly by identified parameters & domains

Approaches to Input Modeling

2 Main approaches:

1) Interface based
   - Guided directly by identified parameters & domains
   - Simple
   - Automatable

Approaches to Input Modeling

2 Main approaches:

1) Interface based
   - Guided directly by identified parameters & domains
   - Simple
   - Automatable
2) Functionality/Requirements based
   - Derived from expected input/output relationship by spec.

Approaches to Input Modeling

2 Main approaches:

1) Interface based
   - Guided directly by identified parameters & domains
   - Simple
   - Automatable

2) Functionality/Requirements based
   - Derived from expected input/output relationship by spec.
   - Requires more design & more thought
   - May be better (smaller, goal oriented, …)

Interface Based Modeling

- Consider parameters individually

Interface Based Modeling

- Consider parameters individually
  - Examine their types/domains
  - Ignore relationships & dependences

Interface Based Modeling

- Consider parameters individually
  - Examine their types/domains
  - Ignore relationships & dependences

How does this apply to our triangle classifier?

Functionality Based Modeling

- Identify characteristics corresponding to behaviors/functionality in the requirements

Functionality Based Modeling

- Identify characteristics corresponding to behaviors/functionality in the requirements
  - Includes knowledge from the problem domain

Functionality Based Modeling

- Identify characteristics corresponding to behaviors/functionality in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relationships between parameters

Functionality Based Modeling

- Identify characteristics corresponding to behaviors/functionality in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relationships between parameters
  - Same parameter may appear in multiple characteristics
    - Need to reason about constraints & conflicts!

Functionality Based Modeling

- Identify characteristics corresponding to behaviors/functionality in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relationships between parameters
  - Same parameter may appear in multiple characteristics
    - Need to reason about constraints & conflicts!

How might this apply to our triangle classifier?

Finding Typical Characteristics

What might typical characteristics be?

Finding Typical Characteristics

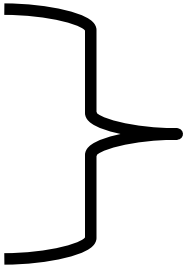
What might typical characteristics be?

- Preconditions
- Postconditions

Finding Typical Characteristics

What might typical characteristics be?

- Preconditions
- Postconditions



Invariants

Finding Typical Characteristics

What might typical characteristics be?

- Preconditions
- Postconditions
- Relationships to special values
- Relationships between variables

Finding Typical Values

How might you select values for a block?

Finding Typical Values

How might you select values for a block?

- Expected values (e.g. exampled from spec)
- Invalid, valid, & special values
- Boundary values

Finding Typical Values

How might you select values for a block?

- Expected values (e.g. exampled from spec)
- Invalid, valid, & special values
- Boundary values

Thought experiment:
What do boundary values as a selection approach indicate?

An Interface Based Example

- Consider our triangle classifier
  - Takes 3 integers for sides 1, 2, & 3

An Interface Based Example

- Consider our triangle classifier
  - Takes 3 integers for sides 1, 2, & 3

Characteristic	b1	b2	b3
Side 1 <?> 0	Side 1 > 0	Side 1 = 0	Side 1 < 0
Side 2 <?> 0	Side 2 > 0	Side 2 = 0	Side 2 < 0
Side 3 <?>0	Side 3 > 0	Side 3 = 0	Side 3 < 0

An Interface Based Example

- Consider our triangle classifier
  - Takes 3 integers for sides 1, 2, & 3

Characteristic	b1	b2	b3
Side 1 <?> 0	Side 1 > 0	Side 1 = 0	Side 1 < 0
Side 2 <?> 0	Side 2 > 0	Side 2 = 0	Side 2 < 0
Side 3 <?>0	Side 3 > 0	Side 3 = 0	Side 3 < 0

How many tests does this create?

An Interface Based Example

- Consider our triangle classifier
  - Takes 3 integers for sides 1, 2, & 3

Characteristic	b1	b2	b3
Side 1 <?> 0	Side 1 > 0	Side 1 = 0	Side 1 < 0
Side 2 <?> 0	Side 2 > 0	Side 2 = 0	Side 2 < 0
Side 3 <?>0	Side 3 > 0	Side 3 = 0	Side 3 < 0

How many tests does this create?

What will this test well?
What won't this test well?

Refining the Example

- We can subdivide partitions to cover more behavior

Characteristic	b1	b2	b3	b4
Value of side 1	Side 1 > 1	Side 1 = 1	Side 1 = 0	Side 1 < 0
Value of side 2	Side 2 > 1	Side 2 = 1	Side 2 = 0	Side 2 < 0
Value of side 3	Side 3 > 1	Side 3 = 1	Side 3 = 0	Side 3 < 0

{Side n > 0} → {Side n = 1},{ Side n > 1}

Refining the Example

- We can subdivide partitions to cover more behavior

Characteristic	b1	b2	b3	b4
Value of side 1	Side 1 > 1	Side 1 = 1	Side 1 = 0	Side 1 < 0
Value of side 2	Side 2 > 1	Side 2 = 1	Side 2 = 0	Side 2 < 0
Value of side 3	Side 3 > 1	Side 3 = 1	Side 3 = 0	Side 3 < 0

How many tests now?

Refining the Example

- We can subdivide partitions to cover more behavior

Characteristic	b1	b2	b3	b4
Value of side 1	Side 1 > 1	Side 1 = 1	Side 1 = 0	Side 1 < 0
Value of side 2	Side 2 > 1	Side 2 = 1	Side 2 = 0	Side 2 < 0
Value of side 3	Side 3 > 1	Side 3 = 1	Side 3 = 0	Side 3 < 0

How many tests now?

Is it still disjoint? Complete?

Refining the Example

- We can subdivide partitions to cover more behavior

Characteristic	b1	b2	b3	b4
Value of side 1	Side 1 > 1	Side 1 = 1	Side 1 = 0	Side 1 < 0
Value of side 2	Side 2 > 1	Side 2 = 1	Side 2 = 0	Side 2 < 0
Value of side 3	Side 3 > 1	Side 3 = 1	Side 3 = 0	Side 3 < 0

How many tests now?

Is it still disjoint? Complete?

What does it test well? Not well?

A Functionality Based Example

- Consider our triangle classifier again
  - What might our characteristics & partitions be?

A Functionality Based Example

- Consider our triangle classifier again
  - What might our characteristics & partitions be?
  - Are there alternatives?

A Functionality Based Example

- Consider our triangle classifier again
  - What might our characteristics & partitions be?
  - Are there alternatives?
  - Why might you use them?

A Richer Functionality Based Example

- Suppose we have a simple function:

```
symmetricDifference(s1, s2)
```

that returns all elements unique to either s1 or s2.

A Richer Functionality Based Example

- Suppose we have a simple function:

```
symmetricDifference(s1, s2)
```

  that returns all elements unique to either s1 or s2.

- Try to construct a functionality based input domain model.
  - Keep disjointness and completeness in mind.

Try it out, and we'll discuss

A Classic Example

- Start with a component / specification:

Command	FIND
Syntax	FIND <pattern> <file>
Function	The FIND command is used to **locate one or more instances** of a given **pattern in a text file**. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is **written only once**, regardless of the number of times the pattern occurs on it. The pattern is any sequence of characters whose **length does not exceed** the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be **enclosed in quotes (")**. To include a quotation mark in the pattern, **two quotes in a row ("")** must be used.

A Classic Example

- Step 1: Analyze the specification
  - What is the component?
  - What are the parameters?
  - What are the characteristics?

A Classic Example

- Step 1: Analyze the specification
  - Wha
  - Wha
  - What are the characteristics?

```
Parameters:
  Pattern
  Input file (& its contents!)
```

A Classic Example

- Step 1: Analyze the specification
  - Wha
  - Wha
  - What are the characteristics?

```
Parameters:
  Pattern
  Input file (& its contents!)
```

```
Characteristics:
  Pattern
  Input file
  Pattern Size
  Quoting
  Embedded Quotes
  File Name
Environment / System Characteristics:
  # of pattern occurrences in file
  # of occurrences on a particular line:
```

A Classic Example

- Step 2: Partition the Input Space
  - Guided by intelligence and intuition
  - *Combine* interface and functionality based approaches as necessary

A Classic Example

- Step 2: Partition the Input Space
  - Guided by intelligence and intuition
  - *Combine* interface and functionality based approaches as necessary

```
Parameters:
  Pattern Size:
    Empty
    Single character
    Many characters
    Longer than any line in the file
  Quoting:
...
```

A Classic Example

- Familiar Idea:
  - Select one block per characteristic at a time
  - Combine into test *frames* (test case plans)
  - e.g. …

A Classic Example

- Familiar Idea:
  - Select one block per characteristic at a time
  - Combine into test *frames* (test case plans)
  - e.g.

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

A Classic Example

- Familiar Idea:
  - Select one block per characteristic at a time
  - Combine into test *frames* (test case plans)
  - e.g.

Problem?

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

A Classic Example

- Familiar Idea:
  - Select one block per characteristic at a time
  - Combine into test *frames* (test case plans)
  - e.g.

Problem?

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

A Classic Example

- Step 3: Identify ***constraints*** among the characteristics & blocks

```
Pattern Size:
  Empty                                  [Property Empty]
  Single character                       [Property NonEmpty]
  Many characters                        [Property NonEmpty]
  Longer than any line in the file       [Property NonEmpty]
```

A Classic Example

- Step 3: Identify ***constraints*** among the categories

```
Pattern Size:
  Empty                                    [Property Empty]
  Single character                         [Property NonEmpty]
  Many characters                          [Property NonEmpty]
  Longer than any line in the file         [Property NonEmpty]
```

```
Quoting:
  Pattern is quoted                        [Property Quoted]
  Pattern is not quoted                    [If NonEmpty]
  Pattern is improperly quoted             [If NonEmpty]
```

A Classic Example

- Step 3: Identify ***constraints*** among the categories

```
Pattern Size:
  Empty                                    [Property Empty]
  Single character                         [Property NonEmpty]
  Many characters                          [Property NonEmpty]
  Longer than any line in the file         [Property NonEmpty]
```

```
Quoting:
  Pattern is quoted                        [Property Quoted]
  Pattern is not quoted                    [If NonEmpty]
  Pattern is improperly quoted             [If NonEmpty]
```

What should this do to the number of tests?
To the quality of tests?

A Classic Example

- Step 4
  - Create tests by selecting values that satisfy the selected blocks for each frame
  - Eliminate tests that cover redundant scenarios

A Classic Example

- Step 4
  - Create tests by selecting values that satisfy the selected blocks for each frame
  - Eliminate tests that cover redundant scenarios

Why might scenarios be redundant?

A Classic Example

- Step 5:
  - Take your generated test cases and automate them

The next steps…

- We have talked so far as if we have a single input and a single model, but real world programs have many!