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.

```python
for test in allPossibleInputs:
    run_program(test)
```
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. \( \text{abs}(x) \)

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. \( \text{abs}(x) \)

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

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

\text{e.g. abs(}x)\text{)}

Input Domain: \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots

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

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. \( \text{abs}(x) \)

Input Domain: \( \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \)

Partitions: \( \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \)

Impressive! How do we do it?
Input Space Partitioning

1) Identify the component
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

$$D = \begin{align*}
A & \quad B \\
C &
\end{align*}$$
Partitioned Input Domain

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

What are *characteristics*?

$$D = \text{A \quad B \quad C}$$
Partitioned Input Domain

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

\[ D = \]

\[ \begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\end{array} \]
Partitioned Input Domain

• **Partition** the domain D on characteristics
• Must satisfy 2 criteria:
  – Disjoint: $A \cap B \cap C = \emptyset$
  – 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:

\[
\text{classifyParallelogram}(p1)
\]

(Informal)Characteristic: “The subtype of parallelogram”
Partitioning is Subtle

- Suppose we have:

  ```java
  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
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?
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

\[ \text{Invariants} \]
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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side 1 &lt;&gt; 0</td>
<td>Side 1 &gt; 0</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Side 2 &lt;&gt; 0</td>
<td>Side 2 &gt; 0</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Side 3 &lt;&gt;0</td>
<td>Side 3 &gt; 0</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>
An Interface Based Example

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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side 1 &lt;?&gt; 0</td>
<td>Side 1 &gt; 0</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Side 2 &lt;?&gt; 0</td>
<td>Side 2 &gt; 0</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Side 3 &lt;?&gt; 0</td>
<td>Side 3 &gt; 0</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

How many tests does this create?
An Interface Based Example

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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side 1 &lt;?&gt; 0</td>
<td>Side 1 &gt; 0</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Side 2 &lt;?&gt; 0</td>
<td>Side 2 &gt; 0</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Side 3 &lt;?&gt;0</td>
<td>Side 3 &gt; 0</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of side 1</td>
<td>Side 1 &gt; 1</td>
<td>Side 1 = 1</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Value of side 2</td>
<td>Side 2 &gt; 1</td>
<td>Side 2 = 1</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Value of side 3</td>
<td>Side 3 &gt; 1</td>
<td>Side 3 = 1</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

\{\text{Side } n > 0\} \rightarrow \{\text{Side } n = 1\}, \{\text{Side } n > 1\}
Refining the Example

- We can subdivide partitions to cover more behavior

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of side 1</td>
<td>Side 1 &gt; 1</td>
<td>Side 1 = 1</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Value of side 2</td>
<td>Side 2 &gt; 1</td>
<td>Side 2 = 1</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Value of side 3</td>
<td>Side 3 &gt; 1</td>
<td>Side 3 = 1</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

How many tests now?
Refining the Example

- We can subdivide partitions to cover more behavior

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of side 1</td>
<td>Side 1 &gt; 1</td>
<td>Side 1 = 1</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Value of side 2</td>
<td>Side 2 &gt; 1</td>
<td>Side 2 = 1</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Value of side 3</td>
<td>Side 3 &gt; 1</td>
<td>Side 3 = 1</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

How many tests now?

Is it still disjoint? Complete?
## Refining the Example

- We can subdivide partitions to cover more behavior

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of side 1</td>
<td>Side 1 &gt; 1</td>
<td>Side 1 = 1</td>
<td>Side 1 = 0</td>
<td>Side 1 &lt; 0</td>
</tr>
<tr>
<td>Value of side 2</td>
<td>Side 2 &gt; 1</td>
<td>Side 2 = 1</td>
<td>Side 2 = 0</td>
<td>Side 2 &lt; 0</td>
</tr>
<tr>
<td>Value of side 3</td>
<td>Side 3 &gt; 1</td>
<td>Side 3 = 1</td>
<td>Side 3 = 0</td>
<td>Side 3 &lt; 0</td>
</tr>
</tbody>
</table>

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:
  
  \[ \text{symmetricDifference}(s1, s2) \]

  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:

<table>
<thead>
<tr>
<th>Command</th>
<th>FIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>FIND &lt;pattern&gt; &lt;file&gt;</td>
</tr>
<tr>
<td>Function</td>
<td>The FIND command is used to <strong>locate one or more instances</strong> of a given <strong>pattern in a text file</strong>. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is <strong>written only once</strong>, regardless of the number of times the pattern occurs on it. The pattern is any sequence of characters whose <strong>length does not exceed</strong> the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be <strong>enclosed in quotes (&quot;&quot;)</strong>. To include a quotation mark in the pattern, <strong>two quotes in a row (&quot;&quot;&quot;)</strong> must be used.</td>
</tr>
</tbody>
</table>
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
  - What is the component?
  - What are the parameters?
  - What are the characteristics?

Parameters:
- Pattern
- Input file (& its contents!)
A Classic Example

- Step 1: Analyze the specification
  - What is the component?
  - What are the parameters?
  - 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 \textit{frames} (test case plans)
  – e.g.

\begin{verbatim}
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
\end{verbatim}
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

<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>[Property Empty]</th>
<th>[Property NonEmpty]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single character</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
<tr>
<td>Many characters</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
<tr>
<td>Longer than any line in the file</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
</tbody>
</table>
A Classic Example

• Step 3: Identify *constraints* among the categories

<table>
<thead>
<tr>
<th>Pattern Size:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>Single character</td>
</tr>
<tr>
<td>Many characters</td>
</tr>
<tr>
<td>Longer than any line in the file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quoting:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern is quoted</td>
</tr>
<tr>
<td>Pattern is not quoted</td>
</tr>
<tr>
<td>Pattern is improperly quoted</td>
</tr>
</tbody>
</table>
A Classic Example

• Step 3: Identify *constraints* among the categories

<table>
<thead>
<tr>
<th>Pattern Size:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>[Property Empty]</td>
<td></td>
</tr>
<tr>
<td>Single character</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
<tr>
<td>Many characters</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
<tr>
<td>Longer than any line in the file</td>
<td>[Property NonEmpty]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quoting:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern is quoted</td>
<td>[Property Quoted]</td>
<td></td>
</tr>
<tr>
<td>Pattern is not quoted</td>
<td>[If NonEmpty]</td>
<td></td>
</tr>
<tr>
<td>Pattern is improperly quoted</td>
<td>[If NonEmpty]</td>
<td></td>
</tr>
</tbody>
</table>

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!