CITS5501 Software Testing and Quality Assurance
Input Space Partition Testing, continued

Unit coordinator: Arran Stewart

Input Space Partitioning

ISP technique

Let's use the `findElement` method we saw at the start of the lecture as an example.

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

ISP is about considering the domain for the function – all its possible inputs – and choosing finite sets of values from the input domain to use as **test values**.

Input parameters define the scope of the input domain.

- Parameters to a method
- Data read from a file
- Global variables
- User level inputs

ISP technique

- The domain for *each* input parameter is partitioned into regions
- At least one value is chosen from each region

Not just methods

We ccan apply the ISP technique not just to Java methods, but *anything* we're able to model as a function.

- Systems - e.g. a database system. We could consider it as taking *in* use requests (e.g. a manager requests a report on quarterly revenues) and spitting *out* reports.
  - (Bearing in mind that we have to ensure we account for *all* the parameters, not just the obvious ones.)
- Hardware - e.g. an Internet-controllable toaster. We can consider as taking *in* toasters settings and untoasted bread, and spitting *out* toast.

Benefits of ISP

The ISP technique has some useful advantages:

- Can be equally applied at several levels of testing
    - Unit
    - Integration
    - System
- Easy to adjust the procedure to get more or fewer tests

Relationship to other techniques

ISP subsumes several other techniques you might see mentioned in textbooks or online:

- equivalence partitioning
- boundary value analysis
- domain testing

These techniques are collectively referred to as “partition testing”.

Relationship to other techniques

ISP ignores a distinction you might see made between what is called “white box testing” and “black box testing” – more on this later.

Steps in ISP

- Identify testable functions
- Identify **all** *parameters* to the functions
- Model the input domain in terms of *characteristics*, each of which gives rise to a set of partitions.
    - example: “sign of a number”, used for abs
- Choose particular partitions, and values from within those partitions
- Refine into test values
- Review!

Steps in ISP

Some questions:

- what is a partition?
- what is a characteristic?
- how do we come up with them?

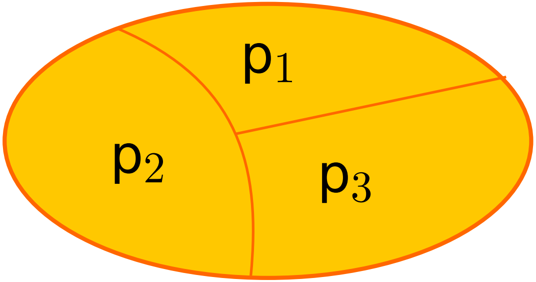
Partitioning

Partitions

- Informally:

  partitions are a collection of disjoint sets of some domain *D* which *cover* the domain.

- They are pairwise disjoint (i.e. none overlap each other)



Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{ \text{ numbers } < 0\}$
- $p_2 = \{ \text{ numbers } > 0\}$

Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{ \text{ numbers } < 0\}$
- $p_2 = \{ \text{ numbers } > 0\}$

It is not – it leaves out 0, so the proposed partitioning doesn't *cover* the domain.

Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{ \text{ numbers } \leq 0\}$
- $p_2 = \{ \text{ numbers } \geq 0\}$

Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{$ numbers $\leq 0\}$
- $p_2 = \{$ numbers $\geq 0\}$

It is not – the sets p_1 and p_2 overlap (they both include 0) – so they are not *disjoint*, and can't be valid partitions.

Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{ \text{ numbers } < 0\}$
- $p_2 = \{0\}$
- $p_3 = \{ \text{ numbers } > 0\}$

Partitions

Is the following a valid partitioning of the integers?

- $p_1 = \{$ numbers $< 0\}$
- $p_2 = \{0\}$
- $p_3 = \{$ numbers $> 0\}$

It is – the sets p_1, p_2 and p_3 cover the domain (nothing is left out), and none of them overlap each other.

Partitions

Suppose we have some parameter `Integer n` that we're trying to partition.

We divide the domain of `n` up into positive numbers, negative numbers, and 0. Is that a partition?

Partitions

Suppose we have some parameter `Integer n` that we're trying to partition.

We divide the domain of `n` up into positive numbers, negative numbers, and 0. Is that a partition?

It is not. `Integer` is what's called a *reference type* in Java. Whereas an `int` represents a concrete 4 bytes of memory, an `Integer` is a "pointer" to some bytes of memory residing ... "elsewhere". (Technically, on the *heap*.)

It can be positive, negative, or zero, but it can also take on the value **`null`**.

Partitions

- $p_1 = \{ \text{ null } \}$
- $p_2 =$ not null; the union of
  - $r_1 = \{ \text{ numbers } < 0\}$
  - $r_2 = \{0\}$
  - $r_3 = \{ \text{ numbers } > 0\}$

Do we need to remember to include the possibility of **`null`** values when testing Java systems?

Partitions

- $p_1 = \{ \text{ null } \}$
- $p_2 =$ not null; the union of
  - $r_1 = \{ \text{ numbers } < 0\}$
  - $r_2 = \{0\}$
  - $r_3 = \{ \text{ numbers } > 0\}$

Do we need to remember to include the possibility of **`null`** values when testing Java systems?

It depends.

In *most* cases, we can assume that a **`null`** shouldn't be passed as a parameter, and that if it *is* passed, the JVM will simply throw a `NullPointerException` at some point.

nulls – the usual case

If that's the case:

- we *don't* bother mentioning this in the method documentation – it's taken as read that **`null`**s are invalid
- we *don't* bother testing this – why would we bother? We wouldn't be testing *our* software, we'd be effectively testing the JVM's ability to detect nulls and throw exceptions. And it's unlikely we have time for that.
- you *shouldn't*, if asked to come up with a useful test case, or a characteristic for partitioning, mention "null-ness" and expect to get marks for it. We will not be impressed.
    - You may wish to mention it *for completeness* – to cover all possibilities. But on its own, we won't consider it a useful partitioning or characteristic.

nulls – unusual cases

Occasionally in Java methods – we probably won't see many of them – **null** has a special meaning.

- e.g. the `java.util.TreeMap<K,V>` class allows you to store and look up values of type `V` using "keys" of type `K`. (You might for instance store student's marks in a `TreeMap<Student,Double>`.)
- The documentation for `java.util.TreeMap.get` says:

```
/** Returns the value to which this TreeMap maps the
  * specified key. Returns null if the TreeMap contains
  * no mapping for this key.
  */
public V get(K key);
```

nulls – unusual cases

```
/** Returns the value to which this TreeMap maps the
  * specified key. Returns null if the TreeMap contains
  * no mapping for this key.
  */
public V get(K key);
```

So in this case, **null** is a value that is intentionally returned, and some of the other TreeMap methods intentionally take it as an argument value.

(As it happens, that was probably a poor choice on the part of the Java library designers, and languages like Python and C++ and Rust do things differently, but that's by the by.)

Characteristics

A characteristic is just some property of an input value which can be used to partition the domain of the value.

Some examples:

- *evenness* is a characteristic of `ints`, the partitions them into *even* and *odd*.
- *signedness* is the characteristic of `intss` that partitions them into *positive*, *negative*, and *zero* (no sign).
- *nullness* is a characteristic of reference types that partitions them into being either **`null`** or not-**`null`**. And there are probably sub-partitions within the latter.
    - (But as noted before: unless **`null`** values are significant for a method or component we're looking at, we will not expect to mention nullness.)
- *all-caps-ness* is a characteristic of non-**`null`** strings that divides them into those strings that are wholly capitalized, and those that are not.

Characteristics

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

What about the our `findElement` method?
(Both its arguments *could* be `null` – but we'll ignore that.)

What are some properties of *lists* that we could partition on?

Characteristics

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

What about the our `findElement` method?
(Both its arguments *could* be `null` – but we'll ignore that.)

What are some properties of *lists* that we could partition on?

Some possible characteristics:

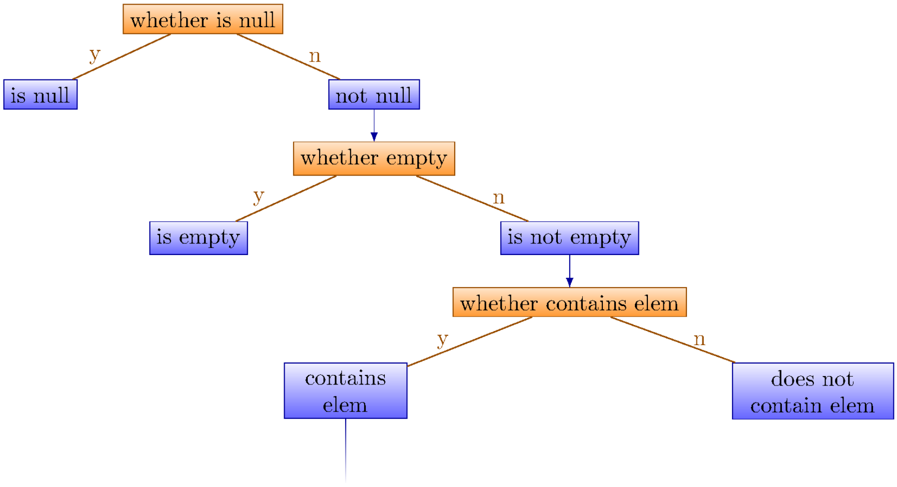
- "is empty" – not the same as nullness! We can have a list that is not null (it has been properly created), but no elements have been added to it yet.
- "contains the element `elem`" – this actually is a characteristic of *both* parameters in combination – that's okay, it's allowed.
- "contains the element `elem` more than once" – this divides the domain into "lists containing `elem` at least twice" and "lists containing `elem` 0 or 1 times".

More characteristics

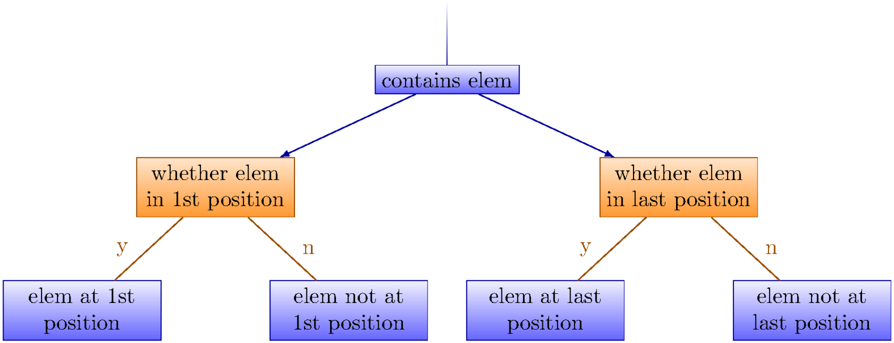
```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

- We might consider the partition of "lists that contain the element `elem`, and decided to *sub*-partition it.
- We could use as a characteristic: "contains the element `elem`, as the first element of `list`".
- In fact while we're at it, we might as well add as a characteristic: "contains the element `elem`, as the last element of `list`.
- What would've made us come up with those two characteristics? The fact that we know programmers tend to make errors around *boundaries*, and the first and last positions form the boundaries of the set of valid positions.

Our characteristics



Our characteristics


contains elem
whether elem in 1st position
whether elem in last position
y
n
y
n
elem at 1st position
elem not at 1st position
elem at last position
elem not at last position

More characteristics

- And there are other characteristics we might come up with.
- For instance: what happens if the element appears *more* than once? Presumably it shouldn't make a difference, but it doesn't hurt to check.

Bad characteristics

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Suppose we have some program which sorts items in a file F
- We might pick as a characteristic of F, "the ordering of the file", and partition it into three partitions:

  p1 = sorted in ascending order
  p2 = sorted in descending order
  p3 = arbitrary order

Bad characteristics

- But is this really a partitioning?

  What if the file is of length 1?
  The file will be in all three blocks . . .
  That is, disjointness is not satisfied

Bad characteristics

Solution:
Each characteristic should address just one property

- File F sorted ascending
  - b1 = true
  - b2 = false
- File F sorted descending
  - b1 = true
  - b2 = false

In general, it's better to have *many* characteristics, each of which partitions its domain into just a few partitions, than to try and have only a few large and complex characteristics.

Bad characteristics

If we decide we've come up with more characteristics than we want – then we can always ignore a few.

But complex characteristics lead more easily to mistakes, and it is harder to spot and fix those.

Properties of Partitions

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design attempt
- Different alternatives should be considered

ISP review

Review of steps

Let's review the steps in applying the ISP technique:

- Identifying testable functions
- For each function, find all the parameters
- Model the input domain in terms of *characteristics*
- Choose particular partitions, and values from within those partitions
- Refine into test values

We'll now look at these in a bit more detail.

Step 1 – Identifying testable functions

Recall that we can apply the ISP technique to methods (or functions or procedures, in languages other than Java), classes, components, programs, systems - anything we can treat as function-like.

- Individual methods or functions normally have one testable function
- Classes will have multiple testable functions
- Whole programs and larger systems may have many functions, and complex characteristics – modelling and design documents such as UML use cases or user stories can be useful here
- Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc

Step 2 – Find all the parameters

- Often fairly straightforward
- Important to be complete, though

Applied to different levels:

- Methods: Actual method parameters, plus *state* used
  - *state* includes: state of the current object; global variables; files etc. read from
- Components: Parameters to methods, plus relevant state
- System: All inputs, including files and databases

Step 3 – Model the input domain

- We need to characterise the input domain, and divide it into partitions –
  where each partition represents a set of values
- This is a creative design step – different test designers might come up with different ways of modelling the input domain
- . . . and there's not really a mechanical way of checking whether a modelling is "correct" – needs human review.

Step 4 – Choose combinations of values

- So, we've come up with our characteristics and partitions
- These help us divide up the entire input domain (usually of enormous size) into a much smaller and more tractable set of partitions.
- Can we now simply take all (feasible) combinations of partitions, and write tests?
- Usually not – there'll often be too many partitions to try all combinations.
- *Coverage criteria* are criteria for choosing *subsets* of combinations (more later)

Step 5 – refine combinations into test inputs

- . . . At the end of this step, we have actual test cases.

Input domain modeling

Approaches to Input Domain Modeling

So we said that in step 3, we model the input domain – characterise it and divide it into partitions.

We've done that so far by staring at a method specification and hoping for inspiration.

If we want to try something more principled, there are two general approaches we can take.

Two approaches

1. Interface-based approach
   - Develops characteristics directly from individual input parameters
   - Simplest application
   - Can be partially automated in some situations
2. Functionality-based approach
   - Develops characteristics from a behavioral view of the program
   - under test
   - Harder to develop – requires more design effort
   - May result in better tests, or fewer tests that are as effective

Interface-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
- Ignores relationships among parameters
  - It wouldn't come up with the "is the element in the list?" characteristic we saw for `findElement (List<Integer> list, Integer elem)`

Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Characteristics

- Candidates for characteristics :
  - Preconditions and postconditions
  - Relationships among variables
  - Relationship of variables with special values (zero, null, blank, . . .)
- Better to have more characteristics with few partitions

Interface vs Functionality-Based modelling

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

Interface-Based Approach:

- Two parameters : list, element
- Characteristics:
  list is null (block1 = true, block2 = false)
  list is empty (block1 = true, block2 = false)

Interface vs Functionality-Based modelling

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

Functionality-Based Approach:

- Two parameters : list, element
- Characteristics:
  number of occurrences of element in list
    (0, 1, $>$1)
  element occurs first in list
    (true, false)
  element occurs last in list
    (true, false)

Strategies for modelling

Recall that once we have *partitions*, we'll want to choose particular values from within those partitions.

- Include valid, invalid and special values
- Sub-partition some blocks
- Explore boundaries of domains
- If a value is of an *enumerated type*, can draw from each possible value
- Include values that represent "normal use"
- Try to balance the number of blocks in each characteristic
- Check for completeness and disjointness

Interface-Based IDM example – triType

Suppose we have a method
`String triType(int l1, int l2, int l3)` that takes in the lengths of three sides of a triangle, and returns a string telling us what sort it is.

Possible outputs are:

- “invalid” – not a triangle. E.g. $(1, 1, 5)$, $(-5, 3, 4)$.
- “equilateral” – all sides are the same
- “isosceles” – not equilateral and not invalid, and two sides are the same
- “scalene” – everything else

Interface-Based IDM example – triType

How might we categorize the inputs?
(Applying just the simple interface-based approach.)

Characteristic	$\mathbf{I_1}$	$\mathbf{I_2}$	$\mathbf{I_3}$
q1 = "Rel. of side 1 to 0"	greater than 0	equal to 0	less than 0
q2 = "Rel. of side 2 to 0"	greater than 0	equal to 0	less than 0
q3 = "Rel. of side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of $3 \times 3 \times 3 = 27$ tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests . . .

Functionality-Based IDM – TriTyp

- So this is the *interface* based approach – just looks at parameters and types
- A semantic level characterization could use the fact that the three integers represent a triangle
  - The combination of parameters $(1, 1, 2)$ represents exactly the same triangle as $(1, 2, 1)$ and $(2, 1, 1)$.
  - (For the math-inclined – we're looking for, and finding ways to ignore, *symmetries* in the input domain)

Characteristic	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$\mathbf{p_4}$
q1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Using more than one modelling

- Some programs may have dozens or even hundreds of parameters
- Create several small IDMs
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of rigor
  - For example, some IDMs may include a lot of invalid values
- It is okay if the different IDMs overlap
  - The same variable may appear in more than one IDM

Step 4 – Choosing Combinations of Values

- Once characteristics and partitions are defined, the next step is to choose test values
- We use criteria – to choose effective subsets
- An obvious criterion is to choose all combinations . . .

  All Combinations (ACoC): All combinations of blocks from all characteristics must be used.
- Number of tests is the product of the number of blocks in each
  - This will often be far too large – we will look at ways of using fewer.

Test criteria

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time
- When continued testing causes no new failures

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time
- When continued testing causes no new failures
- When continued testing reveals no new faults

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time
- When continued testing causes no new failures
- When continued testing reveals no new faults
- When we cannot think of any new test cases

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time
- When continued testing causes no new failures
- When continued testing reveals no new faults
- When we cannot think of any new test cases
- When some specified *test coverage* level has been attained

When to stop testing

How do we know when we have tested enough? When should we stop testing? How many tests do we need?

Some possibilities:

- When all faults have been removed
- When we run out of time
- When continued testing causes no new failures
- When continued testing reveals no new faults
- When we cannot think of any new test cases
- When some specified *test coverage* level has been attained
- When we reach a point of diminishing returns

When to stop testing

Some other possibilities:

- Fault seeding: We deliberately implant a certain number of faults in a program. If our tests reveal x% of the implanted faults, we assume they have also only revealed x% of the original faults; and if our tests reveal 100% of the implanted faults, we are more confident that our tests are adequate.

  (What assumptions are we making here?)

When to stop testing

other possibilities, cont'd:

- Mutation testing: We *mutate* parts of our program (e.g. altering constants, negating conditionals in loops and "if" statements). Overwhelmingly, our new mutated program should be *wrong*; if no tests identify at as such, we may need more tests.

  (And if some of our tests never seem to kill mutated programs, they may be ineffective.)

When to stop testing

other possibilities, cont'd:

- Risk-based: We identify *risks* to our project, and put in place strategies (including testing) to mitigate or reduce those risks.

  We estimate the effort required for those strategies, and their likely pay-off, and stop when the risk has been reduced to whatever we consider a tolerable level.

(Also applies to "How formally should we specify our system?")

When to stop testing

other possibilities, cont'd:

- Risk-based: We identify *risks* to our project, and put in place strategies (including testing) to mitigate or reduce those risks.

  We estimate the effort required for those strategies, and their likely pay-off, and stop when the risk has been reduced to whatever we consider a tolerable level.

(Also applies to "How formally should we specify our system?")

(In fact, applies to almost every question of the from "How much X should we do?
Answer: Enough to bring the risks to a tolerable level.)

Test coverage

- Sometimes test plans will specify that tests ought to have some specified level of *coverage* of the code.
- *Test coverage* is some measure of the extent to which the source code of a program has been executed when a particular test suit runs.
- Coverage is often measured using *test coverage tools*.

Test coverage tools

- How do test coverage tools work?
- Typically, they do what is called *instrumenting* the code in some way – adding extra instructions which record how many times some piece of code has been executed.
- This might be done at the source code level, but more often is done at the byte-code or machine-code level.

Java test coverage tools

Some common test coverage tools for Java include:

- JCov
- Cobertura
- OpenClover

Java test coverage example

Suppose we want to record test coverage using JCov. The steps are:

- Compile code as normal (e.g. using `javac`, an IDE, or a build tool such as `ant`)
- "Instrument" the compiled bytecode:

```
$ java -jar jcov.jar Instr [class1.class class2.class ...]
```

- Run our program (or, some test suite). This produces a `result.xml` file.

```
$ java -classpath jcov_file_saver.jar:. MyProg
```

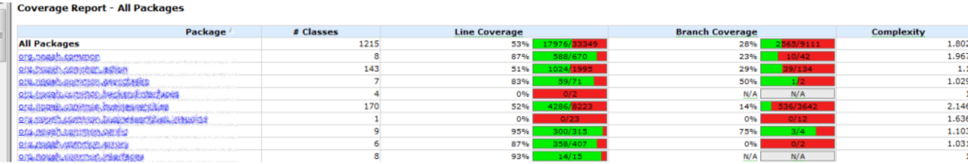
Java test coverage example, cont'd

- Generate a report from the XML file

```
$ java -jar jcov.jar RepGen result.xml
```

Code coverage reports

Code coverage results are often produced in HTML format, or displayed in the IDE. Fragment of a sample report from Cobertura:



Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	1215	53%	17976/33349	28%	2565/9111	1.802
[illegible]	8	87%	588/670	23%	10/42	1.967
[illegible]	143	51%	1024/1995	29%	39/134	1.1
[illegible]	7	83%	59/71	50%	1/2	1.029
[illegible]	4	0%	0/2	N/A	N/A	1
[illegible]	170	52%	4286/8223	14%	536/3642	2.146
[illegible]	1	0%	0/23	0%	0/12	1.636
[illegible]	9	95%	300/315	75%	3/4	1.103
[illegible]	6	87%	358/407	0%	0/2	1.031
[illegible]	8	93%	14/15	N/A	N/A	1

Code coverage reports

Typical measures of coverage given by code coverage tools are:

- Line coverage (% of lines executed)
- Branch coverage (% of branches taken)
- Method coverage (% of methods executed)
- Condition or predicate coverage (% of boolean conditions evaluated to both true and false)

Custom code coverage

What if we want to calculate some code coverage measure which our tool doesn't supply by default?
For instance, "prime path coverage" (which we will see in the lecture on graph-based testing) is not usually one of them.

Custom code coverage

What if we want to calculate some code coverage measure which our tool doesn't supply by default?
For instance, "prime path coverage" (which we will see in the lecture on graph-based testing) is not usually one of them.

Some tools provide an API which lets us write our own custom measures of coverage – for instance, JCov does this.

Limits of code coverage tools

- Code coverage tools give us measures of coverage based on *source code*.
- But sometimes our tests aren't based on source code as a model
- For instance, we might be writing tests based on a state chart or activity diagram of the system.
- And Input Space Partitioning isn't based on *source code*, exactly – it's based on *specifications* for some view of the system (or a part of it) as a *function*. Knowing how many functions or methods were executed as a result of our ISP-based tests isn't a great measure of what degree of coverage the tests provide of the input domain.

General coverage criteria

- Therefore, we want more general measures of coverage, which can be applied to things other than source code.
- For each of the types of model-based testing covered in this course (ISP, graph-based, logic-based, syntax-based) we will also look at coverage criteria which let us estimate how throrough our tests are.
- Our coverage calculations will largely be manual, in this case, since we have no equivalent of a "code coverage" tool to tell us (say) when paths through an activity diagram have been thoroughly executed.

ISP criteria

ISP criteria

- We'll illustrate our criteria using the idea of a program which classifies triangles, based on their edge lengths (this is an old example in the testing literature)

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }

public Triangle triType (int side1, int side2, int side3)
```

ISP criteria – interface approach

```
public Triangle triType (int side1, int side2, int side3)
```

- Simply considering the parameters alone doesn't give us much help.
- We might come up with a characteristic for each, namely, "How does it compare with 0?", and partition the domain by asking "Is the parameter less tham, equal to, or great than 0?"

ISP criteria – functionality-based approach

- A better approach is to consider the *semantics* (functionality) of the method.
- It deals, after all with *triangles*.
- => model the input space in terms of that
- The *order* of parameters is not important, rather their relation is.

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

  - scalene

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

  - scalene
  - isosceles

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

  - scalene
  - isosceles
  - equilateral

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

  - scalene
  - isosceles
  - equilateral
  - invalid

ISP criteria – functionality-based approach

- One attempt:

  Partition the input domain using a geometric classification: do the parameters represent a triangle which is

  - scalene
  - isosceles
  - equilateral
  - invalid
- What's the problem here?

ISP criteria – functionality-based approach

- Equilateral triangles are a *subset* of isosceles triangles - our “partitions” are not disjoint.
- Refine the partitions to:
    - scalene
    - non-equilateral isosceles
    - equilateral
    - invalid

ISP criteria – functionality-based approach

- We might then come up with some inputs which fall into each partition:

geometric type	**input value**
sca	(4,5,6)
iso	(3,3,4)
equ	(3,3,3)
inv	(3,4,8)

ISP criteria – functionality-based approach

- The guideline of “prefer more characteristics, with few partitions” on the other hand, suggests the following:

characteristic	**partitions**
is scalene	(T,F)
is isosceles	(T,F)
is equilateral	(T,F)
is invalid	(T,F)

ISP criteria – all combinations

- How many values should we choose?
- One possibility: "all combinations" (ACoC)
  - The number of tests would be (no. of partitions for char. 1) * (no. of partitions for char. 2) * ...
- If we used the interface approach (partitioning each parameter by whether it is less than, equal to, or greater than 0) we get 3 blocks with 3 partitions, so the no. of tests is 3 * 3 * 3 = 27 – Probably more than we would like.
- Using the functionality approach ...
  - We will end up with *constraints* which rule out some combinations. *If* a triangle is scalene, it follows it can't be isosceles, equilateral, or invalid
  - We'll end up with only 8 tests (much more tractable)

ISP criteria – all combinations

Suppose we have a method `myMethod(boolean a, int b, int c)`, and we partition the paramaters as follows:

- the boolean into `true` and `false` (let's call these partitions T and F)
- parameter b into "> 0", "< 0" and "equal to zero" (let's call these partitions LTZ, GTZ, and EQZ)
- parameter c into "even" and "odd" (let's call these EVEN and ODD).

ISP criteria – all combinations

Using the “all combinations” criterion, we’d need to write

$$|\{T, F\}| \times |\{LTZ, GTZ, EQZ\}| \times |\{EVEN, ODD\}|$$
$$= 2 \times 3 \times 2$$
$$= 12 \text{ tests.}$$

Often this will be far more than is feasible.

ISP criteria – base choice

- *Base choice* criteria recognize that some values are important – they make use of domain knowledge of the program.
- For each characteristic, we choose a *base choice* partition, and construct a *base* test by using all the base choice values.
- Then we construct subsequent tests by holding all but one base choice constant, and varying just *one* characteristic (using all the partitions for that characteristic)
- Number of tests is one base test + one test for each other partition:

  $1 + (|char_1| - 1) + (|char_2| - 1)...$

ISP criteria – base choice

Considering our `myMethod(boolean a, int b, int c)` and the partitions we specified, if we made our base choices T, GTZ and $EVEN$, the required tests would be:

- $(T, GTZ, EVEN)$
- $(F, GTZ, EVEN)$ (vary first parameter)
- $(T, LTZ, EVEN)$ (vary second parameter)
- $(T, EQZ, EVEN)$ (vary second parameter)
- (T, GTZ, ODD) (vary third parameter)

ISP criteria – base choice

How do we choose a “base choice”?

- must be feasible

Could be:

- most likely from an end-use point of view
- simplest
- smallest
- first in some ordering

Test designers should document why a particular base choice was made

ISP criteria – multiple base choice

- Sometimes there are multiple plausible choices for a base choice.
- Multiple Base Choice (MBC):
  One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.
- e.g. For the interface-based approach to the triTyp method, we might decide both (2,2,2) and (1,1,1) are good base choices.

ISP criteria – multiple base choice

- Base choice (2,2,2):

  (-**1**,2,2), (**0**,2,2)
  (2,-**1**,2), (2,**0**,2)
  (2,2,-**1**), (2,2,**0**)

- Base choice (1,1,1):

  (-**1**,1,1), (**0**,1,1)
  (1,-**1**,1), (1,**0**,1)
  (1,1,-**1**), (1,1,**0**)

ISP criteria – constraints

- Sometimes combinations of partitions are infeasible (e.g. the functionality-based case for triangles)
- For “all combinations” as a criterion, we simply drop infeasible combinations
- For Base Choice and Multiple Base Choice – we change a base value to a non-base one to find a feasible combination.