CITS5501 Software Testing and Quality Assurance
Input Space Partition Testing

Unit coordinator: Arran Stewart

Highlights

- How we choose values for tests
- Approaches to testing
- Model-based testing
- Input space partitioning

Approaches to testing

Approaches to testing

For most software components (and other artifacts, such as machinery, etc.), it's possible to consider them in two ways, when testing:

- knowing *nothing* about the internal workings of the component, we can focus on its intended functionality, and conduct tests that demonstrate each aspect of the functionality, and attempt to uncover any errors.
  - This approach is called "black-box" testing
- knowing the internal workings of the components, we can write tests that try to check the internal operations are correctly performed, and that all internal components have been adequately exercised.
  - This approach is called "white-box" testing

In reality, many testing approaches make use of aspects of both.

Approaches to testing

- Example of “black-box” testing:
    - The sorts of unit tests we have seen so far: they are derived from the *specifications* for methods, and treat the method as a “black box” that takes in input and produces output, without considering how it does it.
- Example of “white-box” testing:
    - Looking at the source code for a method, and ensuring that *paths* of execution through the method have been adequately tested.

Black-box testing

Signature of method, plus specification using Javadoc:

```
/** Remove/collapse multiple spaces.
*
* @param String string to remove multiple spaces from.
* @return String */
public static String collapseSpaces(String argStr)
```

Black-box testing

- Specifications need not be for *methods*, they can be for software components, or hardware, or whole systems

White-box testing

A Java method for collapsing sequences of blanks, taken from the `StringUtils` class of Apache Velocity (http://velocity.apache.org/), version 1.3.1.

```java
/** Remove/collapse multiple spaces.
*
* @param String string to remove multiple spaces from.
* @return String */
public static String collapseSpaces(String argStr) {
  char last = argStr.charAt(0);
  StringBuffer argBuf = new StringBuffer();
  for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++) {
    char ch = argStr.charAt(cIdx);
    if (ch != '␣' || last != '␣') {
      argBuf.append(ch);
      last = ch;
    }
  }
  return argBuf.toString();
}
```

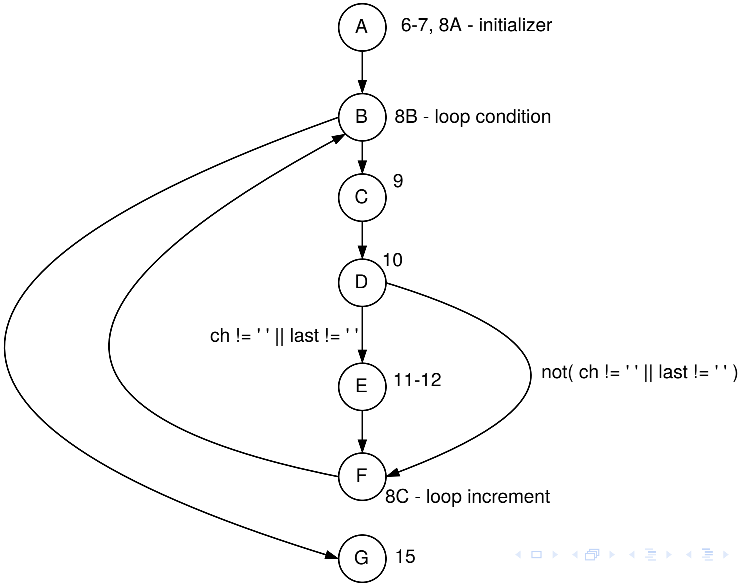
Control-flow testing outline

1. Use the source code (or pseudocode) to produce a control flow graph.
2. Using the graph produce a set of tests for the given program.

Constructing the graph

- In a control flow graph, nodes represent points in the program control flow can go “from” or “to”
- Loops, thrown exceptions and gotos (in languages that have them) are locations control flow can go *from* – statements representing these spots are “sources”
- Locations control flow can go *to* are “sinks”

Constructing the graph



Black-box techniques

- When we design tests based on the interface – “black-box” testing – we normally work off the *specification* for the item, we don’t care about the details of the implementation
- Input space partition testing (this lecture). We don’t need to look at the code *within* an item being tested – we just consider its parameters or inputs.

Other black-box techniques

- In the Pressman textbook you'll see mention of other black box techniques, e.g. "boundary value analysis"
  - i.e., include tests which have inputs at the "boundaries" of ranges of values
  - this helps detect, for instance "off-by-one" and "fencepost" errors
- Boundary value analysis is actually incorporated into the ISP testing procedure covered in this lecture
  - when "modeling the input domain", we identify valid values, invalid, boundaries, "normal use", and so on

Benefits of black box testing

- Helps find
  - functionality that is specified but not implemented
  - functionality that is implemented but incorrect

White-box testing

- We can also design tests by looking at the *internal* details of an item to be tested – “white-box” or “clear-box” testing.
- This is also sometimes called *structural testing*, since it looks at the internal structure of an item to be tested
- Here, we do care about the implementation

White-box testing – examples

- As part of white box testing, we might try to ensure that
  - all internal data structures have been checked
  - all loops have been checked
  - where there is some sort of branching statement (if-else, case, etc.), all the possible branches have been tested
  - . . . and so on.

Why perform white-box testing

- Why perform white-box testing?
  - Isn't black box enough? – after all, it tests the functionality

Why perform white-box testing (2)

- What if we've failed to identity some particular scenario (set of inputs) in black box testing, and not written a test for it?
  - It can be difficult to think of unusual inputs/scenarios
- What if the environment, or some other part of the system, changes?
  - code that was previously "dead code", and never executed, might now become "live" – and may contain errors
- Some sorts of errors (e.g. typos) are as likely to occur on unusual or uncommon paths of execution, as on anywhere else.
  - White box testing helps ensure we've considered those paths.

Why perform white-box testing (3)

- One question that is often asked is “Do we have enough tests?”
- White box testing may not answer that question – but it can identify parts of a system that *haven't* been tested.

Types of white-box testing

- In practice and in the literature, many different techniques are identified:
  - branch/decision testing
    - have all branches in decisions been exercised?
    - have all parts of boolean expressions been exercised?
  - control flow testing
    - uses a program's control flow graph as a model
  - data flow testing
    - flow of data between variables – are there variables that are declared but not used, or vice versa? Declared multiply? Not initialized before use? Deallocated before use? Used before being validated?
  - statement coverage
    - is every statement executed at least once?
  - modified condition/decision coverage (used in avionics)
  - path testing
  - prime path testing

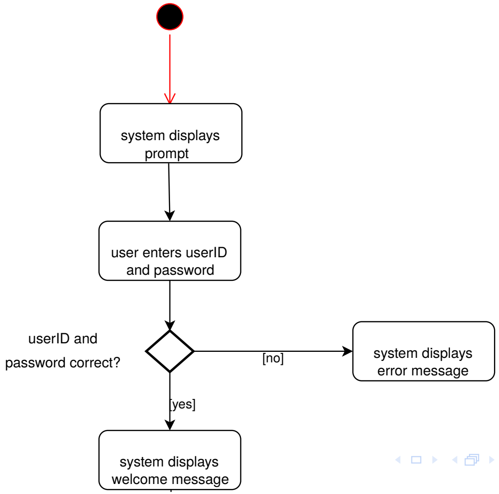
Alternative view – model-based testing

When doing white-box testing of the `collapseSpaces` function, we look at the control-flow *graph* for the function, and try to ensure our tests adequately exercise paths through the graph (called checking the *test coverage* of the graph).

But there are many other sorts of "graphs" we might want to check for test coverage, and not all are "internal", "white-box" views of something.

Activity diagrams

For instance, *activity diagrams* are way of modelling a user's interactions with a system.


system displays prompt
user enters userID and password
userID and password correct?
[no]
system displays error message
[yes]
system displays welcome message

Activity diagrams

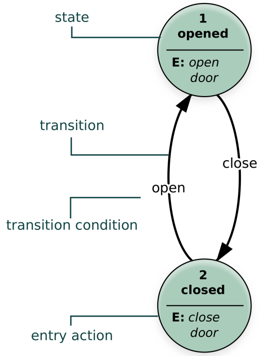
These too form a sort of graph, and we can ask whether out tests have exercised paths through the graph sufficiently.

Activity diagrams don't look at "source code" or the "inside" of a system – they consider the "outside" (a user's interaction with the system).

So they are a sort of "black-box" testing, yet the same methods we use for control-flow analysis – a form of "white-box" testing – are applicable.

State diagrams

State diagrams show states something can be in, and transitions between them.[1]



[1] Courtesy Wikipedia, https://en.wikipedia.org/wiki/State_diagram.

State diagrams

A state diagram also is a kind of graph, so we can look at whether our tests have exercised paths through it sufficiently.

Is it “black-box” or “white-box” testing?

Alternative view – model-based testing

Rather than classifying something as being “black-box” or “white-box” testing, a more useful approach is to consider various *models* of a software system, and ask “What sort of model is this? And what sort of testing techniques can be applied?”

Model-based testing – functions

- If we can treat the model as a *function* from inputs to outputs – then we can apply *input-space partitioning* to it.
    - Example: Unit tests based on Javadoc specification
    - Example: System testing based on specifications

Model-based testing – graphs

- If we can treat the model as a *graph* – a network of nodes – then we can apply *graph-based* techniques to it.
    - Example: Control flow analysis

Model-based testing – logic

If particular parts of the system make “choices” based on combinations of logical conditions, we can apply *logic-based* techniques to it.

- Example: Avionics systems are required to have a particular level of coverage of logic expressions
- Sample specification for a system [from Ammann]:

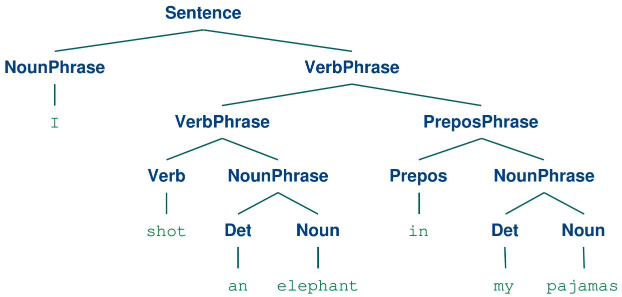
  If the moon is full and the sky is clear, release the monster.
  If the sky is clear and the wind is calm, release the monster.

Model-based testing – logic vs graphs

- Graph-based techniques look at what *edges* we traverse between nodes, they don't look "inside" the nodes –
- For any "decision node", however complex, graph-based techniques only consider "Which edge do we take out of the node?"
- By contrast, logic-based testing looks "inside" the parts of boolean expressions making up a "decision point", and asks whether we've tested those parts sufficiently thoroughly.

Model-based testing – syntax

- If the model can be treated as having a "syntax" (a sort of tree-like, potentially recursive structure), then we can apply *syntax-based* techniques to it.
- One example of things with "syntax" is, unsurprisingly, natural language sentences:[2]



[2]Diagram adapted from Bird *et al* (2009). Dialogue from "Animal Crackers" (1930, dir. V. Heerman).

Model-based testing – syntax

But other things that can be modelled as having a syntax are things like Java source code (a text format), or binary file formats (such as PNG graphics files or executable files).

Model-based testing

Our “models” don’t have to be models of source code – they can be models of, say, database structure, or user interaction with a system, or class hierarchies, or any other way we find it useful to consider our system (or some part of it).

Input Space Partitioning

Problem – how to choose test values

- An example Java method we might want to test:

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

- What are the possible values for `list`?
  For `elem`?

Input Domains

- The input domain for a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about choosing finite sets of values from the input domain
- *Input parameters* define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Benefits of ISP

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

Input parameters

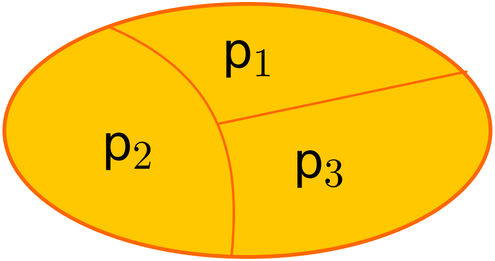
- N.B.: "Input parameters", when used to define the cope of the input domain, cover much *more* than just "*parameters* to a method" –
  and "tests" covers more than just "unit tests"
- Suppose we are doing final system tests of a binary program that reads an input file – then the file constitutes an input parameter.
- If the behaviour of a method depends on the values of instance variables – then those instance variables are also input parameters.
- Input parameters include all values that can affect the behaviour of the item being tested.

Partitioning domains

- 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)



Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What constitues the input domain here?

- The set of pairs (l, e), where l is drawn from all possible values for `list`, and e is drawn from all possible values for `elem`

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What constitues the input domain here?

- The set of pairs (l, e), where l is drawn from all possible values for `list`, and e is drawn from all possible values for `elem`
- What are the possible values for `elem`?

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What constitues the input domain here?

- The set of pairs (l, e), where l is drawn from all possible values for `list`, and e is drawn from all possible values for `elem`
- What are the possible values for `elem`?
  - it could be `null`

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What constitues the input domain here?

- The set of pairs (l, e), where l is drawn from all possible values for `list`, and e is drawn from all possible values for `elem`
- What are the possible values for `elem`?
    - it could be `null`
    - it could be `non-null`

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What constitues the input domain here?

- The set of pairs (l, e), where l is drawn from all possible values for `list`, and e is drawn from all possible values for `elem`
- What are the possible values for `elem`?
  - it could be `null`
  - it could be non-`null`
    - if it's non-`null`, it could be 0, 1, -1, 2, -2, . . . (2^{32} distinct values)

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

What about the values for `list`?

- `list` could be `null`, or it could be non-`null`
- if it is not null, it could have 0 members, 1, 2, 3, . . .
- each of those members could then be `null` or non-`null`, and if non-`null`, could take on values 0, 1, -1, 2, -2, . . . (2^{32} distinct values)

Input domain example

```
public boolean findElement (List<Integer> list, Integer elem)
// Effects:
//  if list or elem is null throw NullPointerException
//  else return true if elem is in the list, false otherwise
```

So our input domain is infinite.[3] It's true that any one `Integer` can only take on a finite set of values (either `null`, or 2^{32} possible non-null values), but `List`s can be of arbitrary length.

[3]Or at least, we can treat it as such, for now. We are likely to run into problems with lists of more than 2^{31} members.

Input domain – enum

How many possible value does the following function accept?

```
enum Weekday {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY
}

// ...

public static isWorkday(Weekday w) {
  // ...
}
```

Input domain – classes

Java's other main way of modelling "alternatives" is to use classes:

```
class Person  { /* ... */ }
class Student { /* ... */ }
class Staff   { /* ... */ }
```

Overview of input-space partitioning approach

Here's what we'll be doing:

1. Identify testable functions
2. Identify all *parameters* to the functions (easy for functions or methods that don't depend on state)
3. Model the input domain in terms of *characteristics*, each of which can be partitioned.

   (Two general ways of doing this – interface-based and functionality-based)
4. Choose particular partitions, and values from within those partitions
5. Refine into test values
6. Review!

Characteristics

- A characteristic is just some property of an input value which can be used to partition the domain of the value.
- For instance, of a list parameter to a method, we might consider the following characteristics:
  - Is the list null, yes or no?
    - This partitions the domain into two
  - Is the list empty, yes or no?
    - Again, this partitions the domain into two

Characteristics, cont'd

- If we were considering the `findElement` method from earlier, some other characteristics we might consider are:
  - How many occurrences are there of `elem` in `list`?
    - We might partition this into "0 times", "1", "more than 1"
  - Does `elem` occur as the first element of `list`, yes or no?
    - This partitions the domain into two
  - Does `elem` occur as the last element of `list`, yes or no?
    - This partitions the domain into two

Characteristics, cont’d

- 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

Characteristics, cont'd

- 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

Characteristics, cont'd

Solution:
Each characteristic should address just one property

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

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

Identifying testable functions

- This means functions in the sense of mappings from a domain to results.
- Can apply to methods, classes, components, programs, systems
  - We can treat a whole program (or hardware+software system) as a *function* in this sense – "If I supply these particular inputs, what output do I get?"

Step 1 – Identifying testable functions

- Individual methods or functions have one testable function
- Classes will have multiple ‘testable functions”
- Programs have more complicated characteristics – modeling documents such as UML use cases can be used to design characteristics
- Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc

Step 2 – Find all the parameters2

- Often fairly straightforward, even mechanical
- 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

- Each test input has possible values, which we've partitioned
- But even considering all the combinations of partitions, we end up with a very large number
- *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

Two Approaches to Input Domain Modeling

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

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

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

Interface-Based Approach:

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

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)

Modeling the Input Domain

- Strategies for obtaining partitions from a characteristic:
  - 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?

Characteristic	**I_1**	**I_2**	**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

- Previous example is *interface* based – just looks at parameters and types
- A semantic level characterization could use the fact that the three integers represent a triangle

Characteristic	p1	p2	p3	p4
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

Covered more later.

- 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?")

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
```

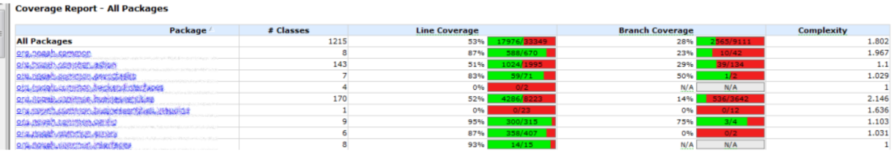
Java test coverage example, cont'd

- 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
org.noosh.common	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:

  $1 + (|char_1| - 1) + (|char_2| - 1) \ldots$

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.

References

Bird, S., Klein, E., & Loper, E. (2009) *Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Inc. URL: https://github.com/nltk/nltk_book