Specification-Based Testing 1

Stuart Anderson

Overview

- Basic terminology
- A view of faults through failure
- Systematic versus randomised testing
- A systematic approach to specification-based testing
- A simple example

Terminology

- **Independently Testable Feature (ITF):** depends on the control and observation that is available in the interface to the system – design for testability will focus on providing the means to test important elements independently.
- **Test case:** inputs, environment conditions, expected results.
- **Test case specification:** a property of test cases that identifies a class of test cases.
- **Test suite:** a collection of test cases. A test suite for a system may comprise several test suites for each ITF in the system.
- **Test:** the activity of executing a system for a particular test case.

Faults, Errors and Failures

- **Fault:** a mistake in coding that has the potential to give rise to an error.
- **Error:** is a discrepancy between the expected state of the system and its actual state - this is usually inside the system boundary so it can be hard to observe. Faults can give rise to errors, a fault is called active if it has done so and dormant otherwise.
- **Failure:** misbehaviour of the system in an externally observable behaviour. Failures are usually caused by errors but depending on the structure of the code the error may be trapped and repaired or the error may be such that in some cases it does not give rise to a failure.

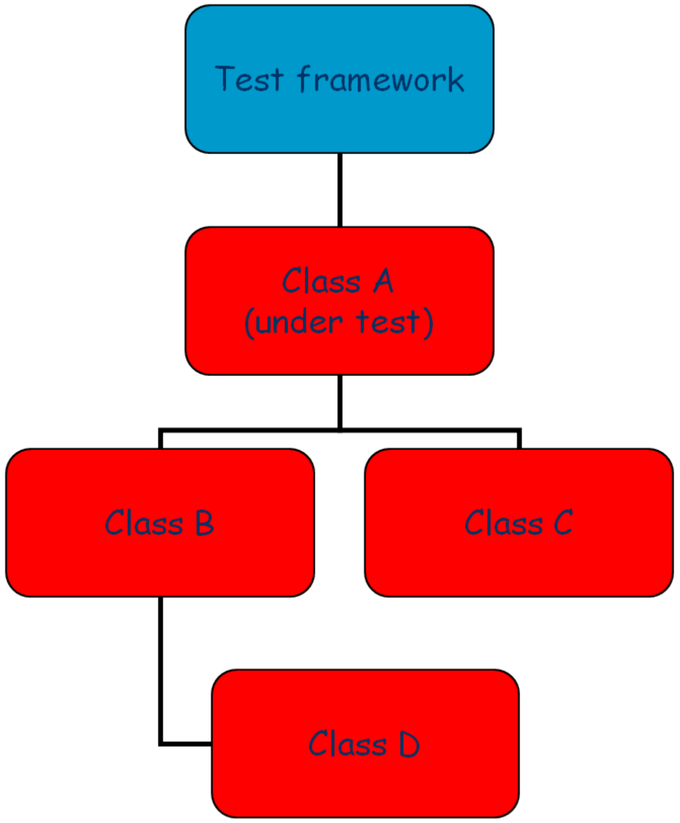
An Example

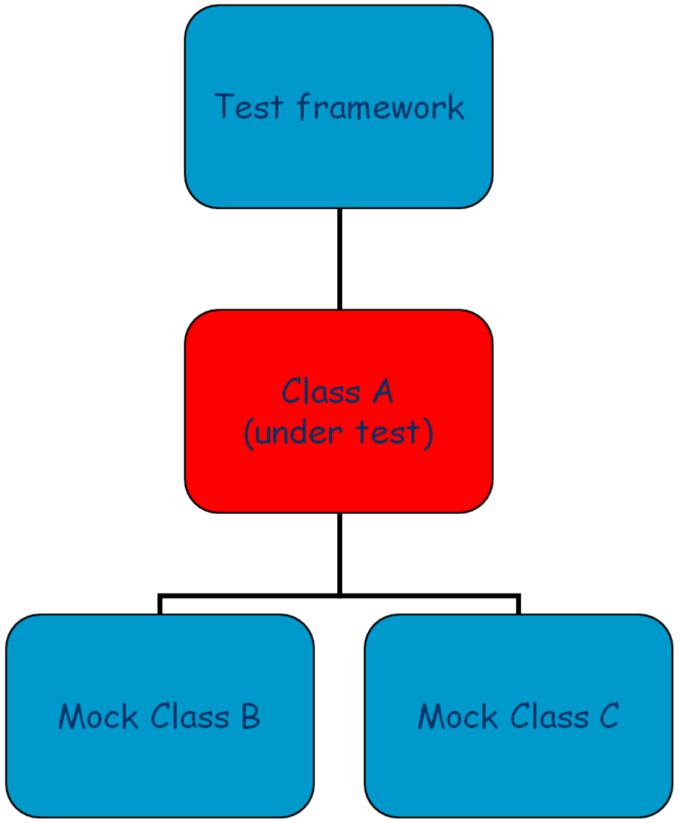
- **Fault:** the programmer forgets properly to dispose of some dynamically allocated store.

- **Error:** if the code is executed then the system state is erroneous because there is store that cannot be recovered as garbage that is not in use.

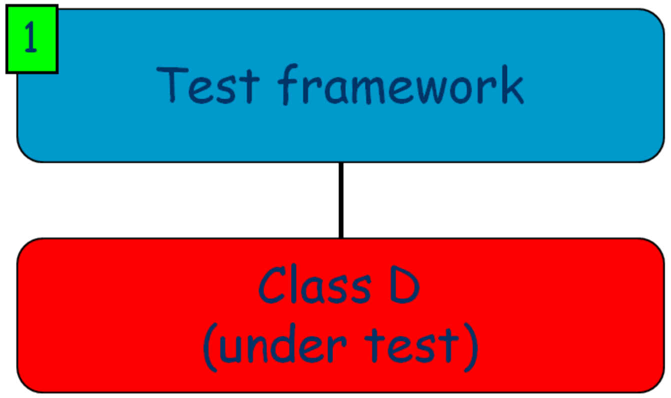
- **Failure:** If the faulty code is executed once only in any execution of the program then the error may never cause a fault. If the fault is executed often in a run of the program it will eventualy cause an out of memory problem and that will either cause the system to fail or the problem may be trapped and cleaned up somehow.

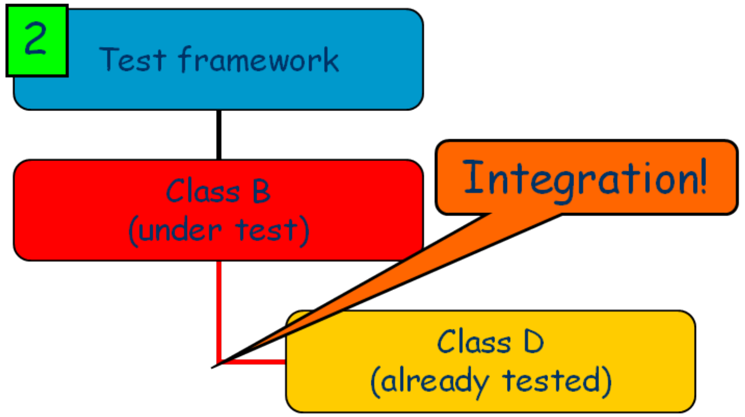
Unit Testing

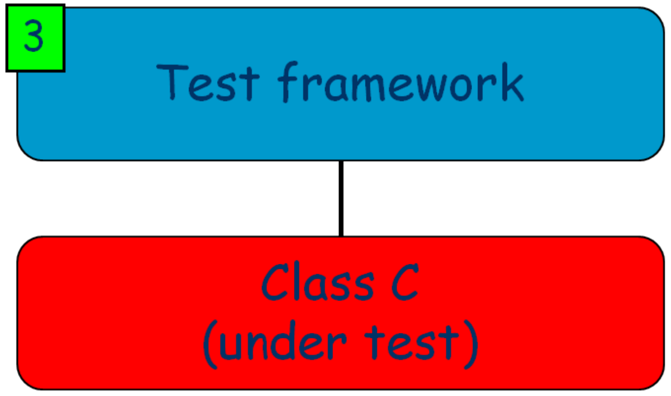


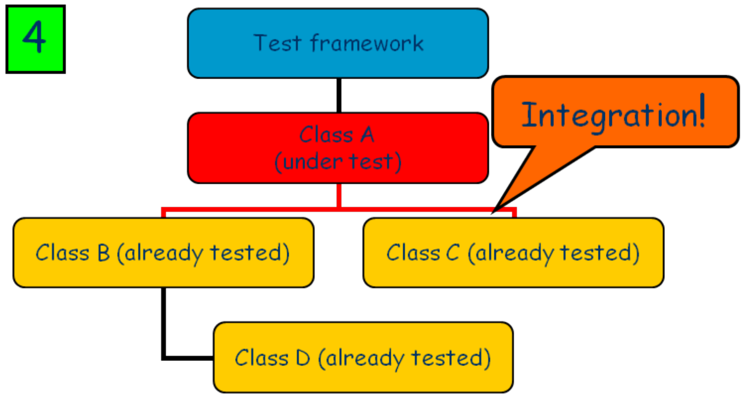


Dependent Tests ($\sim$ integration)

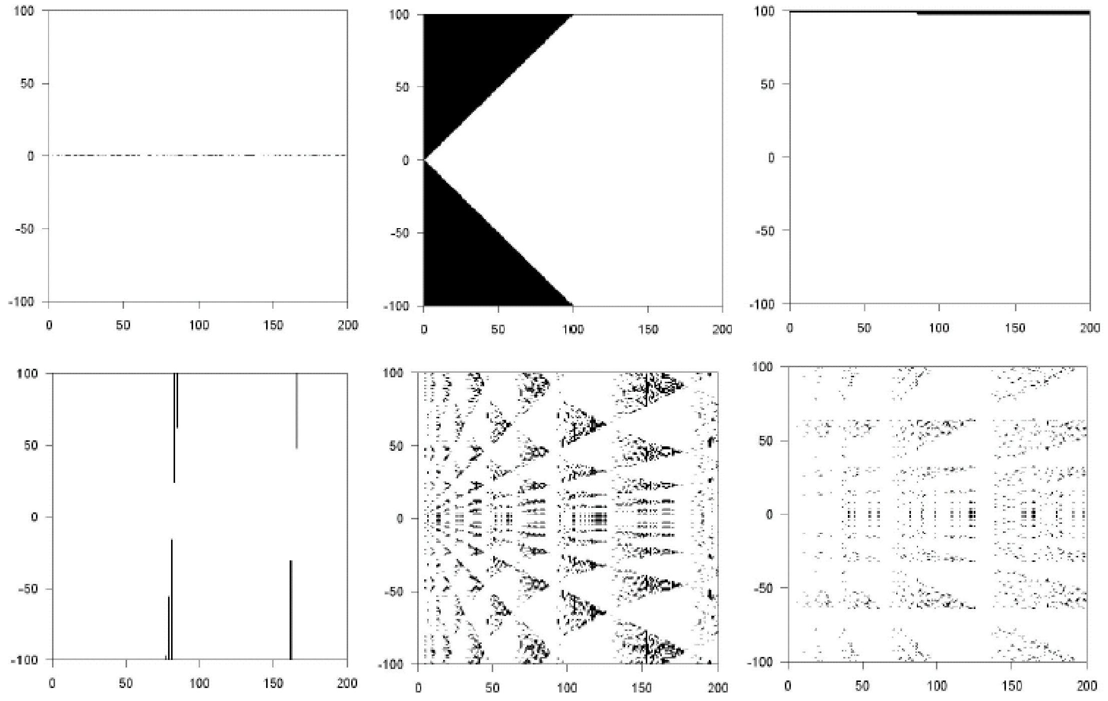








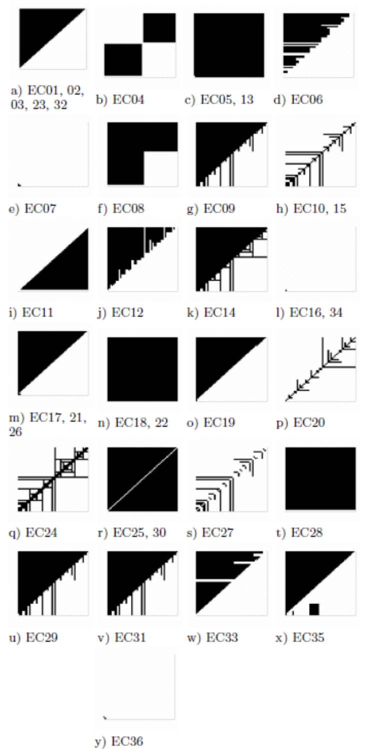
The Shape of Faults



"Failure regions for some of the infrequent equivalent classes"

The Shape of Faults

Some simple faults have regular shapes



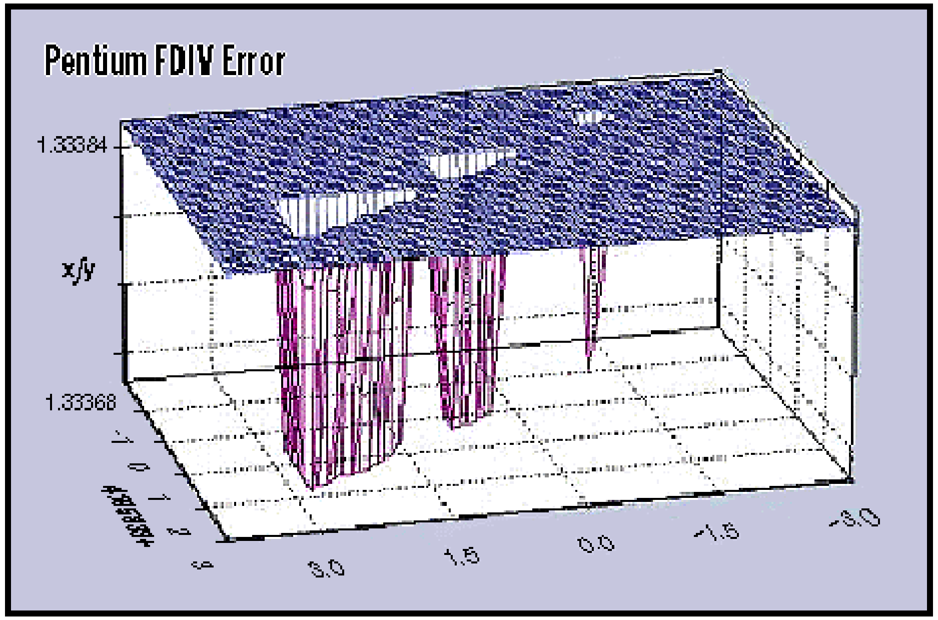
"Failure sets for the equivalent classes"

Small faults are hard to find using random tests

- These graphs are from very simple programs.
- In some cases there is a clear connection between faults in the code and patterns of failure in the software (argues for some structural testing).
- But some faults manifest as a few isolated points in the input space.
- Such faults are hard to find with random testing (because all points are equally probable – or at least there is some distribution derived from use of the software).
- Such faults often manifest at “boundaries” between different behaviours of the system.

Pentium FDIV bug

Pentium FDIV bug (1994): 10^{-9} probability of occurring (Intel), maximum error less than 10^{-5} (but probability of that $< 10^{-11}$).



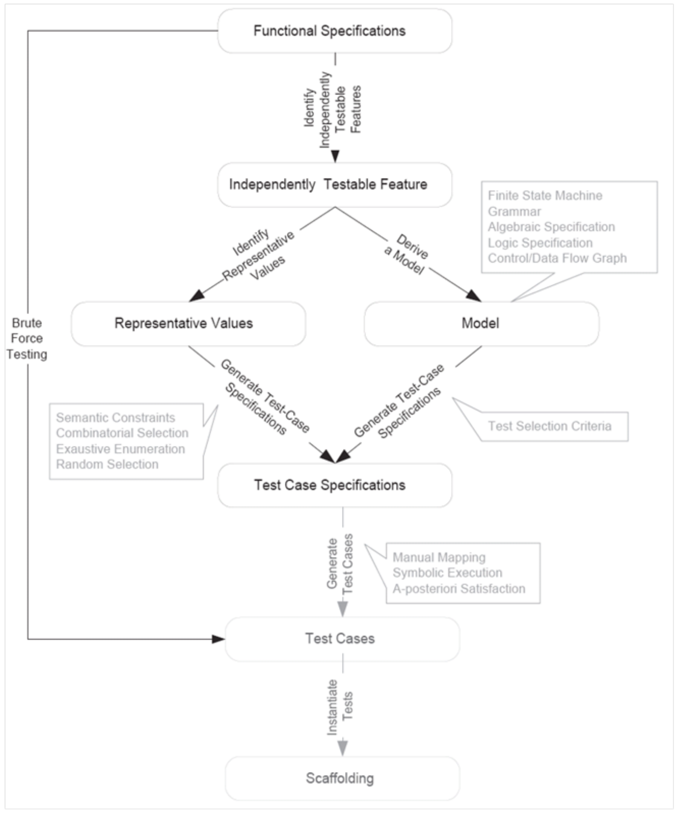
[image by Dusko Koncaliev]

Systematic vs Random Testing

- Spaces are very large e.g. a system with 2 32 bit integers as inputs has 264 possible test cases (i.e. approx 1020).
- Relative to this number of potential tests, the number of tests we can apply is always tiny.
- Random sampling means we can automate and apply a very large number of tests but even then the coverage will remain very small (particularly for complex problems).
- For example, in the case of buffer overrun failure, the likelihood of adding a very long sequence of elements is very small (why?)
- So faults with small profiles and the size of input spaces force a hybrid where we must consider some systematic testing – possibly reinforced with randomised testing.

A Systematic Approach

1. Analyse specification:
   – Identify ITFs.
2. Partition categories:
   – Significant cases for each parameter
3. Determine constraints:
   – Reduce size of test space.
4. Write and process test specification.
   – Produce test frames.
   – May need to return to categories and constraints.
5. Create test cases.
6. Execute test cases.
7. Evaluate results.



[Textbook, P&Y p. 169: Figure 10.3]

Functional Specifications

- This can be some kind of formal specification that claims to be comprehensive.
- Often it is much more informal comprising a short English description of the inputs and their relationship with the outputs.
- For some classes of system the specification is hard to provide (e.g. a GUI, since many of the important properties relate to hard to formalise issues like usability).

Independently Testable Features

- Here we slice the specification into features (that may spread across many code modules).
- Each feature should be independent of the other, i.e. we can concentrate on testing one at a time.
- The design of the code will make this easier or more difficult depending on how much attention has been given to testability in the systems design.

(sometimes muddled terminology: same as Independently Testable Function)

Modelling or Choice of Representative Values

- We consider model-based testing in a later lecture.
- For each of the inputs we consider classes of values that will all generate similar behaviour.
- This will result in a very large number of potential classes of input for non-trivial programs.
- We then identify constraints that disallow certain combinations of classes. The goal is to reduce the number of potential test cases by eliminating combinations that do not make sense.

Test Case Specifications and Test Cases

- From the partitions we can generate test case specifications.
- An important issue is identifying the expected output for a given input, this is also a good way of checking the specification defines a homogeneous group of test cases.
- These define a property that identifies a particular test case as belonging to that specification.
- There may be very many test case specifications (need for management). It may not be possible fully to automate the process of checking a test case matches a specification.
- It may not be possible fully to automate the process of running a test case on the system: *Is the answer correct? Is the environment set up properly?*

An Example

- **Command:** find
- **Syntax:** `find <pattern> <filename>`

  Function: The find command is used to locate one or more instances of the given pattern in the named file. All matching lines in the named file are written to standard output. A line containing a pattern is written out exactly once regardless of the number of times the pattern occurs in the line.

  The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (“”). To include a quotation mark in the pattern, two quotes in a row ““ must be used.

  Observation: origin is simplification of MS-DOS find command.

Aside: system(), exec(), java.lang.Runtime.exec()

- The C library call `system()` takes a string and invokes a shell with it (`/bin/sh c <string>`). This means that the shell will break the string up into a command and its arguments, and interpret it in a shell environment – quoting, redirection, pipes, etc.
- C `exec()` takes a list of words; the first is the $path$ to the executable, and the rest are the arguments (including the 0^{th} argument, which replaces the executables name) — No interpretation done; No redirection.
- Javas `exec()` on a single string uses Javas `StringTokenizer` to break the string up into words (purely on whitespace, so no quoting) — No redirection, even though it looks like `system()`.
- Javas `exec()` on a string array works like C `exec()`, but with $path$ and $arg0$ being the same thing — No interpretation done; No redirection.

Aside: system(), exec(), java.lang.Runtime.exec()

- So, `system(''echo 'hi there' foo > bar'')` in C would give the string to `sh`, which would almost certainly break the string up into {*"echo"*, *"hi there"*, *"foo"*, *">"*, *"bar"*}, and then run the command *"echo"*, *"hi there"'*, *"foo"* with its output sent to the file *"bar"*.

- While `exec(''echo 'hi there' foo > bar'')` in Java would break the string up into {*"echo"*, *"'hi"*, *"there"'*, *"foo"*, *">"*, *"bar"*} (note the single quotes are still there), and simply run the command {"echo", "'hi", "there"', "foo", ">", "bar"}, with no output redirection.

- Yet another example of how environment can matter, and how it is a bad idea to conflate the execution environment with the program being executed.

Identifying Independently Testable Features

- The specification is small so we might just identify a single ITF, that the collection of lines resulting from a “find” all contain the pattern and those lines that are not in that collection do not contain the pattern.
- We might consider the identification of various error conditions as ITFs e.g. problems with the file or filename, and malformed patterns. Each of these is to, to some extent, an ITF because the functionality should be independent.
- So our ITFs could be:
  1. Correctly identifies malformed patterns on the input.
  2. Correctly identifies malformed filenames and other file related issues.
  3. Correctly identifies exactly the lines in the file that match the pattern. This might decompose (in some more detailed analysis) into (a) identifies pattern within a line (b) identifies all lines.

Identifying Categories (for the matching ITF)

- **Parameter:** pattern
  - Pattern size
  - Quoting
  - Embedded blanks
  - Embedded quotes
- **Parameter:** filename
  - Assume it is valid for this ITF
- **Environment:** file corresponding to filename
  - Number of occurrence of pattern in file
  - Number of occurrence on a target line
- For the malformed filename ITF we might have fewer categories e.g. File name in the parameters section and name/file correspondence in the Environments section.

Partitioning categories (for matching)

- **Parameter:** pattern
  - Pattern size: empty, single, multiple, too long
  - Quoting: pattern quoted, pattern not quoted
  - Embedded blanks: none, one, several
  - Embedded quotes: none, one, several
- **Parameter:** filename
  - Valid filename
- **Environment:** file corresponding to filename
  - Number of occurrence of pattern in file: none, one, several
  - Number of occurrence on a target line: (none,) one, several
- With no restrictions this is: $4 * 2 * 3 * 3 * 3 * 2 = 432$ different test frames.
- We use constraints to help reduce this number

Identify Constraints

- **Parameters:**
  - Pattern size:
    - empty, [property Empty]
    - single, [property nonEmpty]
    - multiple, [property nonEmpty]
    - too long, [property nonEmpty]
  - Quoting:
    - pattern quoted, [property Quoted]
    - pattern not quoted, [if nonEmpty]
  - Embedded blanks:
    - none, [if nonEmpty]
    - one, [if nonEmpty and Quoted]
    - several [if nonEmpty and Quoted]
  - Embedded quotes: ...
- Can lead to a significant reduction in the number of categories

Generate Test Case Specifications

- The specification describes a collection of test partitions
- The number is significantly smaller than the simple product would suggest.
- E.g. we only need to consider embedded blanks where we have quoted strings.
- E.g. a quoted multiple character pattern with one embedded blank, several embedded quotes applied to a file with several occurrences of the pattern at most once per line.

Generating and Running Tests

- We will consider this in more detail in the next lecture.
- There are still issues in checking a test matches some specification.
- The test plus the system specification should determine valid outputs (if the test is intended to create valid output) but this can be an issue.
- Much research has gone into creating automated oracles that check the output of a test for validity.

  Difficult, e.g. I worked on a ~ 3 *person-year project of about* $140,000$ *loc which produced over* $5,000$ *lines of output from nightly tests. The tests succeeded or failed overall in a binary fashion, but there is a lot of grey area, such as checking compiler flags and warnings across platforms, etc.*

Example: "cat"

Here we consider testing the UNIX "cat" command using the category-partition method we have been looking at.

Functional Specification

```
NAME  cat - concatenate files and print on the standard output
SYNOPSIS  cat [OPTION] [FILE]...
DESCRIPTION  Concatenate FILE(s), or standard input, to standard output.
       -A, --show-all equivalent to -vET
       -b, --number-nonblank number nonblank output lines
       -e     equivalent to -vE
       -E, --show-ends display $ at end of each line
       -n, --number number all output lines
       -s, --squeeze-blank never more than one single blank line
       -t     equivalent to -vT
       -T, --show-tabs display TAB characters as ^I
       -u     (ignored)
       -v, --show-nonprinting use ^ and M- notation, except for LFD and TAB
       --help display this help and exit
       --version output version information and exit
       With no FILE, or when FILE is -, read standard input.
EXAMPLES
       cat f - g Output f's contents, then standard input, then g's contents.
       cat    Copy standard input to standard output.
```

Identifying Independently Testable Features

- Here we might think that there are three ITFs associated with cat:
  1. Error checking for the option string – checking the syntax is OK and for consistent combinations of options.
  2. Error checking the syntax of filenames.
  3. That cat functions correctly given a legal option string and a syntactically correct sequence of filenames.
- Here we will consider generating categories and partitions of those categories for the third ITF we have identified.
- For each ITF we begin by identifying:
  - Parameters relevant to the feature.
  - Other elements of the execution environment that the ITF is dependent on. Typical elements are: databases, the file system, hardware devices, ...

Identifying parameters relevant to the ITF

- Parameters:
  - Option string:
  - Filename sequence:
- Environment:
  - The file system (this is a mapping from valid filenames to file contents)
- The parameters relevant to an ITF are those that cause a change in behaviour in the Feature when they are changed.
- The next stage is to identify categories – these are elementary characteristics of the parameters which are either found explicitly in the specification or they are implicit – i.e. they arise from the experience of the tester.

Identifying Categories

- Parameters:
  - Option string:
    - Option string length
  - Filename sequence:
- Environment:
  - The file system (this is a mapping from valid filenames to file contents):
  - Standard input:

Summary

- We have seen how a systematic approach to testing can be based on the specification.
- We have also seen how this can make the specifications inadequacies apparent.
- We have looked at the stages of that approach:
  1. Identifying testable features
  2. Identify categories
  3. Identifying constraints
  4. Deriving test case specifications
  5. Deriving test cases that match the specifications
  6. Executing the cases.
- This approach is important to most test situations but it becomes more difficult to apply the richer the environment becomes.

Readings

Required Readings

- **Textbook (Pezzè and Young):** Chapter 10, Functional Testing
- T. J. Ostrand and M. J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. Commun. ACM 31, 6 (June 1988), 676-686. http://dx.doi.org/10.1145/62959.62964