Verification & Validation 2

Lecture Plan:

- Structural Testing
- Integration Testing
- Testing and the Software Engineer
- Version and Configuration Control

Unit Testing Strategies

Specification Testing:

Test cases are derived from the specification of the component.

The component is a *black box*: its implementation *isn't* examined.

Structural Testing:

Test cases are derived from the implementation of the component.

The component is a *white box*: its implementation *is* examined.

Structural Testing

Test cases are derived from the implementation of the program.

The component is a *white box*, and its implementation *is* examined.

Test selection strategies:

- In *path coverage*, we must ensure every possible path is executed at least once during testing
- In *branch coverage*, we must ensure that every branch is executed at least once during testing

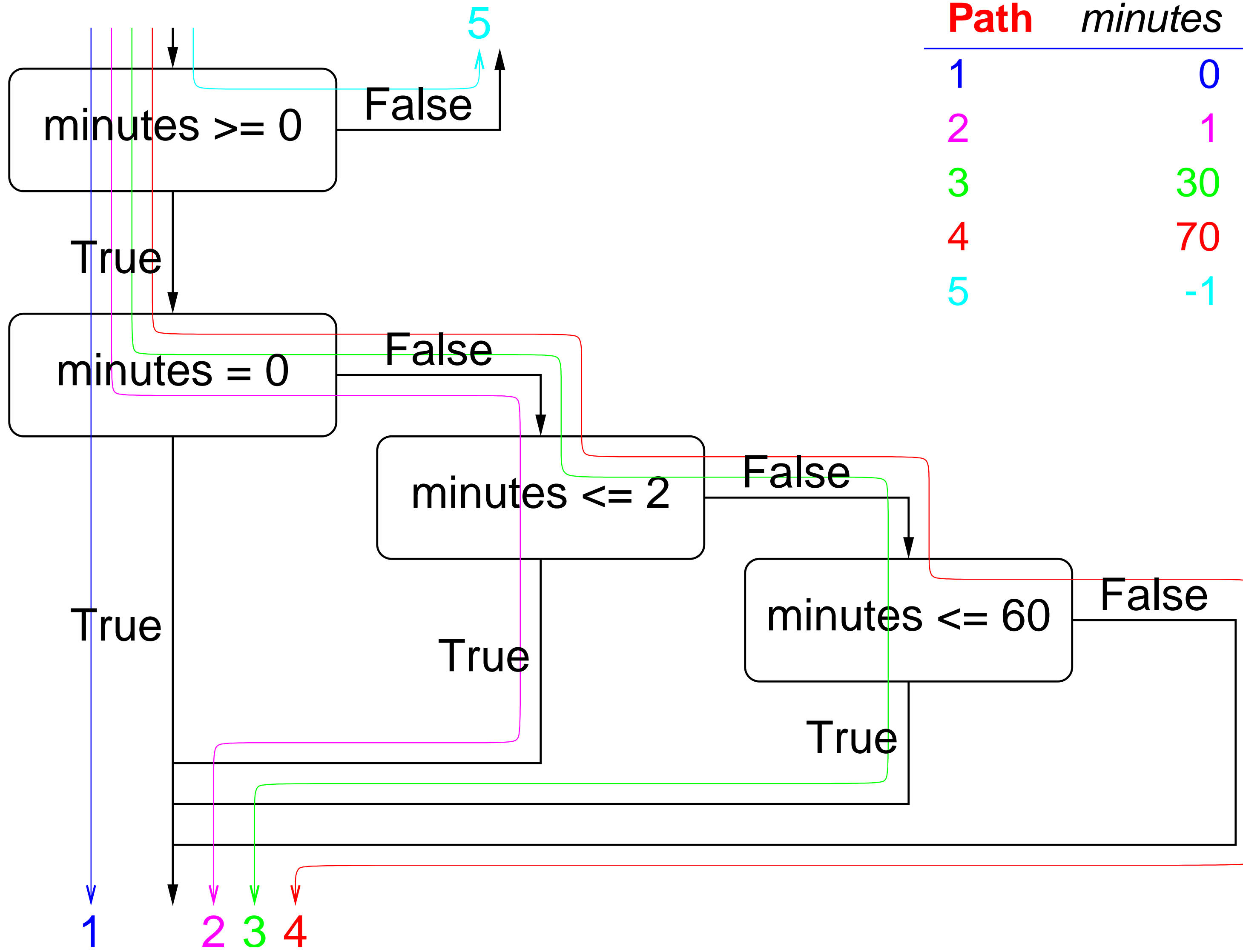
```
class interface CALL
    -- Telephone call billing information.
feature(s) from CALL
    minutes: INTEGER
        -- Duration of the call in minutes or part thereof.
    cost: INTEGER
        -- The cost of the call in cents.
        require
            minutes >= 0
        ensure
            minutes = 0 implies Result = 0;
            0 < minutes and minutes <= 2 implies Result = 20;
            2 < minutes and minutes <= 60 implies
                Result = 20 + (minutes - 2) * 30;
            60 < minutes implies Result = 2000
end of CALL
```

Call *cost* Implementation

```
cost : INTEGER is
        -- The cost of the call in cents.
    require
        minutes >= 0
    do
        if minutes = 0 then
            Result := 0
        elseif minutes <= 2 then
            Result := 20
        elseif minutes <= 60 then
            Result := 20 + (minutes - 2)*30
        else
            Result := 2000
        end
    ensure

    end
```

Structural Testing of the *cost* Query

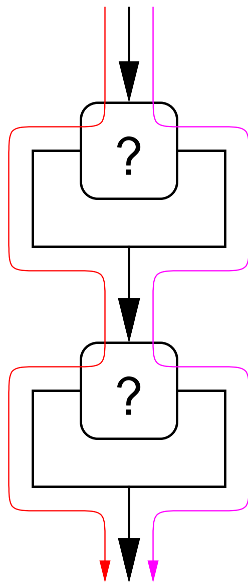


Path	*minutes*	Expected Result
1	0	0
2	1	20
3	30	860
4	70	2000
5	-1	error

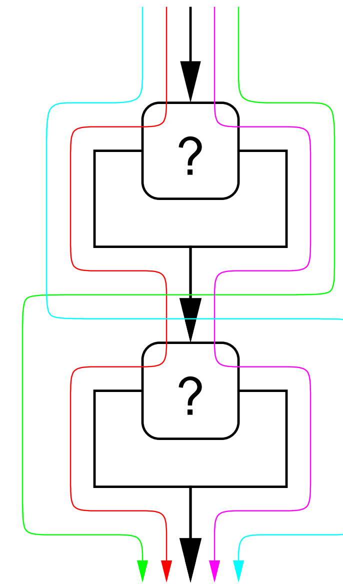
Branch Coverage vs Path Coverage

Example: 2 choice points

Branch Coverage



Path Coverage

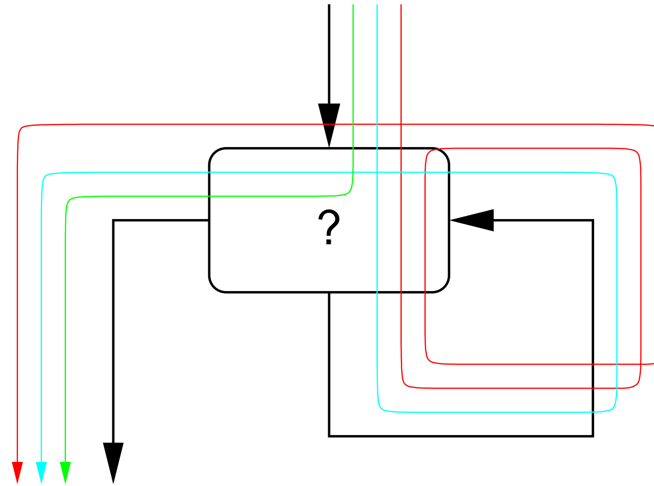


Path coverage is a more thorough test.

Path testing code with n branches requires up to 2^n tests.

This may not be practical for complex code.

Branch Coverage for Loops



Branch coverage requires:

- at least one test in which the loop is not entered
- at least one test in which the loop is entered

Integration Testing

- progresses incrementally to localise faults
- at each increment, add only those classes need for the functionality being tested
- finds class interaction faults

Top-Down Testing

- top-level classes integrated and tested first
- lower-level classes represented by *stubs* (same interface as the true class, but of limited functionality)

✔ design faults found early

✘ basic classes not thoroughly tested until later

Bottom-Up Testing

- bottom-level classes integrated and tested first
- upper-level classes replaced by *harnesses* (programs to exercise the class under test with test data)

✔ basic classes are thoroughly tested

✘ design faults are not uncovered until later

Combining Top-Down and Bottom-Up Testing

What to test top-down:

- classes with application-specific logic
- classes which occur near the top of the dependency hierarchy

What to test bottom-up:

- reusable classes with generic functionality (basic classes)
- classes near the bottom of the dependency hierarchy

In practice, both Top-Down and Bottom-Up Testing are used

The combination is sometimes called Sandwich Testing

Testing and the Software Engineer

Main Entry: professional

1 c (2) : exhibiting a courteous, conscientious, and generally businesslike manner in the workplace

Merriam-Webster's Dictionary

- Software Engineers have test plans
- Software Engineers record the results of their testing
- Software Engineers record the changes made to classes during testing

"Maybe the reason that things aren't going to plan
is that there never was a plan."

Version and Configuration Control

Testing (and maintenance) may produce many *versions* of a module

- A *revision* is a new version of a module that is intended to replace the old one.
  e.g., when you fix a fault in the module.

- A *variation* is a new version of a module intended to co-exist with the old one.
  e.g., when you port a system to a new operating system.

You need to keep all the versions of all the modules because:

- A new revision may introduce a worse bug than it fixes.

- Different customers may have different variations of the system.

- The client may decide they liked the old revision better.

Keeping track of all these versions can be aided by a tool.

What Can a Version/Configuration Tool Do?

- Give easy storage and access to all the versions of a module
- Remember which versions of which modules go together
- Store descriptions of each change made to a module
- Be able to identify the version of any module, either source or object code
- Cooperate with a build tool to build particular configurations of a system
- Ensure two programmers don't modify the same version at the same time

Suggested Reading

From the COMP1110 Reading Brick:

- **RCS–A System for Version Control**

  Tichy, Walter F (1985) ”RCS–A System for Version Control”

  Software–Practice and Experience, Vol.15(7) (July 1985) pp 637–654