Software Testing

Philipp Reinecke

ReineckeP@cardiff.ac.uk

What is testing?

We have been very busy building our product…

Have we done a good job?

Does it do what the customer wants?

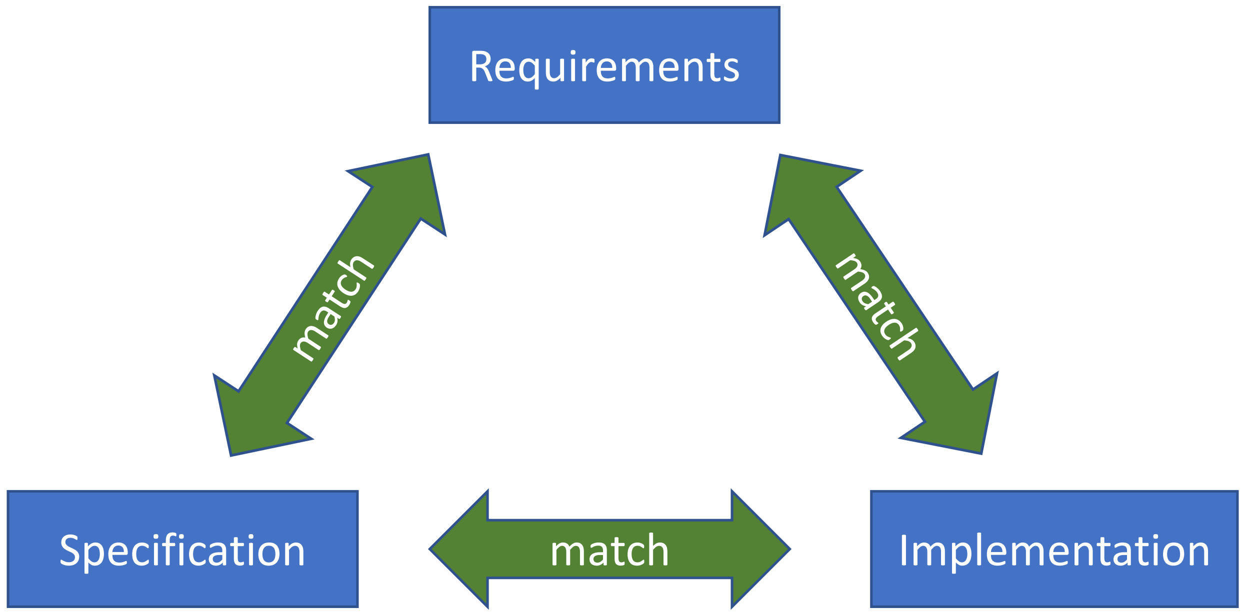
- Functionality: Does it work?
- Performance: Is it fast enough?
- Load: Does it scale?
- Reliability: Does it break?
- Security: Is it secure?

Caution: Testing can only show presence of errors, not their absence!

Two ways to find out:

- Prove that it does – difficult
- Show that it does not NOT – try to break it by testing

What does the customer want?



What have we done?

Testing Objectives [Myers 79]

1. Testing is a process of executing a program with the intent of finding an error
2. A good test case is one that has a high probability of finding an as-yet undiscovered error
3. A successful test is one that uncovers an as-yet undiscovered error

Testing: When, Who and How?

- Testing begins when something which can be tested (i.e. something executable) is produced
- But: Test case **design** starts in the requirements engineering phases and will be updated in design and implementation phases.
- A separate test group may be involved in testing
  - Avoids biases:
    - Developers might not want to break their system
    - Dedicated testers have different perspectives
  - Important for complex business-critical or safety-critical systems
- Different kinds of tests are needed
  - Black-box, white-box, component testing, integration testing, system testing, stress testing …

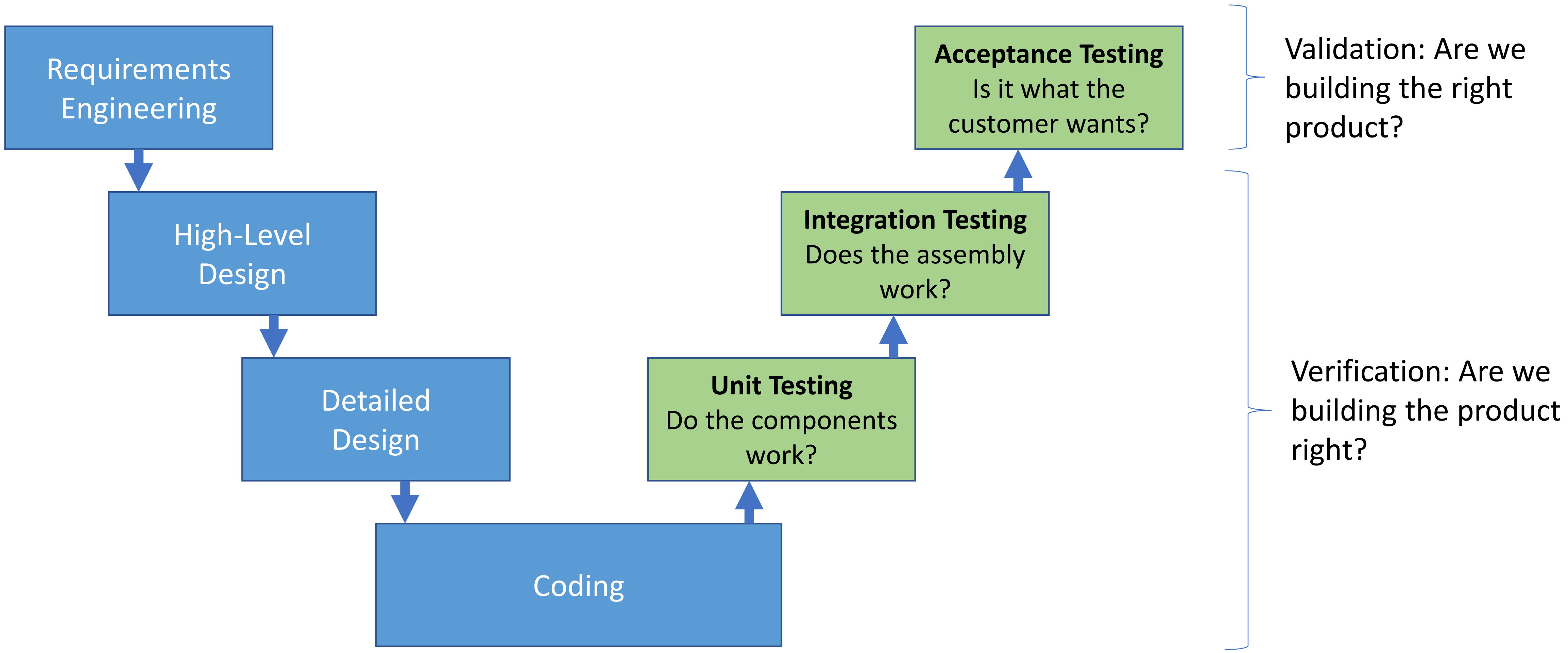
Challenges of Testing

- Testing is a psychologically difficult activity
  - It is seen as destructive
  - Testers should try their best to break the system
- Mistaken attitudes can also affect testing such as:
  - A successful test is one which does not reveal any errors
  - Testing cannot start until coding has finished
  - Testing can be reduced to meet deadlines
  - As long as the software runs it does not need testing
- Testing is expensive
- Testing happens at crunch time
- Exhaustive testing is not possible

Davis' [1] Testing Principles

- All tests should be traceable to customer requirements
- Tests should be planned long before testing begins
- The Pareto principle applies to software testing: 80% of errors will be traceable to 20% of all components
- Testing should begin "in the small" and progress toward testing "in the large"
- Exhaustive testing is not possible
- To be most effective, testing should be conducted by an independent third party

Testing Strategy: V Model



Generating Test Cases

- Goal: Design tests that
  - are most likely to find errors
  - use minimum of resources
- Test coverage:
  - Proportion of potential paths through program that are covered by test set
  - Maximise coverage while minimising number of tests
- Approaches:
  - White-box testing
    - Use knowledge of internal structure to design efficient tests
    - Done during development
  - Black-box testing
    - Assume software is a black box – only interact with the interface
    - Done later in development process (e.g. acceptance testing)
- Designing for testability can help

White-box testing

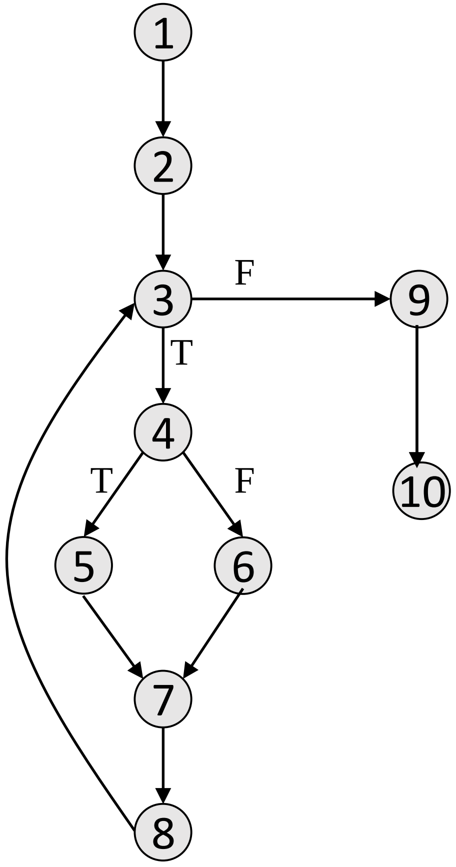
- Internal knowledge helps design test cases
  - Execute all control flows
  - Exercise all logical decisions for both true and false
  - Execute all loops at boundaries and within their bounds
  - Exercise all internal data structures

```
     RecordCount
1       initialise count
2       read first record
3       WHILE NOT eof
4         IF record okay THEN
5            increment count
          ELSE
6            report error
7         ENDIF
8         read next record
(3)     ENDDO
9       report count
10   END
```

Control Flows

- Show the flow of control in a program or module
- Nodes: Points in computation
- Edges: Flow of computation between points

```
   RecordCount
1      initialise count
2      read first record
3      WHILE NOT eof
4        IF record okay THEN
5            increment count
         ELSE
6            report error
7        ENDIF
8        read next record
(3)    ENDDO
9      report count
10 END
```



Basis Path Testing [McCabe 76]

- Definitions:
  - A basis set of paths through a program executes each instruction in that program at least once
  - An independent path in a basis set is one which differs from other paths in the set in at least one way
  - Cyclomatic complexity: Number of independent graphs through the control-flow graph – upper limit for number of test cases
- Method:
  - Determine cyclomatic complexity v(G)
  - Determine basis set
  - Generate v(G) test cases for executing each path in basis set

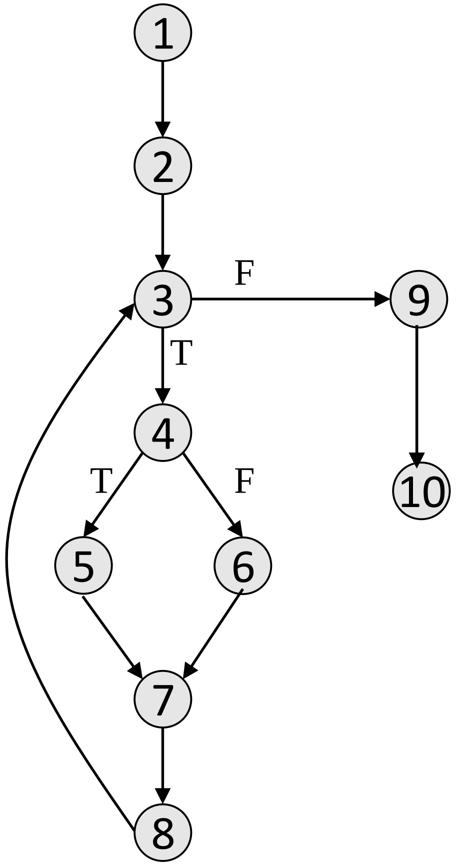
Basis Path Testing: Basis sets

Example paths:

- 1, 2, 3, 9, 10
- 1, 2, 3, 4, 5, 7, 8, 3, 9, 10
- 1, 2, 3, 4, 6, 7, 8, 3, 9, 10

- A basis set of paths through a program executes each instruction in that program at least once
- An independent path in a basis set is one which differs from other paths in the set in at least one way

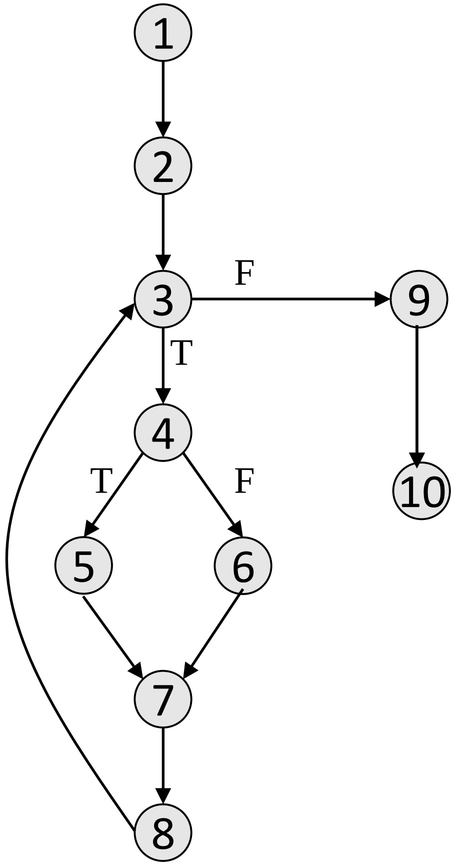
```
     RecordCount
1       initialise count
2       read first record
3       WHILE NOT eof
4         IF record okay THEN
5            increment count
          ELSE
6            report error
7         ENDIF
8         read next record
(3)     ENDDO
9       report count
10   END
```



Cyclomatic Complexity

- The number of independent paths through a graph is called the cyclomatic complexity of the graph
- $v(G) = e(G) - n(G) + 2$
  - $e(G)$ = number of edges in G
  - $n(G)$ = number of nodes in G
- Gives an upper limit for number of test cases that need to be run to exercise every part of the program at least once

Cyclomatic Complexity Example



$e(G) = 11$

$n(G) = 10$

$$v(G) = 11 - 10 + 2 = 3$$

Example Test Case – Path 1

Test Case ID: RecordCount01			
Test Purpose: Test if there are no records (Test path 1,2,3,9,10)			
Environment: Written in Fiji v1.4 running under WendOS v2.3			
Pre-conditions:			
Test Case Steps: Ensure record file has no records			
Step No.	**Procedure**	**Expected Results**	**P/F**
1	Call Record Count Routine	Count of 0 is reported	P
Comments:			
Author: WKI	Date: 05/03	Checker: WKI	Date: 18/04

Example Test Case – Path 2

Test Case ID: RecordCount02			
Test Purpose: Test when all records are OK (Test path 1,2,3,4,5,7,8,3,9,10)			
Environment: Written in Fiji v1.4 running under WendOS v2.3			
Pre-conditions: Ensure record file contains appropriate records			
Test Case Steps:			
Step No.	Procedure	Expected Results	P/F
1	Call Record Count Routine	Display count as no. of records in file	F
Comments:			
Count was one less than number of records in the file			
Author: WKI	Date: 05/03	Checker: WKI	Date: 18/04

Example Test Case – Path 3

Test Case ID: RecordCount03			
Test Purpose: Test path 1,2,3,4,6,7,8,3,9,10			
Environment: Written in Fiji v1.4 running under WendOS v2.3			
Pre-conditions: Ensure record file has a faulty record			
Test Case Steps:			
Step No.	**Procedure**	**Expected Results**	**P/F**
1	**Call Record Count Routine**	**Error is reported**	**P**
		Display count as (no. of records in file – 1)	**P**
Comments:			
Author: WKI	**Date: 05/03**	**Checker: WKI**	**Date: 19/04**

Black Box Testing (1)

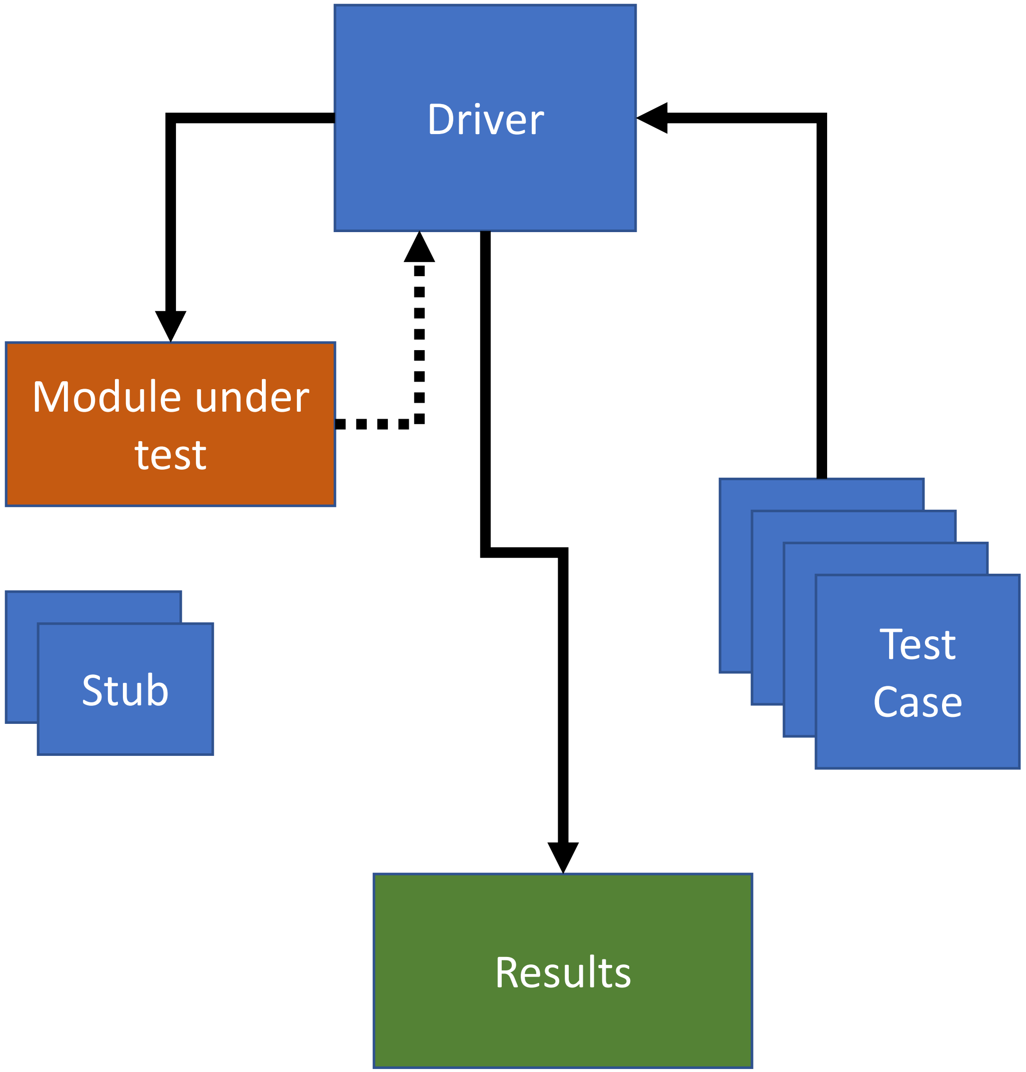
- Aim is to derive a set of tests that fully exercises all functional requirements for a program
  - Try to minimise the number of test cases needed
- Focuses on specification rather than design
- Typically undertaken during later phases of system development
  - Integration, Validation and Acceptance Tests
- Black box testing complements white box testing, rather than replacing it

Black Box Testing (2)

- Good black box testing hopes to find problems with the program before it is released to clients
- Aims to find the following kinds of error:
  - incorrect or missing functionality
  - interface errors
  - errors in data access (internal or external)
  - initialisation and termination errors
- Uses knowledge of frequent mistakes in coding to help design effective test cases.

Unit Testing

- Test each individual module independently
  - Ideally when it is written
- Test:
  1. Interface: Does data flow in and out correctly?
  2. Local data structures: Does the unit store local data?
  3. Boundary conditions: Does it operate correctly at boundaries?
  4. Independent paths: Execute all paths
  5. Error handling paths: Does it handle errors?
- Frameworks:
  - For Java: jUnit
  - For Python: unittest

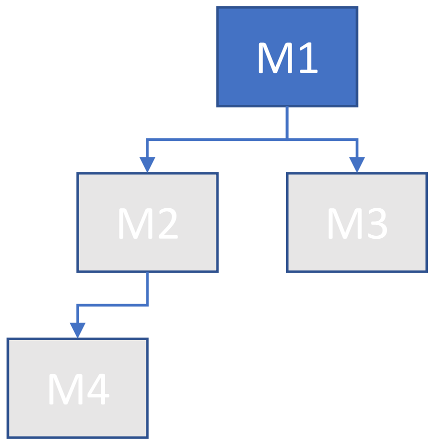


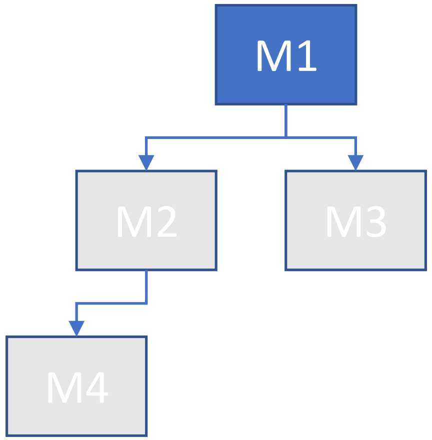
Integration Testing

- Goal: Ensure individual modules work together
- Potential issues:
  - Communication between modules
  - Timing between modules
  - Side-effects
  - Wrong assumptions
- Approaches:
  - Put them all together and see what happens
  - Incremental integration:
    - Allows errors to be located and eliminated before proceeding

Top-Down Integration Testing

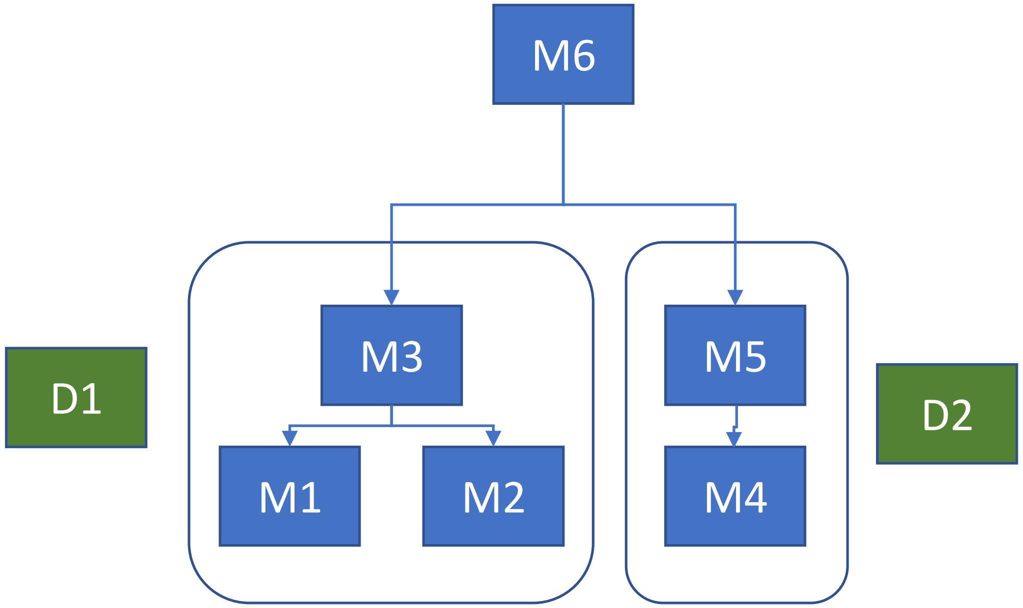
- Build up program structure starting from main control module
- Use stubs for subordinate modules, replace by actual modules
- Steps:
  1. Test main control module with stubs for all modules called by it
  2. Replace subordinate stubs one by one with actual modules
  3. Run tests after each new module has been added
  4. Run regression tests to ensure no new errors have been introduced
  5. Repeat from 2 until entire program is built
- Depth-first or breadth-first strategies





Bottom-up Integration Testing

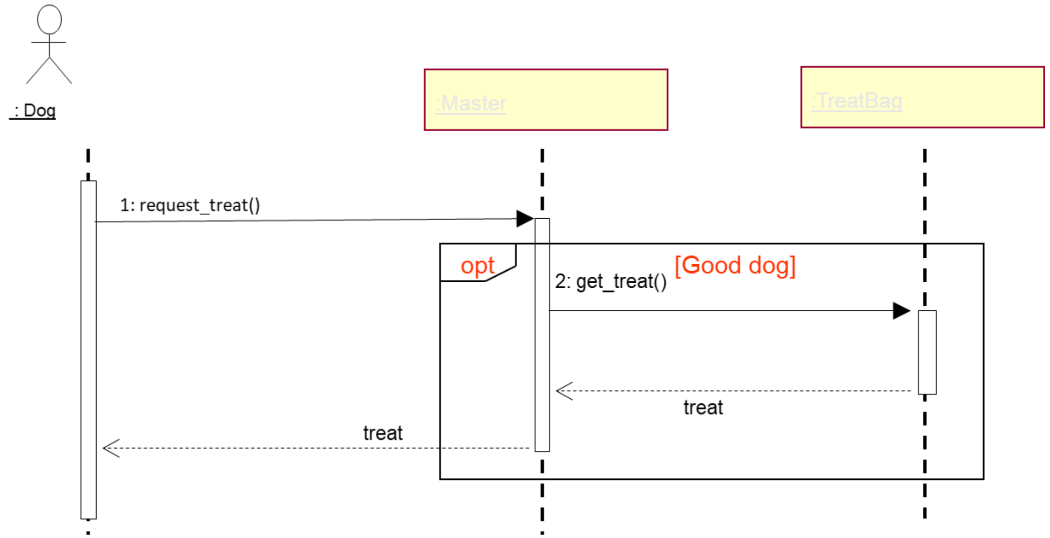
- Build up program structure from components
- No stubs needed
- Steps:
    1. Combine low-level components into builds (or clusters) to perform specific subfunction
    2. Write driver to run tests
    3. Test build
    4. Remove drivers and combine builds, moving upwards
    5. Repeat until whole program is built



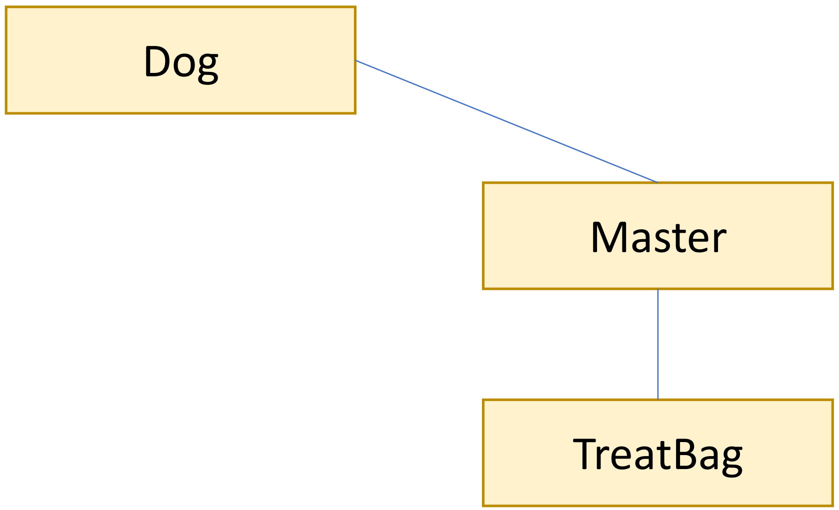
Regression Testing and Smoke Testing

- Regression Testing
  - Goal: Ensure that no new errors have been introduced
  - Method: Repeat (sub-set) of tests after every change
  - Include tests for errors that are likely after change
- Smoke Testing
  - Build and test whole program daily throughout development
  - Advantages:
    - Avoids surprises in integration
    - Gives continuous feedback on project progress
  - Continuous Integration

Object-Oriented Testing



- Unit Testing
  - Smallest unit is the class
  - Test operations within class context, not on their own
  - Example: Inherited methods in class structure must be tested in each subclass
- Integration Testing
  - Top-down or bottom-up integration has no meaning without hierarchical control
  - Strategies for OO
    - Thread-based: Integrate set of classes needed to respond to one input, test sets individually
    - Use-based: Start by testing classes that use very few other classes, then test dependent classes
    - Cluster-based: Determine clusters of collaborating classes from CRC diagram



Testing non-functional attributes

- Non-functional attributes: How good is the software?
  - Performance, Security, Dependability, Energy-efficiency, …
- Benchmarking: Run system with specified parameters to generate comparable data
- Performance testing:
  - Generate system load and measure response times
- Security testing:
  - Penetration testing
  - Fuzzing: Generate random inputs
- Dependability testing:
  - Inject faults and observe failures
- Energy efficiency:
  - Generate load and measure energy usage

Design for Testability [Bach 94]

- Operability: Avoid bugs
  - Less overhead for analysis and reporting
  - No blocking of tests
- Observability: Make the system transparent
  - Report errors
  - Produce clear and distinct output
- Controllability: Make the system controllable
  - All code is executable via some inputs
  - Enables automation
- Decomposability:
  - Enables independent unit tests
- Simplicity: Structural and functional
  - The less there is to test, the easier it is
- Stability: Avoid unnecessary changes to software
  - Reduces disruptions to testing
- Understandability:
  - The more we understand, the smarter we can design our tests

Conclusion

- Testing is important
  - Helps find errors before the customer finds them
- Testing is destructive
  - Good tests find problems
- Testing should be done systematically and early
- Tools can help
  - Unit testing: jUnit, unittest, Eclipse
  - Integration testing/continuous integration: Jenkins
  - Load testing: JMeter