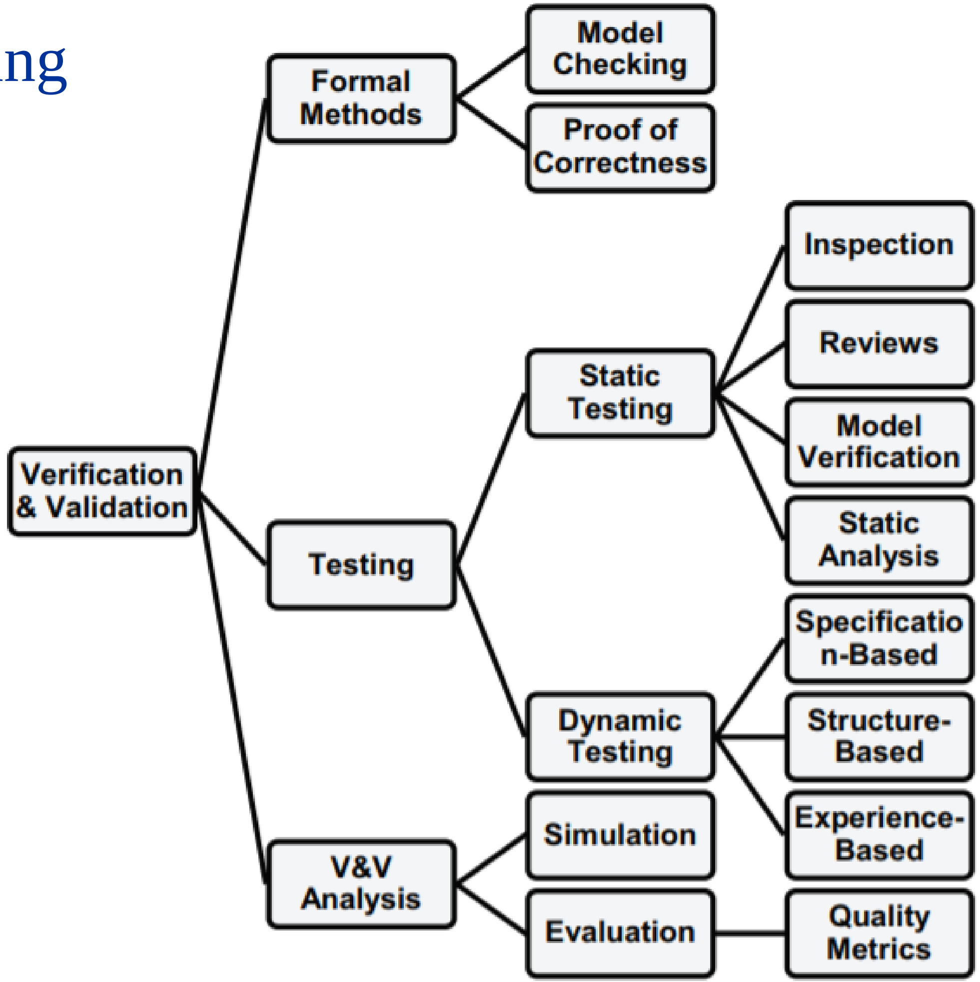
Lecture 18: Testing

Testing in V&V

- Both verification and validation involves testing
- Verification
  - Whether the code conform with software specification
  - Conformance testing
- Validation
  - Functional testing
  - Scenario testing
  - Risk based testing



Who and when?

- Testing is performed by *test engineers* after the initial development
  - Test suite should be constructed during development

- Test designer may be different from test engineers

- The test engineers should be able to do it without participating the development process

Terminologies

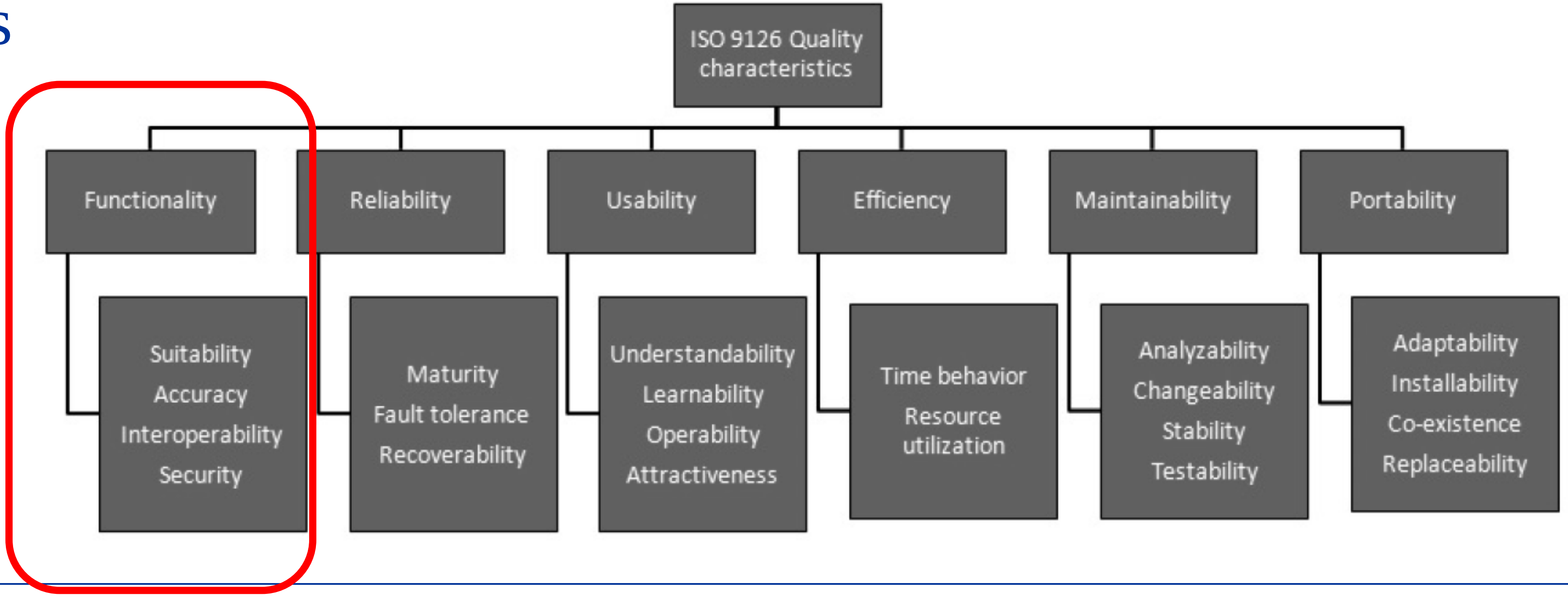
- International Software Testing Qualifications Board (ISTQB)
  - error: human action at the root of a defect;
  - defect: result of a human action (i.e. error), which is present in the test object
  - failure: result from the execution of a defect by a process (whether the process is automated or not).

Objectives of testing

- Software testing focuses on two complementary but distinct aspects:
  - Defect and failure detection
    - Fix the bugs
    - Improve the quality of the product delivered to customers and users
  - Establishing confidence to the software
    - Level of risks associated with the delivery of the software to the market
    - Capabilities of the developer team

Software characteristics

- Functional characteristics
- Non-functional characteristics
- Non-functional tests should be performed together with functional tests



Testing vs. Risk management

- There is a tradeoff on how much should be tested
  - Principle similar to risk evaluation process
- Testing can be used to confirm the existence of risk

Integrity levels (IEEE 829-2008)

- level 4: catastrophic:
  - software must execute correctly or grave consequences (loss of life, loss of system, environmental damage, economic or social loss) will occur. No mitigation is possible;
- level 3: critical:
  - software must execute correctly or the intended use (mission) of system/software will not be realized causing serious consequences (permanent injury, major system degradation, environmental damage, economic or social impact). Partial-to-complete mitigation is possible;
- level 2: marginal:
  - software must execute correctly or the intended function will not be realized causing minor consequences. Complete mitigation possible;
- level 1: negligible:
  - software must execute correctly or the intended function will not be realized causing negligible consequences. Mitigation not required.

Testing vs. debugging

- Testing
  - Identify defects in software
- Debugging
  - Fix defects

Early Testing

- The cost of finding (and fixing) a defect
  - In the design phase is “1”.
  - In the coding phase, multiplied by “10”
  - During the test phase (system test or acceptance test), multiplied by “100”.
  - In production (by the customer or by a user), multiplied by “1,000”.

Defect clustering

- If you find a defect, it might be efficient to look for other defects in the same piece of code.
  - A complex section of an algorithm
  - Components designed by the same person
  - A group of components designed in the same period of time

Testing mentality

- A developer will try to find *one* solution to a particular problem, and will design the program based on the solution envisaged;
- A tester will try to identify *all* possible failures that might have been forgotten.
- A developer usually has two hats: the developer's and the tester's
- Our brain has a tendency to hide defects because it is looking for usual patterns.
- Need to have different perspectives

Independence level

- Same person
- Pair programming
- Independent tester from the same team
- From the same company
- Third party

Testing is not a destructive activity

- The defects were not introduced by testers, but developers
- It is not personal, it is just business
  - Point out defects should not look bad for developers
  - need good relational skills

Types of tests

- Component tests
- Component integration tests
- System tests, on the completely integrated system
- Acceptance tests, a prerequisite for delivery to the market or production.

Attributes of testing

- *Test object:* the target of the tests
  - a function, a sub-program, or a program, a software application, or a system made up of different sub-systems;
- *Specific objectives:* the reasons why the tests will be executed
  - To discover certain types of defects, to ensure correct operation, or provide any other type of information (such as coverage);
- *Entry and exit criteria:* when a task can start (the pre-requisites) and when it can be considered as finished.

Component level test/Unit test/Class test

- *Test object*:
  - components, program modules, functions, programs,
- *Objective*:
  - detect failures in the components,
- *Entry criteria*:
  - the component is available, compiled, and executable in the test environment;
  - the specifications are available and stable.
- *Exit criteria*:
  - the required coverage level has been reached;
  - defects found have been corrected;
  - the corrections have been verified;
  - regression tests have been executed on the last version and do not identify any regression;
  - traceability from requirements to test execution is maintained

Integration level testing

- *Test object*:
  - components, infrastructure, interfaces, database systems, and file systems.
- *Objective*:
  - detect failures in the interfaces and exchanges between components.
- *Entry criteria*:
  - at least two components that must exchange data are available, and have passed component test successfully.
- *Exit criteria*:
  - all components have been integrated and all message types (sent or received) have been exchanged without any defect for each existing interface;
  - statistics (such as *defect density*) are available;
  - the *defects* requiring correction have been corrected and checked as correctly fixed;
  - the impact of not-to-fix defects have been evaluated as not important.

Types of integration

- Integration of software components
  - typically executed during component integration tests

- Integration of software components with hardware components

- Integration of sub-systems
  - typically executed after the system tests for each of these sub-systems and before the systems tests of the higher-level system.

Integration approaches

- Big bang integration
  - Con: difficult to isolate defects found in the next level
- Bottom-up integration
  - Con: need drivers to execute
- Top-down integration
  - Con: need mock objects (stubs) to execute
- Sandwich integration
  - Combinations of bottom up and top down approach
- Integration by functionalities
  - Intuitive
- Neighborhood integration

System Test

- *Test object*:
  - the complete software or system, its documentation, the software configuration and all the components that are linked to it
- *Objective*:
  - detect failures in the software, to ensure that it corresponds to the requirements and specifications, and that it can be accepted by the users.
- *Entry criteria*:
  - all components have been correctly integrated, all components are available.
- *Exit criteria*:
  - the level of coverage has been reached;
  - must-fix defects have been corrected and their fixes have been verified;
  - regression tests have been executed on the last version of the software and do not show any regression;
  - bi-directional traceability from requirements to test execution is maintained;
  - statistical data are available, the number of non-important defects is not large;
  - the summary test report has been written and approved.

Acceptance Test

- *Test object*:
  - the complete software or system,
- *Objective*:
  - obtain customer or user acceptance of the software.
- *Entry criteria*:
  - all components have been correctly tested at system test level and are available,
  - the last fixes have been implemented and tested without identification of failures,
  - the software is considered sufficiently mature for delivery.
- *Exit criteria*:
  - the expected coverage level has been reached;
  - must-fix defects have been corrected and the fixes have been verified;
  - regression tests have been executed on the latest version of the software and do not show regression;
  - traceability from requirements to test execution has been maintained;
  - user and customer representatives who participated in the acceptance test accept that the software or system can be delivered in production.

Acceptance Test

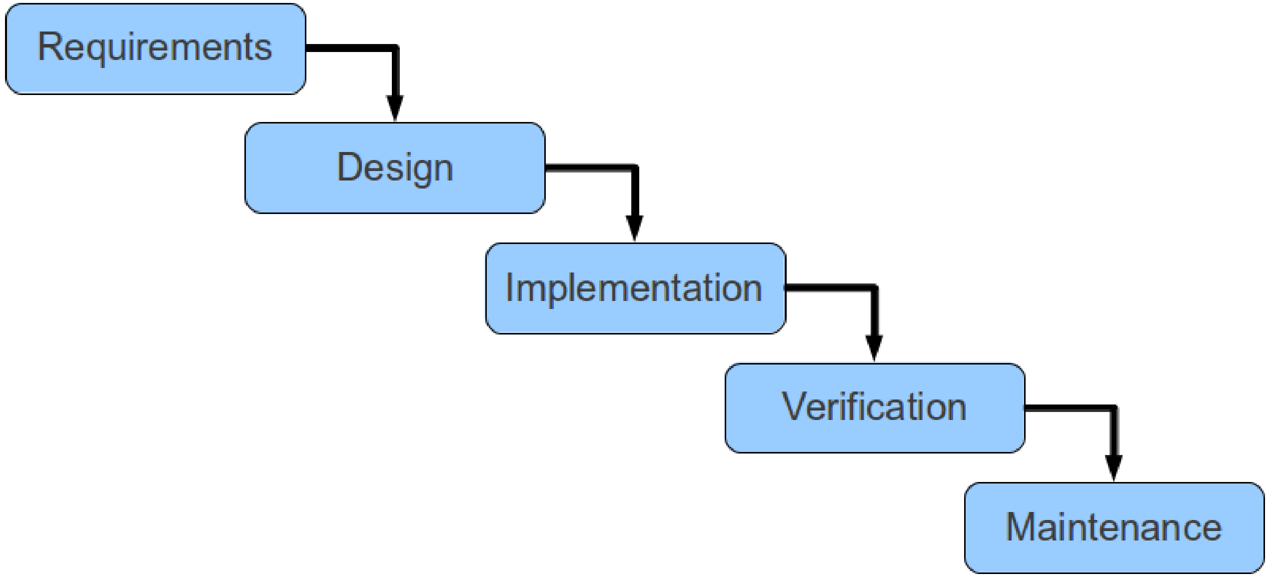
- user acceptance
- operational acceptance
- contractual acceptance
- regulatory acceptance
- alpha tests
- beta tests
- pilot phase

Model-independent testing principles

- Each design activity, and each deliverable, must have a corresponding test activity that will search for defects introduced by this activity or in this deliverable;
- Each test level has its own specific objectives, associated with that test level, so as to avoid testing the same characteristic twice;
- Analysis and design of tests for a given level start at the same time as the design activity for that level, thus saving as much time as possible;
- Testers are involved in document review as soon as drafts are available, whichever the development model selected.

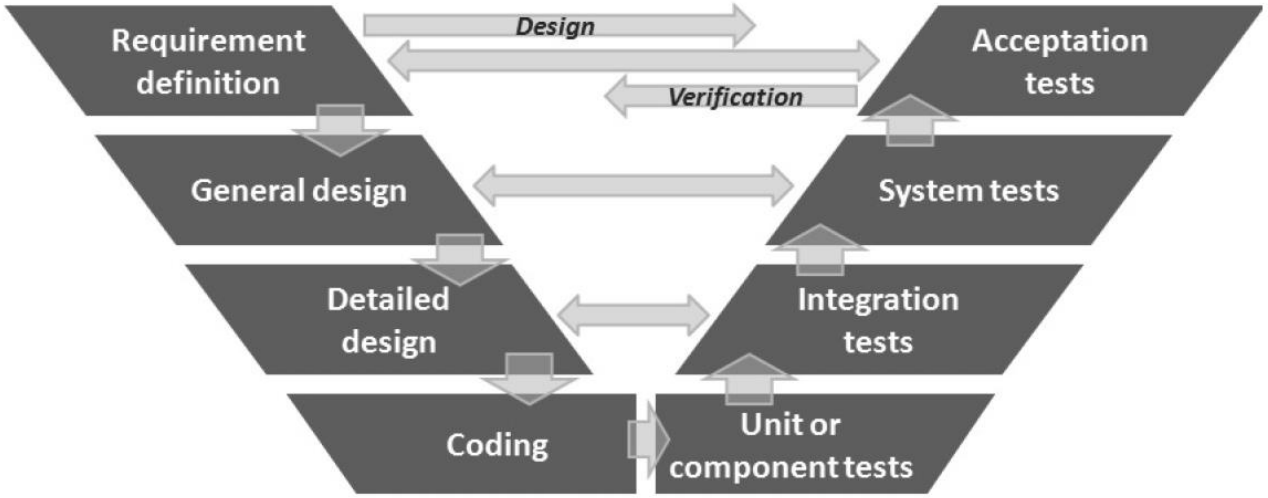
Waterfall software development model

- Applicable to applications with confirmed requirements at early stage
- Cons: Cost is huge once validation fails at later development phase



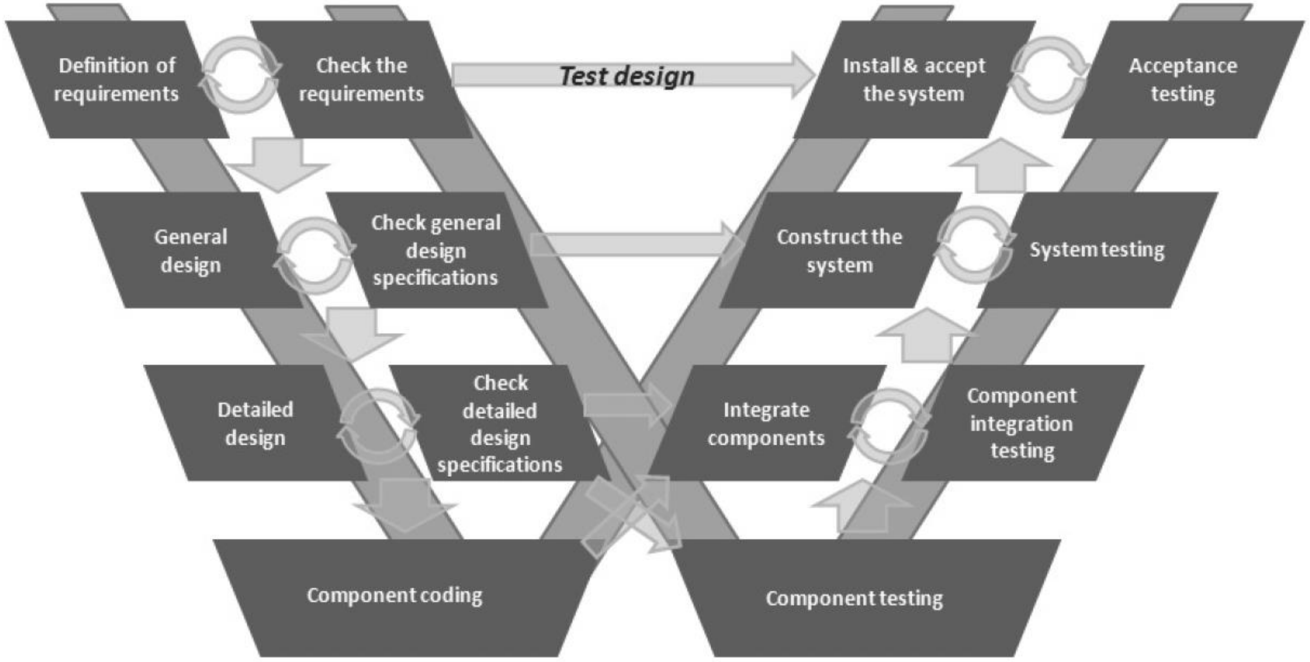
The V model

- Testing for each sequential stages
- Test design at the same time with the development stage



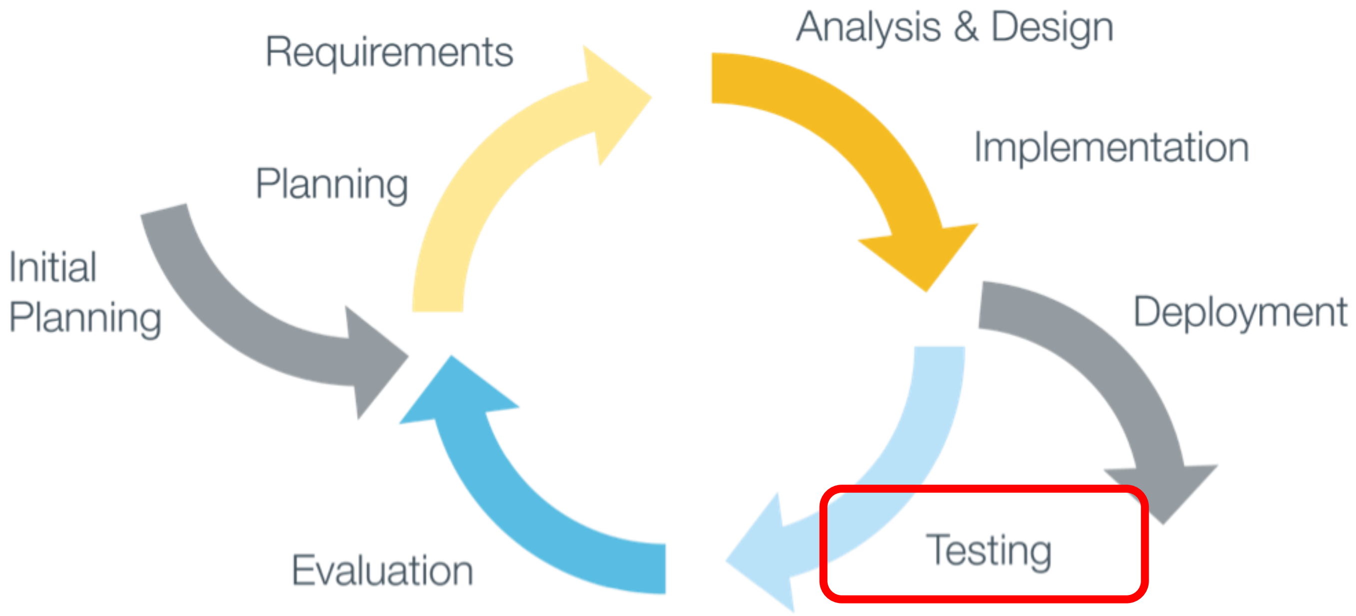
The W/VVV model

- Incorporating static testing techniques
- Early V&V



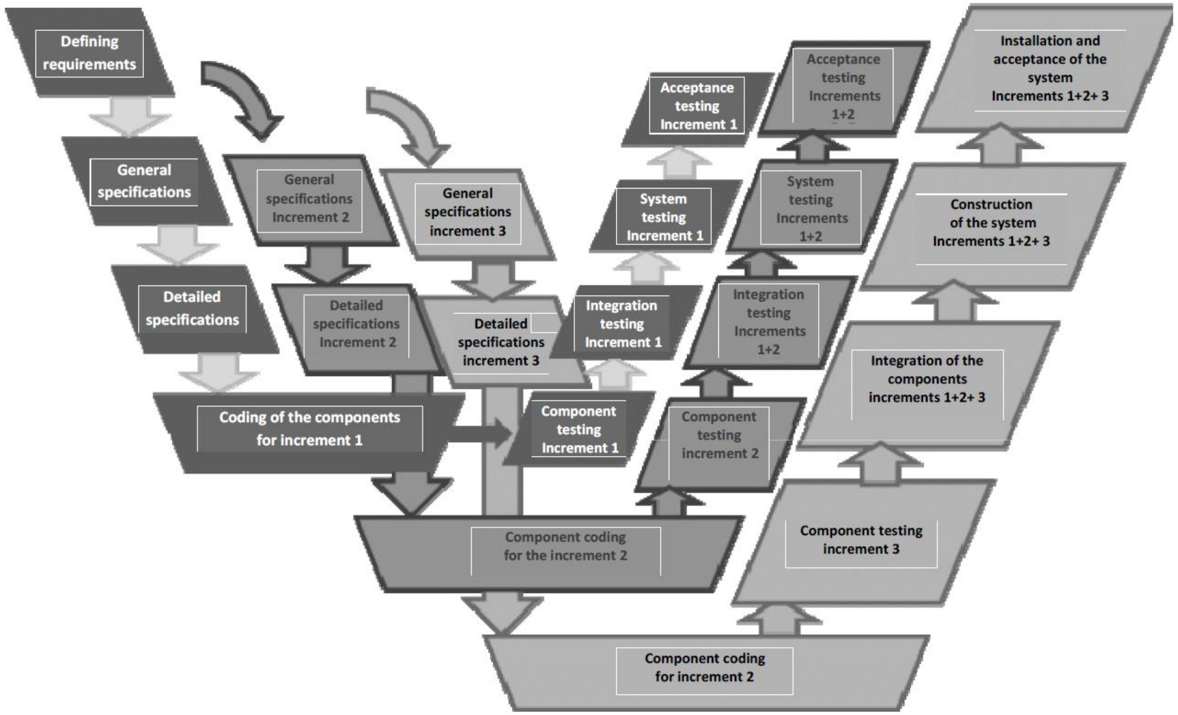
Iterative software development model

- Develop core functionalities first
- Improve/refine software in later iterations
- Problem: later iterations may damage previous iterations



Incremental model

- Increments expected before design
- Increments defined during design
- Cons: spend too much time developing an increment if system is not divided properly



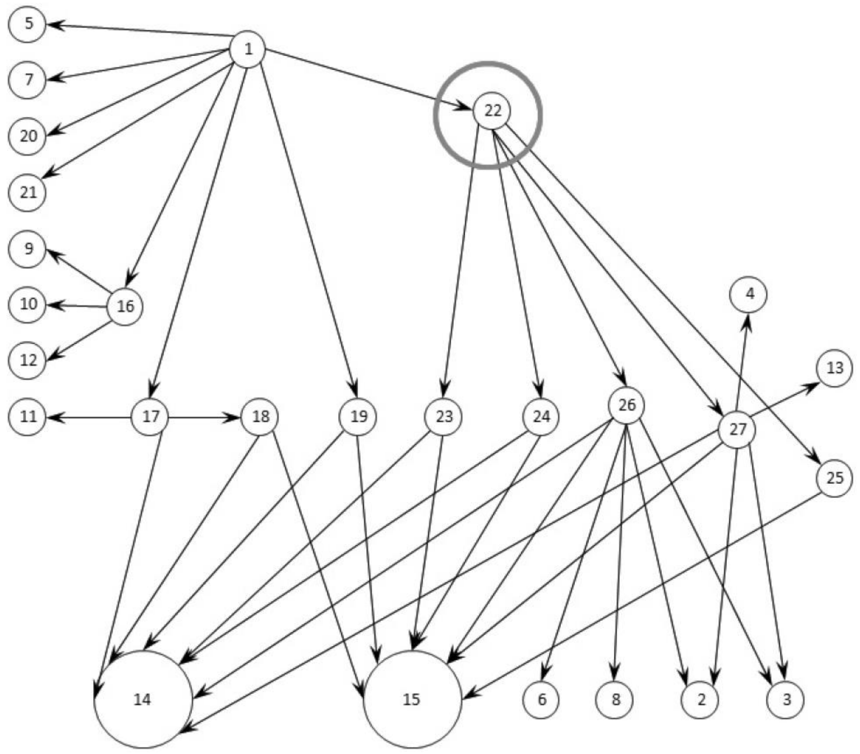
After a defect has been corrected

- Confirmation tests or retests
  - Verifying that the defect has been corrected and the software operates as expected

- Regression tests
  - Make sure that the correction did not introduce any side effects (regression) on the rest of the software

Regression test example

- Changes in node 22
- Direct impact: 1, 23, 24, 25, 26, and 27
- Indirect impact: 14, 15, 6, 8, 2, 3, 4, and 13



International Standard

- ISO/IEC/IEEE 29119 Software and Systems Engineering – Software Testing
    1. Concepts and definitions
    2. Test process
    3. Test documentation
    4. Test techniques

- An informative standard, not conformance standard

- Available on Blackboard

Test cases and Test suite

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite

Test cases and Test suite

- A test case is a triplet [I,S,O]:
  - I is the data to be input to the system,
  - S is the state of the system at which the data is input,
  - O is the expected output from the system.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an optimal test suite:
  - of reasonable size
  - to uncover as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - Many test cases do not contribute to the significance of the test suite,
  - Do not detect errors not already detected by other test cases in the suite.

- The number of test cases in a randomly selected test suite:
  - Not an indication of the effectiveness of the testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.

- Consider an example:
  - finding the maximum of two integers x and y.

Design of Test Cases

- If (x>y) max = x;
  else max = x;

- The code has a simple error:

- Test suite {(x=3,y=2);(x=2,y=3)} can detect the error,

- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

Test Design (TD) Process

- TD1: Identify feature set
- TD2: Derive test conditions
- TD3: Derive test coverage items
- TD4: Derive test cases
- TD5: Assemble test sets
- TD6: Derive test procedures

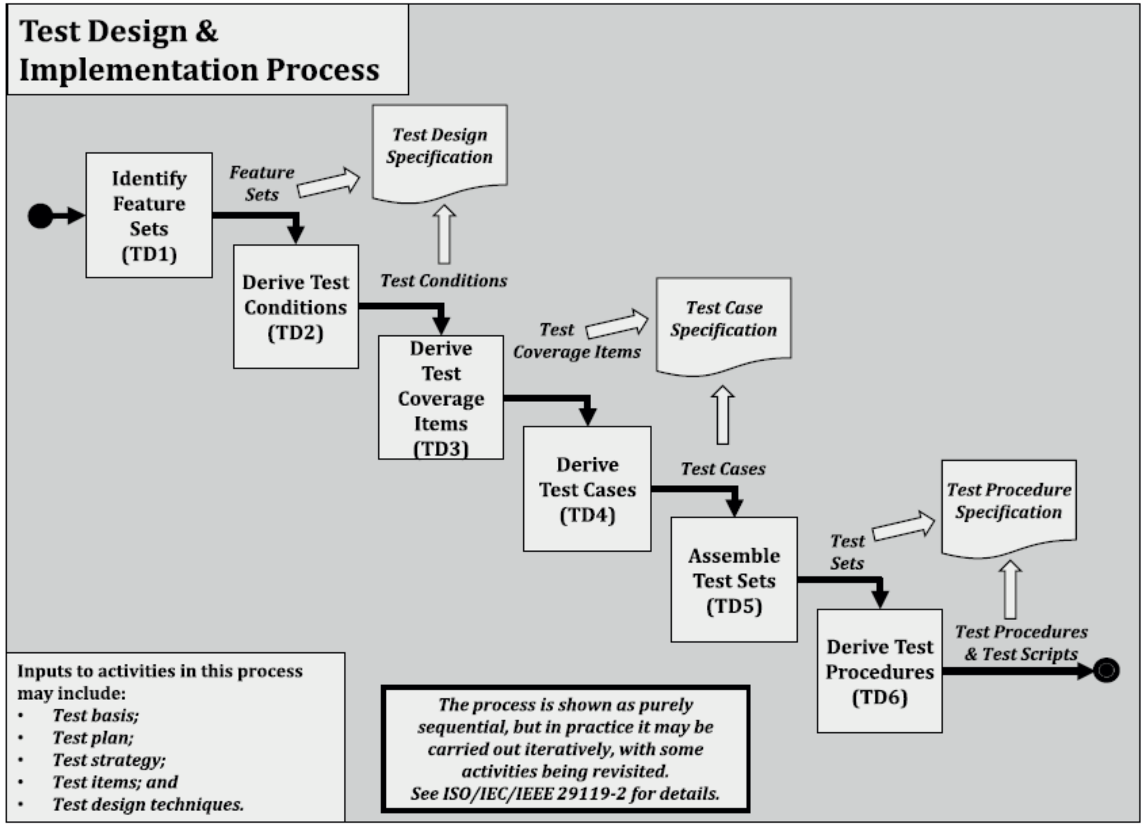


Figure 1 — ISO/IEC/IEEE 29119-2 Test Design and Implementation Process

Test Design Techniques

- Specification-based Testing
  - Black-box Testing

- Structure-based Testing
  - White-box Testing

- Experience-based Testing

Specification-based Testing

- Equivalence Partitioning
- State Transition Testing
- Scenario Testing

EQUIVALENCE PARTITIONING

Example: Equivalence Partitioning

- Homework (HW) 25pt
- Exam 75pt
- Specification: A function *generate_grading*
  - HW+Exam>=70 -> ‘A’
  - 50<= HW+Exam<70 -> ‘B’
  - 30<= HW+Exam<50 -> ‘C’
  - HW+Exam<30 -> ’D’
  - Invalid inputs -> ‘FM’

Step 1: Identify Feature Sets (TD1)

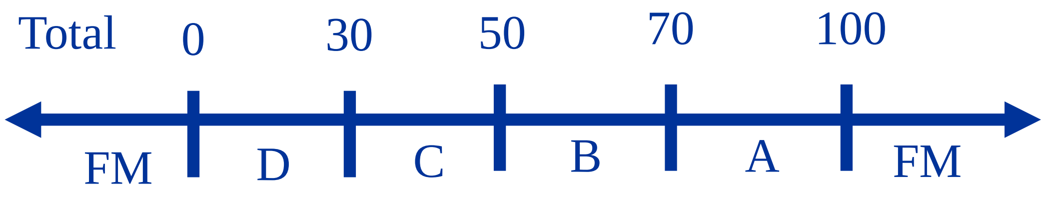
- FS1: generate_grading function

Step 2: Derive Test Conditions (TD2)

- Input Partitions
  - Valid
    - TCOND1: 0<=Exam<=75
    - TCOND2: 0<=HW<=25
  - Invalid
    - TCOND3: Exam<0
    - TCOND4: Exam>75
    - TCOND5: HW<0
    - TCOND6: HW>25
    - TCOND7: non-integer Exam input
    - TCOND8: non-integer HW input

Step 2: Derive Test Conditions (TD2) (CONT)

- Output Partitions
  - TCOND9: ‘A’ induced by 70<=Total<=100
  - TCOND10: ‘B’ induced by 50<=Total<70
  - TCOND11: ‘C’ induced by 30<=Total<50
  - TCOND12: ‘D’ induced by 0<=Total<30
  - TCOND13: ‘Fault Message’ (FM) induced by Total>100
  - TCOND14: ‘Fault Message’ (FM) induced by Total<0
  - TCOND15: ‘Fault Message’ (FM) induced by non-integer inputs



Step 3: Derive Test Coverage Items (TD3)

- Specify a test coverage item (TCOVER) for each test condition (TCOND)
  - TCOVER1: 0<=Exam<=75 (for TCOND1)
  - TCOVER2: 0<=HW<=25 (for TCOND2)
  - …

Step 4: Derive Test Cases (TD4)

- Attempt to “hit” Test Coverage Items
  - One-to-One: One test case for EACH Test Coverage item
    - More test cases but easy to automate
  - Minimized: Each test case may exercise more than one Test Coverage Items
    - Less test cases

Step 4: Derive Test Cases (TD4)
One-to-one

- Test cases for input Exam
- Test cases for input HW
- Test cases for non-integer inputs
- Test cases for valid output

Step 4: Derive Test Cases (TD4)
One-to-one

Test Case	1	2	3
Input (Exam)	60	-10	93
Input (HW)	15	15	15
Total	75	5	108
Test Coverage Item	TCOVER1	TCOVER3	TCOVER4
Partition Tested	0<=Exam<=75	Exam<0	Exam>75
Expected Output	‘A’	‘FM’	‘FM’

Step 4: Derive Test Cases (TD4) Minimized

Test Case	1	2	3	4
Input (Exam)	60	50	35	19
Input (HW)	20	16	10	8
Total	80	66	45	27
Test Coverage Item	TCOVER1 TCOVER2 TCOVER9	TCOVER1 TCOVER2 TCOVER10	TCOVER1 TCOVER2 TCOVER11	TCOVER1 TCOVER2 TCOVER12
Partition Exam	0<=Exam<=75	0<=Exam<=75	0<=Exam<=75	0<=Exam<=75
Partition HW	0<=HW<=25	0<=HW<=25	0<=HW<=25	0<=HW<=25
Partition Total	70<=Total<=100	50<=Total<70	30<=Total<50	0<=Total<30
Expected Output	‘A’	‘B’	‘C’	‘D’

Test Coverage Measurement

- $Coverage = \left(\frac{N}{T} \times 100\right)\%$
  - N is the number of test coverage items covered by test cases
  - T is the number of identified test coverage items

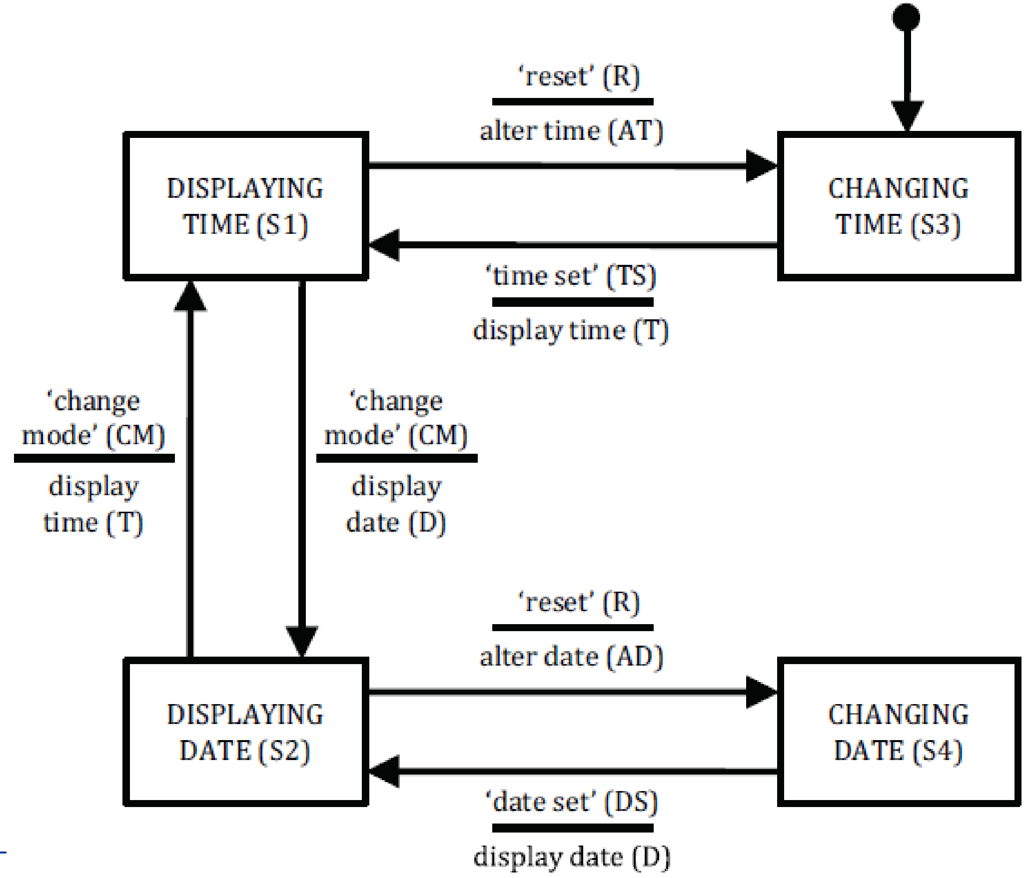
- Coverage is only measured for a particular criteria

- Coverage criteria has strength

STATE TRANSITION TESTING

Example: Manage Display

- A function: manage_display_changes
- 4 inputs
  - Change Mode (CM)
  - Reset (R)
  - Time Set (TS)
  - Date Set (DS)

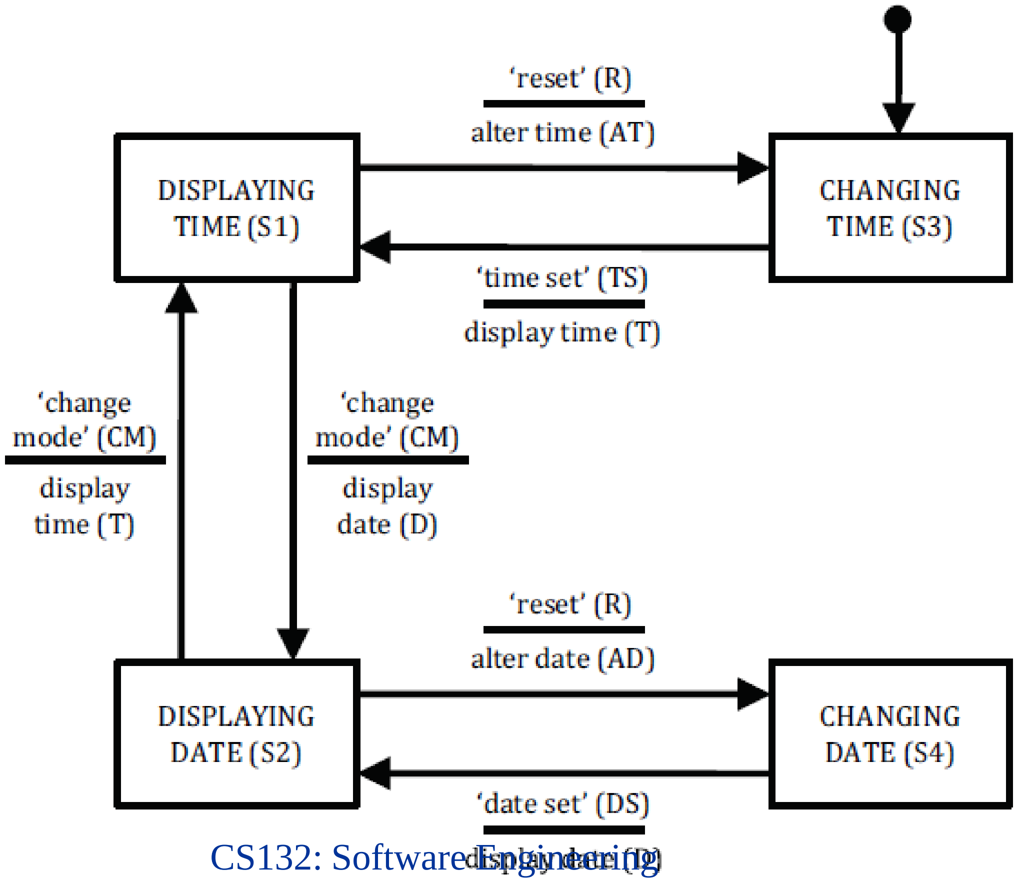


Step 1: Identify Feature Sets (TD1)

- FS1: manage_display_changes

Step 2: Derive Test Conditions (TD2)

- The state model is the test condition



Step 3: Derive Test Coverage Items

- All states
  - Test cases should visit all states in the model
- Single transition (0-switch coverage)
  - Only valid transitions
- All transitions
  - Both valid and invalid transitions
- Multiple transitions (N-switch coverage)
  - Valid sequences of N+1 transitions in the state model

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Step 3: Derive Test Coverage Items (TD3)

- TCOVER1: S1 to S2 with input CM (valid)
- TCOVER2: S1 to S3 with input R (valid)
- …

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Step 4: Derive Valid Test Cases (TD4)
0-switch test cases

- 0-switch test cases
- Invalid test cases should not cause state changes

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Table B.33 — 0-switch test cases for manage_display_changes

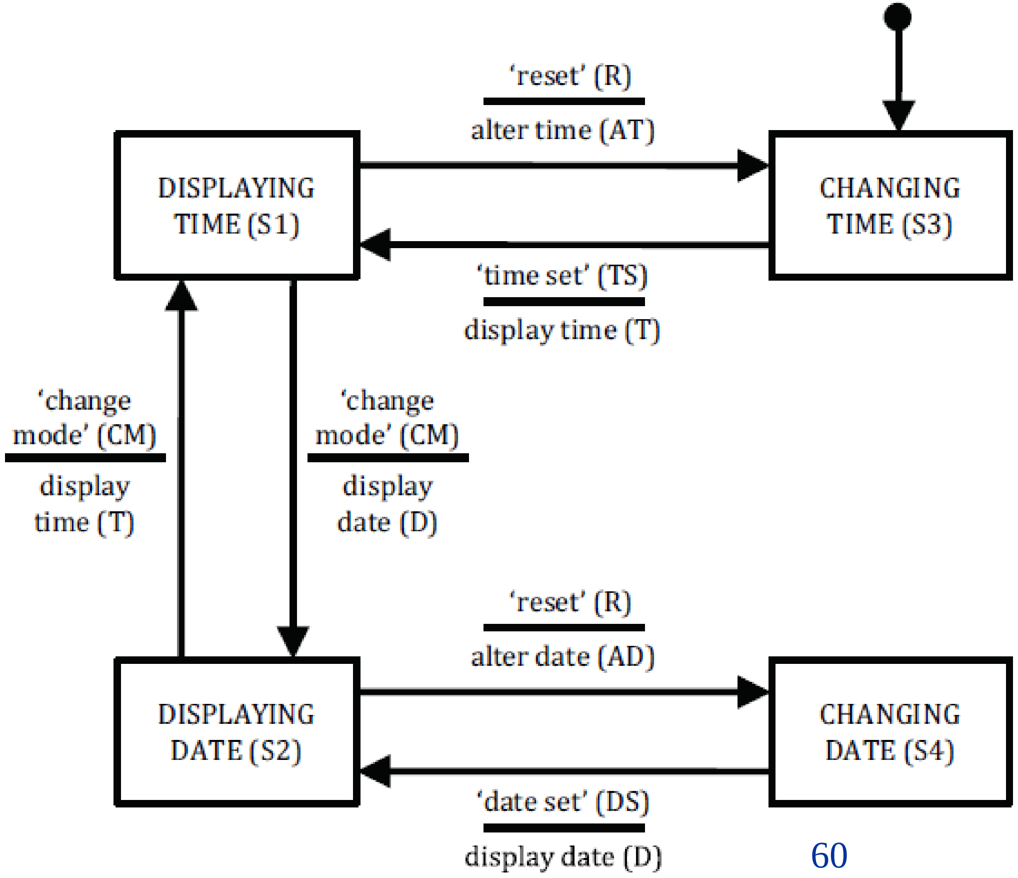
Test Case	1	2	3	4	5	6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	CM	R	TS	DS
Expected Output	D	AT	T	AD	T	D
Finish State	S2	S3	S1	S4	S1	S2
Test Coverage Item	1	2	5	6	11	16

Step 4: Derive Valid Test Cases (TD4)
1-switch test cases

- TCOVER 17: S1 to S2 to S1 with inputs CM and CM
- …

Table B.35 — 1-switch test cases for manage_display_changes

Test Case	17	18	19	20	21	22	23	24	25	26
Start State	S1	S1	S1	S3	S3	S2	S2	S2	S4	S4
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Expected Output	D	D	AT	T	T	T	T	AD	D	D
Next State	S2	S2	S3	S1	S1	S1	S1	S4	S2	S2
Input	CM	R	TS	CM	R	CM	R	DS	CM	R
Expected Output	T	AD	T	D	AT	D	AT	D	T	AD
Finish State	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4
Test Coverage Item	17	18	19	20	21	22	23	24	25	26



Step 5: Assemble Test sets

- TS1: 0 switch test cases - Test cases 1,2,3,4,5,6
- More efficient if rearranged to 5,1,4,6,3,2
  - The finish state of test case n is the start state of test case n+1

Table B.33 — 0-switch test cases for manage_display_changes

Test Case	1	2	3	4	5	6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	CM	R	TS	DS
Expected Output	D	AT	T	AD	T	D
Finish State	S2	S3	S1	S4	S1	S2
Test Coverage Item	1	2	5	6	11	16