Software Quality and Testing

Spring 2017

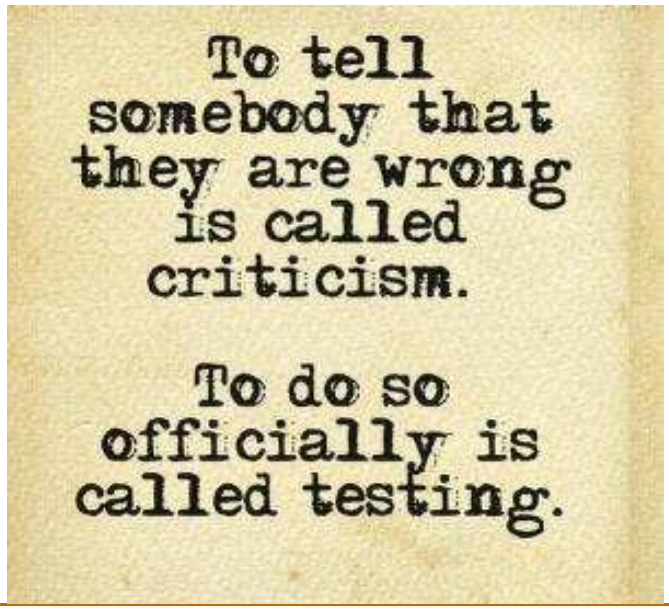
Instructor: Attila Kovács
Eötvös Loránd University, Budapest

Who am I ?

- Associate Prof. of Eötvös University, Budapest, Hungary, Faculty of Informatics
- 25 years of experience in Computer Science and Software Engineering
- MSc in Computer Science and Mathematics
- PhD in Informatics
- Founding member of the Hungarian Testing Board
- Consultant in the field of Software Quality and Testing
- ISTQB, IREB and Agile trainer

2

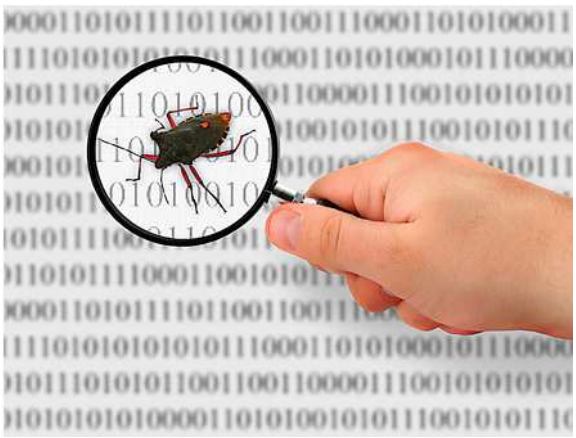
What is Testing?



3

Content

- Basic Notions
- Testing Principles
- Testing in the Project
- Levels of Testing
- Test Planning
- Quality and Risk
- Test Estimations
- Psychology of Testing
- Analysis and Design
- Static Testing, Reviews
- Supporting processes
- Test Automation
- Test Tools



4

Detailed Content

http://compalg.inf.elte.hu/~attila/materials/SoftwareQuality_tematika.pdf

5

Basic Notions

Why We Test?

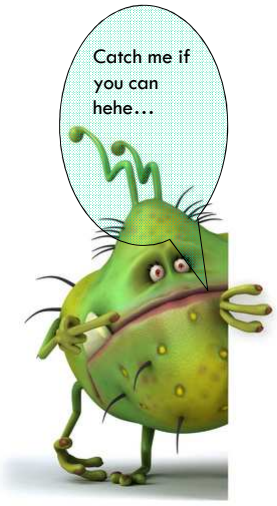
- *To use up spare budget*
- *To provide jobs for people who can't code*
- *To provide a good excuse why the project is late*
- *To prove that the software works as intended*

- *To provide confidence in the system*
- *To provide an understanding of the overall system*
- *To provide sufficient information to allow an objective decision on applicability to deploy*
- *Establish the extent that the requirements have been met*
- *Establish the degree of Quality*

7

Definitions

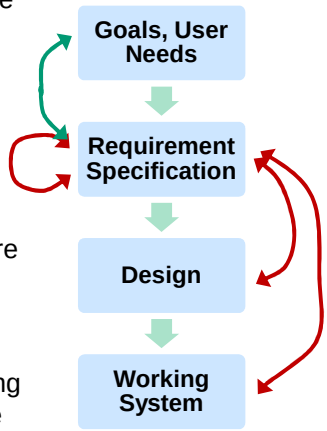
- ***Error / Mistake:***
  - Represents mistakes made by people
- ***Fault / Defect / Bug:***
  - Result of an error. May be categorized as
    - *Fault of Commission* – we enter something into representation that is incorrect
    - *Fault of Omission* – Designer can make error of omission, the resulting fault is that something is missing that should have been present in the representation
- ***Failure:***
  - Occurs when fault executes
- ***Incident:***
  - Behavior of fault. An incident is the symptom(s) associated with a failure that alerts user to the occurrence of a failure



8

Definitions

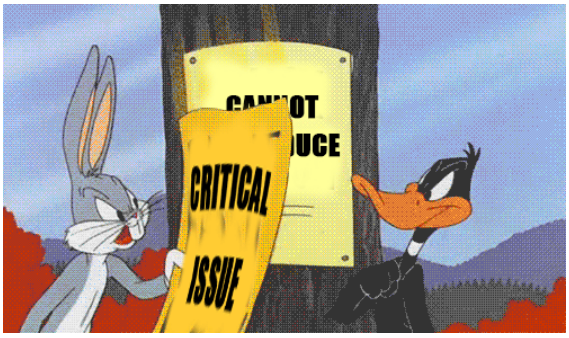
- ***Verification***: Demonstration of consistency, completeness, and correctness of the software artifacts at each stage of and between each stage of the software life-cycle.
  - Different types of verification: manual inspection, testing, formal methods
  - Verification answers the question:
    Am I building the product right?
- ***Validation***: The process of evaluating software at the end of the software development to ensure compliance with respect to the customer needs and requirements.
  - Validation can be accomplished by verifying the artifacts produced at each stage of the software development life cycle
  - Validation answers the question:
    Am I building the right product?



9

Definitions

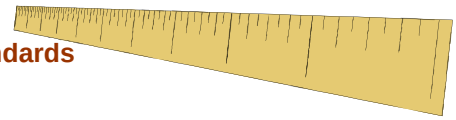
- ***Quality*** is the totality of the characteristics of an entity that bear on its ability to satisfy stated or implied needs. Or in other words, quality is the degree to which a component, system or process meets specified requirements and/or user needs and expectations.
- Does testing improve Quality?
  - *Testing does not build Quality into the software*
  - Testing is a means of ***determining the Quality*** of the Software under Test



10

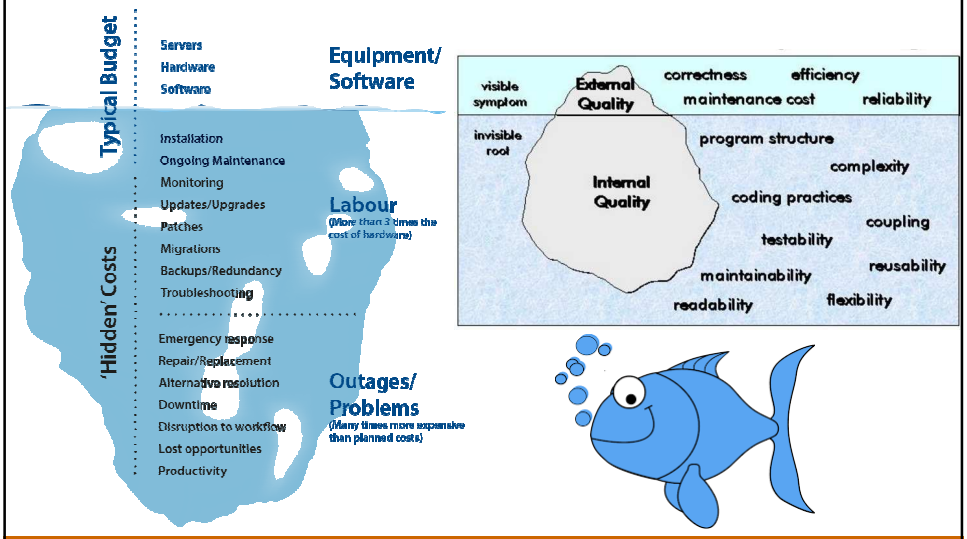
Some SW Quality Factors

- **Testability** – how easy is to test the application, clear, unambiguous requirements
- **Maintainability** – how easy it is for developers to maintain the application and how quickly maintenance changes can be made
- **Modularity** – how much of a system or computer program is composed of discrete components and a change to one component has minimal impact on another component
- **Reliability** – the time or transactions processed between failures in the software (MTBFs)
- **Efficiency** – how well a component performs its designated functions using minimal resources
- **Usability** – the ease of use of the software by the intended users
- **Reusability** – how easy it is to re-use elements of the solution in other identical situations
- **Legal requirements/Standards**
- etc.



11

Cost & Quality Iceberg



12

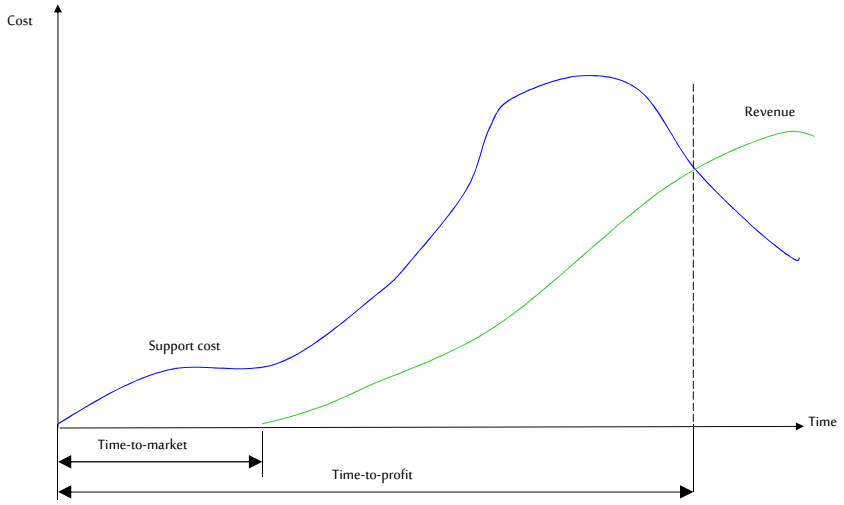
Some Faulty Assumptions

- Fallacy 1: Quality requirements dictate the schedule
  - Facts: For most software systems, market forces and competition dictate the schedule
- Fallacy 2: Quality = reliability
  - Fact: Reliability is only one component of the quality
- Fallacy 3: Users know what they want
  - Fact: User expectations are vague and general, not detailed and feature-specific. This is especially true for business software products. This phenomenon has led to *feature bloat*
- Fallacy 4: The requirements will be correct
  - Fact: Engineers are people, they evolve good requirements through trial-and-error experimentation
- Fallacy 5: Product maturity is required
  - Fact: Price and availability are far more important considerations in most business applications

13

Economics of Testing

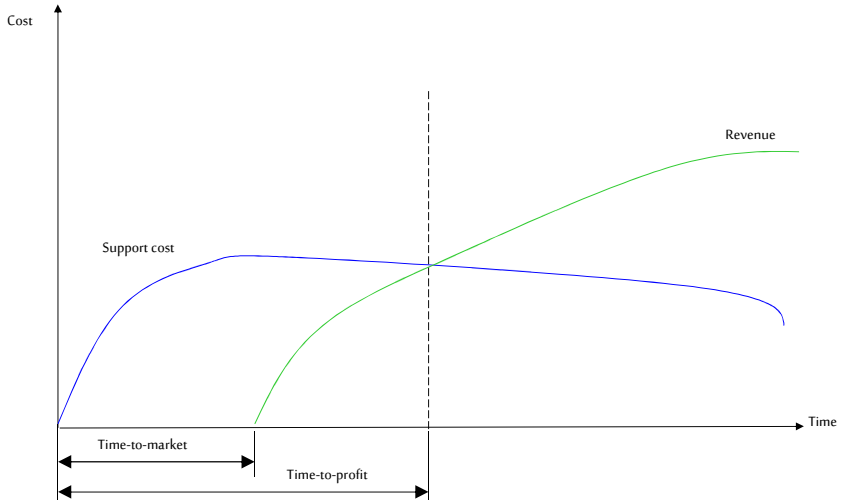
Focus on time-to-market



14

Economics of Testing

Focus on time-to-profit



15

Software Testing

- **The process** of executing a program with the intent to **certify its Quality** − Mills
- **The process** of executing a program with the intent of **finding faults / failures** − Myers
- **The process** of exercising software **to verify** that it **satisfies** specified functional & non-functional **requirements**
- **Examination of the behavior** of a program by executing the program on sample data sets.

16

Testing Dimensions

- *Testing* is not debugging
  - Debugging is the process to identify causes of failures in code and undertake corrections (remove them)
  - **Testing is a systematic exploration of a component or system to find and report defects**
  - Both are needed for achieving a good quality
- Testing has types: **static and dynamic**
  - Static testing: the code is not executed (e.g.: Reviews)
  - During dynamic testing the program under test is executed with some test data
- *Testing* is a **process**
  - Testing means not just test execution but, design, record, checking for completion
  - We design test process to ensure not to miss critical steps and do things in the right order
- *Testing* is a **set of techniques**
  - Always apply general principles of testing
  - We should use the best selection from different well proven test design techniques

17

Expectations on Software Testing

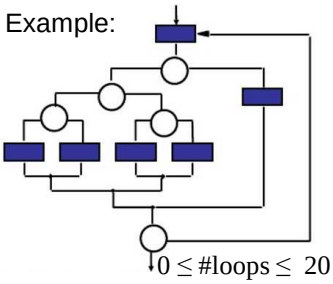
- In general, it is ***not possible to prove*** using testing that there are ***no faults*** in the software
- Testing should help **locate errors**, not just detect their presence
  - a "yes/no" answer to the question "is the program correct?" is not very helpful
- Testing should be **repeatable**
  - could be difficult for distributed or concurrent software

18

Is Testing Hard?

- If you are testing a bridge's ability to sustain weight, and you test it with 1000 tons you can infer that it will sustain weight ≤ 1000 tons
- This kind of reasoning does not work for software systems
  - software systems are not linear nor continuous
- ***Exhaustive Testing*** is a test approach in wich all possible data combinations are used.
- Exhaustively testing all possible input/output combinations is too expensive
  - the number of test cases increase exponentially with the number of input/output variables

Example:



How many possible path are there?

10^{14}

19

Definitions

- Let *P* be a program and let *D* denote its input domain
- A ***test data*** *t* is an element of input domain $t \in D$
  - a test data results a valuation for the input variables of the program
- A ***test data set*** *T* is a finite set of test data, i.e., a subset of *D*, $T \subseteq D$
- Exhaustive testing corresponds to setting $T = D$

The basic difficulty in testing is finding a test set that will uncover the faults in the program

20

Example – more tests results in better testing?

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

- Number of possible test data (assuming 32 bit integers)
  - $2^{32} \times 2^{32} = 2^{64}$
- How many tests then?
  - Test set
    {(x=3,y=2), (x=2,y=3)}
    will detect the error
  - Test set
    {(x=3,y=2),(x=4,y=3),(x=5,y=1)}
    will not detect the error although it has more test data
- **The power of the test data set is not determined by the number of its elements**

21

Example – is random testing enogh?

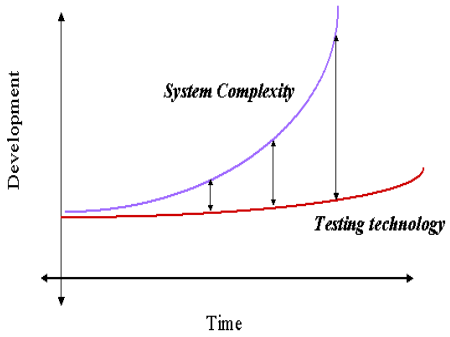
```
bool isEqual(int x, int y)
{
  if (x = y)
    z := false;
  else
    z := false;
  return z;
}
```

- If we pick test data randomly it is unlikely that we will pick a case where x and y have the same value
- If x and y can take 2^{32} different values, there are 2^{64} possible test combinations.
  - the probability of picking a case where x is equal to y is 2^{-32}
- It is not a good idea to pick the test data randomly (with uniform distribution) in this case
- So, **naive random testing is hopeless as well**…

22

Why Is Testing Hard?

- Budget-, time- and personnel constraints
- Great diversity of SW-environments
- Applications are be-coming increasingly large and complex
- SW-developers are neither trained nor motivated to test
- Testers are willing but incapable
- Lack of a testing culture



23

Why Errors / Defects Occur?

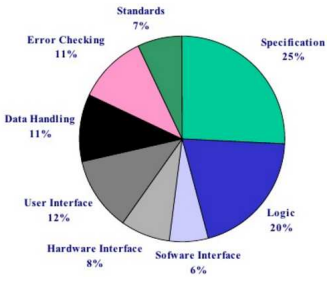
- No one is perfect! We all make mistakes or omissions
- The more pressure we are under the more likely we are to make mistakes
- Poor Training
- In IT development we have time and budgetary deadlines to meet

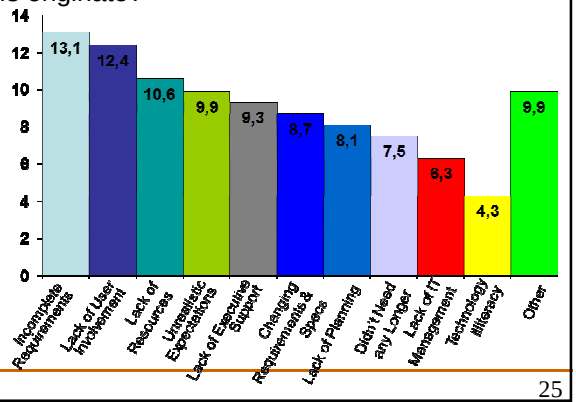
FAST & CHEAP then GOOD enough?
GOOD & CHEAP can be made FAST?
FAST & GOOD can be CHEAP?

24

Why Errors / Defects Occur?

- Poor Communication
- Requirements not clearly defined
- Requirements change & are not properly documented
- Data specifications not complete
- ASSUMPTIONS!
- But where quality problems originate?

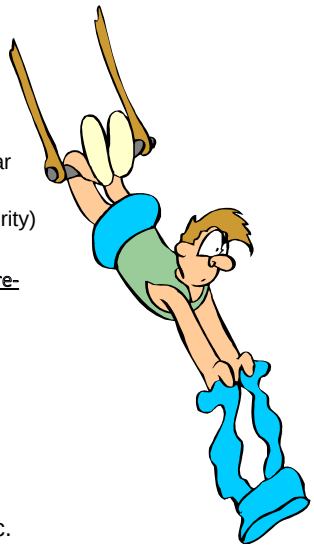




25

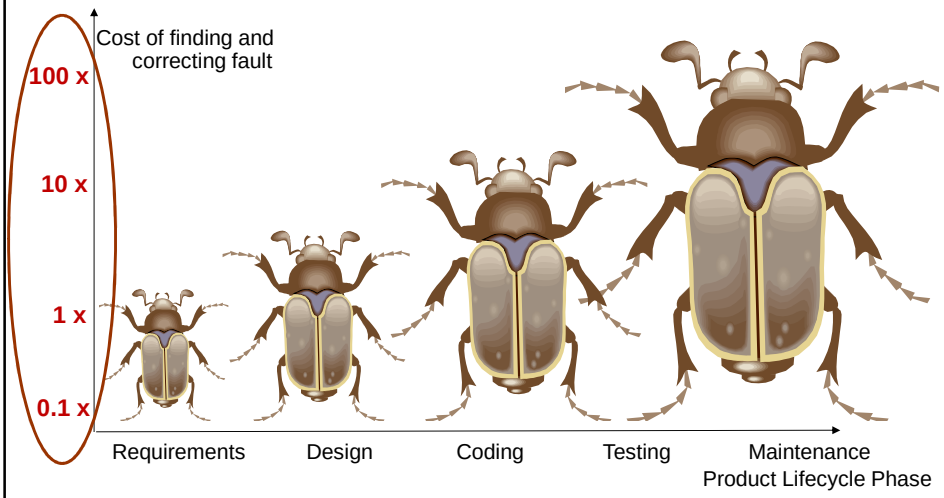
The Cost of Errors / Defects

- A single failure may incur little cost - or millions
  - Report layouts can be wrong - little cost
  - Or a significant error may cost millions…
    - Ariane V, Venus Probe, Mars Explorer and Polar Lander
    - UK government online filling of tax refund (security)
    - Denver Airport 1995, Pentium Chip 1994
    - https://raygun.com/blog/2014/05/10-costly-software-errors-history/
- In extreme cases a software or systems error may cost LIVES
  - Therac 25 Radiation Machine 1985-1987
- Usually safety critical systems are tested rigorously
  - Aeroplane, railway, nuclear power systems etc.



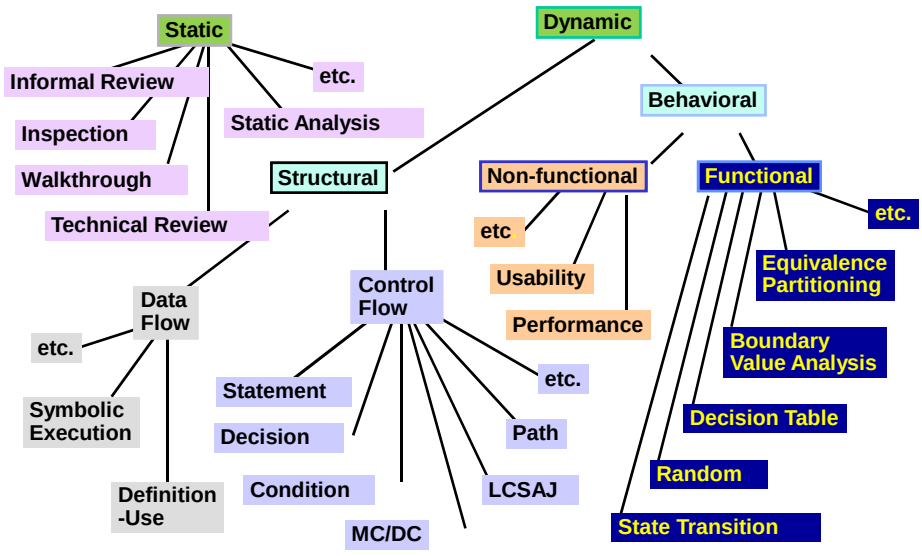
26

The Cost of Errors / Defects



27

Test Techniques



28

Basic Questions

- All IT project managers know that they must do some testing
- The basic questions are;
  - Is testing easy?
  - How much we need?
  - What sort?
  - By whom?
  - When?
  - How?
- These questions are difficult to answer

29

Remember: Why Testing Necessary?

- because software is likely to have faults ✓
- to learn about the reliability of the software ✓
- to fill the time between delivery of the software and the release date ✗
- to prove that the software has no faults ✗
- only because testing is included in the project plan ✗
- because failures can be very expensive ✓
- to avoid being sued by customers ✓
- to stay in business ✓
- to provide a measure of quality ✓

30

Remember: Exhaustive Testing

- What is exhaustive testing?
  - when all the testers are exhausted ✗
  - when all the planned tests have been executed ✗
  - exercising all combinations of inputs and preconditions ✓
- How much time will exhaustive testing take?
  - infinite time ✗
  - not much time ✗
  - impractical amount of time ✓

31

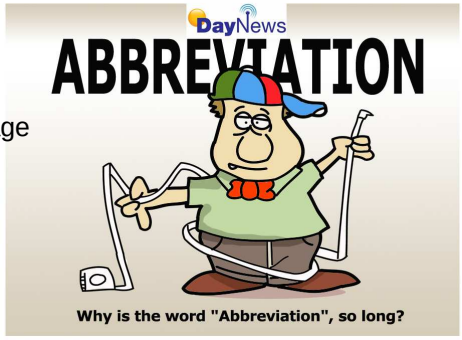
Abbreviations

- AUT – Application Under Test
- BS – British Standard
- BVA – Boundary Value Analysis
- CAST – Computer Aided Software Testing
- CM – Configuration Management
- COTS – Commercial Off-The-Shelf
- CR – Change Request
- FMEA – Failure Mode and Effect Analysis
- GUI – Graphical User Interface
- IEEE – Institute of Electrical and Electronics Engineers
- LOC – Lines of code
- MTBF – Mean Time Between Failure
- QA – Quality Assurance
- RAD – Rapid Application Development
- SUT – Software Under Test

32

Abbreviations

- RTM – Requirements traceability matrix
- SEI – Software Engineering Institute
- SDLC – Software Development Life-cycle
- SUT – Software Under Test
- TDD – Test-driven development
- TMM – Test Maturity Model
- UML – Unified Modelling Language
- V&V – Verification and validation
- XP – eXtreme Programming



33

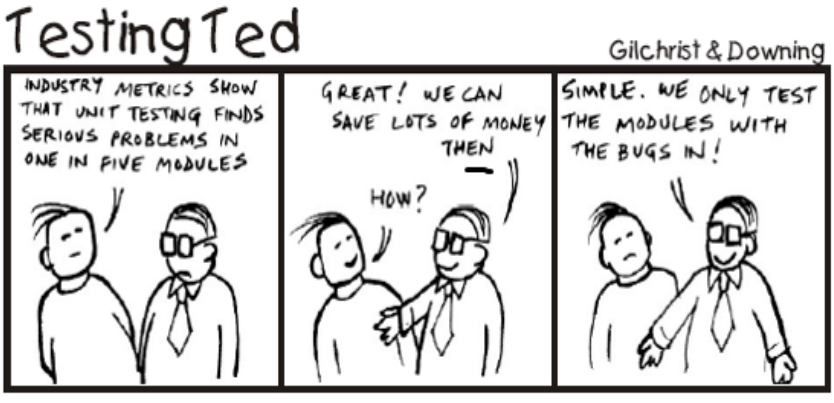
Testing Principles

Testing Principles

1. **Exhaustive testing** is impossible
2. **Early Testing**
   - Tests should be planned long before testing begins: testing should **start as early as possible**
3. **Testing is context dependent**
4. **Pesticide paradox**
   - Running the same set of tests continually will not find new defects.
5. **Pareto Principle / Defect Clustering**
   - Approx. 80% of faults occur in 20% of code

35

Defect Clustering



36

Testing Principles

6. **Absence of errors fallacy**
   - Just because testing didn't find any defects in the software, it doesn't mean that the software is perfect, or ready to be shipped
7. **Presence of defects**
   - „Testing can only show the presence of bugs, not their absence" (Dijkstra)
8. **Testing is** not just a process for measuring the quality of the product. It needs to be able to add to **the value of the product**.
9. **All tests should be traceable** to customer requirements
10. **Tests must be ranked** so that, whenever you stop testing, you have done the best testing in the time available.

37

How to Prioritize?

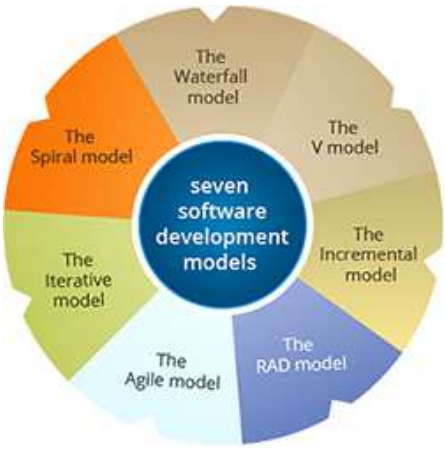
- Possible ranking criteria
  - test where failures would be most visible
  - test where failures are most likely
  - ask the customer to prioritize the requirements
  - what is most critical to the customer's business
  - areas changed most often
  - areas with most problems in the past
  - most complex areas, or technically critical

38

Testing in the Project

Software Development Models

- Testing does not exist in isolation
  - Test activities are related to software development activities
- Software development models consist of **processes and methodologies** that are being selected for the development of the project depending on the project's aims and goals
- Different development life-cycle models need different approaches to testing



40

Software Development Life-cycle

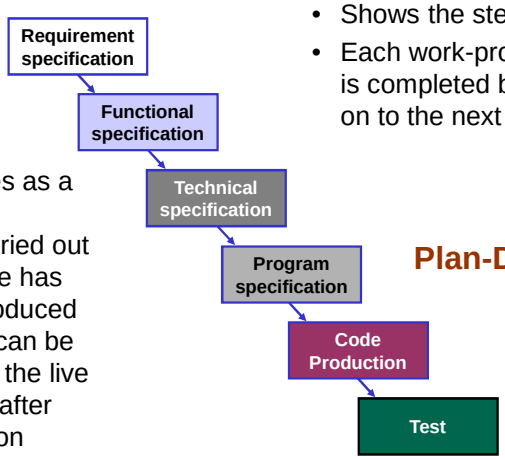
- What is involved in a development life-cycle for a software product?

Typical life-cycle phases include:

Requirement Specification
Conceptual Plan
Architectural Design
Detailed Design
Component Development
Integration
System Qualification
Release
System Operation & Maintenance
Retirement / Disposal

41

Sequential Models - Waterfall



- Shows the steps in sequence
- Each work-product or activity is completed before moving on to the next

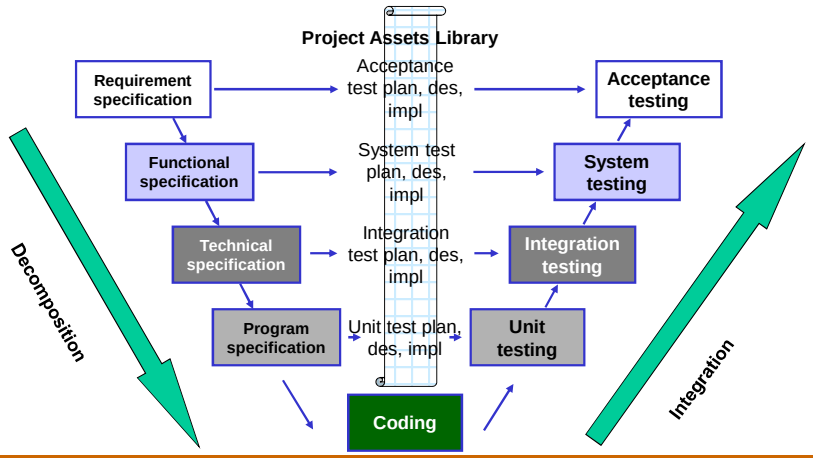
- Testing serves as a quality check
- Testing is carried out once the code has been fully produced
- The product can be released into the live environment after test completion

Also called: Plan-Driven Model

42

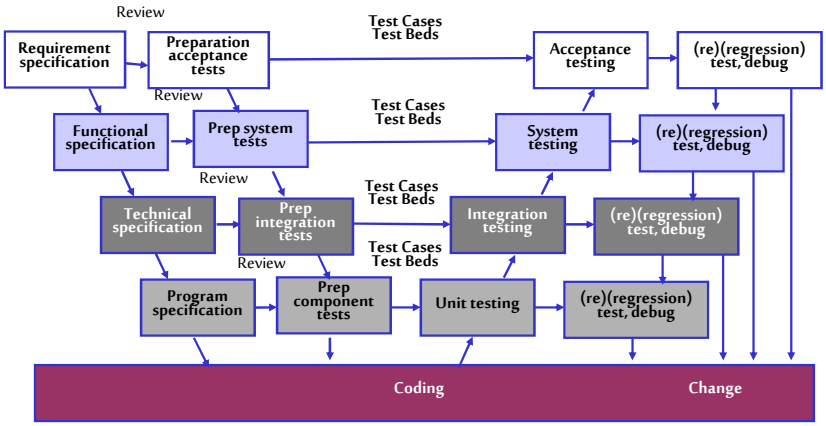
Sequential Models – V-model

- Extension of the waterfall model
- **Defects can be identified as early as possible in the life-cycle**
- In practice, a V-model may have fewer, more or different levels of development and testing, depending on the project and the software product



43

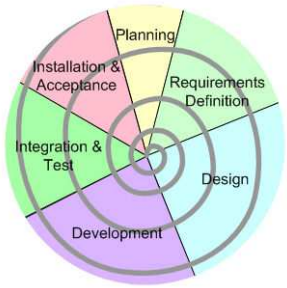
W-model – NOT sequential any more

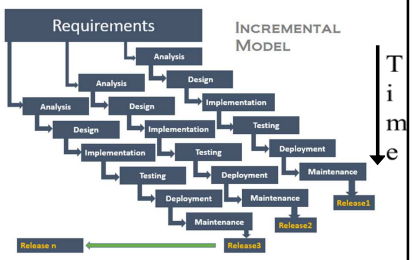


44

Iterative & Incremental

- **Iterative**
  - Iterative development involves a **cyclical process**. **Learning from one iteration informs the next iteration.** An iterative process embraces the fact that it is very difficult to know upfront exactly what the final product should look like
- **Incremental**
  - Incremental development involves **breaking a large chunk of work into smaller portions**. This is typically preferable to a monolithic approach where all development work happens in one huge chunk.





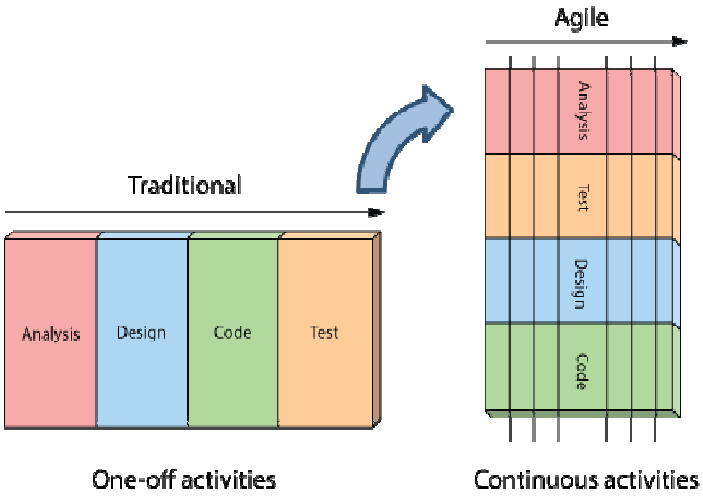
45

Iterative Models

- Forms of iterative development include
  - Prototyping
  - Rapid application development (RAD) („tool supported development")
  - Unified Process (former RUP)
  - Agile software development
    - XP
    - Scrum,
    - Kanban

46

Agile



47

Agile

The Agile Manifesto

Individuals and interactions	over	Processes and Tools
Working Product	over	Comprehensive Documentation
Customer Collaboration	over	Contract Negotiation
Responding to change	over	Following a plan

That is, while there is value in the items on the right, we value the items on the left more.

www.agilemanifesto.org

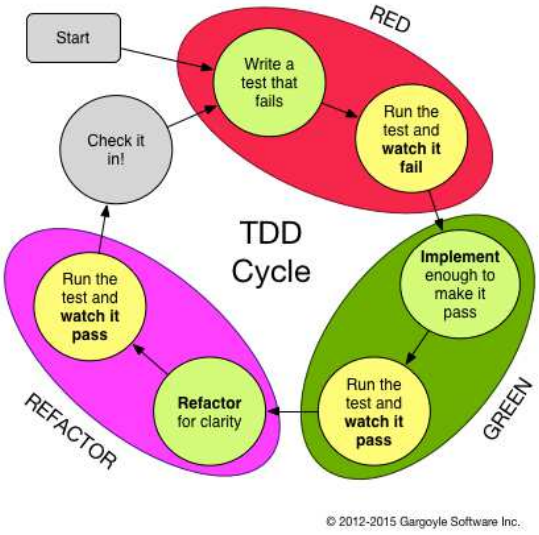
48

XP Principles

- **Planning Game**: requirements and those priorities defined by client
- **Small releases**: very simple, working system which is developed continually
- **Simple design**: the simplest solution to a requirement
- **Metaphor**: common system naming conventions in the project
- **Test before coding** (**TDD**)
- **Refactoring**
- **Pair programming**
- **Common ownership** of code
- **Continuous integration**, build
- **40-hours work** per week
- **Internal client**
- **Coding regulations**

49

Test Driven Development



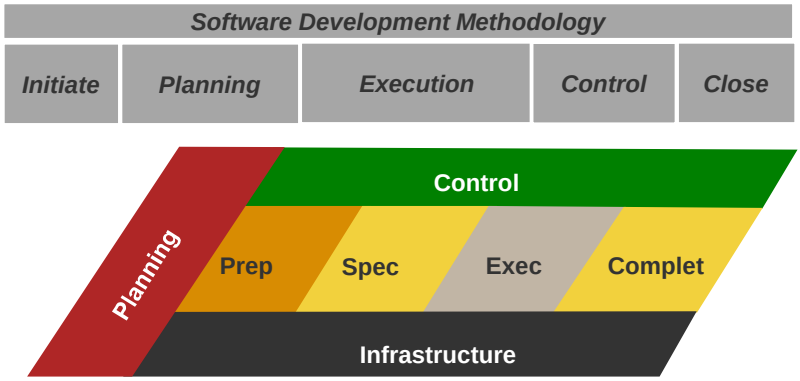
50

SCRUM Principles and Processing

- **Split your organization** into small, cross-functional, self-organizing teams.
- **Split your work** into a list of small, concrete deliverables. Sort the list by priority and estimate the relative effort of each item.
- **Split time** into short fixed-length iterations (usually 1-4 weeks), with potentially shippable code demonstrated after each iteration
- **Optimize the release plan and update priorities** in collaboration with the customer, based on insights gained by inspecting the release after each iteration.
- **Optimize the process** by having a retrospective after each iteration.

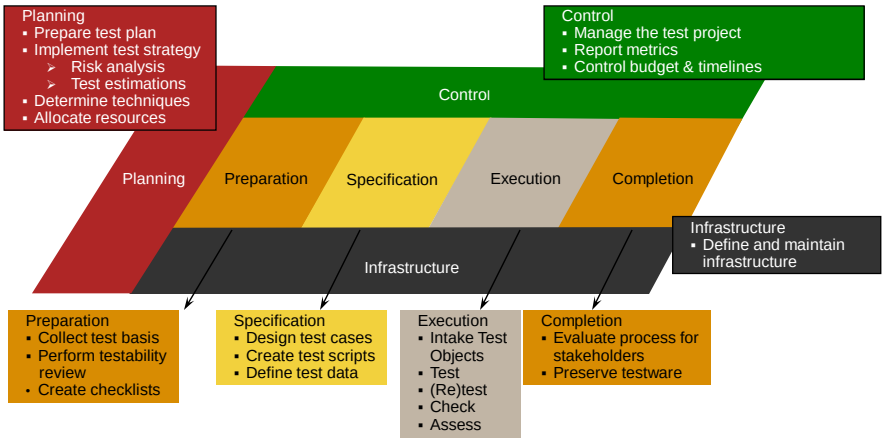
51

Testing Life-Cycle



52

Testing Life-Cycle



53

Notions

- ***A Test Scenario*** is a high level classification of test requirements grouped depending on the functionality. It can be derived from use cases. A good test scenario reduce the complexity and repeatability of the product.

Identifier
Scenario_01
Conditions satisfied
Req 12 **TRACEABILITY**
Scenario description
Run the application so that the UI panel appears. Enter invalid values for **a**, **b** and **c** respectively. Click on „Compute".
Output
The result appears on the screen or error message is displayed, „Unable to compute…"

no specific values for input data

54

Notions

- ***A Test Case*** is a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
  - Example:

Identifier	Scenario_01_Test_01
Test case creator	Attila Kovács
Version	0.1
Name of test case	Verify that an error message appears when entering invalid values
Identifier of requirement	PR12 **TRACEABILITY!!!!!**
Purpose	To verify that an error message appears when entering invalid values, such as non-integers, chars or strings.
Dependencies	None
Test environment	Java 1.5.0 installed
Initialization	Run the program so that the UI panel appears
Finalization	Close the UI panel
Actions	In the UI panel enter the following data, in the corresponding field: a = 99, b = 1, c = „Blahblah” Click on „Compute" button.
Input data	a = 99, b = 1, c =„Blahblah”
Expected result	The following error message is displayed, written in red italic: „Computation error, reason: invalid input parameters."
Actual result	

55

Test Case Structure – Information Part

- **Identifier** is a unique identifier of the test case for further references, for example, while describing found defect
- **Test case owner/creator** is name of tester or test designer, who created test or is responsible for its development
- **Version & Date** of current test case definition
- **Name of test case** should be human-oriented title which allows to quickly understand test case purpose and scope
- Identifier of **requirement** which is covered by test case
- **Purpose** contains short description of test purpose, what functionality it checks
- **Priority** It is useful while executing (e.g. Low, Medium, High)
- **Dependencies**

56

Test Case Structure – Activity Part

- **Testing environment/configuration** contains information about configuration of hardware or software which must be met while executing test case
- **Initialization** describes actions, which must be performed before test case execution is started For example, we should open some file
- **Finalization** describes actions to be done after test case is performed. For example if test case crashes database, tester should restore it before other test cases will be performed
- **Executed by**
- **Expected average execution duration**
- **Actions** step by step to be done to complete test.
- **Input data** description

57

Test Case Structure – Results Part

- **Expected results** contains description of what tester should see after all test steps has been completed
- **Actual results** contains a brief description of what the tester saw after the test steps has been completed. This is often replaced with a **Pass/Fail**. Quite often if a test case fails, reference to the defect involved should be listed in this column
  - **Actual results** field will be filled in **after running** (executing) the test case
- **Status** (PASS / FAIL)
- **Note**

58

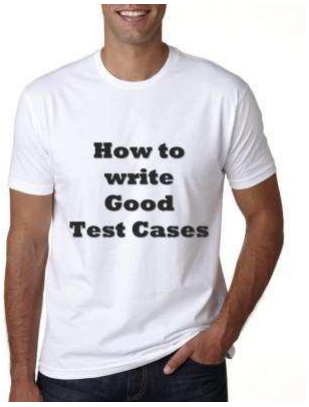
Approaches to Test Case Design

- **Testing-by-contract** is based on the design-by-contract philosophy
  - Its approach is to create test cases *only for the situations* in which the *pre-conditions are met*. Pre-conditions define what that SUT requires so that it can meet its post-conditions. *Post-conditions* define what a SUT promises to do. Pre-conditions and post-conditions establish a contract between a SUT and others that invoke it.
- **Defensive test case design**
  - In this case the module is designed to accept any input. If the normal preconditions are met, the module will achieve its normal post-conditions. If the normal pre-conditions are not met, the module will notify the caller by returning an error code or throwing an exception. This notification is actually one of the SUT's post-conditions. It is an approach that *tests under both normal and abnormal pre-conditions*.

59

A Good Test Case

Accurate: Exacts the purpose
Economic: Cheap to use, no unnecessary steps
Effective: Find faults
Exemplary: Represents others
Evolvable: Easy to maintain
Executable independently by other testers
Repeatable: Can be used to perform the test over and over
Reusable: Can be reused if necessary
Traceable: Capable of being traced to requirements



60

Notions

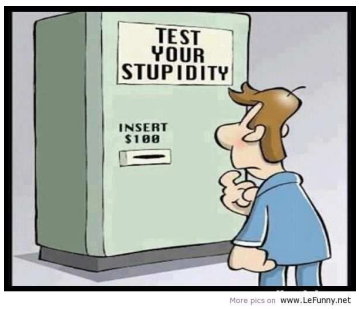
- ***Test Condition*** is an item or event of a system that **could be verified by one or more test cases**. Eg; transaction, function, structural element etc. Test condition can be a piece of functionality or anything you want to verify.
  - Example test condition:
    - ***Given*** a Login Form ***When*** User Name and Password are valid ***Then*** application will move forward
- ***Test Basis*** is defined **as the source of information** or the document that is needed to write test cases and also for test analysis.
  - Test basis should be well defined and adequately structured so that one can easily identify test conditions from which test cases can be derived
  - Typical Test Basis:
    - Business document, Requirement document
    - Legacy
    - Codes Repository

61

Notions

- ***Test Script*** (test procedure) is a *sequence of actions for the execution of a test.*
  - Can be manual or automated
  - Example (in Python):

```
def sample_test_script (self):
    type ("TextA")
    click (ImageButtonA)
    assertExist (ImageResultA)
```



- ***Testware:*** *Artifacts produced during the test process* required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.

Testware		
Test Materials	**Test Results**	
	Products	**By-Products**
Input, Scripts, Documentation, Data, Expected Outcome	Actual Outcome	Log, Status, Difference Report

62

Conditions, Scenarios, Cases, Scripts

- Identify *Test Condition (what)*

Test Condition is an item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality, attribute or structural element.

- Specify *Test Scenario and Test Cases (with which)*

Test Scenario is for testing the end-to-end functionality of a software application and ensure the business processes and flows are functioning as needed.

Test Case is a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

- Specify *Test Procedure (how)*

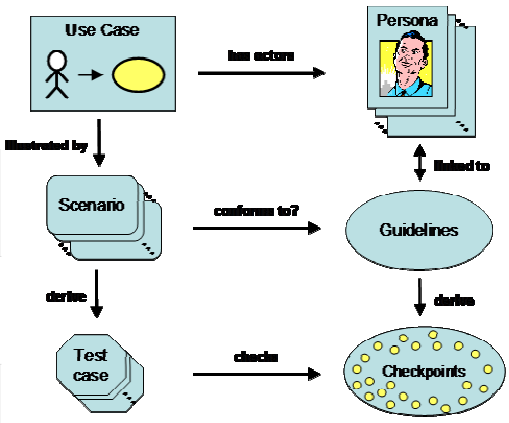
Test Procedure specification is a sequence of actions for the execution of a test (Test Script).

63

Test Scenario or Test Case? Example.

Requirement is to test your phone WiFi:

Test Scenario	Verify that device automatically connects to Wi-Fi if user creates new profile
Test cases	Test case 1: Create WiFi profile and verify that it created successfully
	Test case 2: Verify that device is able to connect to Wi-Fi



64

Levels of Testing

Test Levels

Unit test | Integration test | System test | Acceptance test

- For all types of developments testing plays a significant role
- Tests are frequently grouped by where they are added in the software development process.
- There are generally four recognized levels of tests:
  - unit testing,
  - integration testing,
  - system testing and
  - acceptance testing.

66

Test Levels – Unit Tests

Unit test | Integration test | System test | Acceptance test

- Generally code is written in parts or units
- The goal of unit testing is to isolate each part of the program and show that the individual parts are correct
- Intuitively, **one can view a unit as the smallest testable part of an application.**
- Units are also called modules or components
- Units are usually **constructed in isolation** for integration at a later stage
- Unit test
  - Ensures that the code meets its specification prior to its integration with other units
  - Verify that all of the code that has been written for the unit can be executed
  - Usually performed by the developer who wrote the code
- Defects found and fixed during unit testing are often not recorded

67

Test Levels – Unit Tests

Unit test | Integration test | System test | Acceptance test

- Unit testing include testing of **functionality** and may include specific **non-functional characteristics** such as resource-behaviour (e.g. memory leaks), performance or robustness testing, as well as **structural testing**.
- A unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming a unit is often an entire interface, such as a class, but could be an individual method as well.

Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions.

A ***memory leak*** occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released. In object-oriented programming, a memory leak may happen when an object is stored in memory but cannot be accessed by the running code.

68

Test Levels – Unit Tests

Unit test | Integration test | System test | Acceptance test

- Structure based unit test methods:
  - Statement testing
  - Decision testing
  - Condition testing
  - Condition determination testing
  - Multiple condition testing
  - Path testing
  - Loop testing
- Unit tests are usually supported by tools (e.g. JavaCov, JUnit for Java)

69

Test Levels – Integration Tests

Unit test | Integration test | System test | Acceptance test

- The purpose is to **expose defects in the interfaces and in the interactions between integrated components or systems**
- Integration test types:
  - Component integration test

***Component integration** test* focuses on the interactions between software components and is done after component (unit) testing.

  - System integration test

***System integration** test* focuses on the interactions between different sub-systems and may be done after system testing of each individual sub-systems.

70

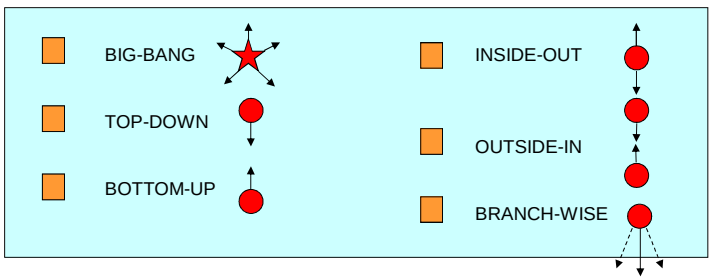
Example: Is Integration Testing Important?

71

Test Levels – Integration Tests

Unit test | Integration test | System test | Acceptance test

- Integration strategies:

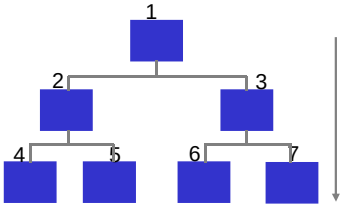


72

Test Levels – Integration Tests

Unit test | Integration test | System test | Acceptance test

- Top-down integration
  - The system is built in stages, starting with components which call other components
  - Components which call others are usually placed above those that are called
  - *Stubs or mocks* are used for components *not yet integrated*



Stub is a passive component, called by other components. It replaces the called component.

73

Stubs and Mocks

- ***Stubs*** can be thought of as ***inputs to*** the code under test. When called, they behave a certain way – return a fixed value, throw an exception, calculate a return value based on parameters, pull from a sequence of values, etc.
- ***Mocks*** can be thought of as ***outputs from*** the code under test making assertions on behavior. Mocks say *„I expect you to call foo() with bar, and if you don't there's an error".*

Typically, using a Mock will consist of three phases:

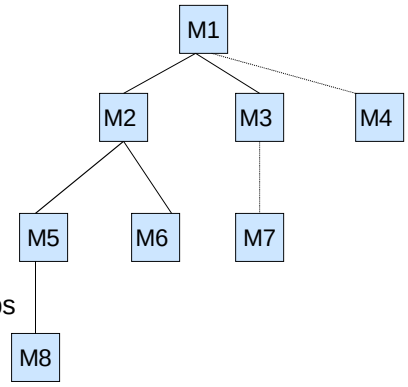
- Creating the instance
- Defining the behavior
- Asserting the calls

74

Test Levels – Integration Tests

- Top-down integration: modules subordinate to main control module are incorporated
  - Depth-first
  - Breadth-first

Pairs: What get's tested first in breadth first?



  - M1 tested with stubs for M2, M3 and M4
  - Then M1-M2 tested with stubs for M3, M4, M5, and M6
  - Then M1-M2-M3 tested with stubs for M4, M5, M6, and M7

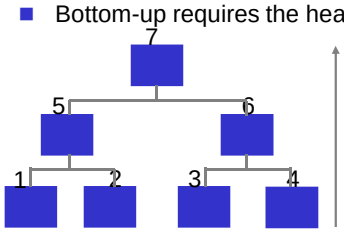
…

75

Test Levels – Integration Tests

Unit test | Integration test | System test | Acceptance test

- Bottom-up integration
  - The opposite of top-down integration
  - Components are integrated in a bottom-up order
  - These are then tested and added to the modules above them to form larger sub-systems which are then tested
  - Bottom-up requires the heavy use of *drivers* instead of *stubs*



Driver is a specially designed user interface which enables test data to be inputted and passed to a sub-system which is being tested. Only used in testing and does not have any place in the final system.

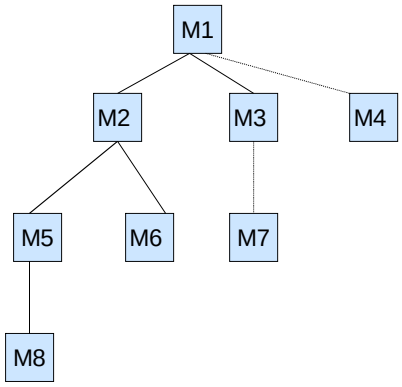
76

Test Levels – Integration Tests

- Bottom-up integration

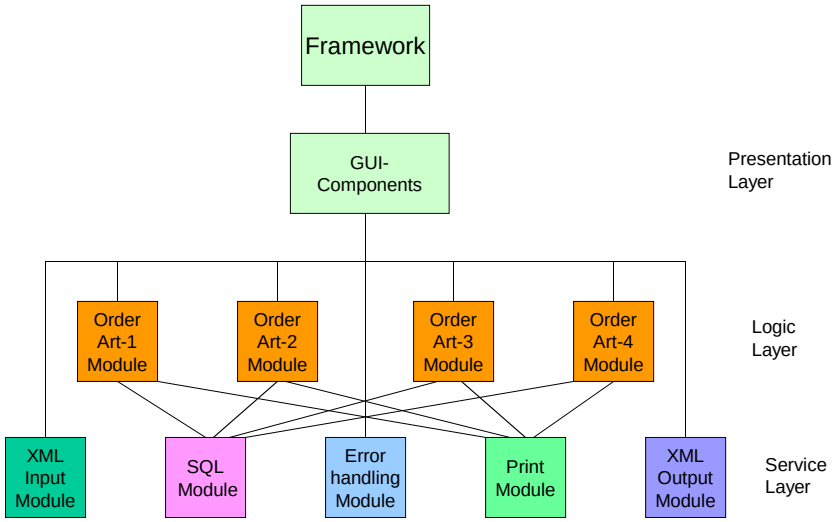
Pairs: What get's tested first?

– M8 tested with drivers
– M5-M8 is tested with drivers for M5
– M5-M6-M8 is tested with drivers for M5-M6
…



77

Integration Tests - Example

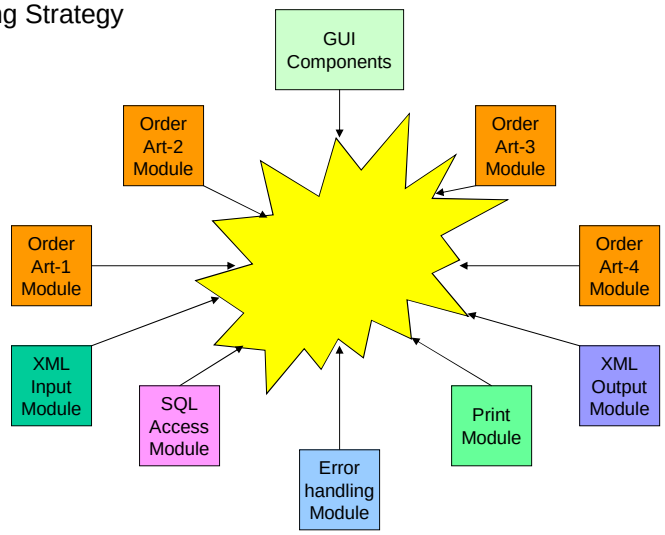


Order Entry Processing System
Sample of a three-layered Program

78

Integration Tests – Big Bang

Big Bang Strategy



79

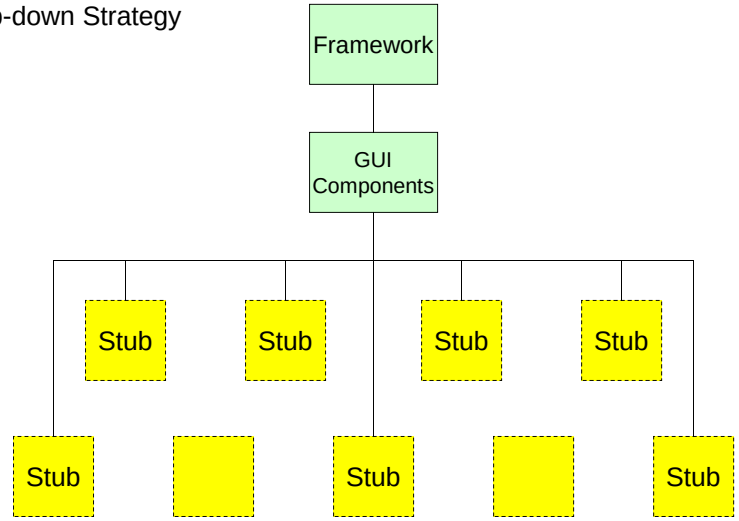
Integration Tests – Big Bang

- Big Bang Strategy
  - Every module is unit tested in isolation
  - After all of the modules are tested they are all integrated together at once and tested
  - No driver or stub is needed
  - However, in this approach, it may be hard to isolate bugs…

80

Integration Tests – Top-down Strategy

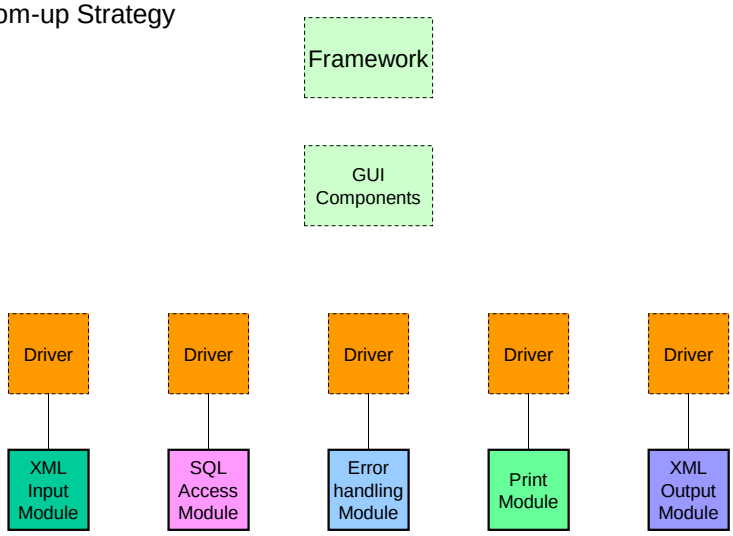
Top-down Strategy



81

Integration Tests – Buttom-up Strategy

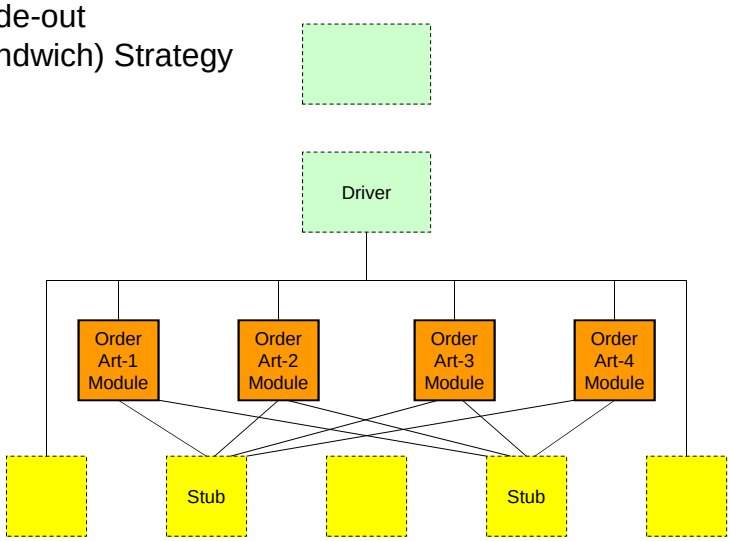
Buttom-up Strategy



82

Integration Tests – Inside-out Strategy

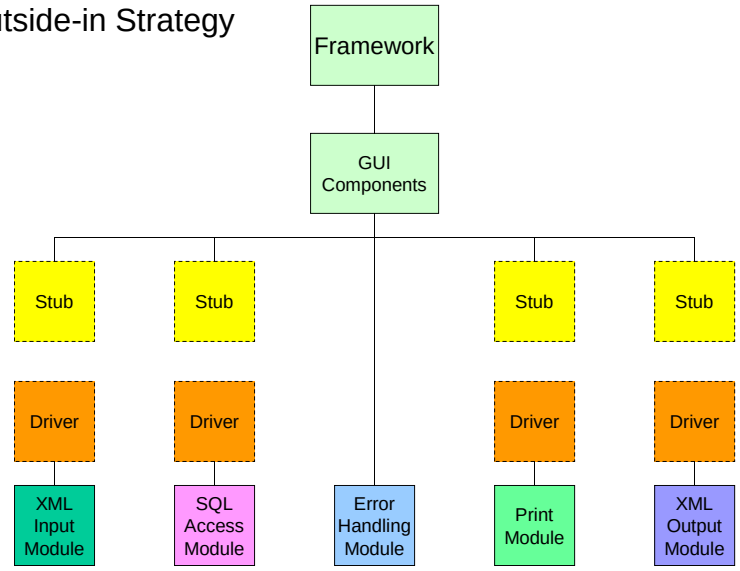
Inside-out
(Sandwich) Strategy



83

Integration Tests – Outside-in Strategy

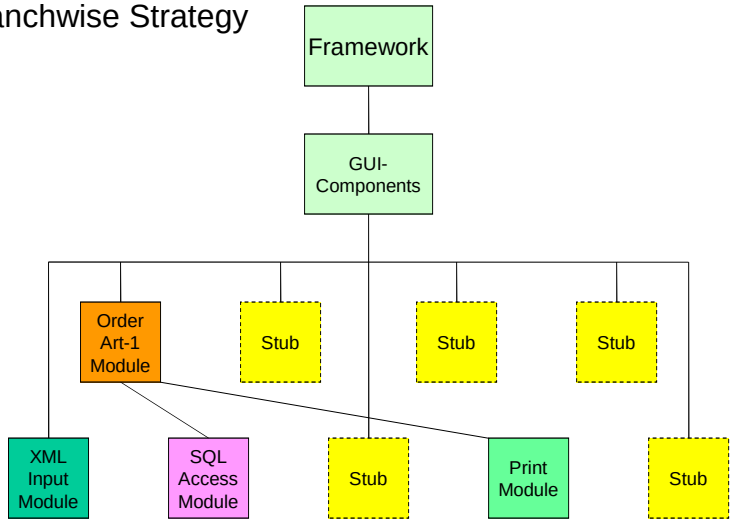
Outside-in Strategy



84

Integration Tests – Branchwise Strategy

Branchwise Strategy



85

Test Levels – System Test

Unit test | Integration test | System test | Acceptance test

- **System test focuses on the behaviour of the whole system or product in a live environment**
- Non-functional tests:
  - Installability – installation procedures
  - Interoperability – testing the software to check if it can inter-operate with other software component, software's or systems.
  - Maintainability – ability to introduce changes easily to the system
  - Load handling – behaviour of the system under significant load
  - Stress handling – behaviour of the system at or beyond the limits of its specified requirements
  - Portability – the ease with which a component or application can be moved from one environment to another
  - Recovery – recovery procedures on failure
  - Reliability – software's ability to function in given environmental conditions for a particular amount of time
  - Usability – ease with which users can engage with the system

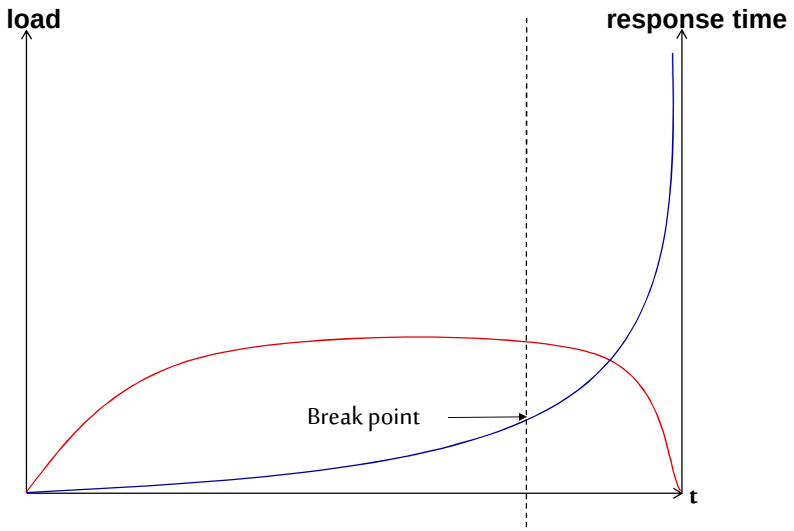
86

Test Levels – System Test

- Other non-functional tests:
  - Spike – type of load test. The object of this type of performance test is to verify a system's stability during bursts of concurrent user and or system activity to varying degrees of load over varying short time periods
  - Soak (endurance) – monitoring the system parameters under sustained use (e.g. memory leaks), with a typical production load, over a continuous availability period
  - Volume – what happens if large amounts of data are handled
  - Configuration – test the various software and hardware configurations
  - Compatibility – tests whether the application or the software product built is compatible with the hardware, operating system, database, other system or not.
  - Environment – test tolerances for heat, humidity, motion, portability, magnetic fields, etc.

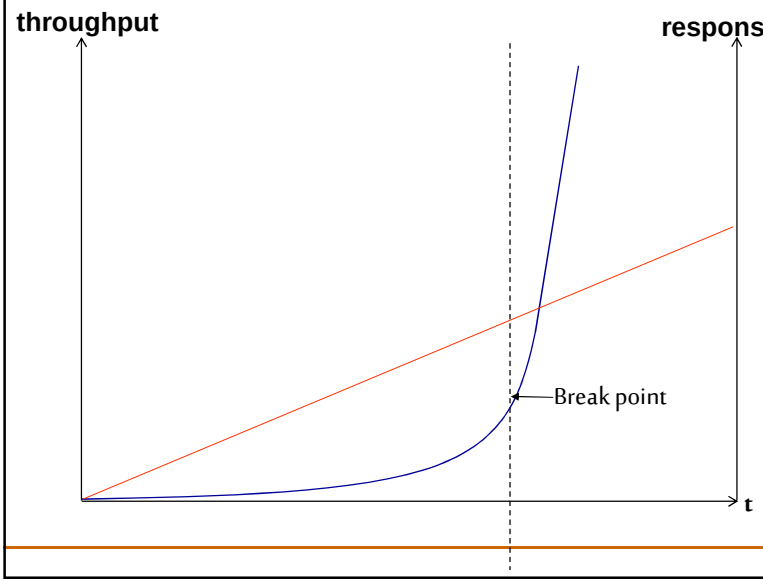
87

Load (Stability) Test



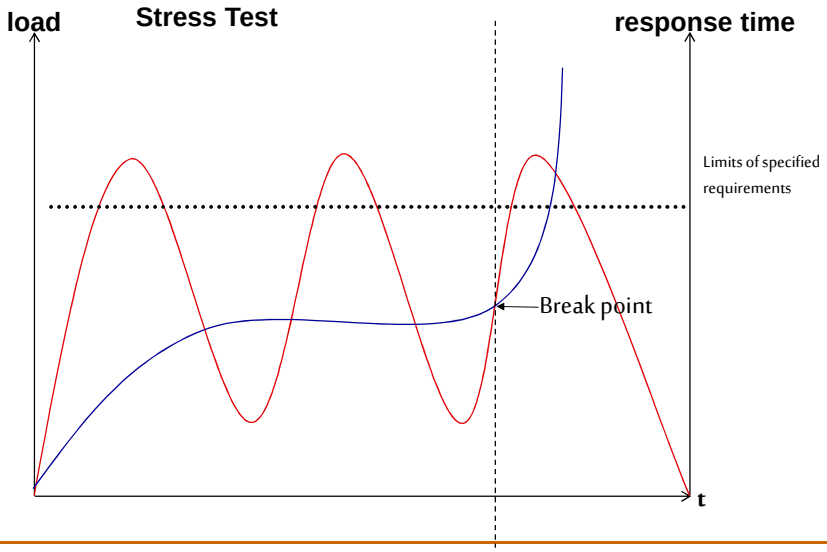
88

Scalability Test



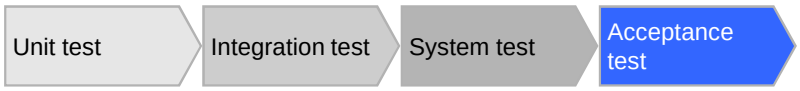
89

Stress Test



90

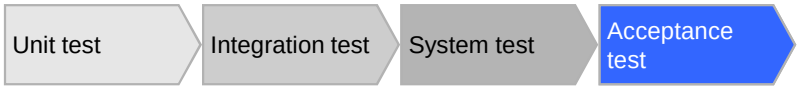
Test Levels – Acceptance Test



- **Provide the end users with confidence that the system will function according to their expectations**
- Acceptance testing will be carried out using the requirement specification as a basis for test
- Acceptance testing is often the responsibility of the customers or users of a system although other project team members may be involved as well

91

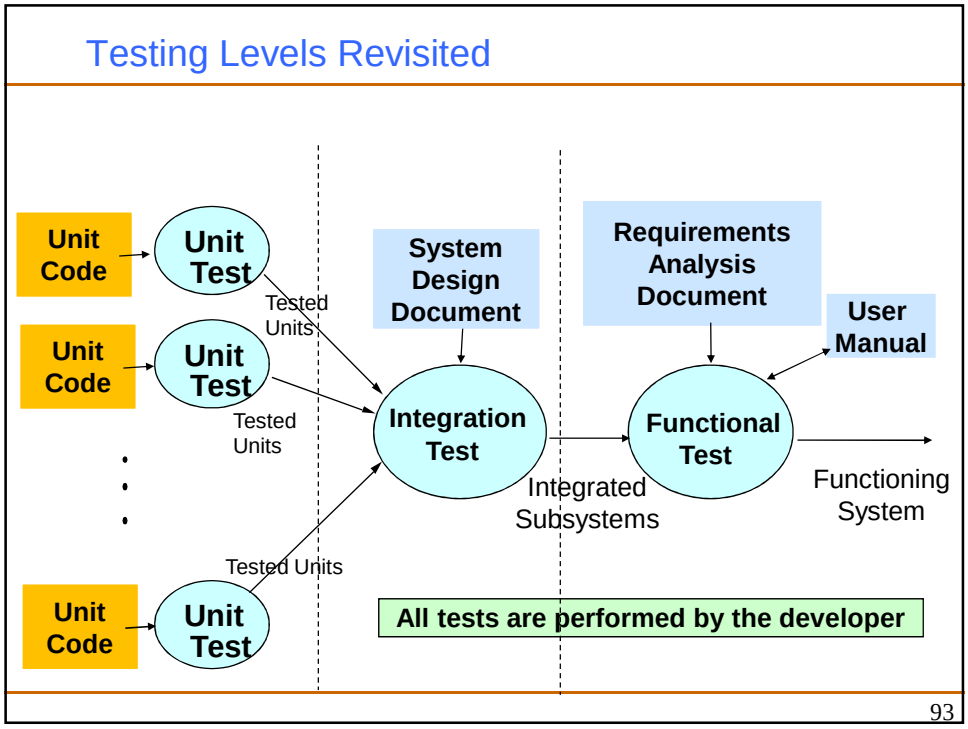
Test Levels – Acceptance Test

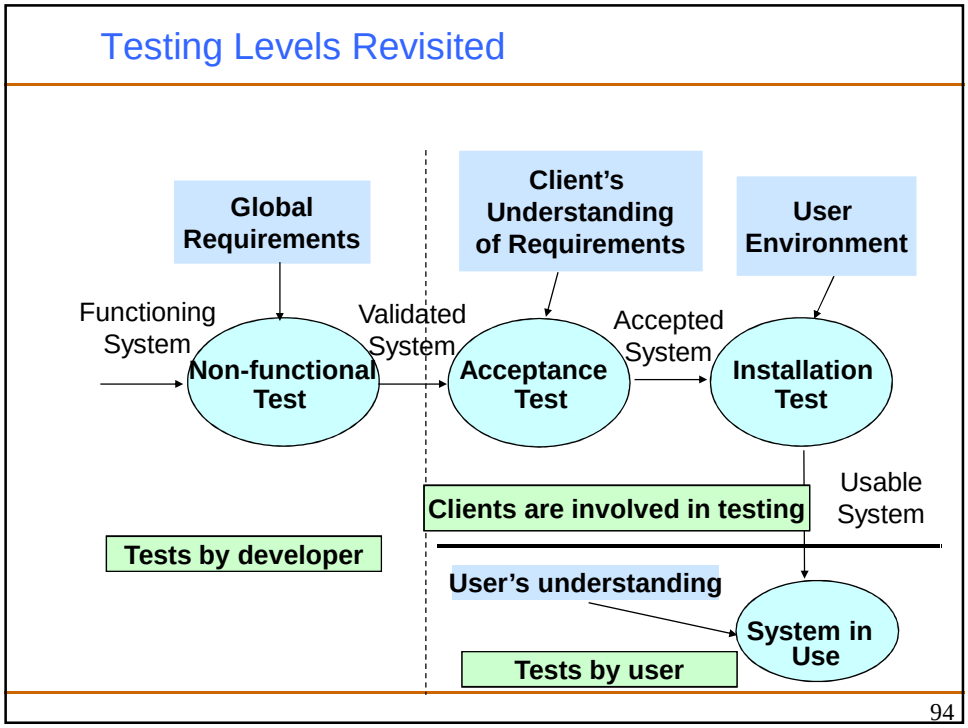


***Alpha testing**:* Testing by potential users/customers or an independent test team at the developer's site, but outside the development organization. Alpha testing is often employed for custom software as a form of **internal acceptance testing**.

***Beta testing**:* Testing by potential and/or existing users/customers at an external site not involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of **external acceptance testing** for off-the-shelf software in order to acquire feedback from the market.

92





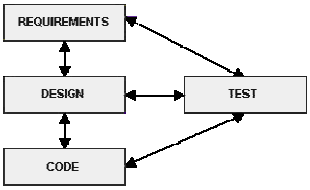
Maintenance Testing

- The system is eventually released into the live environment
- During this deployment it may become necessary to change the system
- (Planned) changes may be due to
  - **Additional features** being required (perfective modification)
  - The system being **migrated** to a new operating platform (adaptive)
  - New **faults being found** (corrective modification)
  - The system being **retired**
- **Testing which takes place on a system which is in operation in the live environment is called Maintenance Testing**
- An understanding of the parts of the system which could be affected by the changes could reduce the amount of regression testing
  - Impact analysis

95

Traceability

- Traceability Matrix (www.wikipedia.com)
  - Traceability refers to the ability to link requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases.
  - Traceability supports numerous software engineering activities such as change impact analysis, verification of code, regression test selection, and requirements validation.
  - In traceability, the relationship of driver to satisfier can be one-to-one, one-to-many, many-to-one, or many-to-many.



96

Traceability and Change

Horizontal Traceability: The relationship of the collections of components across collections of workproducts e.g. a design component is traced to the code components that implement that part of the design.

Vertical Traceability: The relationship among the parts of a single workproduct (discipline) e.g. requirements.

Impact analysis tries to assess the impact of changes on the rest of the system: when a certain component changes, which system parts will be affected, and how will they be affected?
Change impact analysis is defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change".

97

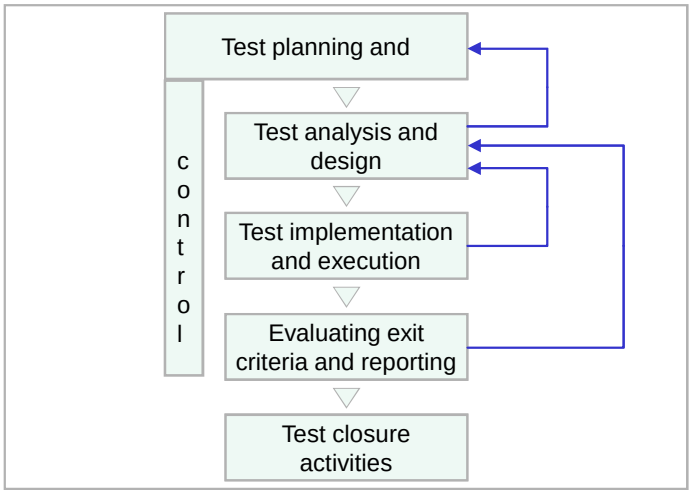
Maintenance Testing Revisited

- IEEE 1219 defines maintenance as the **modification** of a software product after delivery to correct faults, to improve performance or other attributes, or to **adapt the product to a modified environment**. Modification requests are
  - logged and tracked,
  - the impact of proposed changes determined,
  - code and other software artifacts are modified,
  - testing is conducted,
  - and a new version of the software product released.

98

Test Planning

Fundamental Test Process – Iteration of Activities



100

Fundamental Test Process

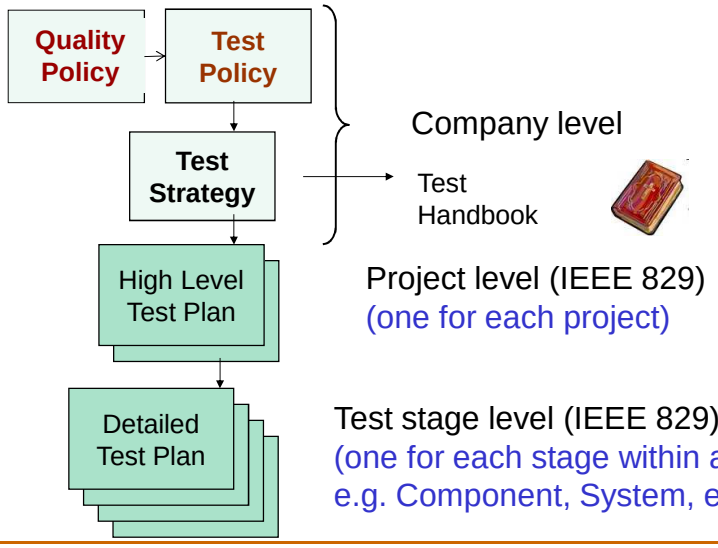
Test planning and control | Test analysis and design | Test implementation and execution | Evaluating exit criteria and reporting | Test closure activities

Test planning
- Defining the scope and objectives of testing and identifying risks
- Determining test approaches (techniques, test items, coverage, testware)
- Implementing the test policy and/or the test strategy (high level approach) (Document exceptions if exist, e.g. one test case design technique enough for this functional area because it is less critical)
- Determining the required test resources (people, test environment, PCs, tools)
- Scheduling test analysis and design tasks (WBS)
- Scheduling test implementation, execution and evaluation (WBS)
- Determining the exit criteria

Test control
- Monitoring, measuring and analyzing results
- Comparing expected and actual progress, test coverage and exit criteria
- Making corrections if things go wrong and deciding actions

101

Fundamental Test Process – Test Planning in Different Levels



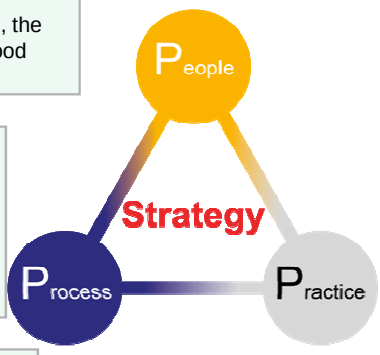
102

Terminology

Test policy describes the philosophy of organization towards testing. It is generally developed by the top level IT management. It usually contains the definition of testing, the process or procedure (or standard) need to follow for good quality product, metrics, improvement approaches.

A *test strategy* is an outline that describes the testing portion of the software development cycle. It is created to inform project managers, testers, and developers about some key issues of the testing process. In the test strategy is described how the product risks of the stakeholders are mitigated in the test levels and which test types are performed in the test levels.

Test approach (operational): The implementation of the test strategy for a specific project. It typically includes the decisions made based on the (test)project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed.



103

Test Strategy (Handbook)

- It is for all possible test levels. For each level
  - Entry, exit criteria
  - Test specification techniques (based on the given quality characteristics)
  - Degree of independence
  - Standards to be complied with
  - Test environment
  - Test approach toward test automation, reusability, regression testing, testing tools
  - Measures, metrics to be recorded, documentation
  - Incident management

104

Test Plan

- Test planning is the most important activity undertaken by a test leader
- It is used in development and implementation projects
- The main document produced in test planning is often called a **master test plan** or a **project test plan**
- The details of the test level activities are documented within test level plans
- The contents sections of a test plan for either the master test plan or test level plans are mainly identical
- There should be (a minimum of) 16 sections present in a test plan
- Test planning is a continual activity

105

Test Plan

- As the plans will have been **baselined** (locked down) after initial sign-off, the changes would normally be managed by the project change process
- Baselining a document effectively secures it from further change unless authorised via a change control process
- A useful revision aid to help remember the 16 sections of the IEEE 829:2008 test plan is the acronym 'SPACEDIRT'
  - S scope (test items, features to be tested and features not to be tested)
  - P people (including responsibilities, staff, training, approvals)
  - A approach (test levels and types, test strategy, tools, bug tracking)
  - C criteria (including item pass / fail criteria and suspension and resumption requirements)
  - E environment needs, estimates
  - D deliverables (tests, reports, test metrics)
  - I identifier and introduction (Name, version, date, overview, objectives)
  - R risks and their mitigation, references
  - T testing tasks and schedule

106

Entry Criteria

Entry criteria is the set of generic and specific conditions for permitting a process to go forward with a defined task, e.g. test phase.

- Determines at what point can be a particular test level or phase start
- Some typical entry criteria
  - Aquisition and supply
  - Test items
  - Money available
  - Risk

107

Exit Criteria

Exit criteria is the set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished.

- Determines completeness
- Can be defined for all of the testing activities
- Should be included in the relevant test plans
- Some typical exit criteria
  - All tests planned have been run
  - Cost - when the budget has been spent
  - Schedule has been achieved
  - Certain level of requirements coverage has been achieved
  - No high-priority or severe defects are left outstanding
  - All high-risk areas have been fully tested with only minor residual risks left outstanding
- Agreed as early as possible in the life cycle
- Often are subject to controlled changes
- A successful project is a balance of quality, budget, schedule and feature considerations

108

Test Approaches

- There are many approaches that can be employed
  - **Analytical approaches** use some formal or informal analytical technique, usually during the requirements and design stages of the project
    - Risk-based testing
    - Requirements based testing
  - **Model-based approaches** create or select some formal or informal model for critical system behaviors
    - Stochastic testing, e.g. exercising all transitions in a communication protocol model
  - **Standard-compliant approaches**, specified by industry-specific standards
  - **Process-compliant approaches**
    - Agile developments → TDD

109

Test Approaches

  - **Methodical approaches** adhere to a pre-planned, systematized approach that has been developed in-house, assembled from various concepts developed inhouse and gathered from outside
    - Fault-based, failure-based, check-list based and quality-characteristic-based
    - Fault-based example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs
  - **Dynamic** (like exploratory) and **heuristic** (like error guessing) **approaches**
  - **Consultative (directed) approaches** (ask users)
  - **Regression-averse** approaches
  - Etc.

110

Test Approaches – Key Factors

- Factors to be considered when defining the approach
  - Risk of failure of the project, hazards to the product and risks of product failure to humans, the environment and the company
  - Skills and experience of the people in the proposed techniques, tools and methods
  - The objective of the testing endeavour and the mission of the testing team
  - Regulatory aspects
  - The nature of the product and the business

- Excercise: Discuss which are preventive, and which are reactive approaches

111

Test Control

Test control is a test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned.

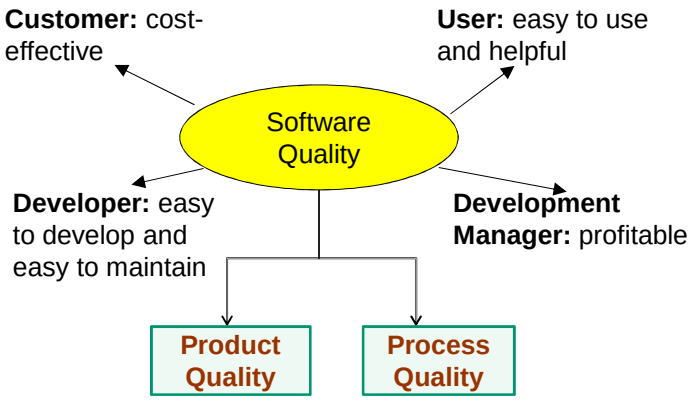
- Uses test metric information for decisions
- Particularly required when the planned test activities are behind schedule
- Test-control activities – **test leader's responsibility**
  - Reprioritizing tests
  - Changing the test schedule
  - Setting an entry criterion requiring fixes to be retested by a developer before accepting them into a build
  - Reviewing of product risks
  - Adjusting the scope of the testing
- Test-control activities – **project manager's responsibility**
  - Descoping of functionality
  - Delaying release
  - Counting testing after delivery into the production environment

112

Quality and Risk

What is Quality Software?

- Answer may depend on the stakeholder you ask:



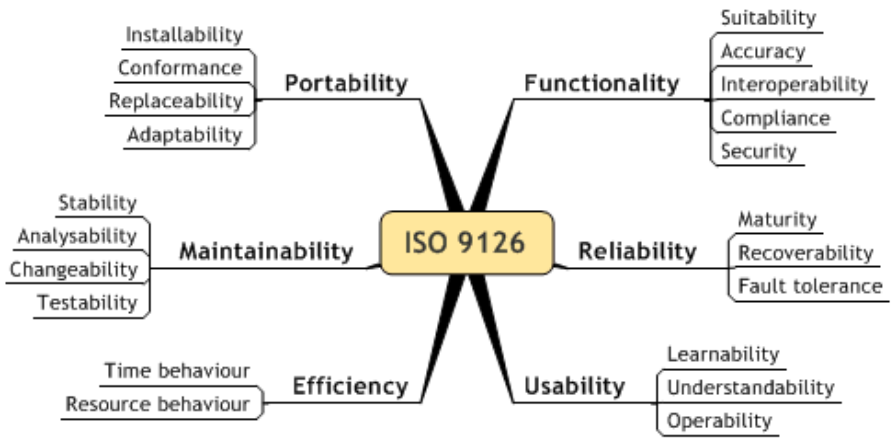
114

Software Process Quality

- Ensuring that each intermediate product resulting from the steps of development process is of good quality and satisfies the internal customers who have to perform the next step.
- Ensuring that the methodology, tools, and technologies employed for the process are under control and are improving.
- Several organizational frameworks have been proposed to **improve quality**
  - Plan-Do-Check-Act *(improving/optimizing a process)*
  - Quality Improvement Paradigm/Experience Factory *(building cont. improving)*
  - SEI Capability Maturity Model Integrated *(staged and continuous process improvement)*
  - *SPICE (Automotive, Bank, Healthcare)*

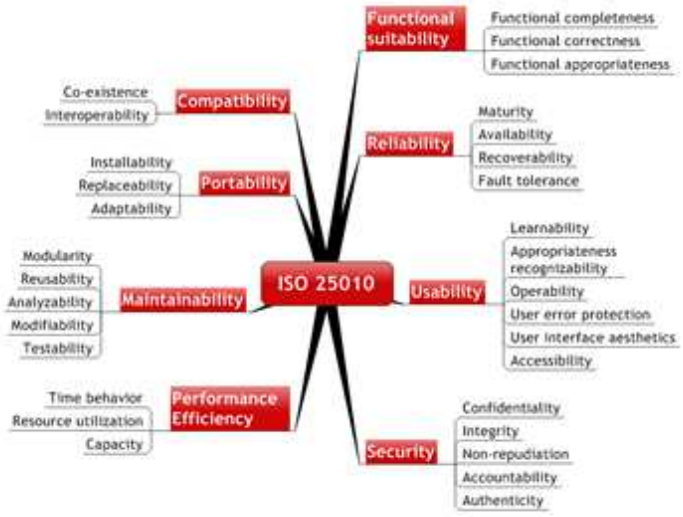
115

SW Product Quality ISO/IEC 9126



116

SW Product Quality ISO/IEC 25010:2011



117

Testing Quality Characteristics

- Exhaustive testing is not possible
- If we can't test everything, what can we do?
  - Managing and Reducing RISK
    - Risk is inherent in all software developement
    - **Uncertainties** become more significant as the system complexity grows and the implications of failure increase
  - Carry out a RISK **Analysis** of the application
    - **Prioritize tests** to focus on the main areas of risk
    - **Apportion time** relative to the degree of risk involved
    - **Understand the risk** to the business if the software is not functioning correctly

118

How much Testing?

- It depends on RISK
  - risk of important faults being there
  - risk of failure costs
  - risk of releasing untested or under-tested software
  - risk of losing credibility and market share
  - risk of missing a market window
  - risk of over-testing, ineffective testing

119

How much Testing?

- Testing time will always be limited
- Use RISK to determine:
  - what to test first
  - what to test most
  - how thoroughly to test each item
  - what not to test (this time)

} **i.e. where to place emphasis**

- Use RISK to
  - allocate the time available for testing by prioritizing testing ...

120

Risk and Testing

Risks

- **Project (planning) risks**
- **Product (quality) risks**

Risk is a factor that could result in future negative consequences, usually expressed as **likelihood** and **impact**.

- The calculation of the risk

Level of risk = **probability of the risk occuring** **x** **impact if it did happen.**

- Can be qualitative (e.g. low, medium, high) or quantitative (e.g. 25%)

121

Testing = Risk Management

Should be determined

- What are the project and product risks?
- What constraints affect testing?
- What is most critical?
- Which aspects of the product are testable?
- What should be the overall test execution schedule and how should we decide the order in which to run specific tests?

122

Risk Management (continued)

1. Risk Identification:
   - Determine which risks might affect the project and document their characteristics
   - Tools and techniques include: Documentation Reviews, Brainstorming, Interviewing Key Experts, Risk templates, Checklists, Experiences
   - **Output:** raw list of risks and associated symptoms or warning signs (that require further analyses)
2. Risk Analysis:
   - Assess the impact and likelihood of occurrence
   - Prioritize according to the potential effect on project objectives (e.g. greater impact leads to a higher priority)
   - Sample risk probability ratings: 20% = surprised if it happens; 50% = 50-50; 80% = surprised if it does NOT happen
   - The impact scale reflects the potential severity of the effect on the project objectives: 1 = low impact; 2 = medium impact; 3 = high impact; etc.
   - **Output:** list of prioritized risks based on *probability* x *impact*
   - (Possible weights: execution frequency, criticality)

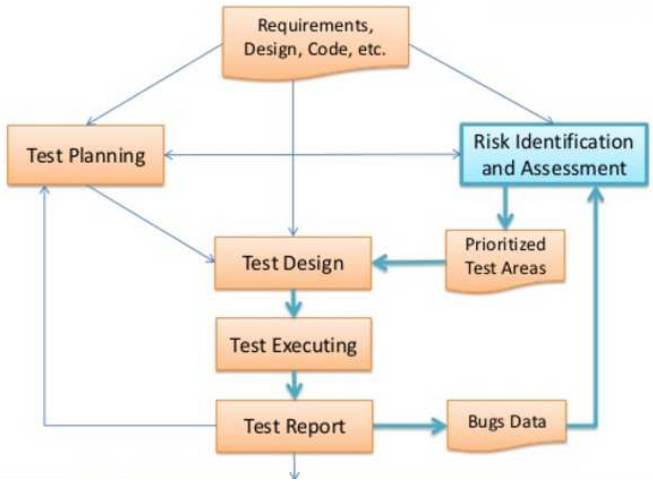
123

Risk Management (continued)

3. Risk Response - Contingency Plans:
   - Several risk response *strategies* are possible. Select the most effective strategy for each risk. The available strategies are:
     - **Avoidance: change the project plan to eliminate the risk or condition inducing risk**
     - **Transference: shift the consequences of the risk to another party (e.g. insurance)**
     - **Mitigation: reduce the probability and/or consequence of an adverse risk event to an acceptable threshold**
     - **Acceptance: a conscious decision not to change the project plan**
   - Develop **contingency plans** to reduce threats to the project objectives
   - Identify **individuals to take responsibility** for each critical risk response in the form of a contingency plan
   - These plans are **added to the project action list** – to be reviewed on a regular basis
   - Control risks by **continually** monitoring and correcting risky conditions: reviews, inspections, development of fall back positions, etc.

124

Risk Management (continued)



125

Risk Priority Table - Example

Example: ATM Machine
Functions: Withdraw cash, transfer money, read balance, make payment, buy train ticket.
Attributes: Security, ease of use, availability

Features & Attributes	Occurrence Likelihood	Failure Consequence	Priority (L x C)
Withdraw cash	High = 3	High = 3	9
Transfer money	Medium = 2	Medium = 2	4
Read balance	Low = 1	Low = 1	1
Make payment	Low = 1	High = 3	3
Buy train ticket	High = 3	Low = 1	3
Security	Medium = 2	High = 3	6

126

Ordered Risk Priority Table

Features & Attributes	Occurrence Likelihood	Failure Consequence	Priority (L x C)
Withdraw cash	High = 3	High = 3	9
Security	Medium = 2	High = 3	6
Transfer money	Medium = 2	Medium = 2	4
Make payment	Low = 1	High = 3	3
Buy train ticket	High = 3	Low 1	3
Read balance	Low = 1	Low = 1	1

Next step: Risk response planning

127

Product (Quality) Risk

- Potential failure areas in software are known as product risks, as they are **risks to the quality** of the product
- Product risks include
  - Error-prone software delivered
  - Poor requirements leading to badly defined and built software
  - A defect in the software / hardware could cause harm to an individual or company
  - Poor software quality characteristics leading to poor user feedback
  - The software does not meet the requirements and delivers functionality that was not requested
  - System forces user to spend more time than he feels appropriate
  - System crashes resulting in loss of business
  - Corrupts data
  - Delivers slow performance
  - Incorrect Documentation
- Risk are determined to decide where to start testing
- Product risks also provide indirect information on how much testing should be carried out
- Mitigating product risks may also involve non-test activities

128

Project (Planning) Risk

- The Software Project Manager is responsible for project risk management. Whilst managing the testing project a Test Leader will use project risks to manage the capability to deliver
- Project risks include
  - Supplier issues (contractual, third party failure, etc.)
  - Organizational factors (personal skills, lack of training, failure to follow up on information found in testing and reviews, problems that stop testers communicating their needs and test results, etc.)
  - Specialist issues (requirements, design, and code quality)
- Project risks should be documented in the IEEE 829 Test Plan

  IMPORTANT!
  - Proper risk management implies *control* over future events, and is *proactive* rather than reactive.
  - Risk register should be maintained by the Test Leader

129

Risk and Testing Summary

- Identify Risks as early in the project as possible
- There are different ways to manage risks
- Communication is the key element
- Focus on the important things first

Risks for testing:

1. Insufficient/not available/poor test basis
2. Agressive delivery
3. Lack of test expertise
4. Poor test management processes
5. Bad estimations/effort overrun/schedule delay
6. Problems with test environment
7. Poor test coverage, …

130

Test Estimations

Test Estimation

- **Individual estimation** – amalgamate the individual estimates when received
- **Group estimation** – discuss (agree and / or debate) the estimate in a meeting
- Either of the above approaches can be used individually or mixing and matching them as required

132

Test Estimation – Metrics Based

- Relies upon **data collected** from previous or similar projects
  - Number of test conditions
  - Number of test cases written
  - Number of test cases executed
  - Time taken to develop test cases
  - Time taken to run test cases
  - Number of defects found
  - Number of environment outages and how long on average each one lasted
- Possible to estimate quite accurately what the cost and time required for a similar project would be. Important that the actual costs and time for testing are accurately recorded

133

Test Estimation – Expert-Based

- Uses the **experience** of owners of the relevant tasks or experts to derive an estimate
- Experts
  - Business experts
  - Test process consultants
  - Developers
  - Technical architects
  - Analysts and designers
  - Anyone with knowledge of the application

134

Test Estimation – Affecting Factors

- Complexity
  - Difficulty of comprehending the problem the system is built for
  - Innovative technologies
  - Intricate or multiple test configurations
  - Geographical distribution of the team
- Product characteristics
  - Size of the test basis
  - Complexity of the final product
  - Amount of non-functional requirements
  - Security requirements (the security standard)
  - Amount of documentation required
  - Availability and quality of the test basis
- Development process characteristics
  - Timescales
  - Amount of budget available
  - Skills of those involved in the testing and development activity
  - Tools being used across the life cycle
- Expected outcome of testing
  - Amount of errors
  - Test Cases to be written

135

Psychology of Testing

Remember why Test

- build confidence ✓
- prove that the software is correct ✗
- demonstrate conformance to requirements ✓
- find faults ✓
- reduce costs ✓
- show system meets user needs ✓
- assess the software quality ✓

Testing Approach – First Step

- Show that the system
  - does what it should
  - doesn't do what it shouldn't

Goal:	**show working**
Success:	**system works**

Fastest achievement: easy test cases

→ **Result: faults left in**

138

Testing Approach – Second Step

- Show that the system
  - does what it shouldn't
  - doesn't do what it should

Goal:	**find faults**
Success:	**system fails**

Fastest achievement: difficult test cases

→ **Result: fewer faults left in**

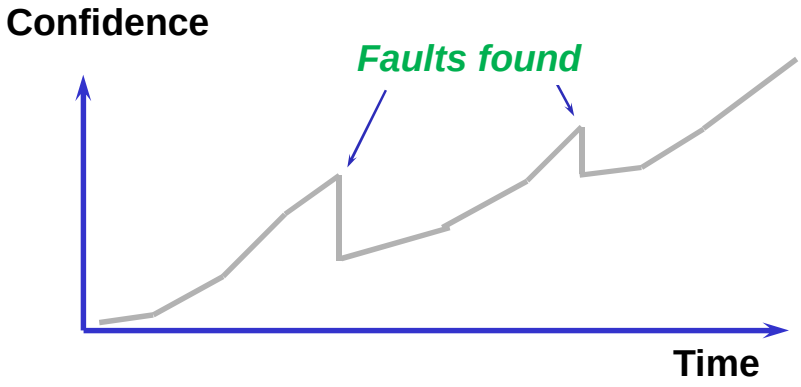
139

Testing Approach – Third Step

- **Reduce the perceived risk** of not working to an acceptable value.
- **Goals are**
  - to understand the quality of the software in terms of its defects,
  - to furnish the programmers with information about the software's deficiencies, and
  - to provide management with an evaluation of the negative impact on the organization if products are shipped to customers in its present state

140

Confidence



No faults found = confidence?

141

The Testing Paradox

Purpose of testing: to find faults

Finding faults destroys confidence

Purpose of testing: destroy confidence?

The best way to build confidence is to try to destroy it

142

Who Wants To Be a Tester?

- A destructive process
- Bring bad news
- Under worst time pressure (at the end)
- Need to take a different view, a different mindset
- (“What if it isn’t?”, “What could go wrong?”)
- How should fault information be communicated (to authors and managers?)

143

Tester’s Have the Right To

- accurate information about progress and changes
- insight from developers about areas of the software
- delivered code tested to an agreed standard
- be regarded as a professional (no abuse!)
- find faults!
- challenge specifications and test plans
- have reported faults taken seriously
- make predictions about future fault levels
- improve her own testing process

144

Testers Have Responsibility To

- follow the test plans, scripts etc. as documented
- report faults objectively and factually (no abuse!)
- check tests are correct before reporting s/w faults
- remember it is the software, not the programmer, that you are testing
- assess risk objectively
- prioritizing
- communicate the truth

145

The Psychology of Testing

- Very different people can be involved in software testing
  - Developers
  - Professional testers
  - Specialists
  - Users
- Developers want to prove their code does work
- Testers want to prove that the code does not work

146

The Psychology of Testing

developer

independent tester

Understands the system but will test "gently" and is **driven by delivery**

Must learn about the system but will attempt to break it and is **driven by quality**

147

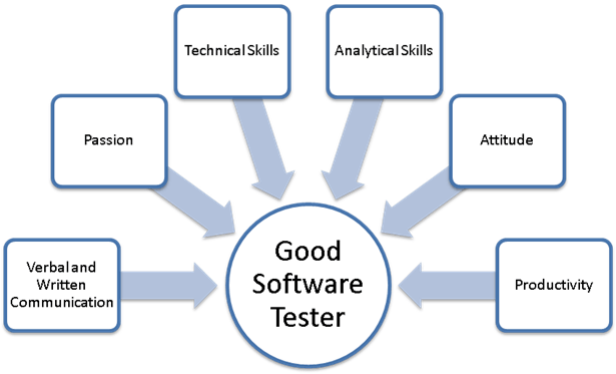
The Psychology of Testing – Defect Reporting

- Keep the focus on delivering a quality product
- Results should be presented in a non-personal way
- Attempt to understand how others feel
- At the end of discussions, confirm that you have both understood and been understood
- Be constructive!

148

The Psychology of Testing

- A strong skill mix is critical to a successful testing team
- Technical skills and domain knowledge alone will not make a tester successful. **Successful tester must be able to communicate.** Effective testers are effective communicators
- Assume everything you write will some day become public



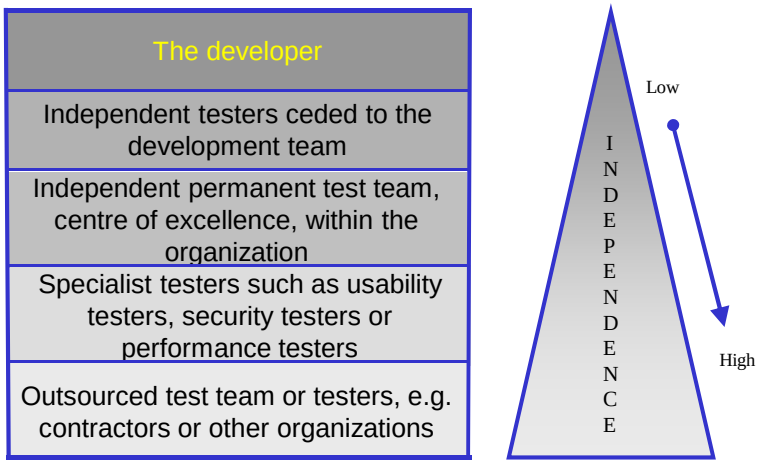
149

The Psychology of Testing

- Test independence – testing should be done by different people than the creator of the code
  - Those who wrote the software under test
  - Another member(s) of the same development team
  - Members from a different organizational group (e.g. an independent test team)
  - Members from a different organization or company (e.g. outsourcing to another company)

150

Levels of Independent Testing



151

Features of Indep. Testing

Benefits	Drawbacks
• The tester sees other and different defects than the author	• Isolation from the development team – The tester is totally dependent on the test basis
• The tester is unbiased	• The tester may be seen as the bottleneck – As independent test execution is normally the last stage
• The tester can see what has been built – Rather than what the developer thought had been built	• Developers lose a sense of responsibility for quality – As it may be assumed that they need not worry about errors
• The tester makes no assumptions regarding quality	• The fully independent view sets developers and testers on either side of an invisible fence

152

Independent Testing – Exercise

Which of the following demonstrates independence in testing?

(i) Independent testers may be external to the organization
(ii) Independent testers may be part of the development team
(iii) Independent testers may be from the user community
(iv) Programmers who wrote the code serve as independent testers
(v) Customer who wrote the requirements serve as independent testers

A) (i), (iii), and (v)
B) (i), (ii), (iii), and (v)
C) (ii), (iv), and (v)
D) (i), (iii), (iv), and (v)

Solution: Programmers cannot objectively test their own codes. Because choices C and D include statement iv, both choices can be ruled out as valid choices. Statement ii distinguishes choice A from B. Given that statement ii is valid, we can deduce that choice B is a superior choice.

153

Technical Skills Needed in Testing

- Test managers (test management, reporting, risk analysis)
- Test analyst (test case design)
- Test automation experts (programming test scripts)
- Test performance experts (creating test models)
- Database administrator or designer (preparing test data)
- User interface experts (usability testing)
- Test environment manager
- Test methodology experts
- Test tool experts
- Domain experts

154

Other Skills Needed

- Read and write documentation
  - Good vocabulary
  - Able to track changes (e.g. requirements)
- Prepare the tests
  - Understand the test plan
  - Read and write test scenarios
  - Read and write test cases
  - Read and write test procedures
- Able to perform the test process
- Able to report issues and bugs
- Able to review various documents, codes

155

Test Organization – Testing Roles

Testing roles
- Test Leader
- Tester

What is the difference between a testing role and a testing job?

Role	An activity or a series of activities given to a person to fulfill, e.g. the role of a test leader
Job	Effectively what an individual is employed to do, so one or many roles could make up a job e.g. a test leader could also be a tester

- Within a test project one person may have more than one role
- A job may contain many roles and tasks

156

Test Organization – Tasks of a Test Leader

- Typical test leader tasks may include
  - Responsible for the test strategy
  - Collaborates with the project management and with other participants
  - Takes notice of the organization's testing policies
  - Coordinates the testing activities with other project activities
  - Coordinates the design, specification, implementation, execution of tests
  - Monitors the test results and the exit criteria
  - Decides on the implementation of the test environment
  - Maintains the plans based on the test progress and test results

157

Test Organization – Tasks of a Test Leader

- Contd…
  - Determinates what should be automated, to what degree, and how, ensuring it is implemented as planned
  - Responsible for the test support tools and for the required training for these tools
  - Decides on the proper metrics for measuring test progress and evaluating the quality of the testing delivered and the product
  - Writes the test summary report based on the information gathered during testing

158

Test Organization – Tasks of a Tester

- Typical tester tasks may include
  - Reviewing and contributing the test plans
  - Analysing, reviewing and assesing user requirements, specifications and models for testability
  - Creating test specifications from the test basis
  - Setting up the test environment
  - Preparing and acquiring / copying / creating test data
  - Implementing tests on all test levels, executing and logging the tests, evaluating the results and documenting the deviations from expected results as defects
  - Using test administration and test monitoring tools
  - Automating tests
  - Where required running the tests
  - Reviewing tests developed by other testers

159

Code of Ethics

- Software testing enables individuals to learn confidential and privileged information.
- A code of ethics is necessary, among other reasons to ensure that the information is not put to inappropriate use.
- **PUBLIC -** Software testers shall act consistently with the public interest
- **CLIENT AND EMPLOYER -** Software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest
- **PRODUCT -** Software testers shall ensure that the deliverables they provide meet the highest professional standards possible

160

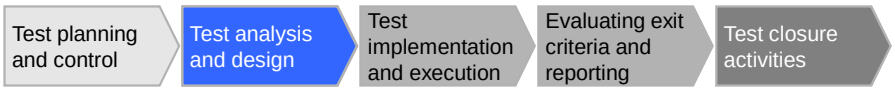
Code of Ethics

- **JUDGMENT -** Software testers shall maintain integrity and independence in their professional judgment
- **MANAGEMENT -** Software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing
- **PROFESSION -** Software testers shall advance the integrity and reputation of the profession consistent with the public interest
- **COLLEAGUES -** Software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers
- **SELF** - Software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession

161

Analysis and Design

Fundamental Test Process



Test analysis
- Reviewing the test basis (such as requirements, architecture, design, interfaces)
- Evaluating testability of the test basis and test objects
- Identifying and prioritizing test conditions based on analysis of test items, the specification, behavior and structure

Test design
- Test design involves predicting how the Software Under Test (SUT) should behave in a given set of circumstances
- Designing and prioritizing test scenarios and test cases
- Designing sets of tests, different test sets for different objectives
- Identifying necessary test data to support the test conditions and test cases
- Designing the test environment set-up and identifying any required infrastructure and tools

163

Test Design Techniques

Test Design Techniques

Test design categories

- **Specification-based (black box) techniques**
  - Equivalence partitioning (EP)
  - Boundary value analysis (BVA)
  - Decision table testing and Cause-Effect graphing
  - State transition testing
  - Orthogonal arrays and all-pairs tables
  - Use Case testing
- **Structure-based (white box, glass box) techniques**
  - Statement testing
  - Decision testing, branch testing
  - Condition / multiple condition / condition determination testing
  - Path testing, cyclomatic testing
  - LCSAJ testing
  - Loop testing
- **Experience-based techniques**
- **Defect-based techniques**

165

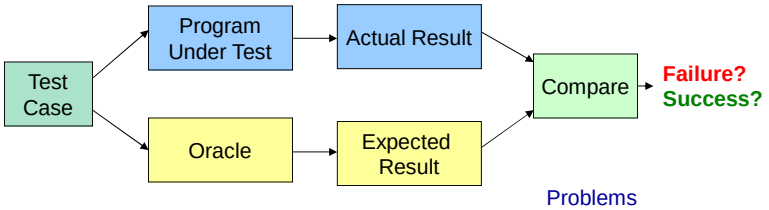
Specification or Structure-based?

- **Specification-based** vs. **Structure-based** testing
  - ***Specification-based testing***: Generating test cases based on the specification of the software
  - ***Structure-based testing***: Generating test cases based on the structure of the program
  - **Black box** testing and **white box** testing are synonyms for functional and structural (or glass box) testing, respectively.
    - In black box testing the internal structure of the program is hidden from the testing process
    - In white box testing internal structure of the program is taken into account
  - The expected result of a test can be computed via oracle.
  - **Oracle** states precisely what the outcome of a program execution will be for a particular test case. However, this may not always be possible

166

Test Oracle

- Oracle is a function that determines easily whether or not the results of executing a program under test conform with the program's specifications.



Problems
- Correctness of Oracle
- Correctness of specs
- Generation of Oracle

167

Spec. Based Techniques

Specification-based techniques

Techniques	Description
Equivalence partition	Grouping test conditions into partitions that will be handled the same way
Boundary value analysis	Defining and testing for the boundaries of the partitions
Decision table testing , cause-and-effect graphing	Defining and testing for combinations of conditions
State transition testing	Identifying all the valid states and transactions that must be tested
All pairs / orthogonal array testing	Determining the combinations of configurations to be tested
Use case testing	Determining usage scenarios and testing accordingly

169

Equivalence Partitioning

- **Equivalence class testing (equivalence partitioning)** is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage
- Partition the input domain to ***equivalence classes***
- Example (factorial):
  - If the input integer value *n* is less than 0 then an appropriate error message must be printed. If $0 \leq n < 20$, then the exact value of *n*! must be printed. If $20 \leq n \leq 200$, then an approximate value of *n*! must be printed in floating point format using some approximate numerical method. The admissible error may be 0.1% of the exact value. Finally, if $n > 200$, the input can be rejected by printing an appropriate error message.
- Possible equivalence classes: $D_1 = \{n<0\}$, $D_2 = \{0 \leq n < 20\}$, $D_3 = \{20 \leq n \leq 200\}$, $D_4 = \{n > 200\}$
- **Choose at least one test case per equivalence class to test**

170

Equivalence Partitioning

- Equivalence partitioning reduces the number of needed test cases
- The inputs of a program can be 'chunked' into groups of similar inputs
  - Grouping based on the properties of the possible inputs
- Example: a program accepts any integer values between -10.000 and +10.000 (inclusive)
  - Valid positive values ($0 < x \leq 10.000$)
  - Valid negative values ($-10.000 \leq x < 0$)
  - Valid zero value ($x = 0$)
  - Invalid positive values ($x > 10.000$)
  - Invalid negative values ($x < -10.000$)
  - Invalid non-integer real numbers (e.g.: 2,82)
  - Invalid character values (e.g.: „p")

Valid Equivalance Partitions

Non-valid Equivalance Partitions

171

Equivalence Partitioning

- For *unordered sets* select two values
  - 1 in, 0 not in
- For *equality* select 2 values
  - 1 equal, 0 not equal
- For *sets, lists* select two cases
  - 1 not empty, 0 empty

- Example: We have a bank account program which offers variable interest rates
  - 0.5% for the first 1,000 $ credit;
  - 1% for the next 1,000 $ credit;
  - 1,5% for the rest

  What would be the appropriate test cases?

172

Equivalence Partitioning

Example: A mail-order company selling flower seeds charges $3.95 for **postage and packing** on all orders up to $20 value and $4.95 for orders above $20 value and up to $40 value. For orders above $40 value there are no charge for postage and packing.
If you were using equivalence partitioning to prepare test cases for the postage and packing charges what valid input partitions would you define? What about non-valid partitions?

Example: Consider that **wordCount** method takes a word w and a filename f as input and returns the number of occurrences of w in the text contained in the file named f. An exception is raised if there is no file with name f.

173

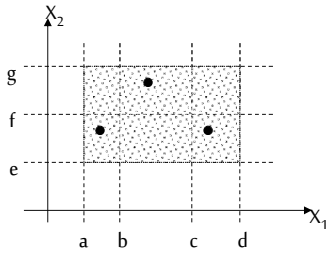
Equivalence Partitioning

- In some cases we have different equivalence classes for the same input domain. In these cases we can try to minimize the number of test cases while choosing representatives from different equivalence classes.
- Example: D_1 = {x is even}, D_2 = {x is odd}, D_3 = {x ≤ 0}, D_4={x > 0}
  - Test set {x=48, x= –23} covers all the equivalence classes
- On one extreme we can make each equivalence class have only one element which turns into exhaustive testing
- The other extreme is choosing the whole input domain D as an equivalence class which would mean that we will use only one test case

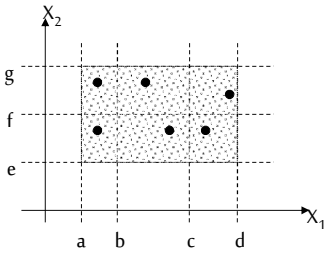
174

Generalized Equivalence Partitioning

- Weak normal EP: using at least one test case from each valid equivalent class



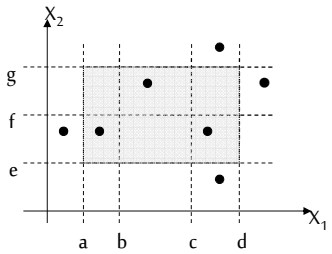
- Strong normal EP: Cartesian product of the weak normal equivalence classes



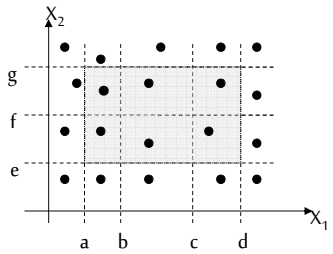
175

Generalized Equivalence Partitioning

- Weak robust EP: (at least) one value from each valid class with some invalid values



- Strong robust EP: Cartesian product of the weak robust equivalence classes



176

EP Advices & Applicability

- In order to mask errors in another field test **one invalid value at a time** to verify the system detects it correctly.
- In some cases use equivalence classes to **examine the outputs** rather than the inputs.
  - Divide the outputs into equivalence classes, then determine what input values would cause those outputs. This has the advantage of guiding the tester to examine, and thus test, every different kind of output. But be careful, this approach can be deceiving!!
- Equivalence class testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

177

Boundary Value Testing (BVA)

- Mistakes made by programmers cause faults that tend to cluster around boundaries. **Boundary value testing focuses on the boundaries**
- Faults usually appear when values just outside the range are incorrectly accepted or values just inside the range are incorrectly rejected
- In an **ordered set** for each range [R_1, R_2] listed in either the input or output specifications, we may choose five cases:
  - Values less than R_1
  - Values equal to R_1
  - Values greater than R_1 but less than R_2
  - Values equal to R_2
  - Values greater than R_2

- IMPORTANT! Since we have to deal with ordered sets we need to clarify first the the domain of the ordering, i.e. the **accurracy** (number of decimal digits, etc.)

178

Boundary Value Testing

- The maximum and minimum values of a partition are its boundary values
  - The boundary value of a valid partition is a **valid boundary value**
  - The boundary value of an invalid partition is an **invalid boundary value**
- Example: The partition is [-100,+100] (inclusive) with integers. Then
  - -101 is a maximum invalid boundary value of the invalid partition]-∞,-101]
  - -100 is a minimum valid boundary value of the valid partition [-100,+100]
  - +100 is a maximum valid boundary value of the valid partition [-100,+100]
  - +101 is a minimum invalid boundary value of the invalid partition [+101,+∞[
- To apply boundary value analysis, we will take the minimum and maximum (boundary) values from the valid partition (-100 and 100 in this case) together with the first or last value respectively in each of the invalid partitions **adjacent to the valid partition** (-101 and 101)
- In some cases we choose the **boundary value and both of its neighbours**.
- The chosen method is driven by the **risk of the application**.

179

Boundary Value Testing

- For the factorial example, ranges for variable *n* are:
  - [–∞, 0], [0,20], [20,200], [200, ∞]
  - A possible boundary value test set for the range [0,20] is
    - {n = -1, n=0, n=20, n= 21} **(2 values BVA testing)**
  - Another possibility could be
    - {n = -1, n=0, n=1, n=19, n=20, n= 21} **(3 values BVA testing)**
- Example: An exam has a pass boundary at 40 per cent, merit at 60 per cent and distinction at 80 per cent. What would be the boundary values for this exam?

180

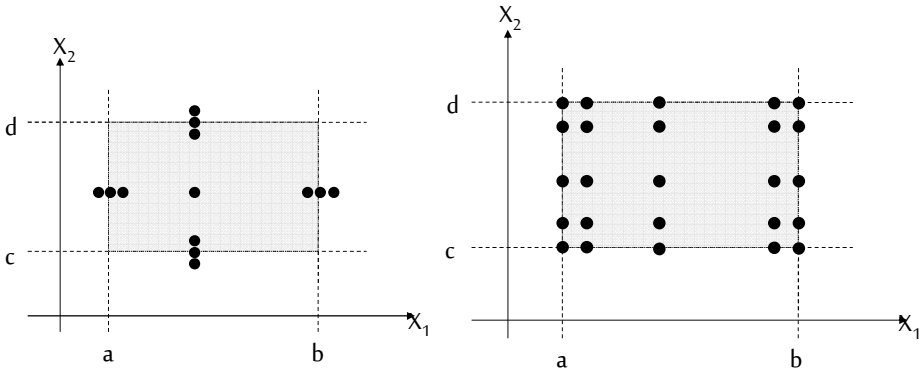
EP and BVA together

- If you do boundaries only, you have covered all the partitions as well
  - technically correct and may be OK if everything works correctly!
  - if the test fails, is the whole partition wrong, or is a boundary in the wrong place you have to test mid-partition anyway
  - testing only extremes may not give confidence for typical use scenarios (especially for users)
  - boundaries may be harder (more costly) to set up
- The value zero is special. We should always test it if it is possible or reasonable.

181

EP-BVA – in higher dimensions

- Robustness test (for independent variables)
- Worst-case testing (for independent variables)



182

Example – Book Ordering

Country values

- Parameters
  - Valid Country
    - Hungary
    - Other European Countries
    - Rest of the world
  - Invalid Country
    - Empty field
    - „All other values” (perhaphs not possible)

Grouping is required because of postage price determination

183

Example – Book Ordering

Form of Payment

- EP Parameters
  - Valid Payment
    - Visa, MasterCard
    - Bank transfer (invoice needed)
    - Cach on delivery
  - Invalid Payment
    - Emty field
    - Other value??

ACCOUNTS, I HAVE AN OVERDUE PAYMENT NOTICE FOR YOU.

184

Example – Book Ordering

Surname field

- EP Parameters
  - Valid Surname
    - 1 – 20 charecters
    - a – z, A – Z
    - áéíöőíüű, ÁÉÍÖŐÍÜŰ, ß, etc.
    - Van de Hut, van der Heide, O'Connor, etc.
  - Invalid Surname
    - Empty field
    - > 20 charecters
    - Special characters, @&+! etc.

Surname field

- BVA Parameters
  - Valid boundary values
    - 1 character a – Ű
    - 20 characters, a – Ű
  - Invalid boundary values
    - 0 character
    - 21 characters

185

EP – BVA Open Boundaries Guide

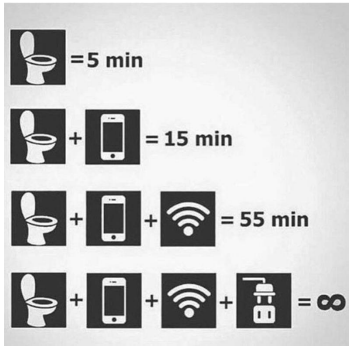
- In some cases, there are no boundaries defined for a value
  - Ages for humans
  - Size of an input field
- Then the following guidelines are suggested:
  - **Discuss requirement** or design specification in order to check whether there really is no boundary – sometimes there are boundaries even though they are not written down
  - **Set a fictional boundary** which is beyond a realistic maximum value (e.g. a human can live to 120 years)
  - **Research technical limitations** – for example, the size of an input field or database
  - **Look for boundaries in the rest of the system**, or systems you communicate with

186

Example – Book Ordering

Amount

- EP Parameters
  - Valid values
    - 1 – 99
  - Invalid values
    - 1 > Integer > 99
    - Empty field
    - Non-integers between 1 and 99
    - Letters, special characters, non-numerics
- Boundary Value Parameters
  - Valid
    - 1, 99
  - Invalid
    - 0, 100



187

Decision Tables

- Decision tables are used to record **complex business rules** that a system must implement. In addition, they can serve as a guide to creating test cases
- Apply it for system requirements that contain **logical conditions** (business rules). Hence, decision tables testing connects **combinations of conditions** with the **actions** that should occur

A single business rule

Conditions	1	2	3	4	5	6	7	8
Condition 1	Y	Y	Y	Y	N	N	N	N
Condition 2	Y	Y	N	N	Y	Y	N	N
Condition 3	Y	N	Y	N	Y	N	Y	N
Actions								
Action 1	Y	Y	N	N	Y	N	N	N
Action 2	Y	N	Y	Y	N	Y	N	N

188

Decision Tables

- Each business rule says, in essence
  - "Under this particular combination of conditions carry out this particular combination of actions„
- The **coverage criterion** for decision tables is expressed by rule
  - One test per column in the decision table have to be derived
- The **number of columns** (business rules) in a decision table is equal to 2^n
  - For n = the number of conditions
  - Applied when conditions are strictly boolean **True or False**
- Not all columns in a decision table are actually needed
  - We can sometimes **collapse the decision table**, combining columns, to achieve a more concise decision table
  - Performed when the value of one or more particular conditions can't affect the actions for two or more combinations of conditions

189

Decision Tables

- To **combine columns** we should look for two or more columns that result in the same combination of actions

Conditions	1	2	3	4	5	6	7	8
Condition A	Y	Y	Y	Y	N	N	N	N
Condition B	Y	Y	N	N	Y	Y	N	N
Condition C	Y	N	Y	N	Y	N	Y	N
Actions								
Action A	Y	Y	Y	N	N	N	N	N
Action B	Y	N	Y	Y	Y	Y	N	N
Action C	Y	Y	N	Y	Y	N	N	N

- In these columns – some of the conditions will be the same, and some will be different – the different ones don't seem to affect the outcome
- Insignificant values can be replaced with "–" (dash) or „* ” (dontcare)

190

Decision Tables

- Collapse columns:

Conditions	1	2	3	4	5	6	7
Condition A	Y	Y	Y	Y	N	N	N
Condition B	Y	Y	N	N	Y	Y	N
Condition C	Y	N	Y	N	Y	N	–
Actions							
Action A	Y	Y	Y	N	N	N	N
Action B	Y	N	Y	Y	Y	Y	N
Action C	Y	Y	N	Y	Y	N	N

- Note: not everything can be collapsed

191

Example: Student Access

A university computer system allows students an access for disc space depending on their project work status. Students without projects should use their allotted space, while student with university projects may use unlimited disc space. This is assuming they have logged on with a valid username and password.

What are the input conditions and actions?

Input conditions:

- Valid username (authenticate)
- Valid password (authorize)
- Work on project (access control)

Actions:

- Login accepted
- Restricted access

192

Determine input combinations

- Add columns to the table for each unique combination of input conditions.
- Each entry in the table may be either ‘T’ for true, ‘F’ for false.

Input Conditions								
Valid username	T	T	T	T	F	F	F	F
Valid password	T	T	F	F	T	T	F	F
Work on project	T	F	T	F	T	F	T	F

193

Rationalize input combinations

- Some combinations may be impossible or not of interest
- Some combinations may be ‘equivalent’
- Use “don’t care”

Input Conditions				
Valid username	F	T	T	T
Valid password	*	F	T	T
Work on project	*	*	F	T

194

Complete the table

- Determine the expected actions for each combination of input conditions
- Each column is at least one test case

Input Conditions				
Valid username	F	T	T	T
Valid password	*	F	T	T
Work on project	*	*	F	T
Actions				
Login accepted	F	F	T	T
Restricted access	*	*	T	F
Tags	**A**	**B**	**C**	**D**

195

Design Test Cases

- Usually one test case for each column but can be none or several

Test	Description	Expected Outcome	Tag
1	Username Booby	Invalid username	A
2	Username thisusernametoolong	Invalid username	A
3	Username R2D2 Password DarthVader	Invalid password	B
4	Valid user, no disc space	Restricted access	C
5	Valid user with disc space	Unrestricted access	D

196

Decision Table Testing - Remarks

- Always check for **redundancy** (same columns) and **inconsistency** (action sets are different for the same condition sets)
- A decision table is **non-deterministic** if there is no way to decide which rule to apply
- In general, entries can be more than just 'True' or 'False'
- There is a technique to generate decision tables automatically from the requirements (cause-effect graphing)
- If outputs or effects are mutually exclusive, i.e. T occurs in only one place in each column, we can combine them:

X	T	F	F
Y	F	T	F
Z	F	F	T

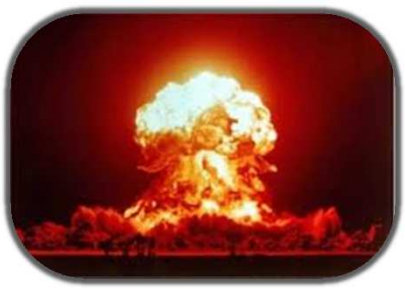
is equivalent to:

Action	X	Y	Z

197

Avoiding Combinatorial Explosions

- Combinatorial explosions can be avoided:
  - Identify the possible combinations
  - Use risk to weight those combinations
  - Test only the important combinations
- Other techniques are also applicable:
  - Classification trees
  - Pairwise testing



198

Excercise

1. Analyse your daily activities in week-days, holidays and traning-days. Your actions can be staying home, going to work or going to picnic.
2. Analyze the following decision table for the triangle problem. How many test cases do this imply?

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	x	x	x								
a2: Scalene											x
a3: Isosceles						x			x	x	
a4: Equilateral				x							
a5: Impossible					x	x		x			

199

State Transition Testing

- Useful technique when **actions are triggered** by changes of the input conditions, or changes of ‘state’
- The basis of the test is the system’s ***State Transition Diagram***

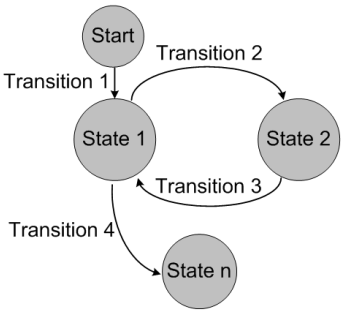
State Transition Diagram consisting of nodes to represent states and directed line segments to represent transitions between the states. One or more actions (outputs) may be associated with each transition. The diagram may represent a labelling transition system or a finite state machine.

- Testers are able to analyze the behaviour in terms of what happens when a transition occurs
  - Transitions caused by events
  - An event can be anything that acts as a trigger for a change
  - An event can produce an output and / or can change the system’s internal state
- All possible events and states recorded in a state table
  - Shows the relationships between states and inputs
  - Highlights possible but invalid transitions

200

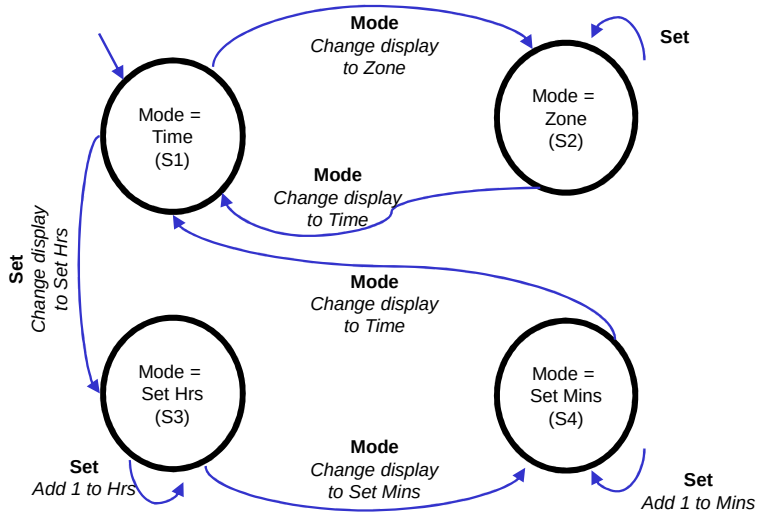
State Transition Testing

- Tests can be designed to
  - Cover typical sequence of states
  - Cover every state
  - Exercise specific sequences of transitions
  - Test invalid transitions
- State transition testing is useful
  - To test embedded softwares
  - To test technical automations
  - For modelling business objects having states
  - For testing screen-dialogue flows
  - For testing internet based applications



201

State Transition Diagram – Example (Clock)



202

State Transition Testing

Test cases containing one transaction

Test case	1	2	3	4	5	6	7	8
Initial state	S1	S1	S2	S2	S3	S3	S4	S4
Input	Mode	Set	Mode	Set	Mode	Set	Mode	Set
Expected output	Z	H	T	Z	M	H+	T	M+
Final state	S2	S3	S1	S2	S4	S3	S1	S4

203

State Transition Testing

Test cases containing two transactions

Test case	1	2	3	4	5	6	7	8	…	16
Initial state	S1	S1	S1	S1	S2	S2	S2	S2	…	S4
Input	Mode	Mode	Set	Set	Mode	Mode	Set	Set	…	Set
Expected output	Z	Z	H	H	T	T	Z	Z	…	M+
Next state	S2	S2	S3	S3	S1	S1	S2	S2	…	S4
Input	Mode	Set	Mode	Set	Mode	Set	Mode	Set	…	Set
Expected output	T	Z	M	H+	Z	H	T	Z	…	M+
Final state	S1	S2	S4	S3	S2	S3	S1	S1	…	S4

204

State Transition Testing

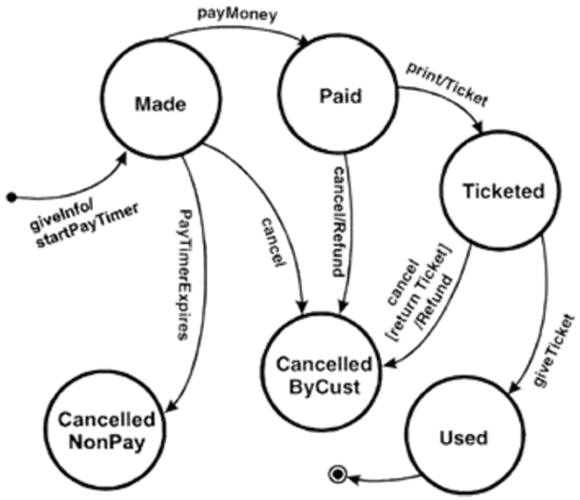
- **Finite state machines** are widely used in modeling of all kinds of systems. Generation of tests from FSM specifications assists in testing the conformance of implementations to the corresponding FSM model.
- Most system/subsystem can be modeled as a finite state machine (FSM), e.g. elevator designs, automobile components (locks, transmission, stepper motors, etc), nuclear plant protection systems, steam boiler control, etc.
- Note that FSMs are a part of UML 2.0 design notation
- Their most common representation is:

State transition table

Input \ Current state	State A	State B	State C
Input X	…	…	…
Input Y	…	State C	…
Input Z	…	…	…

205

Example – Reservation System



206

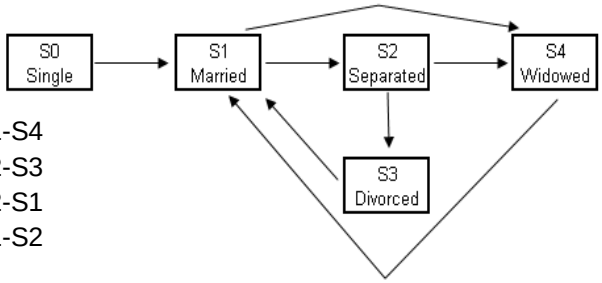
Example – Creating Test Cases

1. **Create a set of test cases such that all states are "visited" at least once under test**. The set of three test cases shown below meets this requirement. Generally this is a weak level of test coverage.
2. **Create a set of test cases such that all events are triggered at least once under test.** Note that the test cases that cover each event can be the same as those that cover each state. Again, this is a weak level of coverage.
3. **Create a set of test cases such that all paths are executed at least once under test.** While this level is the most preferred because of its level of coverage, it may not be feasible: there can be loops in the system. Testing of loops can be important if they may result in accumulating computational errors or resource loss (locks without corresponding releases, memory leaks, etc.).
4. **Create a set of test cases such that all transitions are exercised at least once under test.** This level of testing provides a good level of coverage without generating large numbers of tests. This level is generally the one recommended.

207

State Transition Testing – Exercise

Which test suite will check for an invalid transition of marriage status sequence?



a) S0-S1-S2-S3-S1-S4
b) S0-S1-S4-S1-S2-S3
c) S0-S1-S3-S1-S2-S1
d) S0-S1-S2-S3-S1-S2

Solution

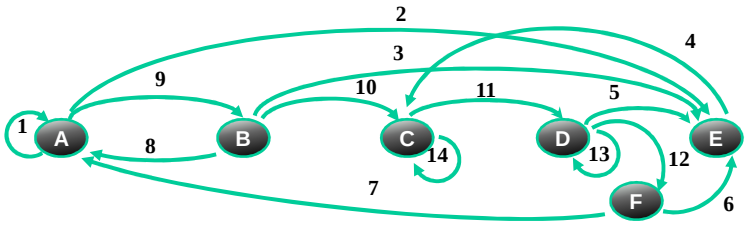
Simple review each potential answer to determine if the states and transitions are valid based on the state transition diagram. Take note that the question above was looking for invalid transitions – there were two of such kind choice C (S1 to S3 skipped S2, as S2 to S1). The diagram states that S2 cannot transition to S1, but in reality that transition can be made.

208

State Transition Testing

N-switch (Chow) testing is a form of state transition testing in which test cases are designed to execute **all valid sequences** of N+1 transitions.

- Switch Coverage is a technique for generating sequences of transitions
  - State labels are replaced in the diagram with letters and the transition labels with numbers
  - A state/transition pair can be specified in a table as a letter followed by a number

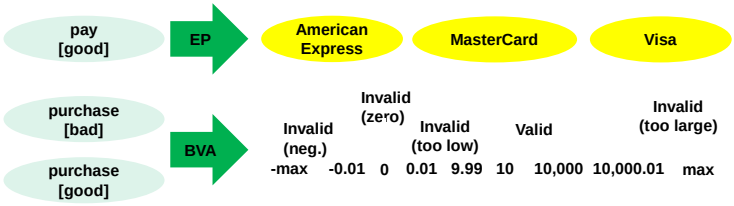


209

State Transition Testing

0-switch			1-switch								
A1	A2	A9	A1A1	A1A2	A1A9				A9 B10	A9B8	A9B3
B10	B8	B3	B10 C14	B10 C11	B10 C4	B8A1	B8A2	B8A9			
C14	C11	C4	C14 C14	C14 C11	C14 C4	C11 D13	C11 D12	C11 D5			
D13	D12	D5	D13 D13	D13 D12	D13 D5	D12 F6	D12 F7				
F6	F7					F7A1	F7A2	F7A9			

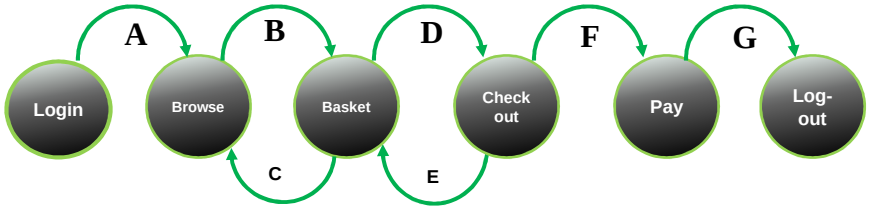
- State-based testing can be well combined with equivalence partitioning and boundary value analysis



210

State Transition Testing – Exercise

- Given the following state transition diagram which of the following series of state transitions contains an INVALID transition which may indicate a fault in the system design?



A. Login Browse Basket Checkout Basket Checkout Pay Logout

B. Login Browse Basket Checkout Pay Logout

C. Login Browse Basket Checkout Basket Logout

D. Login Browse Basket Browse Basket Checkout Pay Logout

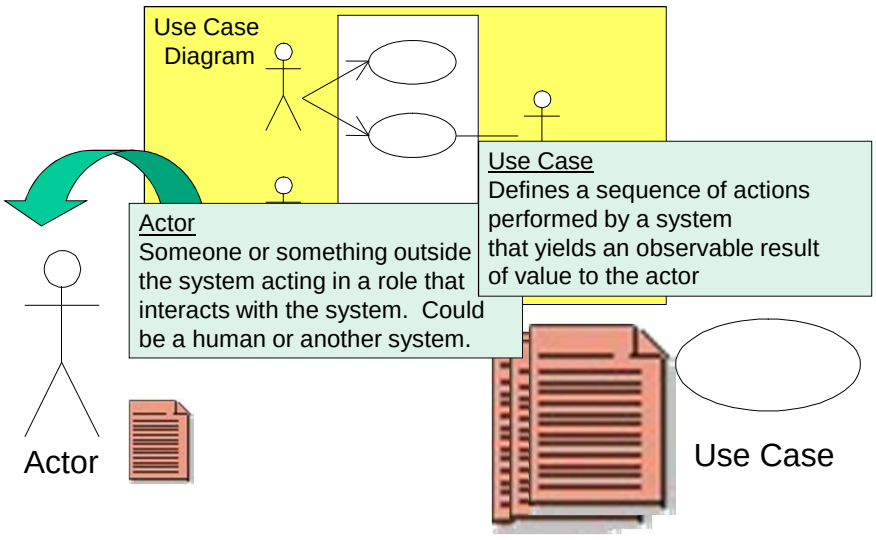
211

Use Case Testing

- **Use cases** are descriptions of interactions between users (actors) and the system in a high level view of the requirements
- The main advantage is that we **exercise real user processes or business scenarios**
  - Uncovering **defects in the process flow**
  - Helping to uncover **integration defects** caused by the interactions of different components
- The main principles applied elsewhere can be applied, also here
  - First test the highest priority Use Cases by taking typical examples
  - Then exercise some attempts at incorrect process flows
  - Followed by exercising the boundaries
- Each Use Case has
  - **Preconditions** which need to be met for a use case to start successfully
  - **Post-conditions** which are the results or the final state of the system after the Use Case scenario has been completed
- Use Case testing is very useful for **designing acceptance tests** with customer participation

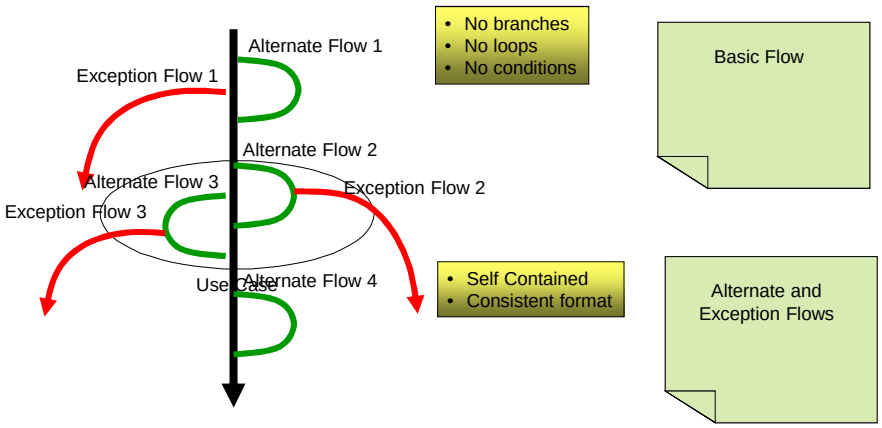
212

Use Case Model



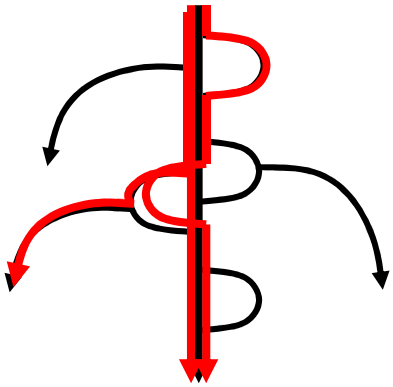
213

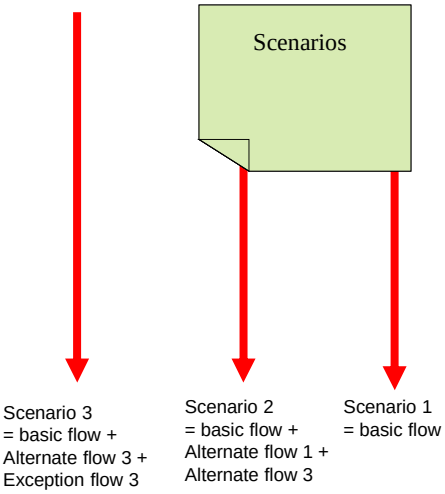
Testable Flows



214

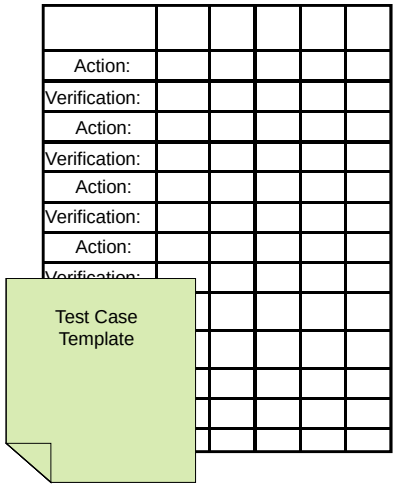
Use Case Scenarios

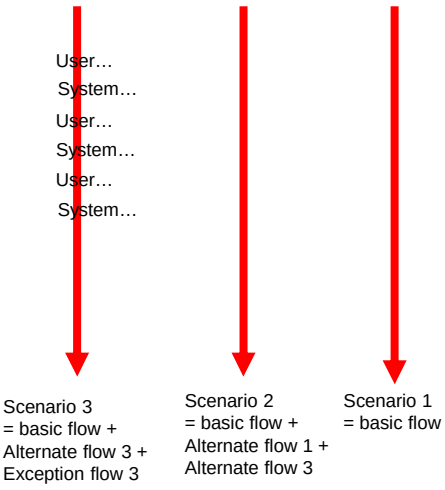




215

Test Scenarios





216

Fully Dressed Use Case Format

Use Case Name	Start with verb
Scope	System boundaries (corporate, program)
Level	Subfunction, etc.
Primary Actors	Primary system users
Stakeholders	Who cares and what they want
Preconditions	Must be true to start
Postconditions	What is guaranteed by success
Main Success Scenario	Typical, unconditional path scenario
Extensions/Exeptions	Alternative success or failure scenarios
Special Requirements	Related non-functional requirements
Technology & Data Variations List	Varying IO methods and data formats
Frequency of Occurrence	Is this system used often?
Miscellaneous	Open issues; eg. unmanageable failure scenarios

217

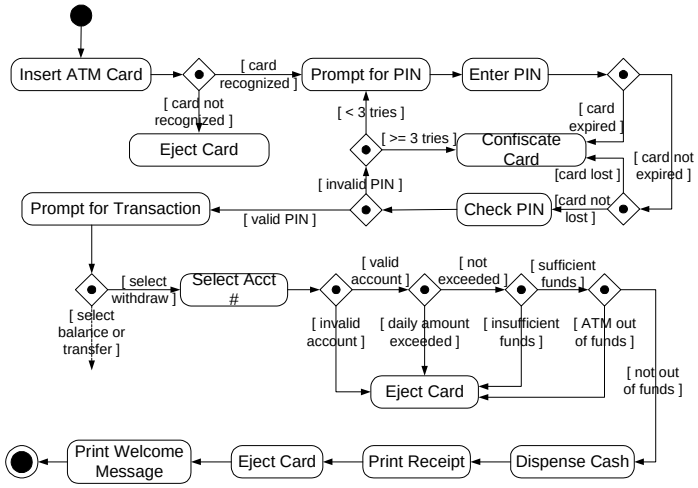
Fully Dressed Use Case Format - Example

ID:	UC-6
Title:	Register for courses
Description:	Student accesses the system and views the courses currently available for him to register. Then he selects the courses and registers for them.
Primary Actor:	Student
Preconditions:	Student is logged into system
Postconditions:	Student is registered for courses
Main Success Scenario:	1. Student selects “Register New Courses” from the menu. 2. System displays list of courses available for registering. 3. Student selects one or more courses he wants to register for. 4. Student clicks “Submit” button. 5. System registers student for the selected courses and displays a confirmation message.
Extensions:	2a. No courses are available for this student. — 2a1. System displays error message saying no courses are available, and provides the reason & how to rectify if possible. — 2a2. Student either backs out of this use case, or tries again after rectifying the cause.5a. Some courses could not be registered. — 5a1. System displays message showing which courses were registered, and which courses were not registered along with a reason for each failure.5b. None of the courses could be registered. — 5b1. System displays message saying none of the courses could be registered, along with a reason for each failure.
Frequency of Use:	A few times every quarter
Status:	Pending Review
Owner:	Attila Kovács
Priority:	P3 – Medium

218

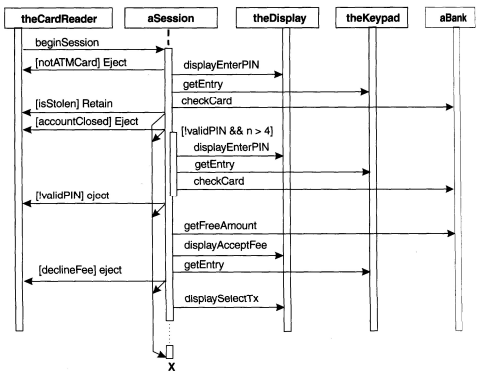
Writing test cases

- Too many scenarios → risk based testing
- Business case focused → model it with Activity graph



Writing test cases

- Technology focused (OO programming) → model it with sequence diagram



Examine

- Incorrect or missing output
- Missing function/feature in an object
- Incorrect parameter values → boundary value analysis
- Correct message - wrong object
- Incorrect message - right object
- Incorrect object state
- Message sent to destroyed object
- Incorrect exception thrown
- Incorrect exception handling
- Incorrect assumption about receiver's class

Structure Based Techniques

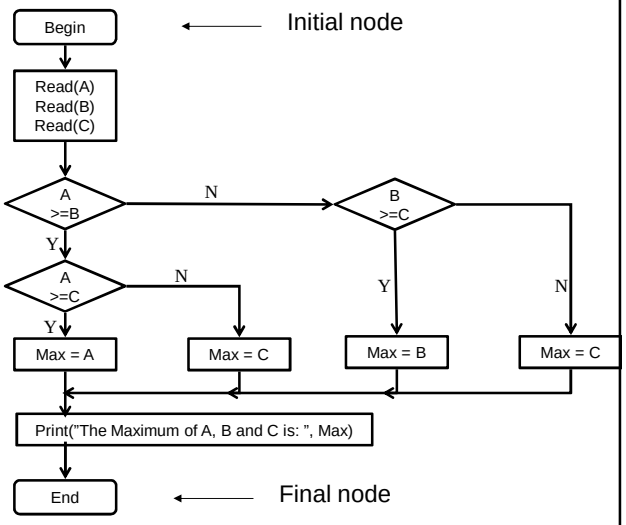
Structure based Techniques – Background

- **Structure based or white-box (or glass box) testing** is based on an identified structure of the software or system
  - **Procedure level**: backup, recovery, maintenance procedures, control scripts (batch processing)
  - **Component level**: the structure is the code itself (statements, decisions, branches)
  - **Integration level**: the structure can be the call-tree (a diagram in which modules call other modules)
  - **System level**: the structure may be the whole menu structure of the system, a business process or web page structure
- Test data is derived from the **structure of the software**
  - Generating test cases from the code itself (pseudo code)
  - Code reading and analysis are mostly done by tools

222

Code Structure Examples

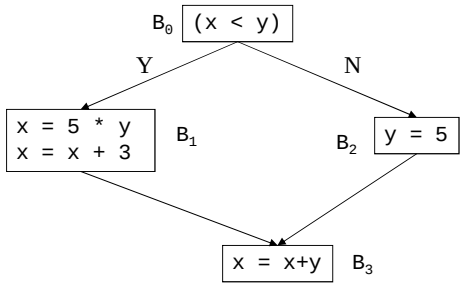
```
Program MaxSelector
    A, B, C, Max: Integer
begin
    Read(A);
    Read(B);
    Read(C);
    if (A >= B) then
        if (A >= C) then
            Max = A;
        else
            Max = C;
        endif
    else
        if (B >= C) then
            Max = B;
        else
            Max = C;
        endif
    endif
    print("The Maximum of A, B and C is: ", Max);
end
```



Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A ***basic block*** (**segment**) is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

```
if (x < y) {
  x = 5 * y;
  x = x + 3;
}
else
  y = 5;
x = x+y;
```



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

The Idea of Test Coverage

- **Test coverage** is a very important statement about how effective the testing has been or what has been achieved
  - It provides a **quantitive measure** of the quality of testing by measuring what has been achieved
  - It provides a way of **estimation** how much more testing needs to be done
- Test coverage can be applied to any systematic technique
- Coverage measures
  - May be part of the completion criteria defined in the Test Plan (first step of the Fundamental Test Process)
  - Used to determine when to stop testing in the final step of the Fundamental Test Process

225

Structure-based Testing and Coverage

- ***Coverage metrics examples***
  - ***Statement coverage***: all executable statements in the programs should be executed at least once.
  - ***Decision coverage*****:** all programming decisions for both true and the false exits of conditions should be executed
  - ***Branch coverage***: all branches in the program should be executed at least once
    - Select a test set *T* such that by executing program *P* for each test case *d* in *T*, each edge of *P*' s control flow graph is traversed at least once
  - ***Path coverage***: all execution paths in the program should be executed at lest once
    - Select a test set *T* such that by executing program P for each test case *d* in *T*, **all** paths leading from the initial to the final node of *P*'s control flow graph are traversed
    - The best case would be to execute all paths through the code, but this is almost impossible

226

Statement Testing

- The aim of statement testing is to exercise programming statements
  - Only **executable statements** counts.
    - An executable statement performs an action. It can call a procedure, branch to another place in the code, loop through several statements, or evaluate an expression. An assignment statement is a special case of an executable statement. Statements classification:
      - Declarative statements: variable declarations, constants, procedure and function headers, ...
      - Executable statements: the ones that actually do something, i.e.
        - » Control statements: conditionals, jumps, loops and error handling statements.
        - » Non-control statements: the ones that execute sequentially.

227

Statement Testing and Coverage

- Statement coverage shows the percentage of the statements exercised
  - If we test all the statements in the code we reach the 100 per cent statement coverage

$$\text{Statement Coverage} = \frac{\text{Number of Statements exercised}}{\text{Total Number of Statements}} \times 100\ \%$$

- Achieving 100 per cent statement coverage doesn't mean we have tested everything
  - There are much more rigorous coverage measures
  - Statement coverage just provides a baseline
  - In general it is too weak to be considered as an adequate measure of test effectiveness

228

Statement Testing and Coverage

- Two types of statement coverage:
  - Line coverage
  - Basic blocks coverage
- 100% statement coverage means perfection?

```
Public String FullStatementCoverage(boolean cond)
  String foo = null;
  if (cond) {
    foo = ” ” + cond;
  }
  return foo.trim();
}
```

229

Decision Testing and Coverage

- Decision coverage shows the percentage of the exercised decisions (with both exits)
  - If we test all the decisions in the code we reach the 100 per cent decision coverage

$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes exerc.}}{\text{Total Number of Decision Outcomes}} \times 100\%$$

- Complete decision test also doesn't guarantee revealing of all mistakes in the code
- 100% decision coverage (basically) guarantees 100% statement coverage

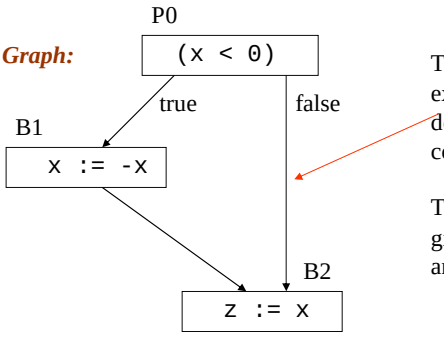
230

Statement vs. Decision/Branch Coverage

```
assignAbsolute(int x)
{
  if (x < 0)
    x := -x;
  z := x;
}
```

Consider this program segment, the test set T = {x=–1} will give full statement coverage, however neither decision nor branch coverage

Control Flow Graph:



Test set {(x=–1)} does not execute this edge, hence, it does not give decision/branch coverage

Test set {(x= –1), (x=2)} gives both statement, decision and branch coverage

231

Exercise

```
int gcd(int x, int y){
    while (x != y){
        if (x>y) then
             x=x-y;
        else y=y-x;
     }
    return x;
}
```

How many test cases we need to achieve 100% statement and decision coverage?

By choosing the test set {(x=4, y=3)} all statements are executed at least once.

By choosing the test set {(x=4, y=3)} all decisions exit with both true and false. Try {(x=6, y=3)} and {(x=3, y=4)}

232

Exercise

How many test cases are needed to achieve 100% decision coverage?

```
if (p = q)
      s = s + 1;
      if (s < 5) t = 10; end if
else if (p > q) t = 5; end if
end if
```

a) 3 b) 4 c) 5 d) 6

- 2 test cases for 100% statement coverage (all statement line will be executed)
  - `p = q and s < 4;  p > q;`
- 2 more test cases for 100% decision coverage (all decisions in the code must be tested)
  - `p = q and s >= 4;  p < q`

233

Branch Coverage

Almost the same as decision coverage:

- Branch Coverage reveals, if **all branches** were executed. (For example, an if-instruction has two branches, the then-branch and the else-branch.)
- Decision Coverage reveals, if **all decisions** evaluated to both true and false. (For example, the decision of an if-instruction is what is between the parentheses.)

BUT in short circuit languages:

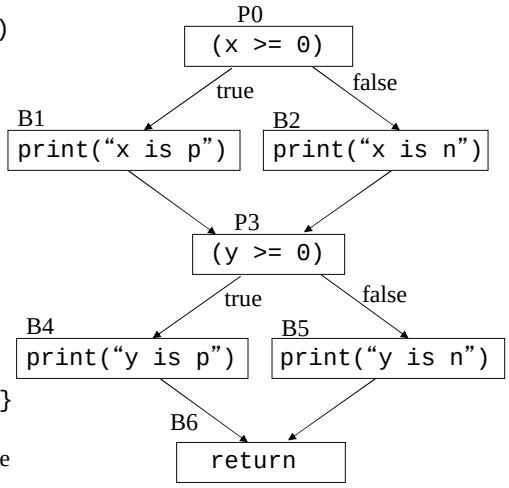
```
if (condition1 && (condition2 || function1()))
  statement1;
Else statement2;
```

they can be different

234

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```



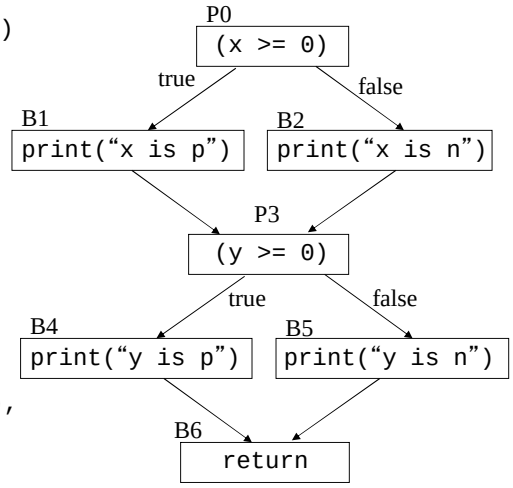
Test set:
T_2 = {(x=12,y=–5), (x=–1,y=35)}
gives both branch and statement coverage but it does not give path coverage

Set of all execution paths: {(P0,B1,P3,B4,B6), (P0,B1,P3,B5,B6), (P0,B2,P3,B4,B6), (P0,B2,P3,B5,B6)}
Test set T_2 executes only paths: (P0,B1,P3,B5,B6) and (P0,B2,P3,B4,B6)

235

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```



Test set:
T_1 = {(x=12,y=5), (x= –1,y=35), (x=115,y=–13),(x=–91,y= –2)}
gives both branch, statement and path coverage

236

Path Coverage

- Number of paths is exponential in the number of conditional branches
  - testing cost may be expensive
- Note that every path in the control flow graphs may not be executable
  - It is possible that there are paths which will never be executed due to dependencies between branch conditions
- In the presence of cycles in the control flow graph (for example loops) we need to clarify what we mean by path coverage
  - Given a cycle in the control flow graph we can go over the cycle arbitrary number of times, which will create an infinite set of paths
  - Redefine path coverage as: each cycle must be executed 0, 1, ..., k times where k is a constant (k could be 1 or 2)

237

Basis Path Testing

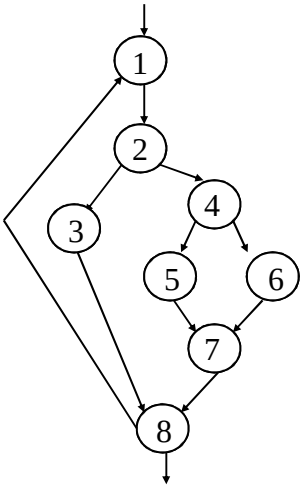
- Recall that path coverage reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit.
- It may be straightforward to identify **linearly independent paths or basis paths** (introducing at least one new node or edge that is not included in any other paths) of simple programs.
- Unfortunately, for complicated programs it is not so easy to determine the number of independent paths.
- McCabe Cyclomatic metric **upper bounds** the number of independent paths.
  - Given a non-linear control flow graph G, cyclomatic complexity V(G):

    V(G)= E-N+2
    - N is the number of nodes in G
    - E is the number of edges in G

238

Path Testing - Example

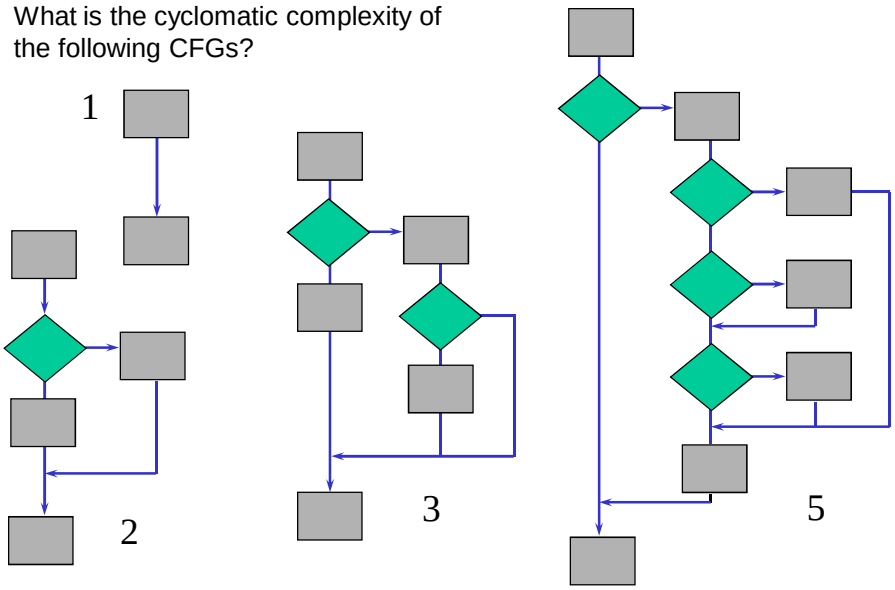
- Derive a set of paths that cover graph G
  - Path 1: 1-2-3-8
  - Path 2: 1-2-3-8-1-2-3-8
  - Path 3: 1-2-4-5-7-8
  - Path 4: 1-2-4-6-7-8
- Derive an independent (basis) path set.
- Prepare test cases that will force the execution of each path in the basis set



239

Example

What is the cyclomatic complexity of the following CFGs?



240

Experience Based Techniques

241

Experience Based Techniques

- Experience based techniques are useful when testers have not enough time to execute a full structured test set
  - It demands tester's experience to reveal the most effective tests to be run
- Experience-based techniques
  - *Error guessing*

Error guessing is a test design technique where the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specially to expose them.

  - *Exploratory testing*

Exploratory testing is a test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests (compare to „twenty questions" game)

242

Experience Based Techniques

- ***Checklist-based testing***
  - Is used by experienced testers who are using checklists to guide their testing.
  - The checklist is a remainder list of what to be tested (list of rules, particular criteria, data conditions to be verified, etc.)
  - Developed personally over time.
- ***Attack Testing***
  - Focuses on trying to include a specific type of failure.
  - Target can be: user interface, OS, DB interfaces, SW vulnerabilities

243

Defect Based Techniques

Defect Based Techniques

- Useful for focusing on and **detecting particular types of defects**
- The target ***defects are determined based on taxonomies*** (hierarchical lists) that list root causes, defects, and failures
- There are different kinds of given taxonomies
  - Beizer (defect taxonomies)
  - Kaner (general taxonomy, more than 400 defects)
  - Binder (OO taxonomy)
  - Vijayaraghavan (taxonomy for e-commerce applications)
- Many organizations have their own classification system
  - Logic, Requirements, Design, etc.
- May classify in more detail
  - Initialization, interface, etc.

245

Defect Based Techniques

- The tester who uses the taxonomy can sample from the list, selecting a potential problem for analysis.
  - The tester's question is whether the software under test could have a bug analogous to the one from the list.
  - If so, the next question is what type of test would expose this type of bug.
- Tester who has run out of good test ideas looks for plausible failure modes in the risk list, then creates tests looking for those types of failures
- Tester unfamiliar with an aspect of the program looks for potential failure modes in the risk list, then explores the program looking for those types of failures.

246

Choosing Test Techniques

- To ensure that many defects from different classes are found we must use different test techniques
- The selection of which test technique(s) to use depends on a number of factors, including internal and external factors

Internal Factors	External Factors
• Models used • Tester knowledge / experience • Likely defects • Test objective • Documentation • Life-cycle model • Previous experience of types of defects found	• Level and type of risk • Customer / contractual requirements • Type of system • Regulatory requirements • Time and Budget

247

Exercise

What techniques would be MOST appropriate if the specifications are outdated?

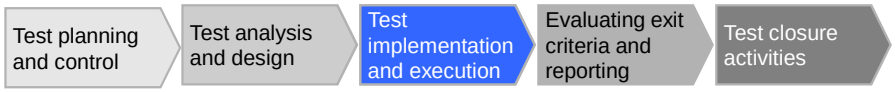
a) Structure-based and experience-based techniques
b) Black-box and specification-based techniques
c) Specification-based and structure-based techniques
d) Structure-based technique and exhaustive testing

- Solution
  - Black-box technique and specification-based technique both rely on the analysis of existing specifications to derive test cases. If the specification is outdated, it is likely that invalid test cases will be produced or developed. For this reason, we can eliminate response options b) and c). Response option d) is a plausible answer except that exhaustive testing is not a technique but a test approach. Response option a) is considered to be the correct answer.

248

Implementation & Execution

Fundamental Test Process



Test Implementation
- Developing and implementing the test cases, creating test data and prepare expected results
- Developing and prioritizing test procedures, optionally, preparing test harnesses and writing automated test scripts
- Creating test suites from the test procedures for effective test execution
- Verifying that the test environment has been set up correctly

Test Execution
- Executing test procedures either manually or by using test execution tools, according to the planned sequence (most important ones first)
- Recording the identities and versions of the Software Under Test, test tools and testware
- Logging the outcome of test execution (Pass / Fail) and comparing actual results with expected results, logging the coverage levels achieved
- Reporting discrepancies as incidents and analyzing them in order to establish their cause
- Repeating test activities as a result of action taken for each discrepancy (confirmation or re-testing, regression testing)

250

Terminology

A test harness is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. It has two main parts: the test execution engine and the test script repository. It is a test environment comprised of stubs and drivers needed to execute a test (for more details see later).

Test suite: A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.

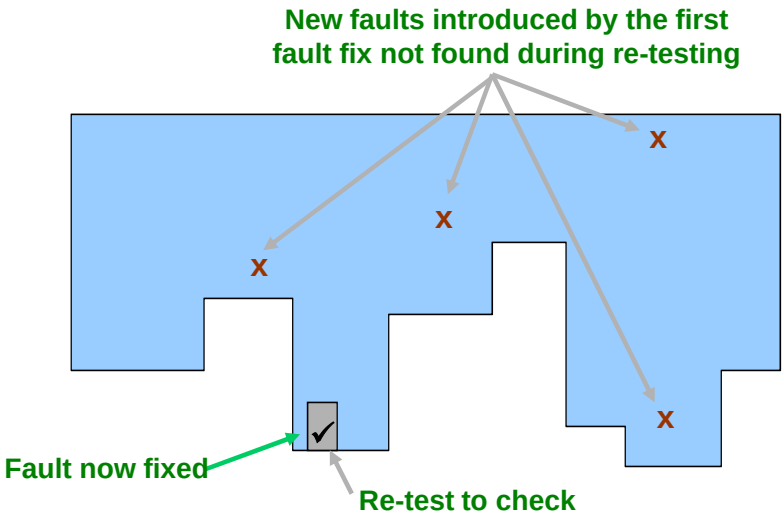
Test log: A chronological order of relevant details about the execution of tests.

Re-testing (confirmation testing)*:* Testing that runs test cases that failed the last time they run, in order to verify the success of corrective actions.

Regression testing*:* Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software as the result of the changes made.

251

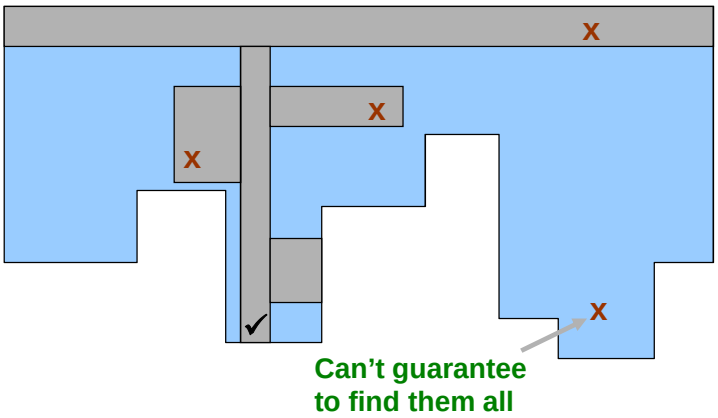
Re-testing (re-running a failed test)



252

Regression testing

- Looking for any unexpected side-effects



253

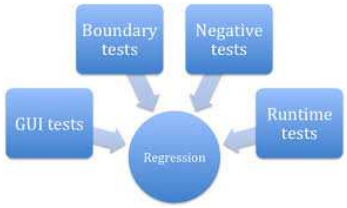
Regression testing

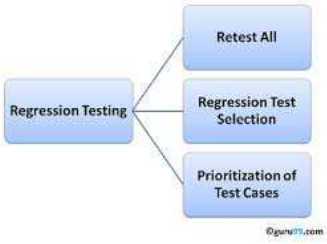
- Misnomer: "anti-regression" or "progression"
- Standard set of tests - regression test pack
- At any level (unit, integration, system, acceptance)
- Well worth automating
- A developing asset but needs to be maintained
- Regression tests are performed
  - after software changes, including faults fixed
  - when the environment changes, even if application functionality stays the same
  - for emergency fixes (possibly a subset)

254

Regression testing

- Regression test suites
  - evolve over time
  - are run often
  - may become rather large
- Maintenance of the regression test pack
  - eliminate repetitive tests (tests which test the same test condition)
  - combine test cases (e.g. if they are always run together)
  - select a different subset of the full regression suite to run each time a regression test is needed
  - eliminate tests which have not found a fault for a long time (e.g. old fault fix tests)





255

Regression testing

- Test execution tools (e.g. capture-replay) are regression testing tools - they re-execute tests which have already been executed
- Once automated, regression tests can be run as often as desired (e.g. every night)
- Automating tests is not trivial (generally takes 2 to 10 times longer to automate a test than to run it manually
- Don't automate everything - plan what to automate first, only automate if worthwhile

256

Static Testing

Static Testing

Static testing is the term used for testing where the code is not exercised. Remember: ***Dynamic testing*** involves the execution of the software.

- Testing of work products other than code and the testing of code without actually executing it
  - Requirements
  - Specification documents
  - Any other documents
- Techniques
  - ***Review*** – typically used to find and remove errors and ambiguities in documents before they are used in the development process
  - ***Static analysis*** – enables code to be analyzed for structural defects or systematic programming weaknesses that may lead to defects

258

Reviews and the Test Process

- A review is a systematic examination of a document by one or more people with the main aim of finding and removing errors
- Reviews can be used to test anything that is written or typed
  - Requirement Specification
  - System design
  - Code
  - Test Plan
  - Test Case

- **Reviews are the first form of testing** in the system development
  - Review document before code is written
  - Review code against development standards before test execution done

259

Benefits of Reviews

- Benefits of finding the defects early in the life cycle
  - Development productivity can be improved
  - Ambiguous content can be discovered and refined
  - Testing costs and the time can be reduced
  - Reduction of lifetime costs can be achieved
  - Cost and time saving
- Defect types found by reviews
  - Deviations from standards
  - Requirements defects
  - Design defects
  - Insufficient maintainability
  - Incorrect interface specifications

260

Review Process

- Level of formality
  - The more mature the process, the more formal the review
  - Legal or regulatory requirements determine the formality level
  - The need for an audit trail
- Objectives
  - Finding defects (not failures!!)
  - Gaining understanding
  - Generating discussion
  - Making decision by consensus
- Basic review process steps
  - Studying the document
  - Identifying issues and problems and inform the author (by reviewers)
  - Updating the document (by the author)

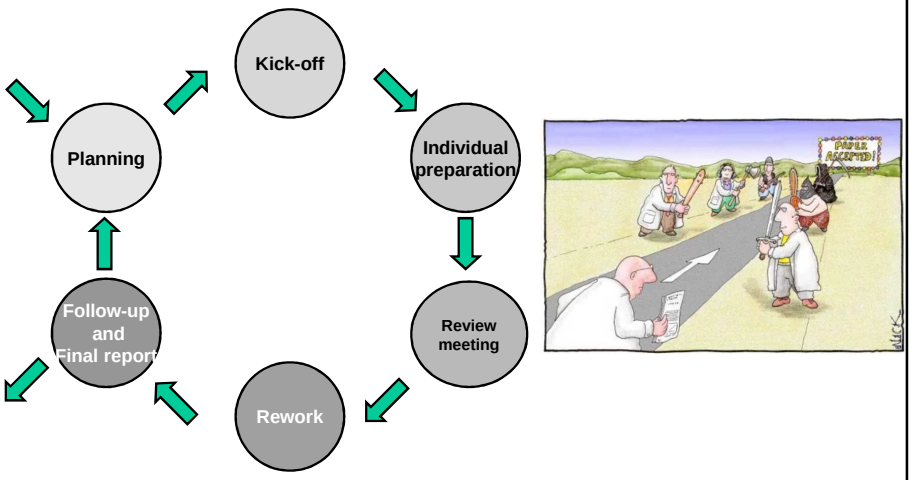
261

Roles and Responsibilities

- Manager
  - Decides what is to be reviewed
  - Ensures sufficient time allocation
  - Determines if the review objectives have been met
- Moderator (Review Leader)
  - Plans the review
  - Runs the meeting
  - Does the follow-up
- Author
  - Writer or person with chief responsibility for the develoment of the documents
  - Responsible for fixing any found defects
- Reviewer
  - Individuals with specific technical or business knowledge
- Scribe (Recorder)
  - Documents all the issues and defects, problems and open points identified during the meeting

262

Formal Review Process



263

Planning

Role	Action
Author	• Select / request a moderator • Assist moderator in his / her tasks
Moderator	• Assign roles • Create review package • Schedule meetings

- Selecting the personnel
  - Moderator is chosen submitting the deliverable to him / her and ask for review
  - The moderator first assesses the readiness of the work product for review (line numbers, stable and available references, automatic checks are ready, …)
  - The moderator selects individuals for the review team (ideal size is 4-6 person)
- Allocating roles
  - Each reviewer is given a role to provide them with a unique focus on the document under review (focus on higher level documents, on standards, on quality aspects, etc.)
- Defining the entry and exit criteria
- Selecting the parts of the document to be reviewed
  - This depends on the size of the document
  - A large document may need to be split into smaller parts

264

Kick-off

Role	Action
Moderator	• Distribute review package • Conduct meetings • Secure commitments
Author	• Present high-level overview of items to be reviewed
Reviewers	• Listen, learn and ask questions
Recorder	• Log duration of the meeting and a list off attendees

- Distributing documents
- Explaining objectives, process and documents to the participants
  - The team gets information about the work product
  - Author gives a brief summary of the article, and may draw attention to areas of concern
  - The team decides how much time they need for the review and schedule the meeting
  - The moderator should remind the team members of the purpose of the technical review
  - The purpose is to develop findings for the work product, not to redesign the article
- Checking entry criteria

265

Individual Preparation

Role	Action
Reviewers	• Review the deliverable • Record findings • Transmit findings to the moderator before the discussion.
Moderator	• Monitor reviewers' process • Help reviewers with process issues.
Author	• Answer reviewer questions

- The reviewers perform their individual review and gather their findings (use checklist!!)
- Defects by **severity** – critical, major, minor - and **type**, like
  - Error - A statement is incorrect
  - Conflict - A statement is in disagreement with a different statement in the work product
  - Missing - Some necessary or required information is not present
  - Extra - Some feature or design has been added that is not necessary
  - Unclear - A statement in the work product has several possible interpretations
- Questions
- External Issues
- Praise

266

Review Meeting

Role	Action
Moderator	• Keep discussion focused on identification, not solution • Keep discussion focused on the work product, not the author
Reviewers	• Present findings • Participate in determining review result
Recorder	• Finalize the findings log • Summarize findings at end of discussion • Log duration and attendees
Author	• Listen and learn • Answer questions • Thank the reviewers for their help
Manager	• Stay away

- The formal or informal process of providing the defect list to the author
- In case of the desktop check, the reviewer provides only the list of defects without any formal meeting.
- Checking fulfilment of exit criteria

267

Review Meeting Process

- **Moderator distributes copies** of agenda and collation report to reviewers
- **Moderator describes purpose of meeting**, process, and ground rules
- **Moderator polls reviewers**, asking how much time they spent in their preparation
  - If a reviewer is not prepared, she should change role to observer
  - If too many reviewers are not prepared, the meeting should be rescheduled for another time
- Moderator polls reviewers for any general comments before beginning a point-to point review
- Each of the **identified findings is presented** in sequence

268

Review Meeting Process

- **Recorder notes findings**. This includes the description, severity (critical, major, minor) and type
- **Recorder reads the list of findings**
  - This gives a chance to add clarification and to ask any final questions about it
- **Team determines the outcome**
  - Accept as is
  - Accept with changes
  - Revise with informal confirmation
  - Revise and hold another formal review

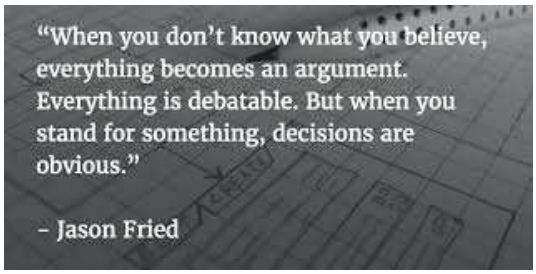
- **Author collects and marked packages** that may contain the defects

269

Rework

Role	Action
Author	• Revise the work product as indicated by the review findings

- The process of correcting or rewriting the deliverable by the author
- The author has to indicate where changes are made.



270

Follow-up

Role	Action
Moderator	• Monitor revision progress • Assure all revisions have been made
Verifier	• Confirm revisions as assigned during the discussion

- The process of checking whether all of the bugs reflected in the recorded minutes are indeed corrected
- Documents can be distributed and collected again
- For more formal review types the moderator checks for compliance to the exit criteria

271

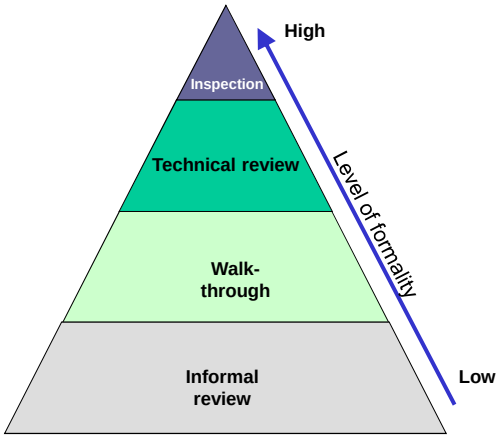
Final Report

Role	Action
Moderator	• Summarize the verification • Perform statistical analysis as needed • Make recommendations as appropriate

- The process of recording the review process
- All the activities and the relevant information should be recorded and stored, as well as the lessons learnt
- The statistical analysis (number of defects found, defects found per page, time spent checking per page, total review effort, etc.) addresses issues

272

Review Process Types



273

Informal Review

Description	• An informal review (desk check) is a one-to-one peer review • The main purpose is to **find defects** • Inexpensive but may not be very effective at finding defects • There is no formal process underpinning the review • The review may be documented but this is not required
Roles	• Author • Reviewer
Advantages	• Simple • Low overhead • Better than no review at all • Requires a little or no management approval or intervention • Easy to "legitimize" and force it to happen
Disadvantages	• Unless mandated, likely to happen only when convenient • No control over level of attention given by peer • May be a subjective evaluation • Can vary significantly in efficiency and effectiveness • No deadline

274

Walk-through

Description	• A step-by-step presentation by the author in order to **(1) find anomalies (2) consider alternative implementations (3) evaluate conformance to standards and specs (4) gather information and establish a common understanding of content.** • Issues are discussed/raised at peer level. • **Detailed study** of the documents in advance is **not always required** • Usually used to check (use case) **scenarios and program code**
Roles	• Author • Leader, Recorder (optional) • Multiple reviewers (maximum 4-6)
Advantages	• Still relatively simple • May be more effective at finding defects than informal reviews • Easy for management to „legitimize" and force it to happen
Disadvantages	• Higher overhead than in case of an informal review • Still no control over level of attention by peers • May be a subjective evaluation of the deliverable • May get side-tracked in proposing solutions • May not be any more efficient than informal review

275

Technical Review

Description	• A peer group discussion activity that focuses on **(1) achieving consensus on the technical approach** to be taken, confirms that product **(2) conforms to specifications (3) adheres to regulations, standards, guidelines, plans (4) changes are properly implemented (5) changes affect only those system areas identified by the change specification** • Defects are found by experts who focus on the content of the document. • **Preparation required** • Vary in practice from the informal to very formal and have number of purposes (discussion, decision making, evaluation of alternatives, finding defects, solving technical problems etc.)
Roles	• Moderator (review leader) • Author • Recorder • Multiple reviewers (architects, chief designers and key users)

276

Technical Review

Advantages	• Known to be more efficient and effective than walkthroughs • Control the meeting process • Less subjective
Disadvantages	• Higher overhead than walkthrough • Visibility of meeting may force management per-approval • The increased formality can be seen as „draconian”
Input	• Software requirements specification • Software design description • Software test documentation • Software user documentation • Installation procedure • Release notes
Output	• Software product reviewed • List of resolved and unresolved software defects • List of unresolved system or hardware defects • List of management issues • Action item status • Recommendations for unresolved issues • Whether software product meets specification

277

Inspection

Description	• The most formal peer review that relies on visual **examination of documents to detect defects,** confirming that the software product satisfies (1) specifications, (2) specified quality attributes (3) regulations, standards, guidelines, plans (4) identifies deviations from standard and specification • Individual inspectors work within defined roles • Based on **rules and checklists, uses entry and exit criteria** • Pre-meeting preparation, list of **findings with metrics are essential**
Roles	• Moderator (review leader) • Author • Recorder • Multiple reviewers (inspectors) - Verifier

278

Inspection

Advantages	• Known to be more efficient and effective than walkthroughs • Control over the level of attention by reviewers • Much less subjective • High level of audit trail
Disadvantages	• Requires a significant „infrastructure" investment • It is not a simple process • The increased formality can be seen as „draconian" • Requires management pre-approval and support
Input	• Software requirements specification • Software design description • Source code • Software test documentation • Software user documentation • Maintenance manual • Release notes
Output	• Software product inspected • Size of the materials inspected • Defect list (detail) and summary list • Disposition of the software product • Estimation of the rework effort and completion date

279

Review Processes Types – Summary

	Informal Review	Walk-through	Technical Review	Inspection
Planning	–	mandatory	mandatory	mandatory
Kick-off	–	–	optional	mandatory
Individual Preparation	–	optional	mandatory	mandatory
Review Meeting	mandatory	mandatory	mandatory	mandatory
Rework	mandatory	mandatory	mandatory	mandatory
Follow-up	–	optional	optional	mandatory

280

IEEE 1028 - Software Reviews

- Defines systematic software reviews having
  - team participation
  - documented results of the review
  - documented procedures for conducting the review
- Covers five types of reviews
  - management reviews
  - technical reviews
  - inspections
  - walkthroughs
  - audits

281

Cost and Quality of Inspections

- There is a “cost of quality” associated with inspections. In software, person-hours are the highest measurable expense
- Many organizations find that the cost of inspection does not generate a return on investment
  - When not supported by management, easily become “busy work”
- Some inspect a percentage of code
- Others inspect only critical portions
- **When done correctly, reviews are valuable defect finding tools**

282

Review Process – How? When? Why?

- Success factors
  - Clearly defined and agreed objective, the right people should be involved
  - Review techniques suitable to the type and level of work-products
  - Checklists or roles should be used where appropriate
  - Management support
  - Learning, process improvement
- Quantitative approaches
  - How many defects found
  - Time taken to review / inspect
  - Percentage of project budget used / saved

283

Static Analysis by Tools

- Its objective is to find **defects** in
  - *Software source code*

> *Software source code* is any series of statements written in human-readable computer programming language which can be converted to equivalent computer executable code.

  - *Software models*

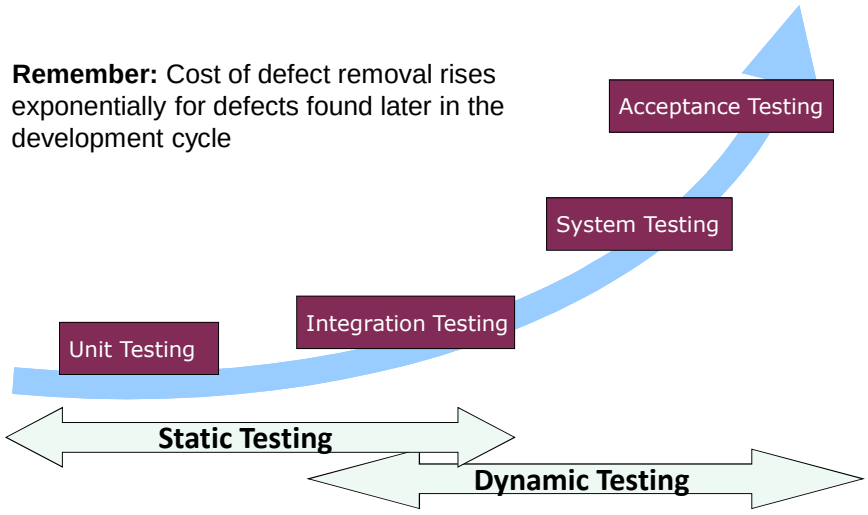
> *Software model* is any artificial model that can be used to express information or knowledge in a structure that is defined by a consistent set of rules. The model is an image of the final software solution

- Static analysis tools give the greatest value when used during component or integration testing. They run automatically and report defects, quality notices

284

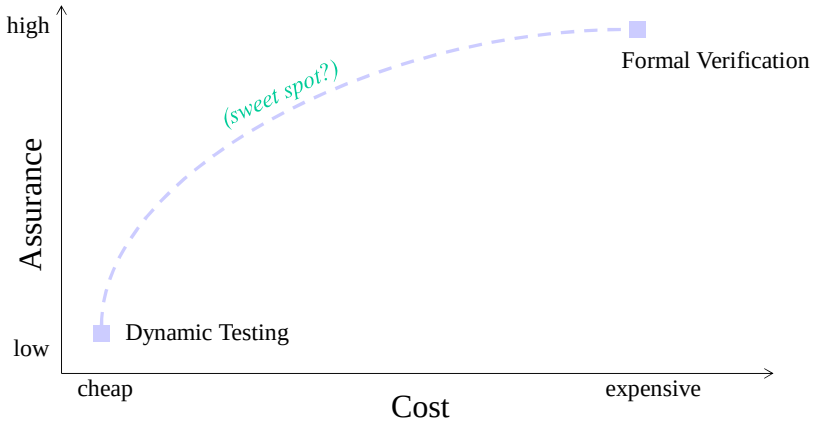
Defect Removal Cost

Remember: Cost of defect removal rises exponentially for defects found later in the development cycle



285

Quality Assurance – Static vs. Dynamic



286

Static Analysis by Tools – Benefits

- Early detection of defects prior to test execution
- Early warning about suspicious aspects of the code / design (e.g. complexity)
- Identification of defects not easily found by dynamic testing, i.e. security vulnerabilities
- Improved maintainability of code and design (i.e. architecture)
- Prevention of defects
- Generating code metrics

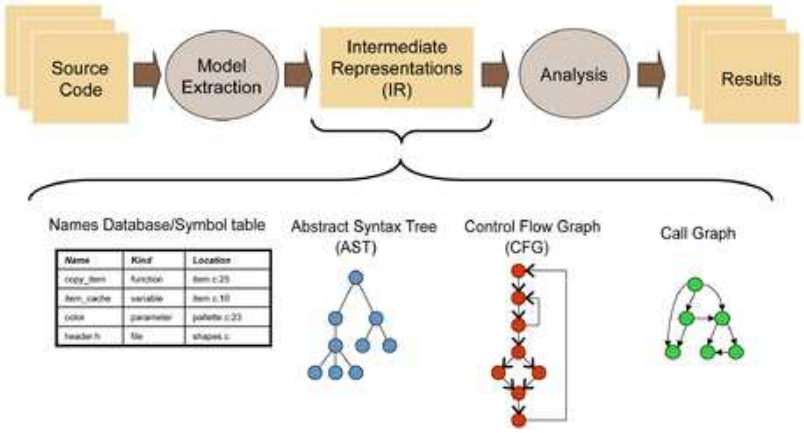
287

Types of Automated Static Analysis

- Syntactic Analysis
- Data Use Analysis
  - Aim is to identify data flows that do not conform to programming practices, e.g. variables are not read before they are written, etc.
- Control Flow Analysis
  - Aim is to detect poorly structured code, e.g. multiple exits from a loop, dead code, etc.
- Interface Analysis
- Program Slicing
  - Program slicing involves focusing on a particular subset of variables within a given program. The parts of the program that are relevant to the subset of variables denotes a program slice.
- Path Analysis

288

Static Analysis by Tools



289

Test Reporting and Closure

Fundamental Test Process

Test planning and control | Test analysis and design | Test implementation and execution | Evaluating exit criteria and reporting | Test closure activities

Evaluating exit criteria

- Checking test logs against the exit criteria specified in test planning
- Determining whether more tests are needed or exit criteria should be changed

Reporting

- Writing a test summary report for stakeholders
- Communicating effectively the findings
- Providing analyzed information and metrics
- Assessment of defects remaining
- Economic benefit of continued testing
- Outstanding risks
- Level of confidence in tested software

291

Fundamental Test Process

Test planning and control | Test analysis and design | Test implementation and execution | Evaluating exit criteria and reporting | Test closure activities

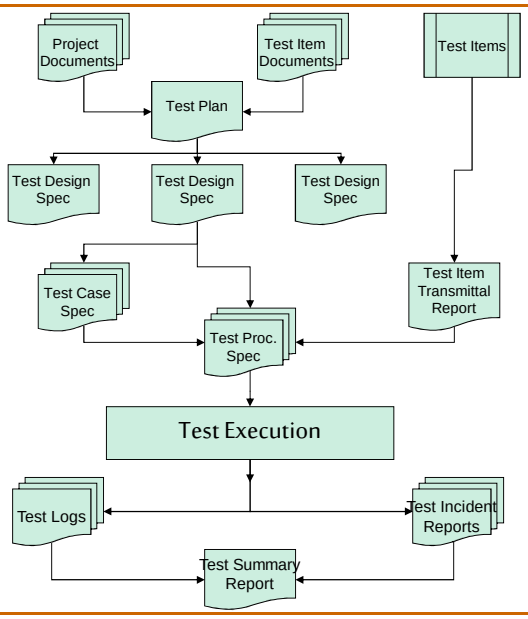
Test closure activities

- Checking which planned deliverables have been delivered, the closure of incident reports or raising of change records for any that remain open, and the documentation of the acceptance of the system
- Finalizing and archiving testware, the test environment and the test infrastructure for later reuse
- Handover of testware to the maintenance organization
- Analyzing lessons learned for future releases and projects, and the improvement of test maturity

292

Test Documentation

IEEE 829 Document Tree



PD: Requirements, specifications
TID: Release notes, defect reports
TI: Description of item to be tested
TP: Scope, approache, resources, scheduling,
TDS: Features to be tested, approach, feature pass/fail criteria
TCS: I/O Spec, environment needs, items/feature to be tested
TITR: Identifies the items being sent for testing, including location and status
TPS: Steps executing a set of test cases
TL: Details about the execution of the tests, including results and environment
TIR: Documentation of any incident that occured during testing that requires investigation
TSR: Summary of the results of the testing activities and evaluation of results

294

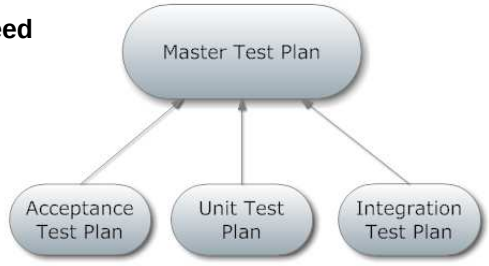
IEEE 829 Software Test Documents

- **Test Plan**: a management document that shows what will be tested, how the testing will be done (including SUT configurations), who will do it, how long it will take (although this may vary, depending upon resource availability), what the test coverage will be, what quality level is required
- **Test Design Specification**: detailing test conditions and the expected results as well as test pass criteria (test scenarios)
- **Test Case Specification**: specifying the test data for use in running the test conditions identified in the Test Design Specification
- **Test Procedure Specification**: detailing how to run each test, including any set-up preconditions and the steps that need to be followed
- **Test Item Transmittal Report**: reporting on when tested software components have progressed from one stage of testing to the next
- **Test Log**: recording which tests cases were run, who ran them, in what order, and whether each test passed or failed
- **Test Incident Report**: consists of all details of the incident such as actual and expected results, when it failed, and any supporting evidence that will help in its resolution.
- **Test Summary Report**: A management report providing an indication whether the software system under test is fit for purpose

295

Test Plan & Levels

- Successful Test Planning enables the mapping of tests to the software requirements and defines the **entry and exit criteria** for each phase of testing
- The **Level of Test Plan** defines what the test plan is being created for e.g. subsections of testing: Integration, Unit, Acceptance
- A Test Plan document will follow the **same structure** for each level of test plan. The only difference being the content and detail.
- Hierarchy of Test Plans exists
- All Test Plans **must be agreed**



296

The Test Plan Document

- Test Plans follow a strict structure to ensure all aspects of testing are covered. This is stated by the **ANSI/IEEE 829:1988 Test Plan Structure**:

1. Plan Identifier	8. Suspension Criteria
2. Test Items	9. Test Deliverables
3. Risk Issues	10. Environmental Requirements
4. Features to be Tested	11. Staffing/Training Needs
5. Features not to be Tested	12. Schedule of Test
6. Test Approach	13. Planning for risks
7. Pass/Fail Criteria	14. Approvals

297

Test Plan

- **Identifier:** identifies the Test Plan, it's test level and the level of software it's related to
- **Test Items:** specifies the things that are to be tested within the scope of this test plan:
  - Functions of the software
  - Requirements stated in the Design stage

  SW, HW and other materials needed for testing will also be listed here
- **Risks:** all risks associated with the software and its testing need to be identified in this section
- **Features to be tested:** the section identifies the features to be tested from a *user's point of view*. This is a low-level non-technical description
- **Features not to be tested:** this section lists the features not to be included in the testing process, identifying the reason behind its exclusion
- **Approach:** the bulk of information on testing techniques and methodologies will be included in this section

298

Test Plan (contd.)

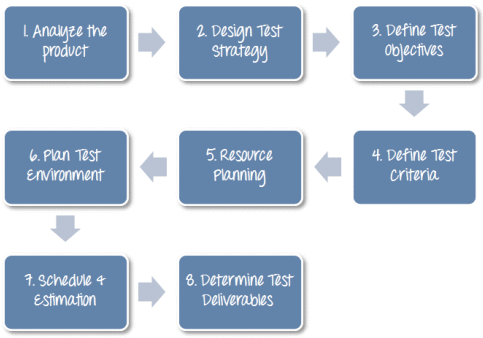
- **Test Pass/Fail Criteria**: a successful Test Plan should indicate when a project stage can or cannot proceed
- **Suspension Criteria:** specifies what constitutes stoppage for a test and what is an acceptable number of defects to allow testing to continue
  - E.g. if the number of defects reaches a point where the follow on testing has no value, it makes no sense to continue the test and waste resources
- **Test deliverables:** e.g. test logs, incident reports, outputs, corrective actions taken
- **Environmental requirements:** states any special requirements for this test plan including necessary HW and SW required for testing to proceed
- **Staffing/Traning needs:** this section identifies all personnel and the hierarchies relevant to the test plan
- **Schedule of Tests**: Milestones should be identified with schedules being specified for each milestone. It should be realistic based on valid estimations

299

Test Plan (contd.)

- **Planning for Risks and Contingencies:** this section aims to identify the overall risks to the project with an emphasis on the testing process. Identified risks are then given possible solutions.
- **Approvals:** Approvals states who can consent a process as complete and allow the project to proceed to the next stage.
  - This depends on the level of test plan and can differ from a test team leader to a more executive employee

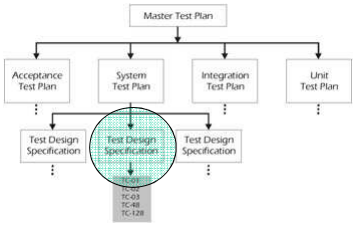
How to create a Test Plan:

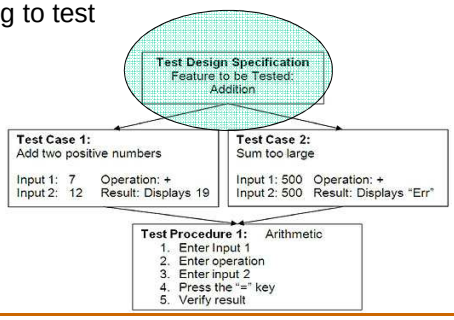


300

Test Design Specification

- The **test design** is the first stage in developing the tests for software testing projects. It records *what needs to be tested*, *which features* of a test item are to be tested, and *how a successful test* of these features *would be recognized*
- The test design does not record the values to be entered for a test, but describes the requirements for defining those values
- This document is very valuable, but is often missing on many projects. The reason is that people start writing test cases before they have decided what they are going to test





301

Test Design Specification Template

- Test design specification identifier
  - Purpose, references, definitions, acronyms, abbreviations, version, revision history
- Features to be tested
  - Features, attributes, characteristics, groupings of features with references
- Approach refinements
  - Selection of specific test techniques, reasons for technique selection, method(s) for results analysis, tools, relationships to the levels of testing, summary information relating to multiple test cases or procedures, shared environment, setup/recovery, dependencies
- Test identification
  - Identification of each test case and procedure
- Feature pass / fail criteria
  - Describing the criteria for assessing the feature or set of features and whether the test(s) were successful of not.

302

Test Case Specification Template

- A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item.
  - Test Case specification identifier
  - Test items (features and conditions)
  - Input specifications (data, ordering, values with tolerances or generation procedures, states, timing, etc.)
  - Output specifications
  - Environmental needs (HW, SW, other)
  - Special procedural requirements
  - Intercase dependencies

303

Test Procedure Specification Template

- Test Procedures are developed based on the Test Design and the Test Case Specification. The document describes how the tester will physically run the test, the physical set-up required, and the procedure steps that need to be followed. The standard defines ten procedure steps that may be applied when running a test.
  - Test procedure specification identifier
  - Objective/purpose
  - Special requirements
  - Procedure steps:
    - Log
    - Set-up
    - Start
    - Proceed
    - Measure
    - Shutdown
    - Restart
    - Stop
    - Wrap-up
    - Contingency

Test procedure: P11 (3)

Objective: This test procedure tests …
Priority: 2
Start up: The form F1 must be current.
Relationships to other: None
Expected duration: 15 min.
Execution time: *Log when* **Initials:** *Log who*
System: *Identify item etc.* **Result:** *Log overall test result*

Case	Input	Expected output	Actual output
3.6 (17)	Enter…	The sum …	*Log result*
3.8 (18)	Enter…	The value..	

Stop and wrap up: Run the "Clean 1" script.

304

Test Summary Report Specification Template

- A detail of all the important information to come out of the testing procedure, including an assessment of how well the testing was performed, an assessment of the quality of the system, any incidents that occurred, and a record of what testing was done and how long it took to be used in future test planning
- This final document is used to determine if the software being tested is viable enough to proceed to the next stage of development.
- Once approved, this report represents the approval and acceptance of the executed tests.
  - Test summary report identifier
  - Summary
  - Variances
  - Comprehensive assessment
  - Summary of results
  - Evaluation
  - Summary of activities
  - Approvals

305

Exercise – IEEE 829 Standard

- In which document should features to be tested, approach refinements, and feature pass / fail criteria BUT not environmental needs be specified?
  a) Test Case Specification
  b) Test Plan
  c) Test Procedure Specification
  d) Test Design Specification
- Solution
  - Test Case Specification identifies the input values, expected results, and executions conditions for a test item.
  - Test Design Specification describes the test conditions or coverage items for a test item or feature. Likewise, it specifies the test approach refinements, high level test cases, and pass / fail criteria.
  - Test Plan records the test planning process. It describes the scope, approach, resources, and schedule of testing activities. Also it identifies the testing tasks, who will be responsible for each task, and any risks that necessitate contingency planning.
  - Test Procedure Specification explains how a tester will run a test. It specifies the sequence of actions or steps to be taken by the tester in order to execute the test. Test Procedure Specification is also known as Test Script or Manual Test Script.

306

Exercise – IEEE 829 Standard

- Features to be tested, approach, item pass / fail criteria, and test deliverables should be specified in which document?
  a) Test Case Specification
  b) Test Procedure Specification
  c) Test Plan
  d) Test Design Specification
- Solution
  - This type of question requires not only familiarization with but also application of IEEE Standard of Software Test Documentation (IEEE Std 829:1998). Sections of a Test Plan include:
    - Features to be tested
    - Approach
    - Item pass / fail criteria
    - Test deliverables

307

Test Monitoring and Control

Test Progress Monitoring

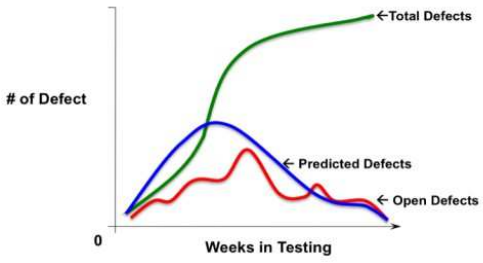
Test monitoring is a test management task that deals with the activities related to periodically checking the status of a test project.

- Tracking the testing work via well-defined *metrics*
- Providing feedback and visibility of the progress activities to the stakeholders
- Estimating and deciding the future course of action based on the *metrics* calculated
- Measure testing against the exit criteria
- Data collection
  - Manually
  - Sophisticated test management tools
  - Automatic output from a tool

309

Metrics for Monitoring

- Test execution metrics (number of test cases pass, fail, blocked, on hold)
- Defect metrics
- Requirement traceability metrics
- Test coverage metrics
- Miscellaneous metrics like level of confidence of testers, dates, milestones, cost, schedule and turnaround time.



Failure rate is the ratio of the number of failures of a given category to a given unit of measure (e.g. failures per number of time unit or transactions)

Defect density is the number of defects identified in a component or system divided by the size of the component or system expressed in standard measurement terms.

310

Test Control

Test control is basically a guiding and taking corrective measures activity, based on the results of test monitoring.

Examples include:

- Prioritizing the testing efforts
- Revisiting the test schedules and dates
- Reorganizing the test environment
- Re prioritizing the test cases / conditions

Test monitoring and control goes hand in hand. Being primarily a manager's activity, a Test Analyst contributes towards this activity by gathering and calculating the metrics which will be eventually used for monitoring and control.

311

Configuration Management

Configuration management

- Since software changes frequently, software systems can be thought of as a set of versions, each of which has to be maintained and managed
- Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems
- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems.

Configuration Management is the process of *identifying and defining the items* in the system, *controlling the change* of these items throughout their lifecycle, recording and *reporting the status* of items and change requests, and *verifying the completeness* and correctness of items – IEEE 729:1983

313

CM activities

- Change management
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- Version management
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- Build Management
  - The management of the process for assembling program components, data and libraries, then compiling these to create an executable system.
- Release management
  - Preparing software for external release and keeping track of the system versions that have been released for customer use.

314

CM terminology

Term	Explanation
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to recreate a baseline from its constituent components.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.

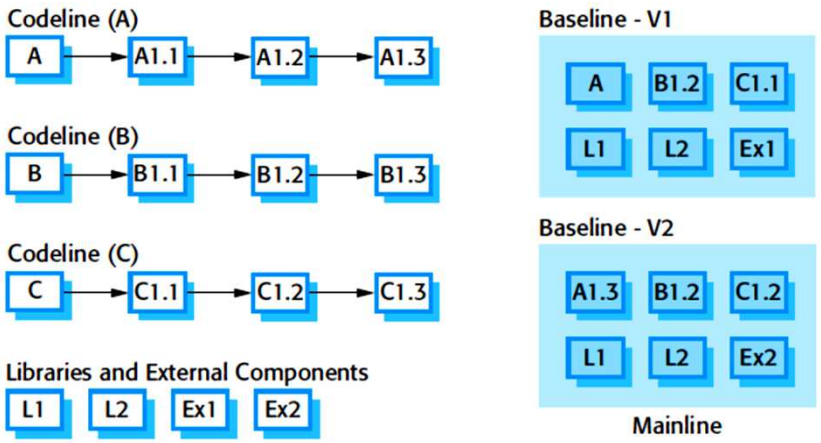
315

CM terminology

Term	Explanation
Mainline	A sequence of baselines representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

316

Codeline, baseline, mainline



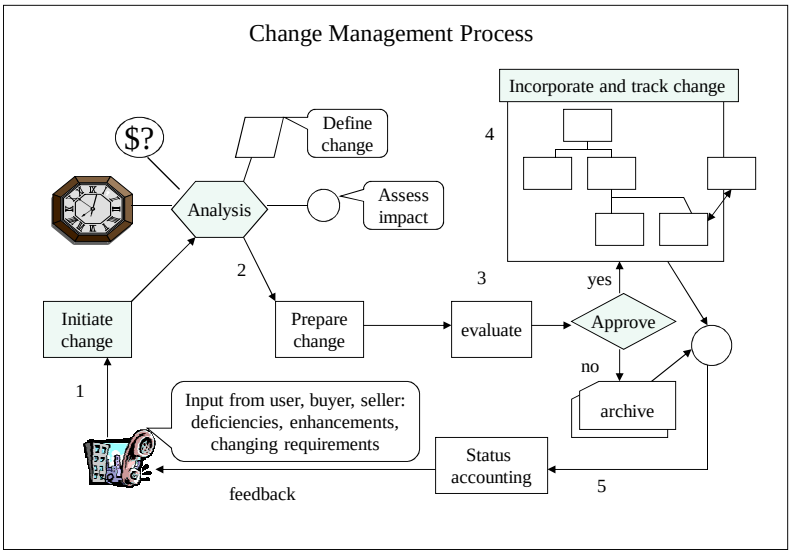
317

I. Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is **intended to ensure that system evolution is a managed process** and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with *analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed*.

318

I. Change management



319

I. Change management and CCB

- After the analysis step, a separate group should make the decision to perform the change. For military and government systems, this group is often called the *change control board* (CCB). In industry, it is often called a *product advisory board*.
- This group should review and approve all change requests. Accepted changes are passed back to the development group; rejected change requests are closed, archived and no further action is taken.
- Small requests (e.g. correcting minor errors on screen displays, webpages, or documents) should be passed to the development team without detailed analysis, as such an analysis could cost more than implementing the change.

320

II. Version management

- Version management (VM) (version control) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing codelines and baselines.
- Managed versions are assigned identifiers when they are submitted to the system.
- To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.

321

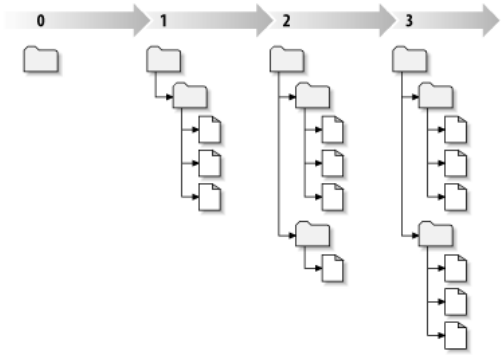
II. Version management

What should be version managed?

- All test documentation and testware
- Documents that the test documentation is based on
- Test environment
- The product to be tested
- Test cases

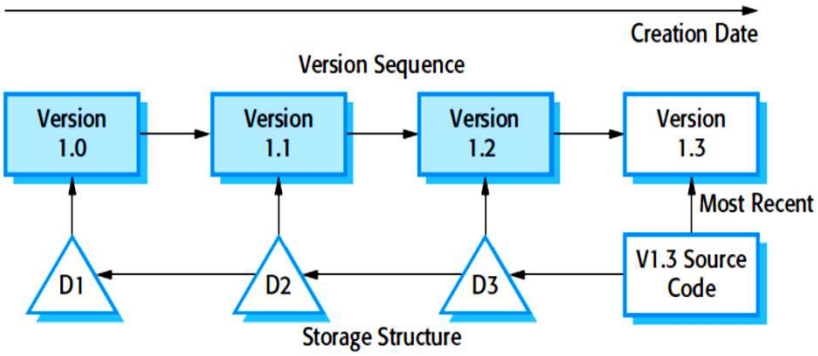
Why?

- Tracebility



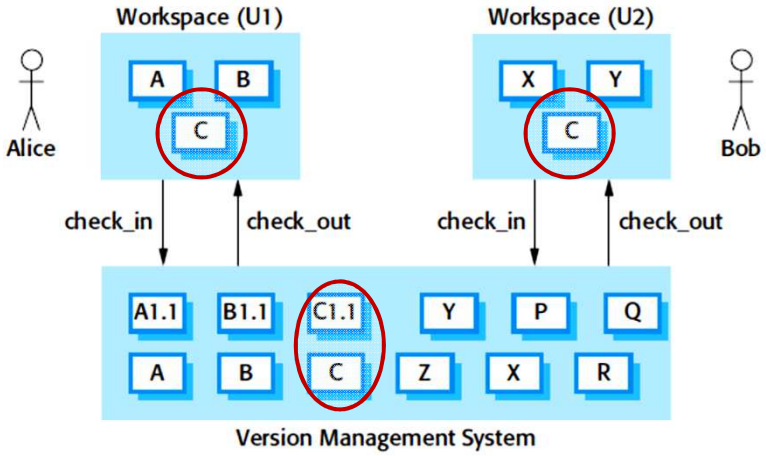
322

II. Storage management



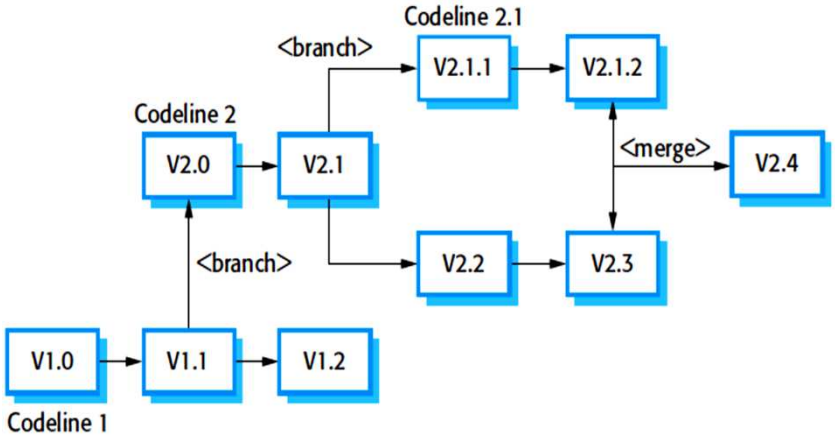
323

II. Check-in and -out from a version repository



324

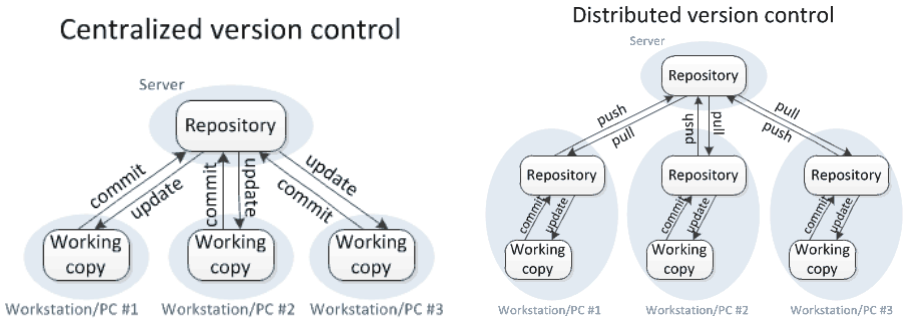
II. Branching and merging



If there are overlaps between the changes made and they interfere with each other the developer has to check for clashes and modify the changes so that they are compatible.

325

II. Version control concepts



SVN and CVS are the popular tools of CVCS
GIT and Mercurial are the popular tools of DVCS

326

III. System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system
- The configuration description used to identify a baseline is also used by the system building tool
- Build platforms:
  - The *development system*, which includes development tools such as compilers, source code editors, etc.
  - The *build server*, which is used to build definitive, executable versions of the system.
  - The *target environment*, which is the platform on which the system executes.

327

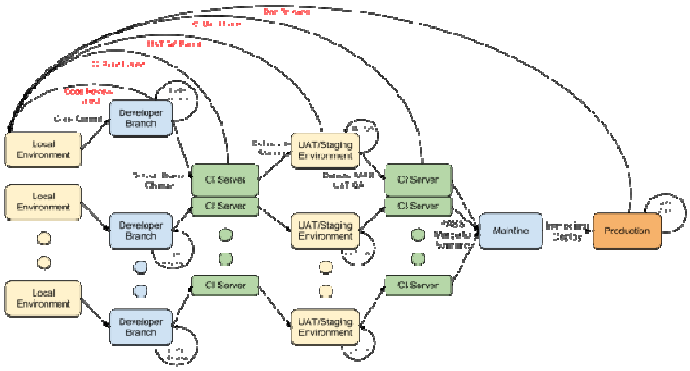
III. Deployment environments

- Deployment architectures vary significantly.
- A common 4-tier architecture is *development, testing, staging, production* (DEV, TEST, STAGE, PROD), with software being deployed to each in order.

Environment/Tier Name	Description
Local	Developer's desktop/workstation
Development/Trunk	Development server. This is where unit testing is performed by the developer.
Integration	Continuous Integration build target, or for developer testing of side effects
Test/QA/Internal Acceptance	This is the stage where interface testing is performed. Quality assurance team make sure that the new code will not have any impact on the existing functionality and they test major functionalities of the system once after deploying the new code in their respective environment (i.e. QA environment)
Stage/Pre-production/External-Client Acceptance	Mirror of production environment
Production/Live	Serves end-users/clients

328

III. Deployment workflow

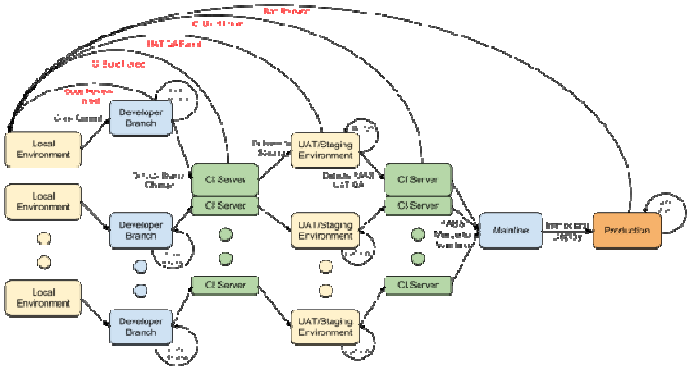


Step 1: Developer checks in code to development branch

Step 2: Continuous integration server picks up the change, reviews the code, merges it with Master/Trunk/Mainline, performs unit tests and votes on the merge to staging environment based on results.

329

III. Deployment workflow



Step 3. If Step 2 is successful, developer deploys it to the staging environment and QA tests the environment.

Step 4. If Step 3 passed, you vote to move to production and the continuous integration server picks this up again and determines if it's ok to merge into production.

Step 5. If Step 4 is successful, it will deploy to production environment.

330

IV. Release management

- A **system release** is a version of a software system that is distributed to customers
- For mass market software, it is usually possible to identify two types of release: *major releases* which deliver significant new functionality, and *minor releases*, which repair bugs and fix customer problems that have been reported
- For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time
- *Release tracking* is important since it may be necessary to reproduce exactly the software that has been already delivered to a particular customer
- *Release timing*
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems

331

Incident Management

Incident management

Incident is any unplanned event occuring that requires further investigation, anything where the actual result is different to the expected result.

Incident (defect) management is the process of recognizing, investigating, taking action and disposing of incidents – by IEEE 1044 (Standard Classification for Software Anomalies).

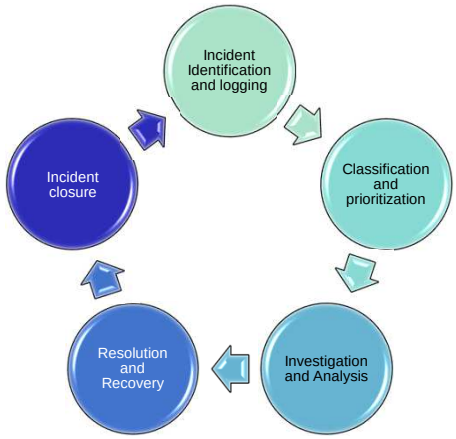
Incident reports

- Provide developers and other parties with feedback on the problem to enable identification, isolation and correction as necessary
- Provide test leaders with a means of tracking the quality of the system under test and the progress of the testing
- Provide ideas for test process improvement (e.g. Fault Slip Through)

333

Incident management

- Incidents can be raised at any time throughout the SDLC
- Incidents are raised on incident reports appointed someone (often called an Incident Manager) to manage / police the process



#1 *Identification* is via testing, user feedback, monitoring, etc. *Logging* simply means recording
#2 *Classification* helps us to partition then based on their type, *prioritization* helps us to identify the order to be handled
#3 *Investigation and analysis* is to better understand the problem, gather information for preventing from re-occurrence
#4 *Resolution and recovery* are taken to remove the incident
#5 *Closure* means retesting

334

Incident recording

- Incident report identifier (issuing organization, change history)
- Incident summary (references to external sources back to the procedure or test case that discovered it, version level of test items, software or system life-cycle process in which the incident was observed)
- Incident description (description of the anomaly)
  - Date and time the incident was discovered
  - Environment (identification or configuration item of the software or system)
  - Inputs
  - Expected and actual results
  - Anomalies with attachements and screen-shots
  - Procedure steps
  - Attempts to repeat
- Impact (severity: degree of impact of stakeholder's interests in a scale, priority to fix)
- Investigation details (who found, who are the key players in its resolution)
- Metrics (record any number of metrics on the type, location and cause of incident; the root cause is set by the programmer when fixing the defect)
- Status and history
- Comments, recommendations, conclusions

335

Incident recording – example fields

– **Title**
  - *Type the problem encountered in the application, the title needs to be understandable*

    You can use the following categories:
    - ➢ *Missing*
    - ➢ *Inaccurate*
    - ➢ *Incomplete*
    - ➢ *Inconsistent*
    - ➢ *Incorrect*

    For example:
    - » Missing validation in "Project" field
    - » Incorrect spelling in "status" drop down list

– **Test Environment**
  - *Include details of the test environment*

    For example:

    Microsoft Windows 8.1

336

Incident recording – example fields

- **Reproduction Steps with Inputs**
  - *Type all the steps to get to the problem, all steps must be cleared*

    For example:

    1 - Login to SYSTEM

    2 - Click on „Add Invoice”

    3 - Type !@#$%% in Project field

    4 - Click on Save
- **Actual Results**
  - *Type the actual results of the action*

    For example:

    The following error message is displayed…
  - Comments:

    Type any comments or notify to the developers of any screenshots (attachments)

    For example: *This defect is reproducible in Project field*. (see attached file)
- **Expected Results**
  - *Type the expected results of the action.*

    For example:

    Data should be saved successfully.

337

Types of defect severity

- **Blocking**
  - Stops the user from using the feature as it is meant to be used
  - No reasonable workaround
- **Critical**
  - Data corruption
  - Repeatably throws an exception
  - No reasonable workaround
  - Feature does not work as expected
- **High**
  - Throws an exception when not following the happy path
  - Confusing UI
  - Has a reasonable workaround
- **Medium**
  - Feature works off the happy path with minor issues
  - Small UI issues
  - One or more reasonable workarounds
- **Low**
  - Cosmetic issues
  - Many workarounds
  - Low visibility to users

Other classification scheme:

 # Types of Severity

1. **Critical**
2. **Major**
3. **Moderate**
4. **Minor**
5. **Cosmetic**

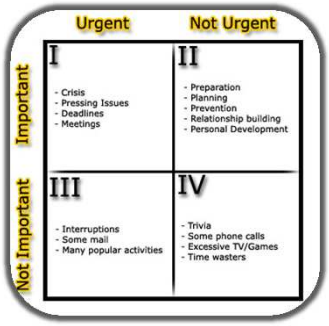
338

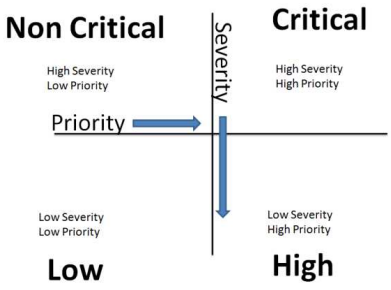
Defects prioritization and categorization

- Defects prioritization:
  - Immediate
  - Next Release
  - On Occasion
  - Open (not planned for now)
- Often based on SLA
- Defects can be categorized by quadrants:

In Agile:

25	- Critical - Must be fixed immediately - Requires notification of responsible executive - Requires customer notification and daily follow up until closed
15 - 20	- Serious - Must be included in the next sprint - Requires notification of a senior manager - Requires customer notification and weekly follow up until closed
6 - 12	- Moderate - Must be scheduled two or three sprints out
1 - 5	- Low priority - Schedule when time is available





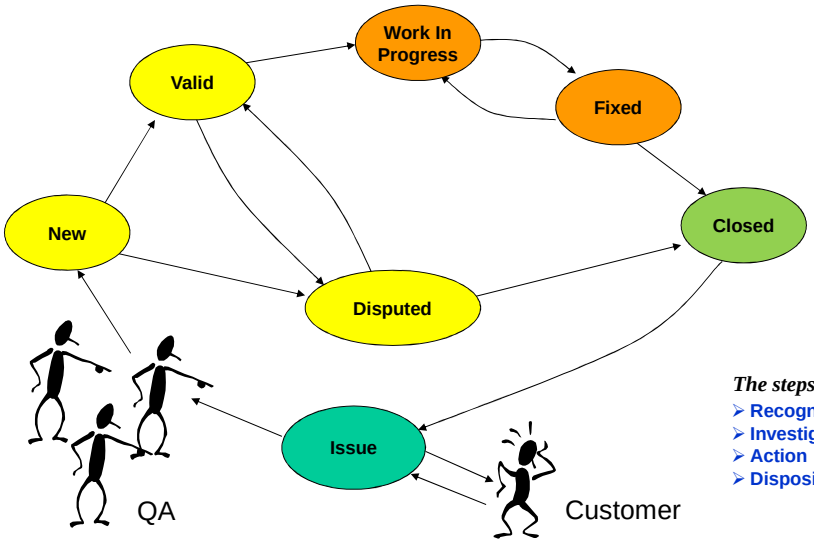
339

Defect management process

- The steps below describe a simple defect tracking process:
  - *Execute the test and compare the actual results* to the documented expected results.
  - If a discrepancy exists, *log the discrepancy* with a status of “open”. Supplementary documentation, such as screen prints or program traces, should be attached if available.
  - The test manager or tester should *review* the problem log with the appropriate member of the development team to determine if the discrepancy is truly a defect.
  - *Assign the defect* to a developer for correction.
  - Once the defect is *correc*ted, the developer will usually enter a description of the fix applied and *update the defect status* to “Fixed” or “Retest”.
  - The defect is routed back to the test team for *retesting*.
  - Additional *regression testing* is performed as needed based on the severity and impact of the fix applied.
  - If the retest result match the expected result, the defect *status is updated* to “closed”. If the test results indicate that the defect is still not fixed, the status is changed to “open” and sent back to the developer.
  - Current status can be New, Open, Re-open Wait, Reject, Fixed, Included to build, Verified, Closed

340

Defect management process



The steps:
- Recognition
- Investigation
- Action
- Disposition

341

Test Tools

Test tools

Test tool is a software product that is used to make testing more effective or efficient.

- Why tools are so useful?

	PEOPLE	COMPUTER
Strong points	■ Pattern recognition	■ Repetitive tasks ■ Comparing ■ Counting

- Take care of using tools (*Probe effect*)

Probe effect is the effect on the component or system (by the probe) when it is being measured.

  - Performance may be slightly worse when performance testing tools are being used
  - If the code is running with the debugger, then the bug disappears. It only re-appears when the debugger is turned off

343

Test tools classification

Test management related tools
- Test management tool
- Requirement management tool
- Incident management tool
- Configuration management tool

Static testing tools
- Review process support tool
- Static analysis tool
- Modelling tool

Test specification tools
- Test design tool
- Test data preparation tool

Test execution related tools
- Test execution tool
- Test harness, unit test framework tool
- Test comparator
- Coverage measurement tool

Performance and test monitoring tools
- Dynamic analysis tool
- Performance testing tool
- Load testing tool
- Stress testing tool
- Monitoring tool

Other tools
- SQL
- Debugging tool

344

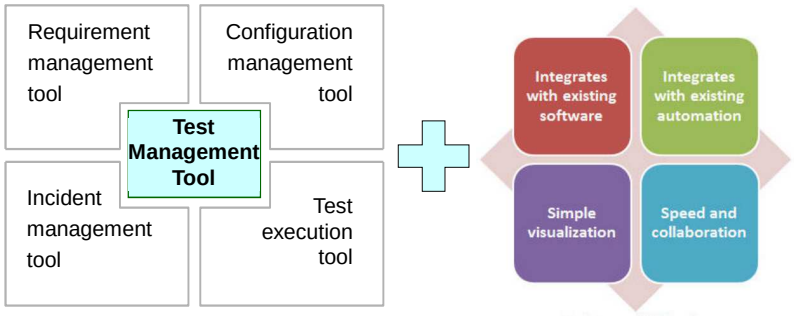
Test management related tools

Tool Type	Definition
Test management tool	• A tool that provides support to the test management and controls part of the test process
Requirement management tool	• A tool that supports the recording of requirements, requirements' attributes and annotations, facilitates traceability through layers of requirements and supports requirements change management
Incident management tool	• A tool that facilitates the recording and status tracking of incidents
Configuration management tool	• A tool that provides support for the identification and control of configuration items, their status over changes and versions and the release of baselines consisting of configuration items

345

Test management tool

- Center of a set of integrated tools



- qTest
- PractiTest
- Zephyr
- Test Collab
- TestFLO for JIRA

- XQual
- TestCaseLab
- QAComplete
- QACoverage
- Stryka

- Inflectra
- xRay
- TestMonitor
- …

346

Static testing tools

Tool Type	Definition
Review process support tool	• A tool that provides support to the review process
Static analysis tool	• A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies
Modelling tool	• A tool that supports the validation of models of the software or system

- Veracode
- Checkmarx
- Coverity
- HP Fortify
- Parasoft
- RIPS
- Clang
- CAST
- CodeSonar
- Understand
- Klocwork
- CppCheck
- Goanna
- ConQAT
- OWASP Code crawler, …

347

Test specification tools

Tool Types	Definition
Test design tool	• A tool that supports the test design activity by generating test inputs from a specification. The tests may be held in a Computer Aided Software Engineering (CASE) tool repository
Test data preparation tool	• A type of test tool that enables data to be selected from existing databases or created, generated, manipulated and edited for use in testing

A simulator tries to duplicate the **behavior** of the system.

An emulator tries to duplicate the **inner workings** of the system.

- Model based testing tools like Conformiq, 4Test, GraphWalker, ModelJUnit, Tcases, TestCast, etc.
- Pairwise testing tools
- Cause and effect graphing tools
- …

348

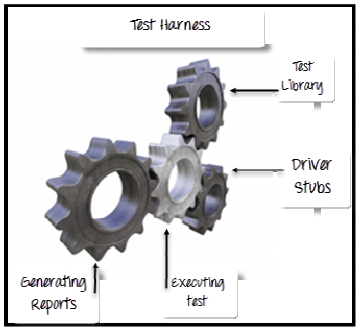
Test execution related tools

Tool Types	Definition
Test execution tool	• A type of test tool that is able to execute other software using an automated test script
Test harness, Unit test frameworks	• Test harness is a test environment comprised of stubs and drivers needed to execute a test • Unit test framework tool is a tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers
Test comparator	• A test tool to perform automated test comparison
Coverage measurement tool	• A tool that provides objective measurement of what structural elements, e.g. statements or decisions have been exercised by a test suite
Security testing tool	• A tool that provides support for testing security characteristics and vulnerabilities

349

Test harness

- *Test harness* enables the automation of tests. It refers to the system test drivers and other supporting tools that requires to execute tests
- Test harness execute tests, by using a test library and generates a report. It requires that your test scripts are designed to handle different test scenarios and test data



Examples:
- Junit for Java,
- Nunit for .Net framework

Why use Test Harness?
- Automate the testing process
- Execute test suites of test cases
- Generate associated test reports
- Support for debugging
- To record the test results for each one of the tests
- Helps the developers to measure code coverage at code level
- Increase the productivity of the system thorough automation
- Enhance the quality of software components and application
- To handle the complex condition that testers are finding difficult to simulate

350

Performance and test monitoring tools

Tool Type	Definition
Dynamic Analysis Tool	• A tool that provides runtime information on the state of the software code
Performance testing, Load testing, Stress testing Tool	• Performance-testing tool is a tool that are concerned with testing at system level to see whether or not the system will stand up to a high volume of usage • Load-testing tool is a test type concerned with measuring the behaviour of a component or system with increasing load • Stress testing is a testing conducted to evaluate a system or component at or beyond the limits of its specified requirements
Monitoring Tool	• A software tool or hardware device that runs concurrently with the component or system under test and supervises, records and/or analyzes the behaviour of the component or system

https://en.wikipedia.org/wiki/List_of_performance_analysis_tools

351

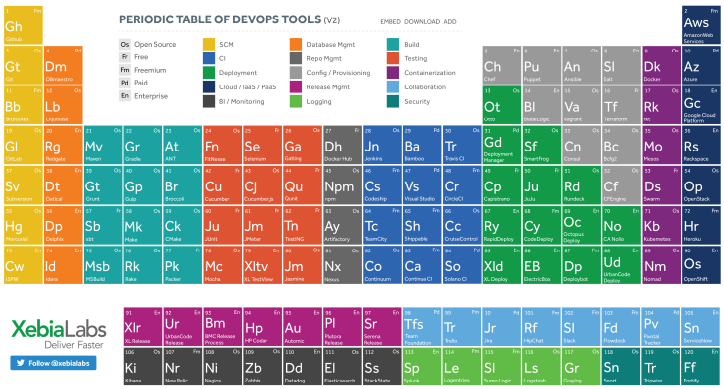
Other tools

Spreadsheet	■ Producing decision tables ■ Working out all the different test scenarios ■ Making weekly or daily test progress reports
Word processor	■ Writing test strategies, test plans, weekly reports
E-mail	■ Communicating with developers about defects ■ Distributing test reports and other deliverables
Back-up and restore utilities	■ Restoring a consistent set of data into the test environment for regression testing
SQL	■ Analysing the data held in database in order to obtain actual or expested results
Project planning tool	■ Estimating resources and timescales and monitor progress
Debugging tool	■ Used by developers to localize and fix defects ■ Enable programmers to execute programs step by step, to halt a program at any program statement and to set and examine program variables

352

Devops tools

DevOps is a term used to refer to a set of practices that emphasize the collaboration and communication of both software developers and information technology (IT) professionals while automating the process of software delivery and infrastructure changes.



353

Risks of using tools

- Unrealistic expectations for the tool
  - Important to have clear an realistic objectives for what the tool can do
- Underestimating the time, cost and effort for the *initial introduction* of a tool
  - Introducing the tool could cause problems that need to be addressed
- Underestimating the time, effort needed to *achieve significant and continuing benefits* from the tool
  - It takes time to develop ways of using the tool in order to achieve what is possible
- Underestimating the effort required to maintain the test assets generated by the tool
  - Insufficient planning for maintenance of the assets that the tool produces
- Over-reliance on the tool
  - Some test are still better executed manually
  - **A tool does not replace the intelligence**

354

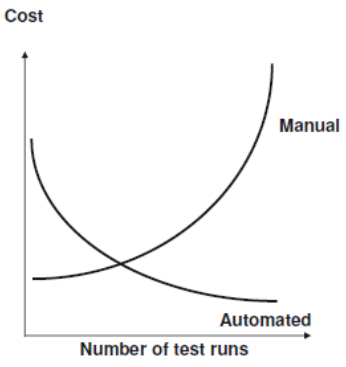
Risks of using tools – skills

- Skill needed to create good tests
- Skill needed to use the tools well, depending on the type of tool
- Skills of a tester
  - Concentrates on what should be tested, what the test cases should be and how to prioritize the testing
- Skills of a tool user
  - Concentrates on how best to get the tool to do its job effectively and how to give increasing benefit from tool use

355

Benefits of using tools

- Reduction of repetitive work
  - Running regression test
  - Checking against coding standards
  - Creating a specific test database
- Gather consistency and repeatability
  - Each time the tool is run the result is consistent
- Objective assessment
  - Subjective bias is removed
  - The assessment is more repeatable and consistently calculated



- Ease of access to information about tests or testing
  - Information presented visually (charts, graph) by tools
  - Give directly information for the tool process

356

Buy, open-source, do-it-yourself

Buy	Open-source	Do-it-yourself
Some tailoring must always be foreseen, either to the tool or to the processes in the company, or both.	The tool may be changed and enhancements should be shared.	The tool can be made exactly as the company wants it (provided it knows what it wants).
The price is usually easy to calculate.	The tool is free but there may be license fees to pay.	It may be even extremely difficult to estimate the final cost.
Usually the payment must be made within a relatively short period of time.	No immediate price needs to be paid.	The development and hence the "payment" can be done at the company's own pace.
Do what you do best – that is what the suppliers do.	The quality depends on the exposure, history, and use of the tool.	Maybe you are the best suited to develop your own tool.

357

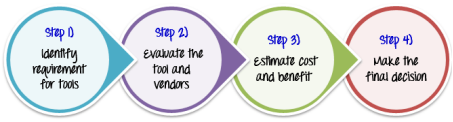
Tool selection

Tool selection criteria

- What you need now
  - Detailed list of technical requirements
  - Detailed list of non-technical requirements
- Long-term automation strategy of your organization
- Integration with your test process
- Integration with your other (test) tools
- Time, budget, support, training

Tool selection process

1. Creation of candidate tool shortlis
2. Arranging demos
3. Evaluation of selected tools
4. Reviewing and selecting tool
5. Tool implementation (plan resources, gain management support, support and mentor, training, pilot project, early evaluation, publicity of early success, test process adjustments)



358

Good practices for introducing tools

- *Define and communicate* guidelines for the use of tools, based on what was learned in the pilot
- *Provide adequate training*, coaching and mentoring of new users
- *Adapt and improve processes*, testware and tool artefacts to get the best fit and balance between them and the use of the tool
- *Incremental roll-out* (after the pilot) to the rest of the organization
- *Implementing a continuous improvement* mechanism as tool use spreads through more of the organization
- *Monitoring the use of the tool* and the benefits achieved and adapting the use of the tool to take account of what is learned

359

360