TVVS – Software Testing Verification and Validation

Introduction

Ana Paiva

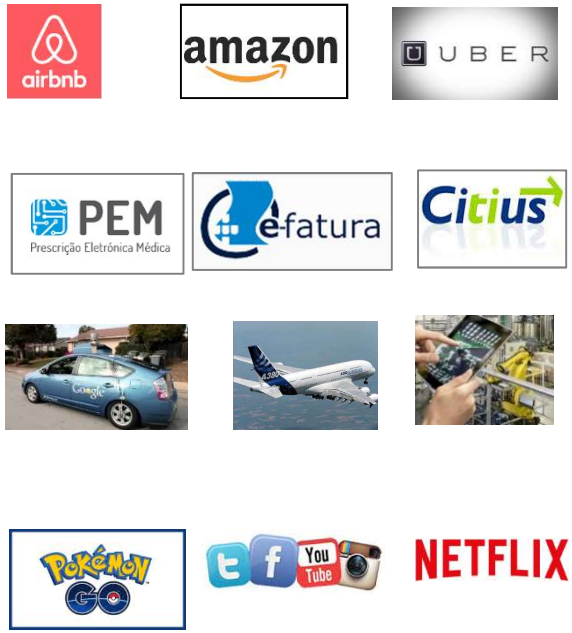
apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- **Current status of software quality**
- Software Testing (introduction)

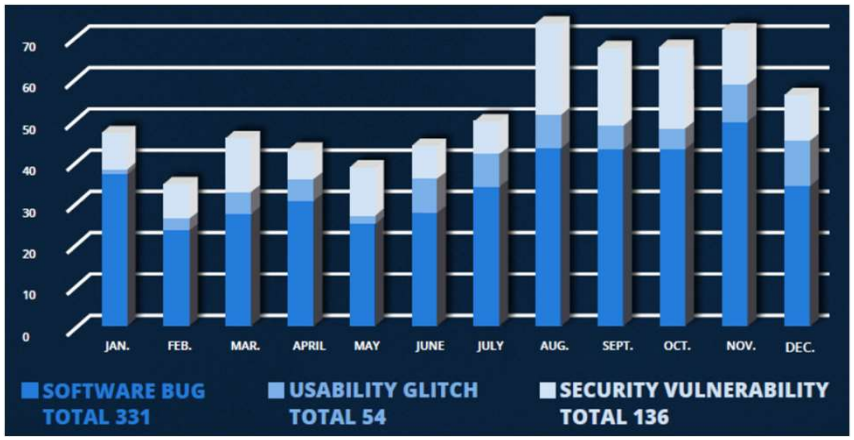
Software is Everywhere

- Enterprise applications & e-commerce
- Public services (healthcare, justice, …)
- Computer controlled systems (transport., I4.0, IoT,…)
- Media and entertainment



Software failures are everywhere: Software FAIL WATCH 2017

- Software failures are bigger than ever: 304 big cases analyzed in 2017
- Estimated financial losses: 1,7 trillion USD (2% world GDP)
- Estimated persons affected: 3,7 billion (50% world pop.)



https://www.tricentis.com/software-fail-watch/

Software failures are everywhere: examples in Public Services in Portugal

P Entrar

BASE DE DADOS FOI A MESMA

Nova empresa resolveu colocação de professores em seis dias

SOFIA RODRIGUES

30 de Setembro de 2004, 9:35

negocios ASSINAR

JUSTIÇA

JUSTIÇA

Juízes: Problemas no Citius e nos tribunais estão a causar " enorme desgaste"

Lusa

04 de outubro de 2014 às 15:18

As conclusões do X Congresso dos Juízes, aprovadas por unanimidade em Troia, referem que esta situação pode causar "danos irreparáveis na confiança dos cidadãos na justiça".

SIM SINDICATO INDEPENDENTE DOS MÉDICOS

Basta de contemporizações com a PEM (Prescrição Electrónica Medicamentos) !

25 março 2014

Partilhar

Imprimir

Os médicos estão fartos dos problemas constantes na prescrição de Medicamentos e de Cuidados Respiratórios Domiciliários com a aplicação informática PEM, desenvolvida pela SPMS e que é de uso obrigatório.

OBSERVADOR

PORTAL DAS FINANÇAS

Finanças demoraram seis meses a responder a alerta de segurança informática

23/8/2018, 8:09 165 6

Software failures are everywhere: Famous Examples Worldwide

- Therac-25 (1985-1987)
  - 5+ patients died because of massive overdoses of radiation caused by a software error (race condition)

- Ariane 5 Explosion (1997)
  - $7 billion lost because of a software error (overflow)

- Air traffic control system in UK (2004-2018)
  - Software failures causes chaos in airports

- Blackberry blackouts (2009-2012)
  - Several service blackouts for several hours

More : http://spectrum.ieee.org/computing/software/why-software-fails

More bad exemples…

- Intel spend $475 m correcting a problema with floating point inside Pentium in1994 (Computer Science, Springer Verlag – 1995)
- PrimeCo Personal Communications canceled a contract of $500M with Motorola because of failures (Wall Street Journal – 24/02/98)
- Time Warner Communications spend $1B in information systems to enter the residential business of the telephone network (Computerworld – 05/05/97)
- National Bank of Australia lost $1,75B due to na errors not detected during 2 years (New York Times – Nov/01)
- Ariane 5 (10 years of development $7B) with a charge of $ 500M, exploded 40 seconds after launch. Software module generated untreated event (ESA - 1996)
- Therac-25 provided incorrect doses of X-rays in patients between 1985 and 1987 - 6 deaths (IEEE Computer - 07/07/93)
- More and constant updates at **http://www.risks.org**

The current status of software quality (1)

- Microsoft Windows XP End-User License Agreement:

  11. LIMITED WARRANTY FOR PRODUCT ACQUIRED IN THE US AND CANADA.

  Microsoft warrants that the Product will perform substantially in accordance with the accompanying materials for a period of ninety days from the date of receipt.
  (…)

  YOUR EXCLUSIVE REMEDY. Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option from time to time exercised subject to applicable law, (a) **return of the price paid** (if any) for the Product, or (b) **repair or replacement** of the Product, that does not meet this Limited Warranty and that is returned to Microsoft with a copy of your receipt.
  (..)

The current status of software quality (2)

	India	Japan	US	Europe & other	Total
Number of projects	24	27	31	22	104
Median output[1]	209	469	270	436	374
Median defect rate[2]	.263	0.020	.400	.225	.150

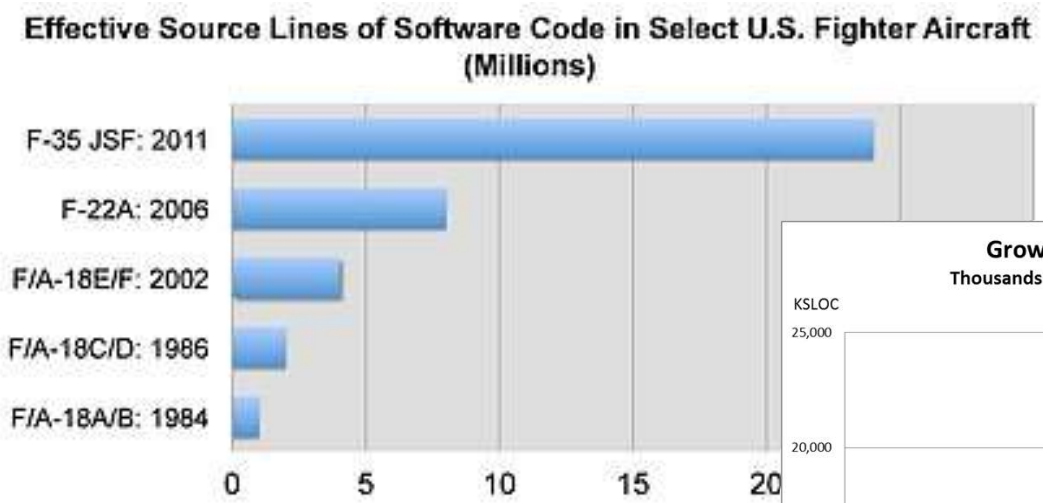
[1] (new?) LOC / programmer-month (considering the whole life cycle)

[2] Number of defects/ KLOC reported by customers in the first year post delivery

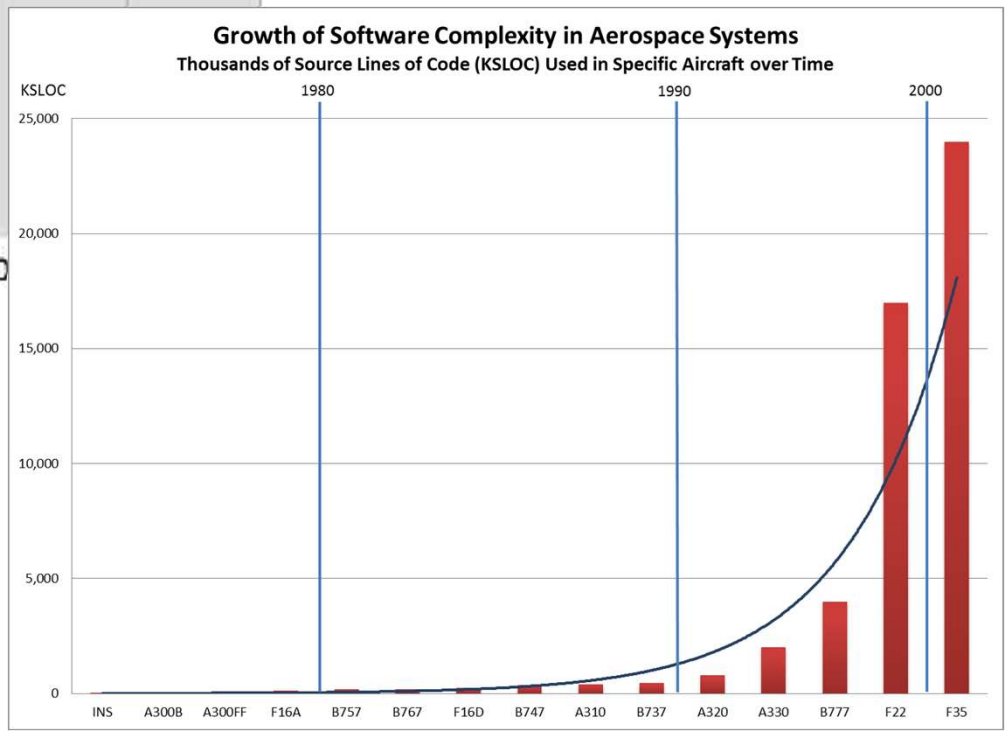
[source: "Software Development Worldwide: The State of the Practice", M. Cusumano (MIT), A. MacCormack (Harvard Univ.), C. F. Kemerer (Pittsburgh Univ.), B. Crandall (HP), IEEE SOFTWARE, 2003]

The importance of software quality (1)

We depend more and more on software …



Effective Source Lines of Software Code in Selected U.S. Fighter Aircraft



Growth of Software Complexity in Aerospace Systems

The importance of software quality (2)

- Software size increases by a factor of 10 every 10 years …
  - 50 KLOC - Word 3.0 for DOS
  - 1 MLOC - Unix, System V, Release 4, 1990
  - 10 MLOC – Linux, 2000
  - 50 MLOC - Windows Vista, 2007

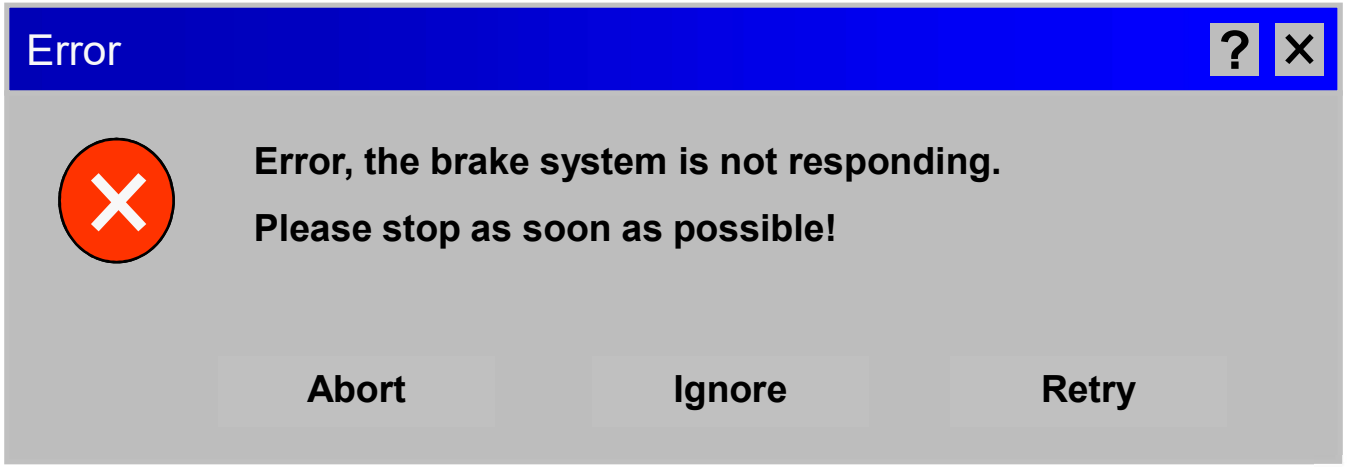
- 50 MLOC * 1 defect/KLOC = 50.000 defects in Windows Vista?

- Ideally: 1 defect / MLOC

KLOC = 1.000 lines of code
MLOC = 1.000.000 lines of code

The importance of software quality (3)

Impact on life and environment …



Tim Davis, Ford Motor Company, 27th International Conference on Software Engineering, 2005

Economic impact …

- According to the National Institute of Standards and Technology (NIST), USA, direct costs of software error represent 0,6 % of GNP (PIB) in the USA

  [source: "The Economic Impacts of Inadequate Infrastructure for Software Testing", NIST, May 2002]

The Need for Testing

- Even experienced developers inject on average one defect per 10 lines of code they write
- Current software systems have millions of lines of code, so thousands of defects are injected during development
- Such large numbers of defects injected have to be removed by applying a combination of “filters”:
  - Static analysis (applied mostly to code, by compilers and specific analysis tools)
  - Reviews and inspections (applied to req. specs, design specs, models & code)
  - Animation and simulation (applied mostly to models)
  - (Dynamic) Testing (at the unit, integration, system & acceptance levels)

Software Testing is Extremely Challenging

- Since exhaustive testing is impossible, how can we generate a set of test cases of manageable size that has a high probability of detecting most of the defects?

- How can we automate the test process (test generation & test execution) in order to increase the efficiency (effort per defect found) and effectiveness (percentage of defects found) of the test process?

Why quality pays?

- Poor-quality software can be life-threatening
  - Or mission/business/environment/economy-threatening …
- Quality work saves time and money
  - Defect density decreases 10 while productivity increases
- Quality work is more predictable
  - The testing and repair effort of a bad quality product is unpredictable

[Source: Watts Humphrey, “Winning with Software”, 2002]

Quality costs

- Costs of **conformance**
  - All costs associated with planning and running tests (and revisions) just one time
- Costs of **nonconformance**
  - Costs due to **internal** failures (before release)
    - Cost of isolating, reporting and regression testing bugs (found before the product is released) to assure that they're fixed (left-hand side of fig. 1.2)
  - Costs due to **external** failures (after release)
    - If bugs are missed and make it through to the customers, the result will be costly product support calls, possibly fixing, retesting, and releasing the software, and – in a worst case-scenario – a product recall or lawsuits (right-hand side of fig. 1.2)

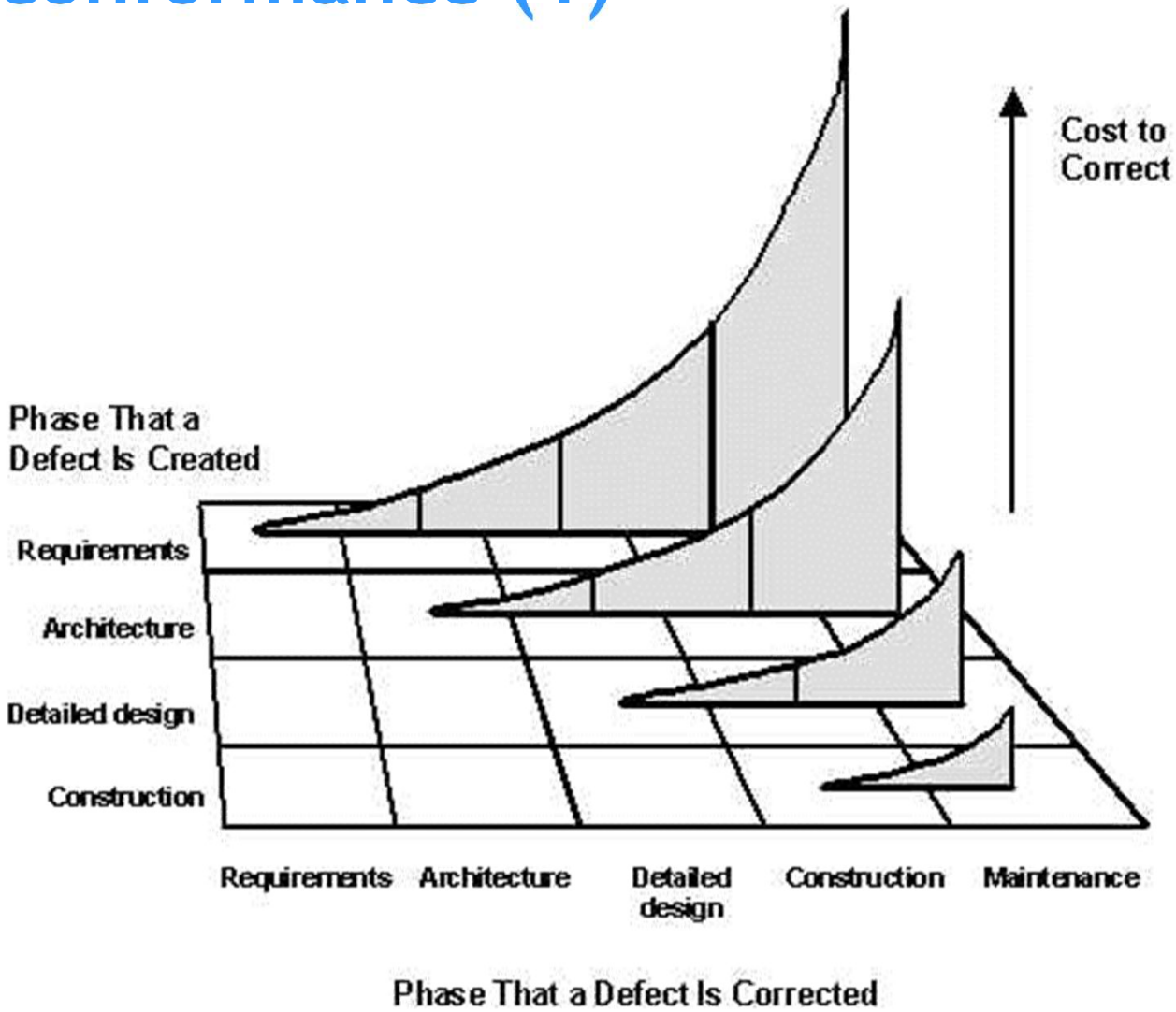
[source: "Software Testing", Ron Patton]

Quality costs
Costs of nonconformance (1)

If a defect is created in the early stages of the development life cycle and corrected in the later stages, it will be very expensive to correct.

If a defect is created and removed in the same phase, it won't be too expensive to fix.

The longer the bug remains, the more expensive it will be to fix.



Quality costs
Quality is free!?

- In his book "Quality is Free: The Art of Making Quality Certain", Philip Crosby argues that the costs of conformance plus the costs of nonconformance due to internal failures is (usually) less than the costs of nonconformance due to external failures

costs of conformance
+
costs of nonconformance due to internal failures

<

costs of nonconformance due to external failures

[source: Ron Patton, "Software testing“]

Which defects are more frequent?

IBM defect data:

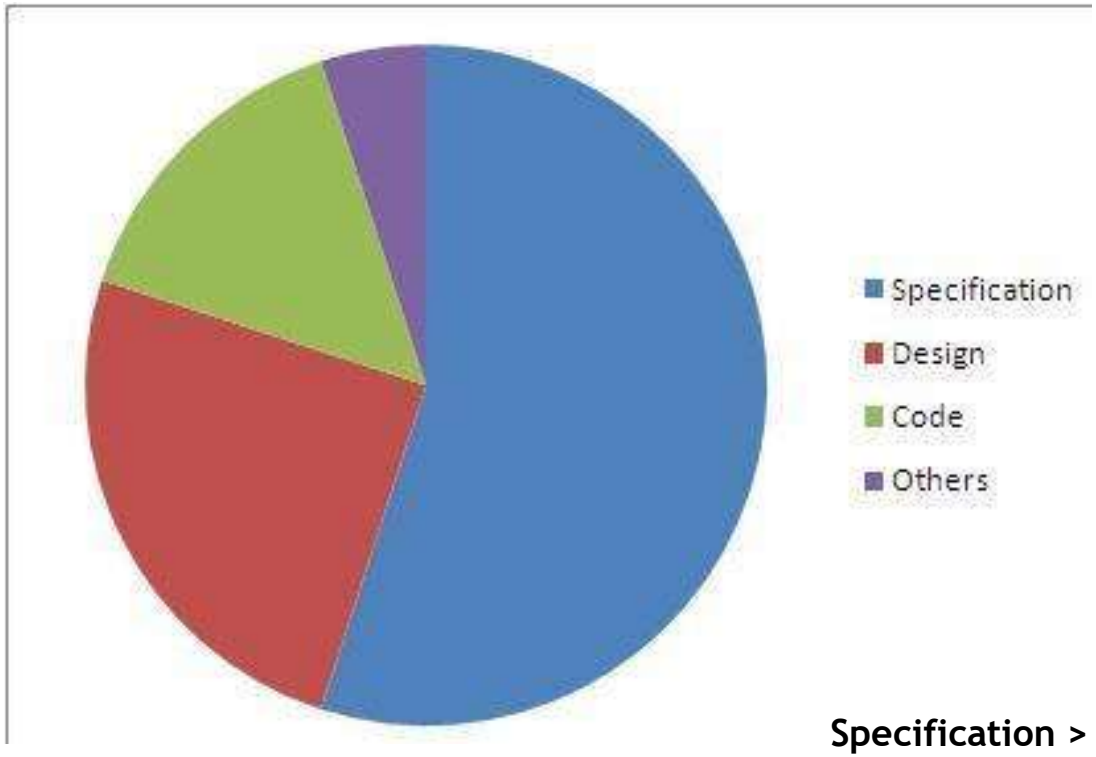
Classificação ODC (*)	Description	Frequency
Algorithm	"execução incorrecta ou em falta que pode ser corrigida sem ser necessário introduzir alterações arquitecturais no software"	43.4 %
Assignment	"valores incorrectamente atribuídos ou não atribuídos"	22.0 %
Checking	"validação de dados incorrecta ou expressões condicionais incorrectas"	17.5 %
Function	"falha que afecta uma quantidade considerável de código e refere-se a uma capacidade do software que está em falta ou construída incorrectamente"	8.7 %
Interface	"interacção incorrecta entre módulos/componentes"	8.2 %

(*) Orthogonal Defect Classification (ODC)

Useful for review check-lists and fault based testing!

[source:Henrique Madeira, Universidade de Coimbra]

Main sources of defects



Specification > Design > Code > Others

So,
take more time to get and understand specifications!

Agenda

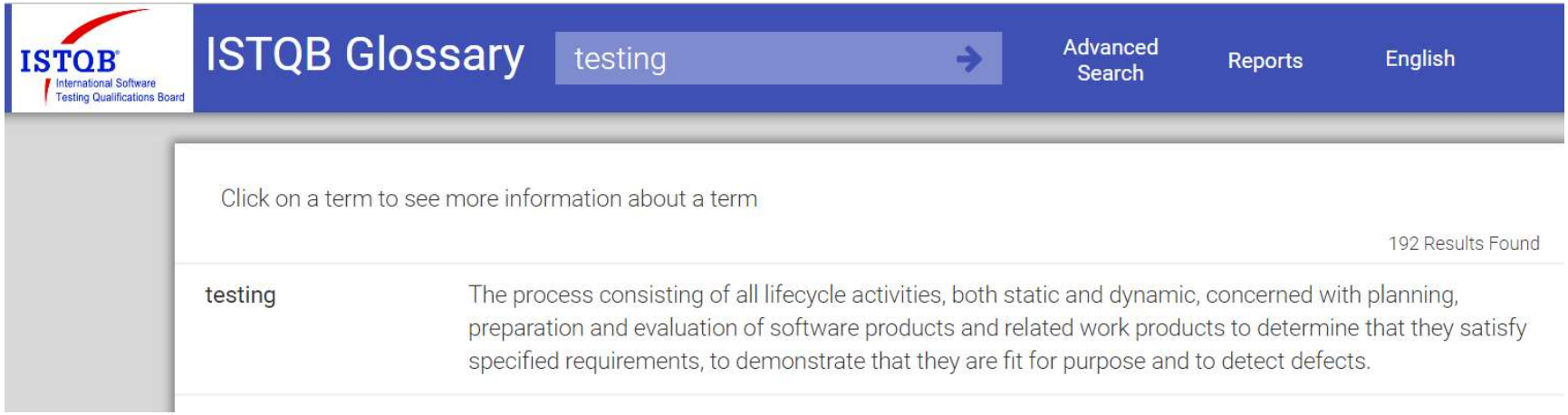
- Current status of software quality
- **Software Testing (introduction)**

Static vs dynamic

- **static testing:** Testing of a software development artifact, e.g., requirements, design or code, without execution of these artifacts, e.g., reviews or static analysis.
- **dynamic testing:** Testing that involves the execution of the software of a component or system.

[http://glossary.istqb.org/]

Software Testing - ISTQB



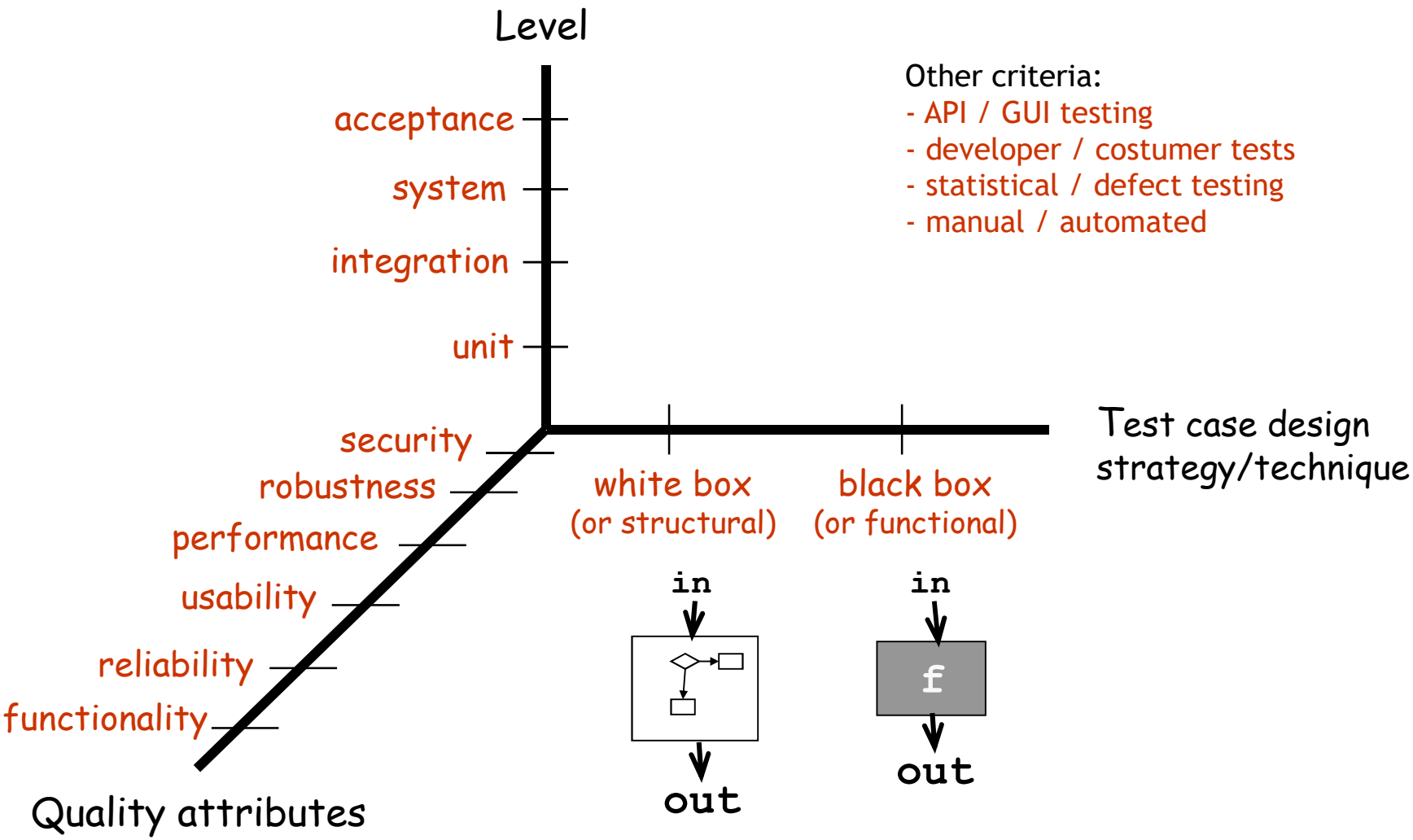
- The process consisting of all lifecycle activities, **both static and dynamic**, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

[http://glossary.istqb.org/]

Purpose of software testing

- **"Program testing can be used to show the presence of bugs, but never to show their absence!"** [Dijkstra, 1972]
  - … because exhaustive testing is usually impossible
- **The primary goal** of software testing is **to find failures**
  - "The goal of a software tester is to find "bugs", find them as early as possible, and make sure that they get fixed" [Ron Patton]
- **A secondarily goal** is to **increase the confidence** on the software correctness and to assess software quality
  - **statistical testing**: A test design technique in which a model of the statistical distribution of the input is used to construct representative test cases.

Test types



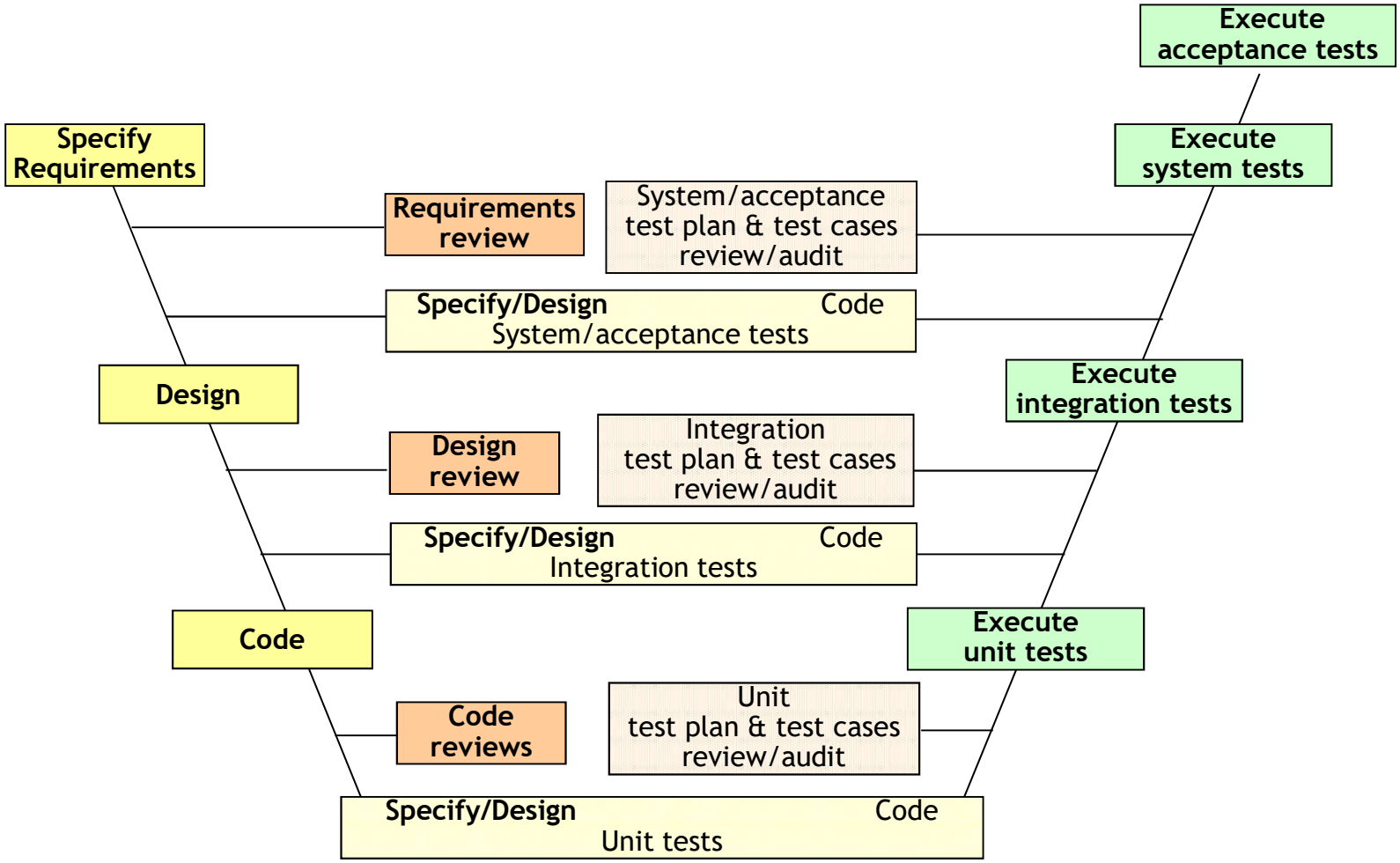
Other criteria:
- API / GUI testing
- developer / costumer tests
- statistical / defect testing
- manual / automated

Test Types

- Classified according to:
  - Level: acceptance, system, integration, unit
  - Quality attributes: functionality, reliability, usability, performance. Robustness, security,…
  - Test case design strategy: white box (or structural) and back box (or functional)
- Other criteria
  - API / GUI
  - Developer /customer tests
  - Statistical / defect testing
  - Manual /automated

V-model

Testing of the product is planned in parallel with a corresponding phase of development in V-model.



The extended V-model of software development [I.Burnstein]

Test levels – Unit Testing

- Testing of individual (hardware or) software units or groups of related units [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]
- Component testing: the testing of individual software components [http://glossary.istqb.org/]
- Usually API testing
- Usually the responsibility of the developer -> developer tests
- Tests are usually based on experience, specifications and code
- A principal goal is to detect functional and structural defects in the unit

Test levels – Integration testing

- Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them.
  - [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]
- Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
  - [http://glossary.istqb.org/]

Test levels – Integration testing

- Usually the responsibility of an independent test team (except sometimes in small projects)
- Tests are usually based on a system specification (technical specifications, designs)
- A principal goal is to detect defects that occur on the interfaces of units (interaction testing)

Test levels – System testing

- Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.
  - [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]
- Testing an integrated system to verify that it meets specified requirements.
  - [http://glossary.istqb.org/]

Test levels – System testing

- Usually GUI testing
- Usually the responsibility of an independent test team
- Tests are usually based on a requirements document
- The goal is to ensure that the system performs according to its requirements, by evaluating both functional behavior and quality requirements such as reliability, usability, performance and security

Test levels – Acceptance testing

- (1) Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. (2) Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component.
  - [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]
- Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.
  - [http://glossary.istqb.org/]

Test levels – Acceptance testing

- Usually the responsibility of the customer -> customer tests
- Tests are usually based on a requirements document or a user manual
- A principal goal is to check if customer requirements and expectations are met

TVVS - Software Testing Verification and Validation

Introduction
Acceptance Testing

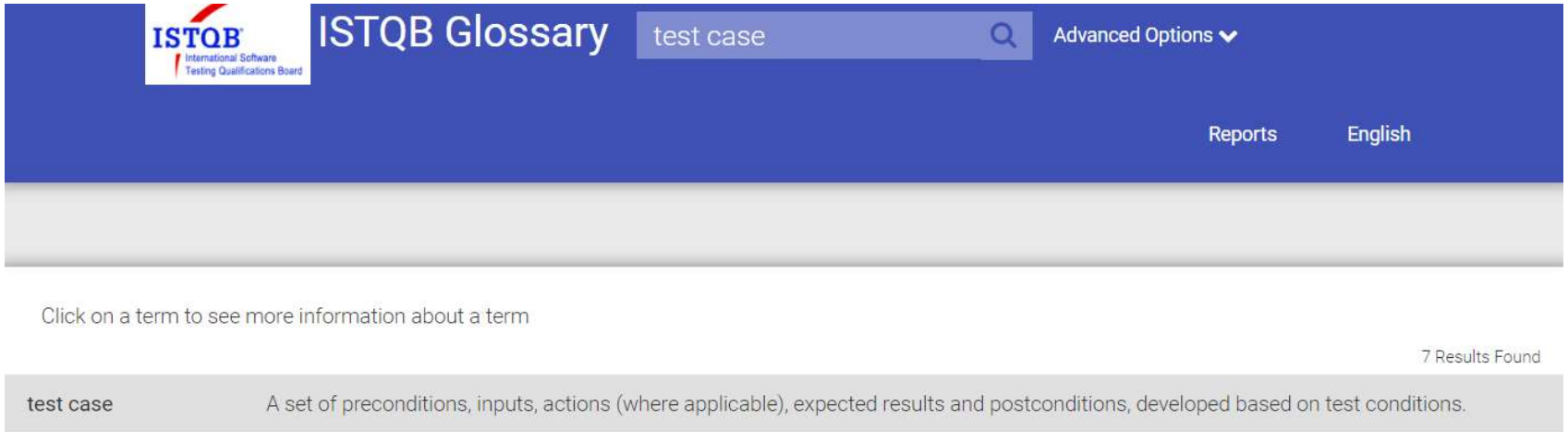
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- **Software Testing (some concepts)**
- Software Testing Process and Documentation
- Acceptance Testing
- Behavior Driven Development

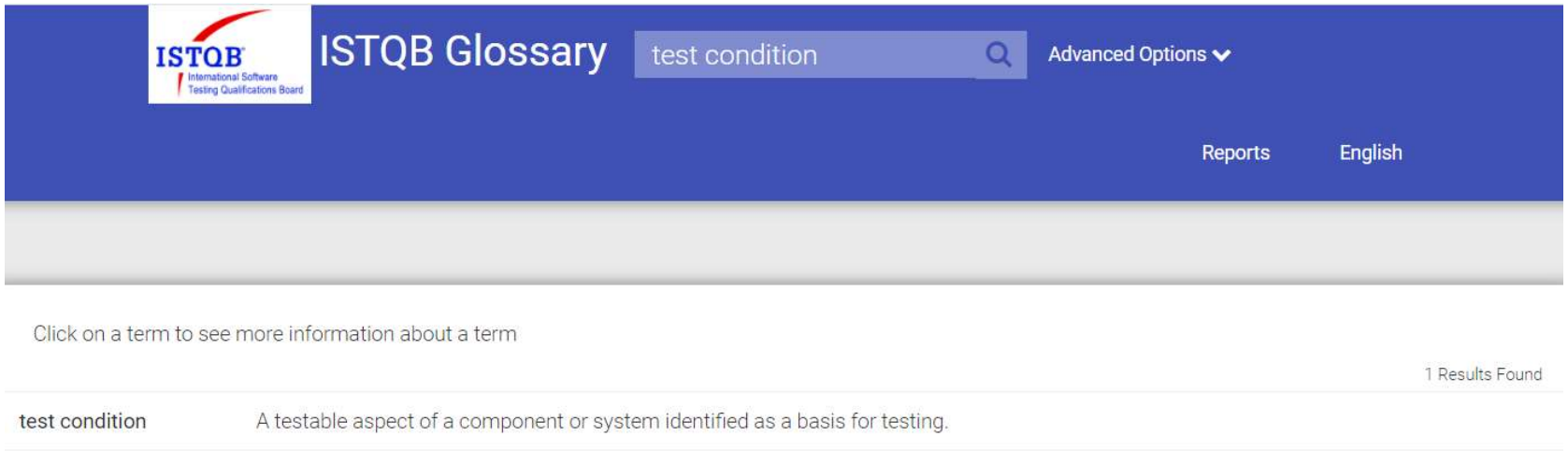
Test case



- A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

[http://glossary.istqb.org/]

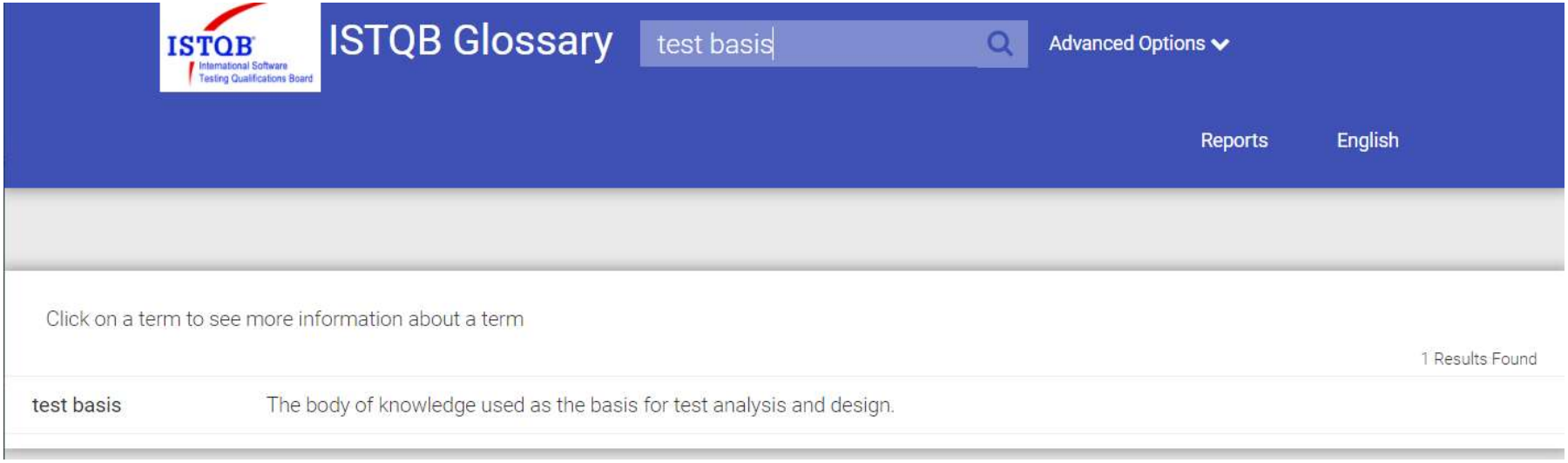
Test condition



- A testable aspect of a component or system identified as a basis for testing.

[http://glossary.istqb.org/]

Test basis



- The body of knowledge used as the basis for test analysis and design

[http://glossary.istqb.org/]

What is a good test case?

- Capability to find defects
  - Particularly defects with higher risk
  - Risk = frequency of failure * impact of failure ≈ cost (of post-release failure)
- Capability to exercise multiple aspects of the system under test
  - Reduces the number of test cases required and the overall cost
- Low cost
  - Development: specify, design, code
  - Execution (fast)
  - Result analysis: pass/fail analysis, defect localization
- Easy to maintain
  - Reduce whole life-cycle cost
  - Maintenance cost ≈ size of test artefacts

(See also: "What Is a Good Test Case?", Cem Kaner, Florida Institute of Technology, 2003)

Test cases

- The **best test cases** are the ones that
  - have higher probability of revealing defects,
  - particularly defects with higher impact
  - particularly defects that occur in the most frequently used features of the system

Test adequacy/coverage criteria

- **Adequacy criteria** - Criteria to decide if a given test suite is adequate, i.e., to give us "enough" confidence that "most" of the defects are revealed
  - Used in the evaluation and in the design/selection of test cases
  - In practice, reduced to coverage criteria
- **Coverage criteria**
  - Requirements/specification coverage (black-box)
    - At least one test case for each requirement/specification statement
  - Code coverage (white-box)
    - Control flow coverage (statement, decision, MC/DC coverage …)
    - Data flow coverage
  - Model coverage
    - State-transition coverage
    - Use case and scenario coverage
  - Fault coverage

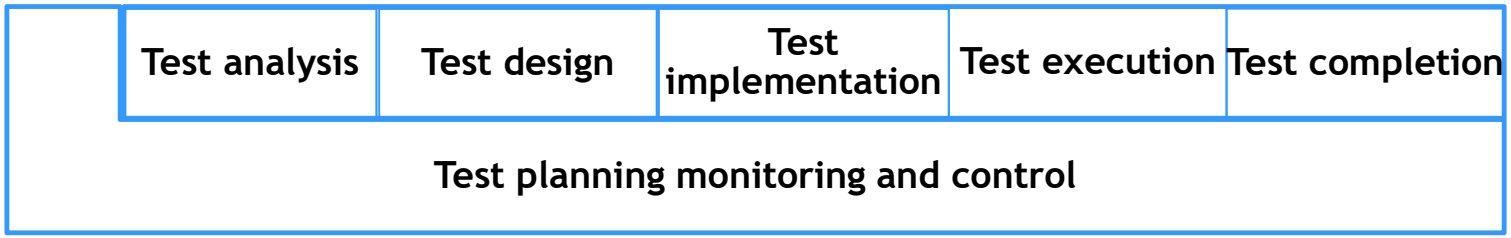
See also: "Software Unit Test Coverage and Adequacy", Hong Zhu et al, ACM Computing Surveys, December 1997

Agenda

- Software Testing (some concepts)
- **Software Testing Process and Documentation**
- Acceptance Testing
- Behavior Driven Development

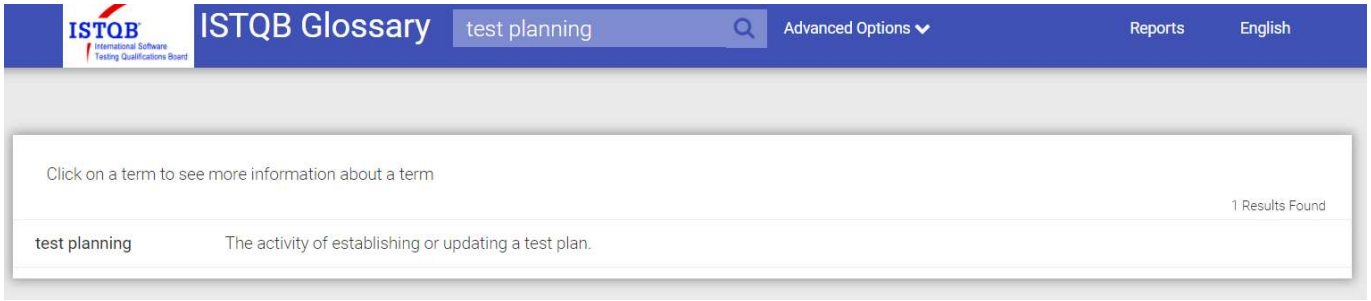
Test Activities and Tasks

- A test process consists of the following main groups of activities:
  - Test planning
  - Test monitoring and control
  - Test analysis
  - Test design
  - Test implementation
  - Test execution
  - Test completion



Test Planning

- The activity of establishing or updating a test plan.



- Test planning involves activities that define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context (e.g., specifying suitable test techniques and tasks, and formulating a test schedule for meeting a deadline).
- Test plans may be revisited based on feedback from monitoring and control activities.

Test Monitoring and Control

- Test monitoring involves the on-going comparison of actual progress against planned progress using any test monitoring metrics defined in the test plan.
- Test control involves taking actions necessary to meet the objectives of the test plan (which may be updated over time).
- Test monitoring and control are supported by the evaluation of exit criteria, which are referred to as the definition of done in some software development lifecycle models.

Test Monitoring and Control

- For example, the evaluation of exit criteria for test execution as part of a given test level may include:
  - Checking test results and logs against specified coverage criteria
  - Assessing the level of component or system quality based on test results and logs
  - Determining if more tests are needed (e.g., if tests originally intended to achieve a certain level of product risk coverage failed to do so, requiring additional tests to be written and executed)
- Test progress against the plan is communicated to stakeholders in test progress reports, including deviations from the plan and information to support any decision to stop testing

Test Analysis

- During test analysis, the **test basis** is analyzed to identify testable features and define associated test conditions. In other words, test analysis determines "**what to test**" in terms of measurable coverage criteria.

Test Analysis

- Test analysis includes the following major activities:
  - **Analyzing the test basis** appropriate to the test level being considered, for example, requirements specifications; design and implementation information; the implementation itself; risk analysis reports.
  - **Evaluating the test basis and test items** to identify defects of various types, such as: ambiguities; omissions; inconsistencies; inaccuracies; contradictions; superfluous statements
  - **Identifying features and sets of features to be tested**
  - **Defining and prioritizing test conditions** for each feature based on analysis of the test basis, and considering functional, non-functional, and structural characteristics, other business and technical factors, and levels of risks
  - **Capturing bi-directional traceability** between each element of the test basis and the associated test conditions

Test Design

- During test design, the test conditions are elaborated into high-level test cases, sets of high-level test cases, and other testware. So, test analysis answers the question "what to test?" while test design answers the question "**how to test?**"
- Test design includes the following major activities:
  - Designing and prioritizing test cases and sets of test cases
  - Identifying necessary test data to support test conditions and test cases
  - Designing the test environment and identifying any required infrastructure and tools
  - Capturing bi-directional traceability between the test basis, test conditions, and test cases
- The elaboration of test conditions into test cases and sets of test cases during test design often involves using test techniques

Test Implementation

- During test implementation, the testware necessary for test execution is created and/or completed, including sequencing the test cases into test procedures. So, test design answers the question "how to test?" while test implementation answers the question "**do we now have everything in place to run the tests?**"
- Test implementation includes the following major activities:
  - Developing and prioritizing test procedures, and, potentially, creating automated test scripts
  - Creating test suites from the test procedures and (if any) automated test scripts
  - Arranging the test suites within a test execution schedule in a way that results in efficient test execution
  - Building the test environment (including, potentially, test harnesses, service virtualization, simulators, and other infrastructure items) and verifying that everything needed has been set up correctly
  - Preparing test data and ensuring it is properly loaded in the test environment
  - Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test suites
- Test design and test implementation tasks are often combined.

Test Execution

- During test execution, test suites are run in accordance with the test execution schedule. Test execution includes the following major activities:
  - Recording the IDs and versions of the test item(s) or test object, test tool(s), and testware
  - Executing tests either manually or by using test execution tools
  - Comparing actual results with expected results
  - Analyzing anomalies to establish their likely causes (e.g., failures may occur due to defects in the code, but false positives also may occur
  - Reporting defects based on the failures observed
  - Logging the outcome of test execution (e.g., pass, fail, blocked)
  - Repeating test activities either as a result of action taken for an anomaly, or as part of the planned testing (e.g., execution of a corrected test, confirmation testing, and/or regression testing)
  - Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test results.

Test Completion

- Test completion activities **collect data from completed test activities to consolidate experience, testware, and any other relevant information**. Test completion activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), an Agile project iteration is finished, a test level is completed, or a maintenance release has been completed.
- Test completion includes the following major activities:
  - Checking whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution
  - Creating a test summary report to be communicated to stakeholders
  - Finalizing and archiving the test environment, the test data, the test infrastructure, and other testware for later reuse
  - Handing over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use
  - Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects
  - Using the information gathered to improve test process maturity

Standard: ISO/IEC/IEEE 29119-3

- The documents that are defined in ISO/IEC/IEEE 29119-3 are as follows[2][4]:
  - **Organizational Test Process Documentation:**
    - **Test Policy**
      - A high-level document describing the principles, approach and major objectives of the organization regarding testing.
    - **Organizational Test Strategy**
      - [***Test strategy***: Documentation aligned with the test policy that describes the generic requirements for testing and details how to perform testing within an organization.]
  - **Test Management Process Documentation:**
    - **Test Plan** (including a Test Strategy):
      - Documentation describing the test objectives to be achieved and the means and the schedule for achieving them, organized to coordinate testing activities [ISTQB].
    - **Test Status**
      - [***Test progress report***: A type of test report produced at regular intervals about the progress of test activities against a baseline, risks, and alternatives requiring a decision.]
    - **Test Completion**
      - [***Test summary report***: A type of test report produced at completion milestones that provides an evaluation of the corresponding test items against exit criteria]

[https://en.wikipedia.org/wiki/ISO/IEC_29119]

Standard: ISO/IEC/IEEE 29119-3

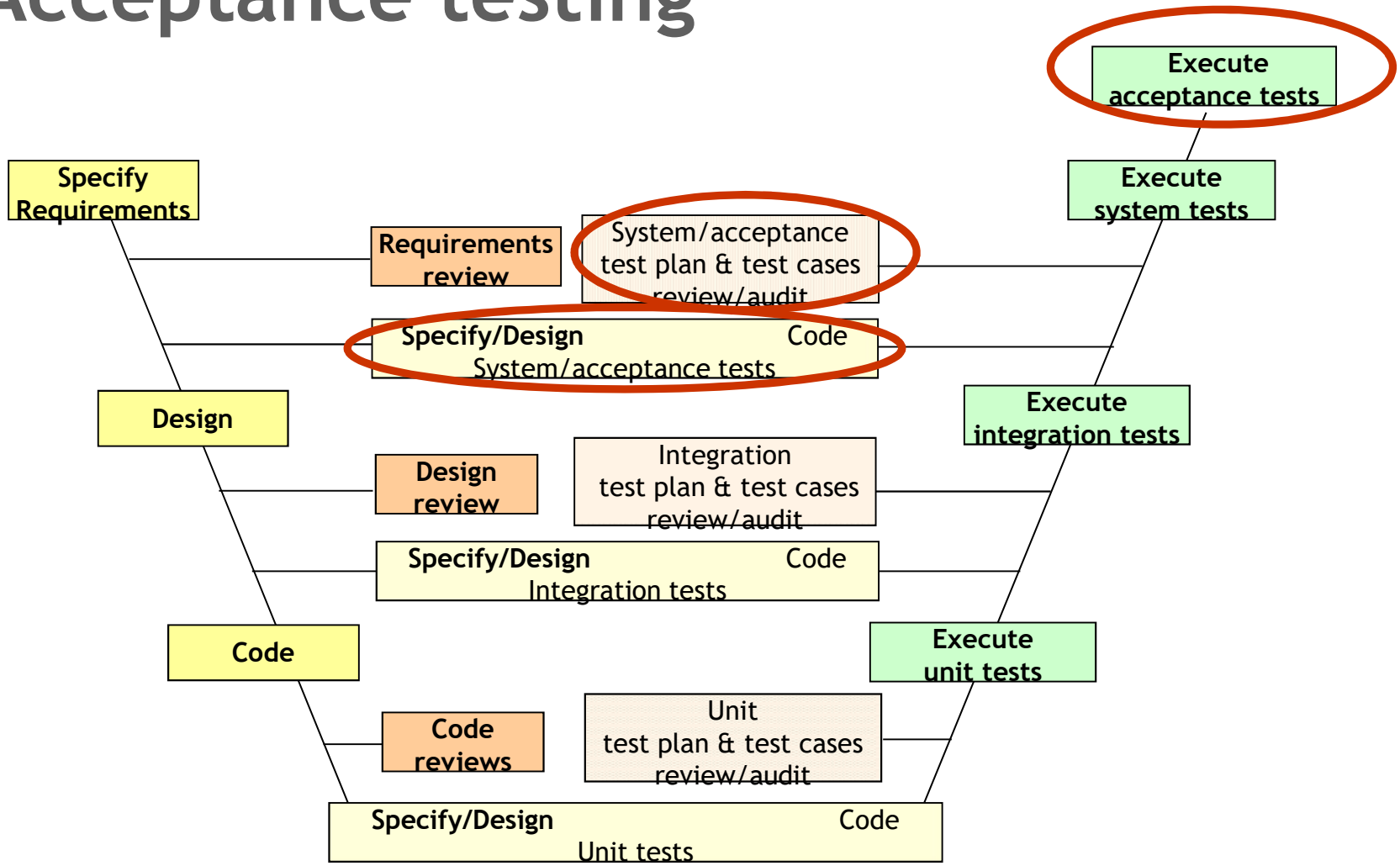
- **Dynamic Test Process Documentation:**
  - Test Design Specification:
    - Documentation specifying the features to be tested and their corresponding test conditions.
  - Test Case Specification:
    - Documentation of a set of one or more test cases.
  - Test Procedure Specification
    - [***test procedure***: A sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution.]
  - Test Data Requirements
  - Test Data Readiness Report
  - Test Environment Requirements
  - Test Environment Readiness Report
  - Actual Results
  - Test Result
  - Test Execution Log
    - [***Test log***: A chronological record of relevant details about the execution of tests.]
  - Test Incident Report
    - Documentation of the occurrence, nature, and status of an incident.

[https://en.wikipedia.org/wiki/ISO/IEC_29119]

Agenda

- Software Testing (some concepts)
- Software Testing Process and Documentation
- **Acceptance Testing**
- Behavior Driven Development

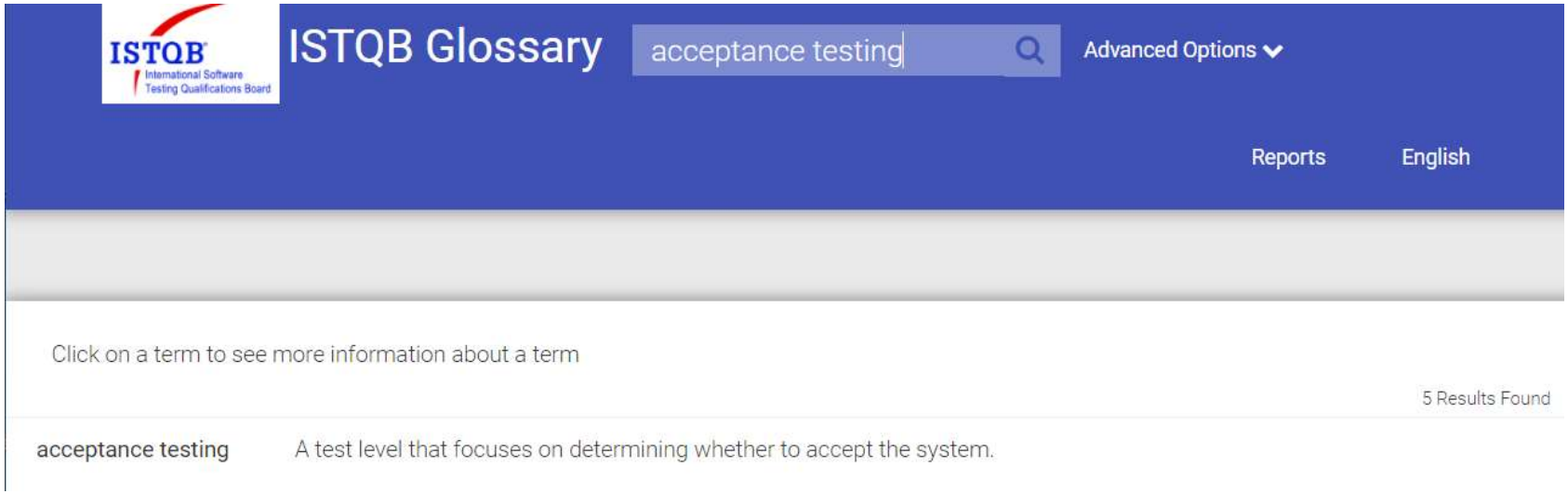
Acceptance testing



The extended V-model of software development [I.Burnstein]

Acceptance testing

- **acceptance testing:** A test level that focuses on determining whether to accept the system.



Acceptance testing

- Different types of acceptance testing:
  - Contractual acceptance testing
    - A type of acceptance testing performed to verify whether a system satisfies its contractual requirements.
  - Operational acceptance testing
    - A type of acceptance testing performed to determine if operations and/or systems administration staff can accept a system.
  - Regulatory acceptance testing
    - A type of acceptance testing performed to verify whether a system conforms to relevant laws, policies and regulations.
  - User acceptance testing
    - A type of acceptance testing performed to determine if intended users accept the system.

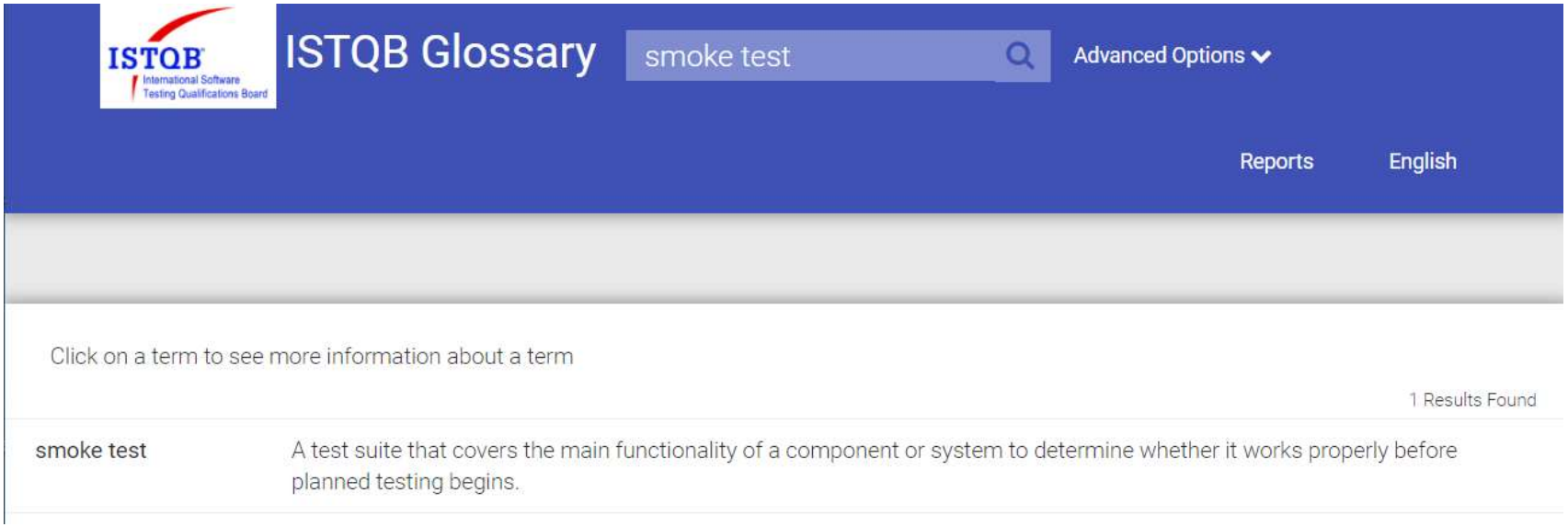
Acceptance testing

- **Alpha testing**: A type of acceptance testing performed in the developer's test environment by roles outside the development organization.
- **Beta testing**: A type of acceptance testing performed at an external site to the developer's test environment by roles outside the development organization.

[ISTQB]

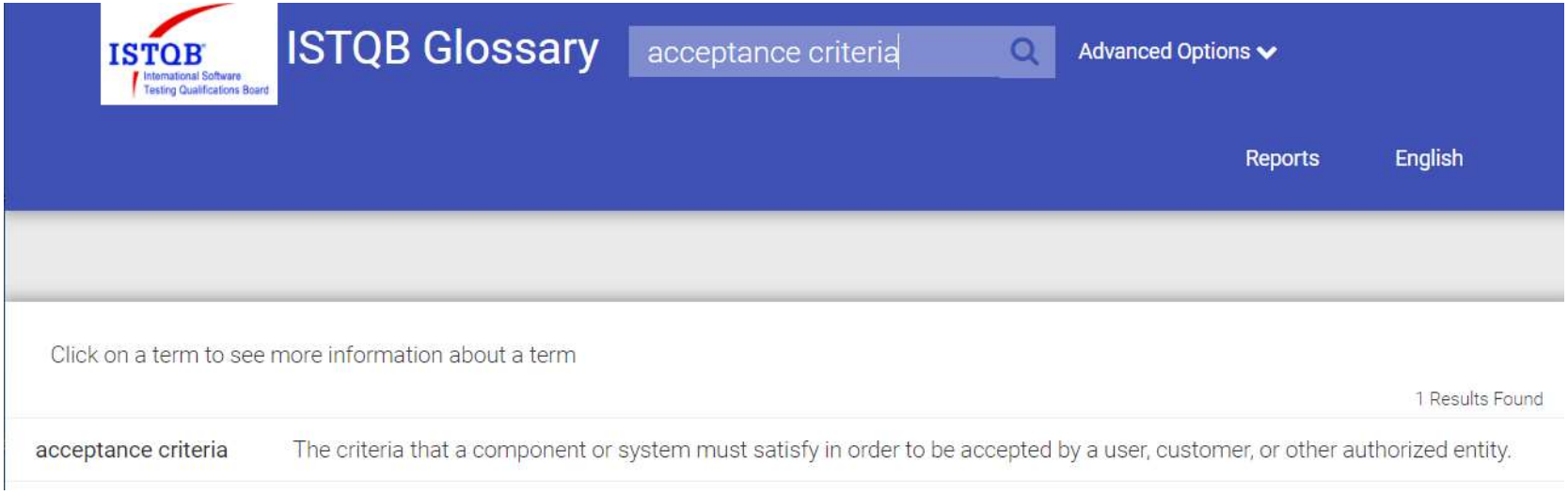
Smoke test

- A test suite that covers the main functionality of a component or system to determine whether it works properly before planned testing begins.



Acceptance criteria

- **acceptance criteria:** The criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity.



Acceptance criteria

The acceptance criteria are defined on the basis of the following attributes:

- **Functional Correctness and Completeness**
- **Accuracy**
- **Data Integrity**
- **Data Conversion**
- **Backup and Recovery**
- **Competitive Edge**
- **Usability**
- **Performance**
- **Start-up Time**
- **Stress**
- **Reliability and Availability**
- **Maintainability and Serviceability**
- **Robustness**
- **Timeliness**
- **Confidentiality and Availability**
- **Compliance**
- **Installability and Upgradability**
- **Scalability**
- **Documentation**

Acceptance Testing in eXtreme Programming

- In XP framework the user stories may be used as acceptance criteria
- The user stories are written by the customer as things that the system needs to do for them
- Several acceptance tests may be created to verify the user story has been correctly implemented
- The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results
- A story is incomplete until it passes its associated acceptance tests
- Ideally, acceptance tests should be automated, either using the unit testing framework, before coding
- The acceptance tests take on the role of regression tests

Acceptance testing tools

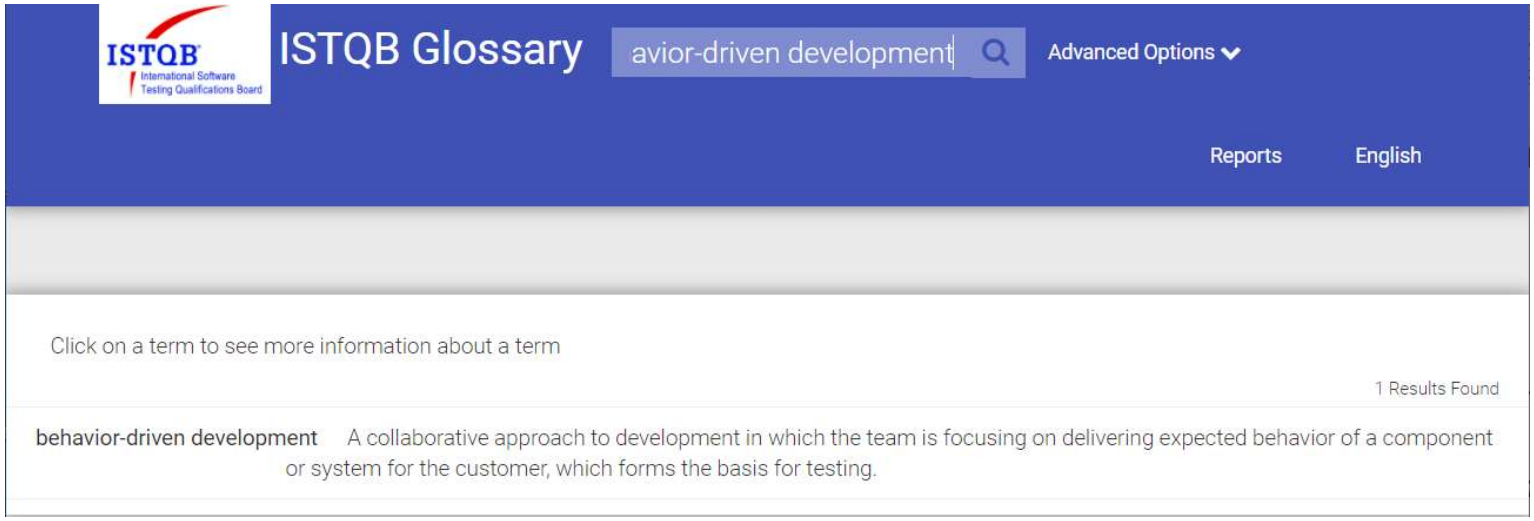
- Framework for Integrated Test (*Fit*)
- FitNesse
- Behat
- Lettuce
- Cucumber
- …

Agenda

- Software Testing (some concepts)
- Software Testing Process and Documentation
- Acceptance Testing
- **Behavior Driven Development**

BDD – Behavior-Driven development

- A collaborative approach to development in which the team is focusing on delivering expected behavior of a component or system for the customer, which forms the basis for testing.



BDD – Behavior-Driven Development

- Behavior Driven testing is an extension of TDD. Like in TDD, in BDD we also write tests first and then add application code. The major difference that we get to see here are
  - *Tests are written in plain descriptive English type grammar*
  - *Tests are explained as behavior of application and are more user focused*
  - *Using examples to clarify requirements*
- This difference brings in the need to have a language which can define behavior in an understandable format.

BDD – Behavior-Driven Development

- Shifting from thinking in “tests” to thinking in “behavior”
- Collaboration between Business stakeholders, Business Analysts, QA Team and developers
- Ubiquitous language, it is easy to describe
- Driven by Business Value
- Extends Test Driven Development (TDD) by utilizing natural language that non technical stakeholders can understand
- BDD frameworks such as Cucumber or JBehave are an enabler, acting a “bridge” between Business & Technical Language

BDD – Behavior-Driven Development

- Features
  - Who's using the system?
  - What are they doing?
  - Why do they care?
    - As a <role>
    - I want <feature>
    - So that <business value>
- Features are defined by one or more scenarios
  - Sequence of steps thru the feature that exercises on path
- Use BDD style – given-when-then
    - Scenario: <description>
    - <step 1>
    - …
    - <step 2>

BDD - Behavior-Driven Development

FEATURES

What are they doing?

Who's Using the System

Feature: Perform and Interview

As an Interviewer
I want to be able to record details about an interview
So that a Resource can be evaluated

Why do they care?

BDD – Behavior-Driven Development

SCENARIOS

- **Given** - Sets up preconditions, or context, for the scenario
- **When** - The action, or behavior, that we're focused on
- **Then** - Checks post-conditions and verifies that the right thing happened in the When stage

```
Scenario: Assign a candidate to an opportunity
  Given a user is logged in as "joe"
  And seeded users are created
  And seeded opportunities are created
  And I am on NewOpportunity1's opportunity edit page
  When I select "smith5" from "opportunity_resources_"
  Then I should see "success"
```

BDD (example: CUCUMBER - GHERKIN)

Feature: Google Searching

As a web surfer, I want to search Google, so that I can learn new things.

Scenario Outline: Simple Google searches

Given a web browser is on the Google page

When the search phrase "<phrase>" is entered

Then results for "<phrase>" are shown

And the related results include "<related>"

Examples: Animals

phrase	related
panda	Panda Express
elephant	Elephant Man

[https://cucumber.io/docs/gherkin/reference/]

TVVS – Software Testing Verification and Validation

Unit Testing

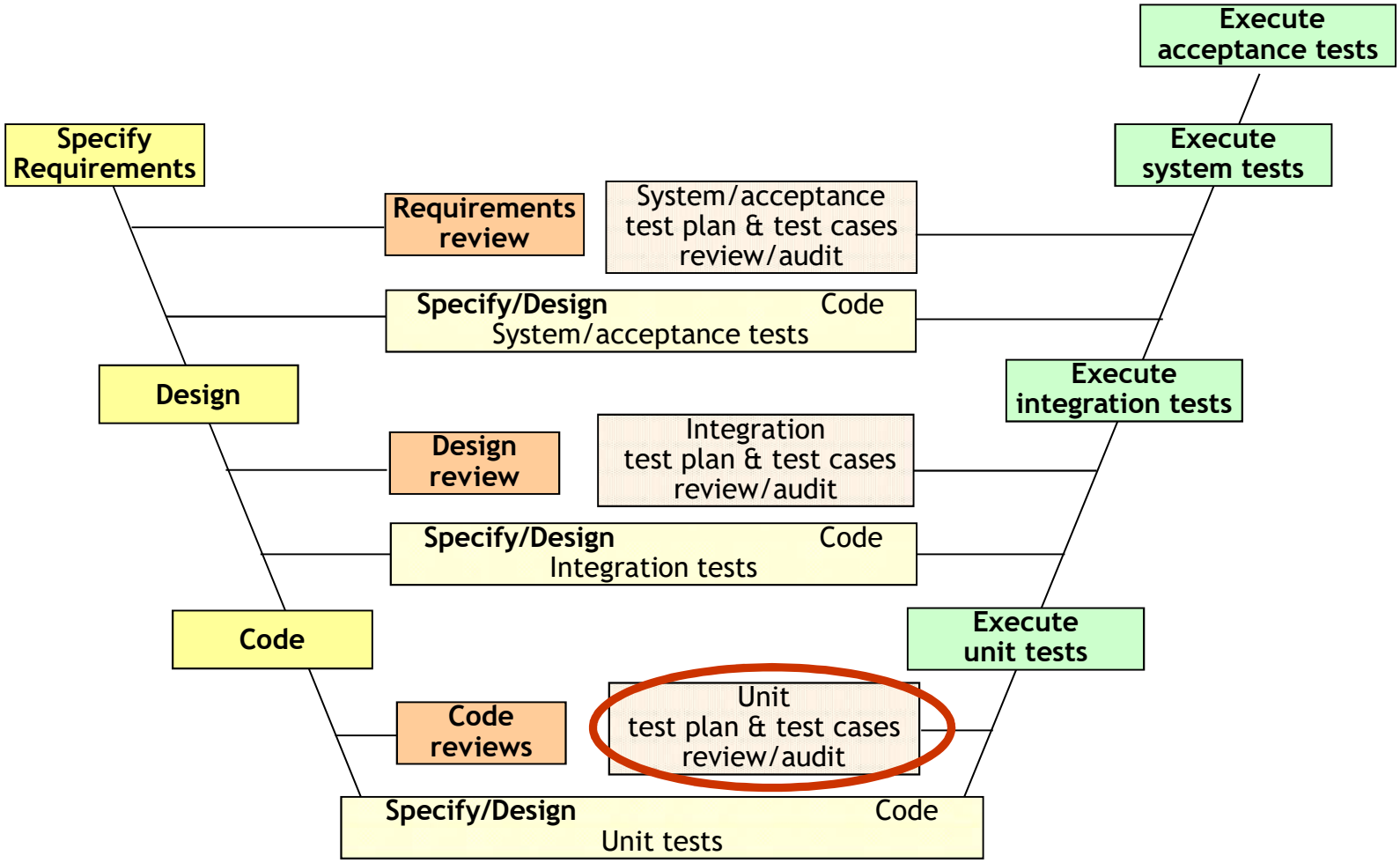
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- **Unit testing**
- Mock objects
- Unit testing within the software development process
  - TDD – Test Driven Development
  - Extreme programming

Unit testing



The extended V-model of software development [I.Burnstein]

Unit testing – definition

- **Unit testing**: Testing of individual hardware or software units or groups of related units [IEEE 90].

- **Unit testing** is a development procedure where programmers create tests as they develop software. The tests are simple short tests that test functionality of a particular unit or module of their code, such as a class or function.

- **Unit testing** is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. This testing mode is a component of **Extreme Programming (XP)**, a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

Benefits of unit testing

- Facilitates change
  - Allows **refactor** code at a later date and make sure the module still works correctly
- Simplifies integration
  - Helps to eliminate uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, **integration testing** becomes much easier
- Documentation
  - Provides a sort of **living documentation** of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API

Unit testing

- The goal of unit testing is to exercise ONE unit at a time
- Unit tests vs. Integration tests
  - Unit tests do not touch real dependencies; Integration tests touch concrete dependencies
  - Unit tests are fast; Integration tests do not need to be
  - Unit tests do not touch databases, web services, etc.; Integration tests do
- What if an Unit depends on other things?
  - A network? A database? A servlet engine?
  - Other parts of the system?
  - We do not want to initialize lots of components just to get the right context for one test to run.

Separation of interface from implementation

- Because some classes may have references to other classes, testing a class can frequently spill over into testing another class.
  - Ex.: classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database.
- This is a mistake
  - unit test should never go outside of its own class boundary.
- Abstract an interface around database connection and implement it with your own **mock objects**
  - the independent unit can be more thoroughly tested
  - this results in a higher quality unit that is also more maintainable.

“Good” unit tests

- What properties a unit test should have?
  - It is automated and repeatable
  - It is easy to implement
  - Once it is written, it stays on for the future
  - Anyone can run it
  - It runs at the push of a button
  - It runs quickly
- Unit tests are code …
  - Maintanability
  - Readability
  - Correctness
  - Documentation
  - …

“Good” unit tests

- **Runs fast, runs fast, runs fast**. If the tests are slow, they will not be run often.
- **Separates or simulates environmental dependencies such as databases, file systems, networks, queues, and so on**. Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
- **Is very limited in scope**. If the test fails, it is obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It is important to only test one thing in a single test.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

“Good” unit tests (2)

- **Runs and passes in isolation**. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.
- **Often uses stubs and mock objects**. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- **Clearly reveals its intention**. Another developer can look at the test and understand what is expected of the production code.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

Six rules of unit testing

- Write the test first
- Never write a test that succeeds the first time
- Start with the null case, or something that doesn't work
- Don't be afraid of doing something trivial to make the test work
- Loose coupling and testability go hand in hand
- Use mock objects

[http://radio.weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html]

Unit testing – best practices

- Keep unit tests small and fast
  - Ideally the entire test suite should be executed before every code check in.
- Unit tests should be fully automated and non-interactive
  - The test suite is noemally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are nto proper unit tests.
- Make unit tests simple to run
  - Configure the development environment so that single tests and test suites can be run by single command or a one button click

Unit testing – best practices

- Keep testing at unit level
  - Unit testing is about testing units. Units should be tested in isolation. Avoid temtation to test an entire workflow using a unit testing framework, as such tests are slow and hard to maintain. Workflow tetsing may have its place, but it is not unit testing and it must be set up and executed independently
- Start of simple
  - One simple test is infinitely better than no tests at all. A simple test class will verify the presence and corerctness of both the build environment, the unit testing environment, the execution environment and the coverage analysis tool, etc.

Unit testing – best practices

- Keep tests independent
  - To ensure testing robusteness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed
- Name tests properly
  - Make sure each test method test one distinct feature of the unit being tested and name the test methods accordingly. The typical naming convention is test[what] such as testSaveAs(), testAddListener(), testDeleteProperty(), etc.

Unit testing – best practices

- Keep tests close to the unit being tested
  - If you are testing a class Foo, the test class should be called FooTest and keep in the same package (directory) as Foo.
- Think black-box
  - Act as a 3rd party class consumer, and test if the class fulfills its requirements. As try to tear it apart.
- Think white-box
  - After all, the test programmer also wrote the class being tested, and extra effort should be put into testing the must complex logic.

Unit testing – best practices

- Test the trivial cases too
  - It is sometimes recommended that all non-trivial cases should be tested and that trivial methods like simple setters and getters can be omitted. However, there are several reasons why trivial cases should be tested too:
    - Trivial is hard to define.It may mean different things to different people.
    - The trivial cases can contain errors too, often as a result of copy-paste operations
- Focus on execution coverage
  - Ensure that the code is actually executed on some input parameters
- Cover boundary cases
  - Make sure the parameter boundary cases are covered, e.g., test negatives, 0, positive, smallest, larger, etc

Unit testing – best practices

- Provide a random generator
  - When boundary are covered, you may use random generated parameters.
- Use explicit asserts
  - Always prefer assertEquals(a,b) to assertTrue(a==b) as the fomer gives more useful information
- Provide negative tests
  - Negative tests intentionally misuse the code and verify robusteness and appropriate error handling

Unit testing – best practices

- Design code with testing in mind
  - Writing and maintaining unit tests are costly, so reducing cyclomatic complexity in the code are ways to reduce this cost
- Know the cost of testing
  - Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%
- Prioritize testing
  - If there is not enough resources to test all parts you should establish a priopitization

Unit testing – best practices

- Prepare test code for failures
  - If a test fails, the remaining tests will not be exected. Always prepare for test failure so that the failure of a single test does not bring down the entire test suite execution
- Write tests to reproduce bugs
  - When a bug is reported, write a test to reproduce the bug (i.e., failing test) and use this test as a success criteria when fixing the code
- Know the limitations
  - Unit tests can never prove the correcteness of code.
- …
- …

Limitations of unit testing

- Testing, in general, cannot be expected to catch every error in the program. Unit tests can only show the presence of errors; it cannot show the absence of errors.
- It only tests the functionality of the units themselves.
- It may not catch integration errors, performance problems, or other system-wide issues.
- Unit testing is more effective if it is used in conjunction with other software testing activities.

Limitations of unit testing

- Software testing is a combinatorial problem.
  - For example, every **boolean decision** statement requires at least **two tests**:
    - one with an outcome of "true" and one with an outcome of "false".
  - As a result, for **every line of code** written, programmers often need **3 to 5 lines of test code**. Therefore, it is unrealistic to test all possible input combinations for any non-trivial piece of software without an automated characterization test generation tool such as JUnit Factory used with Java code or many of the tools.

Limitations of unit testing

- To obtain the intended benefits from unit testing use of a **version control system** is essential
  - If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.
- It is also essential to **implement a sustainable process** for ensuring that test case failures are reviewed daily and addressed immediately
  - If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite — increasing false positives and reducing the effectiveness of the test suite.

Unit testing frameworks

- Help simplify the process of unit testing
  - Log test cases that fail
  - Automatically flag and report in a summary these failed test cases.
  - Depending upon the severity of a failure, the framework may halt subsequent testing.
- Examples
  - JUnit
  - JTest
  - NUnit
  - MbUnit
  - …

[http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks]

Setup and teardown (1)

- You want to avoid duplicated code when several tests share the same initialization and cleanup code.
- … before and after each test method inside a test class
  - **JUnit**: Use the setUp( ) and tearDown( ) methods. Both of these methods are part of the junit.framework.TestCase class.
  - **NUnit**: build methods and annotate them with SetUp and TearDown tags

Setup and teardown (2)

- …. One-time set up and tear down
  - You want to run some setup code one time and then run several tests. You only want to run your cleanup code after all of the tests are finished, ex.: establish a database connection.
  - **JUnit**: Use the junit.extensions.TestSetup class to define **test suites**
    - Pass a TestSuite to the TestSetup constructor. This means that TestSetup's setUp( ) method is called once before the entire suite, and tearDown( ) is called once afterwards.

      TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {…}
  - **NUnit**: methods annotated with TestFixtureSetup and TestFixtureTearDown attributes.

Agenda

- Unit testing
- **Mock objects**
- Unit testing within the software development process
  - TDD – Test Driven Development
  - Extreme programming

Mock objects

- Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.
- Useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:
  - supplies non-deterministic results (e.g., the current time or the current temperature);
  - has states that are difficult to create or reproduce (e.g., a network error);
  - is slow (e.g., a complete database, which would have to be initialized before the test);
  - does not yet exist or may change behavior;
  - would have to include information and methods exclusively for testing purposes (and not for its actual task).

Mock objects (example)

- An alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the **test must wait** until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Steps to use mocks for testing

1. Use an interface to describe the object
2. Implement the interface for production code
3. Implement the interface in a mock object for testing

Note: the code under test only refers to the object by its interface, so it have no idea whether it is using the real object or the mock object

Example: real and mock object

```
public interface Environmental {
    public long getTime();
    // other methods omitted …
}
```

```
public class SystemEnvironment implements Environmental {
    public long getTime() {
        return System.getCurrentTimeMillis();
    }
    // other methods
}
```

```
public class MockSystemEnvironment implements Environmental {
    private long currrentTime;
    public void setTime(long aTime) { currentTime = aTime; }
    public long getTime() {
        return currentTime;
    }
    private Boolean bellRung = false;
    public void ringBell(string s) { bellRung = true;}
    public void resetBell() { bellRung = false;}
    public Boolean wasBellRung() {return bellRung;}
}
```

Example: real and mock object

```
public Timer {
    private Environmental env;
    Timer(Environmental env) { this.env = env; …}
    public void reminder() {
        long t = env.getTime();
        if (laterThanSpm(t)) env.ringBell("ringing.wav");
    }
}
```

```
public TestClassUnderTest {
    @Test
    public void testACaseOfMethodUnderTest() {
        MockSystemEnvironment env = new MockSystemEnvironment();
        Calendar cal = Calendar.getInstance();
        cal.set…. // to the 16:59:00, December 2012
        long t = cal.getTimeMillis();
        env.setTime(t);
        Timer timer = new Timer(env);

        timer.reminder();
        assertFalse(env.bellWasRung());  // too early to ring

        t += (5*60*100); // advance the timer 5 minutes
        env.setTime(t);
        env.resetBell(t);
        timer.reminder();
        assertTrue(env.bellWasRung());
    }
}
```

Mock Frameworks

- Facilitates the construction of Mock Objects
- Allow the construction based on
  - Interfaces
  - Classes
- Allow to define expectations
  - Number of method calls
  - Return values
  - Values of the parameters
  - Throw exceptions
- Integrate with JUnit (Java), NUnit (.Net), etc.
- Usually provide extensible APIs

Mock Frameworks

- Java
  - jMock -- http://jmock.org
  - Mockito -- http://code.google.com/p/mockito
  - rMock – http://rmock.sourceforge.net
  - MockCreator – http://mockcreator.sourceforge.net
  - MockLib – http://mocklib.sourceforge.net
- C#
  - MockLib – http://sourceforge.net/projects/mocklib
  - Rhino Mocks – http://www.ayende.com
  - Nmock – http://nmock.org
- Ruby
  - Mocha – http://mochs.rubyforge.org
  - Rspec – http://rspec.rubyforge.org
- …

Mockito

- Create
  - Mockito supports the creation of mock objects with the static mock() method call. It also supports the creation of mock objects based on the @Mock annotation
    - mathObj = mock(Math.class); //create math object
- Configure
  - You can use when(…).thenReturn(…) can be used to specify a condition and a return values for this condition
    - When(mathObj.add(1,2)),thenReturn(3); // configure it to return 3 when arguments passed are 1, 2
- Exception
  - when(mathObj.div(anyInt(),eq(0))).thenThrow(new ArithmeticException());

  //configure it to return exception when denominator is zero

Mockito

- Verify
  - Mockito keeps track of the method calls and their parameters to the mock objet. You can use the verify() method on the mock object to verify that the specified conditions are met, i.e., that a method has been called with certain parameters

```
// verify whether add method is tested with arguments 1,2
Verify(mathObj).add(eq(1), eq(2));
```

Example with jMock

Class does not exist

```
public class Convert {
    public static double getTemperatureInCelcius(City city) {
        double tempF;
        Sensor sensor = city.getSensor();
        tempF = sensor.getCurrentTemperature();

        //Convert Fahreneit to Celsius degrees
        return (tempF-32.0) * (5.0/9.0);
    }
}
```

Non deterministic result

```
public interface City { Sensor getSensor(); }
```

```
public interface Sensor { double getCurrentTemperature(); }
```

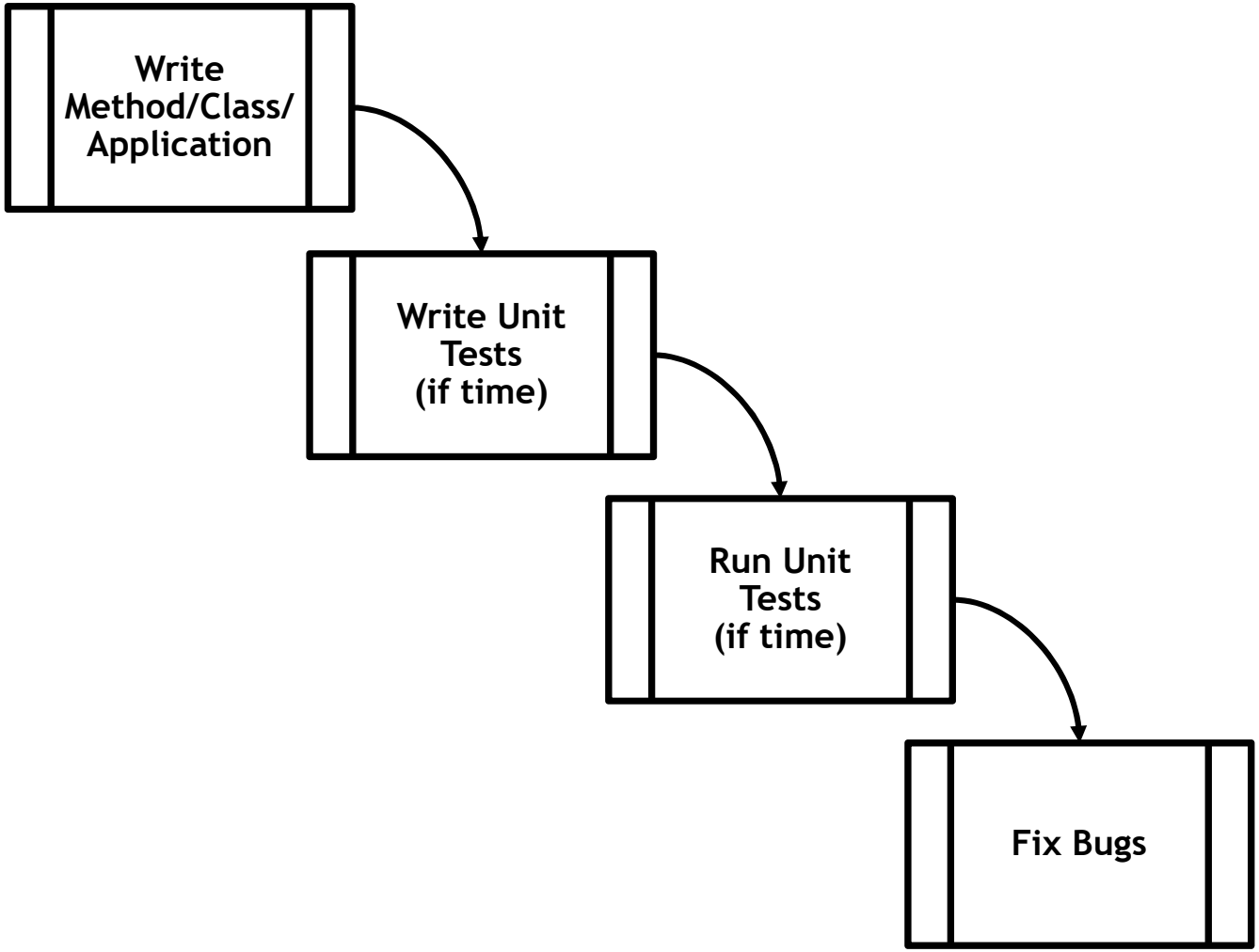
Example with jMock

```
import org.jmock.integration.junit3.MockObjectTestCase;
import org.jmock.Expectations;

public class JMockTestConvert extends MockObjectTestCase {
  // set up
  final City mockCity = mock(City.class);
  final Sensor mockSensor = mock(Sensor.class);

  public void testGetTemperatureInCelcius() {
   // expectations
   checking(new Expectations() {{
    one(mockCity).getSensor(); will(returnValue(mockSensor));
    one(mockSensor).getCurrentTemperature(); will(returnValue(72.5));
   }});
   // execute
   double result = Convert.getTemperatureInCelcius(mockCity);
   // test
   assertEquals(22.5, result);
  }
}
```

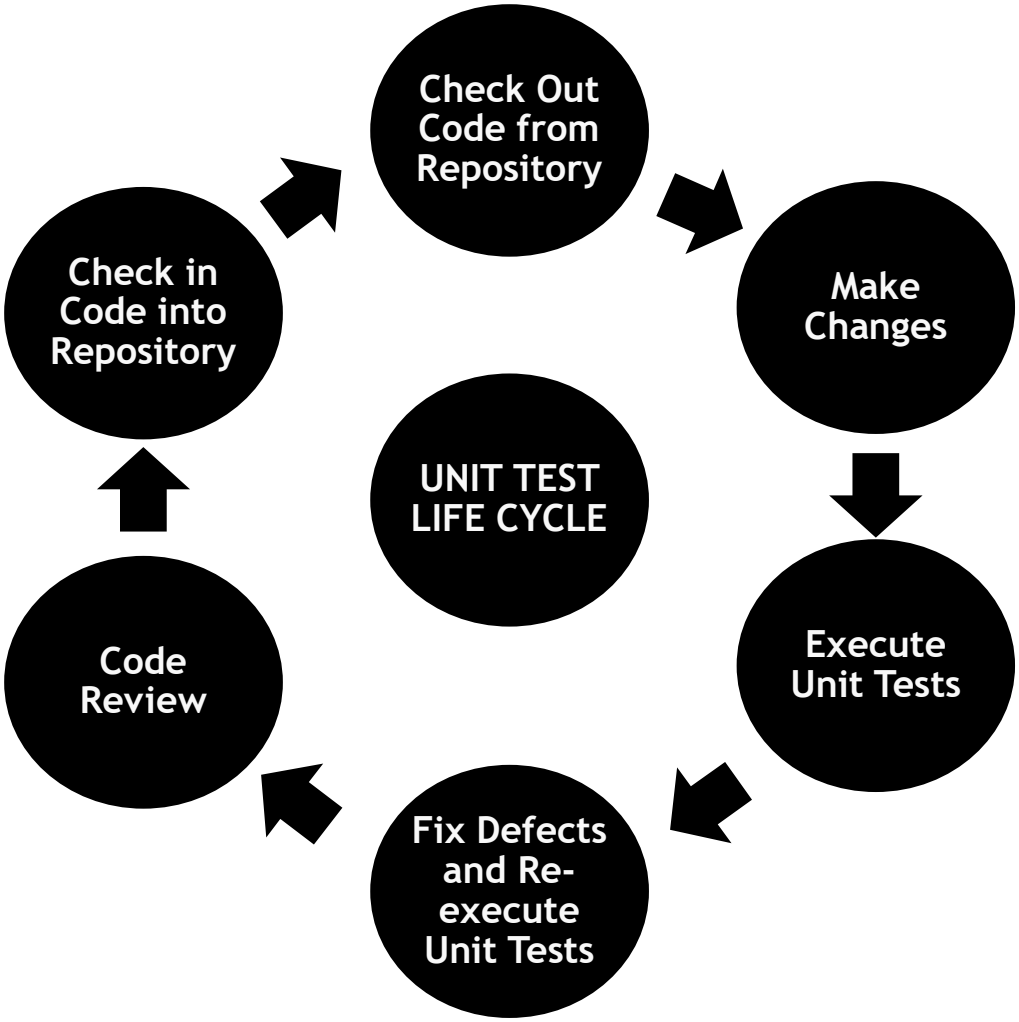
Agenda

- Unit testing
- Mock objects
- **Unit testing within the software development process**
  - **TDD – Test Driven Development**
  - **Extreme programming**

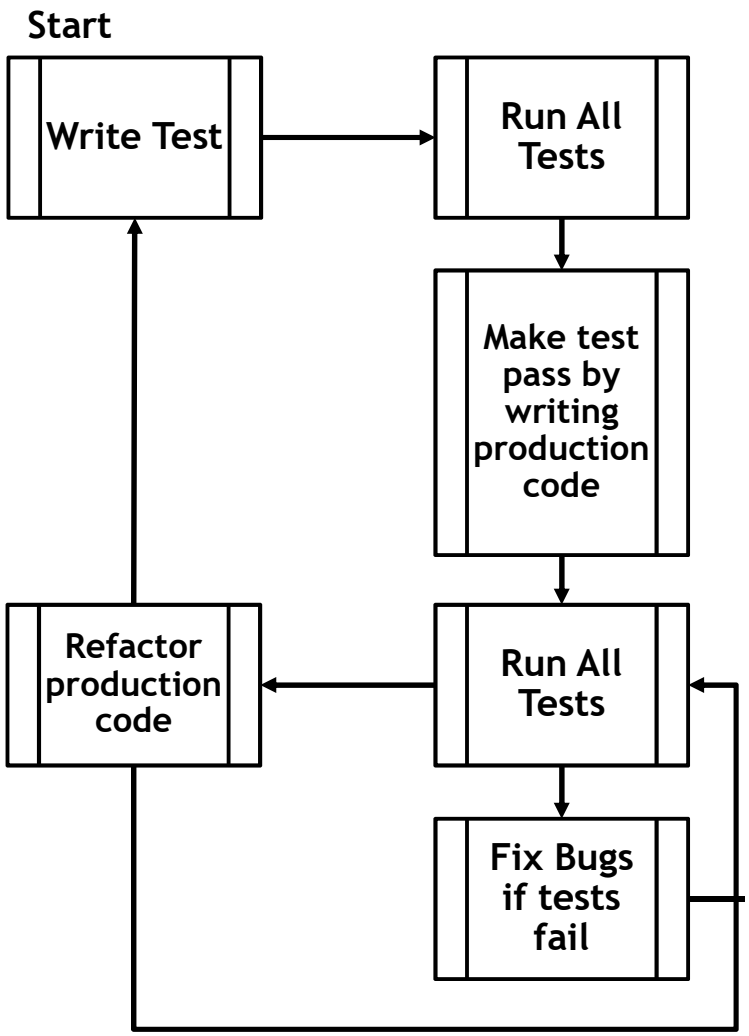
Traditional way of writing unit tests

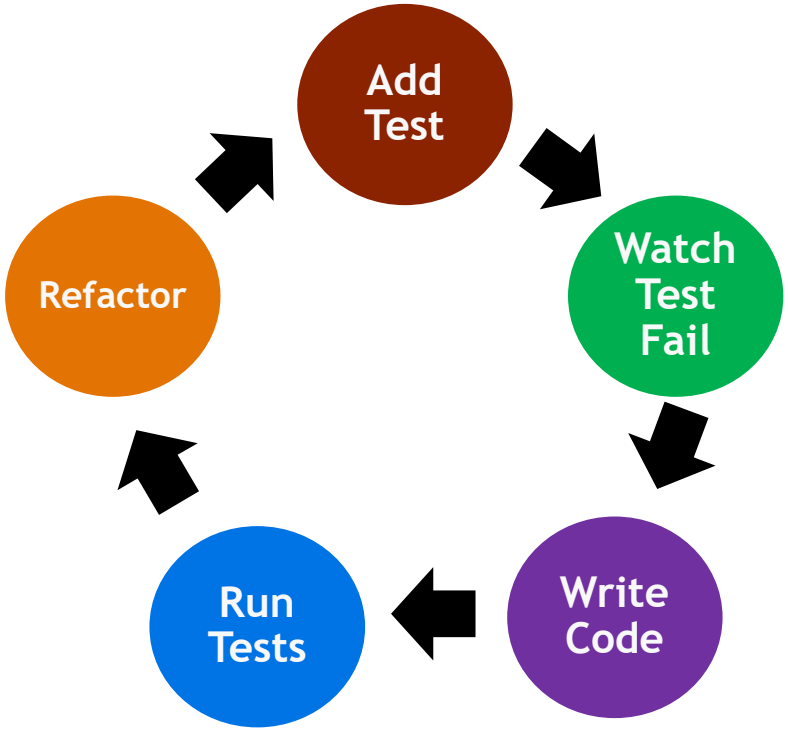


Unit Testing

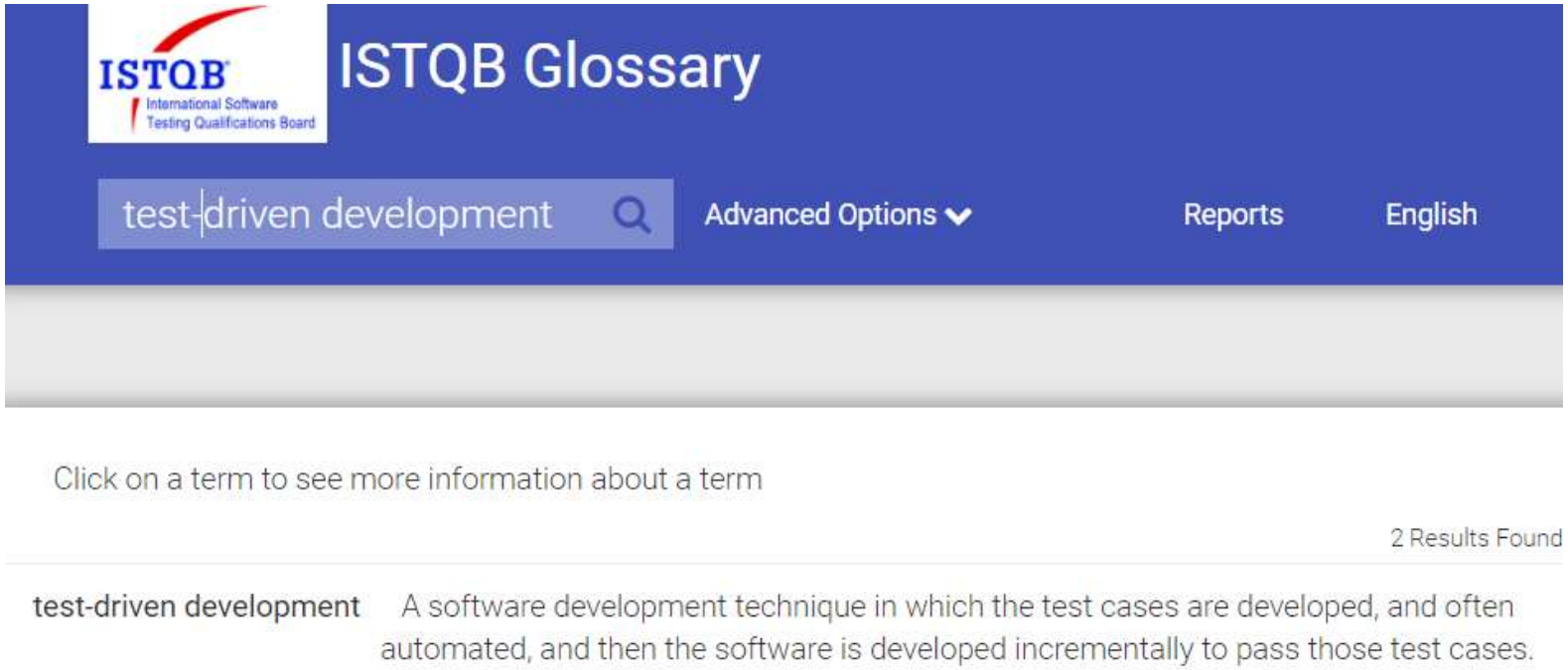


Test-Driven Development (TDD)





TDD - Test Driven Development



- A software development technique in which the test cases are developed, and often automated, and then the software is developed incrementally to pass those test cases.

Test-Driven Development (TDD)

- Development approach appropriate for unit testing
- The rhythm of Test-Driven Development (TDD) can be summed up as follows:
  1. Quickly add a test.
  2. Run all tests and see the new one fail.
  3. Make a little change.
  4. Run all tests and see them all succeed.
  5. Refactor to remove duplication.

[Kent Beck, Test-Driven Development, Addison-Wesley, 2003]

Test-Driven Development (TDD)

- **Red**: Write a failing test to prove code or functionality is missing from the end product
  - The test is written as if the production code is already working; the test failing means there is a bug in the production code.
- **Green**: Make the test pass
  - By writing production code that fits the reality that your test is expecting. It should be written as simple as possible.
- **Refactor:** Refactor your code
  - When the test passes, you are free to move on to the next unit test or refactor your code to make it more readable, remove code duplication, and more

Refactoring

- **Refactoring** is the act of changing a piece of code without changing its functionality.
  - If you've ever renamed a method, you've done refactoring.
  - If you've ever split a large method into multiple smaller method calls, you've refactored your code. It still does the same thing; it's just easier to maintain, read, debug, and change.

Principles of TDD

- Simplicity
- Minimalist approach
- Test-first
  - *During development* - When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
  - *During Debugging* - When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.
- Focus on needed functionality

Benefits of TDD

- Unit tests “prove” that your code actually works. You get more confidence on the code’s quality
- You get a low-level regression-test suite
- You can improve the design without breaking it. Helps code maintenance
- Unit tests demonstrate concrete progress
- Unit tests are a form of sample code and code documentation
- Test-first forces you to plan before you code
- Test-first reduces the cost of bugs. You find bugs earlier
- Unit tests make better designs
- …

Benefits of TDD

- The suite of unit tests provides constant feedback that each component is still working.
- The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- When the test passes and the production code is refactored, it is clear that the code is finished, and the developer can move on to a new test.
- TDD forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

Benefits of TDD

- The software tends to be better designed (loosely coupled and easily maintainable), because the developer is free to make design decisions and refactor at any time with confidence that the software is still working (by running the tests).
- The test suite acts as a regression safety net on bugs: if a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
- Reduced debugging time!

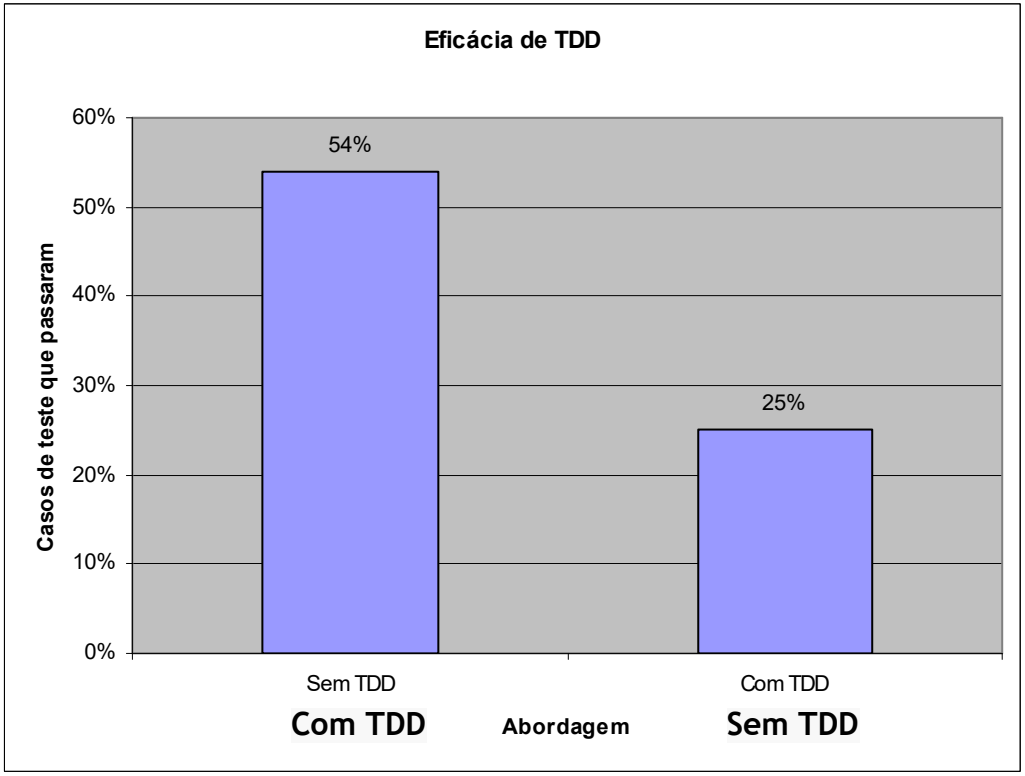
[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

What about arquitecture and design

- TDD does not replace architecture or design
- You need to do up front design and have a vision
- But, you should not do “Big Up Front Design”
- Defer architectural decisions as long as is “responsibly” possible
- TDD will inform and validate (or invalidate) your design decisions
- TDD will almost undoubtedly uncover weaknesses and flaws in your design.

Is it worth?

- Experiment with 20 students (TQS, FEUP, 2006/07
- Convert numbers into words
- Time bounded to one hour
- Half of the students used TDD, the others built tests on the end
- Quality of each solution evaluated with a test suite with 60 tests



With TDD: 54% of the test cases passed
Without TDD: 25% of the test cases passed

Extreme Programming

ISTQB International Software Testing Qualifications Board

ISTQB Glossary | extreme programming | Advanced Search | Reports | English

Click on a term to see more information about a term

2 Results Found

Extreme Programming	A software engineering methodology used within Agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code.
Agile testing	Testing practice for a project using Agile software development methodologies, incorporating techniques and methods, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm.

- XP: A software engineering methodology used within Agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code.

eXtreme programming

- Formulated in 1999 by Kent Beck, Ward Cunningham and Ron Jeffries
- Agile software development methodology (other: Scrum)
- Developed in reaction to high ceremony methodologies

eXtreme programming

- XP embrace change knowing that all requirements will not be known at the beginning and will change
- Use tools to accommodate change as a natural process
- Do the simplest thing that could possibly work and refactor mercilessly
- Emphasize values and principles rather that process

eXtreme Programming (XP)

- Extreme Programming (XP) is a set of values, principles, and disciplined practices that focus the entire team on delivery of value and reduction of waste. A typical day for the programming team would be as follows:
  - 15 minute team *stand-up* meeting
  - Get a story from task board
  - Pair-up
  - Talk with the customer sitting with the team
  - Hold programming episodes
  - Monitor the automated build & test system
  - Deliver a few fully tested, integrated business Stories
  - Go home at a reasonable time

eXtreme Programming (XP)

- Although practices used in XP are actually common practices which are shared by other methodologies, XP goes further by doing those practices to an extreme level (hence the "Extreme" Programming). Below is a table on how XP is extreme.

Good practices are	Pushed to the *extreme*
Code reviews	Review code all the time (pair programming)
Testing	Everybody will test all the time (unit testing) even the customers (functional testing)
Design	Make it part of everybody's daily business (refactoring)
Simplicity	Always leave the system with the simplest design that supports current functionality (simplest thing that could possibly work)
Architecture	Everybody will work defining and refining the architecture all the time (metaphor)
Integration testing	Integrate and test several times a day (continuous integration)
Short iterations	Make iterations really short-seconds, minutes, hours not weeks, months, years (the planning game)

[http://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/#xp]

XP Values

- **Communication** – let everyone know what's happening
- **Simplicity** – don't solve problems you don't have
- **Feedback** – start with the end in mind
- **Courage** – effective Action in the face of fear
- **Respect** – if people don't care, the project will not succeed

[Extreme Programming Explained, by Kent Beck]

References and further reading

- http://glossary.istqb.org/
- www.junit.org
- www.nunit.org
- http://testdriven.net/
- [IEEE 90] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.
- "The art of unit testing – With Examples in .NET", by Roy Osherove, ISBN: 1933988274
- xUnit Test Patterns, Gerard Meszaros, Addison-Wesley, 2007
- http://www.onestoptesting.com
- Test Driven Development, Kent Beck, Addison Wesley, 2002
- Test-Driven Development in Microsoft.Net, James W. Newkirk, Alexei A. Vorontsov, Microsoft Professional, 2004
- IEEE Software, May/June 2007 (Vol. 24, N. 3), Special issue on TDD
- http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3
- "Extreme Programming in the Real World - Java Extreme Programming Cookbook", by Eric M. Burke & Brian M. Coyner, O'Reilly
- http://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/#xp
- http://www.geocities.com/xtremetesting/TestingDictionary.html
- Lean, Agile, and Extreme Programm, By Mark Windholtz, ObjectWind Software Ltd, in the RailsStudio.com
- "Extreme Programming Explained: Embrace Change", By Kent Beck, Published by Addison-Wesley, 2000
- http://christopher-steiner.com/html/?p=37

TVVS - *Software Testing, Verification and Validation*

Mutation Testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- ISO/IEC 25010:2011 Standard
- Some concepts
- Mutation Testing

ISO/IEC 25010:2011

- Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models

What is quality?

ISO/IEC 25010:2011 defines:

- A **product quality** model composed of eight characteristics (which are further subdivided into sub characteristics) that relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products.
- A **quality in use** model composed of five characteristics (some of which are further subdivided into sub characteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.

ISO/IEC 25010:2011- Product quality model

The product quality model defined in ISO/IEC 25010 comprises the eight quality characteristics shown in the following figure:

Functional Suitability

- This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following sub characteristics:
  - **Functional completeness.** Degree to which the set of functions covers all the specified tasks and user objectives.
  - **Functional correctness.** Degree to which a product or system provides the correct results with the needed degree of precision.
  - **Functional appropriateness.** Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Performance efficiency

- This characteristic represents the performance relative to the amount of resources used under stated conditions. This characteristic is composed of the following sub characteristics:
  - **Time behaviour.** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
  - **Resource utilization.** Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
  - **Capacity.** Degree to which the maximum limits of a product or system parameter meet requirements.

Compatibility

- Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. This characteristic is composed of the following sub characteristics:
  - **Co-existence.** Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
  - **Interoperability.** Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.

Usability

- Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. This characteristic is composed of the following sub characteristics:
  - **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
  - **Learnability.** degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
  - **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
  - **User error protection.** Degree to which a system protects users against making errors.
  - **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
  - **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Reliability

- Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. This characteristic is composed of the following sub characteristics:
  - **Maturity.** Degree to which a system, product or component meets needs for reliability under normal operation.
  - **Availability**. Degree to which a system, product or component is operational and accessible when required for use.
  - **Fault tolerance.** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
  - **Recoverability.** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

Security

- degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. This characteristic is composed of the following sub characteristics:
  - **Confidentiality.** Degree to which a product or system ensures that data are accessible only to those authorized to have access.
  - **Integrity.** Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
  - **Non-repudiation.** Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
  - **Accountability.** Degree to which the actions of an entity can be traced uniquely to the entity.
  - **Authenticity.** Degree to which the identity of a subject or resource can be proved to be the one claimed.

Maintainability

- This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. This characteristic is composed of the following sub characteristics:
  - **Modularity.** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
  - **Reusability.** Degree to which an asset can be used in more than one system, or in building other assets.
  - **Analyzability.** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
  - **Modifiability.** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
  - **Testability.** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Portability

- Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. This characteristic is composed of the following sub characteristics:
  - **Adaptability.** Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
  - **Installability.** Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
  - **Replaceability.** Degree to which a product can replace another specified software product for the same purpose in the same environment.

ISO/IEC 25010:2011- Quality in use

- **Effectiveness** - *The accuracy and completeness with which users achieve specified goals*
- **Efficiency**- *The resources expended in relation to the accuracy and completeness with which users achieve goals*
- **Satisfaction**- *The degree to which user needs are satisfied when a product or system is used in a specified context of use*
  - Usefulness
  - Trust
  - Pleasure
  - Comfort

ISO/IEC 25010:2011- Quality in use

- **Freedom from risk** - degree to which a product or system mitigates the potential risk to economic status, human life, health, or the environment
  - Economic risk mitigation
  - Health and safety risk mitigation
  - Environmental risk mitigation
- **Context coverage** - degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified
  - Context completeness
  - Flexibility

Agenda

- ISO/IEC 25010:2011 Standard
- **Some concepts**
- Mutation Testing

Some concepts

- Error / Defect / Failure – What is the difference?

Analyst/Designer/Programmer makes a mistake/error.

Defect appears in the program.

Defect remains undetected during testing.

The program fails during execution i.e. it behaves unexpectedly.

Error / Defect / Failure

- **error**: A human action that produces an incorrect result.
- **defect:** A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.
- **failure:** Deviation of the component or system from its expected delivery, service or result..

Verification vs Validation (V&V)

- **Verification** - Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled. [ISTQB]
  - ensures that you "**build it right**", i.e., that the work products and the final product meet **specified requirements**

"Are we building the product right?"

- **Validation** - Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISTQB]
  - ensures that you "**build the right thing**", i.e., that the product will fulfill its **intended use** in its intended environment

"Are we building the right product?"

Agenda

- ISO/IEC 25010:2011 Standard
- Some concepts
- **Mutation Testing**

Mutation Testing

What tests the tests?

Mutation Testing

- Mutation testing is a **fault-based testing technique** that is based on the assumption that a program is well tested if
- all simple faults are predicted and removed;
- complex faults are combinations of simple faults and are therefore detected by tests that detect single faults.

Mutation Testing

- Mutation testing is **used to test the quality of your test suite**. This is done by mutating certain statements in your source code and checking if your test code is able to find the errors.
  - How do you know that you can trust your unit tests?
  - How do you know that they're really telling you the truth?
  - If they don't find a bug, does that really mean that there aren't any?
  - What if you could test your tests?

Mutation testing

- **Mutation testing** is a method of software testing, which involves modifying program's source code or byte code in small ways.
  - In short, any tests which pass after code has been mutated are defective.
- These, so-called ***mutations***, are based on well-defined ***mutation operators*** that either mimic typical programming errors
  - (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero).
- The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

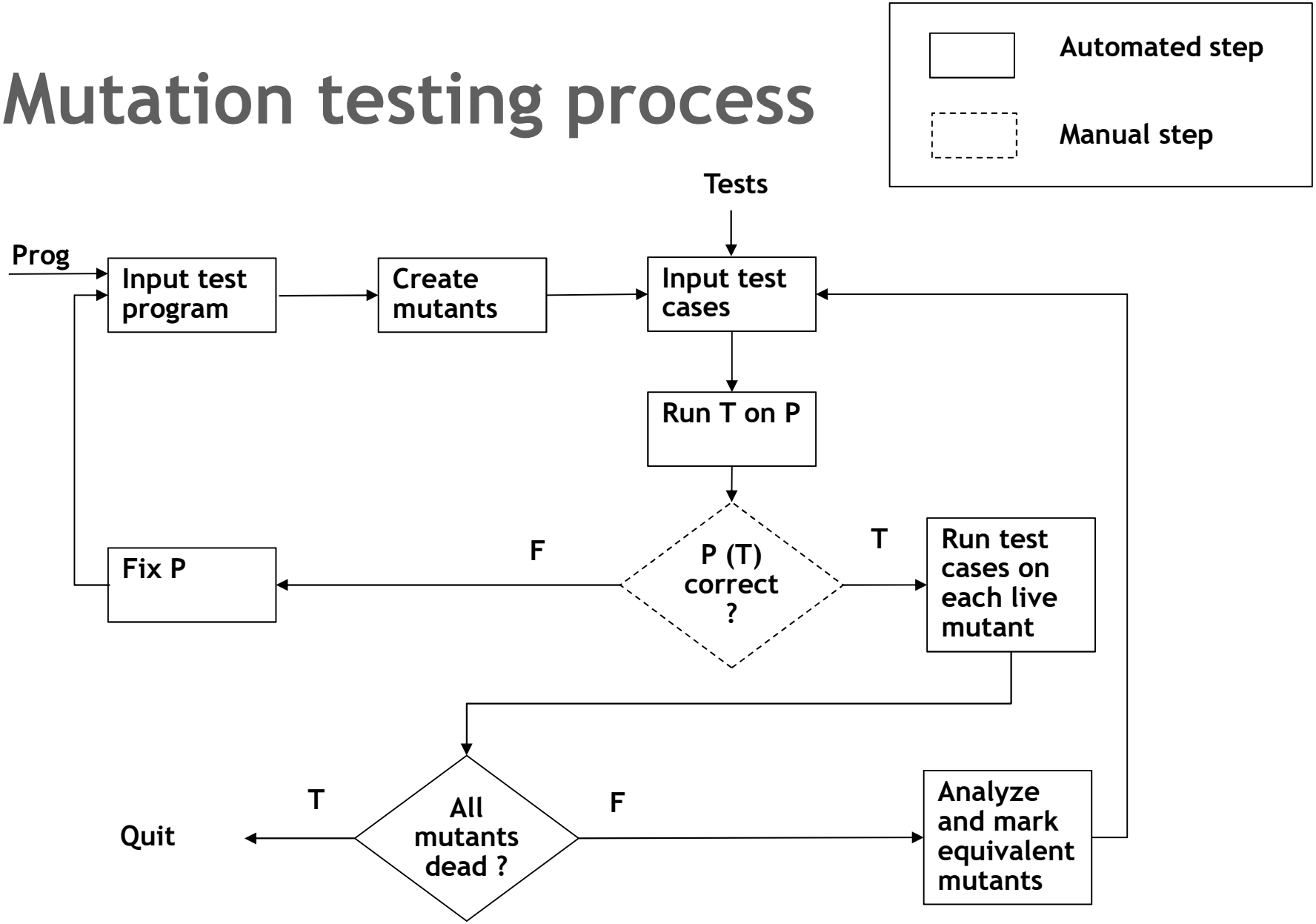
Mutation testing

- **Mutation Testing** is the process of generating tests to improve the mutation analysis score.
- **Mutation Analysis** is the process of measuring how good a test set is (how many mutations it kills).
- The **mutation score** is the ratio of dead mutants over the total number of non-equivalent mutants.
  - Thus, the tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected. A test set that kills all the mutants is said to be adequate relative to the mutants. If (as is likely) mutants are still alive, the tester can enhance the set of test cases by supplying new inputs.

Mutation Testing

- To assess the quality of the tests by performing them on mutated code
- Help construct *more adequate* tests
- Produce a suite of *valid tests* which can be used on real programs

Mutation testing process



Mutation testing process

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is **modified** in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are **run** with each mutant
- **Live mutants** cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as **killing** such mutant

Mutation testing process

- If a mutant remains **live** (passes all the test cases), then either
  - the mutant is **equivalent** to the parent (and is ignored), or
  - it is not equivalent,
- in which case additional test cases should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Mutation testing

- A **mutation operator** is a rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements.

Traditional Mutation Operators

Examples of traditional mutation operators

- **Deletion of a statement**
- **Boolean**
  - Replacement of a **statement** with another, e.g., `==` and `>=`, `<` and `>`
  - Replacement of **Boolean expressions** with true and false, e.g., `a||b` with **True**
- Replacement of **arithmetic**, e.g., `*` and `+`, `/` and `-`
- Replacement of **variable** (ensuring same scope/type)

Method-level Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Deletion
COD	Conditional Operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Mutation operators

- Beside this, there are mutation operators for **object-oriented** languages, for concurrent constructions, complex objects like containers etc. They are called class-level mutation operators.
- For example, *muJava* classifies class mutation operators into four groups: Encapsulation, Inheritance, Polymorphism and Java-Specific Features.

Inter-Class Mutation Operators

Language Features	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHM	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor type

Inter-Class Mutation Operators

Language Features	Operator	Description
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Class type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Argument overloading method call change

Inter-Class Mutation Operators

Language Features	Operator	Description
Java-specific features	JTI	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* modifier insertion
	JSD	*static* modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Some examples

ENCAPSULATION

Example: AMC - Access modifier change:

Original Code	AMC Mutants
public Stack s;	private Stack s; protected Stack s; Stack s;

INHERITANCE

Example: IHD - Hiding variable deletion

Original Code	IHD Mutan

class List {
  int size;
  ...
}
class Stack extends List {
  int size;
  ...
}
	class List {
	  int size;
	  ...
	}
	class Stack extends List {
	  // int size;
	  ...
	}

More examples: https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf

Types of Mutants

- Based on the types of faults seeded, mutants can be classified as
  - **First Order Mutants** (FOMs) and
  - **Higher Order Mutants** (HOMs).
- First order mutants seed only simple faults, generated by a single syntactic change to the original program.
- Higher order mutants combine simple first order faults to simulate more complex faults.

Weak/Strong mutation testing

Imagine the following statement of a program:

if (a && b) c=1; else c=0;

- The condition mutation operator would replace "&&" with '||' and produce the following mutant:

if (a || b) c=1; else c=0;

Weak/Strong mutation testing

Now, for the test to kill this mutant, the following condition should be met:

1) Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this (it goes through **else** in the original and through **then** in the mutated).

2) The value of 'c' should be propagated to the program's output and checked by the testing.

Weak/Strong mutation testing

- ***Weak mutation testing*** (or *weak mutation coverage*) requires that only the first condition is satisfied. ***Strong mutation testing*** requires that both conditions are satisfied.
  - Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.
- **Weak Mutation Testing** is a coverage measure - i.e. tells you about the code that is run by your tests
- **Strong Mutation Testing** measures whether your code needs to be like it is to pass the tests.

Problems with mutation testing

1. High computational cost
2. Trivial mutants
3. Equivalent mutants (human effort)
4. Oracle (human effort)

1. Computational effort

- One of the barriers to the practical use of mutation testing is the unacceptable computational expense of generating and running vast numbers of mutant programs against the test cases.
- The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation.

1. Computational effort

- Approaches to reduce this computational expense usually follow one of three strategies: **do fewer**, **do smarter**, or **do faster**.
  - The **do fewer** approaches seek ways of running fewer mutant programs without incurring intolerable information loss.
  - The **do smarter** approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution.
  - The **do faster** approaches focus on ways of generating and running each mutant program as quickly as possible.

2. Trivial Mutants

- Traditional mutation testing has only applied **first order mutants**.
- However, many of the first order mutants generated by these simple syntactic fault insertions are **readily killed by the simplest of test cases executed**, leading **to much wasted effort** killing rather trivial mutants.
- As a result, many mutation testers observe that even the **most trivial, small and unimaginative test suite will kill a very large proportion of the first order mutants**.

3. Equivalent mutants

- Many mutation operators can produce equivalent mutants. For example, Boolean relation mutation operator will replace “==“ with ">=" and produce the following mutant:

original

```
int index=0;
while (…) { …;
  index++;
  if (index==10)
    break;
}
```

mutant

```
int index=0;
while (…) { …;
  index++;
  if (index>=10)
    break;
}
```

- However, it is not possible to find a test case which could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called equivalent mutants.

3. Equivalent mutants

- Equivalent mutants detection is **one of biggest obstacles** for practical usage of mutation testing. The effort, needed to check if mutants are equivalent or not, can be very high even for small programs.

4. Oracle

- The human **oracle problem** refers to the process of **checking the original program's output** with each test case.
- Strictly speaking, this **is not a problem unique to mutation** testing. In all forms of testing, once a set of inputs has been arrived at, there remains the problem of checking the output.
- However, **mutating testing is effective** precisely because it is a demanding test, and this **can lead to an increase** in the number of test cases, thereby increasing the oracle cost.
- The **oracle cost** is often the **most expensive** part of the overall test activity

Some Tools

- PIT
- Major
- MuJava
- muJava
- MuClipse
- Jumble
- JesTer (for Java)
- PesTer (for Python)
- SQLMutation
- …
- Mothra
- Proteum
- SQLMutation
- Nester (for C#)
- …

Exercise

```
1:  bool A,B,C; int V;
2:  read(A);read(B);read(C);read(V);
3:  while (V<20)  {
4:        if ( A /\ ( B \/ C ) )
5:              V:= V+5;
6:        if (A /\ ~B)
7:              V:=V+10;
8:  }
9:  print V;
```

Id	Mutation	A	B	C	V
1	3: while (V<=20)				
2	4: if (A \/ (B \/ C))				
3	6: if (A \/ ~B)				

Conceive test cases (with the 100% degree of coverage) for the program considering the following mutants. Indicate that the value returned by the program in each of these test cases.

Exercise

```
1: read(A); read(B); read(C);
2: W := 0;
3: while (C>B AND C>A) {
4:        C := C-5;
5:        if (C>10)
6:             W:=5;
7:        if (C==A+B OR C>A-B)
8:             W:=10;
9:        else if (C==10)
10:                 W:=15;
13: }
14: print(C);print(W);
```

Id	Mutation	A	B	C
1	3: while (C>B OR C>A)			
2	5: if (C<10)			
3	9: else if (C<>10)			

Conceive test cases (with the 100% degree of coverage) for the program considering the following mutants. Indicate that the value returned by the program in each of these test cases.

References

- http://www.mutationtest.net/
- http://en.wikipedia.org/wiki/Mutation_testing
- Black, P. E., V. Okun, et al. (2000). Mutation of Model Checker Specifications for Test Generation and Evaluation. Mutation 2000, Jan Jose, California, Kluwer Academic Publishers.
- Murnane, T. and K. Reed (2001). On the Effectiveness of Mutation Analysis as a Black Box Testing Technique. Software Engineering Conference, Australian.
- Ammann, P. E., P. E. Black, et al. (1998). Using Model Checking to Generate Tests from Specifications. 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia, IEEE Computer Society.
- http://cs.gmu.edu/~nli1/oomutation/oomuts.pdf
- https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf
- Higher Order Mutation Testing by Yue Jia; http://www0.cs.ucl.ac.uk/staff/M.Harman/yue-phd.pdf

TVVS - Test, Verification and Validation of Software

Integration and System Testing

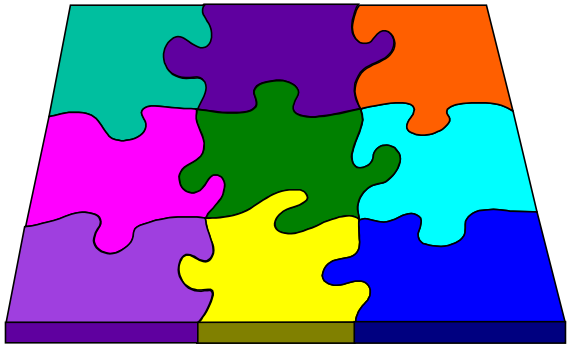
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

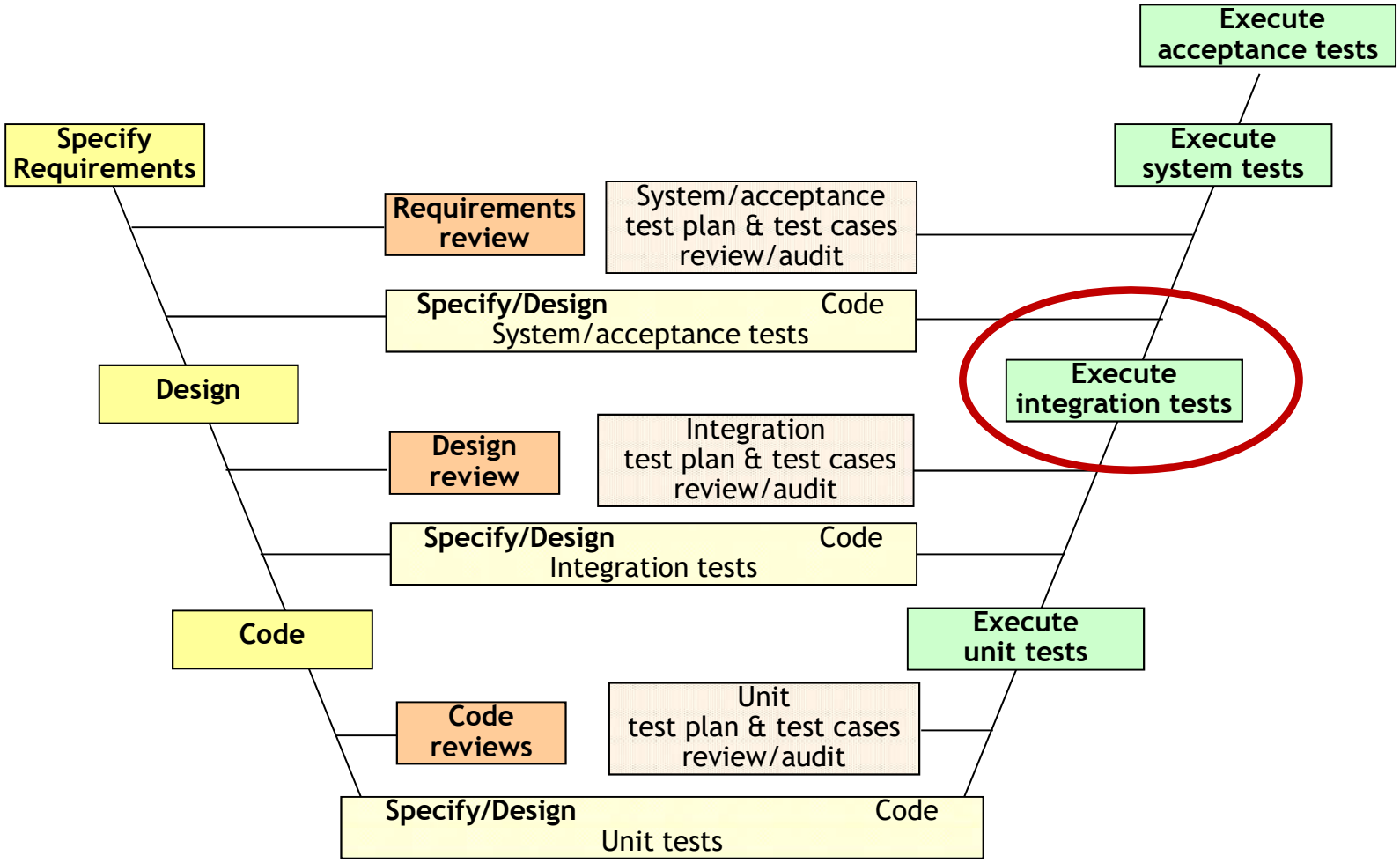
Agenda

- Integration testing
- System testing
- Regression testing

Integration testing



V-model



The extended V-model of software development [I.Burnstein]

Integration testing

- Testing of groups of components integrated to create a sub-system. Components should be tested previously.
- Usually the responsibility of an independent testing team (except sometimes in small projects)
- Integration testing should be black-box testing with tests derived from the technical specification
- A principal goal is to detect defects that occur on the interfaces of units
- Main difficulty is localising errors
- *Incremental* integration testing (as opposed to *big-bang* integration testing) reduces this difficulty

Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces
  - Sub-systems request services from other sub-systems

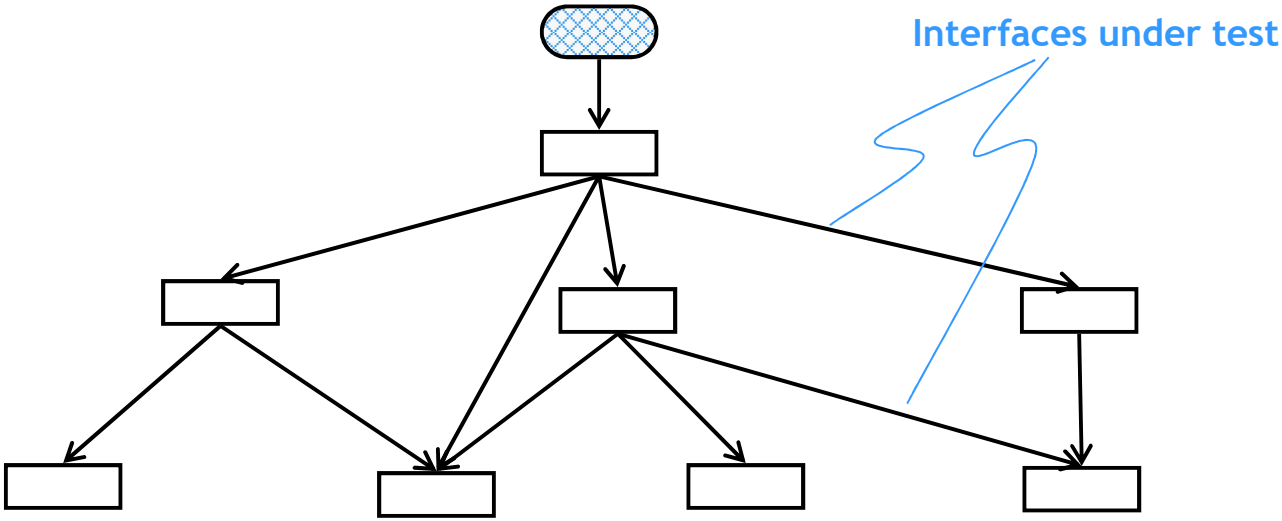
Interface errors

- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated
- …

Big-bang integration testing



Incremental integration testing

- Top-down integration testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up integration testing
  - Integrate individual components in levels until the complete system is created
- Sandwich Testing
  - Combination of Top-down with Bottom-up testing
- Collaboration integration testing
  - Appropriate for iterative development strategies where software components are created and fatten as new use cases are implemented (through a collaboration of objects and components)
  - Scenario based testing
- The integration testing strategy must follow the software construction strategy

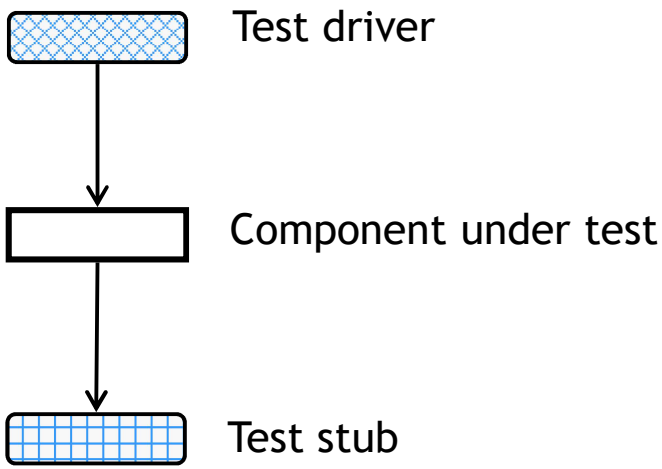
Integration testing - More strategies

- **component integration testing**
  - Testing in which the test items are interfaces and interactions between integrated components.
- **neighborhood integration testing**
  - A type of integration testing in which all of the nodes that connect to a given node are the basis for the integration testing.
- **pairwise integration testing**
  - A type of integration testing that targets pairs of components that work together as shown in a call graph.
- **system integration testing**
  - A test level that focuses on interactions between systems.

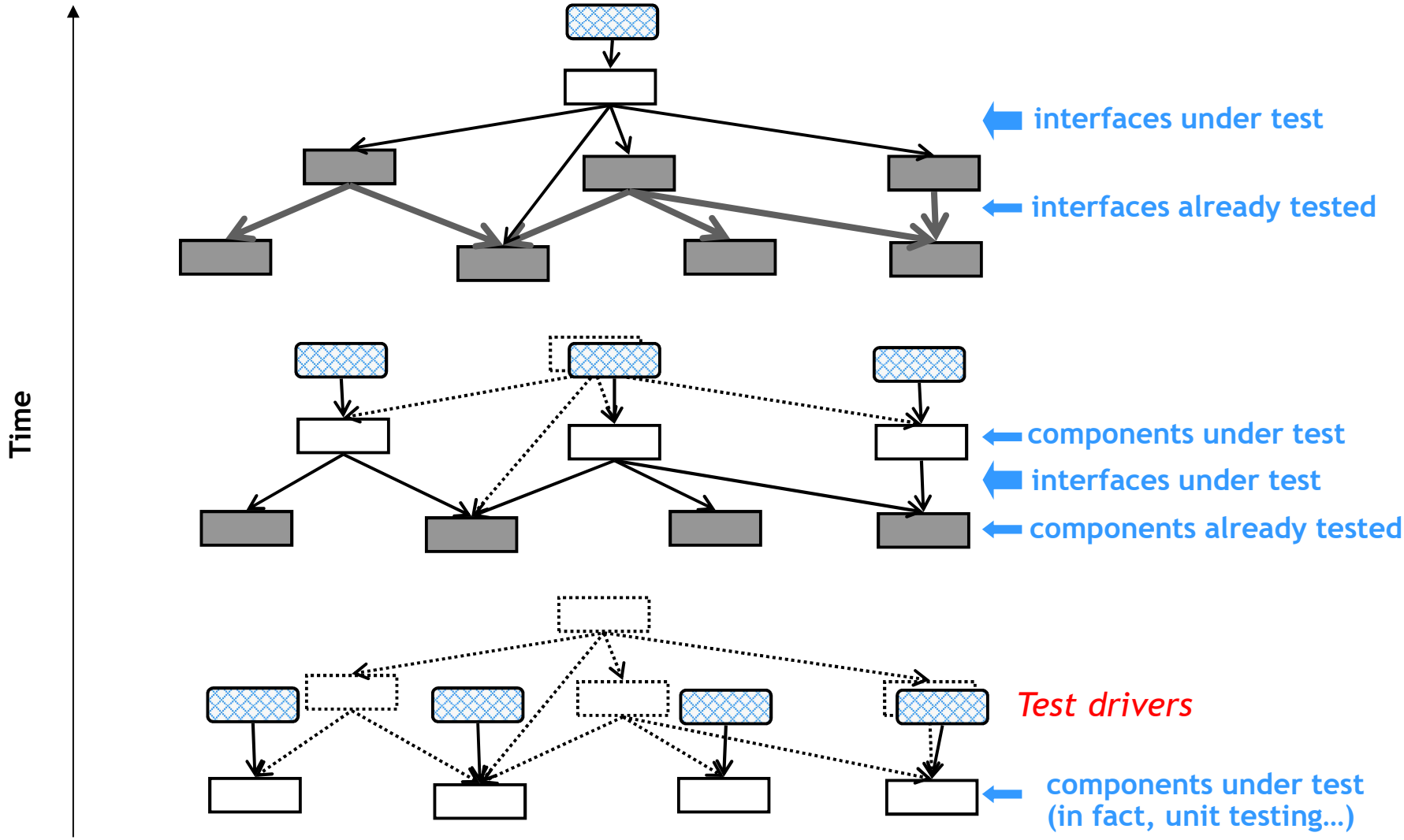
[ISTQB]

Test harness: drivers and stubs

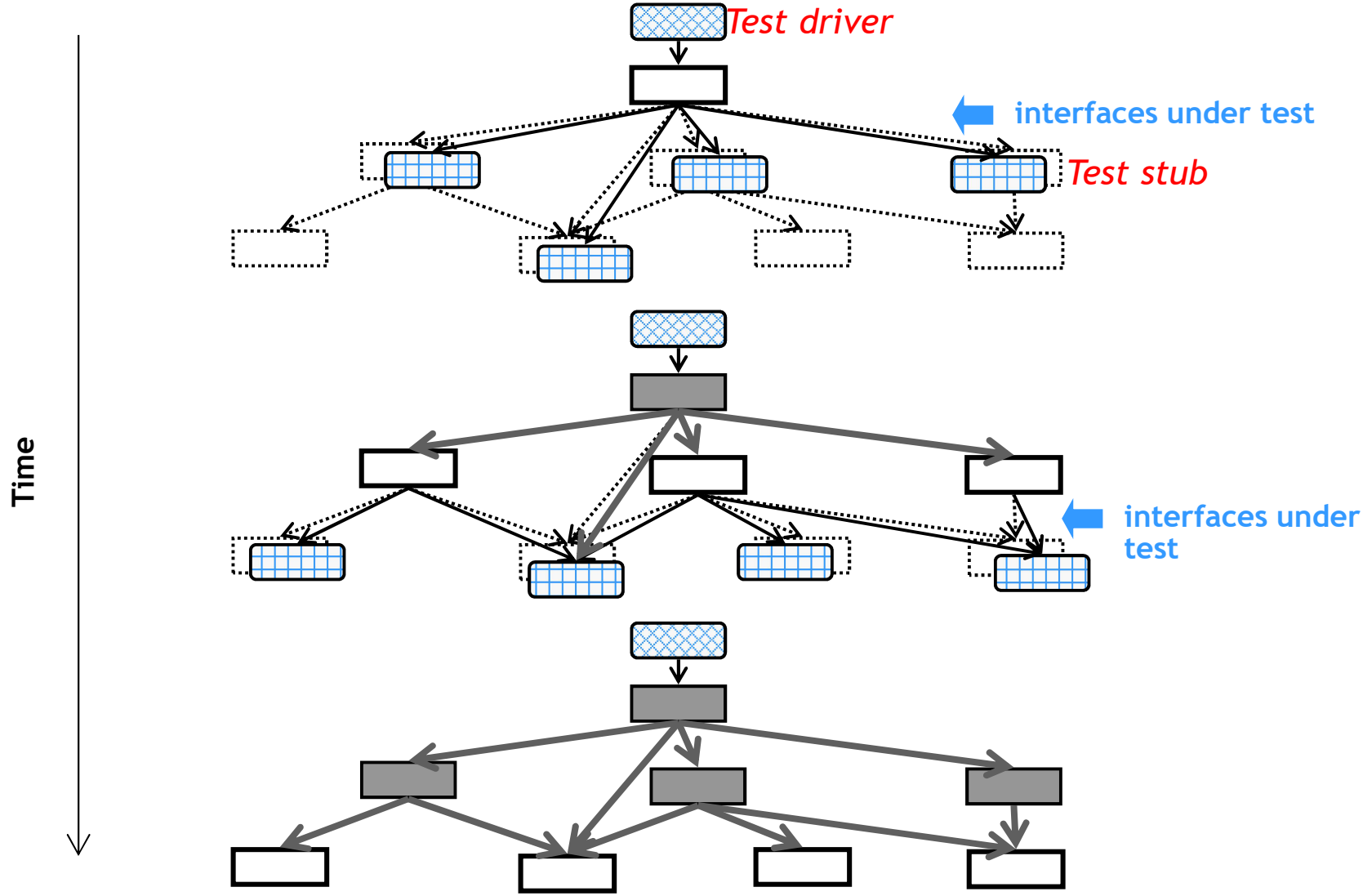
- Test harness: auxiliary code developed to support testing
- Test drivers
  - Call the target code, simulating calling units or a user
  - In automatic testing: implementation of test cases and procedures
- Test stubs
  - Simulate modules/units/systems called by the target code
  - Mock objects can be used for this purpose



Bottom-up integration testing



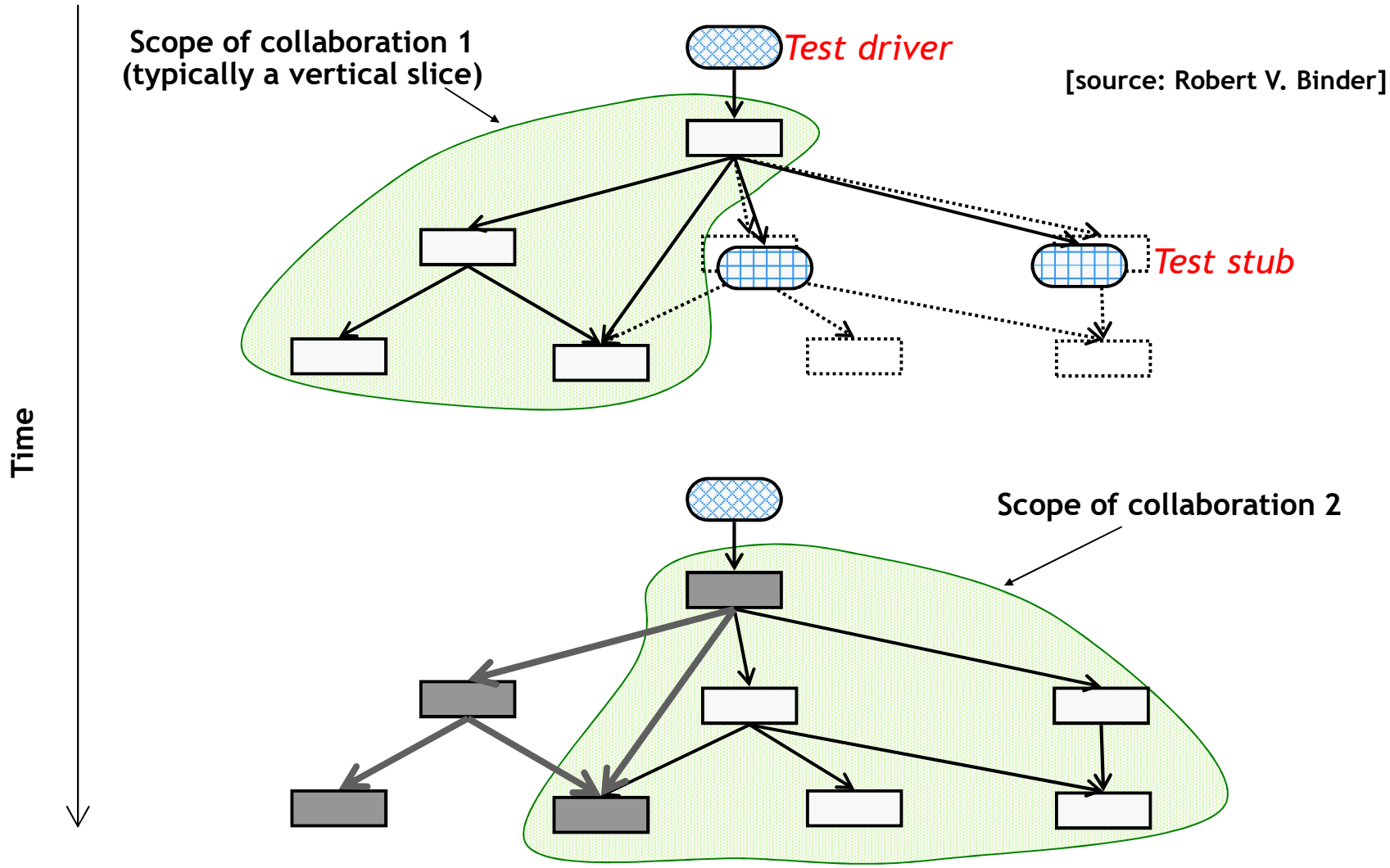
Top-down integration testing



Top-down versus bottom-up

- Design validation
  - Top-down integration testing is better at discovering design flaws that can be fixed earlier
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Top-down integration requires the development of complex stubs to drive significant data upward while bottom-up integration requires drivers. Often a combination of approaches known as sandwich
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Collaboration integration testing



Integration testing

- Top-down approach
  - Requires the highest-level modules be test and integrated first.
  - This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers.
  - Getting I/O functions early can ease test writing.
  - Early demonstration of the main functionality.
  - However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle.
  - Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.

Integration testing

- Bottom-up approach
  - Requires the lowest-level units be tested and integrated first.
  - Units are tested early in the development process and the need for stubs is minimized.
  - Test conditions may be easier to create.
  - The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late.
  - Major faults may be detected towards the end of the integration process.
  - Can not observe a functional system till the end of the process.
  - Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.

Integration testing

- Mixed approach
  - Requires testing along functional data and control-flow paths.
  - First, the inputs for functions are integrated in the bottom-up pattern discussed before.
  - The outputs for each function are then integrated in the top-down manner.
  - The primary advantage of this approach is the degree of support for early release of limited functionality.
  - It also helps minimize the need for stubs and drivers.
  - The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

Integration Testing - common problems

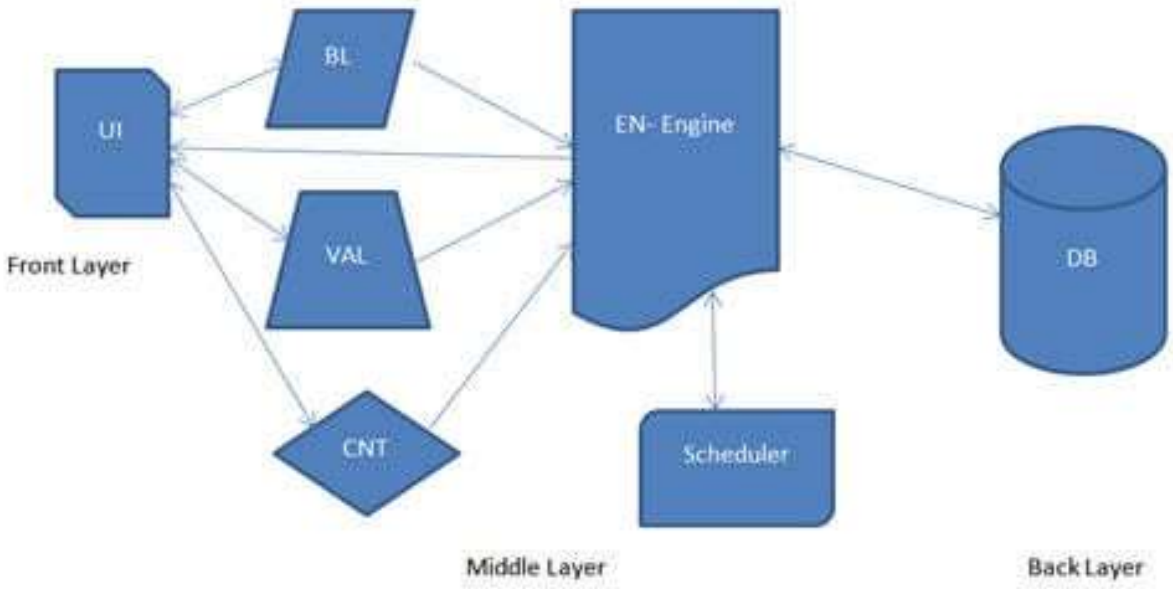
Problem	Description
Insufficient integration test environments (test beds)	Lack of sufficient test beds due to cost or hardware needed elsewhere causes competition for limited resources among test teams
Insufficient schedule	Insufficient time scheduled for adequate testing
Poor test bed fidelity	Insufficient sw, hw, and system fidelity to actual system (e.g., due to inaedquate sw simulations/drivers, prototype or wrong version)
Unavailable components	Not all the components to be integrated are ready at the proper time
Poor code quality	Code contains many defects that should have been found during lower-level testing, which unnecessarily slows integration testing
...	

Integration testing vs unit testing

Unit test	Integration test
Depends on code	Depends on external systems
Easy to write and verify	Setup of integration test might be complicated
Units tested in isolation	More than one component are tested
Dependencies are mocked if needed	No mocking (only unrelated componenents are mocked)
Verifies implementation code	Verifies interconection behaviour when components are used together
Most used by developers	Also useful to QA, DevOps, Help Desk
Failed test is a code problem (if the requirements didn't change)	Failed test may mean that the code is correct but the environment changed
Should be fast	May take long

Example

- **UI** – User Interface module, which is visible to the end user, where all the inputs are given.
  BL – Is the Business Logic module, which has all the all the calculations and business specific methods.
  VAL – Is the Validation module, which has all the validations of the correctness of the input.
  CNT – Is the content module which has all the static contents, specific to the inputs entered by the user. These contents are displayed in the reports.
  EN – Is the Engine module, this module reads all the data that comes from BL, VAL and CNT module and extracts the SQL query and triggers it to the database.
  Scheduler – Is a module which schedules all the reports based on the user selection (monthly, quarterly, semiannually & annually)
  DB – Is the Database.



https://www.softwaretestinghelp.com/what-is-integration-testing/

Example

Integration testing focuses on the flow of data between the modules

- **The questions here are:**
  - **How** the BL, VAL and the CNT module **will read and interpret** the data entered in the UI module?
  - Is BL, VAL and CNT module **receiving the correct** data from UI?
  - In **which format the data** from BL, VAL and CNT is transferred to the EQ module?
  - **How** will the EQ **read the data** and extract the query?
  - Is the query **extracted correctly**?
  - Is the Scheduler **getting the correct data** for reports?
  - Is the **result set** received by the EN, from the database is **correct** and as expected?
  - Is EN **able to send the response back** to the BL, VAL and CNT module?
  - Is UI module **able to read** the data and display it appropriately to the interface?

Example

- **Few other sample test conditions can be as follows:**
  - Are the menu options generating the correct window?
  - Are the windows able to invoke the window under test?
  - For every window, identify the function calls for the window that the application should allow.
  - Identify all calls from the window to other features that the application should allow
  - Identify reversible calls: closing a called window should return to the calling window.
  - Identify irreversible calls: calling windows closes before called window appears.
  - Test the different ways of executing calls to another window e.g. – menus, buttons, keywords.

Example

- **Steps to Kick off Integration Tests**
  - Understand the architecture of your application.
  - Identify the modules
  - Understand what each module does
  - Understand how the data is transferred from one module to another.
  - Understand how the data is entered and received into the system (entry point and exit point of the application)
  - Segregate the application to suit your testing needs.
  - Identify and create the test conditions
  - Take one condition at a time and write down the test cases.

System testing

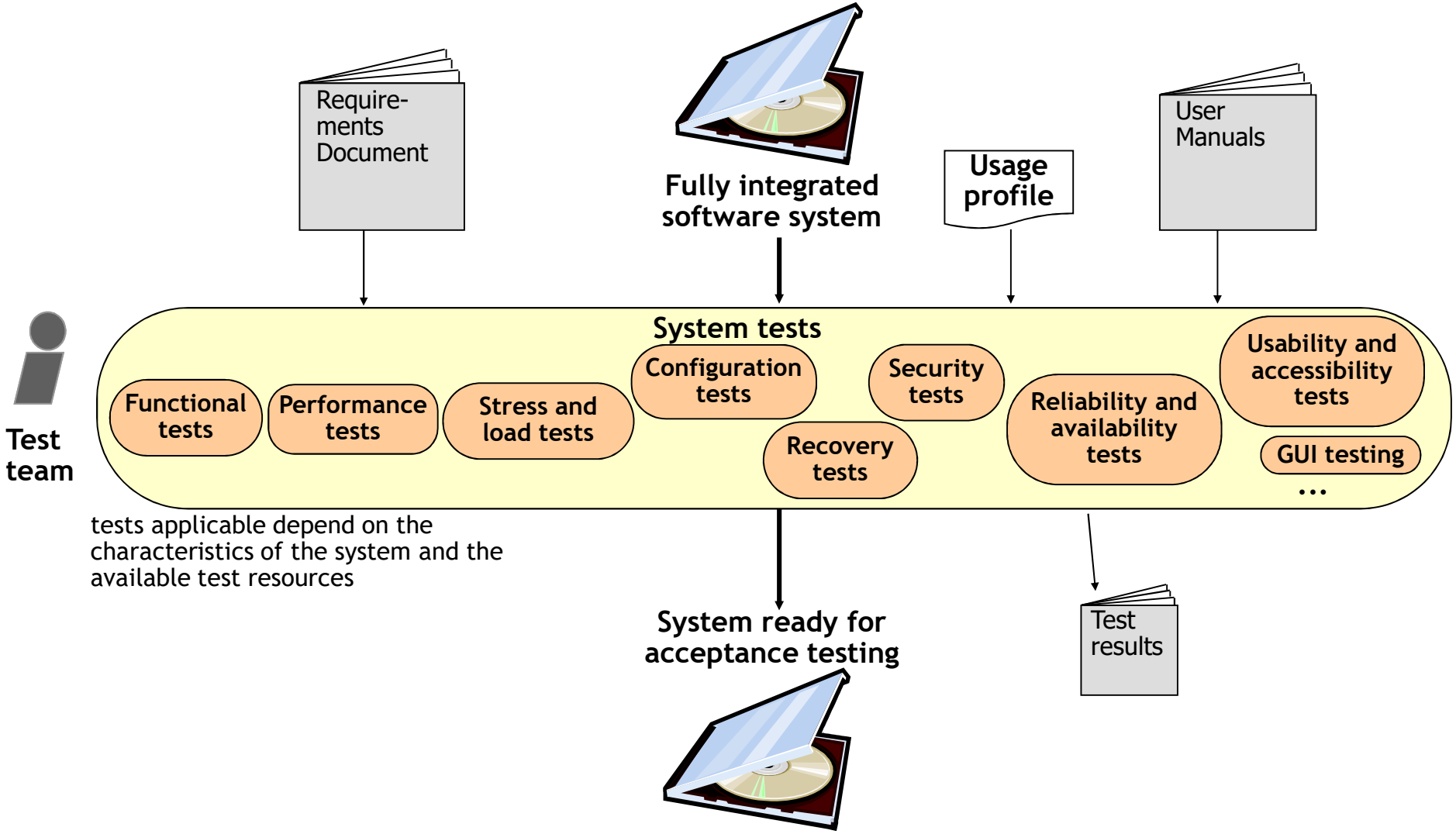
System testing

- Testing the system as a whole by an independent testing team
- Often requires many resources: laboratory equipment, long test times, etc.
- Usually based on a requirements document, specifying both functional and non-functional (quality) requirements
- Preparation should begin at the requirements phase with the development of a master test plan and requirements-based tests (black-box tests)

System testing

- The goal is to ensure that the system performs according to its requirements, by evaluating both functional behavior and quality requirements such as reliability, usability and performance.
- Especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage
- Tests implemented on the parts and subsystems may be reused/repeated, and additional tests for the system as a whole may be designed

Types of system testing



Functional testing

- Ensure that the behavior of the system adheres to the requirements specification
- Black-box in nature
- Equivalence class partitioning, boundary-value analysis and state-based testing are valuable techniques
- Document and track test coverage with a (tests to requirements) traceability matrix
- A defined and documented form should be used for recording test results from functional and other system tests
- Failures should be reported in test incident reports
  - Useful for developers (together with test logs)
  - Useful for managers for progress tracking and quality assurance purposes

Performance testing

- Goals:
  - See if the software meets the performance requirements
  - See whether there are any hardware or software factors that impact on the system's performance
  - Provide valuable information to tune the system
  - Predict the system's future performance levels
- Results of performance tests should be quantified, and the corresponding environmental conditions should be recorded

Performance testing

- Resources usually needed
  - a source of transactions to drive the experiments, typically a load generator
  - an experimental test bed that includes hardware and software the system under test interacts with
  - instrumentation of probes that help to collect the performance data (event logging, counting, sampling, memory allocation counters, etc.)
  - a set of tools to collect, store, process and interpret data from probes

Stress and load testing

- Load testing – maximize the load imposed on the system (volume of data, number of users, ...)
  - Examples:
    - a system is required to handle 10 interrupts / second and the load causes 20 interrupts/second
    - a suitcase being tested for strength and endurance is stomped by a multi tonne elephant
    - testing a word processor by editing a very large document

Stress and load testing

- Stress testing – minimize the resources available to the system (processor, memory, disk space, ...). Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation that are not revealed under normal testing conditions
  - Examples:
    - run processes that consume resources (CPU, memory, disk, network) on the Web and database servers
    - take the database offline, then restart it
- The goal is to try to break the system, find the circumstances under which it will crash, and provide confidence that the system will continue to operate correctly (possibly with bad performance but with correct functional behavior) under conditions of stress
- Supported by many of the resources used for performance testing

Configuration testing

- Configuration testing checks for failure of the system to perform under all of the combinations of hw and sw configurations. Typical sw systems interact with multiple hw devices such as disc drives, tape drives, and printers.
- Objectives [Beizer]:
  - show that all the configuration changing commands and menus work properly
  - show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions
  - show that the systems' performance level is maintained when the devices are interchanged, or when they fail

Configuration testing

- Types of test to be performed:
  - rotate and permute the positions of devices to ensure physical/logical device permutations work for each device
  - induce malfunctions in each device, to see if the system properly handles the malfunction
  - induce multiple device malfunctions to see how the system reacts

Security testing

- Evaluates system characteristics that relate to the availability, integrity and confidentiality of system data and services
- Computer software and data can be compromised by
  - criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy
  - errors on the part of honest developers/maintainers (and users?) who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge
- Both can be perpetuated by those inside and outside on an organization
- Areas to focus: password checking, legal and illegal entry with passwords, password expiration, encryption, browsing, trap doors, viruses, ...
- Usually the responsibility of a security specialist

Recovery testing

- Drive a system to losses of resources in order to determine if it can recover properly from these losses
- Especially important for transaction systems
- Example: loss of a device during a transaction
- Tests would determine if the system could return to a well-known state, and that no transactions have been compromised
  - Systems with automated recovery are designed for this purpose
- Areas to focus [Beizer]:
  - **Restart** – the ability of the system to restart properly on the last checkpoint after a loss of a device
  - **Switchover** – the ability of the system to switch to a new processor, as a result of a command or a detection of a faulty processor by a monitor

Reliability and availability testing

- **Software reliability** is the probability that a software system will operate without failure under given conditions for a given interval
  - May be measured by the mean time between failures (MTBF)
  - MTBF = MTTF (mean time to failure) + MTTR (mean time to repair)
- **Software availability** is the probability that a software system will be available for use
  - May be measured by the percentage of time the system is on or **uptime** (example: 99,9%)
  - A = MTTR / MTBF
- Low reliability is compatible with high availability in case of low MTTR
- Requires statistical testing based on usage characteristics/profile
  - During testing, the system is loaded according to the usage profile
- More information: Ilene Burnstein, section 12.5
- Usually evaluated only by high maturity organizations

Usability testing

- Usability tests are concerned with external Properties: The user's perspective:
  - **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
  - **Learnability.** degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
  - **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
  - **User error protection.** Degree to which a system protects users against making errors.
  - **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
  - **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Accessibility testing

- Accessibility testing is the technique of making sure that your product is accessibility compliant.
- Typical accessibility problems can be classified into following four groups, each of them with different access difficulties and issues:
  - **Visual impairments** such as blindness, low or restricted vision, or color blindness. User with visual impairments uses assistive technology software that reads content loud. User with weak vision can also make text larger with browser setting or magnificent setting of operating system
  - **Motor skills** such as the inability to use a keyboard or mouse, or to make fine movements
  - **Hearing impairments** such as reduced or total loss of hearing
  - **Cognitive abilities** such as reading difficulties, dyslexia or memory loss

GUI testing

- Raises specific challenges
  - GUI test automation is harder than API test automation
  - Observing visible GUI state is difficult
  - Observing invisible GUI state is tricky almost impossible
  - Controlling GUI actions is difficult
    - Event based
    - Non-solicited events
  - State space and test case explosion
    - Multiple ways (mouse, keyboard,…) to achieve the same goal
    - Almost all user actions are enabled most of the time

GUI errors – concrete examples

- Correct functioning
- Missing commands
- Correct window modality
- Mandatory fields
- Incorrect field defaults
- Data validation
- Error handling (messages to the user)
- Wrong values retrieved by queries
- Filling order
- (…)

GUI testing: manual techniques

- Heuristic Methods
  - A group of specialists studies the interface in order to find problems that they can identify.
- Guidelines
  - Recommendations about user interfaces. E.g.: how to organize the display and the menu structure.
- Cognitive walkthrough
  - The developers walk through the interface in the context of core tasks a typical user will need to accomplish. The actions and the feedback of the interface are compared to the user's goals and knowledge, and discrepancies between user's expectations and the steps required by the interface are noted.
- Usability tests
  - The interface is studied under real-world or controlled conditions (real users), with evaluators gathering data on problems that arise during its use.

GUI testing: manual techniques

- Advantages
  - More bugs found per test cases executed (good specialist) (not necessarily by time or money invested)
  - Adaptability: bugs found provide hints to find other bugs
  - Can find bugs (e.g., usability problems) that are difficult to find with automated tests (the converse is also true)
  - Can be supported / made more systematic/repeatable by checklists of standard tests and application specific tests
  - Good for exploratory / initial testing
- Disadvantages
  - Regression testing
  - Effort required
  - Weak coverage
  - Repeatability / reproducibility
  - Good test specialists are difficult to find
  - Depends on the capabilities of the tester

Automated GUI testing approaches

- Capture-Replay tools
  - WinRunner, Rational Robot, Android
- Random input testing tools
  - Rational's TestFactory uses dumb monkey method

- Unit testing frameworks
  - JUnit, NUnit
- Model-based testing tools
  - Spec Explorer (API testing)
  - Spec Explorer with GUI testing extensions
  - Guitar (Atif Memon)

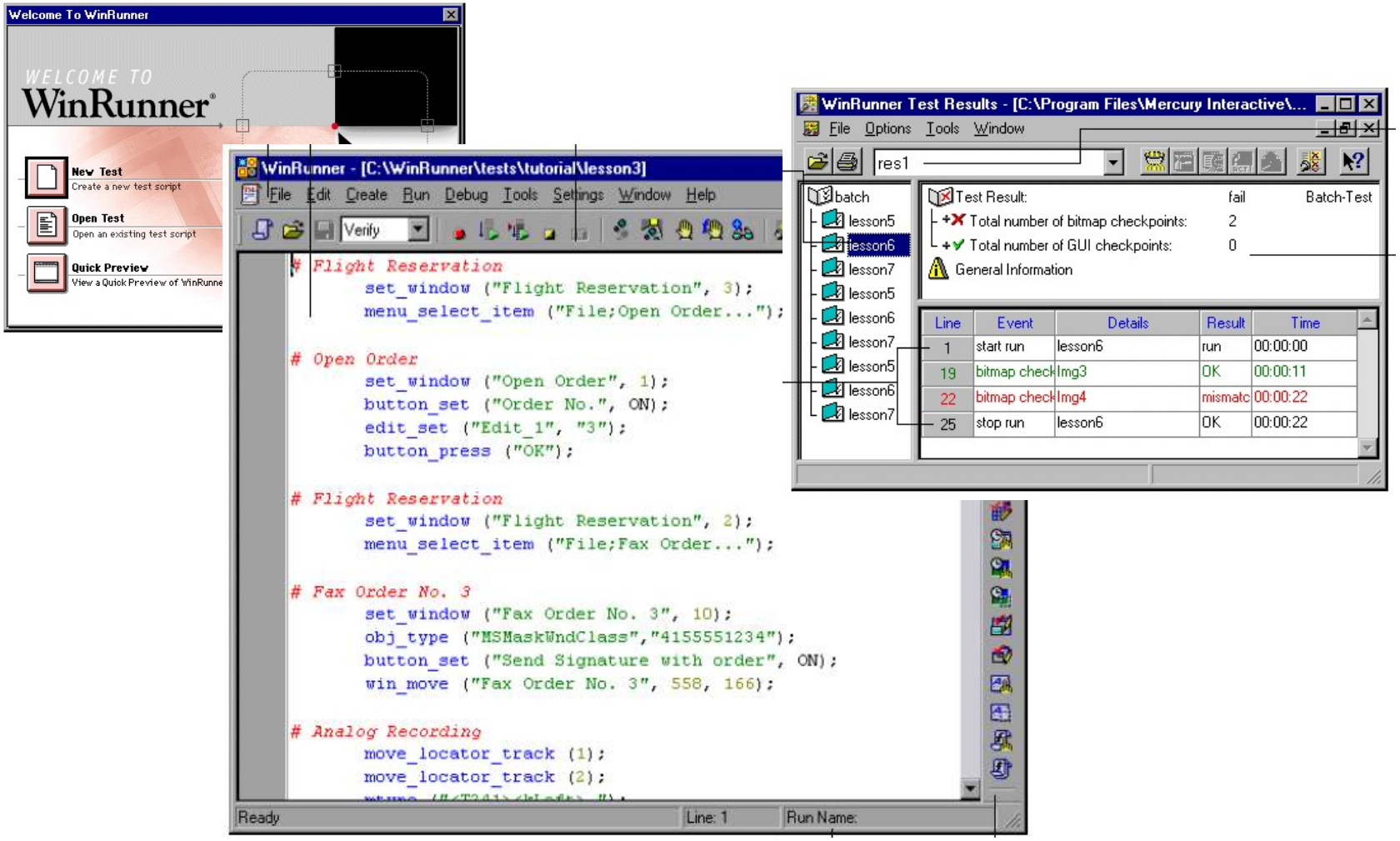
Capture replay tools

- How it works
  - Two different interacting modes
    - Capture
      - The user/tester interacts with the tool
      - The tools saves user actions and output
    - Replay
      - Saved user actions are reproduced
      - The output obtained is compared with the one expected
  - Three different execution modes
    - Position based
    - Object based
    - Mixed

Capture replay tools

- Output verification
  - Property/content comparison
  - Bitmap comparison
  - Optical character recognition
- Scripts
  - Saved user actions
  - From scratch
  - Mixed
  - A higher level of abstraction is needed to reach independence of GUI updates

Capture replay tools



Capture replay tools

- Advantages
  - Test case recording (capture) (similar to macro recording in Excel)
  - Recorded test scripts can be made more generic by programming
    - E.g., use logical names in test script and map logical names to physical objects in a separate file
  - Automatic test case execution (replay)
  - Output checking
  - Good for regression testing
- Disadvantages
  - Can be used only when the software application is working correctly
  - Sensitivity to physical details
  - Do not support automatic generation of test cases

Random testing tools

- Actions performed randomly without knowledge of how humans use the application
- Microsoft says that 10 to 20% of the bugs in Microsoft projects are found by these tools
- Two kinds of tools
  - Dumb monkeys – low IQ; they can't recognize an error when they see one
  - Smart monkeys – generate inputs with some knowledge to reflect expected usage; get knowledge from state table or model of the AUT.

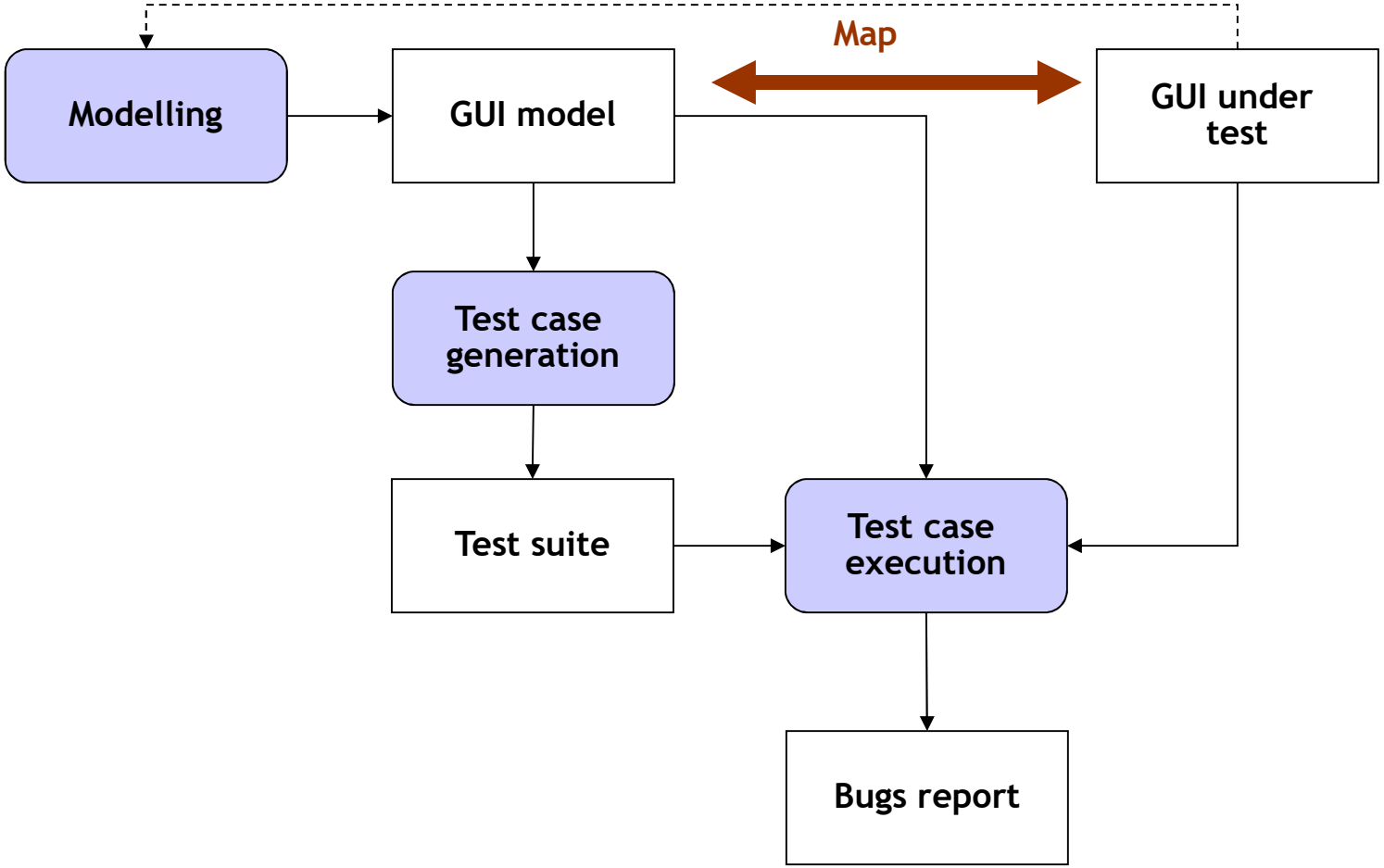
Random input testing tools

- Advantages
  - Good for finding system crashes
  - No effort in generating test cases
  - Independent of GUI updates
  - Increase confidence on the sw when running several hours without finding errors
  - “Easy” to implement
- Disadvantages
  - Not good for finding other kinds of errors
  - Difficult to reproduce the errors
  - Unpredictable

Unit testing frameworks

- GUI testing transformed into API testing
  - E.g., JUnit, NUnit
- Advantages
  - Used in combination with appropriate GUI test libraries (e.g. Maui for windows, Jemmy and Abbot for Java GUI applications), essentially reduce GUI testing to API testing
  - Flexibility
  - Automatic test execution
  - Support TDD (Test Driven Development)
- Disadvantages
  - Test cases are programmed manually
  - Appropriate GUI test libraries not always available

Model-based testing tools



Model-based testing tools

- Advantages
  - Higher degree of automation (test case generation)
  - Allows more exhaustive testing
  - Good for correctness/functional testing
  - Model can be easily adapted to changes
- Disadvantages
  - Requires a formal specification/model
  - Test case explosion
  - Test case generation has to be controlled appropriately to generate a test case of manageable size
  - Small changes to the model can result in a totally different test suite

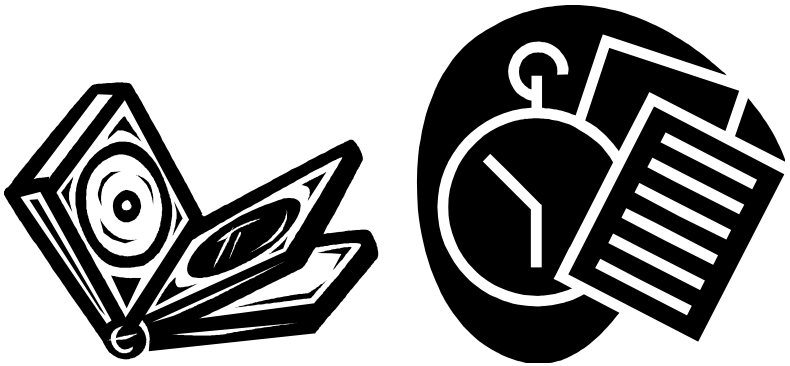
Automated GUI testing - vision

- Combination of approaches
  - ***Abstract layer*** separates logical names from physical properties of GUI objects and can be used in several approaches
  - ***GUI instrumentation library*** is reused by capture-replay tools, unit testing frameworks and model-based testing tools
  - ***Recording techniques*** describe how high-level user actions described in a model are mapped to concrete actions in the application (to see next)
  - ***Integrated environment*** supports all the testing automation approaches (also supporting automated static analysis)

GUI testing tools

- In the Marketplace:
  - CompuWare TestPartner - www.compuware.com
  - Rational Robot
  - Rational Visual Test - www.rational.com
  - WinRunner
  - LoadRunner - www.mercuryinteractive.com
  - Segue's SilkTest - www.segue.com
- Open Source
  - Abbot
  - GUITAR
  - Pounder
  - qftestJUI
  - Android - www.wildopensource.com/activities/larry-projects/android.php
  - GUI test drivers – www.testingfaqs.org/t-gui.html

Regression testing



Regression testing

- **regression testing:** A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software. [ISTQB]

- Note: Regression testing is not a testing level

Regression testing

- Regression testing is not a level of testing, but it is the *retesting of software* that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes **[Source: Burstein]**
- Regression tests are especially important when multiple software releases are developed **[Source: Burstein]**
- Sometimes the execution of all tests is not feasible so there is the need to select a subset of those tests in order to reduce the time for regression testing. Some techniques to support such selection are execution trace, execution slice and test prioritization **[source: Aditya P. Mathur]**

Regression testing

- Advantages
  - It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code.
  - It ensures that the bugs found earlier are NOT creatable.
  - Regression testing can be done by using the automation tools
  - It helps in improving the quality of the product.
- Disadvantages
  - If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again.
  - Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality.

Regression testing vs retesting

- **Regression testing** is carried out to ensure that the existing functionality is working fine and there are no side effects of any new change or enhancements done in the application. In other words, Regression Testing checks to see if new defects were introduced in previously existing functionality.
- **Retesting** is carried out in software testing to ensure that a particular defect has been fixed and its related functionality is working as expected.

Regression testing vs retesting

Regression testing	Retesting
To find issues which may be introduced because of any change or modification in the application	To confirm whether a failed test is working fine after being fixed
To check if changes did not introduce new bugs in existing functionality	To ensure that a particular bug was resolved and the functionality is working as exepected
Even passed tests are executed	Only failed test cases are reexecuted
To check for unexpected side effects	To ensure original issue is working as expected
Verification of bugs are not included in the regression testing	Verification of bugs are included in the retesting
Based on the project and availability of resourced, Regression testing can be carried out parallel with Retesting	Priority of retesting is higher than regression testing, so it is carried out before regression testing
...	

References and further reading

- R. Jeffries, J. R. Miller, C. Wharton, and K. M. Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques," 1991
- M. Y. Ivory and M. A. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces," *ACM Computing Surveys*, vol. 33, pp. 470-516, 2001.
- http://www.geocities.com/model_based_testing/
- Model-based testing tools
  - http://www-verimag.imag.fr/~async/TGV
  - http://research.microsoft.com/fse/AsmL
  - http://research.microsoft.com/SpecExplorer/
  - A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," Pittsburgh, 2001.
- N. Nyman, "Using Monkey Test Tools," in *STQE - Software Testing and Quality Engineering Magazine*, 2000.
- Unit testing frameworks - www.nunit.org; www.junit.org
- http://www.sqa-test.com/White_Paper.doc
- J. Edvardsson, "A Survey on Automatic Test Data Generation," 1999.
- http://research.microsoft.com/projects/specsharp/
- K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition / Decision Coverage," NASA/TM-2001-210876, 2001.

References and further reading

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- Software testing techniques (2nd ed.), by Boris Beizer, ISBN:0-442-20672-0, 1990
- A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing," presented at ICFEM'05, 2005.
- A. C. R. Paiva, J. C. P. Faria, and R. M. Vidal, "Automated Specification-based Testing of Interactive Components with AsmL," presented at 5th edition of the Quatic (Quality: the bridge to the future in ICT) international conference, Porto, 2004.
- A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal, "Modeling and Testing Hierarchical GUIs," presented at ASM 2005 - 12th International Workshop on Abstract State Machines, Paris - France, 2005.

TVVS - Test, Verification and Validation of Software

Test case design

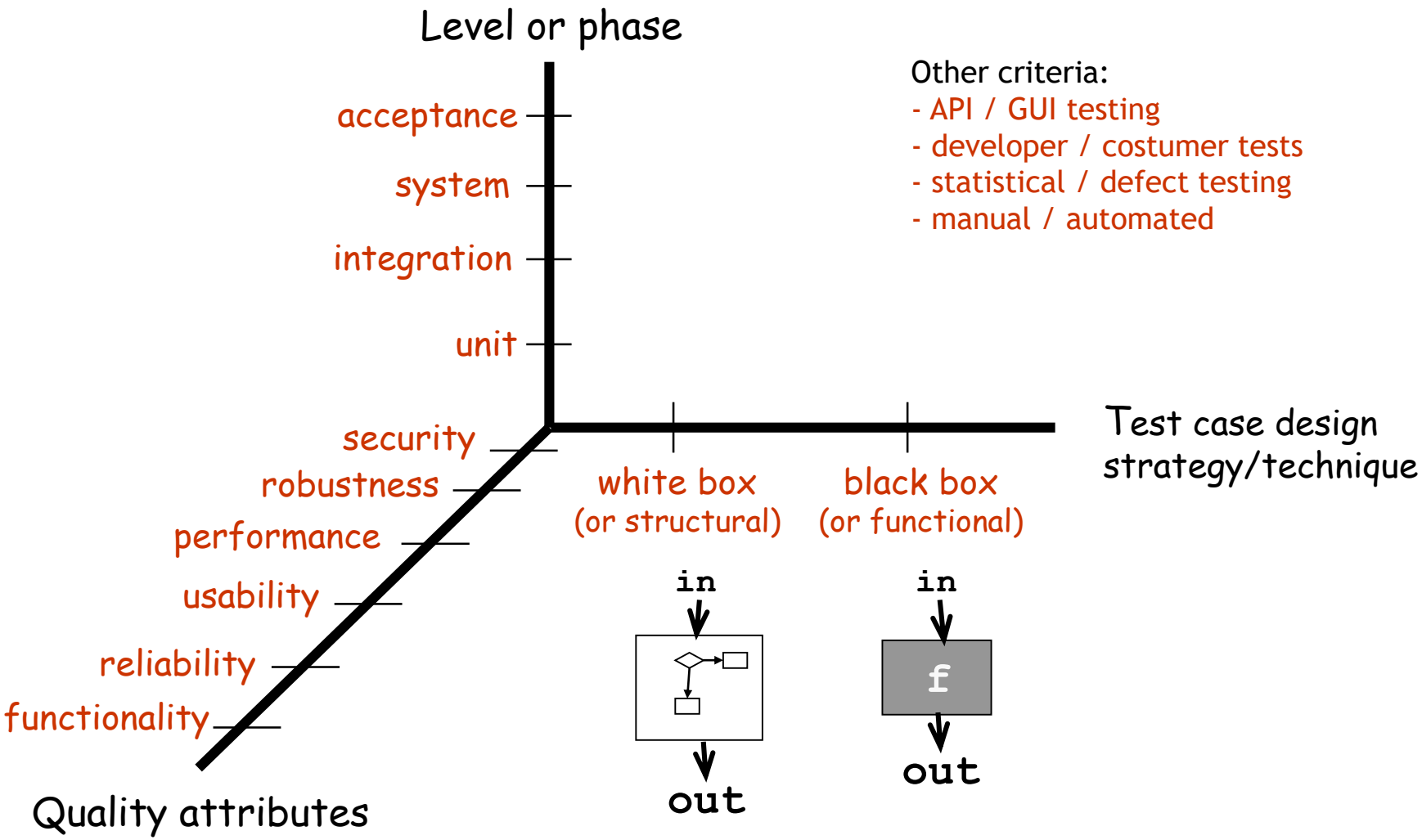
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- Introduction
- Black-box testing techniques
- Exercise

Test types (remember)



Other criteria:
- API / GUI testing
- developer / costumer tests
- statistical / defect testing
- manual / automated

Test cases (remember)

- **test case**
  - (1) 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.

    [according to ISTQB]

  - **(2)** (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item

    [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]

- **test condition**: 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.[according to ISTQB]

Test adequacy (remember)

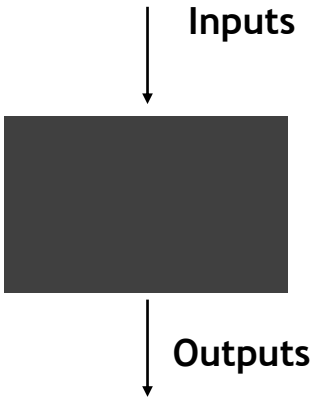
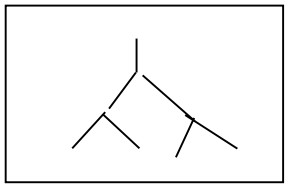
- Adequacy criteria - Criteria to decide if a given test suite is adequate, i.e., to give us “enough” confidence that “most” of the defects are revealed
  - Used in the evaluation and in the design/selection of test cases
  - In practice, reduced to coverage criteria

Coverage criteria (remember)

- Coverage criteria
  - Requirements/specification coverage (black-box)
    - At least one test case for each requirement/specification statement
  - Code coverage (white-box)
    - Control flow coverage (statement, decision, MC/DC coverage …)
    - Data flow coverage
  - Model coverage
    - State-transition coverage
    - Use case and scenario coverage
  - Fault coverage

See also: "Software Unit Test Coverage and Adequacy", Hong Zhu et al, ACM Computing Surveys, December 1997

Test case design strategies and techniques

Strategy	Tester's View	Knowledge sources	Techniques / Methods
Black-box testing (not code-based) (sometimes called functional testing)		Requirements document Specifications User manual Models Domain knowledge Defect analysis data Intuition Experience	Equivalence class partitioning Boundary value analysis Cause effect graphing Pairwise testing Error guessing Random testing State-transition testing Scenario-based testing
White-box testing (also called code-based or structural testing)		Program code Control flow graphs Data flow graphs Cyclomatic complexity High-level design Detailed design	Control flow testing/coverage: • Statement coverage • Branch (or decision) coverage • Condition coverage • Branch and condition coverage • Modified condition/decision coverage • Multiple condition coverage • Independent path coverage • Path coverage • LCSAJ coverage Data flow testing/coverage Class testing/coverage Mutation testing

(adapted from: I. Burnstein, pg.65)

Test automation

- Automatic test case generation
  - Automatic generation of test inputs is easier than the automatic generation of test outputs (usually requires a formal specification)
  - May reduce test development costs
  - Usually, lower capability to find failures per test case, but overall capability may be higher because more test cases can be generated (than manually)
- Automatic test case execution
  - Requires that test cases are written/converted in/into some executable language
  - Increases test development costs (coding) but practically eliminates test (re)execution costs, which is particularly important in regression testing
  - Unit testing frameworks and tools for API testing; Capture/replay tools for GUI testing
  - Performance testing tools

Agenda

- Introduction
- Black box testing techniques
- Exercise

Black-box testing

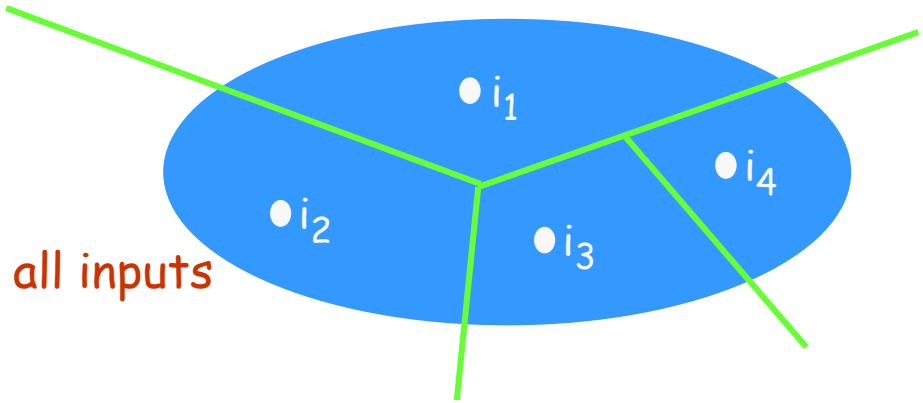
- Equivalence class (or domain) partitioning
- Boundary-value analysis
- Cause-effect graphing (or decision table)
- Pairwise testing
- Error guessing
- Risk based testing
- Testing for race conditions
- Random testing
- Requirements based testing
- Model-based testing (MBT)
- …

Equivalence class partitioning

Divide all possible inputs into classes (partitions) such that :

- There is a finite number of input equivalence classes
- You may reasonably assume that
  - the program behaves analogously for inputs in the same class
  - one test with a representative value from a class is sufficient
  - if the representative detects a defect
    then other class members would detect the same defect

(Can also be applied to outputs)

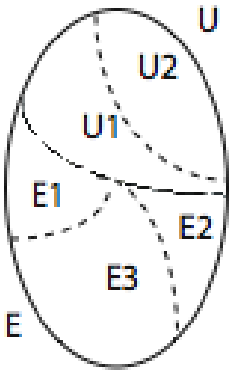


Equivalence class partitioning

Faults targeted

- The entire set of inputs to any application can be divided into at least two subsets: one containing all the expected, or legal, inputs (E) and the other containing all unexpected, or illegal, inputs (U).

- Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g., E1, E2, E3, and U1, U2).



© Aditya P. Mathur 2006

Sistematic procedure for equivalence partitioning (1)

- **1. Identify the input domain:** Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.
  - Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.
- **2. Equivalence classing:** Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable, is done based on the expected behavior of the program.
  - Values for which the program is expected to behave in the "same way" are grouped together. Note that "same way" needs to be defined by the tester.

© Aditya P. Mathur 2006

Sistematic procedure for equivalence partitioning (2)

- **3. Combine equivalence classes:** This step is usually omitted and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.
  - The equivalence classes are combined using the multidimensional partitioning approach described earlier.
- **4. Identify infeasible equivalence classes:** An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test. Such an equivalence class might arise due to several reasons.
  - For example, suppose that an application is tested via its GUI, i.e., data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.

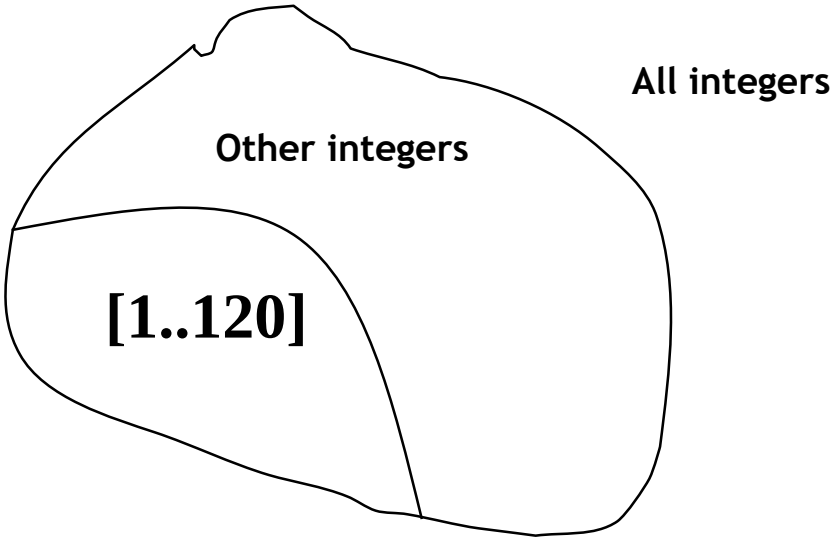
© Aditya P. Mathur 2006

Equivalence class partitioning

- Identify input equivalence classes
  - Based on conditions on inputs/outputs in specification/description
  - Both valid and invalid input equivalence classes
  - Based on heuristics and experience, E.g.,:
    - "input x in [1..10]" $\rightarrow$ classes: $x < 1$, $1 \leq x \leq 10$, $x > 10$
    - "enumeration A, B, C" $\rightarrow$ classes: A, B, C, not$\{A,B,C\}$
    - "input integer n" $\rightarrow$ classes: n not an integer, $n<\min$, $\min \leq n < 0$, $0 \leq n \leq \max$, $n > \max$
- Define one (or a couple of) test cases for each class
  - Test cases that cover valid classes (1 test case for 1 or more valid classes)
  - Test cases that cover at most one invalid class (1 test case for 1 invalid class)
  - Usually useful to test for 0/null/empty and other special cases
- Combine equivalent classes
  - Combine valid with invalid values to increase the capability of detecting missing code
- Identify infeasible classes

Equivalence class partitioning - Example 1

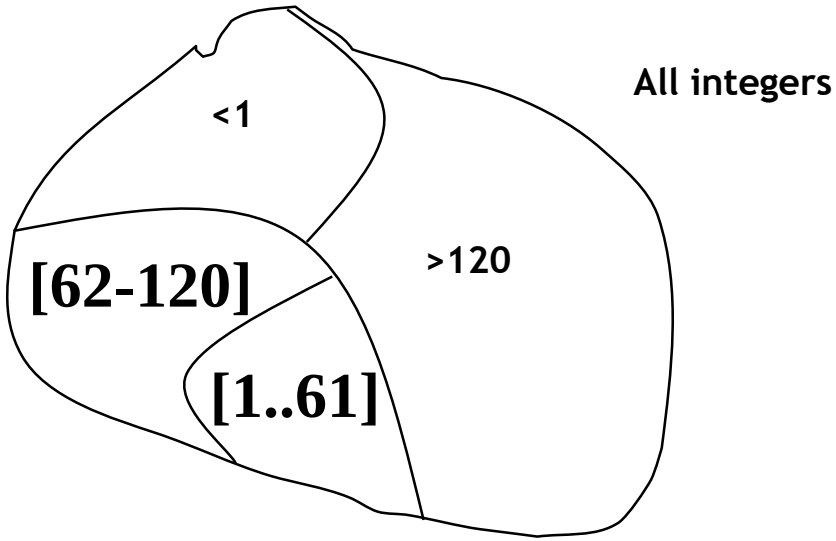
- Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.



© Aditya P. Mathur 2006

Equivalence class partitioning - Example 1

- Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2. Thus E is further subdivided into two regions depending on the expected behavior.
- Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.



© Aditya P. Mathur 2006

Equivalence class partitioning - Example 1

- Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e., two regions containing expected inputs and two regions containing the unexpected inputs.

- It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

© Aditya P. Mathur 2006

Equivalence class partitioning - Example 2

Test a function that calculates the absolute value of an integer *x*

- Equivalence classes :

Criteria	Valid eq. classes	Invalid eq. classes
nr of inputs	1 (1)	0 (2), > 1 (3)
input type	integer (4)	non-integer (5)
particular x	< 0 (6), >= 0 (7)	

- Test cases :

x = -10 (1,4,6)

x = 100 (1,4,7)

x = (2)

x = 10 20 (3)

x = "XYZ" (5)

Equivalence class partitioning - Example 3

- Test a program that computes the *sum* of the first *N* integers as long as this *sum* is less than *maxint*. Otherwise an error should be reported. If *N* is negative, then it takes the absolute value *N*.

- Formally:
  Given integer inputs *N* and *maxint* compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this} <= \text{maxint, error otherwise}$$

Equivalence class partitioning - Example 3

- Equivalence classes:

Condition	Valid eq. classes	Invalid eq. classes
nr of inputs	2	< 2, > 2
type of input	int int	int no-int, no-int int, no-int no-int
abs(N)	$N < 0$, $N \geq 0$	
maxint	$\sum k \leq$ maxint, $\sum k >$ maxint	

- Test Cases :

	maxint	N	result
Valid	100	10	55
	100	-10	55
	10	10	error
Invalid	10	-	error
	10 20	30	error
	"XYZ"	10	error
	100	9.1E4	error

Equivalence classes based on program output

- In some cases the equivalence classes are based on the output generated by the program. For example, suppose that a program outputs an integer.
- It is worth asking: "Does the program ever generate a 0? What are the maximum and minimum possible values of the output?"
- These two questions lead to the two following equivalence classes based on outputs:
  - E1: Output value v is 0.
  - E2: Output value v is the maximum possible.
  - E3: Output value v is the minimum possible.
  - E4: All other output values.
- Based on the output equivalence classes one may now derive equivalence classes for the inputs. Thus each of the four classes given above might lead to one equivalence class consisting of inputs.

© Aditya P. Mathur 2006

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	**speed ∈[60..90]**	**{50}, {75}, {92}**
	area: float area≥0.0	**{{-1.0}, {15.52}}**
	age: int	**{{-1}, {56}, {132}}**
	letter: char	**{{J}, {3}}**

© Aditya P. Mathur 2006

Equivalence classes for variables: strings

Eq. Classes	Example	
	Constraints	**Classes**
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Loooong Name}}

© Aditya P. Mathur 2006

Equivalence classes for variables: enumeration

Eq. Classes	Example	
	Constraints	**Classes**
Each value in a separate class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	up:boolean	{{true}, {false}}

© Aditya P. Mathur 2006

Equivalence classes for variables: arrays

Eq. Classes	Example	
	Constraints	**Classes**
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [ ] aName: new int[3];	{[ ]}, {[-10, 20]}, {[-9, 0, 12, 15]}

© Aditya P. Mathur 2006

Equivalence classes for variables: compound data types

- Arrays in Java and records, or structures, in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object.
- While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure.

```
struct transcript{
        string fName; // First name.
        string lName; //  Last name.
        string cTitle [200]; // Course titles.
        char grades [200]; // Letter grades corresponding
                              to course titles.
}
```

© Aditya P. Mathur 2006

Partitioning

- Unidimensional partitioning
  - One way to partition the input domain is to consider one input variable at a time. Thus each input variable leads to a partition of the input domain. We refer to this style of partitioning as unidimensional equivalence partitioning or simply unidimensional partitioning.

  This type of partitioning is commonly used.

- Multidimensional partitioning
  - Another way is to consider the input domain *I* as the set product of the input variables and define a relation on *I*. This procedure creates one partition consisting of several equivalence classes. We refer to this method as multidimensional equivalence partitioning or simply multidimensional partitioning.
  - Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.

© Aditya P. Mathur 2006

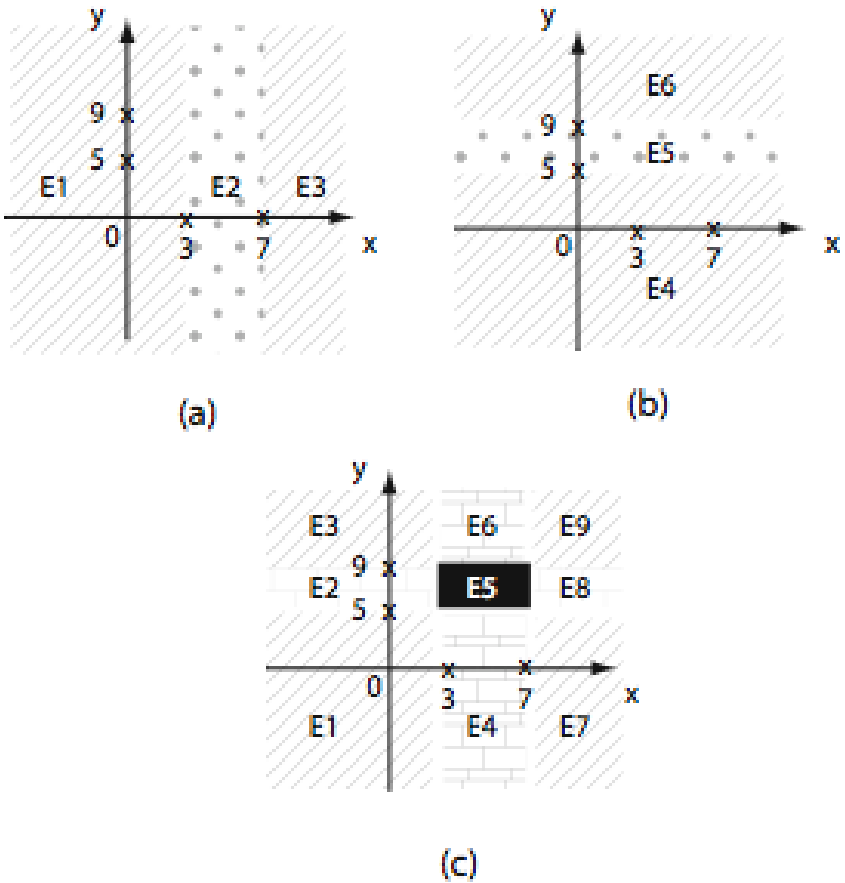
Partitioning Example (1)

- Consider an application that requires two integer inputs *x* and *y*. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.
- For unidimensional partitioning we apply the partitioning guidelines to *x* and *y* individually. This leads to the following six equivalence classes.
- E1: $x<3$ E2: $3 \leq x \leq 7$ E3: $x>7$ (y ignored)
- E4: $y<5$ E5: $5 \leq y \leq 9$ E6: $y>9$ (x ignored)
- For multidimensional partitioning we consider the input domain to be the set product X x Y. This leads to 9 equivalence classes.

© Aditya P. Mathur 2006

Partitioning Example (2)

- 9 equivalent classes
  - E1: x<3, y<5
  - E3: x<3, y>9
  - E2: x<3, 5≤y≤9
  - E4: 3≤x≤7, y<5
  - E5: 3≤x≤7, 5≤y≤9
  - E6: 3≤x≤7, y>9
  - E7: x>7, y<5
  - E8: x>7, 5≤y≤9
  - E9: x>7, y>9



© Aditya P. Mathur 2006

Boundary value analysis

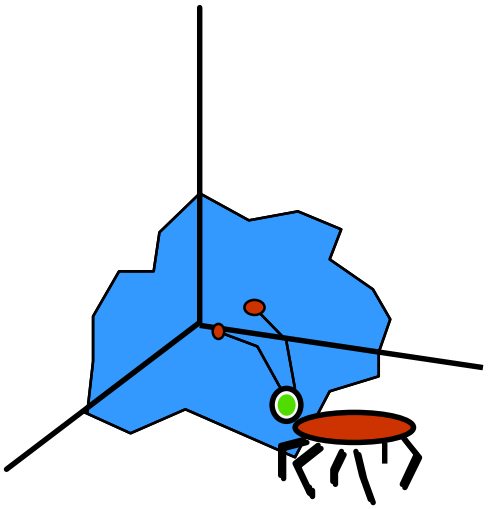
Based on experience / heuristics :

- Testing boundary conditions of equivalence classes is more effective, i.e., values directly on, above, and beneath edges of classes
  - If a system behaves correctly at boundary values, than it probably will work correctly at "middle" values
- Choose input boundary values as tests in input classes instead of, or additional to arbitrary values
- Choose also inputs that invoke output boundary values (values on the boundary of output classes)
- Example strategy as extension of equivalence class partitioning:
  - choose one (or more) arbitrary value(s) in each equivalent class
  - choose values exactly on lower and upper boundaries of equivalent classes
  - choose values immediately below and above each boundary (if applicable)

Boundary value analysis

“Bugs lurk in corners and congregate at boundaries.”

[Boris Beizer, "Software testing techniques"]



Boundary value analysis
Example 1

Test a function for calculation of absolute value of an integer

- Valid equivalence classes :

Condition	Valid eq. classes	Invalid eq. Classes
particular abs	< 0, >= 0	

- Test cases :

class x < 0, arbitrary value: x = -10

class x >= 0, arbitrary value x = 100

classes x < 0, x >= 0, on boundary : x = 0

classes x < 0, x >= 0, below and above: x = -1, x = 1

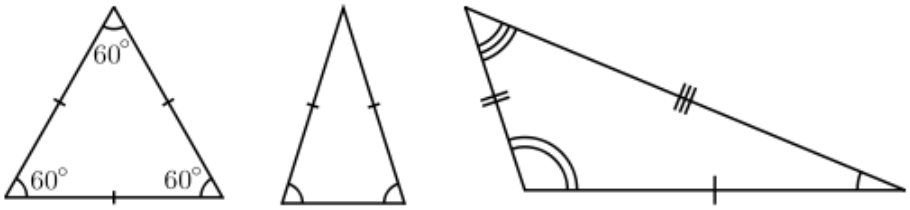
Boundary value analysis
A self-assessment test 1 [Myers]

- "A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene (all lengths are different), isosceles (two lengths are equal), or equilateral (all lengths are equal)."

Write a set of test cases to test this program.

Inputs: l_1, l_2, l_3 , integer, $l_i > 0$, $l_i < l_j + l_k$

Output: error, scalene, isosceles or equilateral



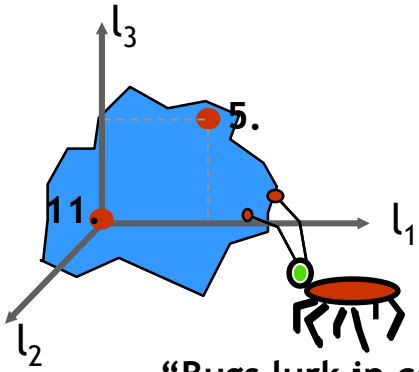
Equilateral Isosceles Scalene

Boundary value analysis
A self-assessment test 1 [Myers]

Test cases for:

valid inputs:

1. valid scalene triangle ?
2. valid equilateral triangle ?
3. valid isosceles triangle ?
4. 3 permutations of previous ?



"Bugs lurk in corners and congregate at boundaries."

invalid inputs:

5. side = 0 ? (boundary "plane")
6. negative side ?
7. one side is sum of others ? (boundary)
8. 3 permutations of previous ?
9. one side larger than sum of others ?
10. 3 permutations of previous ?
11. all sides = 0 ? (boundary "corner")
12. non-integer input ?
13. wrong number of values ?

Boundary value analysis
Example 2

- Given inputs maxint and N compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this <= maxint, error otherwise}$$

- Valid equivalence classes :

condition	valid eq. classes	boundary values.
abs(N)	$N < 0$, $N \geq 0$	N = (-2), -1, 0, 1
maxint	$\sum k \leq$ maxint,	$\sum k$ = maxint-1,
	$\sum k >$ maxint	maxint,
		maxint+1,
		(maxint+2)

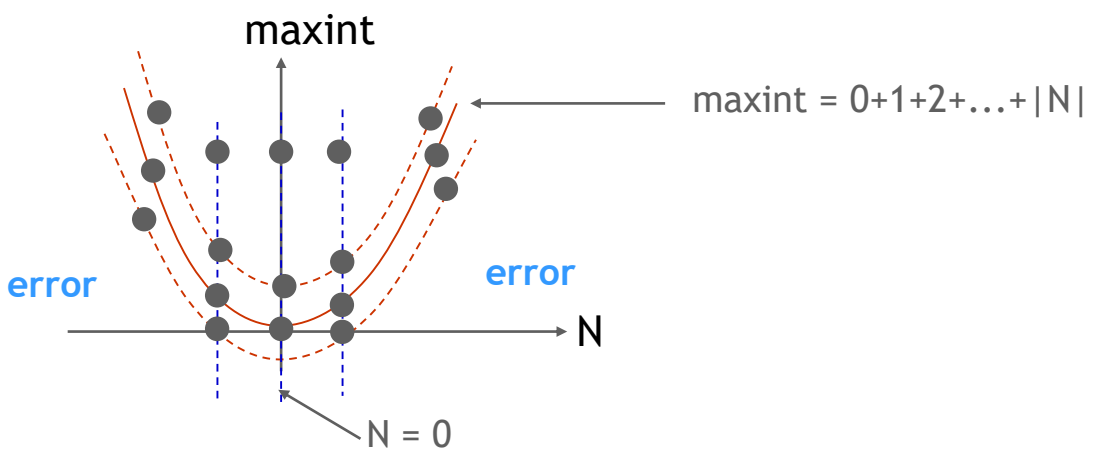
Boundary value analysis
Example 2

- Test Cases :

maxint	*N*	result
55	10	55
54	10	error
56	10	55
0	0	0

maxint	*N*	result
100	0	0
100	-1	1
100	1	1
…	…	…

- How to combine the boundary conditions of different inputs ?
  Take all possible boundary combinations ? This may blow up ……



Boundary value analysis
Example 3: search routine specification

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;
      Found : out BOOLEAN; L: out ELEM_INDEX) ;

Pre-condition
            -- the array has at least one element
            T'FIRST <= T'LAST
Post-condition
            -- the element is found and is referenced by L
            ( Found and T (L) = Key)
or
            -- the element is not in the array
            ( not Found and
            not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

(source: Ian Sommerville)

Boundary value analysis
Example 3 - input partitions

- P1 - Inputs which conform to the pre-conditions (valid)
  - array with 1 value (boundary)
  - array with more than one value (different size from test case to test case)
- P2 - Inputs where a pre-condition does not hold (invalid)
  - array with zero length
- P3 - Inputs where the key element is a member of the array
  - first, last and middle positions in different test cases
- P4 - Inputs where the key element is not a member of the array

Boundary value analysis

Example 3 – test cases (valid cases only)

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

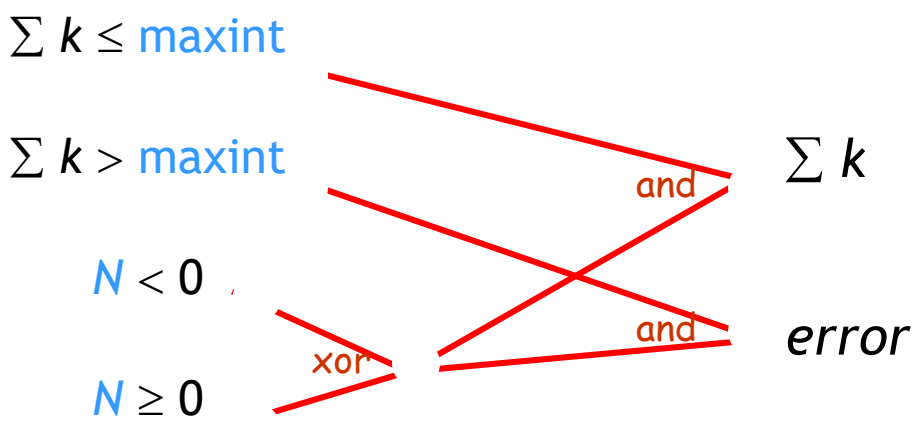
Cause-effect graphing

- Black-box technique to analyze combinations of input conditions
- Identify causes and effects in specification

  causes ↓ inputs / initial state conditions

  effects ↓ outputs / final state conditions
- Make Boolean Graph linking causes and effects
- Annotate impossible combinations of causes and effects
- Develop decision table from graph with in each column a particular combination of inputs and outputs
- Transform each column into test case

Cause-effect graphing
Example 2



Decision table

("truth table")

Each entry in de decision table is a 0 or a 1 depending on whether or not the corresponding condition is false or true

causes	$\sum k \leq$ maxint	1	1	0	0
(inputs)	$\sum k >$ maxint	0	0	1	1
	$N < 0$	1	0	1	0
	$N \geq 0$	0	1	0	1
effects	$\sum k$	1	1	0	0
(outputs)	*error*	0	0	1	1

Cause-effect graphing

- Systematic method for generating test cases representing combinations of conditions
- Differently from eq. class partitioning, we define a test case for each possible combination of conditions
- Combinatorial explosion of number of possible combinations
  - In the worst case, if *n* causes are related to an effect *e*, then the maximum number of combinations that bring *e* to a *1-state* is 2^n.

Pairwise testing

- **pairwise testing:** A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters.
- The number of pairwise test cases will be O(*mn*), where *m* and *n* are the number of possibilities for each of the two parameters with the most choices.

$$O(mn) = Max(X) \times Max(X \backslash Max(X))$$

and X is the set with the numbers of possibilities for every parameters

Pairwise testing

Parameter Name	Value 1	Value 2	Value 3	Value 4
Enabled	True	False	*	*
Choice Type	1	2	3	*
Category	a	b	c	d

$X = \{2,3,4\}$

$O(mn) = 4 \times 3 = 12$

Enabled	Choice Type	Category
True	3	a
True	1	d
False	1	c
False	2	d
True	2	c
False	2	a
False	1	a
False	3	b
True	2	b
True	3	d
False	3	c
True	1	b

Error guessing

- Just ‘guess’ where the errors are ……
- Intuition and experience of tester
- Ad hoc, not really a technique
- But can be quite effective
- Strategy:
  - Make a list of possible errors or error-prone situations (often related to boundary conditions)
  - Write test cases based on this list

Risk based testing

- **Risk-based testing** (RBT) is a type of software testing that prioritizes the features and functions to be tested based on priority/importance and likelihood or impact of failure. More sophisticated 'error guessing'

- Try to identify critical parts of program (high risk code sections):
  - parts with unclear specifications
  - developed by junior programmer while his wife was pregnant ……
  - complex code :
    measure code complexity - tools available (McGabe, Logiscope,…)

- High-risk code will be more thoroughly tested
  (or be rewritten immediately ……)

Testing for race conditions

- Also called bad timing and concurrency problems
- Problems that occur in multitasking systems (with multiple threads or processes)
- A kind of boundary analysis related to the dynamic views of a system (state-transition view and process-communication view)
- Examples of situations that may expose race conditions:
  - problems with shared resources:
    - saving and loading the same document at the same time with different programs
    - sharing the same printer, communications port or other peripheral
    - using different programs (or instances of a program) to simultaneously access a common database
  - problems with interruptions:
    - pressing keys or sending mouse clicks while the software is loading or changing states
  - other problems:
    - shutting down or starting two or more instances of the software at the same time
- Knowledge used: dynamic models (state-transition models, process models)

[source: Ron Patton]

Random testing

- Input values are randomly generated
- Do you know that a monkey using a piano keyboard could play a Vivaldi opera? Could the same monkey, using your application, discovery defects?
- Two kinds of tools
  - **Dumb monkeys** – low IQ; they can't recognize an error when they see one
  - **Smart monkeys** – generate inputs with some knowledge to reflect expected usage; get knowledge from state table or model of the AUT.
- Microsoft says that 10 to 20% of the bugs in Microsoft projects are found by these tools

Random testing

- Advantages
  - Good for finding system crashes
  - Particularly adequate for performance testing (it's not necessary to check the correctness of outputs)
  - No effort in generating test cases
  - Independent of updates
  - Increase confidence on the software when running several hours without finding errors
  - “Easy” to implement
- Disadvantages
  - Not good for finding other kinds of errors
  - Difficult to reproduce the errors (repeat test cases / sequence of inputs)
  - Unpredictable
  - May not cover special cases that are discovered by "manual" techniques

Deriving test cases from requirements and use cases

- Particularly adequate for system and acceptance testing
- From requirements:
  - You have a list of requirements
  - Define at least one test case for each requirement
  - Build and maintain a (tests to requirements) traceability matrix
- From use cases:
  - You have use cases that capture functional requirements
  - Each use case is described by one or more *normal* flow of events and zero or more *exceptional* flow of events
  - Define at least one test case for each flow of events (also called *scenario*)
  - Build and maintain a (tests to use cases) traceability matrix

State-transition testing

- Construct a state-transition model (state machine view) of the item to be tested (from the perspective of a user/client). E.g., with a state diagram in UML
- Define test cases to exercise all states and all transitions between states
  - Usually, not all possible paths (sequences of states and transitions), because of combinatorial explosion
  - Each test case describes a sequence of inputs and outputs (including input and output states), and may cover several states and transitions
  - Also test to fail – with unexpected inputs for a particular state
- We will talk about this techniques in more detail in the following lectures

Black box testing: Which One ?

- Black box testing techniques :
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Error guessing
  - ……
- Which one to use ?
  - None of them is complete
  - All are based on some kind of heuristics
  - They are complementary

Black box testing: which one ?

- Always use a combination of techniques
  - When a formal specification is available try to use it
  - Identify valid and invalid input equivalence classes
  - Identify output equivalence classes
  - Apply boundary value analysis on valid equivalence classes
  - Guess about possible errors
  - Cause-effect graphing for linking inputs and outputs

Index

- Introduction
- Black box testing techniques
- Exercise

Exercise

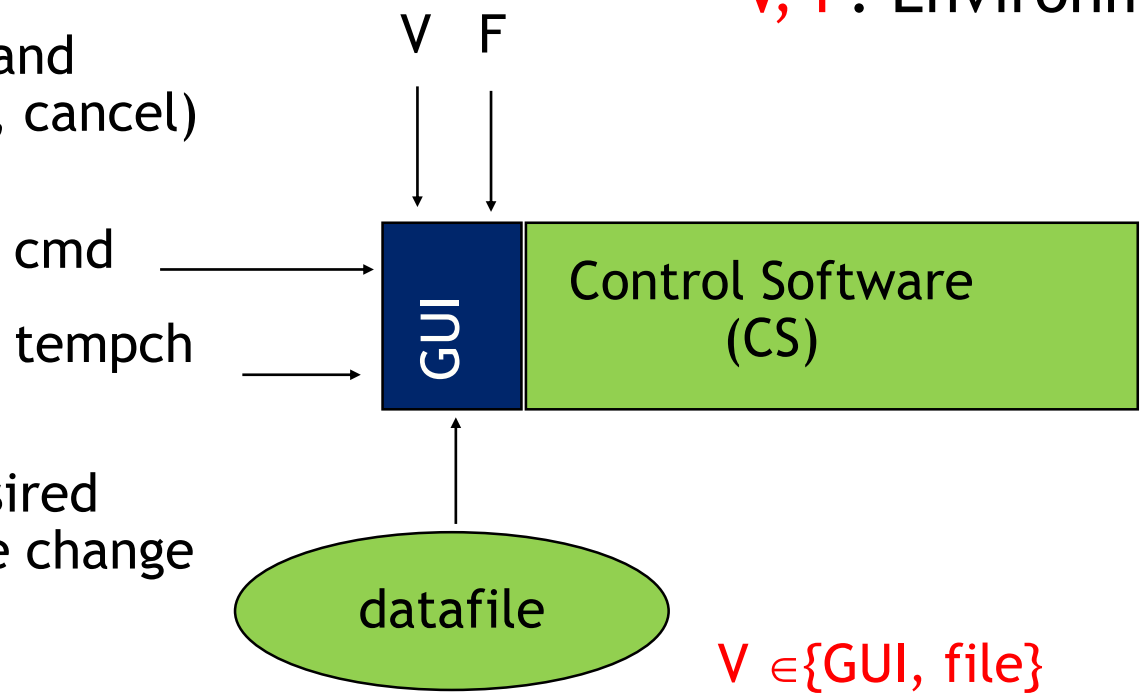
- The control software of BCS, abbreviated as CS, allows a human operator to give one of three commands (*cmd*): change the boiler temperature (*temp*), shut down the boiler (*shut*), and cancel the request (*cancel*).
- Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (*tempch*). Values of *tempch* are in the range [-10..10] in increments of 5 degrees Fahrenheit. An temperature change of 0 is not an option.
- BCS examines variable *V*. If *V* is set to *GUI*, the operator is asked to enter one of the three commands via a *GUI*. However, if *V* is set to *file*, BCS obtains the command from a command file.
- The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is *temp*. The file name is obtained from variable *F*.

© Aditya P. Mathur 2006

Exercise

V, F: Environment variables

cmd: command
(temp, shut, cancel)



tempch: desired temperature change (-10..10)

V ∈{GUI, file}

F: file name if V is set to “file.”

© Aditya P. Mathur 2006

Exercise

- Identify input domain
- Identify equivalence classes
- Combine equivalence classes
- Discard infeasible equivalence classes
- Generate sample tests for BCS from the remaining feasible equivalence classes

© Aditya P. Mathur 2006

Identify input domain

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	{GUI, file}
F	Environment	String	A file name
cmd	Input via GUI or file	Enumerated	{temp,cancel,shut}
tempch	Input via GUI or file	Enumerated	{-10,-5,5,10}

S = V x F x cmd x tempch

Equivalence class partition

Variable	Partition
V	{{GUI},{file},{undefined}}
F	f_valid, f_invalid
cmd	{{temp},{cancel},{shut},{c_invalid}}
tempch	{{-10},{-5},{5},{10},{t_invalid}}

Combine equivalence classes

- Variables V, F, cmd and tempch have been partitioned into 3, 2, 4 and 2 susets, respectively
- Set products of these four variables leads to a total of 3x2x4x5=120

Discard infeasable classes

- The amount by which the boiler temperature is to be changed is needed only when the operator selects **temp** for **cmd**, Thus all equivalent classes that match the following template are infeasible
  - {(V, F, {cancel,shut,c_invalid}, t_valid U t_invalid)} =
- GUI does not allow invalid values of temperature change to be input. Two more equivalent classes infeasible
  - {(GUI, F, temp, t_invalid)} =

Discard infeasable classes

- Carefully designed application might not ask for the values of tempch when **V=file** and **F** contains a file name that does not exist
  - {(file, f_invalid, temp, t_valid U t_invalid)} =
- Application will not allow values of **tempch** to be input when **V** is undefined
  - {(undefined, -, temp, t_valid U t_invalid)} =
- Discard 90+2+5+5=102 equivalence classes
- 18 equivalent classes remain

Result

{(GUI, f_valid, temp, t_valid)}	= 4
{(GUI, f_invalid, temp, t_valid)}	= 4
{(GUI, –, cancel, NA)}	= 2
{(file, f_valid, temp, t_valid U t_invalid)}	= 5
{(file, f_valid, shut, NA)}	= 1
{(file, f_invalid, NA, NA)}	= 1
{(undefined, NA, NA, NA)}	= 1
	total = 18

'–' means that data can be input but is not used by the software

'NA' means that data cannot be input to the control software

References

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- The Art of Software Testing, Glenford J. Myers, Wiley & Sons, 1979 (Chapter 4 - Test Case Design)
  - Classical
- Software testing techniques, Boris Beizer, Van Nostrand Reinhold, 2nd Ed, 1990
  - Bible
- Foundations of Software Testing, 1st edition, by Aditya P. Mathur, Pearson Education, 2008
- Testing Computer Software, 2nd Edition, Cem Kaner, Jack Falk, Hung Nguyen, John Wiley & Sons, 1999
  - Practical, black box only
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- Test Driven Development, Kent Beck, Addison Wesley, 2002
- http://www.swebok.org/
  - Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE Computer Society
- http://www.mccabe.com/iq_research_metrics.htm
- What Is a Good Test Case?, Cem Kaner, Florida Institute of Tecnology, 2003
- Software Unit Test Coverage and Adequacy, Hong Zhu et al, ACM Computing Surveys, December 1997

TVVS - Test, Verification and Validation of Software

Security Testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Security Testing

- Security testing is a process to determine that an information system protects data and maintains functionality
- To check whether there is any information leakage
- To test the application whether it has unauthorized access and having the encoded security code
- To finding out the potential loopholes and weaknesses of the system

Security Testing

Security Testing is (according to ISTQB): Testing to determine the security of the software product

Types:

- Vulnerability Scanning
- Security Scanning
- Penetration Testing
- Risk Assessment
- Security Auditing
- Ethical Hacking
- Password cracking
- (…)

Security Testing

- **Vulnerability Scanning**: This is done through automated software to scan a system against known vulnerability signatures.
- **Security Scanning:** It involves identifying network and system weaknesses, and later provides solutions for reducing these risks. This scanning can be performed for both Manual and Automated scanning.
- **Penetration testing**: This kind of testing simulates an attack from a malicious hacker. This testing involves analysis of a particular system to check for potential vulnerabilities to an external hacking attempt.

Security Testing

- **Risk Assessment:** This testing involves analysis of security risks observed in the organization. Risks are classified as Low, Medium and High. This testing recommends controls and measures to reduce the risk.
- **Security Auditing:** This is an internal inspection of Applications and Operating systems for security flaws. An audit can also be done via line by line inspection of code
- **Ethical hacking:** It's hacking an Organization Software systems. Unlike malicious hackers, who steal for their own gains, the intent is to expose security flaws in the system.
- **Password Cracking**: Programs can be used to identify weak passwords

Security Concepts

- Confidentiality
- Integrity
- Authentication
- Authorization
- Availability
- Non-repudiation

Security Concepts

- Confidentiality
  - Ensuring information is accessible only for those with authorized access and to prevent information theft
- Integrity
  - A measure intended to allow the receiver to determine that the information which it is providing is correct
- Authentication
  - The process of establishing the identity of the user

Security Concepts

- Authorization
  - The process of determining that a requester is allowed to receive a service or perform an operation
- Availability
  - Assuring information and communications services will be ready for use when expected
- Non-repudiation
  - A measure intended to prevent the later denial that an action happened, or communication that took place

Some Tools...

- OWASP
- WireShark
- W3af
- Nessus
- Nikto
- Gendarme
- Flawfinder
- …

TVVS - Test, Verification and Validation of Software

Test case design

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

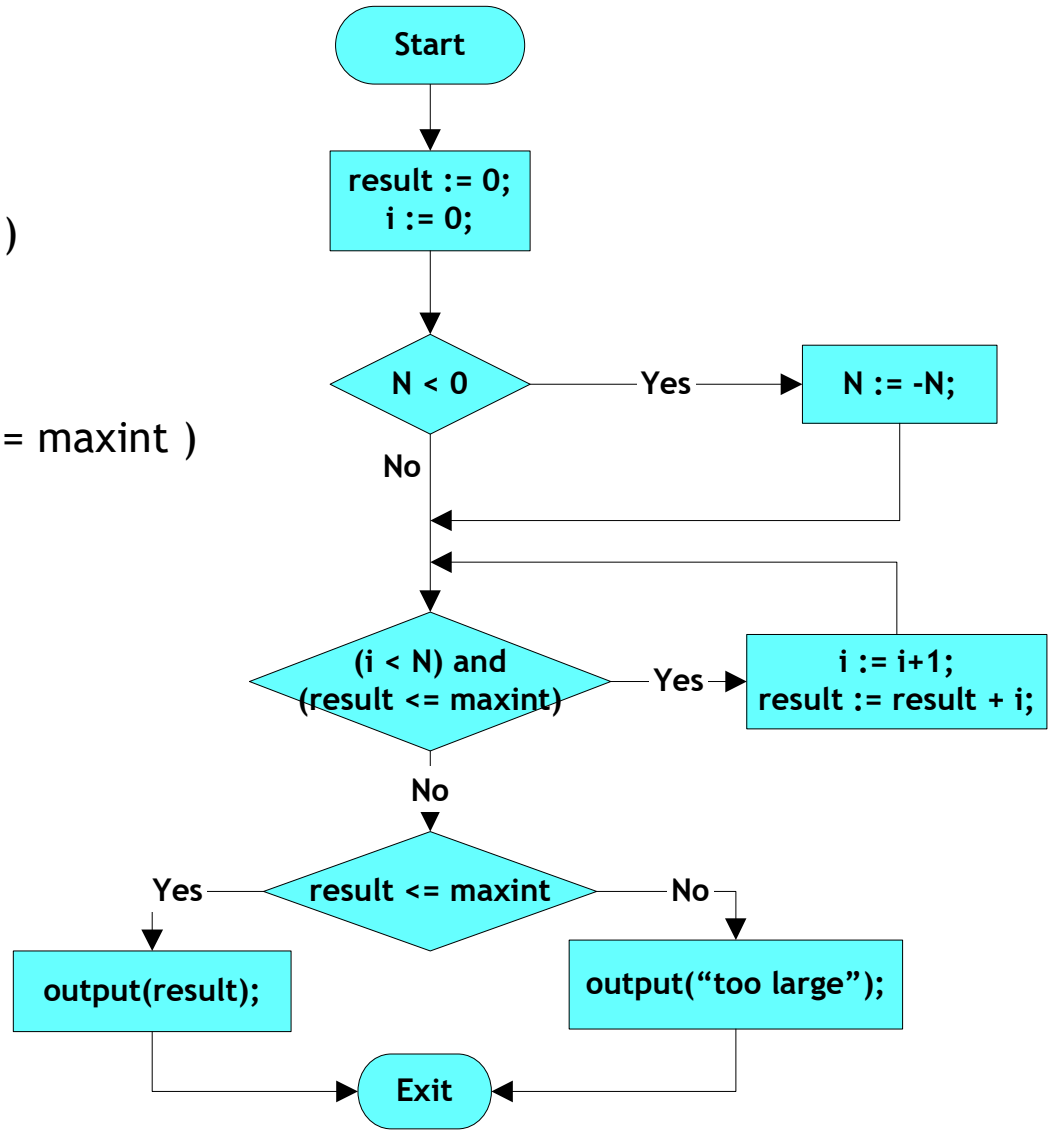
White-box testing

- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Used mainly for unit testing
- Programming language dependent
- Extent to which (source) code is executed, i.e., covered
- Different kind of coverage :
  - based on control flow analysis - statement, decision, condition, decision and condition, MC/DC, path, ...
  - based on data flow analysis – p-use, c-use, ...

Control flow analysis
Example

```
PROGRAM  sum ( maxint, N : INT )
  INT    result := 0 ;   i := 0 ;
  IF   N < 0
  THEN   N  :=  - N ;
  WHILE   ( i < N )  AND  ( result <= maxint )
  DO   i  :=  i + 1 ;
          result  :=  result + i ;
  OD;
  IF   result <= maxint
  THEN   OUTPUT ( result )
  ELSE   OUTPUT ( “too large” )
END.
```

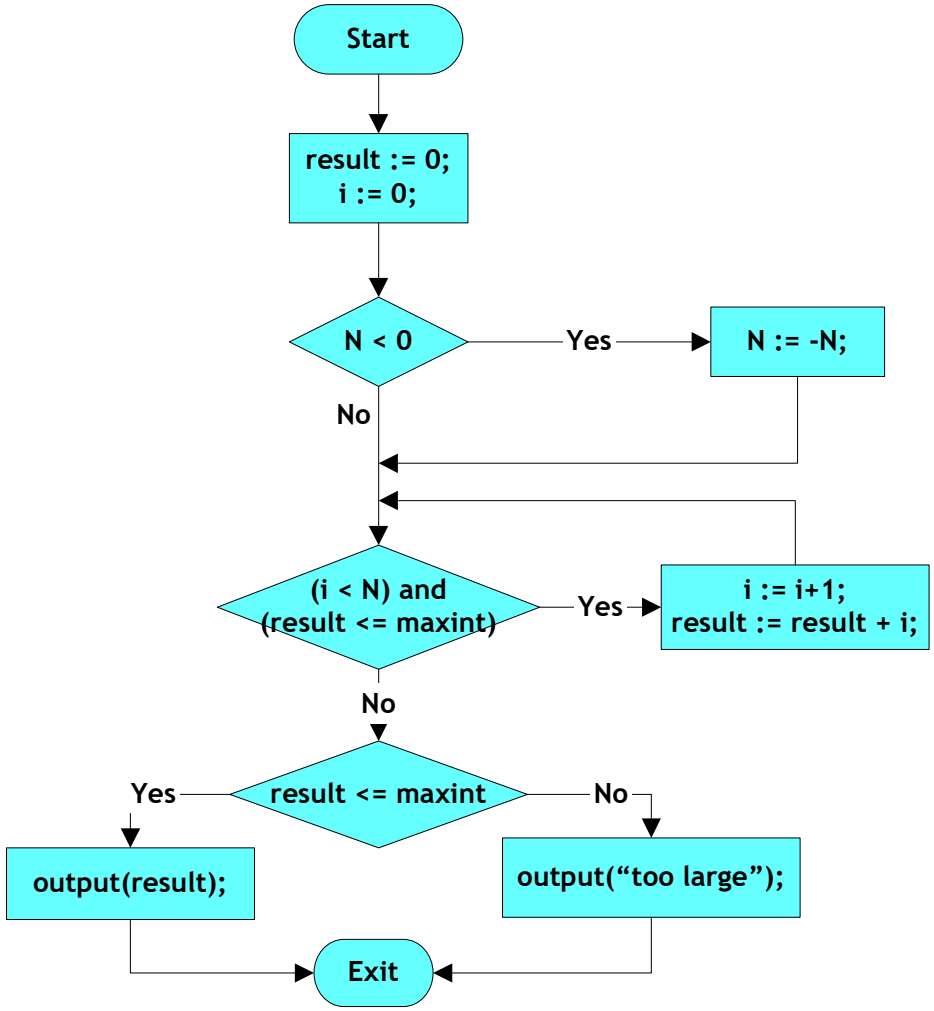
$$\text{result} = \begin{cases} \sum_{K=0}^{|N|} k, & \text{if this} <= \text{maxint} \\ \text{error}, & \text{otherwise} \end{cases}$$



Statement coverage

- Execute (exercise) every statement of a program
  - Generate a set of test cases such that each statement of the program is executed at least once
- Weakest white-box criterion
- Analysis supported by many commercial and freeware tools (test coverage or code coverage tools)
  - Standard Unix tool: *tcov*
  - A listing indicates how often each statement was executed and the percentage of statements executed
- Note: in case of unreachable statements, full statement coverage is not possible

Example : statement coverage



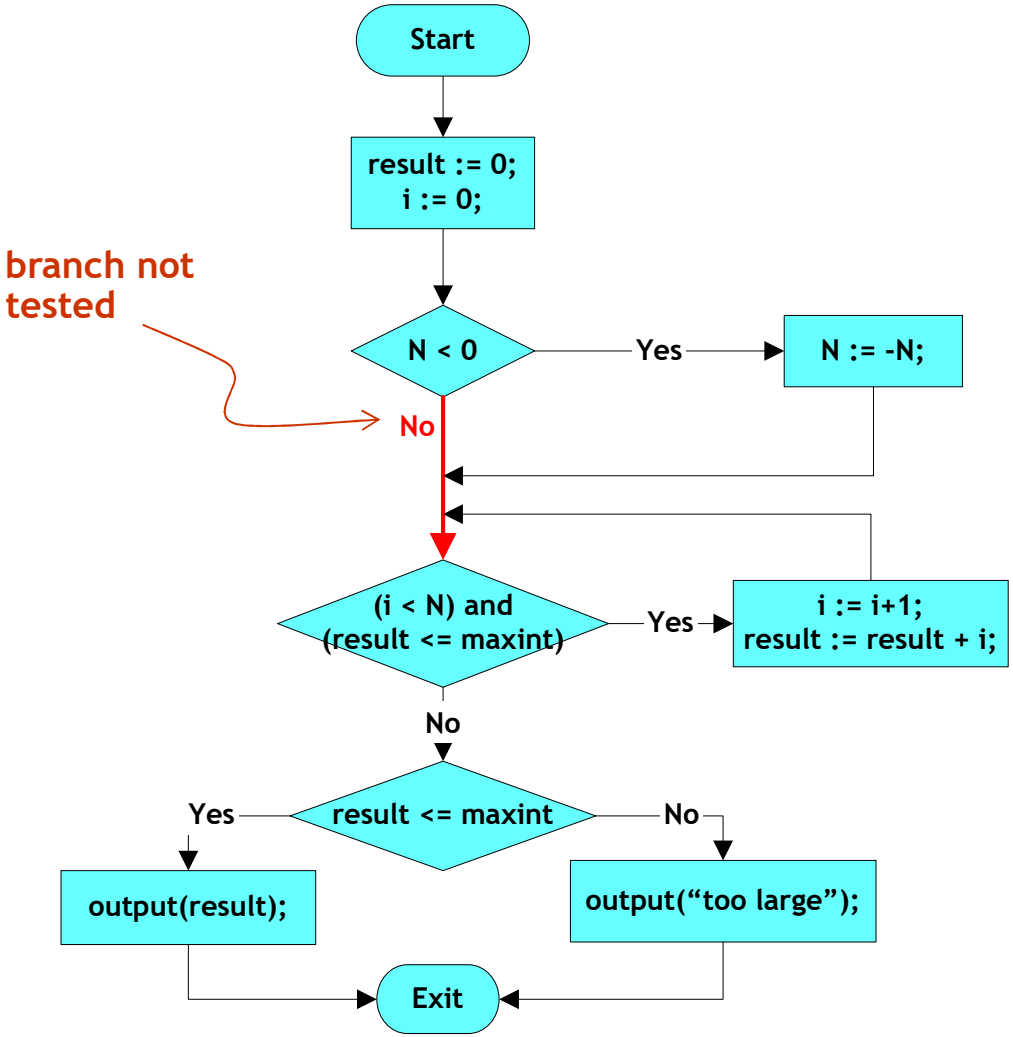
Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

Decision (or branch) coverage

- **Execute every branch of a program :**
  each possible outcome of each decision occurs at least once
- Example:
  - simple decision: IF b THEN s1 ELSE s2
    - b should be tested for true and false
  - multiple decision:
    CASE x OF
    1 : ….
    2 : ….
    3 : ….
- Stronger than statement coverage
  - IF THEN without ELSE – if the condition is always true all the statements are executed, but branch coverage is not achieved (infeasibility)

Example : decision (or branch) coverage



Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

are not sufficient for decision (branch) coverage!

Take:

inputs		outputs
maxint	N	result
10	3	6
0	-1	too large

for complete decision (branch) coverage

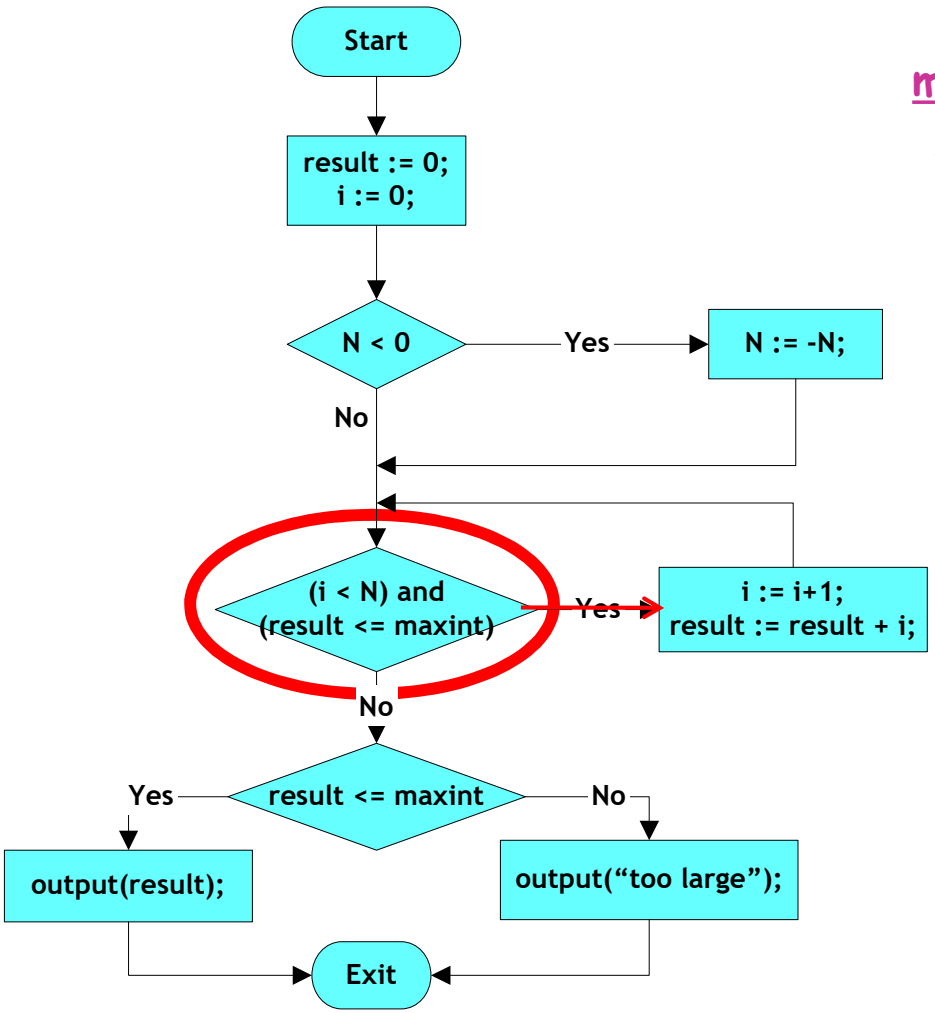
Condition coverage

- Design test cases such that **each possible outcome** of each condition in a decision (composite condition) **occurs at least once**

- Example:
  - **decision** (i < N) AND (result <= maxint)
    consists of **two** conditions : (i < N) , (result <= maxint)
  - test cases should be designed such that each condition gets value **true** and **false** at least once

- Last test cases of previous slides already guarantee condition (and branch) coverage

Condition and decision (or condition / decision) coverage



Test cases:

maxint	N	i	result	i<N	result<=maxint
-1	1	0	0	true	false
1	0	0	0	false	true

give condition coverage
for all conditions

But don't preserve
branch coverage

⇓

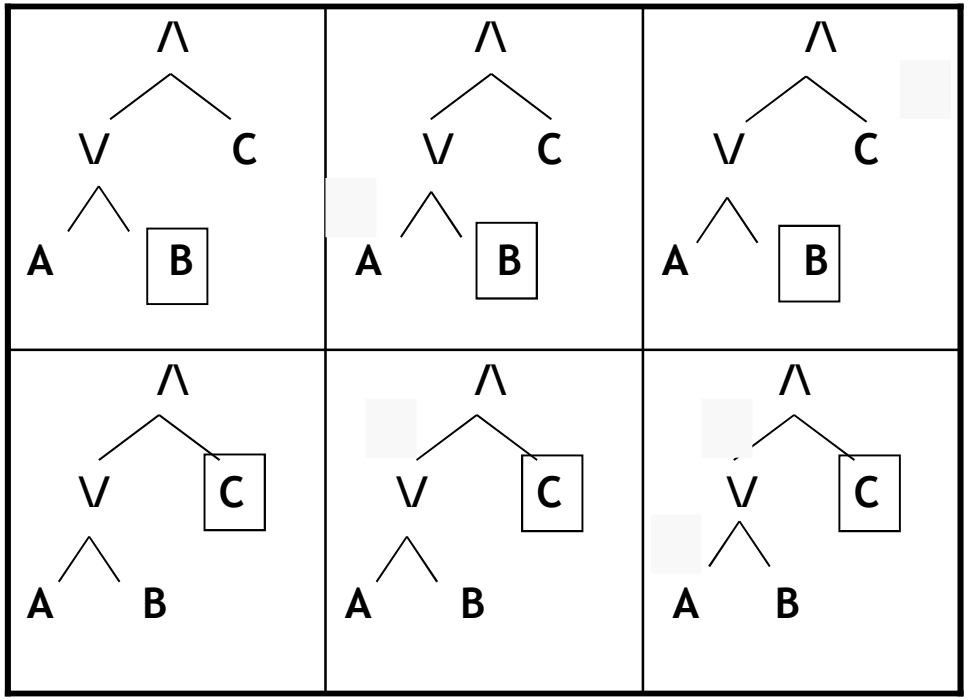
always take care that
condition coverage
preserves branch coverage :
condition and decision coverage

Modified Condition/Decision Coverage

- Also known as MC/DC or MCDC
- Design test cases such that
  - every decision in the program has taken **all possible outcomes at least once** (decision coverage)
  - every condition in a decision in the program has taken **all possible outcomes** at least once (condition coverage)
  - every condition in a decision has been shown to **independently affect that decision's outcome**; a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions
    - **condition** – a Boolean expression containing no Boolean operators
    - **decision** – a Boolean expression composed of conditions and zero or more Boolean operators
- Created at Boeing, required for level A (critical) software for the Federal Aviation Administration (FAA) in the USA by RCTA/DO-178B

Modified Condition/Decision Coverage

(A ∨ B) ∧ C



Modified Condition/Decision Coverage

- Test cases required to meet the MC/DC criteria

Test case	Conditions		Decision
	A	B	A **and** B
1	True	True	True
2	False	True	False
3	True	False	False

Test case	Conditions		Decision
	A	B	A **or** B
1	False	False	False
2	True	False	True
3	False	True	True

Test case	Condition	Decision
	A	**not** A
1	True	False
2	False	True

Test case	Conditions		Decision
	A	B	A **xor** B
1	True	True	False
2	False	True	True
3	True	False	True

(or another combination)

Modified Condition/Decision Coverage

- Consider the following fragment of code:

```
if
    A or (B and C)
then
    do_something;
else
    do_something_else;
end if;
```

- MC/DC may be achieved with the following set of test inputs (note that there are alternative sets of test inputs, which will also achieve MC/DC):

Case	A	B	C	Outcome
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

are not evaluated if logical operators short-circuit

cases 2 a 4 are sufficient for branch and condition coverage, but only if logical operators do not short-circuit

- Because:
  - A is shown to independently affect the outcome of the decision condition by case 1 and case 2
  - B is shown to independently affect the outcome of the decision condition by case 1 and case 3
  - C is shown to independently affect the outcome of the decision condition by case 3 and case 4

Multiple condition coverage

- Design test cases for each combination of conditions
- Example:
  - (i < N) (result <= maxint)

(i < N)	(result <= maxint)
false	false
false	true
true	false
true	true

- Implies decision, condition, decision and condition, modified branch/condition coverage
- But : exponential blow-up ($2^{\text{number of conditions}}$)
- Again : some combinations may be infeasible

Path coverage

- Execute every possible path of a program, i.e., every possible sequence of statements
- Strongest white-box criterion (based on control flow analysis)
- Usually impossible: infinitely many paths (in case of loops)
- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases (each sequence of statements executed for only one value) (doesn't mean exhaustive testing)

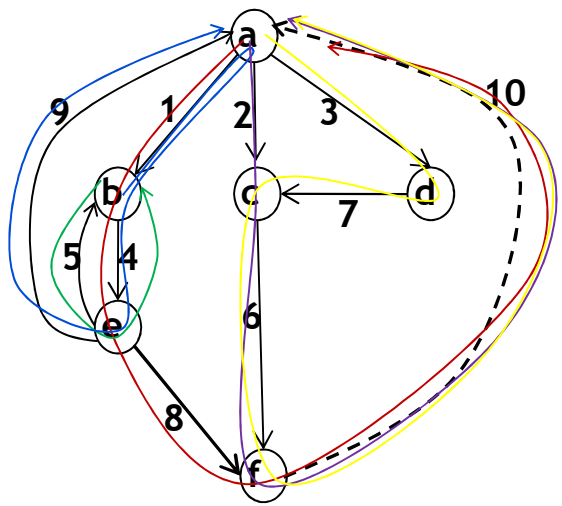
Independent path (or basis path) coverage

- Obtain a maximal set of linearly independent paths (also called a basis of independent paths)
  - If each path is represented as a vector with the number of times that each edge of the control flow graph is traversed, the paths are linearly independent if it is not possible to express one of them as a linear combination of the others
- Generate a test case for each independent path
- The number of linearly independent paths is given by the McCabe's *cyclomatic complexity* of the program
  - Number of edges - Number of nodes + 2 in the control flow graph
  - Measures the structural complexity of the program

Independent path (or basis path) coverage

- Problem: some paths may be impossible to execute
- Also called structured testing (see McCabe for details)
- McCabe's argument: this approach produces a number of test cases that is proportional to the complexity of the program (as measured by the cyclomatic complexity), which, in turn, is related to the number of defects expected
- More information:
  - http://www.mccabe.com/iq_research_metrics.htm
  - "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235

Example: Independent path coverage

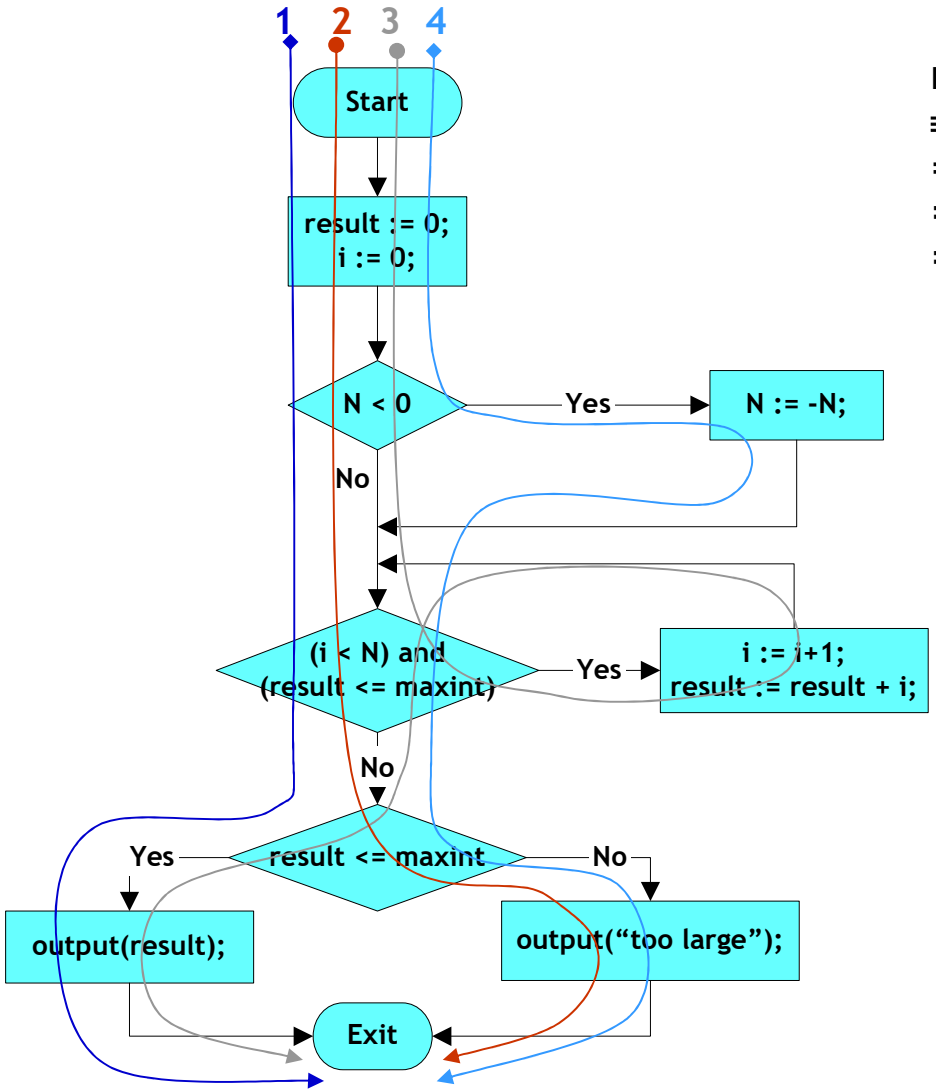


	1	2	3	4	5	6	7	8	9	10
abefa	1	0	0	1	0	0	0	1	0	1
beb	0	0	0	1	1	0	0	0	0	0
abea	1	0	0	1	0	0	0	0	1	0
acfa	0	1	0	0	0	1	0	0	0	1
adcfa	0	0	1	0	0	1	1	0	0	1

Theorem: In a strongly connected graph, G, the cyclomatic number is equal to the maximum number of linear independent circuits.

- 5 independent circuits
  - (abefa),(beb),(abea),(acfa),(adcfa)
- 5 independent paths
  - (abef), (abebef),(abeabef),(acf),(adcf)
  - This set of independent paths forms the basis for the set of all circuits in the graph.
- For instance
  - The path (abea(be)^{3}f) = 2(abebef)-(abef)

Example: Independent path coverage

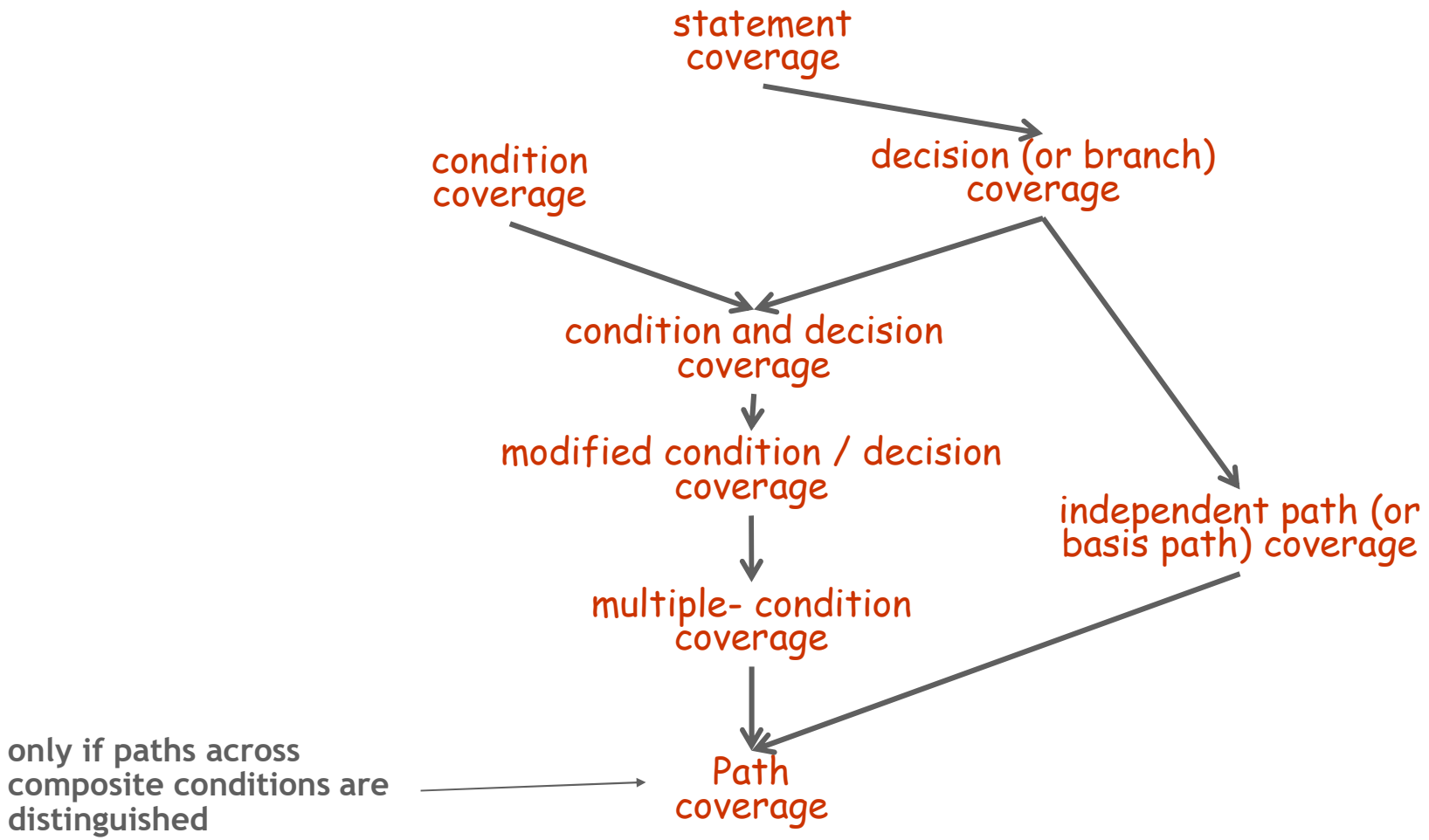


number of independent paths
≡ cyclomatic complexity
= number of edges - number of nodes + 2
= 12 - 10 + 2
= 4

Test cases

Path	inputs		outputs
	maxint	N	result
1	1	0	0
2	-1	0	too large
3	-1	-1	too large
4	10	1	1

White-Box Testing : Overview



Data flow testing

- We say a variable is **defined** in a statement when its value is assigned or changed

  Y = 26 * X

  - This is indicated as a *def* for the variable Y

- We say a variable is **used** in a statement when its value is utilized in a statement. The value of the variable is not changed. Data flow roles:
  - *c-use* for variable X:

    Y = 26 * X
  - *p-use* for variable X:

    if (X > 98)

    Y = max
  - others: *undefined* or *dead*

Data flow testing

- Coverage criteria
  - All def
  - All p-uses
  - All c-uses/some p-uses
  - All p-uses/some c-uses
  - All uses
  - All def-use paths
  - … and several variants of this technique …

- The strongest of these criteria is *all def-use paths*. This includes all *p-* and *c-uses*.

Data flow graph

- A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths thorough the control flow graph are preserved in the data flow graph. An example follows.

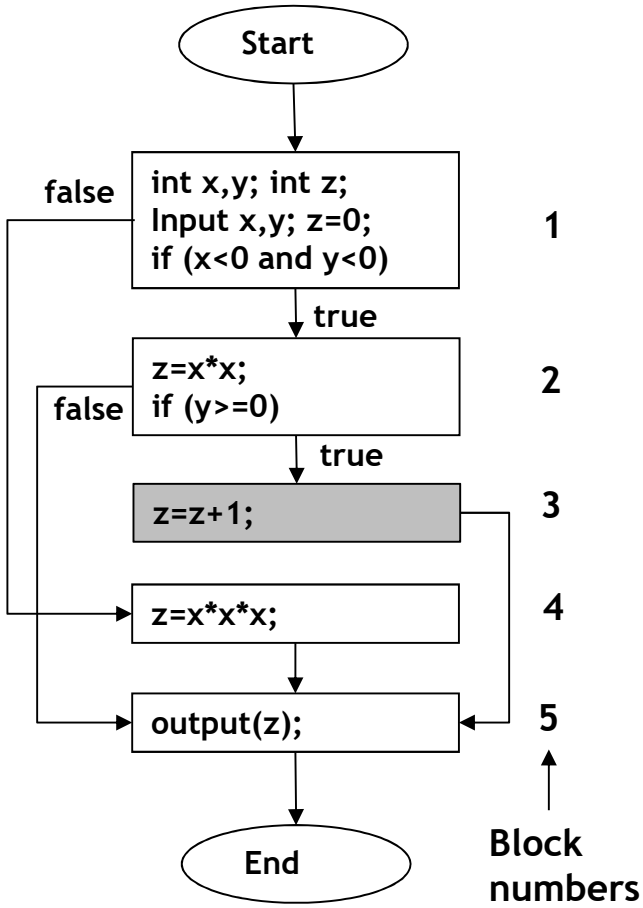
© Aditya P. Mathur 2007

Data flow graph: Example

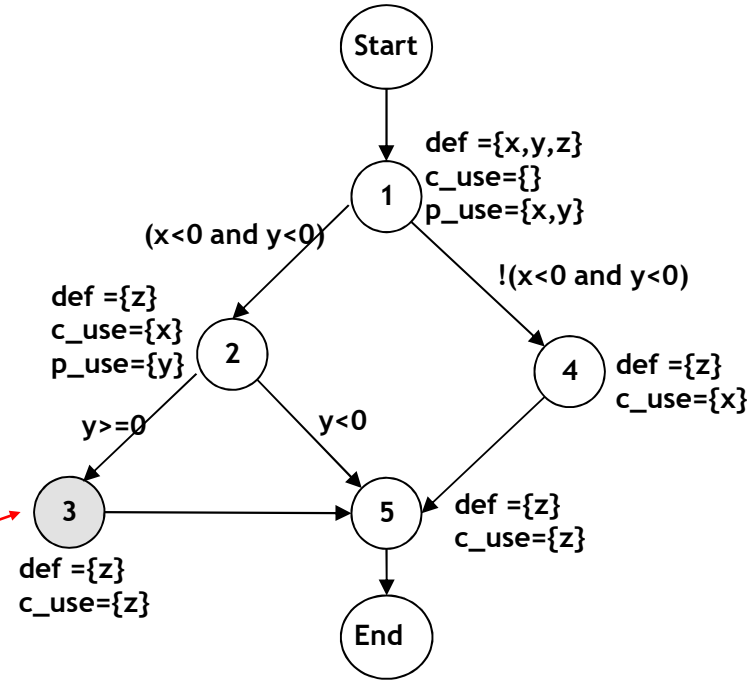
- Given a program, find its basic blocks, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).
- Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken.
- We use $d_i(x)$ to refer to the definition of variable x at node i. Similarly, $u_i(x)$ refers to the use of variable x at node i.

© Aditya P. Mathur 2007

Data flow graph: Example (contd.)



Node (or Block)	def	c-use	p-use
1	{x,y,z}	{ }	{x,y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



Unreachable node

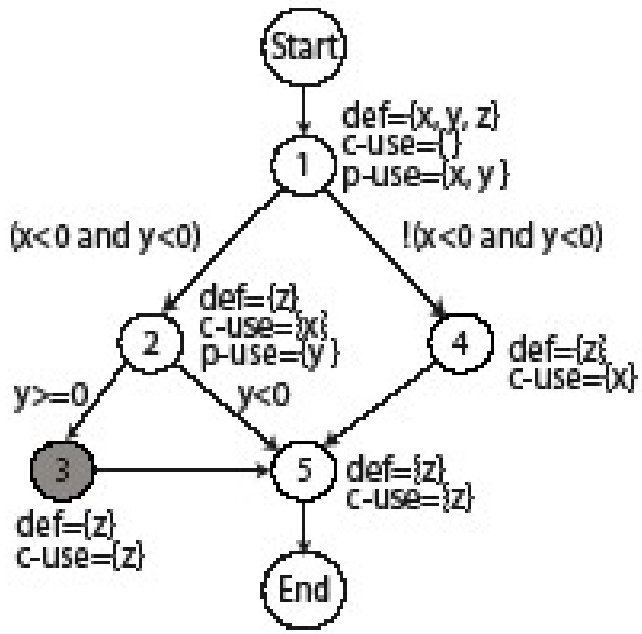
© Aditya P. Mathur 2007

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a *def-clear* path for x.

Path 2-5 is def-clear for variable z defined at node 2 and used at node 5. Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.

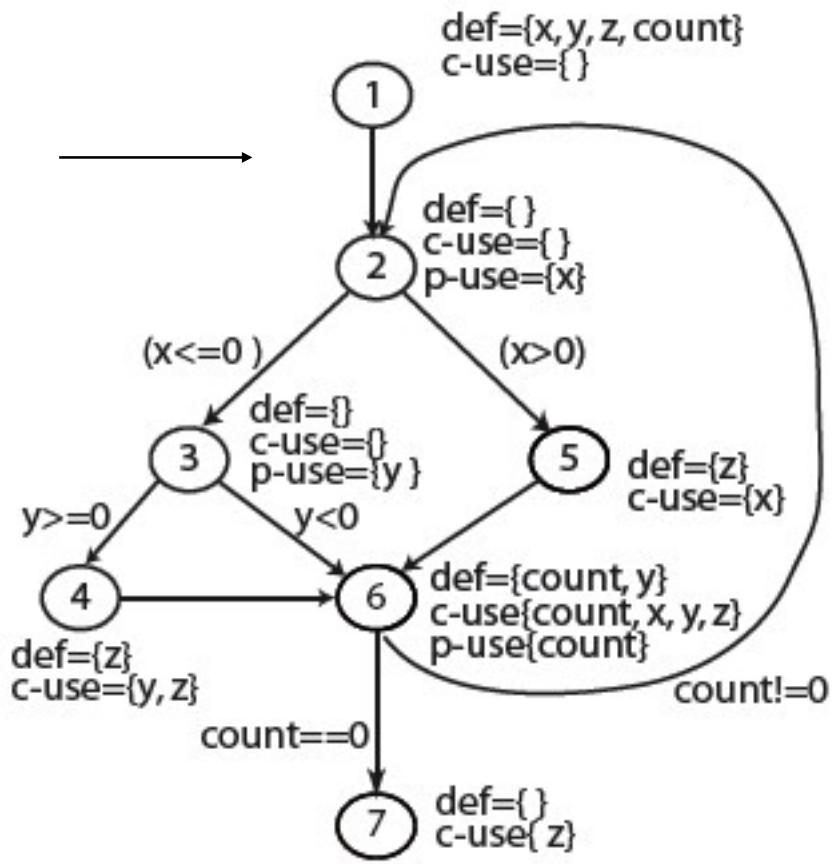


(a)

© Aditya P. Mathur 2007

Def-clear path (another example)

```
1   begin
2     float x, y, z=0.0;
3     int count;
4     input (x, y, count);
5     do {
6       if (x≤0) {
7         if (y≥0) {
8           z=y*z+1;
9         }
10      }
11      else{
12        z=1/x;
13      }
14      y=x*y+z
15      count=count-1
16    while (count>0)
17    output (z);
18  end
```



Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Find def-clear paths for defs and uses of x and z.
Which definitions are live at node 4?

© Aditya P. Mathur 2007

Mutation testing (fault injection)

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is modified in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are run with each mutant
- Live mutants cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as killing such mutant
- If a mutant remains live (passes all the test cases), then either the mutant is equivalent to the parent (and is ignored), or it is not equivalent, in which case **additional test cases** should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Class testing / coverage

- In object oriented systems, the units to be tested are typically classes, but can also be methods or clusters of related classes
- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - *Exercising the object in all possible states*
- Each test case typically exercises a possible object lifecycle, with alternating operation calls to change and query the object's state
- Encapsulation, inheritance and polymorphism complicate the design of test cases
  - Encapsulation – how to check the object's state?
  - Inheritance – how to (unit) test abstract classes, abstract methods and interfaces?
  - Polymorphism – how to test methods with *callbacks* (from super-class to sub-class)? (same problem with event handlers)

White-Box testing : How to Apply ?

- Don't start with designing white-box test cases!
- Start with black-box test cases
  (equivalence partitioning, boundary value analysis,
  cause effect graphing, derivation with formal methods,…)
- Check white-box coverage
  (statement, branch, condition,… , coverage)
- Use a testing coverage tool
- Design additional white-box test cases for not covered code

References and further reading

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- The Art of Software Testing, Glenford J. Myers, Wiley & Sons, 1979 (Chapter 4 - Test Case Design)
- Software testing techniques, Boris Beizer, Van Nostrand Reinhold, 2nd Ed, 1990
- Foundations of Software Testing, by Aditya P. Mathur, Pearson Education, 2008
- Testing Computer Software, 2nd Edition, Cem Kaner, Jack Falk, Hung Nguyen, John Wiley & Sons, 1999
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- http://www.swebok.org/ - Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE Computer Society
- http://www.mccabe.com/iq_research_metrics.htm
- "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235
- RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", Radio Technical Commission for Aeronautics, USA, December 1992
- Chilenski1994 John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.
- A Practical Tutorial on Modified Condition / Decision Coverage, NASA / TM-2001-210876 (http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf)
- A Complexity Measure, Thomas J. McCabe, IEEE Trans. Software Eng., volume 2, number 4, 1976, pages 308-320.

Exercise 1

If the pseudocode below were a programming language ,how many tests are required to achieve 100% statement coverage?

```
If x=3 then
        Display_messageX;
        If y=2 then
                Display_messageY;
        Else
                Display_messageZ;
Else
        Display_messageZ;
```

- 1. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Using the same code example as question 17,how many tests are required to achieve 100% branch/decision coverage?

- 2. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Exercise 2

- **Given the following code, which is true about the minimum number of test cases required for full statement and branch coverage:**

```
Read P
Read Q
IF P+Q > 100 THEN
   Print "Large"
ENDIF
If P > 50 THEN
   Print "P Large"
ENDIF
```

a) 1 test for statement coverage, 3 for branch coverage

b) 1 test for statement coverage, 2 for branch coverage

c) 1 test for statement coverage, 1 for branch coverage

d) 2 tests for statement coverage, 3 for branch coverage

e) 2 tests for statement coverage, 2 for branch coverage

Exercise 3 (1)

- A Program is written to meet the following requirements:
  - R1: Given coordinate positions x, y and z, and a direction valued d, the program must invoke one of the three functions fire-1, fire-2, fire-3 as per conditions bellow:
    - R1.1: Invoke fire-1 when (x<y) AND (z * z > y) AND (prev="East").
    - R1.2: Invoke fire-2 when (x<y) AND (z * z ≤ y) OR (current="South").
    - R1.3: Invoke fire-3 when none of the two conditions above is true.
  - R2: The invocation described above must continue until an input Boolean variable becomes true.

© Aditya P. Mathur 2007

Exercise 3 (2)

```
1  begin
2    float x, y, z;
3    direction d;
4    string prev, current;
5    bool done;

6    input(done);
7    current="North";
8    while (~done)  {                                          ← condition C1
8            input (d);
10           prev=current; current=f(d);
11           input(x,y,z);
12           if ((x<y) and (z*z > y) and (prev=="East"))       ← Condition C2
13                   fire-1(x,y);
14           else if ((x<y) and (z*z<=y) or (current=="South"))  ← Condition C3
15                   fire-2(x,y);
16           else
17                   fire-3(x,y);
17           input(done);
18 }
19 output("Firing completed.");
20 end
```

Exercise 3 (3)

- Verify that the following set T1 of four tests, executed in the given order, is adequate with respect to statement, block, and decision coverage criteria but not with respect to the condition coverage criterion.

Test set T1						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t4	R2	True	-	-	-	-

© Aditya P. Mathur 2007

Exercise 3 (4)

Test set T2						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t4	R2	True	-	-	-	-

Test set T2 is adequate according to MC/DC?

© Aditya P. Mathur 2007

Exercise 3 (5)

- Verify that the following set T3, obtained by adding t6, t7, t8, and t9 to T2 is adequate with respect to MC/DC coverage criterion. Note again that sequencing of tests is important in this case (especially for t1 and t7)!

Test set T3						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t6	R1	False	East	10	5	2
t7	R1	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t8	R1	False	South	10	5	2
t9	R1	False	North	10	5	2
t4	R2	True	-	-	-	-

© Aditya P. Mathur 2007

Exercise 4

- Suppose that condition C=C1 AND C2 AND C3 has been coded as C'=C1 AND C3. Four tests that form an MC/DC adequate set for C' are in the following table. Verify that the following set of four tests is MC/DC adequate but does not reveal the error.

	Test	C	C'	Error detected?
	C1, C2, C3	C1 and C2 and C3	C1 and C3	
t1 t2	true, true, true false, false, false			
t3 t4	true, true, false false, false, true			

© Aditya P. Mathur 2007

TVVS - Test, Verification and Validation of Software

static testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Static testing

- Static testing techniques rely on the manual examination (**reviews**) and automated analysis (**static analysis**) of the code or other project documentation.
- Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution.
- Defects detected during reviews early in the life cycle are often much cheaper to remove than those detected while running tests (e.g., defects found in requirements).

Reviews

- Benefits of reviews include
  - early defect detection and correction
  - development productivity improvements
  - reduced development timescales
  - reduced testing cost and time
  - lifetime cost reductions
  - fewer defects and improved communication
  - reviews can find omissions, for example, in requirements, which are unlikely to be found in dynamic testing
- Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

The values of static analysis

- Early detection of defects prior to test execution.
- Early warning about suspicious aspects of the code or design, by the calculation of metrics, such as a high complexity measure.
- Identification of defects not easily found by dynamic testing.
- Detecting dependencies and inconsistencies in software models, such as links.
- Improved maintainability of code and design.
- Prevention of defects, if lessons are learned in development.

Reviews are cost effective

- 10 times reduction in fault reaching test, testing cost reduced by 50% to 80%
  - [Freedman & Weinberg, Handbook of Walkthroughs, Inspections & Technical Reviews]
- Reduce faults by a factor of 10
  - [Yourdon, Structured Walkthroughs]
- 25% reduction in schedules, remove 80% - 95% of faults each stage, 28 times reduction in maintenace cost, many others
  - [Gilb & Graham, Software Inspection]

Costs of reviews

- Rough guide: 5%-15% of development effort
  - Half a day a week is 10%
- Effort required for reviews
  - Planning (by leader / moderator)
  - Preparation / self study checking
  - Meeting
  - Fixing / editing / follow-up
  - Recording & analysis of statistics / metrics
  - Process improvement (should!)

Reviews

- Typical defects that are easier to find in reviews than in dynamic testing are:
  - deviations from standards,
  - requirement defects,
  - design defects,
  - insufficient maintainability and
  - incorrect interface specifications.

What can be Reviewed/Inspected?

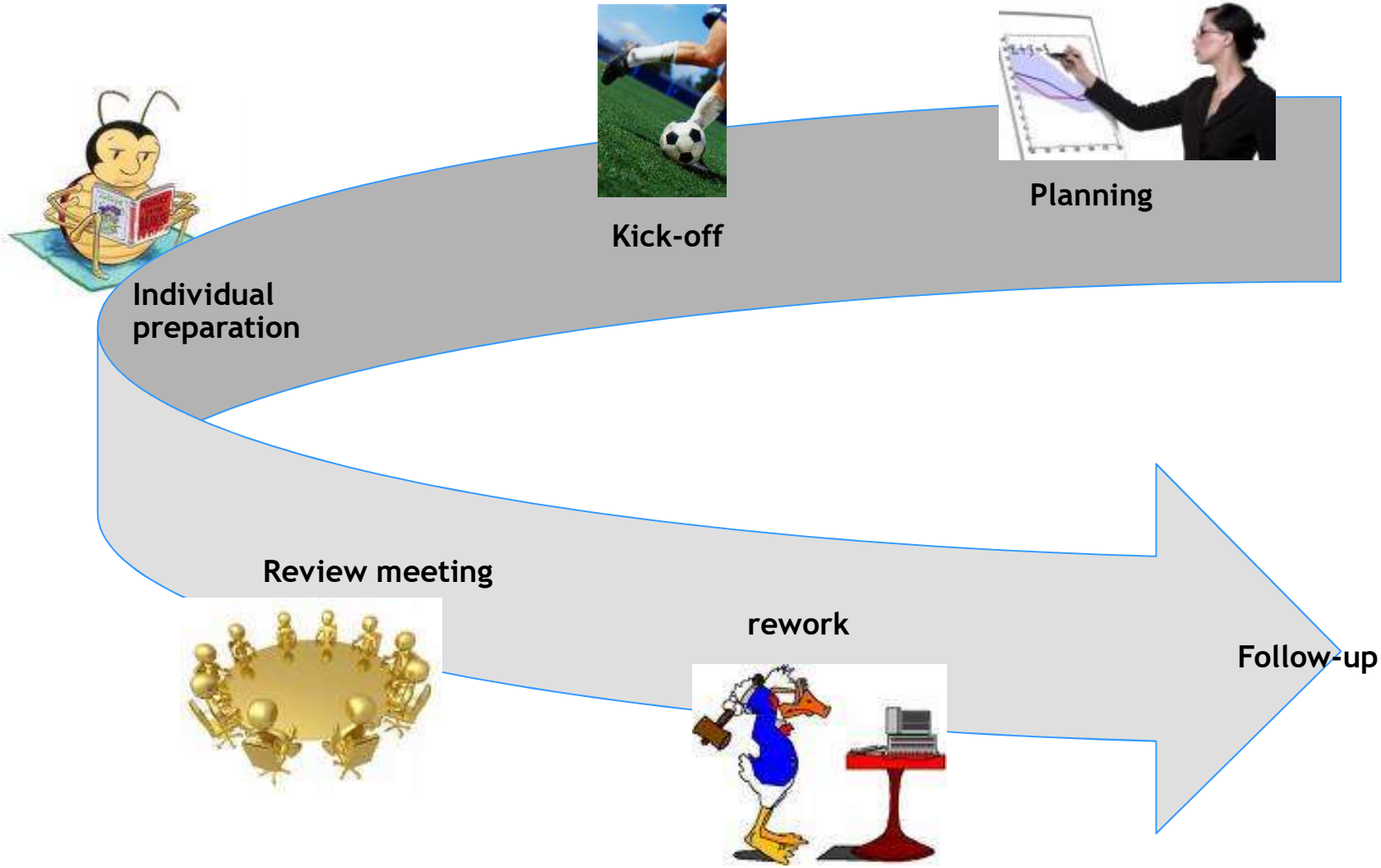
Anything written down can be inspected

- Policy, strategy, business plans, marketing or advertising materials, contracts
- System requirements, feasibility studies, acceptance test plans
- Test plans, test designs, test cases, test results
- System designs, logical & physical
- Software code
- User manuals, procedures, training material

Reviews

- The different types of reviews vary from very informal (e.g., no written instructions for reviewers) to very formal (i.e., well structured and regulated). The formality of a review process is related to factors such as
  - the maturity of the development process,
  - any legal or regulatory requirements,
  - or the need for an audit trail.
- The way a review is carried out depends on the agreed objective of the review (e.g.,
  - find defects,
  - gain understanding,
  - or discussion and decision by consensus).

Phases of a formal review



Phases of a formal review

- **Planning**: selecting the personnel, allocating roles; defining the entry and exit criteria for more formal review types (e.g., inspection); and selecting which parts of documents to look at.
- **Kick-off**: distributing documents; explaining the objectives, process and documents to the participants; and checking entry criteria (for more formal review types).
- **Individual preparation**: work done by each of the participants on their own before the review meeting, noting potential defects, questions and comments.

Phases of a formal review

- **Review meeting**: discussion or logging, with documented results or minutes (for more formal review types). The meeting participants may simply note defects, make recommendations for handling the defects, or make decisions about the defects.
- **Rework**: fixing defects found, typically done by the author.
- **Follow-up**: checking that defects have been addressed, gathering metrics and checking on exit criteria (for more formal review types).

Reviews: Deliverales

- Changes (edits) in review product
- Change requests for source documents (predecessor documents to product being reviewed / Inspected)
- Process improvement suggestions
  - To the review / inspection process
  - To the development process which produced the product just reviewed / inspected
- Metrics

Reviews

Pitfalls (they don't always work!)

- Lack of training in the technique (especially inspection, the most formal)
- Lack of or quality of documentation – what is being reviewed / inspected
- Lack of management support – "lip service" want them done, but don't allow time for them to happen in project schedules
- Failure to improve processes (gets disheartening just getting better at finding the same thing over again)

Roles and responsabilities

- **Manager**: decides on the execution of reviews, allocates time in project schedules and determines if the review objectives have been met.
- **Moderator**: the person who leads the review of the document or set of documents, including planning the review, running the meeting, and follow-up after the meeting. If necessary, the moderator may mediate between the various points of view and is often the person upon whom the success of the review rests.
- **Author**: the writer or person with chief responsibility for the document(s) to be reviewed.

Roles and responsabilities

- **Reviewers**: individuals with a specific technical or business background (also called checkers or inspectors) who, after the necessary preparation, identify and describe findings (e.g., defects) in the product under review. Reviewers should be chosen to represent different perspectives and roles in the review process, and should take part in any review meetings.
- **Scribe** (or recorder): documents all the issues, problems and open points that were identified during the meeting.

Types of reviews

- A single document may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with customers. The main characteristics, options and purposes of common review types are:
  - Informal reviews
  - Walkthrough
  - Technical review
  - Inspection

Informal reviews

Widely viewed as useful and cheap (but no one can prove it!). A helpful first step for chaotic organizations.

- No formal process;
- there may be pair programming or a technical lead reviewing designs and code;
- optionally may be documented;
- may vary in usefulness depending on the reviewer;
- **main purpose**: inexpensive way to get some benefit.

Walkthrough

Author guides the group through a document and his or her thought processes, so all understand the same thing, consensus on change to make

- meeting led by author;
- scenarios, dry runs, peer group;
- open-ended sessions;
- optionally a pre-meeting preparation of reviewers, review report, list of findings and scribe (who is not the author);
- may vary in practice from quite informal to very formal;
- **main purposes**: learning, gaining understanding, defect finding.

Technical review

Includes peer and technical experts, no management participation. Normally documented, fault-finding. Can be rather subjective

- documented, defined defect-detection process that includes peers and technical experts;
- may be performed as a peer review without management participation;
- ideally led by trained moderator (not the author);
- pre-meeting preparation;
- optionally the use of checklists, review report, list of findings and management participation;
- may vary in practice from quite informal to very formal;
- **main purposes:** discuss, make decisions, evaluate alternatives, find defects, solve technical problems and check conformance to specifications and standards.

Inspection

Formal individual and group checking, using sources and standards, according to generic and specific rules and checklists, using entry and exit criteria, leader must be trained & certified, metrics required.

- led by trained moderator (not the author);
- usually peer examination;
- defined roles;
- includes metrics;
- formal process based on rules and checklists with entry and exit criteria;
- pre-meeting preparation;
- inspection report, list of findings;
- formal follow-up process;
- optionally, process improvement and reader;
- **main purpose**: find defects.

Reviews

- Walkthroughs, technical reviews and inspections can be performed within a peer group – colleagues at the same organizational level. This type of review is called a "**peer review**".

Success factors for reviews

- Each review has a clear predefined objective.
- The right people for the review objectives are involved.
- Defects found are welcomed, and expressed objectively.
- People issues and psychological aspects are dealt with (e.g., making it a positive experience for the author).
- Review techniques are applied that are suitable to the type and level of software work products and reviewers.

Success factors for reviews

- Checklists or roles are used if appropriate to increase effectiveness of defect identification.
- Training is given in review techniques, especially the more formal techniques, such as inspection.
- Management supports a good review process (e.g., by incorporating adequate time for review activities in project schedules).
- There is an emphasis on learning and process improvement.

Static analysis by tools

- The objective of static analysis is to find defects in software source code and software models. Static analysis is performed without actually executing the software being examined by the tool; dynamic testing does execute the software code. Static analysis can locate defects that are hard to find in testing. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as, generated output such as HTML and XML.

Static analysis by tools

Descendent from compiler technology

- A compiler statically analyses code and “knows” a lot about it, e.g., variable usage, finds syntax faults
- Static analysis tools extend this knowledge
- Can find unreachale code, undeclared variables, parameter type mis-matches, uncalled functions & procedures, array bound violations, etc.

Static analysis

Typical defects discovered by static analysis tools include:

- referencing a variable with an undefined value;
- inconsistent interface between modules and components;
- variables that are never used;
- unreachable (dead) code;
- programming standards violations;
- security vulnerabilities;
- syntax violations of code and software models.

Static analysis

- Static analysis tools are typically used
  - **by developers** (checking against predefined rules or programming standards) before and during component and integration testing, and
  - **by designers** during software modeling.
- Static analysis tools may produce a large number of warning messages, which need to be well managed to allow the most effective use of the tool.
- Compilers may offer some support for static analysis, including the calculation of **metrics**.

Data flow analysis

This is study of program variables

- Variable defined* when a value is stored into it
- Variable used when the stored value is accessed
- Variable is undefined before it is defined or when it goes out of scope

x is defined, y and z are used

```
x = y + z
if a>b then read(s)
```

a and b are used, s is defined

*defined should not be confused with declared

Data flow analysis

```
n:=0

read(x)

n:=1

while x>y do

    begin

        read(y)

        write(n*y)

        x:=x-n

end
```

Data flow anomaly: n is re-defined without being used

Data flow fault: y is used before it has been defined (first time around the loop)

Control flow analysis

- Highlights
- Nodes not accessible from start node
- Infinite loops
- Multiple entry to loops
- Whether code is well structured, i.e., reducible
- Whether code conforms to a flowchart grammar
- Any jumps to undefined labels
- Any labels not jumped to
- Cyclomatic complexity and other metrics

Unreacheable code example

- Macro definitions (different for different platforms the code runs on)

```
Buffsize: 1000

Mailboxmax: 1000

        if Buffsize < Mailboxmax then

                Error-Exit

        endif
```

- Static analysis finds the then clause unreachable, so will flag a fault

Cyclomatic complexity

- Cyclomatic complexity is a measure of the complexity of a flow graph
  - (and therefore the code that the flow graph represents)
- The more complex the flow graph, the greater the measure
- It can most easily be calculated as
  - Complexity = number of dicisions + 1 or
  - Complexity = Number of edges – number of node + 2

Other static metrics

- Lines of code (LOC)
- Operands & operators (Halstead's metrics): count operators such as "+", "-", "/", and "x", but this measure is not much used now.
- Structural fan-in & fan-out: modules with high fan-in are found at the bottom of hierarchies or in libraries where they are frequently called. Modules with high fan-out are typically at the top of hierarchies, because they call to many modules (e.g., the main menu)
- Nesting levels: relate to how deeply nested statements are within other IF statements.
- Function calls
- OO metrics: inheritence tree depth, number of methods, coupling & cohesion

References

- http://www.astqb.org/educational-resources/syllabi-static3.php
- http://www.slideshare.net/onsoftwaretest/iseb-istqb-static-testing-presentation
- http://knol.google.com/k/istqb-certification-syllabi-chapter-3#
- http://www.aivosto.com/project/help/pm-sf.html

Model Based Testing (MBT)

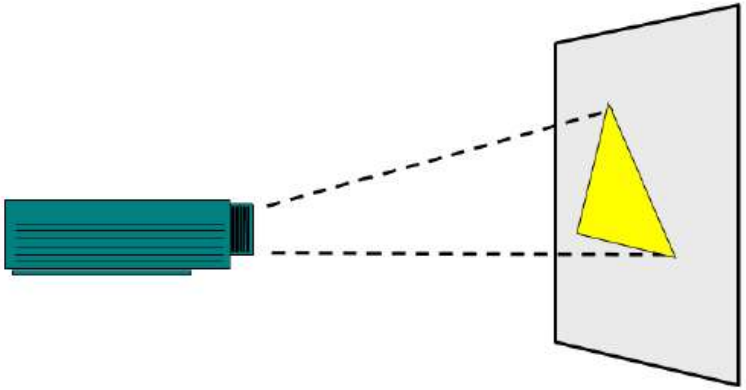
Ana Paiva

apaiva@fe.up.pt

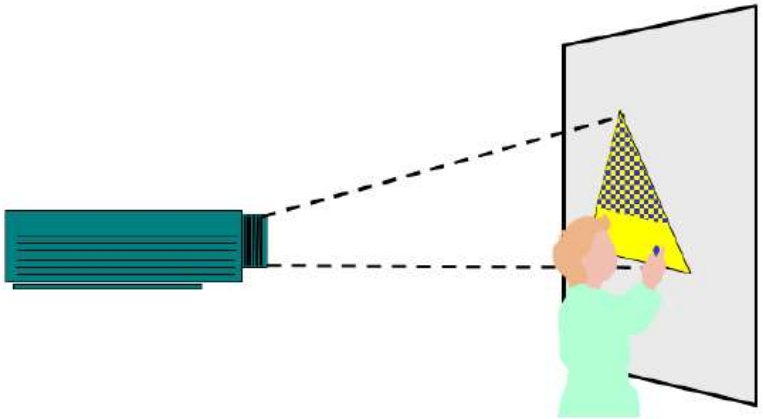
Universidade do Porto
FEUP Faculdade de Engenharia

INESCTEC
TECNOLOGIA E CIÊNCIA | LABORATÓRIO ASSOCIADO | COORDENADO POR INESCPORTO

"Traditional" Automated Testing

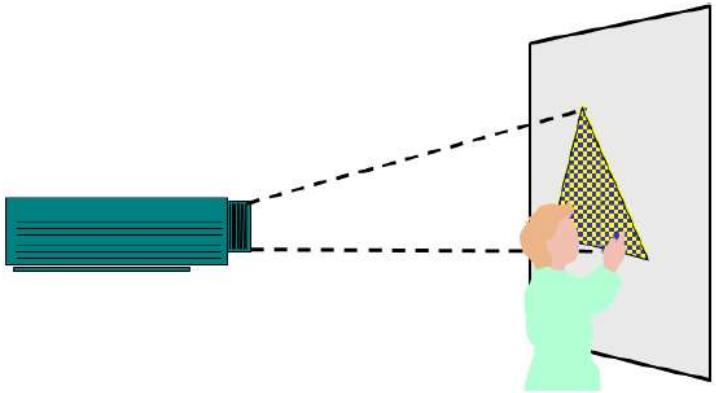


Imagine that this projector is the software you are testing.

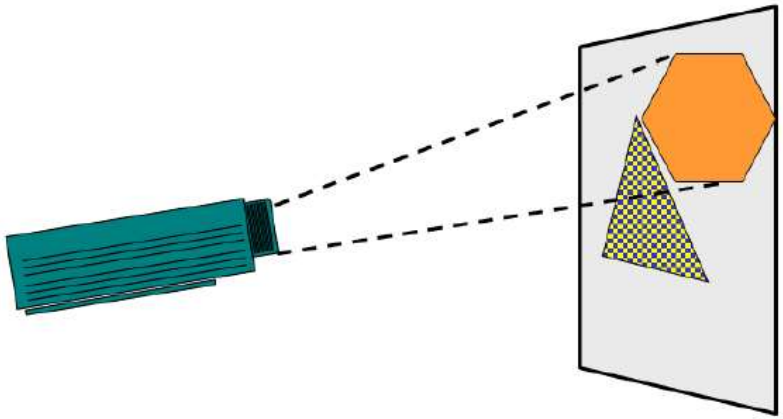


Typically, testers automate by creating static scripts.

"Traditional" Automated Testing

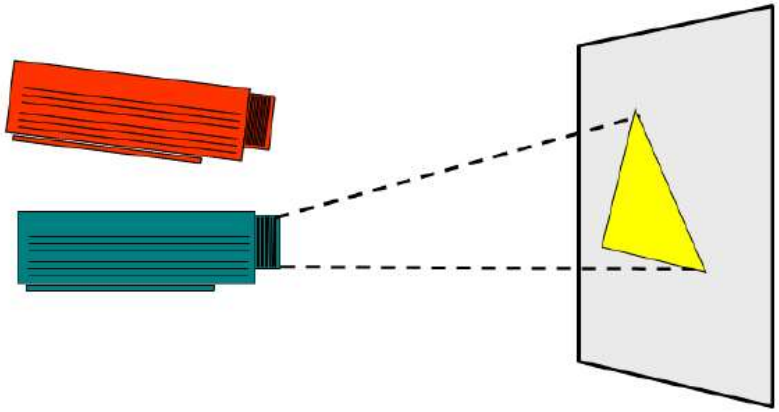


Given enough time, these scripts will cover the behavior.

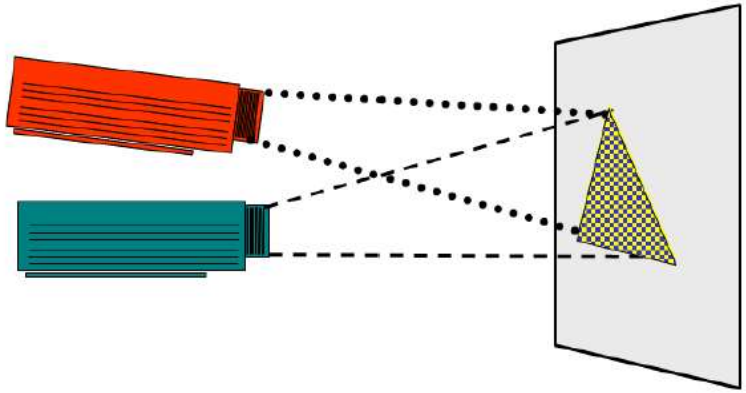


But what happens when the software's behavior changes?

Model Based Testing

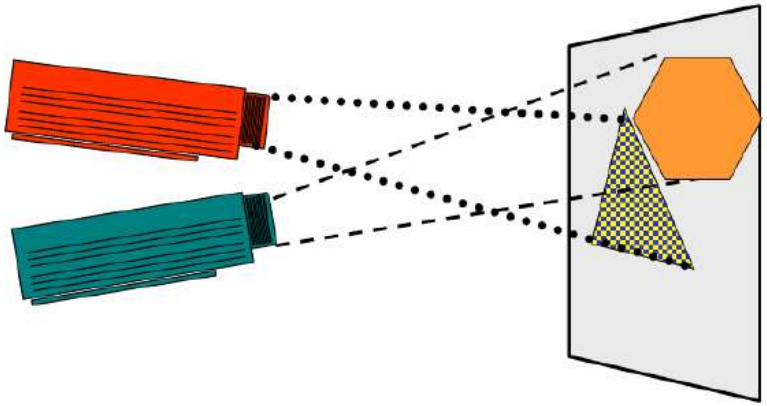


Now, imagine that the top projector is your model.

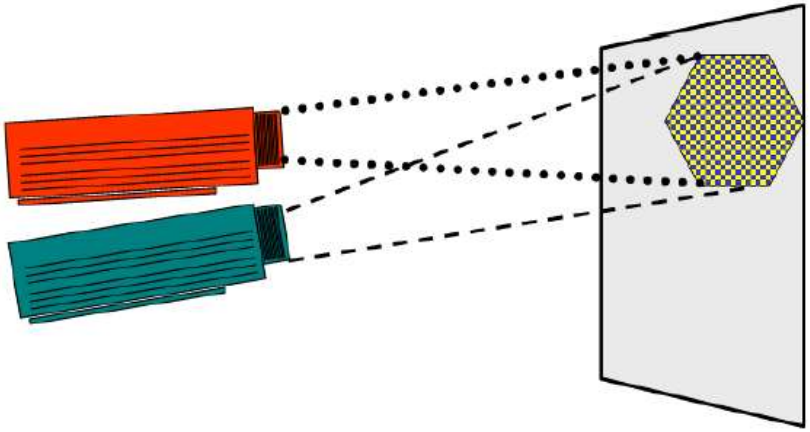


The model generates tests to cover the behavior.

Model Based Testing



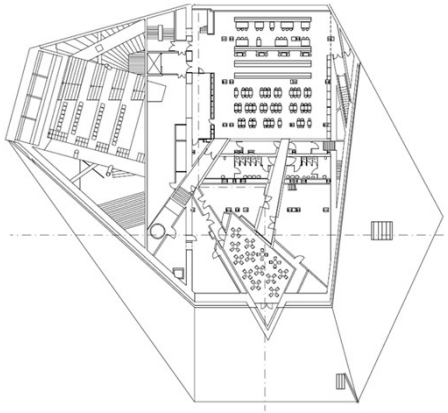
… and when the behavior changes…



… so do the tests.

What is a model?

- A Model is an abstract representation of the system.
- Models are simpler than the systems they describe.
- Models help us understand and predict the system's behavior.



Model

Real

Definition

- Model-based testing is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the System Under Test (SUT).

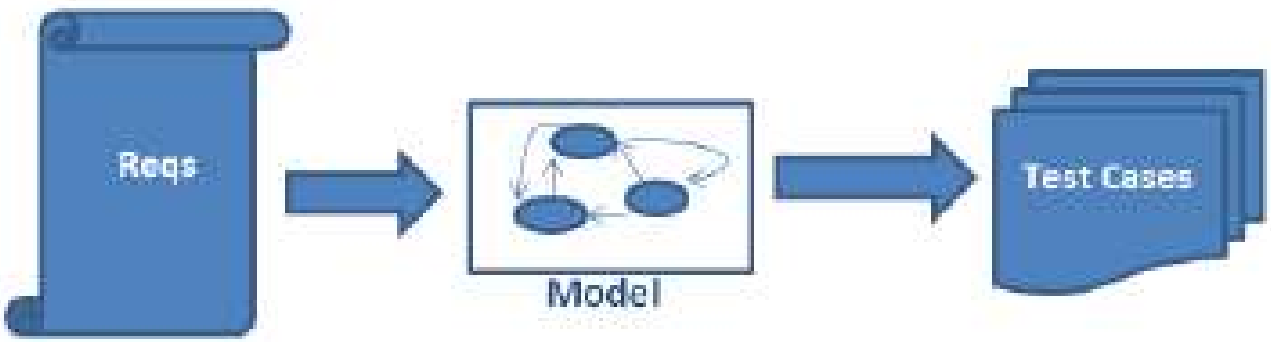
- Model-based testing uses a model program to generate test cases or act as an oracle.

Definition

- An oracle is the authority which provides the correct result used to make a judgment about the outcome of a test case - whether the test passed or failed

Definition

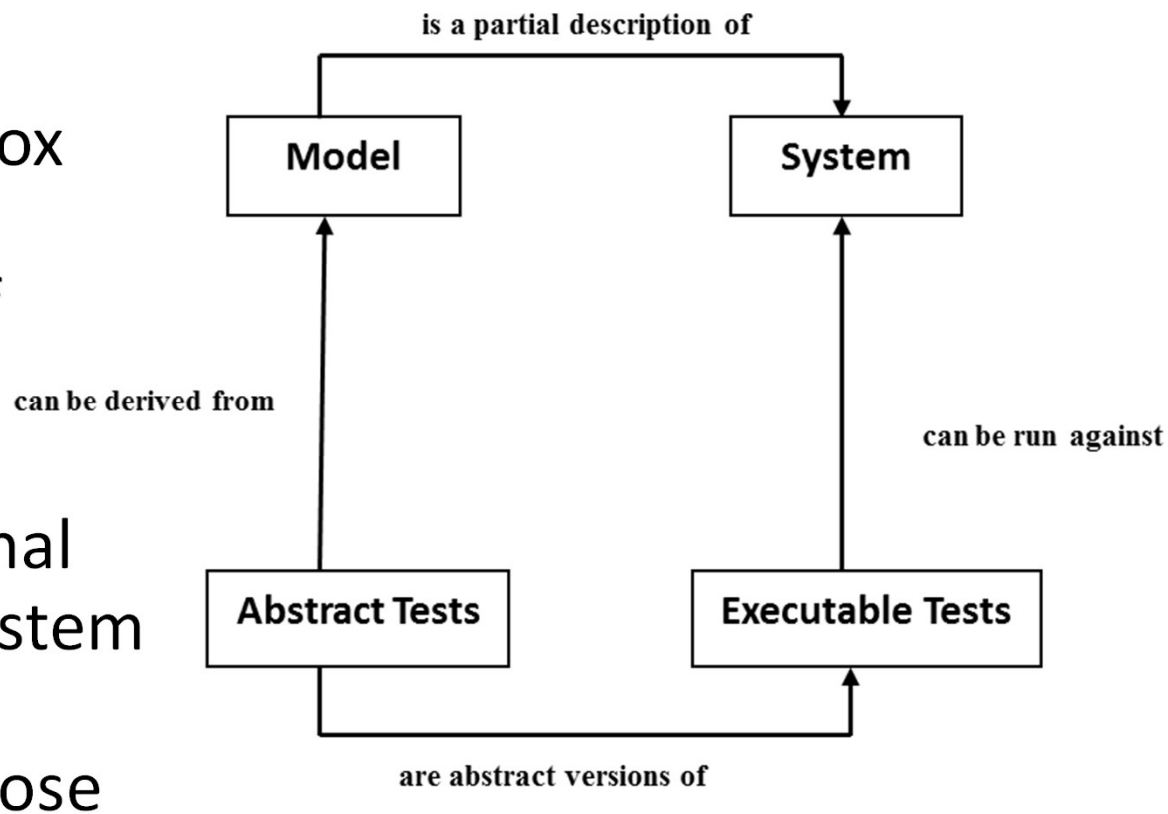
- Model Based Testing is a testing technique where the **runtime behavior** of an implementation under test is checked againts predictions made by formal specification, or **model**." – Colin Campbell, Microsoft Research



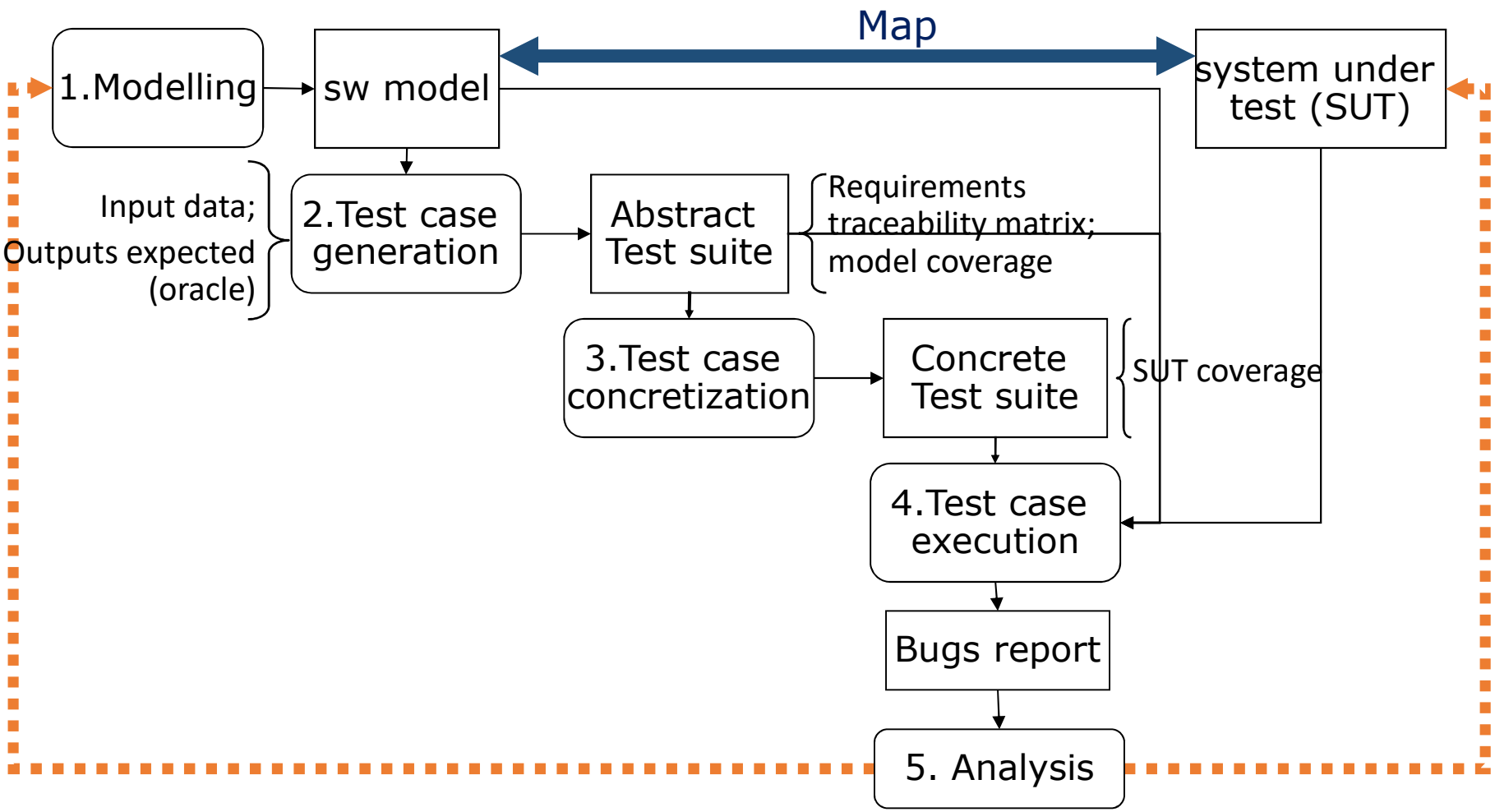
- One way to generate test cases automatically is "model based testing" where a model of the system is used for test case generation.

MBT Process

- "Model-based testing is black box testing process for derivation of test cases from a model that describe functional aspects of the system under test and executing those test cases."



MBT Process


Map
1.Modelling
sw model
system under test (SUT)
Input data;
Outputs expected (oracle)
2.Test case generation
Abstract Test suite
Requirements traceability matrix;
model coverage
3.Test case concretization
Concrete Test suite
SUT coverage
4.Test case execution
Bugs report
5. Analysis

General Test Strategies

- Manual vs automatic
- Static vs dynamic
- White vs Black-box
- Error/Fault based vs specification-based
- API vs UI testing
- Functional vs non-functional (Usability, Performance, Robustness)

Advantages and Disadvantages

- Advantages
  - Improved quality of the product
  - Better quality of the product
  - Higher degree of automation (test case generation)
  - Allows more exhaustive testing
  - Good for correctness/functional testing
  - Model can be easily adapted to changes
  - Early exposure of ambiguities in specification and design
  - Generate variety of test suites from the same model by different test selection criteria

Advantages and Disadvantages

- Disadvantages
  - Requires a formal specification/model
  - Test case explosion problem
  - Test case generation has to be controlled appropriately to generate a test case of manageable size
  - Small changes to the model can result in a totally different test suite
  - Time to analyse failed tests (model, SUT, adaptor code)

Some tools

- ModelJUnit
- SpecTest
- Mulsaw
- Nmodel and SpecExplorer
- PBGT
- GUITAR
- …

Spec Explorer

- Is developed by Windows (so support is assured);
- Not many tutorials…
- But tons of references.
- Uses .NET language.

- The modeling is made by code, only later can the visual model be observed.

GUITAR

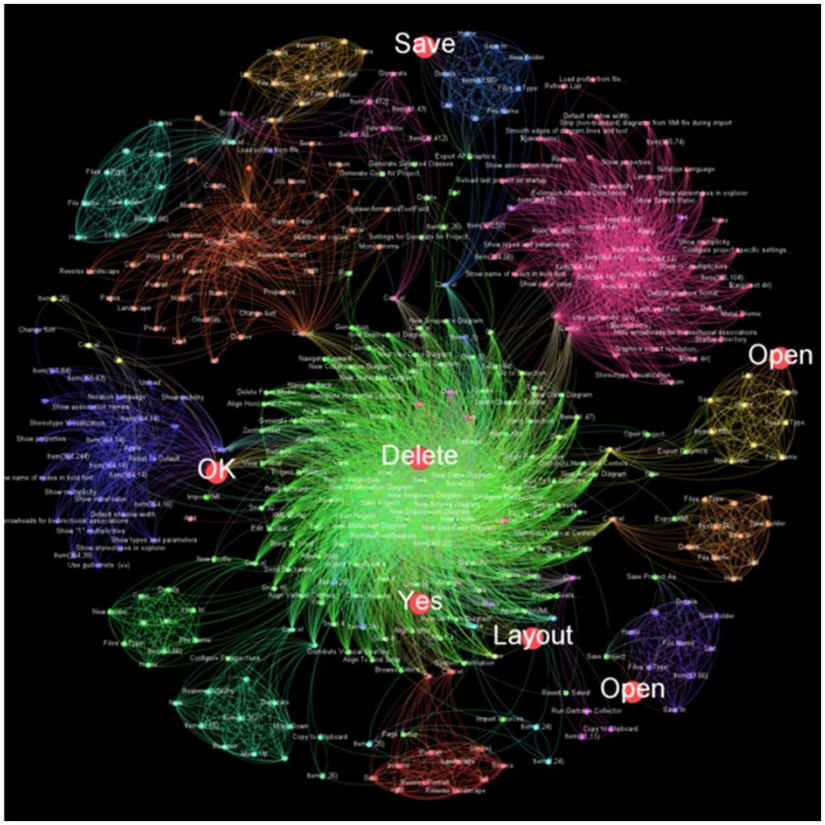
- Nice looking models! ----->

  EFG

- Can test GUI interfaces

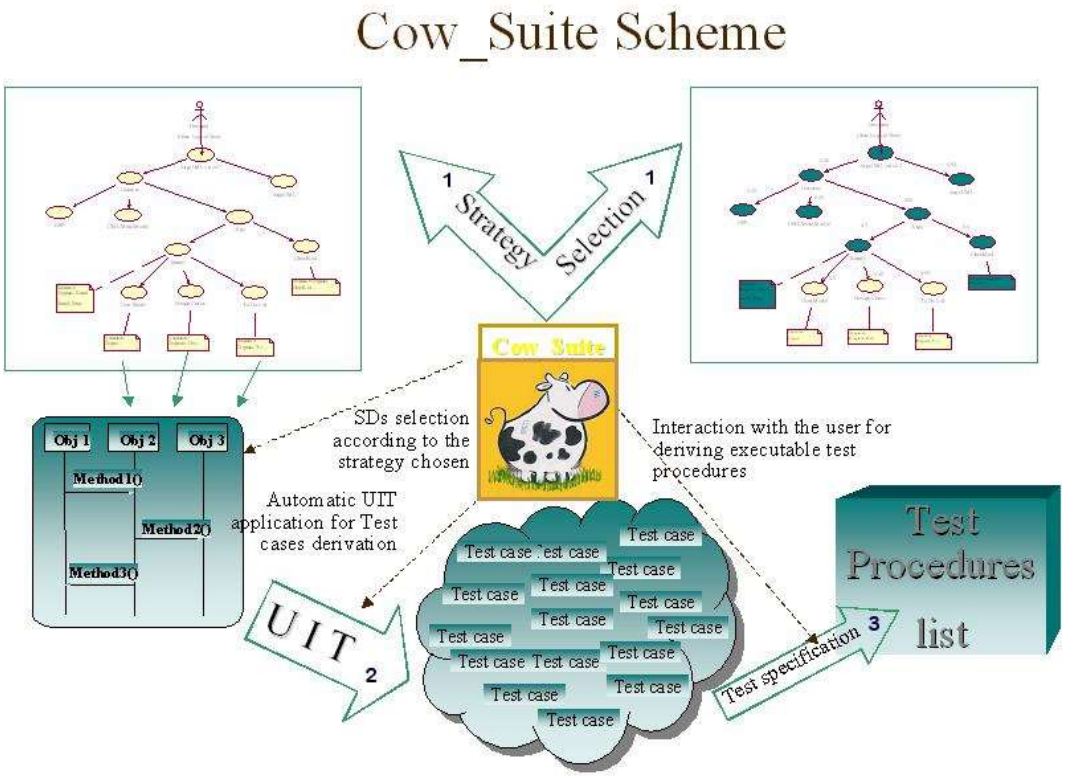
- Very few tutorials

- Difficult to use



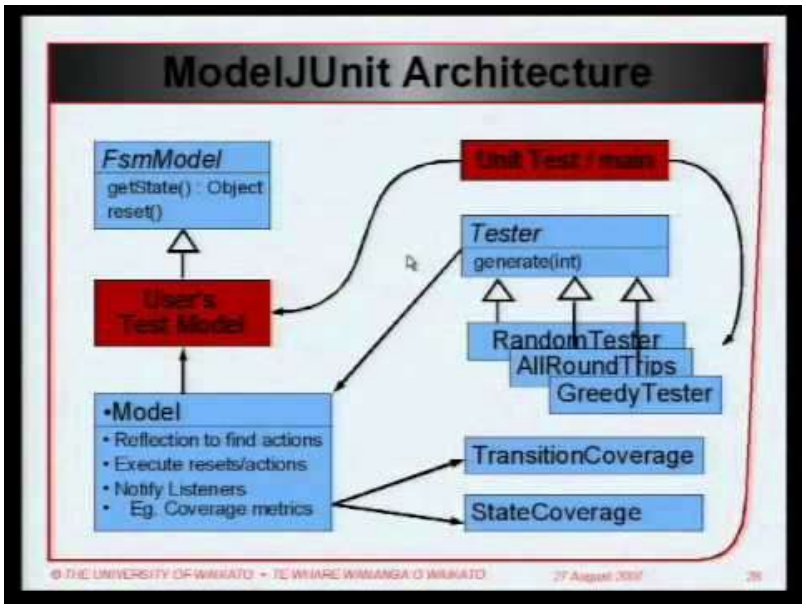
CowSuite

- An automated test strategy based on UML models



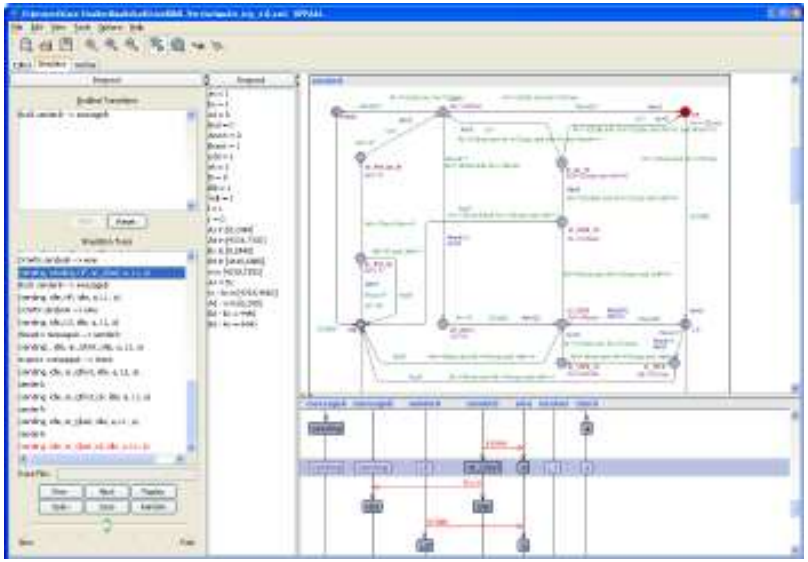
ModelJUnit

- ModelJUnit is a Java library that extends JUnit to support model-based testing. Models are extended finite state machines that are written in a familiar and expressive language: Java. ModelJUnit is an open source tool, released under the GNU GPL license.



T-UPPAAL

- T-UPPAAL - a tool for model based testing of embedded real-time systems that automatically generates and executes tests "online" from a state machine model of the implementation under test (IUT) and its assumed environment which combined specify the required and allowed observable (realtime) behavior of the IUT



Tools survey ...

Tool Name	Type	Modeling Language	Usability	Automatic generation of test cases	Export models to other formats	Model Simulation	Platform
UPPAAL	Academic	UML	Yes	No	Can export to XML file	Yes	Windows
CowSuite	Research	UML	Yes	Yes	-	Yes	Windows
JUMBL	Academic	TML (Markov Chain Models)	No	Yes	-	No	Windows /Unix
Enterprise Architect	Commercial	UML	Yes	Only for Modeling and does not support model based testing			
Spec Explorer	Research	SPEC # and ASML	Yes	Yes	Can export to XML file	No	Windows
Model Junit	Academic	Java	No	Yes	Can export to XML file	No	Windows

Tools survey ...

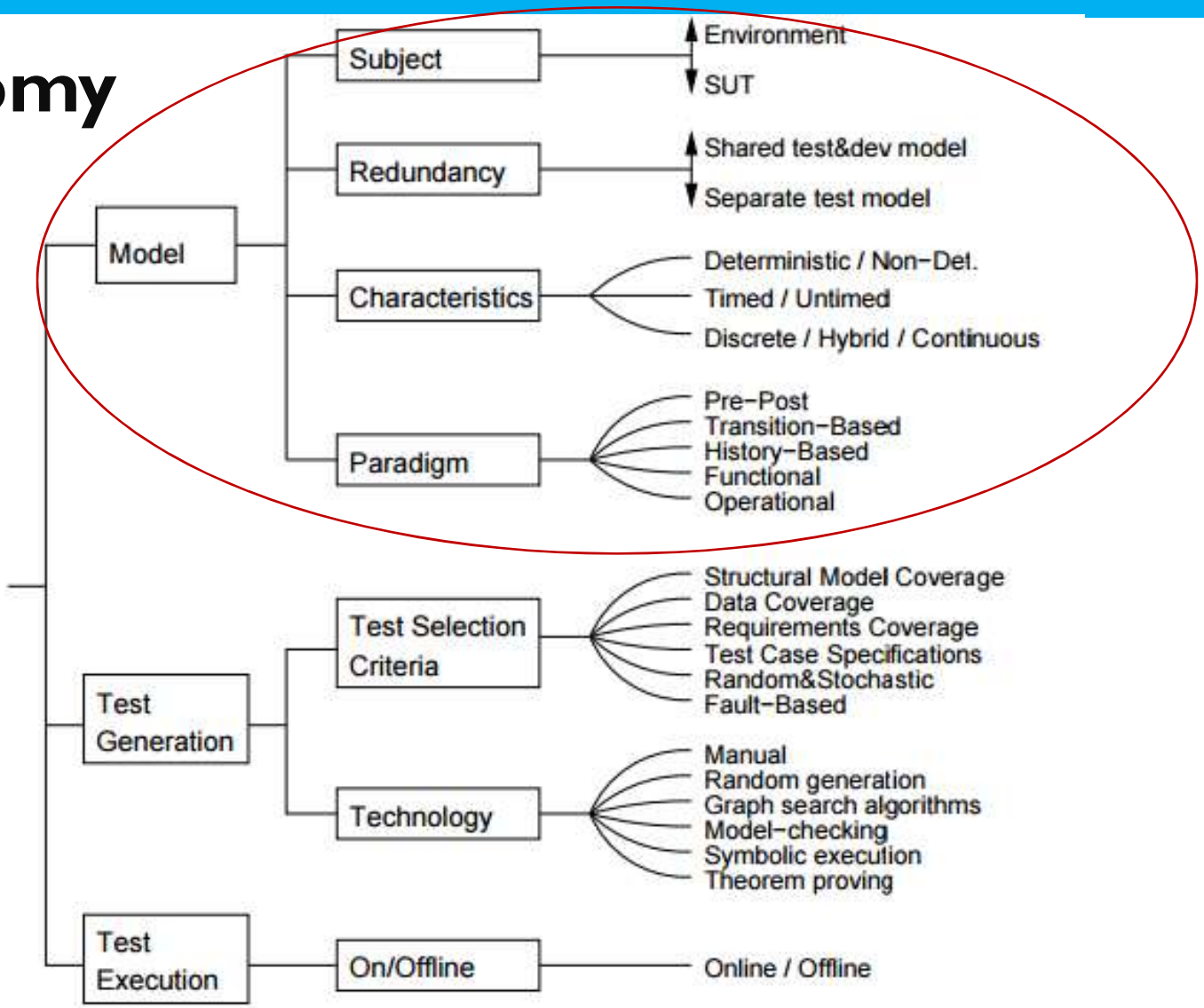
Tool	Modified	Ref.	Input format	Type	Description
BPM-Xchange	2014	[FS14]	BPMN, UML...	Commercial	BPM-Xchange creates test cases from business process models based on different criteria (statement, branch, path, condition). It can import models from several modeling tools, and can export test cases to Excel, HP Quality Center, etc.
Conformiq Designer	2014	[H07]	UML State Machines, QML	Commercial	Models can be created as UML State Machines and in Qtronic Modeling Language (QML). Tests can be exported to test management tools or TTCN-3. (Previously was Conformiq Qtronic.)
DTM	2013		custom activity model	Commercial	The DTM (Dialogues Testing Method) tool uses a custom activity model (actions, decisions...), and selects tests based on structural coverage (multiple condition or all path).
fMBT	2014		Custom (AAL)	Open source	fMBT (free Model-Based Testing) generates test cases from models written in the AAL/Python pre/postcondition language using different heuristics (random, weighted random, lookahead...).
GraphWalker	2014		FSM	Open source	Test generation from Finite State Machines. Search algoritms: A* or random, with a limit for various coverage criteria (state, edge, requirement). Formerly called as mbt.
JSXM	2014	[DBI12]	EFSM (Stream X-machines)	Academic	JSXM is model animation and test generation tool that uses a kind of EFSMs as its input. The generated tests can be transformed to JUnit test cases.
JTorX	2014	[B10]	LTS	Open source	JTorX is a reimplementation of TorX in Java with additional features. The LTS specification can be given in multiple format, and it can interact on-the-fly with the implementation under test.
					MaTeLo (Markov Test Logic) is a commercial

Tools survey …

- Go to Google and look for
  - Model based testing tools survey
  - Model based testing tools comparison
  - …

MBT Phases - Modeling

Taxonomy



[Utting, Pretschner and Legeard]

Modeling the system

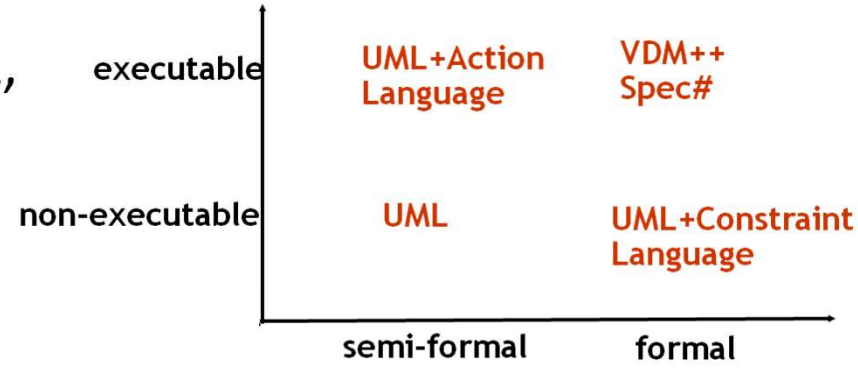
- Design the model to meet your testing goals
- Choose the right level of abstraction (which aspects of SUT you want to test)
- Choose a notation for modelling
- Once the model is written, ensure that it is accurate (verify and validate your model)

Built or Borrow?

- Reuse the development models – 100% reuse
  - Too much detail
  - Usually do not describe the *dynamic* behaviour of the SUT
  - It is neither abstract enough nor precise
- Models to generate code or reverse engineered
  - Lack of independence (implementation and test cases are derived from the same source code)
- Develop test models from scratch – 0% reuse
  - Maximum level of independence

Different modeling notations

- Visual (ex.: UML diagrams) or textual
- Formal models (ex.: Z, VDM, OCL, …)
- Executable (ex.: XTUML)
- Non-executable models (ex.: implicit)
- Deterministic vs non-deterministic
- Timed vs non timed
- Discrete vs continuous
- DSL



Formal notation

- Pre/Post (or Model-based). Ex.: VDM, Z, Spec#.
- Transition-based. Ex.: FSM, Petri nets.
- Behavior-based (or history-based). Ex.: CSP.
- Property-based (or functional-based). Ex.: OBJ.
- Hybrid approaches. Ex.: RAISE.
- UML with OCL
- GUI modeling: Abstract models (e.g., PiE e RED-PiE), grammars, FSM, Property based, Behavioral based, hybrid, EFG, Spec#, etc.

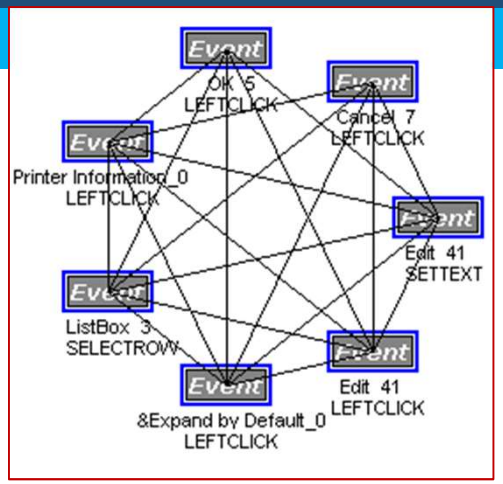
UML

- [“29 New Unclarities in the Semantics of UML 2.0 State Machines” by H. Fecher, J. Schönborn, M. Kyas, W. Roever]

- Unclarities categorized as
  - Incompleteness
  - Inconsistency
  - Ambiguity
  - Equivocality

GUI modeling

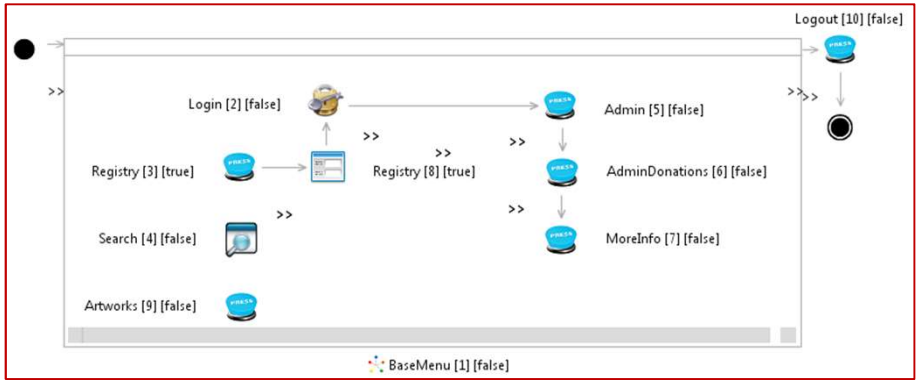
- EFG (GUITAR)



- Spec# (Mapping tool)

```
namespace MyNotepad;

//State variables
string text="",selText="";
bool dirty=false;
int posCursor=0;

// Start and close the Notepad application
[Action] void LaunchNotepad()
requires !IsOpen("Notepad"); {
  AddWindow("Notepad","",false);
  //... state variables initialization ...
}

[Action] void Close()
requires IsEnabled("Notepad"); {
  if (dirty) AddWindow("MsgClose","Notepad",true);
  else { RemoveWindow("Notepad");
      //... reset variables to initial values
      }
}
```

- UI Test Patterns (PBGT)

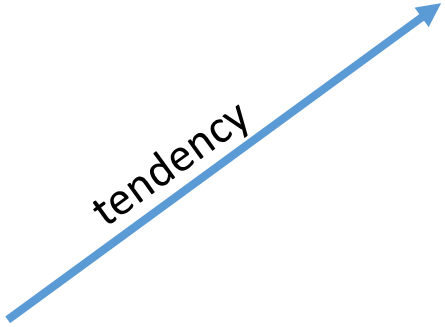


Testing models

- Tendency ?
  - Modelling testing goals

  instead of

  - Modelling SUT behavior



Choosing a modeling notation

- Which characteristics may be important?
  - Expressive power
  - Abstraction level
  - Can it be accepted in industry or not?
  - Simple enough?
  - Tool support

Choosing a modeling notation

- Pre/Post (for modelling complex data) and transition-based (for modelling control) are the **most common** notations used in model-based testing processes

- Whatever notation you choose, it has to be a *somehow* a **formal** language with precise semantics in order to write accurate models so as to be used as test oracles

Testing model

The characteristics of the modeling language will lead to different

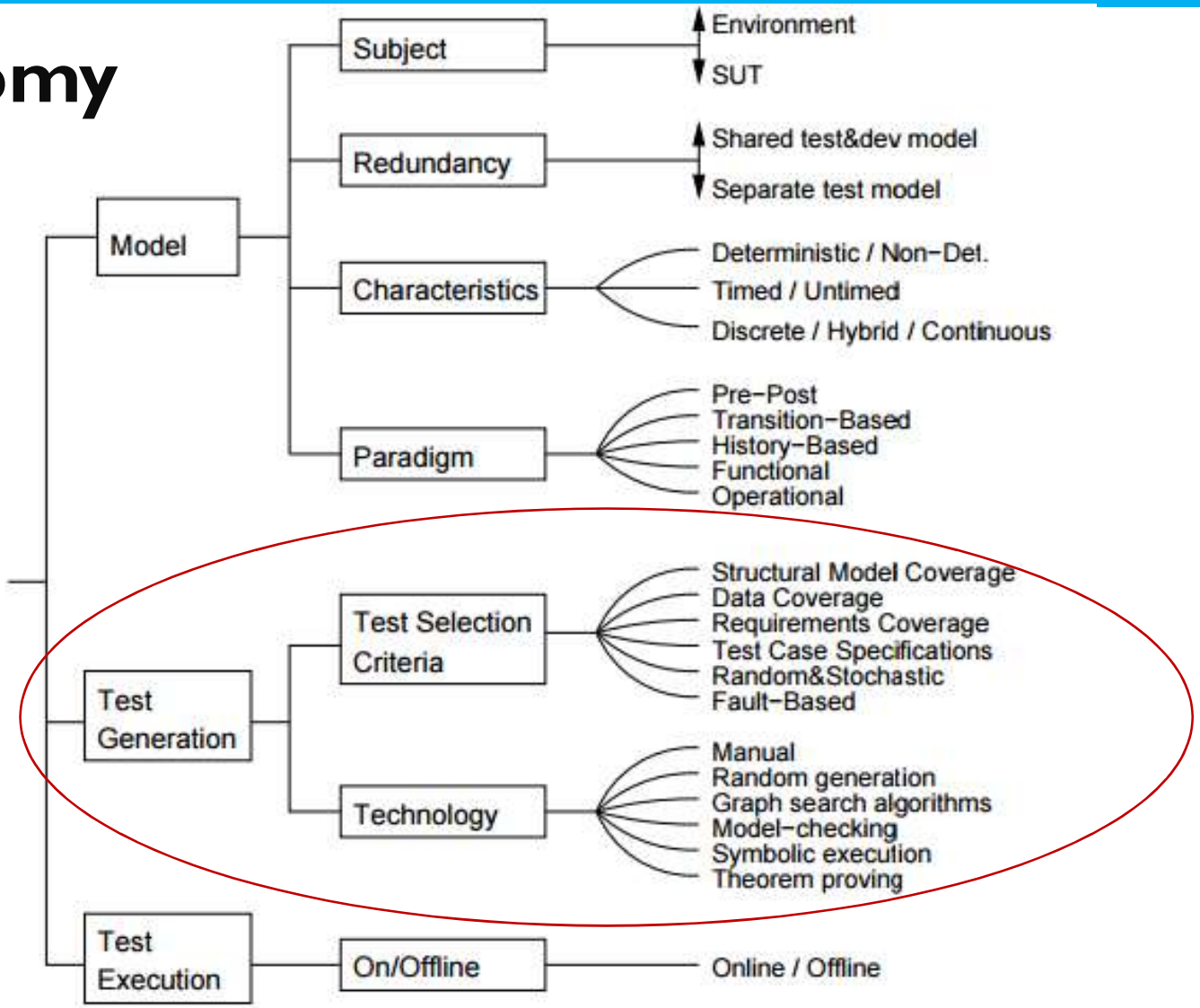
- test case generation techniques

and

- different kinds of test oracles (assertions versus executable specification)

MBT - Test case generation

Taxonomy



[Utting, Pretschner and Legeard]

Challenge

"To design a test generator that addresses the whole test design problem from

choosing input values

and

generating sequence of operation calls

to generate executable tests that include verdict information"

Challenge

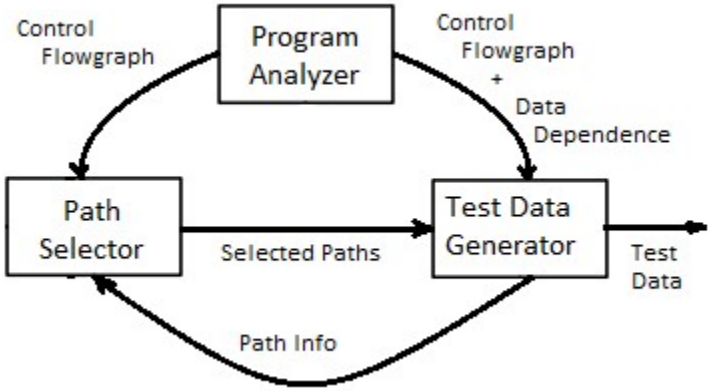
Test case explosion or combinatorial explosion

When SUT have:

- **Several parameters**, each with so many possible values that testing everything is infeasible
- **Many configuration parameters**, each with several possible values that testing each configuration is infeasible, e.g., versions of sw or hw component

Test data generators

- Based on a Mathematical Modelling we can simply state the Test Data Generator Problem as:
- Given a program **P** and a path **u**, generate input **x** ∈ **S**, so that **x** traverses path **u**.
- A Test Data Generator follows the following steps
  - Program Control Flow Graph Construction
  - Path Selection
  - Generating Test Data



Test data generators

- The path selector identifies the paths. Once a set of test paths is determined the test generator derives input data for every path that results in the execution of the selected path. This is done in two steps:
  - Find the predicate for the path
  - Solve the path predicate
- The solution will ideally be a system of equations which will describe the nature of input data so as to traverse the path. In some cases the generator provides the selector with feedback concerning paths which are infeasible etc.

Test data generators

- Random test data generators
  - Probably the simplest method
  - Can be used to generate input for any type of program
  - Does not generate quality test data as it does not perform well in terms of coverage
- Pathwise data generators
  - A computer program and a testing criterion (e.g., total path coverage, branch, etc.) as input and then generate test data that meet the selected criterion. Typical steps are 1) program control flow graph construction, 2) path selection, and 3) test data generation

Test data generators

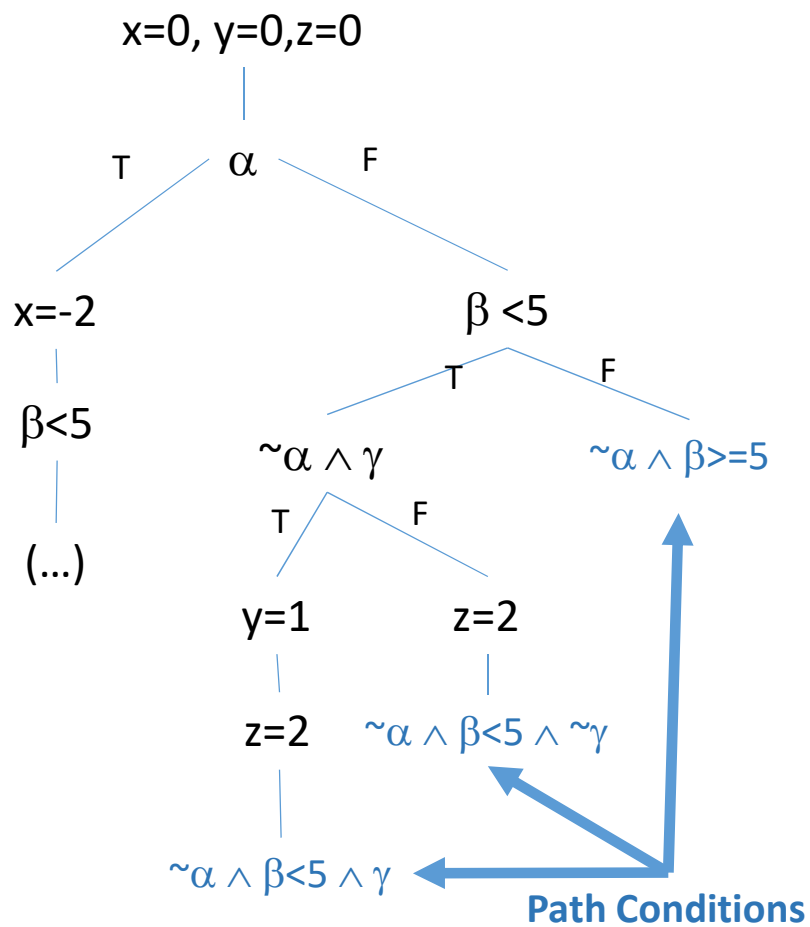
- Goal oriented generators
  - Channing approach tries to identify the nodes of the control flow graph that are vital to the execution of the goal node
  - Assertion oriented tries to find any path to an assertion that does not hold
- Intelligent test data generators
  - depend on sophisticated analysis of the code to guide the search of the test data. Intelligent Test Data Generators are essentially utilize one of the test data generation method coupled with the detailed analysis of the code.

Symbolic execution

- Symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation.
- It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

Symbolic execution

```
1.  int a=α, b=β, c=γ // symbolic
2.  int x=0, y=0, z=0
3.  if (a) {
4.     x = -2;
5.  }
6.  if (b<5) {
7.     if (!a && c) {y=1;}
8.     z=2;
9.  }
10. assert (x+y+z!=3)
```



Symbolic execution

When you get the path condition
you can then use a SAT (Satisfiability solver)
to find the values for the symbols
that satisfy such expression

- Some examples of SAT solvers are
  - Z3, STP and Yices

- SAT solvers are automated theorem provers

Combination strategies

Different combination strategies support different levels of coverage

- **1-wise**: requires that every value of every IPM parameter is included in, at least, one test case in the test suite
- **2-wise**: (also known as pair-wise) requires that every possible pair of values of any two IPM parameters is included in some test case

IPM – input parameter value

A (start): 1:negative, 2: zero, 3: positive
B (length of sequence): 1:zero, 2:one, 3: >1
C (direction of sequence): 1:negative, 2:positive

Test case	A	B	C
TC1			
TC2			
TC3			

1-wise

Test case	A	B	C
TC1			1
TC2			2
TC3			1
TC4	2		
TC5	2		
TC6	2		
TC7		1	
TC8	3	2	2
TC9		3	

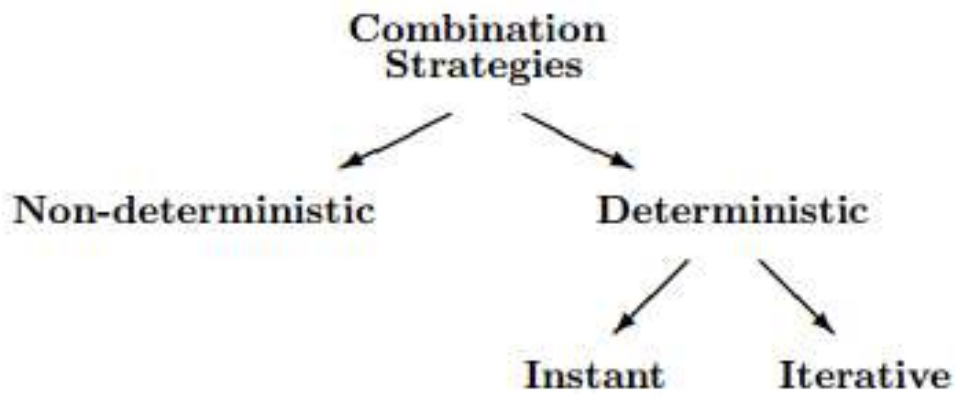
2-wise

Combination strategies

- **t-wise**: Samples the input domain to cover all T-wise combinations
  - 2-wise is pairwise testing
- **N-wise**: all combinations of values of N parameters should be covered at least once by test cases. Often too large to be practical

Combination strategies

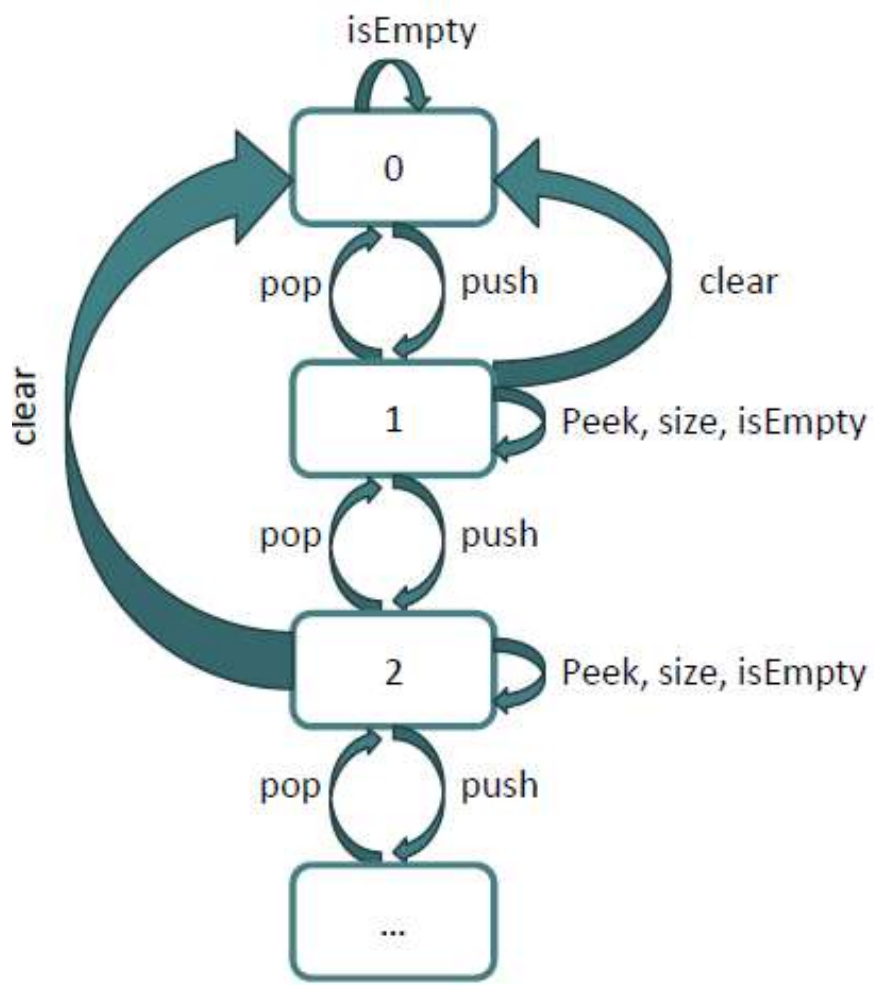
- **Non-deterministic** – rely to some degree of randomness. These solution may produce different solutions to the same problem at different times
- **Deterministic** – always produce the same solution for a given problem
- **Instant** – produce all combinations in an atomic step
- **Iterative** – build the solution one combination at a time



Generic Strategies – navigation through a FSM

- **Random** – traverses randomly the FSM
- **Greedy** – similar to random but give priority to paths not yet covered
- **AllRound** – similar to Greedy but prevents go through cycles of states more than once
- **Lookahead** – analyses the paths that departure from current state and chooses the one that goes into more states and new transitions (extremely slow)

Example - stack



FSM – State explosion problem

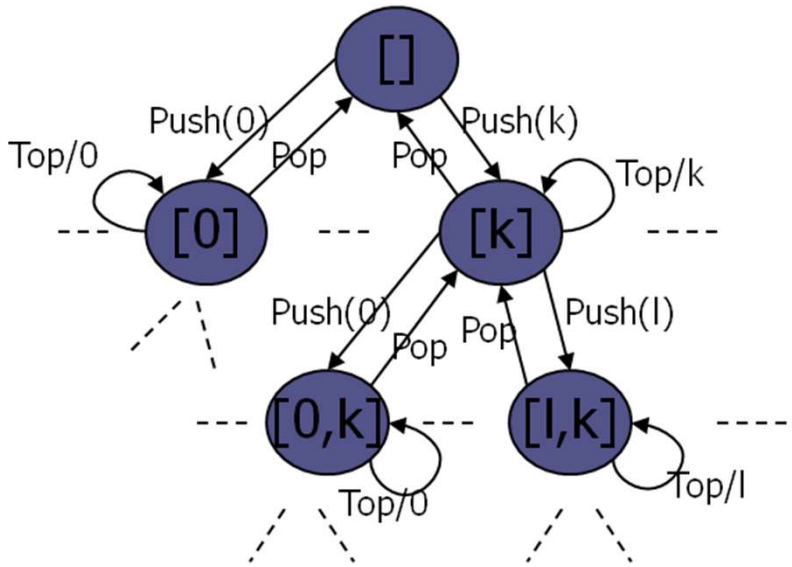
- In general the transition system describing a software system is infinite, but we can impose limits
- Goal
  - Create a state space of manageable size that satisfies a given testing goal
- Two main tasks
  - Restrict action parameter domains to interesting values
  - Restrict the state space to interesting states

- Note:
  The two tasks are not necessarily independent!

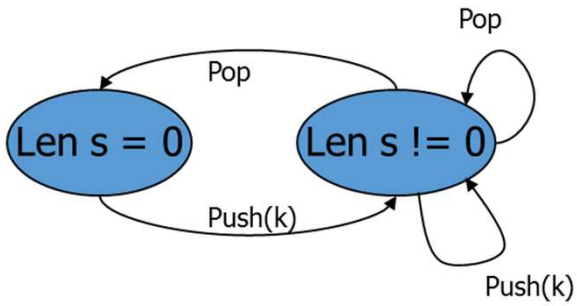
State space of a Stack model - SpecExplorer

Stack Spec# model

```
var Seq<int> content = Seq{};

[Action]
public void Push(int x) {
  content = Seq{x} + content;
}

[Action]
public void Pop() {
  requires !content.IsEmpty;
  content = content.Tail;
}

[Action]
public int Top() {
  requires !content.IsEmpty;
  return content.Head;
}
```



Projection on length



State explosion problem

- The following techniques are used in Spec Explorer
  - State filters
  - Stopping conditions
  - State groupings
  - Search order
  - Enabling conditions on actions

- Usually a combination of all (or some) of the techniques is needed to achieve desired effect

State Grouping

One or more state groupings may be defined

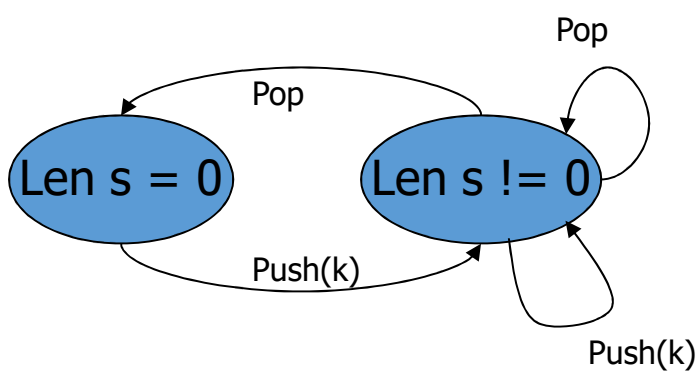
- A grouping G is a sequence of state based on expressions $g_1,\ldots,g_k$
- Two states *s* and *t* are in the same group G or G-equivalent if
  - $g_i^s = g_i^t$ for $1 \leq i \leq k$
- A G-group is the set of all G-equivalent states

Also used in viewing

Main purpose of groupings

- View all states in a G-group as being the same
- A way to define "what is an interesting state" from some testing point of view

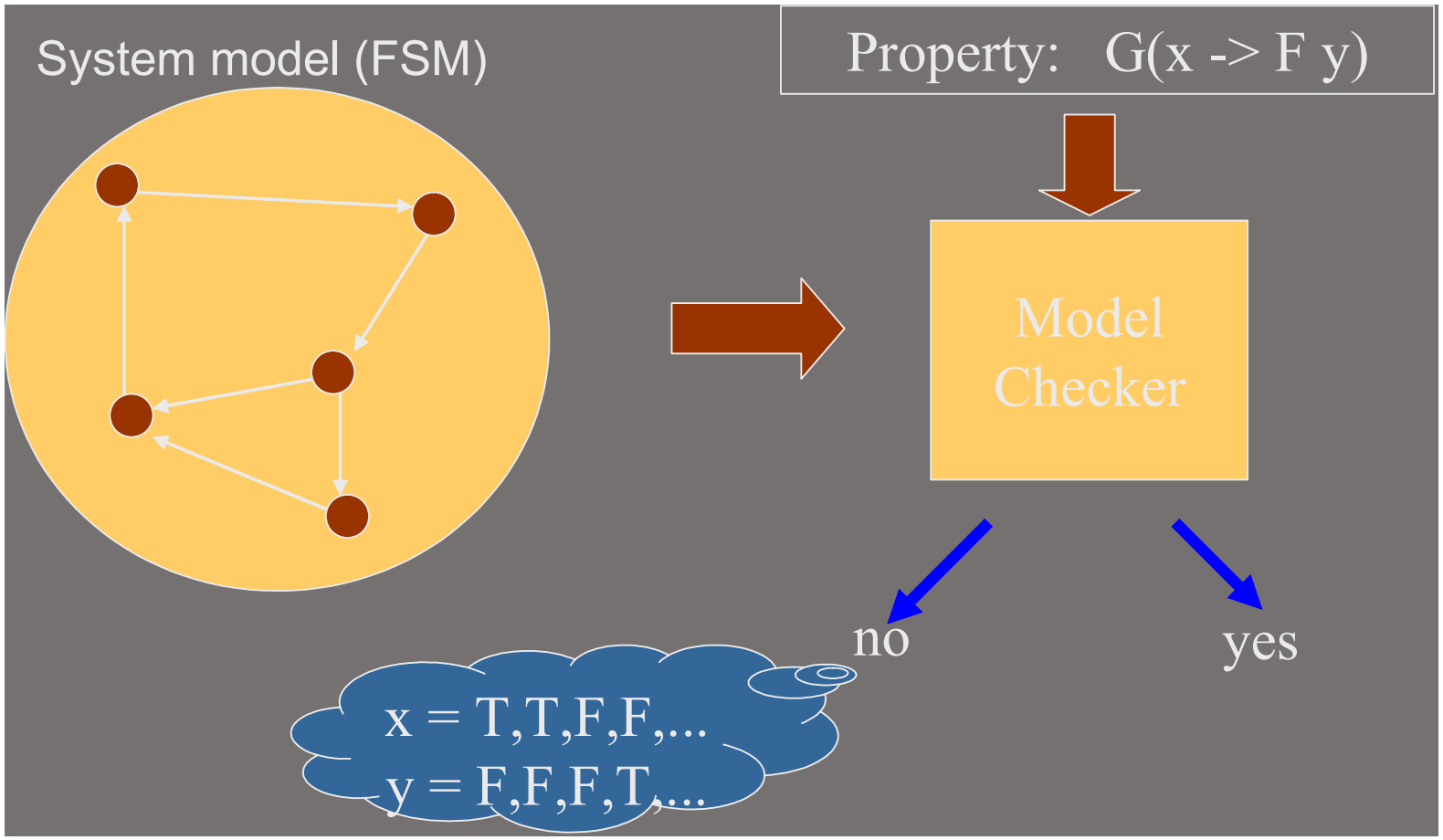
- Example: content.Size in the stack model



Techniques

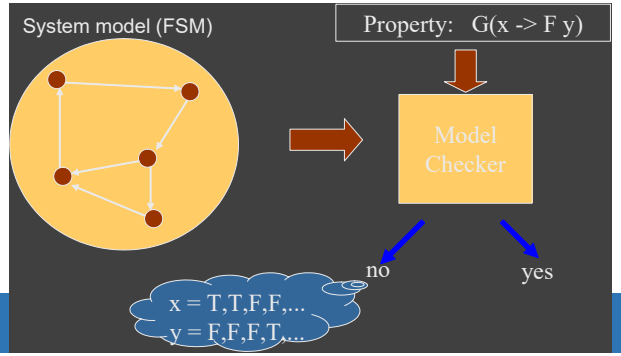
- Depends on the specification characteristics:
  - Pre/Post. Ex.: VDM, Z, Spec#
    - Structural converage
    - Partition testing
    - Generate FSM from the model. Ex.: Spec Explorer.
  - State/Transition based. Ex.: FSM, EFSM, Statecharts.
    - Traversal algorithms (state and transition coverage)
    - Model checking (algorithms based on state exploration; mutation analysis)
  - Property-based. Ex.: Obj
    - Rewriting rules
    - Constraint solving
  - Behaviour-based. Ex.: CSP
    - Trace analysis

Model Checking



Model Checking

- Whenever a property, expressed in temporal logic, does not hold in a system described as a FSM, model-checking tries to generate a counter-example
- When a counter-example is produced, it can be used as a test case – sequence of transitions in the FSM with inputs and expected outputs
- To be effective as a test-case generation technique, the properties about the system should be described in such a way that counter-examples produced by them can be used as test cases



Now that you have tests ...

- How do you know they are good enough?

How do you measure the quality?

- Capability to find defects
  - Particularly defects with higher risk
  - Risk = frequency of failure * impact of failure ≈ cost (of post-release failure)
- Capability to exercise multiple aspects of the system under test
  - Reduces the number of test cases required and the overall cost
- Low cost
  - Development: specify, design, code
  - Execution (fast)
  - Result analysis: pass/fail analysis, defect localization
- Easy to maintain
  - Reduce whole life-cycle cost
  - Maintenance cost ≈ size of test artefacts

(See also: "What Is a Good Test Case?", Cem Kaner, Florida Institute of Technology, 2003)

Test adequay/coverage criteria

- Adequacy criteria - Criteria to decide if a given test suite is adequate, i.e., to give us “enough” confidence that “most” of the defects are revealed
  - Used in the evaluation and in the design/selection of test cases
  - In practice, reduced to coverage criteria
- Coverage criteria
  - Requirements/specification coverage (black-box)
    - At least one test case for each requirement/specification statement
  - Model coverage
    - State-transition coverage
    - Use case and scenario coverage
  - Code coverage (white-box)
    - Control flow coverage (statement, decision, MC/DC coverage …)
    - Data flow coverage
  - Fault coverage / mutation testing

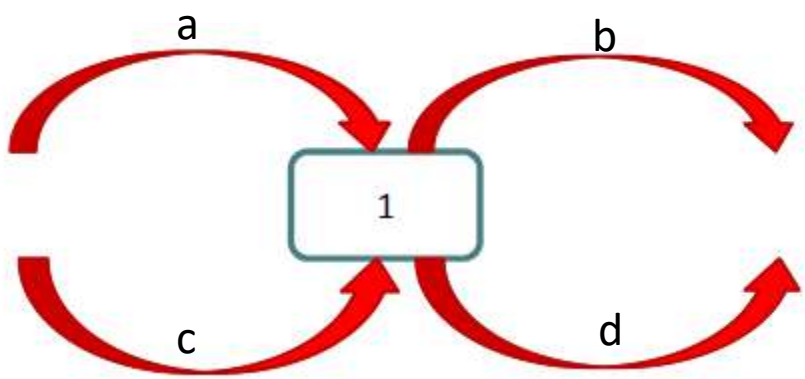
See also: “Software Unit Test Coverage and Adequacy”, Hong Zhu et al, ACM Computing Surveys, December 1997

Coverage analysis

- 1.Structural coverage criteria – aims to exercise code or model concerning some coverage goal
- 2.Data coverage criteria – aims to cover the input data space of an operation or transition
- 3.Fault-based criteria – aims to generate test suites appropriate for detecting specific kinds of faults
- 4.Requirements coverage criteria – aims to ensure that each requirement will be tested
- 5.Others
- …

Coverage analysis – FSMs

- Action coverage
- State coverage
- Transition coverage
- Transition pair coverage
  - Example with four possible pairs (ab,cd,ad,cb)



Coverage analysis – Code

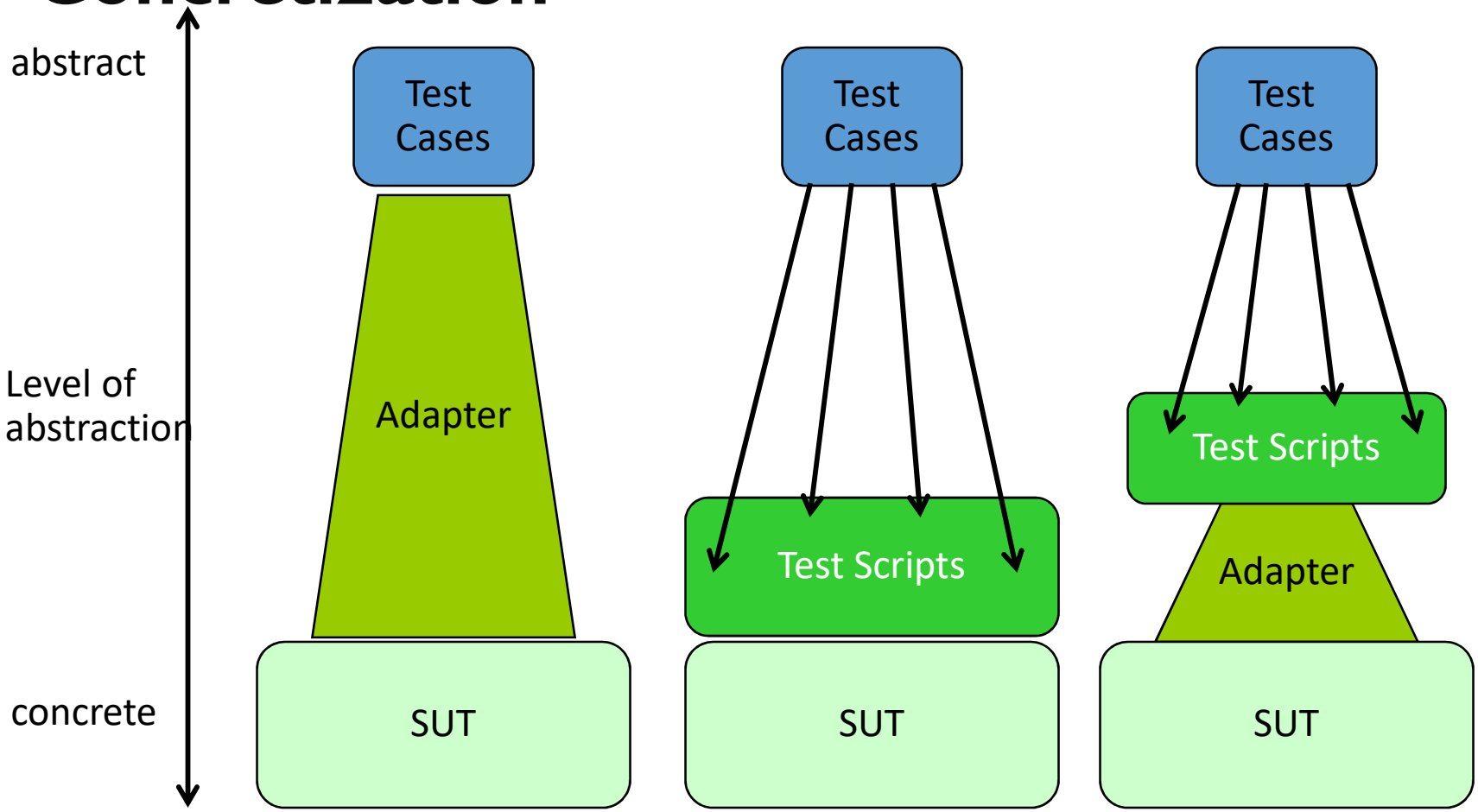
- Statement converage
- Decision coverage
- Condition coverage
- Condition and decision coverage
- MCDC
- …

MBT - Test case concretization

Mapping

- Translate abstract test cases into concreate test cases
- Establish a mapping between abstract and concreate
- **API**
  - Map methods from the model to methods of the SUT
- **GUI**
  - Map methods of the API to Graphical User Interface objects
  - But then you have to identify them during test case execution!

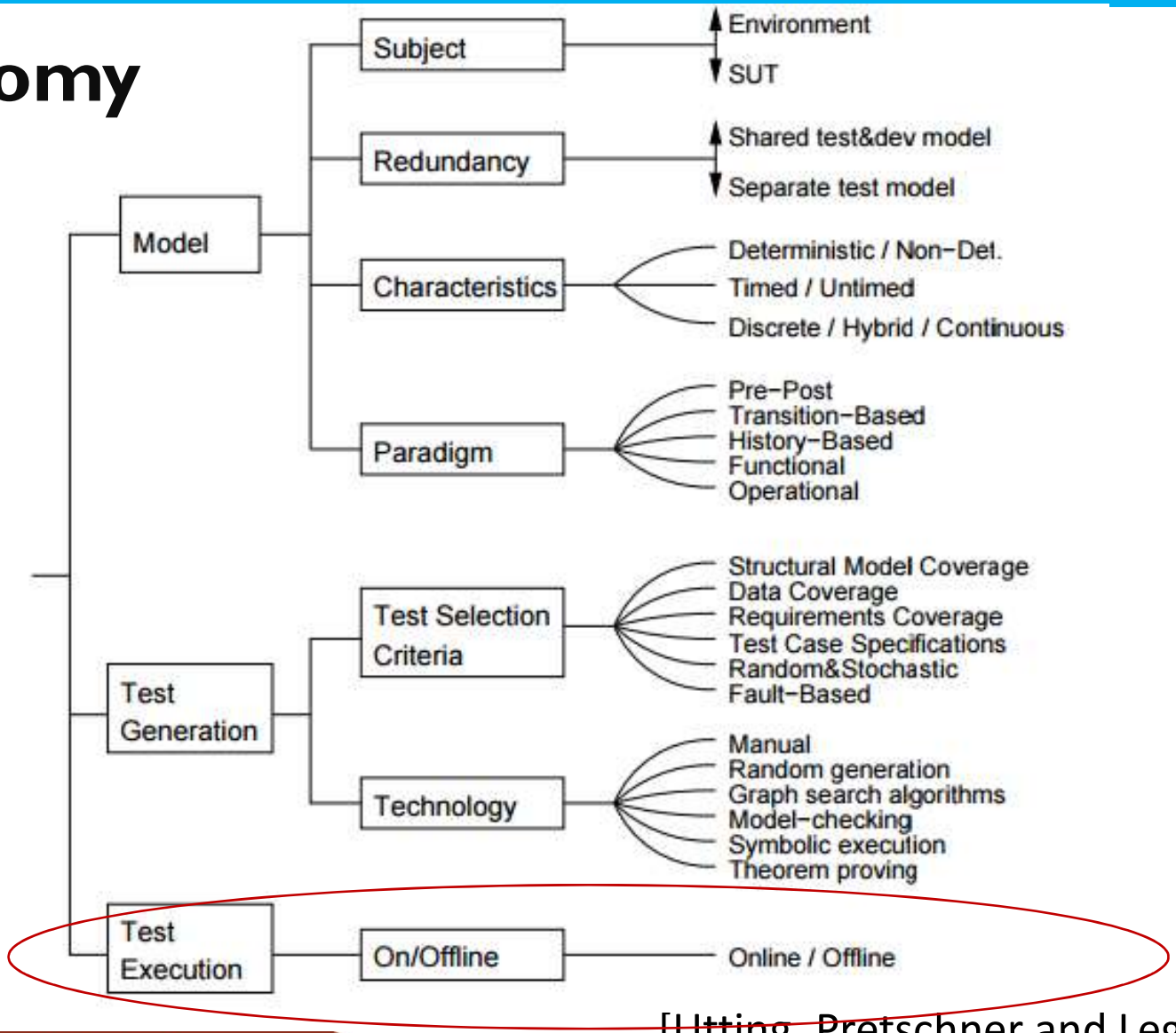
Concretization



a) Adaptation b) Transformation c) Mixed

MBT - Test case execution

Taxonomy



[Utting, Pretschner and Legeard]

Execution

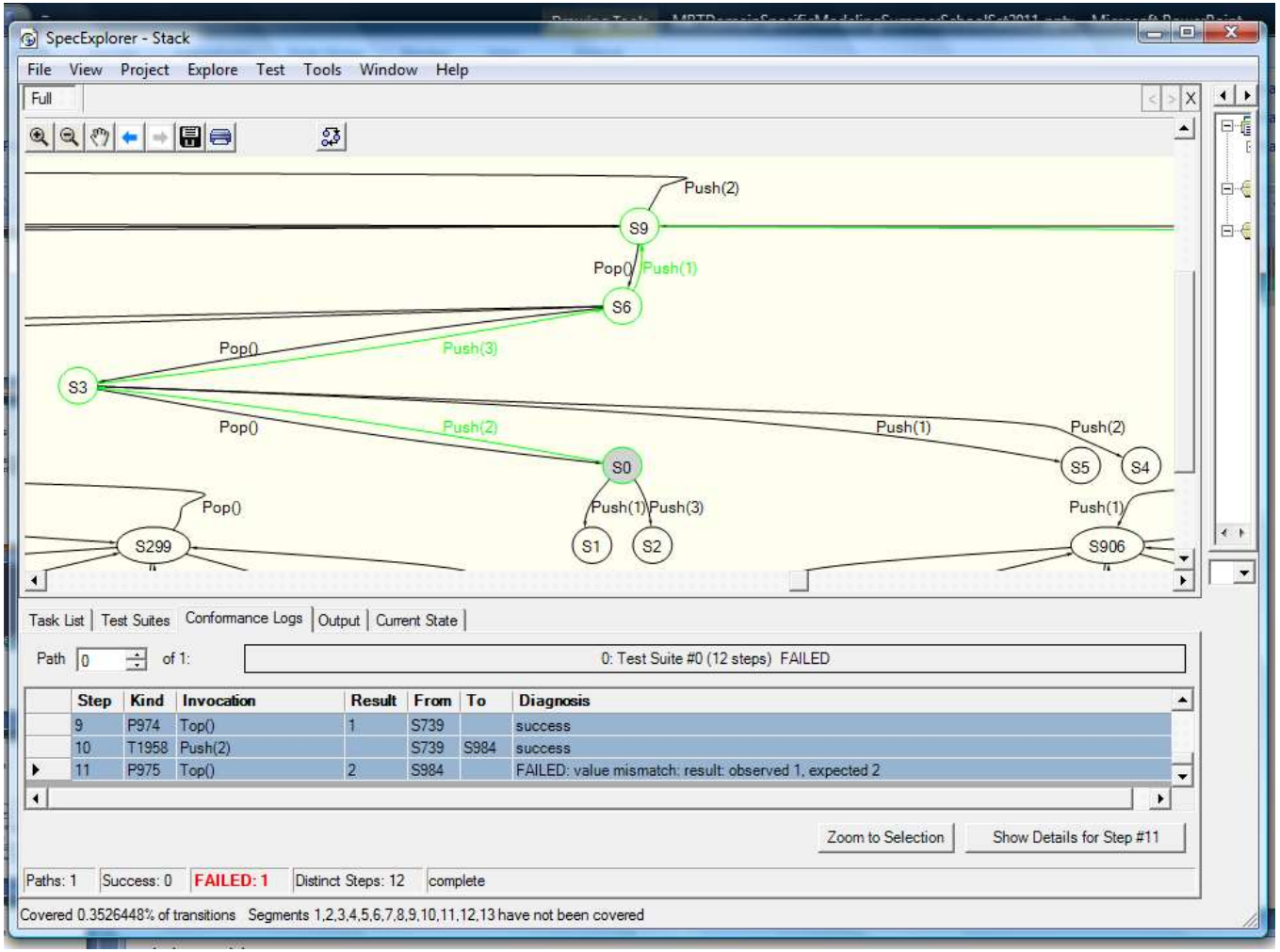
- **Lock-step** mode: results are compared after each step
- **Batch-oriented** mode: the test suite is run as a whole in the specification level, and expected results are kept in memory for later comparison with the results obtained from the execution of the implementation (which is performed in a different execution time instant)
  - One advantage of the batch-oriented way is the need to execute the model only once and not every time test cases are executed. The main drawback is the additional need of memory to keep the results expected
- **On-the fly** combines in a single algorithm the test case generation and execution and executes each operation as a "lock-step" in each level comparing results after each of those execution steps

MBT - Analysis

Locate the faults

- Mainly a manual activity
- Some tools provide information that may be useful to diminish the effort to find the origins of the failures detected
- 50% of the faults are in the code and 50% in the model!

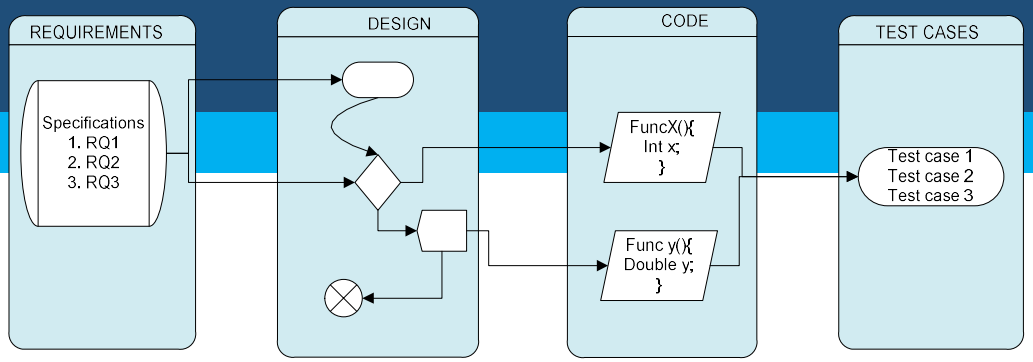
Spec Explorer example



Traceability

Problem statement

- "Achieving traceability between the test cases generated and requirements in a ***model based testing*** (MBT) scenario"



Traceability

- Ensure interrelation between different artifacts involved software engineering
- Enable to perform quick and easy impact analysis
- Enhance the product quality

SDLC Phase	Artifacts
• Specification	- requirements
• Design	- design models
• Implementation	- code (functions, libraries)
• Validation	- test case

Traceability would enable

- Measure relevance of the test case generated from model
- Assess how a modification in the model would effect the generated test case
- Provides the effective coverage

Summary

- It was more common on API testing
- Now we see GUI testing and mobile testing
- It is more common in industry nowadays
- It is a hot research topic
- There are some research problems still open

TVVS - Test, Verification and Validation of Software

Test management

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Some definitions

Test management

- The planning, estimating, monitoring and control of test activities, typically carried out by a test manager. [ISTQB]

Test management tool

- A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting. [ISTQB]

Some definitions

Test manager / test leader

- The person responsible for project management of testing activities and resources, and evaluation of a test object. The individual who directs, controls, administers, plans and regulates the evaluation of a test object. [ISTQB]

Tester

- A skilled professional who is involved in the testing of a component or system. [ISTQB]

Tasks of the Test Leader

- Coordinate the test strategy and plan with project managers and others
- Write or review a test strategy for the project, and test policy for the organization
- Contribute the testing perspective to other project activities, such as integration planning
- Plan the tests – considering the context and understanding the test objectives and risks
- Initiate the specification, preparation, implementation and execution of tests, monitor the test results and check the exist criteria
- Adapt planning based on test results and progress and take any action necessary to compensate for problems

Tasks of the Test Leader

- Set up adequate configuration management of testware for traceability
- Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
- Decide what should be automated, to what degree, and how
- Select tools to support testing and organize any training in tool use for testers
- Decide about the implementation of the test environment
- Write test summary reports based in the information gathered during testing

Tasks of the Tester

- Review and contribute to test plans
- Analyse, review and assess user requirements, specifications and models for testability
- Create test specification
- Set up the test environment
- Prepare and acquire test data

Tasks of the Tester

- Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results
- Use test administration or management tools and test monitoring tools as required
- Automate tests
- Measure performance of components and systems
- Review tests developed by others

Test planning activities

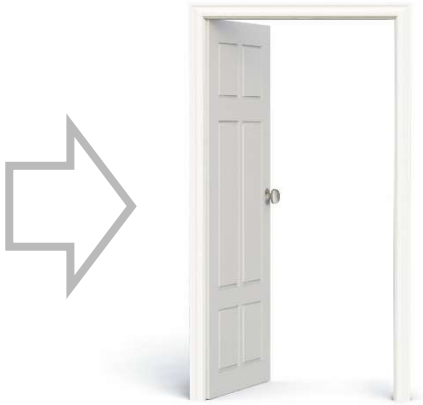
- Determining the scope and risks and identifying the objectives of testing
- Defining the overall approach of testing, including the definition of the test levels and entry and exit criteria
- Integrating and coordinating the testing activities into the software life cycle activities
- Making decisions about what to test, what roles will perform the test activities, how the test activities should be done, and how the test results will be evaluated
- Scheduling test analysis and design activities

Test planning activities

- Scheduling test implementation, execution and evaluation
- Assigning resources for the different activities defined
- Defining the amount, level of detail, structure and templates for the test documentation
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues
- Selecting the level of detail for test procedures in order to provide enough information to support reproducible test preparation and execution

Entry criteria

- Typically entry criteria may cover the following:
  - Test environment availability and readiness
  - Test tool readiness in the test environment
  - Testable code availability
  - Test data availability



Exit criteria

- Thoroughness measures, such as coverage of code, functionality or risk
- Estimates of defect density or reliability measures
- Cost
- Residual risks, such as defects not fixed or lack of test coverage in areas
- Schedules such as those based on time to market

Test estimation

Test estimation

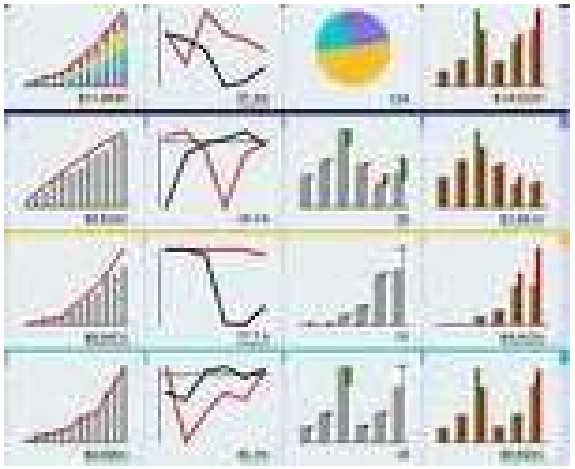
- The calculated approximation of a result related to various aspects of testing (e.g., effort spent, completion date, costs involved, number of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy. [ISTQB]

- How do you estimate the test effort

?

Test estimation

- The metrics-based approach: estimating the testing effort based on metrics of former or similar projects or based on typical values
  - E.g., complexity; functional point analysis, test point analysis, …
- The expert-based approach: estimating the tasks based on estimates made by the owner of the tasks or by experts



Test progress monitoring

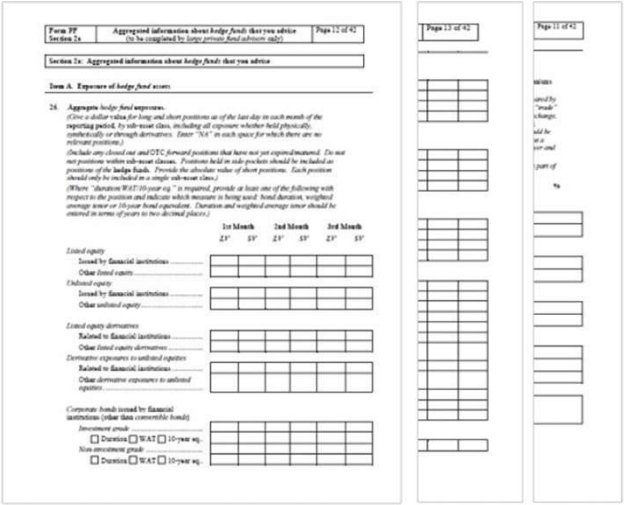
- Percentage of work done in test case preparation
- Percentage of work done in test environment preparation
- Test case execution
- Defect information
- Test coverage of requirements, risks or code
- Subject confidence of testers in the product
- Dates of test milestones
- Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test

Test control

- Making decisions based on information from test monitoring
- Re-prioritizing tests when an identified risk occurs
- Changing the test schedule due to availability or unavailability of a test environment
- Setting an entry criterion requiring fixes to have been re-tested by a developed before accepting them into a build

Test reporting

- What happened during a period of testing, such as dates when exit criteria were met
- Analysed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in the tested software



Test strategy, test approach

Typical approaches include:

- **Analytical approaches**, such as risk-based testing where testing is directed to areas of greatest risk
- **Model-based approaches**, such as stochastic testing using statistical information about failure rates or usage
- **Methodical approaches**, such as failure-based, experience-based, checklist-based, and quality characteristic-based
- **Process- or standard-compliant approaches**, such as those specified by industry-specific standards or the various agile methodologies

Test strategy, test approach

Typical approaches include:

- **Dynamic and heuristic approaches**, such as exploratory testing where testing is more reactive to events than pre-planned, and where execution and evaluation are concurrent tasks
- **Consultative approaches**, such as those in which test coverage is driven primary by the advice and guidance of technology and/or business domain experts outside the test team
- **Regression-averse approaches**, such as those that include reuse of existing test material, extensive automation of functional regression tests, and standard test suites

Risk based approach to testing

- Determine the test techniques to be employed
- Determine the extend of testing to be carried out
- Prioritize testing in an attempt to find the critical defects as early as possible
- Determine whether any non-testing activities could be employed to reduce risk (e.g., providing training to inexperienced designers)

Project risks

- Organizational factors
  - Skill, training and staff shortages
  - Personnel issues
  - Political issues, such as
    - Problems with testers communicating their needs and test results
    - Failure by the team to follow up on information found in testing and reviews
  - Improper attitude toward or expectations of testing

Project risks

- Technical issues
  - Problems in defining the right requirements
  - The extend to which requirements cannot be met given existing constraints
  - Test environment not ready on time
  - Late data conversion, migration planning and development and testing data conversion/migration tools
  - Low quality of the design, code, configuration data, test data and tests
- Supplier issues
  - Failure of a third party
  - Contractual issues

Product risks

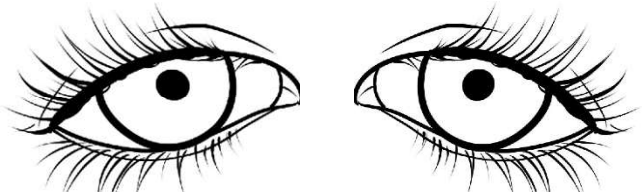
- Failure-prone software delivered
- The potential that the software/hardware could harm to an individual or company
- Poor software characteristics (e.g., functionality, reliability,…)
- Poor data integrity and quality (e.g., data migration…)
- Software that does not perform its intended functions

Test independence

- No independent testers – developers test their own code
- Independent testers within the development teams
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from business organization or user community
- Independent test specialists for specific test types such as usability
- Independent testers outsourced or external to the organization

Benefits of independence

- Independent testers see other and different defects and are unbiased
- Independent testers can verify assumptions people made during specification and implementation of the system



Drawbacks of test independence

- Isolation from the development team (if treated as totally independent)
- Developers may lose a sense of responsibility for quality
- Independent testers may see as a bottleneck or blamed for delays in release

Configuration management

Configuration management

- A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [ISTQB]

Configuration management

- For testing, configuration management may involve ensuring the following:
  - All items of testware are identified, version controlled, tracked for changes, related to each other and related to development items so that traceability can be maintained throughout the test process
  - All identified documents and software items are referenced unambiguously in test documentation

Incident management

Incident management

- The process of recognizing, investigating, taking action and disposing of incidents. It involves logging incidents, classifying them and identifying the impact. [ISTQB]

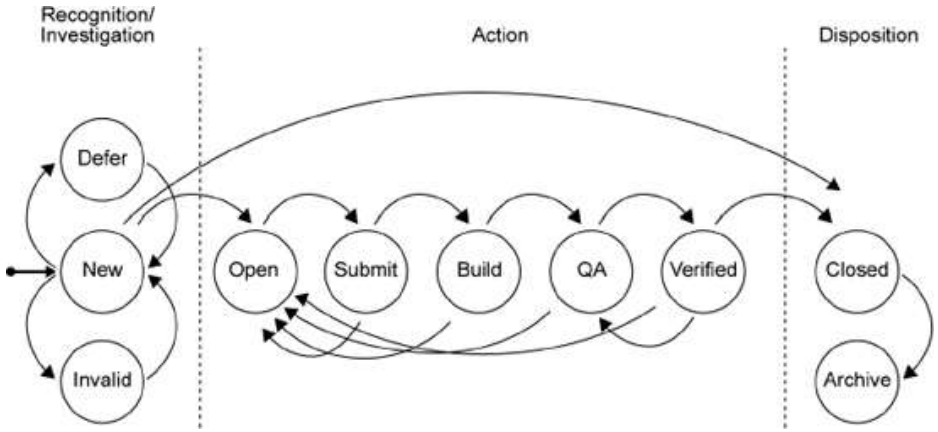
Incident management tool

- A tool that facilitates the recording and status tracking of incidents. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of incidents and provide reporting facilities. [ISTQB]

Incident management

Incident reports have the following objectives:

- Provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
- Provide test leaders a means of tracking the quality of the system under test and the progress of the testing
- Provide ideas for test process improvement



Incident

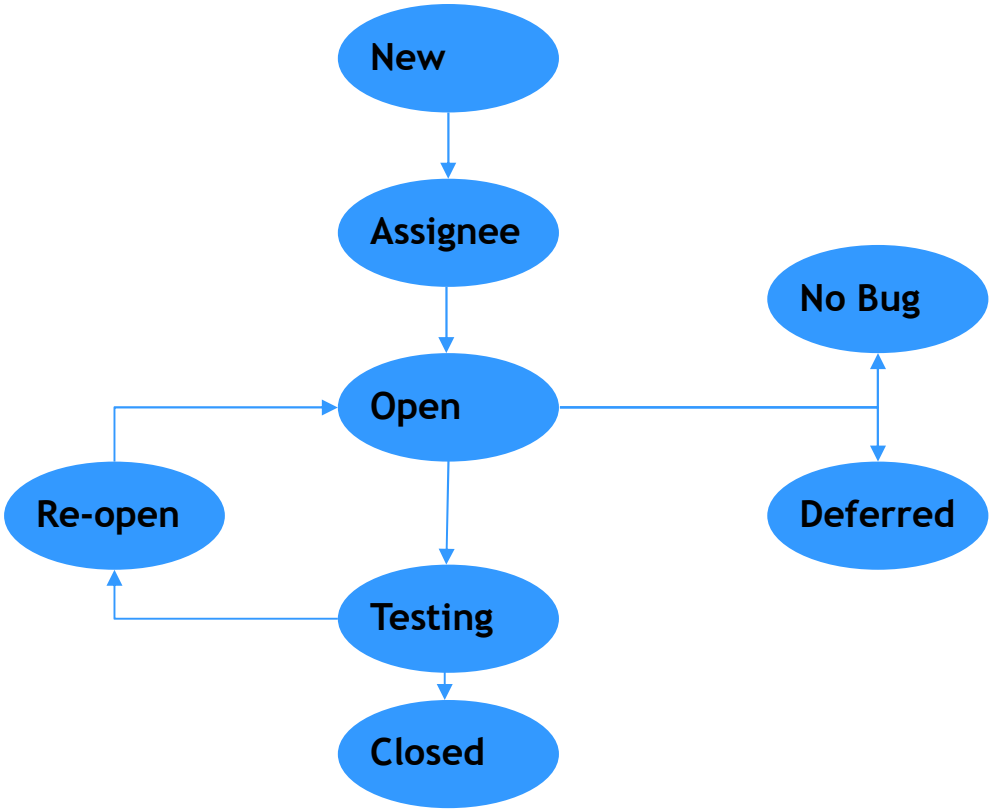
Incident

- Any event occurring that requires investigation. [ISTQB]

Incident

- Incident is raised whenever actual results vary from expected results
- Also called as bugs, defects, problems or issues
- Incident management is very important
- Project status is calculated from incident status
- Tools are available to help incident management

Defect Life Cycle - example



The details that are normally included on an incident report

- Data of issue, issuing organization, assigner, assigne
- Scope, severity and priority of the incident
- References, including the identity of the test case specification that revealed the problem
- Expected and actual results
- Date the incident was discovered
- Identification of the test item (configuration item and environment)
- Software or system life-cycle process in which the incident was observed

The details that are normally included on an incident report

- Description of the incident to enable reproduction and resolution, including logs, database dumps or screenshots
- Urgency/priority to fix
- Status of the incident (e.g., open, duplicate, waiting to be fixed, fixed awaiting confirmation test or closed)
- Conclusions, recommendations and approvals
- Change history, such as the sequence of actions taken by project team members with respoect to the incident to isolate, repair and confirm it as fixed

Test Incident Report - IEEE829

- Test Incident Report Identifier
- Summary
  - Provide enough details to enable others to understand how the incident was discovered
- Incident Description
  - Provide as much details on the incident as possible. For example: Inputs; Expected Results; Actual Results; Anomalies; Date and Time; Procedure Step; Attempts to Repeat; Testers; Observers
- Impact
  - Describe the actual/potential damage caused by the incident.
    - Severity – The potential impact to the system
    - Priority – The order in which the incidents are to be addressed

TVVS - Test, Verification and Validation of Software

Exercises

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Q#1

- An employee's bonus is to be calculated. It cannot be negative, but it can be calculated down to zero. The bonus is based on the length of employment:
  - less than or equal to 2 years,
  - more than 2 years but less than 5 years,
  - 5 to 10 years inclusively or longer than 10 years.
- What is the minimum number of test cases required to cover all valid equivalence partitions for calculating the bonus?
  - a) 3.
  - b) 5.
  - c) 2.
  - d) 4.
- Select ONE option.

Q#2

- A speed control and reporting system has the following characteristics:
  - If you drive 50 km/h or less, nothing will happen.
  - If you drive faster than 50 km/h, but no more than 55 km/h, you will be warned.
  - If you drive faster than 55 km/h but not more than 60 km/h, you will be fined.
  - If you drive faster than 60 km/h, your driving license will be suspended.
  - The speed in km/h is available to the system as an integer value.
- Which would be the most likely set of values (km/h) identified by applying the boundary value analysis, where only the boundary values on the boundaries of the equivalence classes are relevant?
  - a) 0, 49, 50, 54, 59, 60.
  - b) 50, 55, 60.
  - c) 49, 50, 54, 55, 60, 62.
  - d) 50, 51, 55, 56, 60, 61.
- Select ONE option.

Q#3

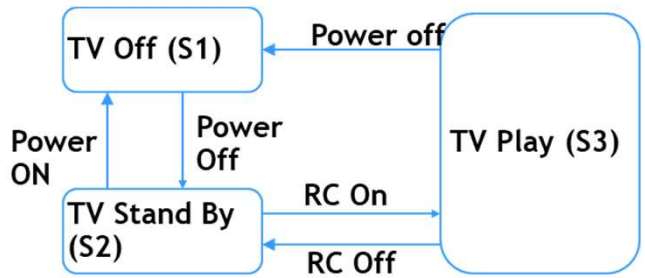
- A company's employees are paid bonuses if they work more than a year in the company and achieve a target which is individually agreed before.
- These facts can be shown in a decision table:

Test-ID		T1	T2	T3	T4
Condition 1	Employment for more than 1 year?	Yes	No	No	Yes
Condition 2	Agreed target?	No	No	Yes	Yes
Condition 3	Achieved target?	No	No	Yes	Yes
Action	Bonus payment	No	No	No	No

- Which of the following test cases represents a situation that can happen in real life, and is missing in the above decision table?
  - a) Condition1 = YES, Condition2 = NO, Condition3 = YES, Action= NO
  - b) Condition1 = YES, Condition2 = YES, Condition3 = NO, Action= YES
  - c) Condition1 = NO, Condition2 = NO, Condition3 = YES, Action= NO
  - d) Condition1 = NO, Condition2 = YES, Condition3 = NO, Action= NO
- Select ONE option.

Q#4

- Which of the following statements about the given state transition diagram and table of test cases is TRUE?



States
S1: TV Off
S2: TV stand by
S3: TV Play
Transitions
S1->S2: Power off
S2->S1: Power On
S3->S1: Power Off
S2->S3: RC On
S3->S2: RC Off

Test Case	1	2	3	4	5
Start State	S1	S2	S2	S3	S3
Input	Power Off	Power On	RC On	RC Off	Power Off
Expected Final State	S2	S1	S3	S2	S1

- a) The given test cases cover both valid and invalid transitions in the state transition diagram.
- b) The given test cases represent all possible valid transitions in the state transition diagram.
- c) The given test cases represent some of the valid transitions in the state transition diagram.
- d) The given test cases represent pairs of transitions in the state transition diagram.

- Select ONE option.

Q#5

- A video application has the following requirement: The application shall allow playing a video on the following display resolution:
  - 1. 640x480.
  - 2. 1280x720.
  - 3. 1600x1200.
  - 4. 1920x1080.
- Which of the following list of test cases is a result of applying the equivalence partitioning test technique to test this requirement?
  - a) Verify that the application can play a video on a display of size 1920x1080 (1 test case).
  - b) Verify that the application can play a video on a display of size 640x480 and 1920x1080 (2 test cases).
  - c) Verify that the application can play a video on each of the display sizes in the requirement (4 test cases).
  - d) Verify that the application can play a video on any one of the display sizes in the requirement (1 test case).
- Select ONE option.

Q#6

- A fitness app measures the number of steps that are walked each day and provides feedback to encourage the user to keep fit.
- The feedback for different numbers of steps should be:
  - Up to 1000 - Couch Potato!
  - Above 1000, up to 2000 - Lazy Bones!
  - Above 2000, up to 4000 - Getting There!
  - Above 4000, up to 6000 - Not Bad!
  - Above 6000 - Way to Go!
- Which of the following sets of test inputs would achieve the highest equivalence partition coverage?
  - a) 0, 1000, 2000, 3000, 4000
  - b) 1000, 2001, 4000, 4001, 6000
  - c) 123, 2345, 3456, 4567, 5678
  - d) 666, 999, 2222, 5555, 6666
- Select ONE option

SAMPLE EXAM (ISTQB)

Q#1

- Which of the following provides the BEST description of a test case?
  - a) A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.
  - b) A set of input values and expected results, with execution preconditions and execution postconditions, developed for a particular test condition.
  - c) An attribute of a system specified by requirements documentation (for example reliability, usability or design constraints) that is executed in a test.
  - d) An item or event of a system that could be verified by one or more test conditions, e.g., a function, transaction, feature, quality attribute, or structural element.
- Select ONE option.

Q#2

- Which of the following is a major objective of testing?
  - a) To prevent defects.
  - b) To validate the project plan works as required.
  - c) To gain confidence in the development team.
  - d) To make release decisions for the system under test.

- Select ONE option.

Q#3

- Which of the following is an example of a failure in a car cruise control system?
  - a) The developer of the system forgot to rename variables after a cut-and-paste operation.
  - b) Unnecessary code that sounds an alarm when reversing was included in the system.
  - c) The system stops maintaining a set speed when the radio volume is increased or decreased.
  - d) The design specification for the system wrongly states speeds in km/h.

- Select ONE option.

Q#4

- Which of the following is a defect rather than a root cause in a fitness tracker?
  - a) Because he was unfamiliar with the domain of fitness training, the author of the requirements wrongly assumed that users wanted heartbeat in beats per hour.
  - b) The tester of the smartphone interface had not been trained in state transition testing, so missed a major defect.
  - c) An incorrect configuration variable implemented for the GPS function could cause location problems during daylight saving times.
  - d) Because she had never worked on wearable devices before, the designer of the user interface misunderstood the effects of reflected sunlight.

- Select ONE option.

Q#5

- As a result of risk analysis, more testing is being directed to those areas of the system under test where initial testing found more defects than average.
- Which of the following testing principles is being applied?
  - a) Beware of the pesticide paradox.
  - b) Testing is context dependent.
  - c) Absence-of-errors is a fallacy.
  - d) Defects cluster together.
- Select ONE option.

Q#6

- Given the following test activities and tasks:
  - A. Test design
  - B. Test implementation
  - C. Test execution
  - D. Test completion
  - 1. Entering change requests for open defect reports
  - 2. Identifying test data to support the test cases
  - 3. Prioritizing test procedures and creating test data
  - 4. Analyzing discrepancies to determine their cause
- Which of the following BEST matches the activities with the tasks?
  - a) A-2, B-3, C-4, D-1
  - b) A-2, B-1, C-3, D-4
  - c) A-3, B-2, C-4, D-1
  - d) A-3, B-2, C-1, D-4
- Select ONE option.

Q#7

- Which of the following BEST describes how value is added by maintaining traceability between the test basis and test artifacts?
  - a) Maintenance testing can be fully automated based on changes to the initial requirements.
  - b) It is possible to determine if a new test case has increased coverage of the requirements.
  - c) Test managers can identify which testers found the highest severity defects.
  - d) Areas that may be impacted by side-effects of a change can be targeted by confirmation testing.

- Select ONE option.

Q#8

- Which of the following qualities is MORE likely to be found in a tester's mindset rather than in a developer's?
  - a) Experience on which to base their efforts.
  - b) Ability to see what might go wrong.
  - c) Good communication with team members.
  - d) Attention to detail.

- Select ONE option.

Q#9

- Given the following statements about the relationships between software development activities and test activities in the software development lifecycle:
  - 1. Each development activity should have a corresponding testing activity.
  - 2. Reviewing should start as soon as final versions of documents become available.
  - 3. The design and implementation of tests should start during the corresponding development. activity
  - 4. Testing activities should start in the early stages of the software development lifecycle.
- Which of the following CORRECTLY shows which are true and false?
  - a) True – 1, 2; False – 3, 4
  - b) True – 2, 3; False – 1, 2
  - c) True – 1, 2, 4; False – 3
  - d) True – 1, 4; False – 2, 3
- Select ONE option.

Q#10

- Given that the testing being performed has the following attributes:
  - based on interface specifications;
  - focused on finding failures in communication;
  - the test approach uses both functional and structural test types.
- Which of the following test levels is MOST likely being performed?
  - a) Component integration testing.
  - b) Acceptance testing.
  - c) System testing.
  - d) Component testing.
- Select ONE option.

Q#11

- Which of the following statements about test types and test levels is CORRECT?
  - a) Functional and non-functional testing can be performed at system and acceptance test levels, while white-box testing is restricted to component and integration testing.
  - b) Functional testing can be performed at any test level, while white-box testing is restricted to component testing.
  - c) It is possible to perform functional, non-functional and white-box testing at any test level.
  - d) Functional and non-functional testing can be performed at any test level, while Whitebox testing is restricted to component and integration testing.

- Select ONE option.

Q#12

- Which of the following statements BEST compares the purposes of confirmation testing and regression testing?
  - a) The purpose of regression testing is to ensure that all previously run tests still work correctly, while the purpose of confirmation testing is to ensure that any fixes made to one part of the system have not adversely affected other parts.
  - b) The purpose of confirmation testing is to check that a previously found defect has been fixed, while the purpose of regression testing is to ensure that no other parts of the system have been adversely affected by the fix.
  - c) The purpose of regression testing is to ensure that any changes to one part of the system have not caused another part to fail, while the purpose of confirmation testing is to check that all previously run tests still provide the same results as before.
  - d) The purpose of confirmation testing is to confirm that changes to the system were made successfully, while the purpose of regression testing is to run tests that previously failed to ensure that they now work correctly.
- Select ONE option.

Q#13

- Which of the following statements CORRECTLY describes a role of impact analysis in Maintenance Testing?
  - a) Impact analysis is used when deciding if a fix to a maintained system is worthwhile.
  - b) Impact analysis is used to identify how data should be migrated into the maintained system.
  - c) Impact analysis is used to decide which hot fixes are of most value to the user.
  - d) Impact analysis is used to determine the effectiveness of new maintenance test cases.

- Select ONE option.

Q#14

- Which of the following statements CORRECTLY reflects the value of static testing?
  - a) By introducing reviews, we have found that both the quality of specifications and the time required for development and testing have increased.
  - b) Using static testing means we have better control and cheaper defect management due to the ease of removing defects later in the lifecycle.
  - c) Now that we require the use of static analysis, missed requirements have decreased and communication between testers and developers has improved.
  - d) Since we started using static analysis, we -find coding defects that might have not been found by performing only dynamic testing.

- Select ONE option.

Q#15

- Which of the following sequences BEST shows the main activities of the work product review process?
  - a) Initiate review – Reviewer selection – Individual review – Issue communication and analysis – Rework
  - b) Planning & preparation – Overview meeting – Individual review – Fix-Report
  - c) Preparation – Issue Detection – Issue communication and analysis – Rework – Report
  - d) Plan – Initiate review – Individual review – Issue communication and analysis – Fix defects & report

- Select ONE option.

Q#16

- Which of the following CORRECTLY matches the roles and responsibilities in a formal review?
  - a) Manager – Decides on the execution of reviews
  - b) Review Leader - Ensures effective running of review meetings
  - c) Scribe – Fixes defects in the work product under review
  - d) Moderator – Monitors ongoing cost-effectiveness

- Select ONE option.

Q#17

- The reviews being used in your organization have the following attributes:
  - ▫ There is a role of a scribe
  - ▫ The purpose is to detect potential defects
  - ▫ The review meeting is led by the author
  - ▫ Reviewers find potential defects by individual review
  - ▫ A review report is produced
- Which of the following review types is MOST likely being used?
  - a) Informal Review
  - b) Walkthrough
  - c) Technical Review
  - d) Inspection
- Select ONE option.

Q#18

- You have been asked to take part in a checklist-based review of the following excerpt from the requirements specification for a library system:

Librarians can:	Borrowers can:	All users (librarians
1. Register new borrowers.	7. Borrow a maximum of 3 books at one time.	13. Can search for books by ISBN, author, or title.
2. Return books from borrowers.	8. View the history of books they have borrowed/reserved.	14. Can browse the system catalogue.
3. Accept fines from borrowers.	9. Be fined for failing to return a book within 3 weeks.	15. The system shall respond to user requests within 3 seconds.
4. Add new books to the system with their ISBN, author and title.	10. Get system responses within 3 seconds.	16. The user interface shall
5. Remove books from the system.	11. Borrow a book at no cost for a maximum of 4 weeks.	
6. Get system responses within 5 seconds.	12. Reserve books (if they are on-loan).	

- You have been assigned the checklist entry that requires you to review the specification for inconsistencies between individual requirements (i.e. conflicts between requirements).
- Which of the following CORRECTLY identifies inconsistencies between pairs of requirements?
  - a) 6-10, 6-15, 7-12
  - b) 6-15, 9-11
  - c) 6-10, 6-15, 9-11
  - d) 6-15, 7-12
- Select ONE option.

Q#19

- Which of the following provides the BEST description of exploratory testing?
  - a) A testing practice in which an in-depth investigation of the background of the test object is used to identify potential weaknesses that are examined by test cases.
  - b) An approach to testing whereby the testers dynamically design and execute tests based on their knowledge, exploration of the test item and the results of previous tests.
  - c) An approach to test design in which test activities are planned as uninterrupted sessions of test analysis and design, often used in conjunction with checklist-based testing.
  - d) Testing based on the tester's experience, knowledge and intuition.
- Select ONE option.

Q#20

- Which of the following BEST matches the descriptions with the different categories of test techniques?
  - 1. Coverage is measured based on a selected structure of the test object.
  - 2. The processing within the test object is checked.
  - 3. Tests are based on defects' likelihood and their distribution.
  - 4. Deviations from the requirements are checked.
  - 5. User stories are used as the test basis.

  - Black - Black-box test techniques
  - White - White-box test techniques
  - Experience - Experience-based test techniques

  - a) Black – 4, 5 White – 1, 2 Experience – 3
  - b) Black – 3 White – 1, 2 Experience – 4, 5
  - c) Black – 4 White – 1, 2 Experience – 3, 5
  - d) Black – 1, 3, 5 White – 2 Experience – 4

Select ONE option.

Q#21

- A fitness app measures the number of steps that are walked each day and provides feedback to encourage the user to keep fit.
- The feedback for different numbers of steps should be:
  - Up to 1000 - Couch Potato!
  - Above 1000, up to 2000 - Lazy Bones!
  - Above 2000, up to 4000 - Getting There!
  - Above 4000, up to 6000 - Not Bad!
  - Above 6000 - Way to Go!
- Which of the following sets of test inputs would achieve the highest equivalence partition coverage?
  - a) 0, 1000, 2000, 3000, 4000
  - b) 1000, 2001, 4000, 4001, 6000
  - c) 123, 2345, 3456, 4567, 5678
  - d) 666, 999, 2222, 5555, 6666
- Select ONE option

Q#22

- A daily radiation recorder for plants produces a sunshine score based on a combination of the number of hours a plant is exposed to the sun (below 3 hours, 3 to 6 hours or above 6 hours) and the average intensity of the sunshine (very low, low, medium, high).
- Given the following test cases:

	Hours	Intensity	Score
• T1	1.5	v. low	10
• T2	7.0	medium	60
• T3	0.5	v. low	10

- What is the minimum number of additional test cases that are needed to ensure full coverage of all valid INPUT equivalence partitions?
  - a) 1
  - b) 2
  - c) 3
  - d) 4
- Select ONE option.

Q#23

- A smart home app measures the average temperature in the house over the previous week and provides feedback to the occupants on their environmental-friendliness based on this temperature.
- The feedback for different average temperature ranges (to the nearest °C) should be:
  - Up to 10°C - Icy Cool!
  - 11°C to 15°C - Chilled Out!
  - 16°C to 19°C - Cool Man!
  - 20°C to 22°C - Too Warm!
  - Above 22°C - Hot & Sweaty!
- Using two-point BVA, which of the following sets of test inputs provides the highest level of boundary coverage?
  - a) 0°C, 11°C, 20°C, 22°C, 23°C
  - b) 9°C, 15°C, 19°C, 23°C, 100°C
  - c) 10°C, 16°C, 19°C, 22°C, 23°C
  - d) 14°C, 15°C, 18°C, 19°C, 21°C, 22°C
- Select ONE option

Q#24

- Decision table testing is being performed on a speeding fine system. Two test cases have already been generated for rules 1 and 4, which are shown below:

	Rules	R1	R4
Conditions	Speed>50	T	F
	School zone	T	F
Actions	$250 Fine	F	F
	Jail	T	F

- Given the following additional test cases:

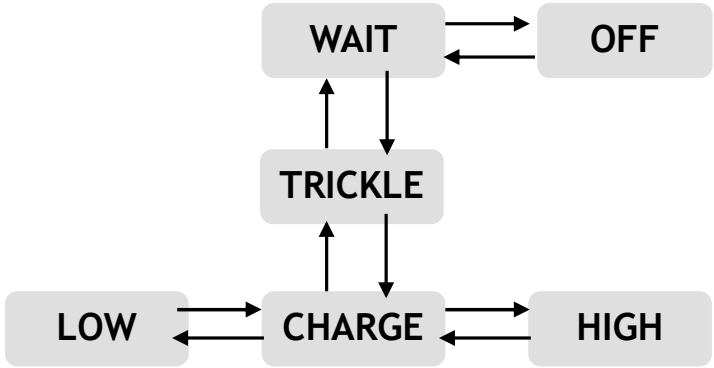
	Rules	DT1	DT2	DT3	DT4
Input	Speed	55	44	66	77
	School Zone	T	T	T	F
Expected Result	$250 Fine	F	F	F	T
	Jail	T	F	T	F

- Which two of the additional test cases would achieve full coverage of the complete decision table (when combined with the test cases that have already been generated for rules 1 and 4)?
  - a) DT1, DT2
  - b) DT2, DT3
  - c) DT2, DT4
  - d) DT3, DT4

Select ONE option.

Q#25

- Given the following state model of a battery charger software:
- Which of the following sequences of transitions provides the highest level of transition coverage for the model?



Wait -> Off
Off -> Wait
Wait -> Trickle
Trickle -> Wait
Trickle ->Charge
Charge ->Trickle
Low->Charge
Charge->Low
Charge->High
High->Charge

a) OFF → WAIT → OFF → WAIT → TRICKLE → CHARGE → HIGH → CHARGE → LOW

b) WAIT → TRICKLE → WAIT → OFF → WAIT →TRICKLE → CHARGE → LOW → CHARGE

c) HIGH → CHARGE → LOW → CHARGE → TRICKLE → WAIT → TRICKLE → WAIT → TRICKLE

d) WAIT → TRICKLE → CHARGE → HIGH →CHARGE →TRICKLE → WAIT → OFF → WAIT

Q#26

- Which of the following statements BEST describes how test cases are derived from a use case?
  - a) Test cases are created to exercise defined basic, exceptional and error behaviors performed by the system under test in collaboration with actors.
  - b) Test cases are derived by identifying the components included in the use case and creating integration tests that exercise the interactions of these components.
  - c) Test cases are generated by analyzing the interactions of the actors with the system to ensure the user interfaces are easy to use.
  - d) Test cases are derived to exercise each of the decision points in the business process flows of the use case, to achieve 100% decision coverage of these flows.
- Select ONE option.

Q#27

- Which of the following descriptions of statement coverage is CORRECT?
  - a) Statement coverage is a measure of the number of lines of source code (minus comments) exercised by tests.
  - b) Statement coverage is a measure of the proportion of executable statements in the source code exercised by tests.
  - c) Statement coverage is a measure of the percentage of lines of source code exercised by tests.
  - d) Statement coverage is a measure of the number of executable statements in the source code exercised by tests.

- Select ONE option.

Q#28

- Which of the following descriptions of decision coverage is CORRECT?
  - a) Decision coverage is a measure of the percentage of possible paths through the source code exercised by tests.
  - b) Decision coverage is a measure of the percentage of business flows through the component exercised by tests.
  - c) Decision coverage is a measure of the 'if' statements in the code that are exercised with both the true and false outcomes.
  - d) Decision coverage is a measure of the proportion of decision outcomes in the source code exercised by tests.

- Select ONE option.

Q#29

- Which of the following BEST describes the concept behind error guessing?
  - a) Error guessing requires you to imagine you are the user of the test object and guess mistakes the user could make interacting with it.
  - b) Error guessing involves using your personal experience of development and the mistakes you made as a developer.
  - c) Error guessing involves using your knowledge and experience of defects found in the past and typical mistakes made by developers.
  - d) Error guessing requires you to rapidly duplicate the development task to identify the sort of mistakes a developer might make.

- Select ONE option.

Q#30

- Which of the following BEST explains a benefit of independent testing?
  - a) The use of an independent test team allows project management to assign responsibility for the quality of the final deliverable to the test team, so ensuring everyone is aware that quality is the test team's overall responsibility.
  - b) If a test team external to the organization can be afforded, then there are distinct benefits in terms of this external team not being so easily swayed by the delivery concerns of project management and the need to meet strict delivery deadlines.
  - c) An independent test team can work totally separately from the developers, need not be distracted with changing project requirements, and can restrict communication with the developers to defect reporting through the defect management system.
  - d) When specifications contain ambiguities and inconsistencies, assumptions are made on their interpretation, and an independent tester can be useful in questioning those assumptions and the interpretation made by the developer.
- Select ONE option.

Q#31

- Which of the following tasks is MOST LIKELY to be performed by the test manager?
  - a) Write test summary reports based on the information gathered during testing.
  - b) Review tests developed by others.
  - c) Create the detailed test execution schedule.
  - d) Analyze, review, and assess requirements, specifications and models for testability.

- Select ONE option.

Q#32

- Given the following examples of entry and exit criteria:
  - 1. The original testing budget of $30,000 plus contingency of $7,000 has been spent.
  - 2. 96% of planned tests for the drawing package have been executed and the remaining tests are now out of scope.
  - 3. The trading performance test environment has been designed, set-up and verified.
  - 4. Current status is no outstanding critical defects and two high-priority ones.
  - 5. The autopilot design specifications have been reviewed and reworked.
  - 6. The tax rate calculation component has passed unit testing.
- Which of the following BEST categorizes them as entry and exit criteria:
  - a) Entry criteria – 5, 6 Exit criteria – 1, 2, 3, 4
  - b) Entry criteria – 2, 3, 6 Exit criteria – 1, 4, 5
  - c) Entry criteria – 1, 3 Exit criteria – 2, 4, 5, 6
  - d) Entry criteria – 3, 5, 6 Exit criteria – 1, 2, 4
- Select ONE option.

Q#33

- Given the following priorities and dependencies for these test cases:

Test case	Priority	Technical dependency on:	Logical dependency on:
TC1	High	TC4	
TC2	Low		
TC3	High		TC4
TC4	Medium		
TC5	Low		TC2
TC6	Medium	TC5	

- Which of the following test execution schedules BEST considers the priorities and technical and logical dependencies?
  - a) TC1 – TC3 – TC4 – TC6 – TC2 – TC5
  - b) TC4 – TC3 – TC1 – TC2 – TC5 – TC6
  - c) TC4 – TC1 – TC3 – TC5 – TC6 – TC2
  - d) TC4 – TC2 – TC5 – TC1 – TC3 – TC6
- Select ONE option.

Q#34

- Which of the following statements about test estimation approaches is CORRECT?
  - a) With the metrics-based approach, the estimate is based on test measures from the project and so this estimate is only available after the testing starts.
  - b) With the expert-based approach, a group of expert users identified by the client recommends the necessary testing budget.
  - c) With the expert-based approach, the test managers responsible for the different testing activities predict the expected testing effort.
  - d) With the metrics-based approach, an average of the testing costs recorded from several past projects is used as the testing budget.

- Select ONE option.

Q#35

- Which of the following BEST defines risk level?
  - a) Risk level is calculated by adding together the probabilities of all problem situations and the financial harm that results from them.
  - b) Risk level is estimated by multiplying the likelihood of a threat to the system by the chance that the threat will occur and will result in financial damage
  - c) Risk level is determined by a combination of the probability of an undesirable event and the expected impact of that event.
  - d) Risk level is the sum of all potential hazards to a system multiplied by the sum of all potential losses from that system.

- Select ONE option.

Q#36

- Which of the following is MOST likely to be an example of a PRODUCT risk?
  - a) The expected security features may not be supported by the system architecture.
  - b) The developers may not have time to fix all the defects found by the test team.
  - c) The test cases may not provide full coverage of the specified requirements.
  - d) The performance test environment may not be ready before the system is due for delivery.

- Select ONE option.

Q#37

- Which of the following is LEAST likely to be an example of product risk analysis CORRECTLY influencing the testing?
  - a) The potential impact of security flaws has been identified as being particularly high, so security testing has been prioritized ahead of some other testing activities.
  - b) Testing has found the quality of the network module to be higher than expected, so additional testing will now be performed in that area.
  - c) The users had problems with the user interface of the previous system, so additional usability testing is planned for the replacement system.
  - d) The time needed to load web pages is crucial to the success of the new website, so an expert in performance testing has been employed for this project.
- Select ONE option.

Q#38

- You are performing system testing of a train booking system and have found that occasionally the system reports that there are no available trains when you believe that there should be, based on the test cases you have run. You have provided the development manager with a summary of the defect and the version of the system you are testing. The developers recognize the urgency of the defect and are now waiting for you to provide more details so that they can fix it.
- Given the following pieces of information:
  - 1. Degree of impact (severity) of the defect.
  - 2. Identification of the test item.
  - 3. Details of the test environment.
  - 4. Urgency/priority to fix.
  - 5. Actual results.
  - 6. Reference to test case specification.
- Apart from the description of the defect, which includes a database dump and screenshots, which of the pieces of information would be MOST useful to include in the initial defect report?
  - a) 1, 2, 6
  - b) 1, 4, 5, 6
  - c) 2, 3, 4, 5
  - d) 3, 5, 6
- Select ONE option.

Q#39

- Given the following test activities and test tools:
  - 1. Performance measurement and dynamic analysis.
  - 2. Test execution and logging.
  - 3. Management of testing and testware.
  - 4. Test design.

  - A. Requirements coverage tools.
  - B. Dynamic analysis tools.
  - C. Test data preparation tools.
  - D. Defect management tools.
- Which of the following BEST matches the activities and tools?
  - a) 1 – B, 2 – C, 3 – D, 4 – A
  - b) 1 – B, 2 – A, 3 – C, 4 – D
  - c) 1 – B, 2 – A, 3 – D, 4 – C
  - d) 1 – A, 2 – B, 3 – D, 4 – C
- Select ONE option.

Q#40

- Which of the following is MOST likely to be used as a reason for using a pilot project to introduce a tool into an organization?
  - a) The need to evaluate how the tool fits with existing processes and practices and determining what would need to change.
  - b) The need to evaluate the test automation skills and training, mentoring and coaching needs of the testers who will use the tool.
  - c) The need to evaluate whether the tool provides the required functionality and does not duplicate existing test tools.
  - d) The need to evaluate the tool vendor in terms of the training and other support they provide.
- Select ONE option.