VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

www.vsb.cz

Software Quality – part: Testing

David Ježek

VŠB – Technical University of Ostrava

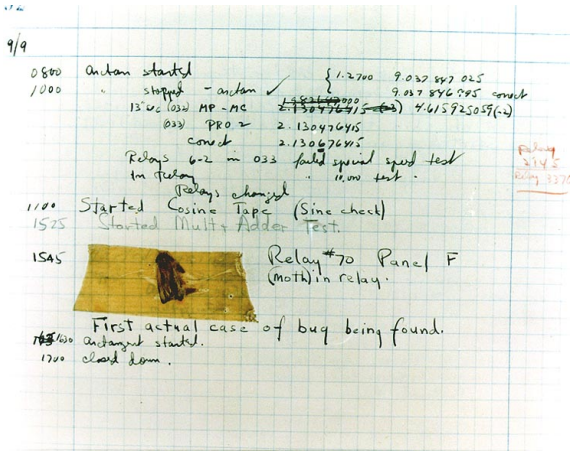
Faculty of Electrical Engineering and Computer Science

Department of Computer Science

History

The First "Computer Bug"

- Moth found trapped between points at Relay # 70, Panel F, of the Mark II



The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.

While Grace Hopper was working on the Harvard Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. Though the term *computer bug* cannot be definitively attributed to Admiral Hopper, she did bring the term into popularity. The remains of the moth can be found in the group's log book at the Smithsonian Institution's National Museum of American History in Washington, D.C..[1]

1.1-What is Software Testing (2)

A) Testing is the demonstration that errors are NOT preset in the program?

B) Testing shows that the program performs its intended functions correctly?

C) Testing is the process of demonstrating that a program does what is supposed to do?

D) Testing is the process of executing a program with the intent of finding errors.

Testing vs. Quality Assurance

- **Testing -** The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. (ISTQB)
- **Quality Assurance -** Activities focused on providing confidence that quality requirements will be fulfilled. (ISTQB)

- Demonstration that errors are NOT present:

If our goal is to demonstrate that a program has no errors, then we will subconsciously be steered toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors. The latter approach will add more value to the program than the former.

- Testing shows that the program performs its intended functions correctly:

- Testing is the process of demonstrating that a program does what is supposed to do:

Programs that do what they are supposed to do still can contain errors. That is, an error is clearly present if a program does not do what it is supposed to do, but errors are also present *if a program does what it is not supposed to do.*

Program testing is more properly viewed as the destructive process of trying to find the errors (whose presence is assumed) in a program. A successful test case is one that furthers progress in this direction by causing the program to fail. Of course, you eventually want to use program testing to establish some degree of confidence that a program does what it is supposed to do and does not do what it is not supposed to do, but this purpose is best achieved by a diligent exploration for errors.

1.1-What is Software Testing (2)

A) Testing is the demonstration that errors are NOT preset in the program?

B) Testing shows that the program performs its intended functions correctly?

C) Testing is the process of demonstrating that a program does what is supposed to do?

D) Testing is the process of executing a program with the intent of finding errors.

> ***Testing is the process of executing a program with the intent of finding errors.***
>
> Glenford J. Myers

Testing vs. Quality Assurance

- **Testing -** The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. (ISTQB)
- **Quality Assurance -** Activities focused on providing confidence that quality requirements will be fulfilled. (ISTQB)

- Demonstration that errors are NOT present:

If our goal is to demonstrate that a program has no errors, then we will subconsciously be steered toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors. The latter approach will add more value to the program than the former.

- Testing shows that the program performs its intended functions correctly:

- Testing is the process of demonstrating that a program does what is supposed to do:

Programs that do what they are supposed to do still can contain errors. That is, an error is clearly present if a program does not do what it is supposed to do, but errors are also present *if a program does what it is not supposed to do.*

Program testing is more properly viewed as the destructive process of trying to find the errors (whose presence is assumed) in a program. A successful test case is one that furthers progress in this direction by causing the program to fail. Of course, you eventually want to use program testing to establish some degree of confidence that a program does what it is supposed to do and does not do what it is not supposed to do, but this purpose is best achieved by a diligent exploration for errors.

What ca be tested

- From testing user requirements to monitoring the system in operation
- From testing the functionality to checking all other aspects of software:
  - Documents (specifications)
  - Design (model)
  - Code
  - Code+platform
  - Production, acceptance
  - Usage, business process

- From testing user requirements to monitoring the system in operation:

Testing is not done only once (e.g. with the first version of the product), but it is an continuous activity throughout product's entire lifecycle (from user requirements, through system design and implementation, to monitoring the system in operation and its maintenance). Testing it most effective in early phases of the development.

- From testing the functionality to checking all other aspects of software:

Testing is not focusing only to the system functionality but to all other attributes of the software:

- Documents (specifications)
- Design (model)
- Code
- Code+platform
- Production, acceptance
- Usage, business process
- Verification:

Its goal is to answer the question: "Have we done the system correctly?" Verification uses a previous development step (i.e. functional specification prior to coding) as the reference. A piece of code that fulfils its specification is verified.

- Validation:

Its goal is to check whether correct product has been built, i.e. whether it fulfils the customers needs. Thus, any step in the development process can be validated against user requirements.

The goal of testing may be verification or validation.

Realities in Software Testing

- Testing can show the presence of errors but cannot show the absence of errors (Dijkstra)
- All defects can not be found
- Testing does not create quality software or remove defects
- Building without faults means – among other – testing very early
- Perfect development process is impossible, except in theory
- Perfect requirements: cognitive impossibility

- Testing can show the presence of errors but cannot show the absence of errors:

There are still some errors never found in the software.

- All defects can not be found:

Even for simple programs/applications, the number of possible input combination or possible paths through the program is so large that all cannot be checked.

- Testing does not create quality software or remove defects:

It is the responsibility of development.

- Building without faults means – among other – testing very early:

A popular “argument” against testing is: “We should build correctly from the very beginning instead of looking for faults when all is ready”. Sure. But “correctly from the very beginning” means among other things thorough checking very early and all the time in the development process. Inspections of requirements specifications and design documents may to some extent replace the system test and acceptance test, but that does not mean “development without test”!

- Perfect development process is impossible, except in theory:

In practice, the way from concept to ready product cannot be guaranteed to be error-free (inaccurate requirements specifications, cognitive errors, organizational errors). Therefore the need to test the final product, regardless how perfect development is.

- Perfect requirements: cognitive impossibility:

Validation of requirements – are they what we really want? – is a kind of testing. But it is often impossible to define all requirements correctly in advance. Testing of the first version of a product is often a kind of additional requirements engineering: “is it what is really needed?

1.2-Testing Terminology

- Not generally accepted set of terms
- ISEB follows British Standards BS 7925-1 and BS 7925-2
  - http://www.testingstandards.co.uk/bs_7925-1.htm
  - http://www.testingstandards.co.uk/bs_7925-2.htm
- ISO/IEC/IEEE 29119 Software Testing (1-5)
- Replace:
  - IEEE 829 Test Documentation
  - IEEE 1008 Unit Testing
  - BS 7925-1 Vocabulary of Terms in Software Testing
  - BS 7925-2 Software Component Testing Standard
- ISTQB Glossary https://www.istqb.org/downloads/glossary.html

- Not generally accepted set of terms:

Different experts, tools vendors, companies, and countries use different terminologies (sometimes very exotic). These problems arise very obviously, e.g. after merge or acquisition of more companies.

- ISEB follows British Standards BS 7925-1 and BS 7925-2:

BS are owned by British Standards Institution (BSI). These two standards were developed by British Computer Society (BCS), Specialist Interest Group In Software Testing (SIGIST) in 1998.

- Other standards in software testing provide partial terminologies:
- QA standards ISO series 9000, 10000, 12000, 15000

Why Terminology?

- Poor communication
- Example: component – module – unit – basic – design – developer,… testing
- There is no ”good” and ”bad” terminology, only undefined and defined
- Difficult to describe processes
- Difficult to describe status

- Poor communication:

If every test manager puts different meaning to each term, he/she spends lot of time on defining what is what.

- Example: component – module – unit – basic – design – developer, ... testing:

Not only names differ but their precise meaning as well, which makes mapping difficult. Still worse if the same word means two completely different things, like “component” (either module, unit or “an independent component for component-based development”).

- There is no ”good” and ”bad” terminology, only undefined and defined:

Some people readily argue about the “right” names for things but almost any defined, standardized and generally accepted terminology is almost always better than a

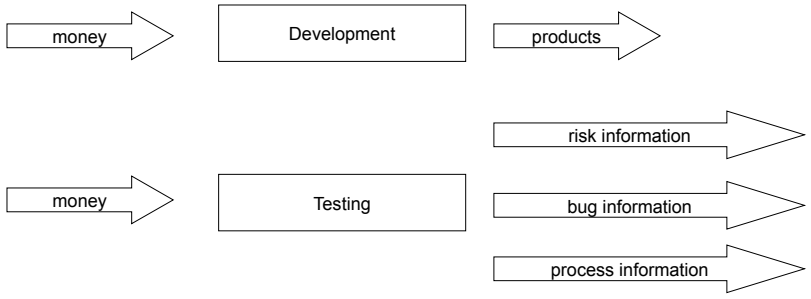
1.3-Why Testing is Necessary (2)

- In *Development*, "*money*" (investment) result in *products*, that can be sold any yield revenue.
- In *Testing*, it's unclear from business perspective how "*money*" (investments) result in anything of value.
- *Testing* produces three kind of outputs:
- *risk information*: probability that the product will fail in operation – this information is necessary for better delivery decisions
- *bug information*: input to development to enable them to remove those bugs (and, possibly, to the customer to let them avoid the bugs)
- *process information*: statistics and other metrics allow to evaluate processes and organization and identify faults in them

Unless there are *customers* for these outputs (managers willing to base their delivery decisions on test results, developers ready to fix defects found in testing, and process owners or projects managers ready to analyze and improve their processes), testing does not produce anything of value.

In other words, high-level testing in low-level environment does not add any immediate value, except as an agent of organizational change.

1.3-Why Testing is Necessary (2)

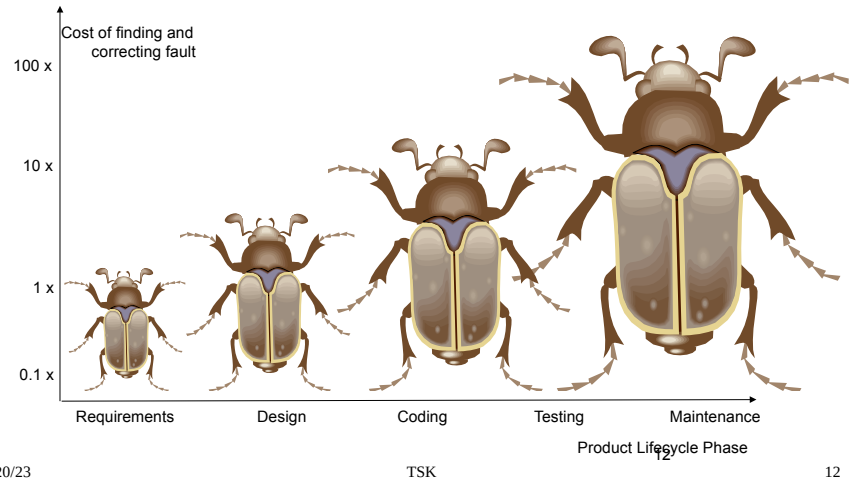


- In *Development*, "*money*" (investment) result in *products*, that can be sold any yield revenue.
- In *Testing*, it's unclear from business perspective how "*money*" (investments) result in anything of value.
- *Testing* produces three kind of outputs:
- *risk information*: probability that the product will fail in operation – this information is necessary for better delivery decisions
- *bug information*: input to development to enable them to remove those bugs (and, possibly, to the customer to let them avoid the bugs)
- *process information*: statistics and other metrics allow to evaluate processes and organization and identify faults in them

Unless there are *customers* for these outputs (managers willing to base their delivery decisions on test results, developers ready to fix defects found in testing, and process owners or projects managers ready to analyze and improve their processes), testing does not produce anything of value.

In other words, high-level testing in low-level environment does not add any immediate value, except as an agent of organizational change.

1.3-Why Testing is Necessary (3)



The cost of discovering, localizing, correcting and removing a fault is often hundreds or thousands of times higher in ready product than it is in the early states of development.
The cost of re-testing, regression testing, and updating or replacing the faulty software multiply very quickly after release (especially in mass production).

Test is most effective in early phases:
Contrary to traditional approach, test need not wait until development is ready. Test – reviews, inspections and other verification techniques for documentation and models - is actually the most effective in very early stages of any development process.

Testing and Quality

- Test measures and supports quality
- Testing is a part of Quality Assurance
- Many qualities:
  - Functional quality (traditional focus)
  - Non-functional quality (e.g. Performance)
  - Quality attributes (maintainability, reusability, testability, ...)
  - Usability for all stackeholders (vendor, retail merchant, operator, end-user, ...)
- Test techniques: tools to measure quality efficiently and effectively
- Test management: how to organize this

- Test *measures* and *supports* quality:

Test has two goals: to *measure* and *visualize* (the level of quality becomes known) quality and to support achieving quality by identifying product and process faults.

- Testing is a part of Quality Assurance:

The goal is to achieve planned and known quality, not to test. If it could be achieved without testing, test would disappear. The goal for testing is therefore to minimize the volume of testing without compromising quality or to achieve as good (reliable) quality measurement as possible with given resources and time, not to “test as much as possible”.

- Many qualities:
- *Functional quality*: the system does what the user required.
- *Non-functional quality*: these aspects (e.g. performance) are growing in importance. Cannot be reliably engineered without extensive testing.
- *Quality attributes:* there are other attributes (e.g. maintainability, reusability, testability, …) that must be check by testing too.
- *Usability for all stakeholders:* “Usability” is not only important but multidimensional. What is comfortable for the operation may be uncomfortable for the end-user. Therefore growing need for measuring and quality assessment in this area.
- Test techniques: tools and methods to measure quality efficiently and effectively:

Test theory contains knowledge how to test *efficiently* (so that desired levels of quality and test reliability are achieved) and effectively (so that it is done as cheaply as possible).

- Test management: how to organize this:

Test management has much in common with general project and process management.

Complexity

- Software – and its environment – are too complex to exhaustively test their behavior
- Software can be embedded
- Software has human users
- Software is part of the organization's workflow

- Software is too complex to exhaustively test its behaviour:

Even for relatively simple programs/applications, the number of possible input combinations or possible paths through the program is so large that all cannot be checked. Then testing is necessary as a king of art of predicting under uncertainty, choosing the few tests we can afford to run that give us best confidence in program's future correct behavior.

- Software environment is too complex to test it exhaustively:

A simple piece of code can be run on different PC-machines, OS (and their versions), with different printers, on different browsers. The number of combination easily becomes huge.

- Software can be embedded:

The testing products means testing SW, HW and "mechanics" around. Again, complexity. Again, methods required to make the best of this mess.

- Software has often human users:

For most applications, the behavior (and needs) of the users cannot be fully predicted by the engineering means only. Testing (acceptance, usability) helps to tackle this aspect.

- Software is part of the organization's workflow:

Engineering considerations are *not* the only important considerations for many software products. Any useful knowledge about product quality is a combination of engineering quality and the product's quality contribution during organizational or marketing usage.

Exhaustive Testing

- Exhaustive testing is impossible
- Even in theory, exhaustive testing is wasteful because it does not prioritize tests
- Contractual requirements on testing
- Non-negligent practice important from legal point of view

- Exhaustive testing is impossible:

Even for modest-sized applications with few inputs and outputs, the number of test cases quickly becomes huge.

- Contractual requirements on testing:

The contract between the vendor and the customer may contain clauses on the required amount of testing, acceptable reliability levels, or even on specific test techniques or test coverage measures.

- Non-negligent practice important from the legal point of view:

If you ever get sued by your customer, his or her lawyers will sure try the trick of accusing you of negligence because your testing was not “exhaustive”. As defense, the impossibility of exhaustive testing should be

How much testing?

- This is a risk-based, business decision
  - Test completion criteria
  - Test prioritization criteria
  - Decision strategy for the delivery
  - Test manager presents products quality
- Test is never ready
- The answer is seldom ”more testing” but rather ”better testing”, see the completion criteria:
  - All test cases executed
  - All test cases passed
  - No unresolved (serious) incident reports
  - Pre-defined coverage achieved
  - Required reliability (MTBF) achieved
  - Estimated number of remaining faults low enough

- This is a risk-based, business decision:
- Test completion criteria – must be specified in advance in test plan
- Test prioritization criteria - scales on which to compare test cases' relative importance (severity, urgency, probability, visibility, business criticality, etc.)
- Decision strategy for the delivery – must be specified in advance (what shall happen if test completion criteria are not fulfilled)
- Test manager presents products quality - he/she is responsible for the estimation and presentation of product quality but the business decision based on this data is made by responsible manager (project manager, project owner, product owner, etc.).
- Test is never ready:

As exhaustive testing is not possible, we can always test a little more, and there is always some justification for it in (the diminishing) probability that more faults will be found. Unless completion criteria are established and test cases prioritized, the probability of finding more faults cannot be reliably estimated

- The answer is seldom “more testing” but rather “better testing”:

Testing must be based on the combination of completion criteria:

- All test cases executed
- All test cases passed
- No unresolved (serious) incident reports
- Pre-defined coverage achieved
- Required reliability (MTBF) achieved
- Estimated number of remaining faults low enough

Risk-Based Testing

-
- Testing finds faults, which – when faults have been removed – decreases the risk of failure in operation
- Risk-based testing

- Testing finds faults, which decreases the risk of failure in operation

Testing can be based on any criteria, but the most important is the risk of failure in operation as this is the most obvious indication of quality software.

- Risk-based testing
- The chosen amount and quality of testing shall be based on how much risk is acceptable
- Test design (choosing *what* to test) shall be based on the involved risks
- The order of testing shall be chosen according to the risks
- Error: the ”mistake” (human, process or machine) that introduces a fault into software:
- *Human mistake*: users forget a need. Requirements engineer misinterprets users’ need. Designer makes a logical mistake. Programmer makes a coding mistake.
- *Process mistake*: requirements not uniquely identifiable, no routines for coping with changing/new requirements, not enough time to perform design inspections, poor control over programmers’ activities, poor motivation, …
- *Machine mistake*: incorrect compiler results, lost files, measurement instruments not precise enough…
- Fault: “bug” or “defect”, a faulty piece of code or HW:

Wrong code or missing code, incorrect addressing logic in HW, insufficient bandwidth of a bus or a communication link.

- Failure: when faulty code is executed, ti may lead to incorrect results (i.e. to failure):

A faulty piece of code calculates an incorrect result, which is given to the user. A faulty SW or HW “crashes” the system. A faulty system introduces longer delays than allowed during heavy load.

When a failure occurs during tests, the fault may by identified and corrected.
When a failure occurs in operation, it is a (small or large) catastrophe.

Base terms connected with “error”

- **Error**: the ”mistake” (human, process or machine) that introduces a fault into software
- **Fault**: ”bug” or ”defect”, a faulty piece of code or HW
- **Failure**: when faulty code is executed, ti may lead to incorrect results (i.e. to failure)

- Testing finds faults, which decreases the risk of failure in operation

Testing can be based on any criteria, but the most important is the risk of failure in operation as this is the most obvious indication of quality software.

- Risk-based testing
  - The chosen amount and quality of testing shall be based on how much risk is acceptable
  - Test design (choosing *what* to test) shall be based on the involved risks
  - The order of testing shall be chosen according to the risks
- Error: the ”mistake” (human, process or machine) that introduces a fault into software:
  - *Human mistake*: users forget a need. Requirements engineer misinterprets users’ need. Designer makes a logical mistake. Programmer makes a coding mistake.
  - *Process mistake*: requirements not uniquely identifiable, no routines for coping with changing/new requirements, not enough time to perform design inspections, poor control over programmers’ activities, poor motivation, …
  - *Machine mistake*: incorrect compiler results, lost files, measurement instruments not precise enough…
- Fault: “bug” or “defect”, a faulty piece of code or HW:

Wrong code or missing code, incorrect addressing logic in HW, insufficient bandwidth of a bus or a communication link.

- Failure: when faulty code is executed, ti may lead to incorrect results (i.e. to failure):

A faulty piece of code calculates an incorrect result, which is given to the user. A faulty SW or HW “crashes” the system. A faulty system introduces longer delays than allowed during heavy load.

When a failure occurs during tests, the fault may by identified and corrected.
When a failure occurs in operation, it is a (small or large) catastrophe.

Cost of Failure

- Reliability: the probability of no failure
- Famous: American Airlines, Ariane 5 rocket, Heathrow Terminal 5
- Quality of life
- Safety-critical systems
- Embedded systems
- Usability requirements for embedded systems and Web applications

- Reliability: the probability of no failure:

The probability that software will not cause the failure of a system for a specific time under specified conditions.

- Famous: American Airlines, Ariane 5 rocket, Heathrow Terminal 5:

The financial cost can be shocking, many billions of dollars. As compared to the estimated cost of additional testing that would probably have discovered the fault (a few hundred thousand dollars).

- *American Airlines*: new booking system Sabre (1988) with complex mathematical algorithms for optimization of the numbers of business class and economy class passengers. It had a fault, which resulted in approximately 4 passengers fewer on every flight. $50 million in lost revenue after a few months' operation were the first indication there was a fault at all!
- *Ariane 5 rocket*: an unmanned Ariane 5 rocket (1996) exploded just forty seconds after its lift-off from Kourou. 10 years of development costing $7 billion, the rocket itself and its cargo were valued at $500 million. 64-bit floating point number relating to the horizontal velocity with respect to the platform was converted to a 16 bit signed integer which overflowed. Could easily be found if tested.
- *Heathrow Terminal 5*: 300 flights were cancelled during the first five days as "teething problems" at the new Terminal 5 caused chaos (2008). It went about a combination of factors. Some were technical, involving glitches with the sophisticated new baggage set-up. But other issues were more mundane. Employees arriving for work, for example, could not find their way to the staff car park. Testing of new terminal took 6 months and 15,000 volunteers were called to help test out facilities. The trials had been designed using lessons learned from the security and baggage delays faced by passengers at other terminals over the past few months.
- Quality of life:

As anyone using a PC realizes, failures need not be catastrophes to sharply reduce the joy of living.

- Safety-critical systems:

More and more safety-critical systems contain software – the necessity of high safety and reliability grows. The cost of failure is injury or human life (railway, aircraft, medical systems). For many safety-critical systems the important attribute is usability (low usability can cause "operator mistake" or "human factor" in an accident, coming usually from confusing or unusable information, especially in stress situations).

- Embedded systems

Embedded systems (whether safety-critical or not), require high-quality software, because of the difficulty (or impossibility) of updates. Remember the cost of software errors in some mobile phones.

- Usability requirements for embedded systems and Web applications:

Embedded systems and Web applications are mass consumer products, where customers require easy usage. Failure to provide it results in lost revenues or market shares, which is a novel experience for software industry, used more to putting requirements on customers than the other way round!

Test Process Definition

- Test Planning
- Test Specification
- Test Execution
- Test Recording & Evaluation
- Completion Criteria

- Test process as part of development or production process

Test is a part of QA, and test process should not be defined separately, but should be seen in the context of overall development process.

- Large companies have own process definitions

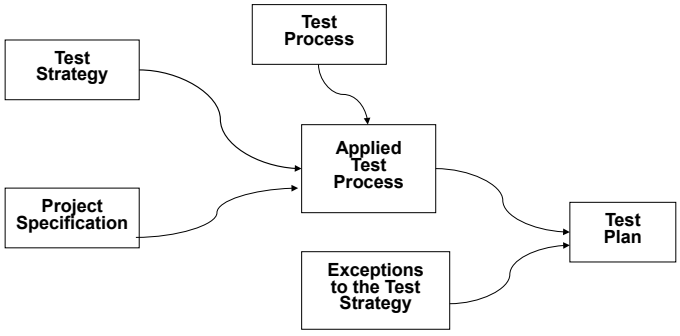
Most development and production companies have own test processes. There can be of course similarities (though used terminologies are often strikingly different), but nevetheless many thousands different test processes exist in industrial reality.

- "COTS" test process

COTS (Commercial Off The Shelf)

It is possible to buy a test processes. The most known vendor today is probably IBM Rational with its RUP – Rational Unified Process (test

Test Planning



A company's Test Strategy together with its Test Process (defined in the organization) are adopted to the current project based on a Project Specification. This results into an Applied Test Process, i.e. an overall vision "how we will test this time". This vision is the implemented (described in detail) in a Test Plan. Often, Test Plan is a document written in a natural language.

The process of creating a Test Plan is test planning. The is mostly done very early during project. Later, the processes of test *estimation*, *monitoring* and *control* may lead to changes in the test plan.

Test Plan's Goal

- High-level test plan and more detailed test plans
- Related to project plan
- Follows QA plan
- Configuration management, requirements, incident management

Even the best test plan will not work unless it is synchronized with other areas, project and technical.

- High-level test plan and more detailed test plans

Depending on project size and complexity, test plan can sensibly be divided into one high-level test plan and some detailed test plans. The division can follow test area or test level, or specific aspects of testing.

- Project plan

Test plan must be inspected for correctness and compliance with overall project plan. Sometimes (in small projects) the test plan is the part of a project plan.

- Follows the QA plan

Hopefully, the function and contents of a test

Test Specification

The complete documentation of the test design, test cases and test procedures for a specific test item. (ISTQB)

- Test specification defines what to test
- Test specification is part of testware
- Basic building blocks of test specifications are test cases
- Test specification – instruction – script
- Test specification – requirements
- Test specification - reporting

- Test specification defines *what* to test

Test specification are repositories of test cases. They should be free from organizational issues, which belong to the test plan(s).

- Test specification is part of testware

Testware – test cases, test scripts, test data, etc. is often under CM control (manages either by test tool or by a separate tool).

- Basic building blocks of test specifications are test cases

Test cases are generated when applying test design techniques. They shell be general and repeatable.

- Test specification – instruction – script

Test cases need not contain all detailed information on *how* to perform them. This information may be put into a separate description, sometimes called *test instructions* (this approach is not practical because of maintenance difficulties).

If test execution is automated, then the instructions for a test tool are called *test script (test program)*. Test script can replace test case (instructions).

- Test specification – requirements

It is desirable that for every test case, there is a link to the requirements behind it and for every requirement, there are links to all test cases that verify it. This is very hard to achieve and maintain without using test tools (test management tools, e.g. Test Manager).

- Test specification – reporting

The test specification must support logging and reporting during and after test execution, mainly through the *identification* of test cases and their steps. This can be easily automated by using test tools (test running tools, e.g. Robot)

Labs

Test Case

- Unique name/title
- Unique ID
- Description
- Preconditions / prerequisites
- Actions (steps)
- Expected results

- Unique name/title

Short test case title enhances readability of the specification and test reports – descriptive unique name of the test case.

- Unique ID

Identification of the test case. All test cases should follow an identical, defined format. This ID must be permanent (adding or removing test cases shell not change ID) – cryptic unique identification of the test case.

- Description

Brief description explaining what functionality the case case covers.

- Preconditions / prerequisites

Exact description of required system state prior the execution of the test case.

- Actions (steps)

Labs

1.4 - Fundamental Test Processs - terms (ISTQB)

- **test case** - A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
- **test case result** - The final verdict on the execution of a test and its outcomes, such as pass, fail, or error. The result of error is used for situations where it is not clear whether the problem is in the test object.
- **test case specification** - A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item.
- **test specification** - A document that consists of a test design specification, test case specification and/or test procedure specification.
- **test script** - Commonly used to refer to a test procedure specification, especially an automated one.
- **test procedure specification** - A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.
- **Test Design Specification** - A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high-level test cases.

- Unique name/title

Short test case title enhances readability of the specification and test reports – descriptive unique name of the test case.

- Unique ID

Identification of the test case. All test cases should follow an identical, defined format. This ID must be permanent (adding or removing test cases shell not change ID) – cryptic unique identification of the test case.

- Description

Brief description explaining what functionality the case case covers.

- Preconditions / prerequisites

Exact description of required system state prior the execution of the test case.

- Actions (steps)

Test Execution

- Manual
- Automated
- Test sequence
- Test environment
- Test data

- Manual

Tester follows the description from the test case and performs step by step all specified actions. Prone to errors, boring (monkey testing) and time-consuming. It is recommended that the author of the test case performs it first.

- Automated

Test tool executes test case according to predefined instructions (test scripts or test program). The automation scope can include any of / all of the following:

- Preparation (set-up to fulfill preconditions)
- Execution
- Result evaluation (comparing actual and expected results)
- Clean-up (putting system back into some known state)
- Test sequence

Sometimes it is not practical to execute each test case separately but it is better to put test cases into a sequence, e.g.:

- Insert new record
- Search existing record
- Modify existing record
- Delete existing record
- Test environment

There are more environments used for developing, testing and maintaining software applications (DEV – development, IT – functional and performance testing, QA – acceptance testing, PROD – production). Configuration files of test environment as part of testware are under CM control.

- Test data

Test data are various input and output files (for expected and actual results) that must be managed properly as part of testware. If test data are taken from the production, they must be degraded.

Test Recording & Evaluation

- Recording actual outcomes and comparison against expected outcomes
- Off-line test result evaluation
- Test log
- Test report
- Recording test coverage
- Incident management

- Recording actual outcomes and comparison against expected outcomes

Manual testing: If actual and expected outcomes match, then test case passed. If not, then test case failed, actual outcomes are recorded and incident (defect) is created and assigned to development.
Automated testing: Comparison is done automatically, everything is recorded and even incidents are created.

- Off-line test result evaluation

Sometimes the immediate result (pass/fail) is impossible (too fast execution to allow on-line evaluation by a lower analysis tool or the final result is available only after some other tests have been performed, etc.), so during test execution the results are gathered for the evaluation, which is done later.

- Test log

It is a log of "all" (*relevant* and *important*) what happened during test execution. This activity (log creation) is best to automate, as it is repetitive, boring and requires exactness. It is used for (1) off-line evaluation,(2) failure analysis and debugging and (3) for archiving and future reference.

- Test report

It is a summary of the results of all executed test cases. Must contain as well complete information on configuration and versions of test environment, testware and test object. Some test tools are capable to produce test report.

- Recording test coverage

If test cases are mapped to requirements, test coverage can be easily derived. When executing test cases, the results are projected into requirements with the information how much functionality was successfully tested.

- Incident management

Answer the following questions:
1. Was this really a failure?
2. What presumably caused this failure?
3. How to assign correction responsibility?

Incident must be repeatable – put enough information to the incident report to enable reproducing the incident by the developer who is fixing it.

Test Completion

- Test completion criteria must be specified in advance
- Decision strategy for the release/delivery decision must be specified in advance
- Test manager is responsible for the estimation and presentation of the product quality, not for release/delivery decision
  - Run TC
  - Passed TC
  - Failed TC
  - Executed TC
  - Failure intensity
  - Number of incident reports
  - Estimation of product quality
  - Reliability of this estimation
  - Projected estimation of product quality

- Test completion criteria must be specified in advance

In the test plan or similar document.

- Decision strategy for the release/delivery decision must be specified in advance

What shall happen if test completion criteria are not fulfilled, but deadlines are approaching and there is strong pressure to release? The strategy for making this decision should be defined in advance.

- Test manager is responsible for the estimation and presentation of the product quality, not for release/delivery decision

It is the responsibility of test manager to preset to project management accurate and up-to-date data on:

1. Number of percentage of run test cases
2. Number and percentage of passed tests
3. Number and percentage of failed tests
4. Trends in test execution (cumulative number of executed test cases)
5. Trends in failure intensity
6. Similar data on the number of incident reports, their status and trends
7. Estimation of product quality based on the data available
8. Reliability (level of significance) of this estimation
9. Projected estimation of product quality and test reliability for various scenarios

Completion Criteria

- All test cases executed
- All test cases passed
- No unresolved incident reports
- No unresolved serious incident reports
- Number of faults found
- Pre-defined coverage achieved
  - Code coverage
  - Functional coverage
  - Requirements coverage
  - If not, design more test cases
- Required reliability (MTBF) achieved
- Estimated number of remaining faults low enough

- All test cases executed

It is a sensible criterion, provided good quality, coverage and reliability of those tests (otherwise the less test cases we have, the easier to achieve completion).

- All test cases passed

The previous criterion plus additionally that there must be no failed tests – strong requirement not achievable in practice.

- No unresolved incident reports

It may be the same as the previous one but not necessarily: some incident reports may be postponed, rejected (e.g. caused by faults of test environment or testware, etc.).

- No unresolved serious incident reports

The previous criterion might be too strong – we can divide incident reports according to severity (e.g. 1 and 2 must be resolved).

- Number of faults found

Generally a useless criterion, as it is the estimated number of *remaining* faults that matter. The assumption is that many found faults means few remaining (this can be wrong – many found faults may mean many remaining).

- Pre-defined coverage achieved

Generally better that "all tests… no incidents…" family, because they address the issue of achieved test quality/reliability as well:

  - Code coverage: there is a number of different code coverage measures that tell what proportion of tested code have been exercised by executed tests.
  - Functional coverage: even very high code coverage does not guarantee that "all" (paths, user scenarios) has been tested. Therefore, it should be complemented by some kind of functional coverage.
  - Requirements coverage: all code and all functions may have been tested, but in order to discover *missing* functionality, tests should cover all requirements.

- Required reliability (MTBF) achieved

This can only be calculated if statistical testing is used (MTBF – Mean Time Between Failures).

- Estimated number of remaining faults low enough

Based on the number and frequency of faults discovered so far during testing, an estimation of the number of remaining faults can be made.

1.5-Re-Testing and Regression Testing

Definitions

- Re-testing: re-running of test cases that caused failures during previous executions, after the (supposed) cause of failure (i.e. fault) has been fixed, to ensure that it really has been removed successfully
- Regression testing: re-running of test cases that did NOT cause failures during previous execution(s), to ensure that they still do not fail for a new system version or configuration
- Debugging: the process of identifying the cause of failure; finding and removing the fault that caused failure

- Definition of Re-testing (BS 7925-1)

Running a test more than once.

- Definition of Regression Testing (BS 7925-1)

Re-testing to a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

- Definition of Debugging (BS 7925-1)

The process of finding and removing the causes of failures in software (don't mix with testing). Debugging is NOT part of testing, but has many aspects in common with testing. During debugging, test cases may be re-run in order to study failure more in detail. Re-running of test cases during debugging is NOT re-test or regression testing. Additional "debugging test cases" may be created and

Re-testing

- Re-running of the test case that caused failure previously
- Triggered by delivery and incident report status
- Running of a new test case if the fault was previously exposed by chance
- Testing for similar or related faults

- Re-running of the test case that caused failure previously

A test case has caused a test object to fail. The fault that (supposedly) caused this failure has been discovered and removed (fixed). The very same test case is executed on the new (corrected) version of the system to ensure that the fault has really been successfully fixed.

- Triggered by delivery and incident report status

Re-testing is normally done after the delivery of a fixed build and after the corresponding incident report has been put into a “fixed” (“corrected”, “re-test”, “put into build”, or similar name) status. Some kind of private re-test before formal release may be used as well.

- Running of a new test case if the fault was previously exposed by chance

When failure occurred by chance without any intentional test case being executed (e.g. by “smoke-test”, “sanity-check”, or “ad-hoc” testing), a new test case should be designed and added. Re-testing means then the execution of this new test case.

- Testing for similar or related faults

During re-testing, even test cases looking for *similar* faults may be executed. For example if a record deletion from a file caused failure, even other record deletion routines may be tested. Re-testing *related* faults is advisable too. For example if a record deletion method has been fixed, then other methods belonging to the same class can be re-tested after correcting the fault. This can be defined as “increased testing”, or new test design caused by faults already found.

Regression Testing

- Regression due to fixing the fault (side effects)
- Regression due to added new functionality
- Regression due to new platform
- Regression due to new configuration or after the customization
- Regression and delivery planning

- Regression due to fixing the fault (side effects)

On average, according to empirical data, 10-25% of all fixes actually introduce new faults, sometimes in areas seemingly “far away” (temporally, functionally or structurally) from the original fault. To be able to discover the new faults, test cases seemingly “far away” from the fixed fault must be executed on fixed builds.

- Regression due to added new functionality

Adding new functionality may introduce faults into already existing functionality, or expose faults existing previously, but not found. Therefore, old functionality must be tested again for releases with new functionality.

- Regression due to new platform

A system that executes correctly in one environment may fail in another environment, either due to hidden faults or interface faults. Therefore, regression testing may be required even when not a single software instruction has been changed.

- Regression due to new configuration or after the customization

Sometimes called “configuration testing”. For example, a Java script depends on HW, operating system and browser of the client machine. Including different versions of them, the number of possible combinations is very large, requiring impossibility large amount of regression testing. Special strategies are available to tackle this.

- Regression and delivery planning

To decrease the amount of regression testing, a regression test suite may be run *once* on a release with *many* fault corrections and new functionality added. If an incremental methodology is used (e.g. RUP), then some increments (usually the latest ones) are focusing only on bug fixing which means that only re-testing and regression testing is needed. Regression testing is often used in *maintenance* when emergency fixes and “extra” functionality is introduced.

Regression Schemes

- Less frequent deliveries
- Round-robin scheme
- Additional selection of test cases
- Statistical selection of test cases
- Parallel testing
- “Smoke-test” for emergency fixes
- Optimisation or regression suite:
  - General (understanding system, test case objectives, test coverage)
  - History (decreased regression for stable functionality)
  - Dependencies (related functionality)
  - Test-level co-ordination (avoiding redundant regression on many levels)

- Less frequent deliveries

If a regression test takes longer than the time between releases, decreasing the delivery frequency may be an option. If a number of fixes and functionality enhancements are delivered together, less frequent deliveries are possible without increasing the overall development time.

- Round-robin scheme

Example: A regression test suite has 300 test cases. It takes 1 day to execute 100 test cases. Releases come every day. Test cases no 1-100 are executed on release N, 101-200 on N+1, 201-300 on N+2, then again 1-100 on N+1, etc. Even if no release is fully regression tested, a relatively good measure of product quality is achieved.

- Additional selection of test cases

The regression test suite may be pruned to fit the available time. A selection of regression test cases may be used for most releases, while the complete test suite will be executed only before external releases, quality checkpoints, project milestones, etc.

- Statistical selection of test cases

Provided that the data on the probability distribution of user actions is available, test cases can be ordered according to their “importance”, i.e. the relative frequency of the user action that they test. In this way, even if the complete regression test suite is not executed, the reliability level can be estimated for releases.

- Parallel testing

By dividing test execution into a number of parallel tracks, that can execute independently and in parallel, calendar test execution time can be significantly decreased. This applies both to manual and to automated testing. The cost is that multiple amount of test equipment and of testers are required.

- “Smoke-test” for emergency fixes

Emergency fix – exceptional release that fixes one fault (or low number of faults) or sometimes introduces a new (small in scope) functionality and its delivery is urgently required. As changes in the system are relatively small, complete testing is not needed.

“Smoke-test” or “sanity-check” means execution of a subset of the most important test cases from the regression suite with the goal to check if there is not major problem in the system after the change. Even in the emerging situation, some kind of “smoke-test” must be performed.

- Optimisation of regression suite
  - *General* – basic test techniques can help choose test cases for regression test suites effectively. Required level of test coverage can be used to estimate the needed amount of regression testing. Good system understanding is required to identify and remove repetitive or less important lest cases. Redundant test cases can be removed.
  - *History* – regression test cases may become obsolete with time. Stable functionality where faults are no longer discovered during regression testing, need not be tested as extensively as new, unstable functionality, or as a system area with a history of many faults.
  - *Dependencies* – provided a well-designed system with clear-cut dependencies and interfaces, it is possible to minimize the amount of regression for areas that are not related and not connected to the area, where recent changes have occurred.
  - *Test-level co-ordination* – savings in regression test time can often be achieved by coordinating tests run on different levels, to avoid repetition.

Regression and Automation

- Regression test suites under CM control
- Incident tracking for test cases
- Automation pays best in regression
- Regression-driven test automation
- Incremental development

- Regression test suites under CM control

All test cases shell be archived and under version control to be able to return back to already not used test cases. Regression test cases are changing from release to release. This applies even more to automated regression testing which increases the amount of testware: test scripts, test programs, test data, test configurations, etc.

- Incident tracking for test cases

Test cases (especially test scripts, test programs, test data) can be faulty or changed for other reasons (e.g. effectiveness). These changes should be controlled and traceable like any software changes. The development and maintenance of testware should be handled like development and maintenance of any other software, i.e. planned, designed, under version management, etc.

- Automation pays best in regression

When test automation is considered, it shall be first of all applied to regression testing. The strategy for regression testing must therefore be known before the automation strategy is developed. Large amount of regression requires automation (the automation is effective starting from number of releases > 3). Performance testing cannot be done without tools (load generation, monitoring, performance measurement, etc.). These tools and test cases may therefore be candidates to be included in regression testing.

- Regression-drive test automation

Introducing test automation into projects must be planned according to the needs of the regression test strategy.

- Incremental development

New development methods (“incremental development”, “daily build”, “Rapid Application Development”, RUP, etc.) become increasingly popular. They are characterized by frequent deliveries, incremental functionality growth, and fast feedback from test to development. Therefore, they require heavy regression testing, which makes both test automation and other techniques for regression optimization especially important.

1.6-Expected Results

Why Necessary?

- Test = measuring quality = comparing actual outcome with expected outcome
- What about performance measurement?
- Results = outcomes; outcomes ≠ outputs
- Test case definition: preconditions – inputs – expected outcomes
- Results are part of testware – CM control

- Test = measuring quality = comparing actual outcome with *expected* outcome

Test is verifying whether something is correct or not – means by definition comparing two values: actual and expected. Random testing is (1) normally not really testing at all (2) or testing actual results against our vague and unspecified outcome expectations.

- What about performance measurement?

Performance measurement = benchmarking.
Performance requirements are notoriously vague or absent, but performance testing is thriving. Explanation? It is then either testing against informal, unspecified “random requirements” or a kind of requirement engineering (trying to find out what the requirements should be) by running ready product.

- Results = outcomes; outcomes ≠ outputs

Application outputs can be test case outcomes, but not all test cases outcomes are outputs – performance levels, state transitions, data modifications are possible test case outcomes which are not application outputs. In order to evaluate them, test environment must provide access to them: through special test outputs, debug tools, etc.

- Test case definition: preconditions – inputs – expected outcomes

When expected test result/outcome is missing, then it is NOT a test case specification at all. Unspecified or insufficiently specified expected outcomes make some failures harder to discover.

- Results are part of testware – CM control

Often, the expected outcome is a data file. Unless it can be incorporated in a test specification, it will require to be under separate CM control. Changing the expected outcome file will have the same effect as directly changing the test specification – a common baseline for them will therefore be required.

Types of Outcomes

- Outputs
- State transitions
- Data changes
- Simple and compound results
- “Long-time” results
- Quality attributes (time, size, etc.)
- Non-testable?
- Side-effects

- Outputs

They are most easily observable, therefore often utilized as outcomes/results. Outputs have very many forms: displayed or changed GUI objects, sent messages or signals, printouts, sounds, movements.

- State transitions

Does the system perform correct state transition for a given set of inputs? Outputs following transitions are often used to judge, but the new state is the expected outcome.

- Data changes

Has data changed correctly?

- Simple and compound results

Results may be simple (“Error message appears”) or compound (“new record put into database, index updated, display adjusted, message sent…”).

- “Long-time” results

For example, testing for long-time stability: system still works correctly after a week.

- Quality attributes (time, size, etc.)

Most non-functional requirements are of this kind.

- Non-testable?

1) Possibly valid requirements, but formulated in a non-testable way, e.g. “sufficient throughput to handle typical traffic”.
2) Valid, measurable requirements, which cannot be measured due to technical constraints.

- Side-effects

Implicitly, every test case has an invisible clause in expected outcome definition “the program does this… and nothing incorrect happens”. “Nothing incorrect” is easily implied, but impossible to verify.

Sources of Outcomes

Finding out or calculating correct expected outcomes/results is often more difficult than can be expected. It is a major task in preparing test cases.

- Requirements
- Oracle
- Specifications
- Existing systems
- Other similar systems
- Standards
- NOT code

- Requirements

Sufficiently detailed requirement specifications can be used directly as the source of expected test results. Most often however, requirements do not have sufficient quality.

- Oracle

According to BS 7925-1 it is “a mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test”; often a program, another similar application, etc.

- Specifications

Specifications other than requirement specification (e.g. design specification, use case specification, interface specification, function specification) are generally a good source of expected outcomes – verification means testing whether system works “according to specification”.

- Existing systems

Previous, verified versions of the same system can be used as oracle for getting correct expected results.

- Other similar systems

Any other software – commercial or not – that has already been sufficiently verified and implements part of the functionality of the tested system, often makes a good oracle.

- Standards

Many standards, e.g. in telecommunications, contain detailed specifications that can be used as expected test results. A good example of a test case suite built entirely around standard specification is Sun’s test suite for verification whether a JVM (Java Virtual Machine) conforms to Sun’s Java standard.

- NOT code

(nor the same specification if specification is the test object), Because anything compared to itself (the same source of expected and actual outcomes) will always give “correct” results.

Difficult Comparisons

- GUI
- Complex outcomes
- Absence of side-effects
- Timing aspects
- Unusual outputs (multimedia)
- Real-time and long-time difficulties
- Complex calculations
- Intelligent and “fuzzy” comparisons

- GUI

Notoriously difficult expected results. Prone to frequent changes, complex, often asynchronous. If treated on pixel level, often useless, require some kind of object approach. Most tools existing today to not cope well with moving or scrolling components.

- Complex outcomes

Actually, GUI outputs are one of them. Comparison may be difficult simple because the results are large and complex.

- Absence of side-effects

For most test cases, there are infinitely many possible outcomes that must not happen. For a test case “press key” with expected outcome “text <<key pressed>> appears” there are innumerable things that are expected NOT to happen: program does not crash, database is not deleted, no – say – blinking green triangle appears in the middle of the screen… etc. Verifying this is impossible, on the other hand some degree of observant caution is necessary.

- Timing aspects

Outcomes that either occur very quickly or last very short time, or are asynchronous, or occur after undefined delay may all be hard to verify correctly.

- Unusual outputs (multimedia)

Video sequences, complex graphics, sounds, smells, etc. are very hard to test.

- Real-time and long-time difficulties

(it is a sub-set of “absence of side effects”)

For real-time, multithread applications there may exist hidden faults that only cause failure when certain rare timing conditions are fulfilled. Such failures are not easily repeatable. During long-time execution a gradual “decay” of software may occur (stability testing aims at those problems). Typical example of such problems are memory-leaks.

- Complex calculations

Their results are hard to verify, may only “look right”. AA booking system fault 1988.

- Intelligent and “fuzzy” comparisons

Whenever correct result is not fully deterministic or analogue rather than discrete, it is difficult to verify.

1.7-Prioritization of Tests

Why Prioritize Test Cases?

- Decide importance and order (in time)
- "There is never enough time"
- Testing comes last and suffers for all other delays
- Prioritizing is hard to do right (multiple criteria with different weights)

- Decide importance and order (in time)

To prioritize test cases means to measure their importance on an ordinal scale, then plan test execution accordingly (typically, in descending order of importance, i.e. more important cases before less important).

- "There is never enough time"

Dedicated testers easily become paranoid – they suspect faults everywhere and want to verify every tiny detail. To balance this desire with business reality, we must choose what is most important to test, i.e. prioritize.

- Testing comes last and suffers for all other delays

The day for customer delivery is often holy, but development is nevertheless delayed. Planned test time is cut as a result, often with

Prioritization Criteria

- Severity (failure)
- Priority (urgency)
- Probability
- Visibility
- Requirement priorities
- Feedback to/from development
- Difficulty (test)
- What the customer wants
- Change proneness
- Error proneness
- Business criticality
- Complexity (test object)
- Difficult to correct

This is a tentative list of possible prioritization criteria (scales on which to compare test cases' relative importance). This list is not ordered (i.e. it gives no clue to which criteria are more important). The criteria are not independent nor exclusive. For operational usage, they must be defined more in details. Put them into columns are mark each test case with the level of importance:

- H – high
- M – medium
- L – low
- Severity (failure); the consequences of failure (in operation): 1 – fatal, 2 – serious, 3 – disturbing, 4 – tolerable, 5 – minor
- Priority (urgency): how important it is to test this particular function as soon as possible: 1 – immediately, 2 – high priority, 3 – normal queue, 4 – low priority
- Probability: the (estimated) probability of the existence of faults and failure in operation
- Visibility: if a failure occurs, how visible it is? (it relates to "severity")
- Requirement priorities: if requirements are prioritized, the same order shall apply to test cases
- Feedback to/from development: do the developers need test results to proceed? (similar to "priority"). Do the developers know a specific tricky area or function?
- Difficulty (test): is this test case difficult to do (resource- and time-consuming?)
- What the customer wants: ask the customer what he prefers (it relates to "requirements priorities")
- Change proneness: does this function change often?
- Error proneness: is it a new, badly designed, or well-knows "stinker" feature?
- Business criticality: related to "severity" and "what the customer wants"
- Complexity (test object): related to "error proneness"
- Difficult to correct: a fault known to be difficult to correct, may be given lower priority (provided severity is sufficiently low)

Prioritization Methods

- Random (the order specs happen)
- Experts' "gut feeling"
- Based on history with similar projects, products or customers
- Statistical Usage Testing
- Availability of: deliveries, tools, environment, domain experts…
- Traditional Risk Analysis
- Multidimensional Risk Analysis
  - analytic hierarchy process (AHP)

- Random (the order specs happen)

No method at all, but "the order test specs happen" may actually mirror both the "priority" and "business criticality" as well as "requirements prioritization" – the not so bad.

- Experts' "gut feeling"

Experts with testing, technical and domain (application) knowledge do the prioritization. Experts are good to have, but their "gut feeling" may often be misleading, unless structured methods (see below) are followed.

- Based on history with similar projects, products or customers

Documented data on previous fault history, priority, severity, etc. is used to prioritize test cases for current project/product according to some chosen criterion (or a chosen

2.1-Models for Testing

Verification, Validation and Testing

- **Verification**: The process of evaluation a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase – building the system right
- **Validation**: The determination of the correctness of the products of software development with respect to the user needs and requirements – building the right system
- **Testing**: The process of exercising software to verify that is satisfies specified requirements and to detect errors

Testing is not only test execution. Static analysis can be performed before the code has been written. Writing and designing test cases is also part of testing. Reviews of requirement specifications and models, and of any other documents, belong to testing as well.

IEEE standards

3.1.36 verification:

- (A) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- (B) The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly).

3.1.35 Validation:

- (A) The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
- (B) The process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions), and satisfy intended use and user needs.
- NOTE—For (A), see IEEE Std 610.12-1990 [B3].

- NOTE—For subdefinition (A), see IEEE Std 610.12-1990 [B3].

BS 7925-1

- **verification**: The process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase. [IEEE]
- **validation**: Determination of the correctness of the products of software development with respect to the user needs and requirements. [IEEE]

ISTQB Glosary

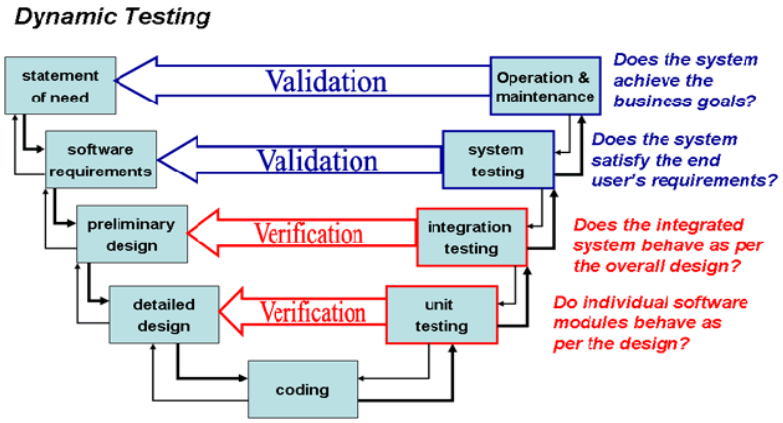
Verification Ref: ISO 9000

- Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.

Validation Ref: ISO 9000

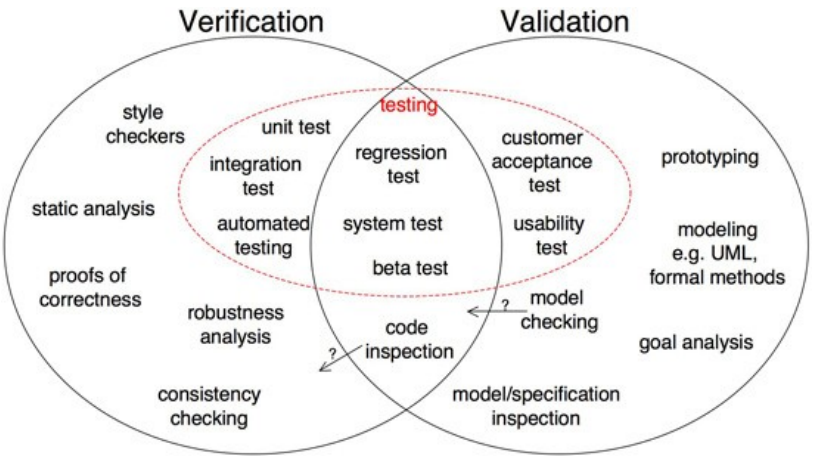
- Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

V&V – Where is truth?



http://www.chambers.com.au/glossary/verification_validation.php

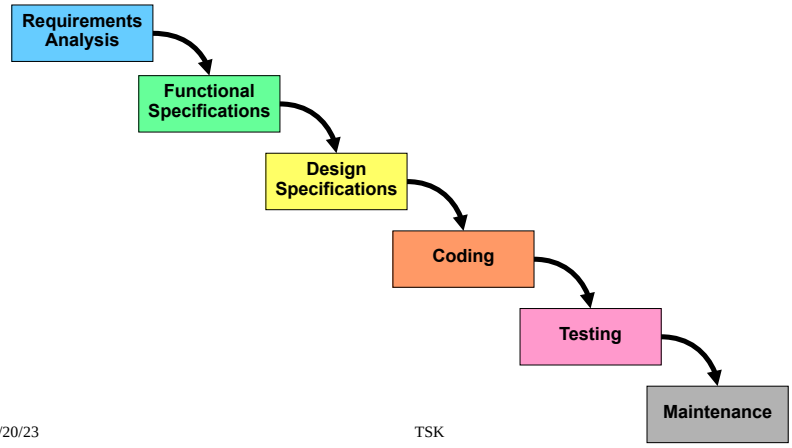
V&V – Where is truth?



http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

2.1-Models for Testing (2)

Waterfall Model



Requirements Analysis
During the requirements analysis phase, basic market research is performed and potential customer requirements are identified, evaluated, and refined. The result of this phase of the process is usually a marketing requirement or product concept specification. Requirements in the concept specification are usually stated in the customer's language.

Functional Specifications
Requirements in the concept specification are reviewed and analysed by software engineers in order to more fully develop and refine the requirements contained in the concept specification. Requirements from the concept specification must be restated in the software developer's language – the functional specification.

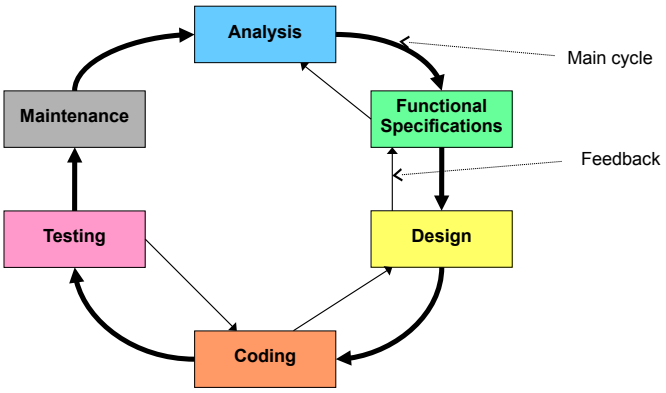
Design Specifications
Once the functional specifications are developed, software engineers should have a complete description of the requirements the software must implement. This enables software engineers to begin the design phase. It is during this phase that the overall software architecture is defined and the high-level and detailed design work is performed. This work is documented in the design specifications.

Coding
The information contained in the design specifications should be sufficient to begin to the coding phase. During this phase, the design is transformed or implemented in code. If the design specifications are complete, the coding phase proceeds smoothly, since all of the information needed by software engineers is contained in these specifications.

Testing
According to the waterfall model, the testing phase begins when the coding phase is completed. Tests are developed based on information contained in the functional and design specifications already in the coding phase. These tests determine if the software meets defined requirements. A software validation test plan defines the overall validation testing process. Individual test procedures (test cases, test scripts, test programs) are developed based on a logical breakdown of requirements. The results of the testing activities are usually documented in a software validation test report. Following the successful completion of software validation testing, the product may be shipped to customers.

Maintenance
Once the product is being shipped, the maintenance phase begins. This phase lasts until the support for the product is discontinued. Many of the same activities performed during the development phases are also performed during the maintenance phase.

Iterative



Requirements Analysis
During the requirements analysis phase, basic market research is performed and potential customer requirements are identified, evaluated, and refined. The result of this phase of the process is usually a marketing requirement or product concept specification. Requirements in the concept specification are usually stated in the customer's language.

Requirements Definition
Requirements in the concept specification are reviewed and analysed by software engineers in order to more fully develop and refine the requirements contained in the concept specification. Requirements from the concept specification must be restated in the software developer's language – the software requirements specification.

Design
Once the SRS is developed, software engineers should have a complete description of the requirements the software must implement. This enables software engineers to begin the design phase. It is during this phase that the overall software architecture is defined and the high-level and detailed design work is performed. This work is documented in the software design description.

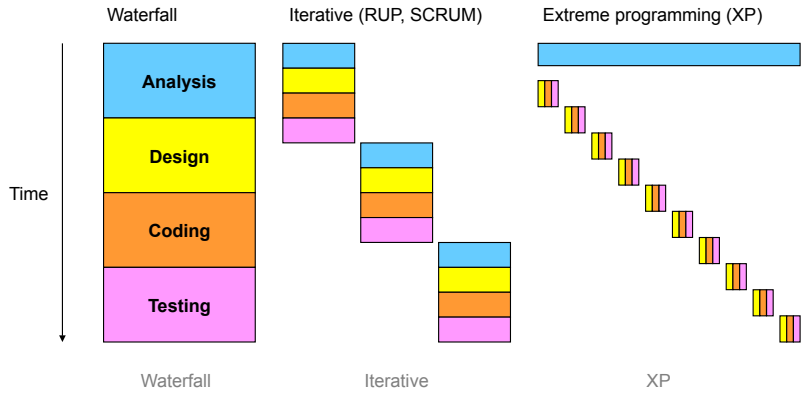
Coding
The information contained in the SDD should be sufficient to begin to the coding phase. During this phase, the design is transformed or implemented in code. If the SDD is complete, the coding phase proceeds smoothly, since all of the information needed by software engineers is contained in the SDD.

Testing
According to the waterfall model, the testing phase begins when the coding phase is completed. Tests are developed based on information contained in the SRS and the SDD already in the coding phase. These tests determine if the software meets defined requirements. A software validation test plan defines the overall validation testing process. Individual test procedures (test cases, test scripts, test programs) are developed based on a logical breakdown of requirements. The results of the testing activities are usually documented in a software validation test report. Following the successful completion of software validation testing, the product may be shipped to customers.

Maintenance
Once the product is being shipped, the maintenance phase begins. This phase lasts until the support for the product is discontinued. Many of the same activities performed during the development phases are also performed during the maintenance phase.

2.1-Models for Testing (4)

2.1-Models for Testing (4)



- *Waterfall*

There is no ideal model. Waterfall model is the right one in ideal world.
Analysis - I understand everything
Design - I design perfect solution with complete and right knowledge of customer and target platform
Coding - Design is coded without bugs
Testing – Well, why the hell test ideal system? Testing can be omitted…
Eureka!!! the system is accepted and it fulfills all stakeholder needs
but ideal does not exist in reality therefore waterfall model is out of touch with reality

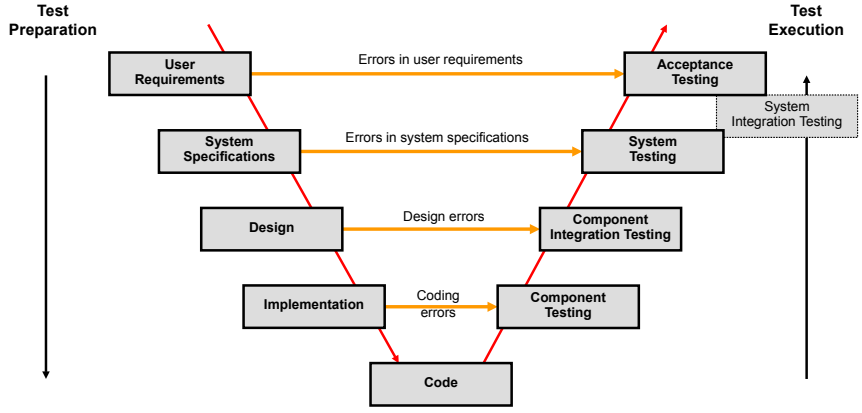
- *Iterative (RUP, SCRUM)*

The development is divided into iterations. In the first iteration we focus on a big picture. The project is split into small pieces (iterations), in which we deliver product to the customer to get customer feedback. Iterations are here to reduce time we are walking the wrong way (one iteration usually takes 2- 3 weeks). The iteration must not be changed during processing, all plans/bugs/etc must be planned for the next iteration. There must be no disturbance from the iteration plan - focus on the target. The iteration should end as planned and evaluated. Unfinished tasks together with bugs found in this iteration must be estimated again and planned for the beginning of the next iteration. Do not save bugs for later, unfixed bug means the work was not done. One or two iterations are planned just to remove bugs (no new functionality is implemented). In SCRUM terminology an iteration is called a Sprint.

- *Extreme programming (XP)*

It goes about agile software development methodology (rapid development), the set of daily practices that embody and encourage particular XP values: communication (simple design, common metaphors, collaboration of users and programmers, frequent verbal communication and feedback), simplicity (starting with the simplest solution), feedback (from the system by writing unit tests and running periodic integration tests, from the customer by acceptance testing, from the team by quick response to new requirements), courage (design and code for today and not for tomorrow – developers feel comfortable with refactoring their code when necessary) and respect between team members.

V-model: Levels of Testing



For each stage in the model there are deliverables to the next stage, both development and testing. Such a delivery is an example of a baseline.

For example, when the user requirements are ready, they are delivered both to the next development stage and to the corresponding test level, i.e. acceptance testing. The user requirements will be used as input to the system specification (where the system requirements will be the deliverable to the next stage) and the acceptance test design.

Note that this is a simplified model. In reality, the arrows should point in both directions since each stage naturally will find faults and give feedback to the previous stages.

Test Levels

- Component testing
- Component integration testing
- System testing (functional and non-functional)
- System integration testing
- Acceptance testing
- Maintenance testing

- The objectives are different for each test level (see the V-model)
- Test techniques used (black- or white- box)
- Object under test, e.g. component, grouped components, sub-system or complete system
- Responsibility for the test level, e.g. developer, development team, an independent test team or users
- The scope of testing

Component Testing

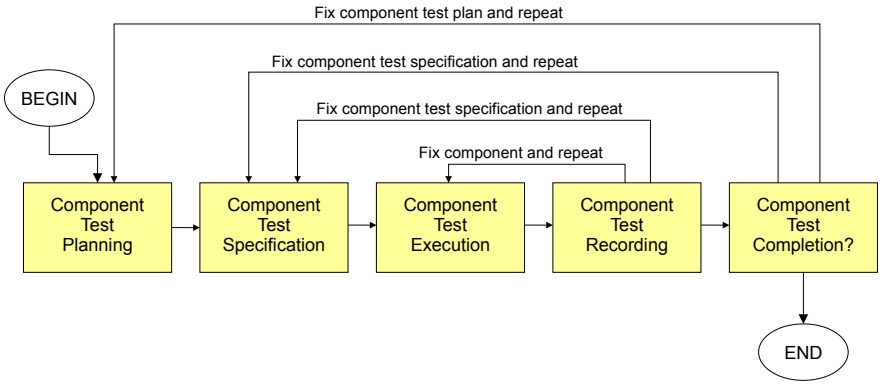
- Component Testing
- First possibility to execute anything
- Different names (Unit, Module, Basic, … Testing)
- Usually done by programmer
- Should have the highest level of detail of the testing activities

The objective of component (unit, module) testing is to find bugs in individual components (units, modules) by testing them in an isolated environments. Component testing is the first *dynamic* testing activity in the development life cycle. Traditionally (and most practically) component testing have been performed by programmers. One major reason for this is that component testing tends to require knowledge of the code which is why developers are well suited for this. Unfortunately component testing is often viewed more as a debugging activity than as a testing activity.

Mature ladder:

- Developers are checking their own modules with little or no documentation (they are blind

Component Test Process (BS 7925-2)



- Component Test Planning

The component test planning contains two phases. *In the first phase* the overall project test strategy (generic) and the project test plan (project specific) are defined. The project test strategy includes test case selection methods, documentation, entry and exit criteria as well as the component test process itself. The project test plan contains information of the scope of the project, the resources needed and how to apply the strategy in the current project. *The second phase* of the planning deals with components individually. For each component a separate component test plan is produced (to list the specific test case design techniques, the test measurement techniques, the tools including stubs and drivers and the test completion criteria that apply to the specific component).

- Component Test Specification

Component test specification is the activity of applying the test design techniques specified in the component test plan to the information in the design specification, producing a number of test cases. The test cases should be documented in the component test case specifications. Each test case should also have a unique name and contain enough detailed instructions on how to perform the test case and a reference to the requirement that is tested by that test case.

- Component Test Execution

During component test execution, the test cases are executed on the actual module, preferable in the priority order. However, things might happen during the execution of the test cases which may force deviations from the planned order of execution. This is normal and quite all right as long as the deviations are conscious choices.

- Component Test Recording

During the execution of test cases, test results are produced. Basically there are two types of results: *logs* and *pass/fail results*. A log is just a chronological list of events that took place during the execution. The second type of result is the result of comparing the actual and the expected output. After a fault is located it usually pays off to investigate where the fault was first introduced in the design process and it is a good practice to correct the fault in all documents that contain the fault. A component test report is the document which contains a summary of all results of the second type.

- Component Test Completion?

Based on the information in the component test reports, the specified exit criteria in the test strategy and/or the test plans, and the current time budget, the decision whether or not to continue testing, can be performed. Here there are also several options:

  - Enough coverage has been obtained and quality of test object is OK => component testing can be ended and the component delivered to the next level of testing (usually to component integration testing).
  - All test cases have been executed but enough coverage has not yet been achieved => more test cases have to be designed and executed to increase the coverage.
  - Time is out but the quality of the test object is too low => negotiate with project stakeholders to get more time to test and to correct faults (this is however typically NOT the responsibility of the test sub-project).

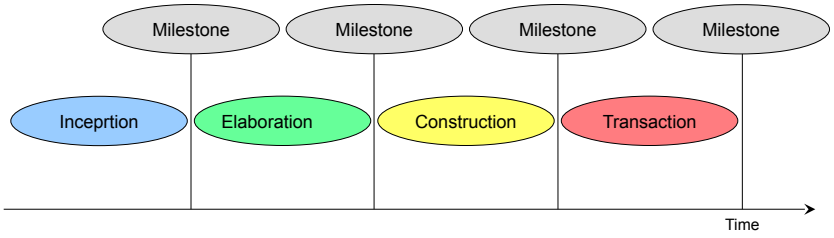
Component Testing Check List

- Algorithms and logic
- Data structures (global and local)
- Interfaces
- Independent paths
- Boundary conditions
- Error handling

- Algorithms and logic:
- - Have algorithms and logic been correctly implemented?
-
- Data structures (global and local):
- - Are global data structures used?
- - If so, what assumptions are made regarding global data?
- - Are these assumptions valid?
- - Is local data used?
- - Is the integrity of local data maintained during all steps of an algorithm's execution?
-
- Interfaces:
- - Does data from calling modules match what this module expects to receive?
- - Does data from called modules match what this module provides?
-
- Independent paths:
- - Are all independent paths through the modules identified and exercised?
-
- Boundary conditions:
- - Are the boundary conditions known and tested to ensure that the module operates properly at its boundaries?
-
- Error handling:
- - Are all error-handling paths exercised?
-

Rational Unified Process (RUP)

- Iterative development
- Requirement management
- Component base architecture
- Visual modeling
- Software Quality Verification
- Change Management



RUP is an example of object-oriented methodologies that emphasize the incremental, iterative, and concurrent nature of software development.

RUP is a product process developed by Rational Software Corporation that provides project teams with a guide to more effective use of the industry-standard Unified Modeling Language (UML). RUP also provides software-engineering best practices through templates, guidelines, and tools. Most of the tools are, as you might guess, also provided by Rational.

The RUP is based on four consecutive phases. The purpose of the *inception phase* is to establish the business case for the project. This is done by creating several high-level use case diagrams, defining success criteria, risk assessment, resource estimate, and an overall plan showing the four phases and their approximate time frames. Some deliverables the inception phase might include are:

- A vision statement
- An initial set of use cases
- An initial business case
- An initial risk assessment
- An initial project plan
- Prototypes

The purpose of the *elaboration phase* is to analyze the problem domain, establish the overall product architecture, eliminate the highest risks, and refine the project plan. Evolutionary prototypes are developed to mitigate risks and address technical issues and business concerns. Some key deliverables this phase might include are:

- A relatively complete use case model supplemented with text as appropriate
- Architecture description
- Revised risk assessment
- Revised project plan
- Initial development plan
- Initial user manual

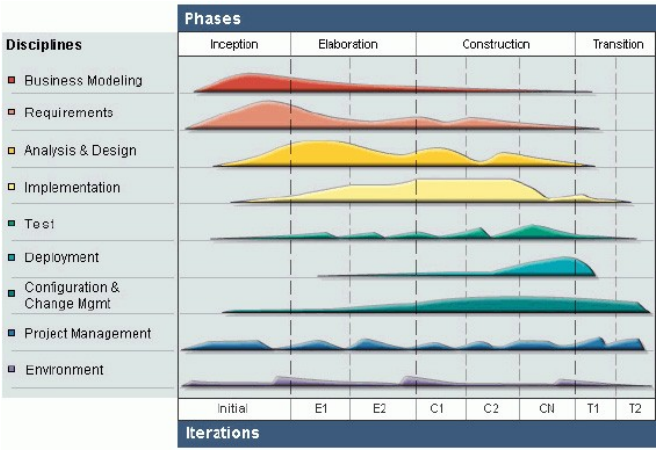
During the *construction phase*, the remaining components are developed, and thoroughly tested. Key deliverables from this phase include:

- Software product operating on target platform
- Revised user manual
- Complete description of current release

The purpose of the *transition phase* is to transition the product from development to the user community. Activities that would typically be performed include:

- Beta testing by users
- Conversion of existing information to new environment
- Training of users
- Product rollout

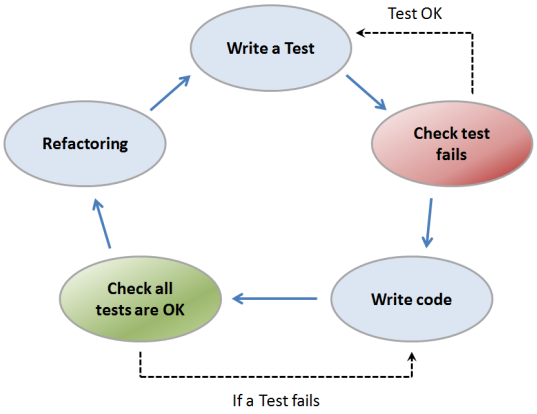
Rational Unified Process (RUP)



Test Driven Development

- TDD adopts a “Test-First” approach in which unit tests are written before code.
- This idea, which dates back to ancient times, was formalized in the mid-1990s by **Kent Beck**, who made it one of the pillars of the Extreme Programming (XP) methodology.
- TDD is a way of managing fear during programming.

Test Driven Development



1. Add a test
   In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.
2. Run all tests and see if the new test fails
   This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.
3. Write the code
   The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.
   At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. Run tests
   If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. Refactor code
   The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from

TDD – Clean Tests

- The test code is as important if not more important than the production code!
  - readability
  - simple, clear and as dense a test as possible
  - a unit test should represent only one concept and contain only one assertion

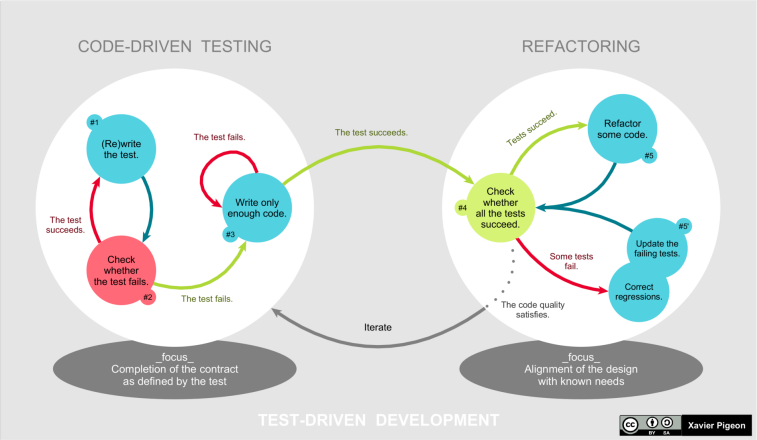
1. Add a test
   In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.
2. Run all tests and see if the new test fails
   This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.
3. Write the code
   The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.
   At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. Run tests
   If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. Refactor code
   The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from

TDD – Clean Tests

- 5 other rules that can be easily memorized using the acronym FIRST:
  - **Fast:** a test must be fast to be executed often.
  - **Independent:** tests must not depend on each other.
  - **Repeatable:** a test must be reproducible in any environment.
  - **Self-Validating:** a test must have a binary result (Failure or Success) for a quick and easy conclusion.
  - **Timely:** a test must be written at the appropriate time, i.e. just before the production code it will validate.

1. Add a test
   In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.
2. Run all tests and see if the new test fails
   This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.
3. Write the code
   The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.
   At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. Run tests
   If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. Refactor code
   The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from

Test Driven Development



1. Add a test
   In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.
2. Run all tests and see if the new test fails
   This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.
3. Write the code
   The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.
   At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. Run tests
   If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. Refactor code
   The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from

Test Driven Development

- **Test structure** - Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.
- **Setup:** Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- **Execution:** Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- **Validation:** Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT.
- **Cleanup:** Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one. [8]

Individual best practices states that one should

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test oracle focused on only the results necessary to validate its test.
- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.

Practices to avoid - "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.

Practices to avoid - "anti-patterns"

- Testing precise execution behavior timing or performance.
- Building "all-knowing oracles". An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.
- Testing implementation details.
- Slow running tests.

Test Driven Development

Myth

- You create a 100% regression test suite

Reality

Although this sounds like a good goal, and it is, it unfortunately isn't realistic for several reasons:

- I may have some reusable components/frameworks/... which I've downloaded or purchased which do not come with a test suite, nor perhaps even with source code. Although I can, and often do, create black-box tests which validate the interface of the component these tests won't completely validate the component.
- The user interface is really hard to test. Although user interface testing tools do in fact exist, not everyone owns them and sometimes they are difficult to use. A common strategy is to not automate user interface testing but instead to hope that user testing efforts cover this important aspect of your system. Not an ideal approach, but still a common one.
- Some developers on the team may not have adequate testing skills.
- Database regression testing is a fairly new concept and not yet well supported by tools.
- I may be working on a legacy system and may not yet have gotten around to writing the tests for some of the legacy functionality.

Test Driven Development

Myth

- You only need to unit test

Reality

- For all but the simplest systems this is completely false.
- The agile community is very clear about the need for a host of other testing techniques.

Test Driven Development

Myth

- TDD is sufficient for testing

Reality

- TDD, at the unit/developer test as well as at the customer test level, is only part of your overall testing efforts.
- At best it comprises your confirmatory testing efforts, but you must also be concerned about independent testing efforts which go beyond this.

Test Driven Development

Myth

- TDD doesn't scale

Reality

This is partly true, although easy to overcome. TDD scalability issues include:

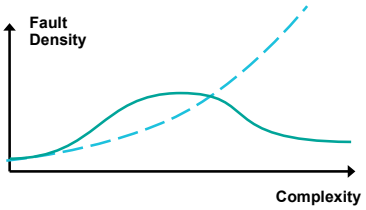
1) Your test suite takes too long to run. This is a common problem:
   - First, separate your test suite into two or more components. One test suite contains the tests for the new functionality that you're currently working on, the other test suite contains all tests. You run the first test suite regularly, migrating older tests for mature portions of your production code to the overall test suite as appropriate. The overall test suite is run in the background, often on a separate machine(s), and/or at night.
   - Several levels of test suite -- development sandbox tests which run in 5 minutes or less, project integration tests which run in a few hours or less, a test suite that runs in many hours or even several days that is run less often.
2) Not all developers know how to test.
   - That's often true, so get them some appropriate training and get them pairing with people with unit testing skills.
3) Everyone might not be taking a TDD approach.
   - Taking a TDD approach to development is something that everyone on the team needs to agree to do.
     - they either need to start
     - they need to be motivated to leave the team
     - team should give up on TDD.

3.3. Static Analysis

Static Analysis

"Analysis of a program carried out without executing the program" – BS 7925-1

- Unreachable code
- Parameter type mismatches
- Possible array bound violations
- Faults found by compilers
- Program complexity



- *Unreachable code*

Part of a code that you can't reach, e.g. uncalled functions or procedures. Also called *dead code*.

- *Parameter type mismatches*

E.g. a variable declared with one type is sent to a procedure, but the procedure expects a variable of another type.

- *Possible array bound violations*

Trying to access an element index outside the boundary value of the array.

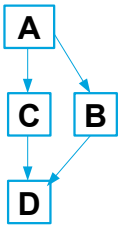
- *Faults found by compilers*

Fault types found by compilers depend first of all on the language – what is legal in it. For example, data type mismatches, missing files, possible division by 0, ranges without stop value, misuse of variables.

Static Analysis

- % of the source code changed
- Graphical representation of code properties:
  - Control flow graph

```
1: (A) int n = read_num();
2: (A) if(n % 2 == 0){
3: (B)   System.out.println(n + " is even.");
4: (C) } else {
5: (C)   System.out.println(n + " is odd.");
6: (D) }
```



  - Call tree
  - Sequence diagrams
  - Class diagrams

Data Flow Analysis

- *% of the source code changed*

Some tools can analyse and tell how many % of the source code have been changed and which parts that have been changed => input to test case generation.

- *Graphical representation of code properties*

Depends on development tools features.

- Considers the use of data (works better on sequential code)
- Examples:
  - Definitions with no intervening use
  - Attempted use of a variable after it is killed
  - Attempted use of a variable before it is defined

```
if(b > c){
  a=3;
  a=5;
  System.out.println(a);
}
```

```
a=3;
if(a < 3){
  b=7;
  System.out.println(b);
}
```


Note: Not to be confused with *data flow testing* which is a dynamic test case selection method.

- *Considers the use of data*

How are the variables used through the code?

- Definitions with no intervening use

IF B > C THEN A = 3;
A = 3; IF A < 3 THEN
A = 5; **B = 7;**
Print A; **Print B;**
END; END;

- Attempted use of a variable after it is killed

For example an attempt to read a variable outside its scope.

- Attempted use of a variable before it is defined

Static Metrics

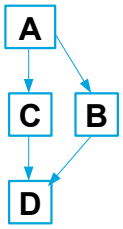
- McCabe's Cyclomatic complexity measure

  M = E − N + 2P

  E = number of edges

  N = number of nodes

  P = number of graph components



- Lines of Code (LoC)
- Fan-out and Fan-in
- Nesting levels

- *McCabe's Cyclomatic complexity*

Is defined as the number of decisions in a program or control flow graph + 1.

- *Lines of Code (LoC)*

Lines of code. It's a common measurement of the size of a program.

- *Fan-out and Fan-in*

Fan-out is the amount of modules *a given module calls*. Modules with high Fan-out are often found in the upper part of the call tree.

Fan-in is the amount of modules that *call a specific module*. Modules with high Fan-in are often found in the lower part of the call tree.

If a module has both high fan-in and fan-out, consider to redesign it.

- *Nesting levels*

For example many IF-statements nested into each other get a deep nesting level. This means that the code is difficult to understand. It is even worse when the cyclomatic complexity is also high.

One nesting level:

```
IF X > 5 THEN
   PRINT "BIG";
ELSE
   PRINT "SMALL";
ENDIF;
```

Who nesting levels:

```
IF X > 5 THEN
   IF X < 10 THEN
      PRINT "BIG UNIT";
   ENDIF;
ELSE
   IF X != 0 THEN
      PRINT "SMALL UNIT";
   ENDIF;
ENDIF;
```

4-Dynamic Testing Techniques

- Black and White box testing
- Black box test techniques
- White box text techniques
- Test data
- Error-Guessing

This part deals with dynamic testing techniques – methods that use executable test cases. These techniques are further divided into two groups (white-box and black-box testing techniques).

4.1-Black- and White-box Testing

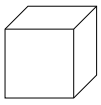
- Strategy
  - What's the purpose of testing?
  - What's the goal of testing?
  - How to reach the goal?
- Test Case Selection Methods
  - Which test cases are to be executed?
  - Are they good representatives of all possible test cases?
- Coverage Criteria
  - How much of code (requirements, functionality) is covered?

A good way of dealing with a testing problem is to first clarify the purpose of testing, then to define a goal and finally to develop a strategy for how to reach the goal.

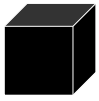
Once the goal has been defined, a test case selection strategy can be constructed. The obvious strategy would be to test everything, but due to infinite possibilities of choosing input this strategy is simply not feasible. Thus we need to carefully select the test cases that are to be executed. These test cases should be good representatives of all the possible test cases. To simplify the selection there exists a large number of test case selection methods, most of them are associated with coverage criteria to determine when to stop testing.

Test Case Selection Methods

- White-box / Structural / Logic driven
  - Based on the implementation (structure of the code)



- Black-box / Functional / Data driven
  - Based on the requirements (functional specifications, interfaces)



Test cases for dynamic execution are usually divided into two groups depending on the source of information used for creating the test case.

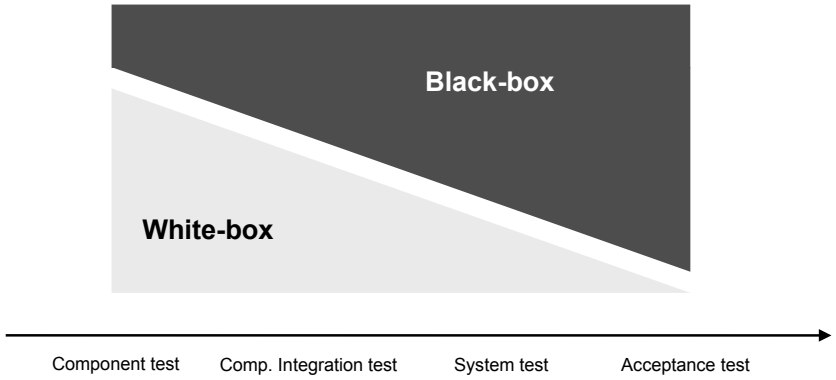
- *White-box*

Test cases are based on information about the implementation of the test object (structure of the code). The inputs of white-box test cases are generated from the implementation information (from the code). The testing is based on the program logic.

- *Black-box*

Test cases are aimed at testing the functionality of the test object. The inputs of black-box test cases are taken either from the requirements or from a model created from the requirements. Testing is based on inputs and

Importance of Test Methods



The two types of test cases are used a little bit differently in the development lifecycle. White-box test cases are mostly used in the early test phases of the development lifecycle and are of less usage higher up in the testing hierarchy.

There are two reasons for this:

1. The most important is that most white-box methods require extensive knowledge of the code and other parts of the implementation. Later test phases are usually performed by dedicated test specialists with neither deep implementation knowledge nor access to this information.
2. The other reason for not using white-box test case selection methods in later test stages is related to coverage. White-box test cases are usually more fine grained than black-box test cases. Fine grained test case selection methods require a large number of test cases in order to reach high coverage.

Black-box testing techniques are used throughout the development lifecycle. The main advantage with black-box testing techniques is that they only depend on the requirements, which means that test cases can be prepared before the implementation is complete.

Both methods are important. If only white-box testing would be performed, some requirements are not tested (performance requirements). On the other hand if only black-box test cases are used, some parts of the code might remain untested (special features called when a certain value is entered in a certain cell).

Measuring Code Coverage

- How much of the code has been executed?
- Code Coverage Metrics:
  - Segment coverage
  - Call-pair coverage

$$\text{Code Coverage} = \frac{\text{Executed code segments/call-pairs}}{\text{All code segments/call-pairs}}$$

- Tool Support:
  - Often a good help
  - For white-box tests almost a requirement

Code coverage metrics respond the question – How much of the code is being executed? There are usually 2 metrics:

- *Segment coverage*

A segment is a set of program statements that are executed unconditionally or executed conditionally based on the value of some logical expression. 85% is a practical coverage value.

- *Call-pair coverage*

A call pair is an interface whereby one module invokes another. Call-pair coverage is especially useful integration testing to ensure that all module interfaces are exercised. 100% is a practical coverage value.

As already has been mentioned, white-box testing techniques use implementation information to derive the input part of the test cases. Most often some aspect of the code, for instance the source code statements, is used for this purpose. Even with quite small programs, the task of keeping track of which statements that have already been tested and which statements that yet remain to be tested is quite difficult. The solution to this problem is to use a tool. There are a large number of *commercial code coverage* tools available for this. They all work in the same manner: before the source code of the object to be tested is compiled, the code is instrumented by adding extra instructions at strategic places in the original code. This is done by the tool.

The source code with the extra instructions is then compiled as usual and test cases are then executed in the normal way. The added instructions continuously log the progress of the testing and from the results of the logging instructions the tool can calculate which parts of the code that have been executed. Obviously the extra inserted instructions consume execution resources thus distorting performance measurements, so this type of tool is not appropriate during system testing.

Nevertheless, the use of such tools increase both the quality and the productivity of the testing in the earlier test phases.

Requirements Based Testing

- How much of the product's features is covered by TC?
- **Requirement Coverage Metrics:**

$$\text{Requirement Coverage} = \frac{\text{Tested requirements}}{\text{Total number of requirements}}$$

- What's the test progress?
- **Test Coverage Metrics:;**

$$\text{Test Coverage} = \frac{\text{Executed test cases}}{\text{Total number of test cases}}$$

The basic for all black-box testing is the requirements.

The simplest but still structured way of creating test cases is to write one test case for each requirement. The main drawback with this approach is that most requirements require more than one test case to be tested thoroughly, and different requirements require a different amount of test cases. In this case we can create the *coverage matrix* that tracks requirements to test cases and vice versa. This feature is usually included in test management tools.

Requirement Coverage responds the question: How much of the product's features is covered by test cases?

Test Coverage responds the question: What's

Creating Models

- Making models in general
  - Used to organize information
  - Often reveals problems while making the model
- Model based testing
  - Test cases extracted from the model
  - Examples
    - Syntax testing, State transition testing, Use case based testing
  - Coverage based on model used

A more elaborate way of creating black-box test cases is to transform a set of requirements into a model of the system and derive the test cases from the model instead of directly from the requirements.

In most model-based testing techniques there are well defined coverage criteria which are simple to calculate and interpret.

The main drawbacks with models are limited scope and validation. Often the purpose of the model and the modeling technique used, limits the scope of the model. For instance a syntax graph only captures the syntax of a language. The semantic of that language must be covered somewhere else. The result is that several models need to be developed and used in order to get a reasonable

Black-box Methods

- Equivalence Partitioning
- Boundary Value Analysis
- State Transition Testing
- Cause-Effect Graphing
- Syntax Testing
- Random Testing

- *Cause-Effect Graphing*

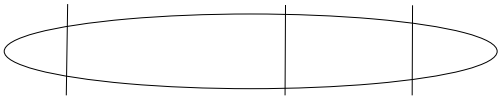
A model based method, which relates effects with causes through Boolean expressions. The main focus is on different combinations of inputs from the equivalence classes. Cause-effect graphing is a way of doing this whilst avoiding the major combinatorial problems that can arise.

- *Syntax Testing*

A model based method, which focuses on the syntax or rules (how different parts may be assembled) of a language (used during implementation). This method generates valid and invalid input data to a program. It is applicable to programs that have a hidden language that defines the data. Syntax generator is needed.

Equivalence Partitioning

- Identify sets of inputs under the assumption that all values in a set are treated exactly the same by the system under test
- Make one test case for each identified set (equivalence class)
- Most fundamental test case technique

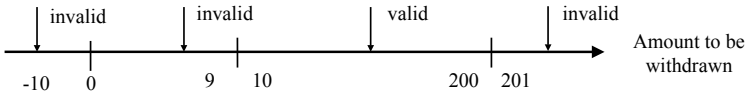


Equivalence class partitioning is one of the most basic black-box testing techniques. The underlying idea is that the input domain can be divided into a number of equivalence classes. The characteristic of an equivalence class is the assumption that all values belonging to that class are handled in exactly the same manner by the program.

If this assumption is true, then it would suffice to select one single test case for each equivalence class, since multiple test cases from the same equivalence class would repeat the same test.

Coverage is measured by dividing the number of executed test cases, i.e. the number of tested equivalence classes by the total number of equivalence classes.

Equivalence Partitioning (Example)



	Negative withdrawal	Even 10 less or equal than 200	Uneven 10 less than 200	More than 200
Enough money in account	*1. Withdrawal refused*	***2. Withdrawal granted***	*3. Withdrawal refused*	*4. Withdrawal refused*
Not enough money in account	*5. Withdrawal refused*	*6. Withdrawal refused*	*7. Withdrawal refused*	*8. Withdrawal refused*

Example: “A withdrawal from an ATM (Automatic Teller Machine) is granted if the account contains at least the desired amount. Furthermore, the amount withdrawn must be an even number of 10 EUR. The largest amount that can be withdrawn is 200 EUR.”

By analyzing the requirements we find several different independent dimensions to this problem:

- Is there enough money in the account?
- Is the desired amount an even 10-number?
- Is the desired amount outside the correct 0-200 range?

One way to organize the information is to make a table as above. Each cell in the table represents an equivalence class, which means that there should be eight test cases to solve this testing problem with equivalence partitioning.

In this example one could argue that negative withdrawal is not technically possible, and even if it was possible, the amount of money in the account would be irrelevant.

This discussion illustrates two difficult questions: how much should we really test? And which tests are most important?

Mostly this boils down to a matter of taste. Our view is that it is better to include too much when designing test cases that to miss vital functionality. Test cases should however always be assigned a priority based on importance to the end user and importance to future testing.

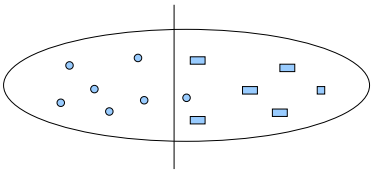
High priority test cases above could be 2, 3, 4 and 6.

Medium priority test cases above could be 7 and 8.

Low priority test cases above could be 1 and 5.

Boundary Value Analysis

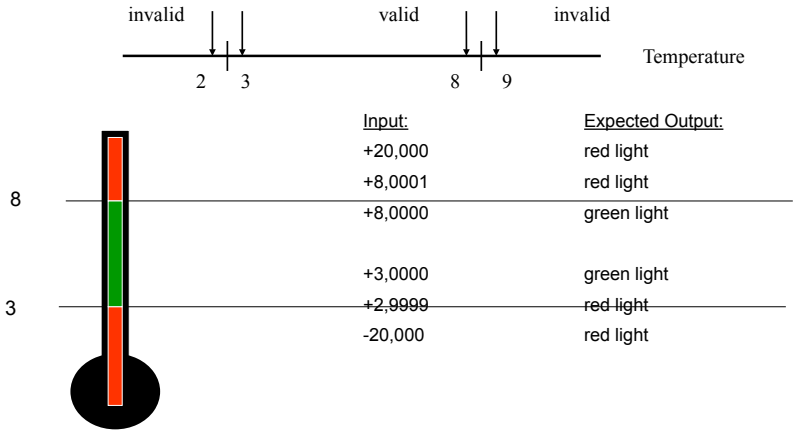
- For each identified boundary in input and output, create two test cases. One test case on each side of the boundary but both as close as possible to the actual boundary line.



Boundary Value Analysis is a refinement of equivalence class partitioning. Instead of choosing any representative from each equivalence class, interest is focused around the boundaries of each class. The idea is to select one test case for each boundary of the equivalence class. The properties of a test case is thus that is belongs to a defined equivalence class and that it tests a value that it is preferable on, or at least reasonably close to one of the boundaries of the equivalence class.

The main reason why boundaries are important is that they are generally used by programmers to control the execution of the program, for instance through if- or case-statements. Since the boundaries are being

Boundary Value Analysis (Example)



Input:	Expected Output:
+20,000	red light
+8,0001	red light
+8,0000	green light
+3,0000	green light
+2,9999	red light
-20,000	red light

Example: “A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 an +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit.”

The temperature range can be divided into three intervals (equivalence classes).

1. From –infinity (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to + infinity

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

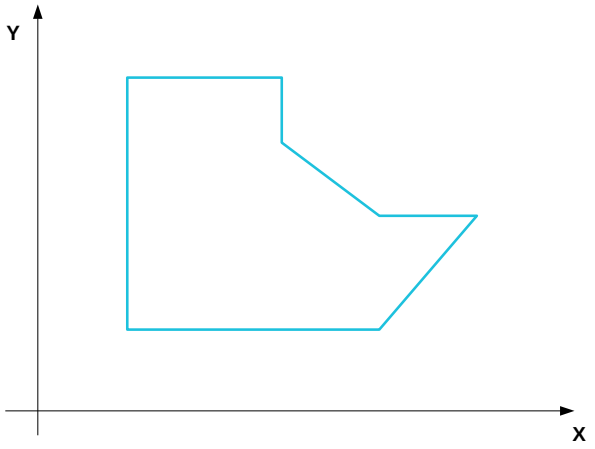
Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Boundary Value Analysis (Example)



Example: “A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 an +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit.”

The temperature range can be divided into three intervals (equivalence classes).

1. From –infinity (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to + infinity

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Cvičení

Boundary Value Analysis (Strings)

- Length min/max
- Empty
- whitespaces and non-visible characters – spaces, tabs, line break
- Separators – semicolon, comma, colon, quotation marks, apostrophes
- Special characters
- UTF-8 - czech, chines, …

Example: “A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 an +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit.”

The temperature range can be divided into three intervals (equivalence classes).

1. From –infinity (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to + infinity

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Boundary Value Analysis - Comparison

- Error detection on common mistakes:

Requirement	Mistake in impl.	EP	BVA
A < 18	A < =18	No	Yes
A < 18	A > 18	Yes	Yes
A < 18	A < 20	Maybe	Yes

- Number of test cases (one dimensional) BVA = 2*EP

Which is better, Equivalence Partitioning (EP) or Boundary Value Analysis (BVA)?

The answer depends on what we mean by better. Test cases made by BVA will catch more types of errors, but on the other hand there will be more test caes, which is more time consuming.

If you do boundaries only, you have covered all the partitions as well:

- Technically correct and may be OK if everything works correctly
- If the test fails, is the whole partition wrong, or is a boundary in the wrong place – have to test mid-partition anyway
- Testing only extremes may not give confidence for typical use scenarios (especially for users)

Test Objectives?

Conditions	Valid Partition	Tag	Invalid Partition	Tag	Valid Boundary	Tag	Invalid Boundary	Tag

- For a thorough approach: VP, IP, VB, IB
- Under time pressure, depends on your test objective
  - minimal user-confidence: VP only?
  - maximum fault finding: VB first (plus IB?)

State Transition Testing

- Model functional behaviour in state machine
- Create test cases
  - A) Touching each state
  - B) Using each transition (0-switch coverage)
  - C) For every possible chain of transition (n-switch coverage)
- Coverage
  - Depends on sub-strategy

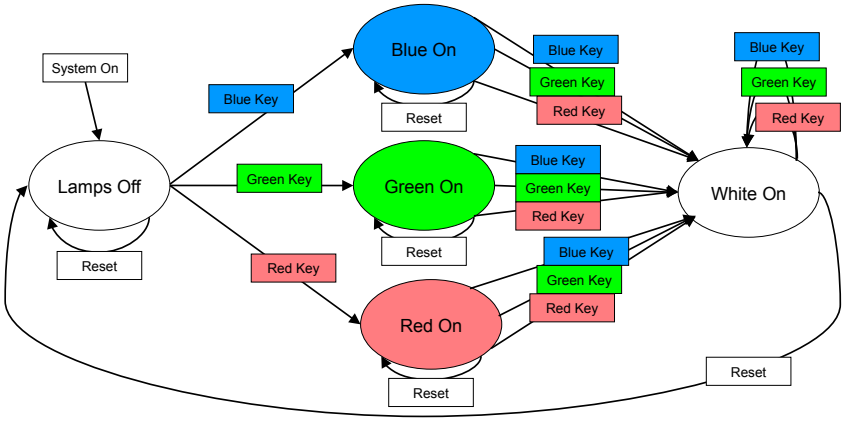
State machine based testing is a quite useful model based black-box testing technique, since any type of functionality that can be represented as a finite state machine can be tested using this technique.

The first step when using state machine testing is to construct the model itself. Sometimes, state machines are used by designers and constructors as implementation tools. In those cases, the state machines can of course be used directly. Otherwise the state machine model has to be constructed based on the requirements by the testers.

Often during construction of the state machine models, faults are found. One of the key properties with a state machine is that all input types can occur regardless of the state of the machine. If a state machine model previously has not been drawn, there are almost always disregarded combinations of state and input, which are very easily discovered when building the model.

When the model is finished, the next step is to construct test cases from it. There are several different strategies. The simplest and least powerful is to cover each state in the model at least once. As soon as there are more than one way of reaching a particular state, state coverage will most likely leave some transitions untested. A more elaborate strategy is therefore to focus on the transitions between the states. 0-switch coverage requires one test case for each possible transition in the model. 1-switch coverage requires a test case for every possible pair of consecutive transitions and finally n-switch coverage requires a test case for every possible n-1 consecutive transitions in the model.

State Transition Testing (Example)



Example:
- Four keys, four lamps
- After the start, all lamps are off
- A colored key turns “its” lamp on, if all lamps are off
- Next colored key turns the white lamp on and the colored off
- The *Reset* key turns the white lamp off and resets the system

There are 5 states.
To determine how many transitions there are, it is helpful to calculate the number of transitions out from each state (in our case there are 4 transitions):
(5*4 + 1) = 21 transitions (0-switch)
(5*4*4 + 4) = 84 pairs of transitions (1-switch)
It’s easy to understand that ‘time-outs’, common is real-time applications, will make it even more advanced.
Create test cases:
- A) touching each state
- 5 test cases – sufficient for such a simple system
- B) using each transition (0-switch coverage)
- 21 test cases – if the white lamp did not turn on after the green lamp, it is necessary to use “each transition” to catch this fault
- C) using every possible *pair* of transitions (1-switch coverage)
- 84 test cases – if the *Reset* key does not work after the red lamp and the blue key (but works after all other keys), finding this fault requires trying “all pairs of transitions”

To discover a fault which, for example, causes the system to hang after a thousand loops, still another strategy is required.
The number of tested inputs is another dilemma. Should all possible inputs be tried in each state? The strategy described here do not answer this question.

Labs

4.3-White-Box Test Techniques

- Test case input always derived from implementation information (code)
- Most common implementation info:
  - Statements
  - Decision Points in the Code
  - Usage of variables
- Expected output in test case always from requirements!

When creating white-box test cases the basis in the implementation. The input part of the test case is derived from the implementation.

Commonly used implementation properties include code structure and how variables are used in the code. Less common but nevertheless interesting implementation properties are call-structures and process/object interactions.

Regardless of the white-box test method chosen, expected output is always extracted from the requirements and not from the implementation itself.

Labs

White-box Test Methods

- Statement Testing
- Branch/Decision Testing
- Data Flow Testing
- Branch Condition Testing
- Branch Condition Combination Testing
- Modified Condition Testing
- LCSAJ Testing

- *Statement Testing*

The idea with statement coverage is to create enough test cases so that every statement in the source code has been executed at least once

- *Branch/Decision Testing*

The idea with decision coverage is to execute every single decision in the code at least twice (both possible outcomes of the decision should be executed in order to reach full decision coverage)

- *Data Flow Testing*

Test cases are designed based on variable usage within the code

- *Branch Condition Testing*

A test case design technique in which test cases are designed to execute branch condition outcomes

- *Branch Condition Combination Testing*

A test case design technique in which test cases are designed to execute combination of branch condition outcomes

- *Modified Condition Testing*

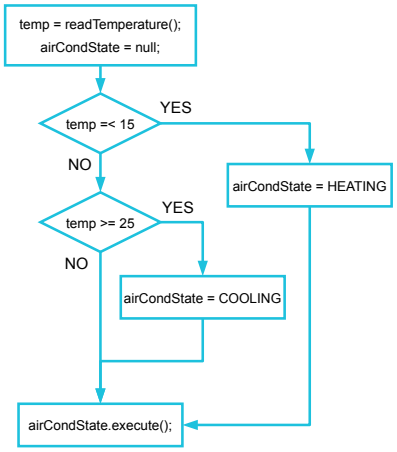
A test case design technique in which test cases are designed to execute branch condition outcomes that independently affect a decision outcome

- *LCSAJ Testing*

Linear Code Sequence And Jump (LCSAJ) – Select test cases based on jump-free sequence of code. It consists of the following three items: the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

Control Flow Graphs

```
public void doAirconditioning() {
  double temp = readTemperature();
  Aircondition airCondState = null;
  if(temp <= 15) {
    airCondState = Aircondition.HEATING;
  }
  else if(temp >= 25) {
    airCondState = Aircondition.COOLING;
  }
  airCondState.execute();
}
```



This is a small piece of code, which implements the temperature regulation. The function “adjust_temperature” is called without arguments. The first thing it does is to read the current temperature, and then depending on the value, either the heater is switched on, the cooler is switched on, or the system is left untouched. The global variable *control* holds the current setting of the heater and cooler.

To the right the is the corresponding control flow graph. To aid the understanding of the control flow graph strategic parts of the code may be inserted in the diamonds and boxes.

McCabe’s cyclomatic complexity measure: No. of diamonds + 1 (2 + 1 = 3) – it says that the more decisions there are in a piece of code,

Labs

Statement Testing

- Execute every statement in the code at least once during test case execution
- Requires the use of tools
  - Instrumentation skews performance
- Coverage

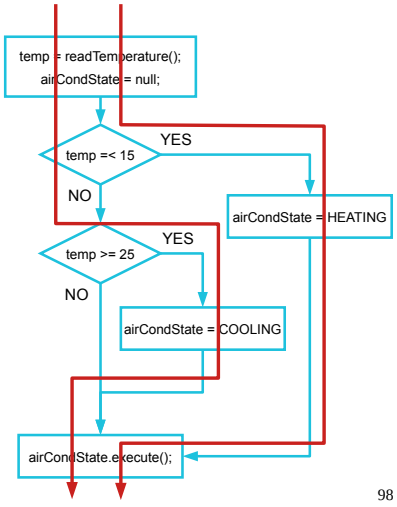
$$\text{Statement Coverage} = \frac{\text{Executed statements}}{\text{Total number of statements}}$$

Statement coverage is a fundamental white-box testing technique. This idea of statement coverage is to create enough test cases so that every statement in the source code has been executed at least once.

The workflow when using statement coverage is to first execute all existing black-box test cases that has been created while monitoring the execution. This monitoring is in all but the simplest test cases performed with tool support. When all black-box test cases have been executed, the tool can report which parts of the code that remain untested. The idea is now to construct new test cases that will cover as many of the remaining statements as possible. Start with the part of the code that should be reached, walk

Statement Coverage

```
public void doAirconditioning() {
  double temp = readTemperature();
  Aircondition airCondState = null;
  if(temp <= 15) {
    airCondState = Aircondition.HEATING;
  }
  else if(temp >= 25) {
    airCondState = Aircondition.COOLING;
  }
  airCondState.execute();
}
```



When creating test cases for statement coverage we can make use of the control flow graph. We know the statement coverage requires statements in the code to be executed. We also know that the boxes and the diamonds represent all the statements in the code.

By following the two blue arrows through the code we cover all the diamonds and all the boxes are covered and thus we have statement coverage (according to the relation with McCabe measure there should be three or less test cases and in this case two were enough).

By examine the relation we can now also deduce that in the optimal choice of test cases, number of test cases for decision

Labs

Branch/Decision Testing

- Create test cases so that each decision in the code executes with both TRUE and FALSE outcomes
  - Equivalent to executing all branches
- Requires the use of tools
  - Instrumentation skews performance
- Coverage

$$\text{Decision Coverage} = \frac{\text{Executed decision outcomes}}{2 * \text{Total number of decisions}}$$

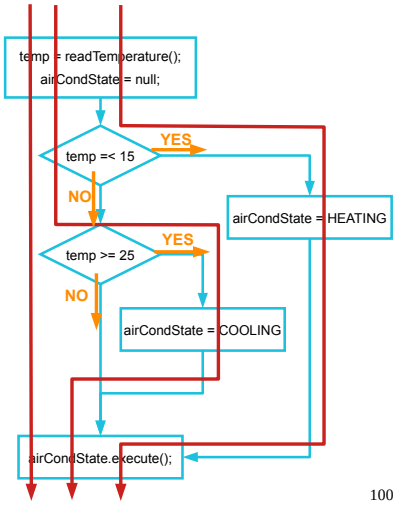
Branch coverage and decision coverage are two names for the same thing.

Decision coverage is a technique similar to statement coverage. The idea with decision coverage is to execute every single decision in the code at least twice. Both possible outcomes of the decision, i.e. true and false, should be executed in order to reach full decision coverage.

By the first glance statement and decision coverage seem to yield exactly the same test cases, since executing every decision with both true and false outcomes will result in all statements being executed, and in order to execute all statements all outcomes of every decision needs to be executed. However this is not entirely true. There is one case in

Branch/Decision Testing

```
public void doAirconditioning() {
  double temp = readTemperature();
  Aircondition airCondState = null;
  if(temp <= 15) {
    airCondState = Aircondition.HEATING;
  }
  else if(temp >= 25) {
    airCondState = Aircondition.COOLING;
  }
  airCondState.execute();
}
```



Branch coverage and decision coverage are two names for the same thing.

Decision coverage is a technique similar to statement coverage. The idea with decision coverage is to execute every single decision in the code at least twice. Both possible outcomes of the decision, i.e. true and false, should be executed in order to reach full decision coverage.

By the first glance statement and decision coverage seem to yield exactly the same test cases, since executing every decision with both true and false outcomes will result in all statements being executed, and in order to execute all statements all outcomes of every decision needs to be executed. However this is not entirely true. There is one case in

Path Coverage

- Coverage for all possible paths through code (all combinations of decisions)
- Code with cycles
  - Test all possible number of iterations → not possible
  - Recommendation: 0 iteration, 1 iteration, n iteration
- Coverage

$$\text{Path coverage} = \frac{\text{number of tested paths}}{\text{2^numberOfDecisions}}$$

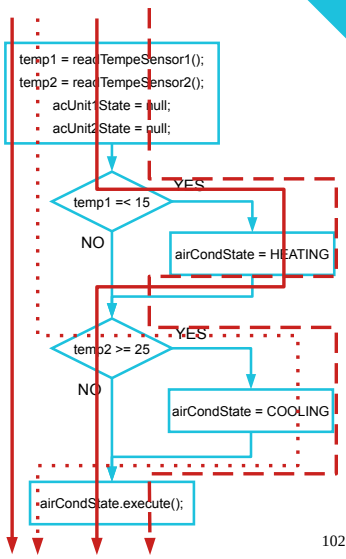
Branch coverage and decision coverage are two names for the same thing.

Decision coverage is a technique similar to statement coverage. The idea with decision coverage is to execute every single decision in the code at least twice. Both possible outcomes of the decision, i.e. true and false, should be executed in order to reach full decision coverage.

By the first glance statement and decision coverage seem to yield exactly the same test cases, since executing every decision with both true and false outcomes will result in all statements being executed, and in order to execute all statements all outcomes of every decision needs to be executed. However this is not entirely true. There is one case in

Path Coverage

```
public void adjustTemperature2() {
  double temp1 = readTempeSensor1();
  double temp2 = readTempeSensor2();
  Aircondition acUnit1State = null;
  Aircondition acUnit2State = null;
  if(temp1 <=15) {
    acUnit1State = Aircondition.HEATING;
  }
  if(temp2>=25) {
    acUnit2State = Aircondition.COOLING;
  }
  acUnit1State.execute();
  acUnit2State.execute();
}
```



Branch coverage and decision coverage are two names for the same thing.

Decision coverage is a technique similar to statement coverage. The idea with decision coverage is to execute every single decision in the code at least twice. Both possible outcomes of the decision, i.e. true and false, should be executed in order to reach full decision coverage.

By the first glance statement and decision coverage seem to yield exactly the same test cases, since executing every decision with both true and false outcomes will result in all statements being executed, and in order to execute all statements all outcomes of every decision needs to be executed. However this is not entirely true. There is one case in

Data flow coverage

```
01: public QResult quadratic(double a,
double b, double c) {
02:   double disc = b*b - 4*a*c;
03:   QResult r = new QResult();
04:     if(disc < 0) {
05:     r.isComplex = true;
06:   } else {
07:     r.isComplex = false;
08:   }
09:   if(!r.isComplex) {
10:     r.r1 = (-b + Math.sqrt(disc))/(2*a);
11:     r.r2 = (-b - Math.sqrt(disc))/(2*a);
12:   }
13:   return r;
14: }
```

line	category		
	definition	c-use	p-use
1	a,b,c		
2	disc	a,b,c	
3	r.isComplex, r.r1, r.r2		
4			disc
5	r.isComplex		
6			
7	r.isComplex		
8			
9			r.isComplex
10	r.r1	a,b,disc	
11	r.r2	a,b,disc	
12			
13		r.isComplex , r.r1, r.r2	
14			

c-use(v): (c for computation) all variables that are used to define other variables in the code corresponding to v

p-use(v; v0): (p for predicate) all variables used in taking the (v; v0) branch out of vertex v.

http://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/07_dataflow1.pdf

Data flow coverage

line	category		
	definition	c-use	p-use
1	a,b,c		
2	disc	a,b,c	
3	r.isComplex, r.r1, r.r2		
4			
5			disc
6	r.isComplex		
7			
8	r.isComplex		
9			
10			r.isComplex
11	r.r1	a,b,disc	
12	r.r2	a,b,disc	
13		r.isComplex, r.r1, r.r2	
14			

Pairs definition → use	variables	
Start → end	c-use	p-use
1→2	a,b,c	
1→11	a,b,c	
1→12	a,b	
2→5		disc
2→11	disc	
2→12	disc	
3→10		r.isComplex
3→13	r.isComplex, r.r1, r.r2	
6→10		r.isComplex
6→13	r.isComplex	
8→10		r.isComplex
8→13	r.isComplex	
11→13	r.r1	
12→13	r.r2	

c-use(v): (c for computation) all variables that are used to define other variables in the code corresponding to v

p-use(v; v0): (p for predicate) all variables used in taking the (v; v0) branch out of vertex v.

http://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/07_dataflow1.pdf

Branch Condition Testing

```
if(A || (B && C)) {
  //do something
} else {
  //do something else
}
```

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	TRUE	TRUE

Každý oberand podmínky se musí provést pro hodnotu true i false.

Modified condition/decision coverage

```
if(A || (B && C)) {
  //do something
} else {
  //do something else
}
```

- Test all combinations of bools operands A, B, C

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	TRUE
5	TRUE	TRUE	FALSE
6	FALSE	TRUE	TRUE
7	TRUE	FALSE	TRUE
8	TRUE	TRUE	TRUE

Každý oberand podmínky se musí provést pro hodnotu true i false.

Modified condition/decision coverage

Case	A	B	C	Output
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE
Case	**A**	**B**	**C**	**Output**
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE
Case	**A**	**B**	**C**	**Output**
C1	FALSE	TRUE	TRUE	TRUE
C2	FALSE	TRUE	FALSE	FALSE
Case	**A**	**B**	**C**	**Output**
1 (A1,B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2,C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

Modified Condition Decision Testing and Coverage

Modified Condition Decision Coverage (MCDC) is a pragmatic compromise which requires fewer test cases than Branch Condition Combination Coverage. It is widely used in the development of avionics software, as required by RTCA/DO-178B.

Modified Condition Decision Coverage requires test cases to show that each Boolean operand (A, B and C) can independently affect the outcome of the decision. This is less than all the combinations (as required by Branch Condition Combination Coverage).

For the example decision condition [A or (B and C)], we first require a pair of test cases where changing the state of A will change the outcome, but B and C remain constant, i.e. that A can independently affect the outcome of the condition:

Cvičení

Linear Code Sequence and Jump (LCSAJ)

```
1.int main (void) {
2.  int count = 0, totals[MAXCOLUMNS], val =
  0;
3.  memset (totals, 0, MAXCOLUMNS *
 sizeof(int));
4.  count = 0;
5.  while ( count < ITERATIONS ) {
6.    val = abs(rand()) % MAXCOLUMNS;
7.    totals[val] += 1;
8.    if ( totals[val] > MAXCOUNT ) {
9.      totals[val] = MAXCOUNT;
10.   }
11.   count++;
12.  }
13.  return (0);
14.}
```

LCSAJ Block	Start	End	Jump to
1	1	5	13
2	1	8	11
3	1	12	5
4	5	5	13
5	5	8	11
6	5	12	5
7	11	12	5
8	13	13	−1

http://en.wikipedia.org/wiki/Linear_code_sequence_and_jump

4.4-Test Data

Test Data Preparation

- Professional data generators
- Modified production data
- Data administration tools
- Own data generators
- Excel
- Test automation tools (test-running tools, test data preparation tools, performance test tools, load generators, etc.)

- *Professional data generators*
- Data generator controlled by syntax and semantics
- Stochastic data generator
- Data generator based on heuristic algorithms
- Combination of previous methods
- *Modified production data*

The data must be degraded (omitting sensitive data) before using as test data. The advantage is that we have test data that are close to real production data. The disadvantage is that data must be modified to have all combinations needed for test cases.

- *Data administration tools*

E.g. File-AID/Data Solution (Compuware), RC Extract (CA), Startool/Comparex (Serena), Relational Tools for Servers (Princeton Softech), detailed knowledge of DB structure and links is needed. Tools are ready for it's sometimes difficult and laborious.

- *Own data generators*

Development resources are needed, suitable when combining with Excel. Sophisticated data can be generated which are tailored to the needs of test cases.

- *Excel*

DB tables are stored in Excel, SQL scripts generate DB structures tailored to the needs of test cases. The initial data has to be stored manually – laborious.

- *Test automation tools*

E.g. WinRunner, QuickTest Pro, LoadRunner, SilkPerformer, etc. Data can be generated during nights, test data can be stored to database, Excel or text files. Tools are often expensive.

What Influences Test Data Preparation?

- Complexity and functionality of the application
- Used development tool
- Repetition of testing
- The amount of data needed

- *Complexity and functionality of the application*

It directly influences the range of testing, mutual linking and the amount of test data (financial systems must be tested in more details than registration systems).

- *Used development tool*

In case of using test automation tools, the development environment must be compatible with used test tools.

- *Repetition of testing*

The efficiency of using test automation tools is higher the higher repetition of the same test cases (regression testing) is (valid not only for test data preparation but also for test execution.

- *The amount of data needed*

Recommendation for Test Data

- Don’t remove old test data
- Describe test data
- Check test data
- Expected results

- *Don’t remove old test data*

Create test data archive for future use.

- *Describe test data*

Create your own information system from metadata describing content, form and effectiveness of test data.

- *Check test data*

Test data must be error free.

- *Expected results*

Don’t underestimate time needed for setting expected results for generated test data.