581361 Software Testing (Ohjelmistojen testaus)

5 cr / op (new degree structure)
3 cu / ov (old degree structure)

Jukka Paakki
Department of Computer Science
University of Helsinki

Lecturer: *Jukka Paakki*

- Lectures 01.11.-08.12.2005
  - Tuesday, 10-12, CK112 (not on 06.12.)
  - Thursday, 12-14, D122
- Office hours:
  - Tuesdays, 9:30-10:00 & Wednesdays, 16:00-17:00 (room D240b)
- E-mail: jukka.paakki@cs.helsinki.fi
- Phones: 191 51387, 0500-852368

Course assistant: *Juha Gustafsson*

- Exercise sessions 07.11 – 09.12.2005
  - Tuesday, 12-14, C222 (not on 06.12.)
  - Thursday, 10-12, CK107

Prerequisites

§ Software Engineering (Ohjelmistotuotanto)

§ Software Engineering Project (Ohjelmistotuotantoprojekti)

Special course (at least) in the Software Engineering specialization area

Objectives of the course

§ general introductory course on software testing
§ main concepts and principles
§ main techniques
§ no deep theory
§ no particular application area
§ practical small-scale training by exercises

Grading

- Maximum: 60 points
  - Course exam: maximum 50 points
  - Exercises: maximum 10 points (not mandatory)
  - Pass: at least 30 points
- Scale: 1, 2, 3, 4, 5

Course material

http://www.cs.helsinki.fi/u/paakki/software-testing-s05.html

- Lecture slides
- Exercises

No lecture notes or primary text book, but the following book covers most of the contents:

R. V. Binder: *Testing Object-Oriented Systems – Models, Patterns, and Tools.* Addison-Wesley, 2000.

Examinations

§ Course exam: Monday, December 12, 16:00-19:00

§ Separate exams:
- Tuesday, February 7, 2006, 16:00-20:00
- Tuesday, April 4, 2006, 16:00-20:00
- Friday, June 9, 2006, 16:00-20:00
- Extra points from the course exercises do not count

Contents

1. Software quality, terminology
2. Principles of software testing
3. Management of testing: process, reports, tools
4. Black-box testing
5. White-box testing
6. State-based testing
7. Testing object-oriented software
8. Integration testing
9. Regression testing
10. Statistical testing
11. Practical aspects of testing (incl. guest lecture?)

1. Software quality

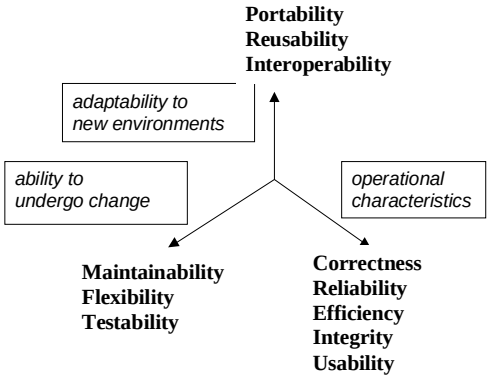
Standard Glossary of Software Engineering Terminology [IEEE610.12]:

> ***Quality***: (1) The degree to which a system, component, or process meets **specified requirements**. (2) The degree to which a system, component, or process meets customer or **user needs or expectations**.

R.S. Pressman: *Software Engineering – A Practitioner's Approach* (5th ed). McGraw-Hill, 2000:

> ***Software quality***: Conformance to **explicitly stated** functional and performance requirements, **explicitly documented** development standards, and implicit characteristics that are expected of all **professionally** developed software.

Dimensions of software quality



§ ***Portability***: The effort required to transfer the system from one hardware and/or software environment to another.
§ ***Reusability***: The extent to which the system (or part of it) can be reused in other applications.
§ ***Interoperability***: The effort required to couple the system to another.
§ ***Maintainability***: The effort required to introduce a modification (usually a correction) into the system.
§ ***Flexibility***: The effort required to modify or customize the system in operation.
§ ***Testability***: The effort required to test the system to ensure that it performs its intended function.

§ ***Correctness***: The extent to which the system satisfies its specification and fulfils the users' needs.
§ ***Reliability***: The extent to which the system can be expected to perform its intended function with required precision and without failure.
§ ***Efficiency***: The amount of computing resources (space, time) required by the system to perform its function.
§ *Integrity*: The extent to which access to the system or its data by unauthorized persons can be controlled.
§ *Usability*: The effort required to learn, operate, prepare input and interpret output of the system.

Measurement ?

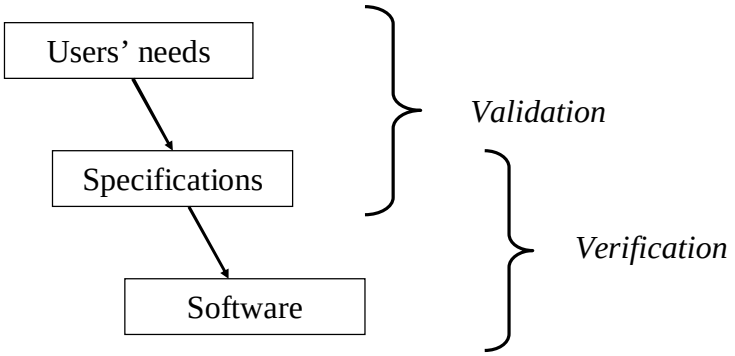
Standard Glossary of Software Engineering Terminology [IEEE610.12]:

Quality assurance: (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured.

§ application of sound technical methods and tools
§ formal technical reviews and inspections
§ ***software testing***
§ enforcement of standards
§ documentation
§ control of change
§ extensive measurement
§ record keeping and reporting of the process

Verification: are we building the *product right* ?

Validation: are we building the *right product* ?



2. Principles of software testing

(1) Testing is a process of ***executing*** a program with the intent of finding a defect. So, there must be some program code to be executed.

(2) A good test case is one that has a high probability of finding an as yet undiscovered defect. So, the test cases (the program input) should be selected systematically and with care, both for correct and incorrect behavior.

(3) A successful test is one that uncovers an as yet undiscovered defect. So, testing is psychologically ***destructive*** since it tries to demolish the software that has been constructed.

(4) Testing cannot show the absence of defects, it can only show that they are present (Dijkstra).
[Testing is not *formal verification*.]

(5) Testing is quite an ineffective method of quality assurance.
[Though, usually the most applicable one.]

(6) Successful testing shall be followed by a separate *debugging* phase.

(7) Testing is also by itself a *process* that must be systematically managed (and assisted with special testing tools).

Error (virhe): A mistake (*human* action) made by a software developer. It might be a total mis-interpretation of user requirements, or a simple typograhpical missprint. An error introduces a defect into the *software code*.

Defect, fault, bug (vika): A difference between the incorrect program and its correct version; a coding error. A defect in the software, if encountered during *execution*, may cause a failure.

Failure (häiriö): An externally observable deviation of the functional software from its specification; an incorrect result of computation.

***Implications*:**

§ One must know what is “correct” and what is “incorrect”
§ There must be a specification against which to check the results of testing
§ Full automation of testing is impossible

– theoretically, the total behavior of a program is undecidable (halting, failures)
– in practice, exhaustive testing is intractable
– tracking of (technical) failures to (human) errors is impossible
– we can never be sure that the testing tool (a program) works correctly

Why testing? Why *systematic* testing?

- According to several empirical studies, a (professionally produced commercial) software system contains 3 – 30 defects in every 1000 lines of code
- ..., the average debugging effort is 12 hours of working time for a single defect
- ..., maintenance eats about 50% of software development costs, mostly in error removal

Example

The program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. *Write test cases* (*specific input values*) *that you feel would adequately test this program.*

- In a valid triangle, no side may have a length of zero or less, and each side must be shorter than the sum of all sides divided by 2.
- *Equilateral* (tasasivuinen) triangle: all sides are of equal length.
- *Isosceles* (tasakylkinen) triangle: two sides are of equal length.
- *Scalene* (epäsymmetrinen) triangle: all sides are of unequal length.

Example (cont.)

In mathematics, the number of integer values is infinite. However, computers have finite space which limits the number of values that can be processed. Let us assume that our triangle program is running in a tiny computer with 10.000 as the largest integer value. Then there are $10^4 \times 10^4 \times 10^4 = 10^{12}$ possible length combinations of triangle sides (including the invalid ones).

- Suppose you are a very fast tester, running and checking 1000 tests per second, 24 hours per day, 365 days per year.
- Then the exhaustive testing effort (testing each possible length combination) would take over 317 years.

Example (cont.)

Myers (*The Art of Software Testing*, 1978) lists 24 test cases:

1.	(5, 3, 4):	scalene
2.	(3, 3, 4):	isoscele
3.	(3, 3, 3):	equilateral
4.	(50, 50, 25):	isoscele
5.	(25, 50, 50):	isoscele (permutation)
6.	(50, 25, 50):	isoscele (permutation)
7.	(10, 10, 0):	invalid (zero)
8.	(3, 3, -4):	invalid (negative)
9.	(5, 5, 10):	invalid (too long)
10.	(10, 5, 5):	invalid (too long, perm.)
11.	(5, 10, 5):	invalid (too long, perm.)
12.	(8, 2, 5):	invalid (Too long)

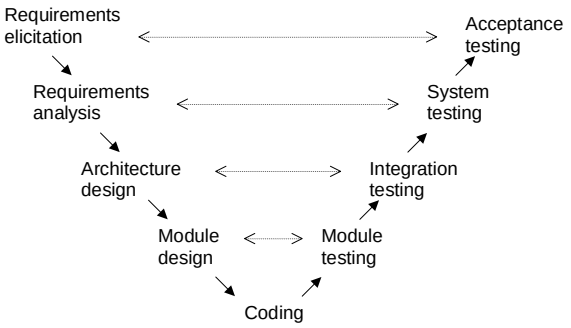
13.	(2, 5, 8): invalid (Too long, perm.)
14.	(2, 8, 5): invalid (Too long, perm.)
15.	(8, 5, 2): invalid (Too long, perm.)
16.	(5, 8, 2): invalid (Too long, perm.)
17.	(5, 2, 8): invalid (Too long, perm.)
18.	(0, 0, 0): invalid (all zeros)
19.	(@, 4, 5): invalid (non-integer)
20.	(3, $, 5): invalid (non-integer)
21.	(3, 4, %): invalid (non-integer)
22.	(, 4, 5): invalid (missing input)
23.	(3,,5): invalid (missing input)
24.	(3, 4,): invalid (missing input)

Example (cont.)

Remarks:

- most test cases represent invalid inputs
- each valid triangle type is tested at least once
- permutations are used to check that the order of the input values does not affect the result
- boundary input values are used (length of exactly zero, length of exactly the sum of all sides divided by 2)
- input values of wrong type (non-integers) are used
- the number of test cases is rather small with respect to the number of all possible inputs

The "V" model of software testing:



Specification *Test planning* *Testing level* (*phase*)

***Module* (*unit*) *testing*:**

§ each independent unit tested separately
§ level: source code
§ usually need for simulated execution environment ("stubs", "drivers")

***Integration testing*:**

§ modules grouped into subsystems for testing
§ "big bang": all the modules tested as a whole
§ incremental approaches: modules into subsystems level-wise ("stubs" and "drivers")
§ level: interfaces between modules

***System testing*:**

§ the whole system (including hardware, databases, sensors, …) tested
§ target: performance, capacity, fault-tolerance, security, configuration, …
§ level: external interface

***Special forms of system testing*:**

§ Volume testing
§ Load / stress testing
§ Security testing
§ Performance testing
§ Configuration testing
§ Installability testing
§ Recovery testing
§ Reliability / availability testing
§ Maintainability testing

§ Protocol conformance testing, etc.

Acceptance testing:

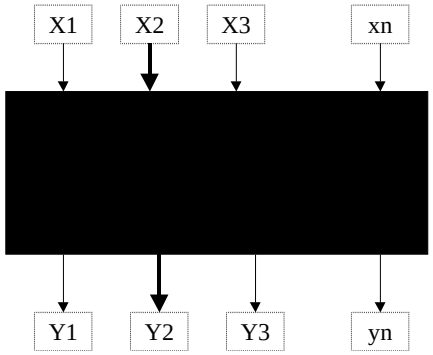
§ user involvement (alpha, beta)
§ "actual needs"

§ ***usability testing*** at the user interface
  ─ development team in development environment: "standard", general usability errors
  ─ real user representatives in laboratory environment: task-specific usability problems (real tasks, talk-aloud, taping, post-analysis by experts)

Black-box (*functional*) *testing:*

§ internal details of modules or subsystems are hidden and cannot be studied from outside
§ concentrates on the *interfaces* of modules and (sub)systems (e.g. user interface)
§ externally observable functionality and input-output behavior
§ based on input *classification*
§ especially suitable for *integration*, *system*, and *acceptance testing*

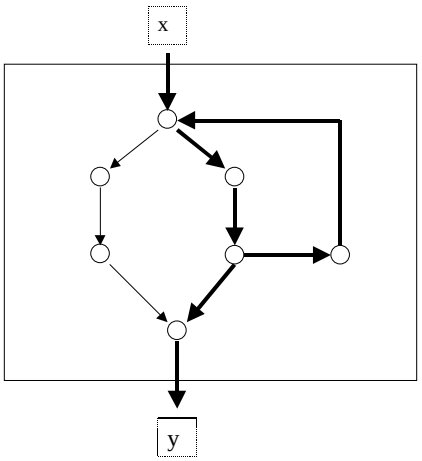
Black box:



White-box* (*structural*) *testing

§ structure of the software is examined in detail at the level of program code
§ objective to traverse as many *paths* over the code as considered necessary
§ based on *control flow* and *data flow*
§ several forms of *coverage* (path, statement, branch, …)
§ especially suitable for *module* (*unit*) testing

White box:

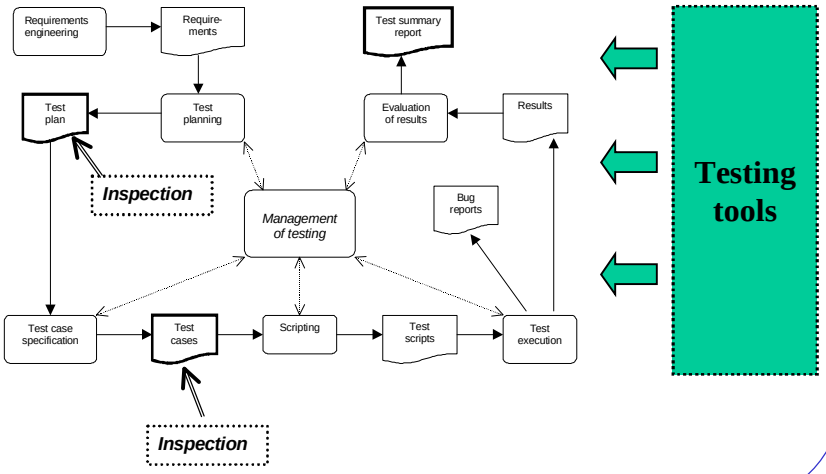


3. Management of testing

§ plan
§ execute
§ evaluate
§ document
§ report

} *process*

Testing process (for each main phase):



***Standard for Software Test Documentation* [IEEE 829]:**

1. ***Test plan***: the scope, approach, resources, and schedule of the testing activities.
2. *Test-design specification*: the refinements of the test approach, and the features to be tested by the design and its associated tests.
3. ***Test-case specification***: a test case identified by a test-design specification.
4. *Test-procedure specification*: the steps for executing a set of test cases or, more generally, the steps used to analyze a software item in order to evaluate a set of features.

5. *Test-item transmittal report*: the test items being transmitted for testing, including the person responsible for each item, its physical location, and its status.
6. *Test log*: a chronological record of relevant details about the execution of tests.
7. *Test-incident report* **(*bug report*):** any event that occurs during the testing process which requires investigation.
8. ***Test-summary report***: the results of the designated activities, and evaluations based on these results.

3.1. Test plan

1. *Test-plan identifier*: specifies the unique identifier assigned to the test plan.
2. *Introduction*: summarizes the software items and software features to be tested, provides references to the documents relevant for testing (overall project plan, quality assurance plan, configuration management plan, applicable standards…).
3. *Test items*: identifies the items to be tested, including their version/revision level; provides references to the relevant item documentation (requirements specification, design specification, user's guide, operations guide, installation guide, …); also identifies items which are specifically excluded from testing.

4. *Features to be tested*: identifies all software features and their combinations to be tested, identifies the test-design specification associated with each feature and each combination of features.
5. *Features not to be tested*: identifies all features and significant combinations of features which will not be tested, and the reasons for this.
6. *Approach*: describes the overall approach to testing (the testing activities and techniques applied, the testing of non-functional requirements such as performance and security, the tools used in testing); specifies completion criteria (for example, error frequency or code coverage); identifies significant constraints such as testing-resource availability and strict deadlines; serves for estimating the testing efforts.

7. *Item pass/fail criteria*: specifies the criteria to be used to determine whether each test item has passed or failed testing.
8. *Suspension criteria and resumption*: specifies the criteria used to suspend all or portion of the testing activity on the test items (at the end of working day, due to hardware failure or other external exception, …), specifies the testing activities which must be repeated when testing is resumed.
9. *Test deliverables*: identifies the deliverable documents, typically test-design specifications, test-case specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, description of test-input data and test-output data, description of test tools.

10. *Testing tasks*: identifies the set of tasks necessary to prepare and perform testing (description of the main phases in the testing process, design of verification mechanisms, plan for maintenance of the testing environment, …).
11. *Environmental needs*: specifies both the necessary and desired properties of the test environment (hardware, communications and systems software, software libraries, test support tools, level of security for the test facilities, drivers and stubs to be implemented, office or laboratory space, …).

12. *Responsibilities*: identifies the groups of persons responsible for managing, designing, preparing, executing, witnessing, checking, and resolving the testing process; identifies the groups responsible for providing the test items (section 3) and the environmental needs (section 11).
13. *Staffing and training needs*: specifies the number of testers by skill level, and identifies training options for providing necessary skills.
14. *Schedule*: includes test milestones (those defined in the overall project plan as well as those identified as internal ones in the testing process), estimates the time required to do each testing task, identifies the temporal dependencies between testing tasks, specifies the schedule over calendar time for each task and milestone.

15. *Risks and contingencies*: identifies the high-risk assumptions of the test plan (lack of skilled personnel, possible technical problems, …), specifies contingency plans for each risk (employment of additional testers, increase of night shift, exclusion of some tests of minor importance, …).
16. *Approvals*: specifies the persons who must approve this plan.

3.2. Test-case specification

1. *Test-case-specification identifier*: specifies the unique identifier assigned to this test-case specification.
2. *Test items*: identifies and briefly describes the items and features to be exercised by this test case, supplies references to the relevant item documentation (requirements specification, design specification, user's guide, operations guide, installation guide, …).
3. *Input specifications*: specifies each input required to execute the test case (by value with tolerances or by name); identifies all appropriate databases, files, terminal messages, memory resident areas, and external values passed by the operating system; specifies all required relationships between inputs (for example, timing).

4. *Output specifications*: specifies all of the outputs and features (for example, response time) required of the test items, provides the exact value (with tolerances where appropriate) for each required output or feature.
5. *Environmental needs*: specifies the hardware and software configuration needed to execute this test case, as well as other requirements (such as specially trained operators or testers).
6. *Special procedural requirements*: describes any special constraints on the test procedures which execute this test case (special set-up, operator intervention, …).
7. *Intercase dependencies*: lists the identifiers of test cases which must be executed prior to this test case, describes the nature of the dependencies.

3.3. Test-incident report (bug report)

1. *Bug-report identifier*: specifies the unique identifier assigned to this report.
2. *Summary*: summarizes the (bug) incident by identifying the test items involved (with version/revision level) and by referencing the relevant documents (test-procedure specification, test-case specification, test log).

3. *Bug description*: provides a description of the incident, so as to correct the bug, repeat the incident or analyze it off-line:

- Inputs.
- Expected results.
- Actual results.
- Date and time.
- Test-procedure step.
- Environment.
- Repeatability (whether repeated; whether occurring always, occasionally or just once).
- Testers.
- Other observers.
- Additional information that may help to isolate and correct the cause of the incident; for example, the sequence of operational steps or history of user-interface commands that lead to the (bug) incident.

4. *Impact*: Priority of solving the incident / correcting the bug (urgent, high, medium, low).

3.4. Test-summary report

1 *Test-summary-report identifier*: specifies the unique identifier assigned to this report.
2. *Summary*: summarizes the evaluation of the test items, identifies the items tested (including their version/revision level), indicates the environment in which the testing activities took place, supplies references to the documentation over the testing process (test plan, test-design specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, …).

3. *Variances*: reports any variances/deviations of the test items from their design specifications, indicates any variances of the actual testing process from the test plan or test procedures, specifies the reason for each variance.
4. *Comprehensiveness assessment*: evaluates the comprehensiveness of the actual testing process against the criteria specified in the test plan, identifies features or feature combinations which were not sufficiently tested and explains the reasons for omission.
5. *Summary of results*: summarizes the success of testing (such as coverage), identifies all resolved and unresolved incidents.

6. *Evaluation*: provides an overall evaluation of each test item including its limitations (based upon the test results and the item-level pass/fail criteria).
7. *Summary of activities*: summarizes the major testing activities and events, summarizes resource consumption (total staffing level, total person-hours, total machine time, total elapsed time used for each of the major testing activities, …).
8. *Approvals*: specifies the persons who must approve this report (and the whole testing phase).

3.5. Inspection checklist for test plans:

1. Have all materials required for a test plan inspection been received?
2. Are all materials in the proper physical format?
3. Have all test plan standards been followed?
4. Has the testing environment been completely specified?
5. Have all resources been considered, both human and hardware/software?
6. Have all testing dependencies been addressed (driver function, hardware, etc.)?
7. Is the test plan complete, i.e., does it verify all of the requirements? (*For unit testing*: does the plan test all functional and structural variations from the high-level and detailed design?)
8. Is each script detailed and specific enough to provide the basis for test case generation?
9. Are all test entrance and exit criteria sufficient and realistic?

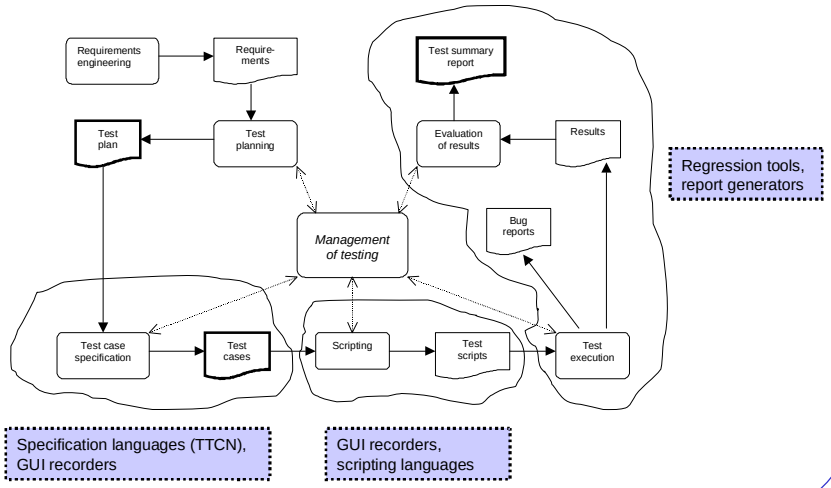
10. Are invalid as well as valid input conditions tested?
11. Have all pass/fail criteria been defined?
12. Does the test plan outline the levels of acceptability for pass/fail and exit criteria (e.g., defect tolerance)?
13. Have all suspension criteria and resumption requirements been identified?
14. Are all items excluded from testing documented as such?
15. Have all test deliverables been defined?
16. Will software development changes invalidate the plan? (*Relevant for unit test plans only*.)
17. Is the intent of the test plan to show the presence of failures and not merely the absence of failures?
18. Is the test plan complete, correct, and unambiguous?
19. Are there holes in the plan; is there overlap in the plan?
20. Does the test plan offer a measure of test completeness and test reliability to be sought?
21. Are the test strategy and philosophy feasible?

3.6. Inspection checklist for test cases:

1. Have all materials required for a test case inspection been received?
2. Are all materials in the proper physical format?
3. Have all test case standards been followed?
4. Are the functional variations exercised by each test case required by the test plan? (*Relevant for unit test case documents only*.)
5. Are the functional variations exercised by each test case clearly documented in the test case description? (*Relevant for unit test case documents only*.)
6. Does each test case include a complete description of the expected input, and output or result?
7. Have all testing execution procedures been defined and documented?
8. Have all testing dependencies been addressed (driver function, hardware, etc.)?
9. Do the test cases accurately implement the test plan?

10. Are all data set definitions and setup requirements complete and accurate?
11. Are operator instructions and status indicators complete, accurate, and simple?
12. Have all intercase dependencies been identified and described?
13. Is each condition tested once and only once?
14. Have all test entrance and exit criteria been observed?
15. Are the test cases designed to show the presence of failure and not merely the absence of failure?
16. Are the test cases designed to show omissions and extensions?
17. Are the test cases complete, correct, and unambiguous?
18. Are the test cases realistic?
19. Are the test cases documented so as to be 100 percent reproducible?
20. Has the entire testing environment been documented?
21. Has configuration management been setup, directories established, and have case data and tools been loaded?

3.7. Test automation



Why automation of testing

- *Test automation*: software that automates some aspects of the testing of a system (in some particular application area)
  - capabilities to generate test inputs and expected results, to run test suites without manual intervention, and to evaluate pass / no pass
  - enhances effective and repeatable testing
- Provides consistent and complete test summary reports
- Makes testing more objective and less dependent on the skills of an individual tester
- Especially useful in white-box testing, regression testing where (almost) the same tests are repeated as such, and performance / stress testing where the system is pushed beyond its design limits
  - analysis of code coverage (what parts of the program have been tested?)
  - validation of response times for a large number of (concurrent, real-time) transactions
  - running tests that exceed the maximum input rate

Limitations of test automation

- Expensive investment
  - high price and complexity of commercial tools
  - personnel costs: education, maintenance of testware (test cases, test suites, test harness = the testing environment)
- Not suitable for testing phases where the user has a central role (usability testing)
- Limited support for tasks that inherently cannot be automated
  - test project planning
  - test case design (exceptions: model-based testing, specification-based testing, conformance testing of communication protocols)
- Manual testing + automated testing an effective practice
  - for instance, manual test case design and black-box testing + automated analysis of code coverage that was reached

Testing tools

(a) Basic tools
- Graphical User Interface (GUI) testers (capture / record)
- Regression testers (automatic replay / playback)
- White-box code coverage analyzers
- Debuggers, dynamic tracers
- Report generators

(b) Advanced tools
- Load/stress testers, performance testers
- Mutation testers
- Application-specific testers (protocols, embedded systems, ...)

(c) Future tools
- Semi-automatic ”algorithmic” debuggers
- ”Regression scripters” for automated testware maintenance
- Self-testing and self-repairing programs

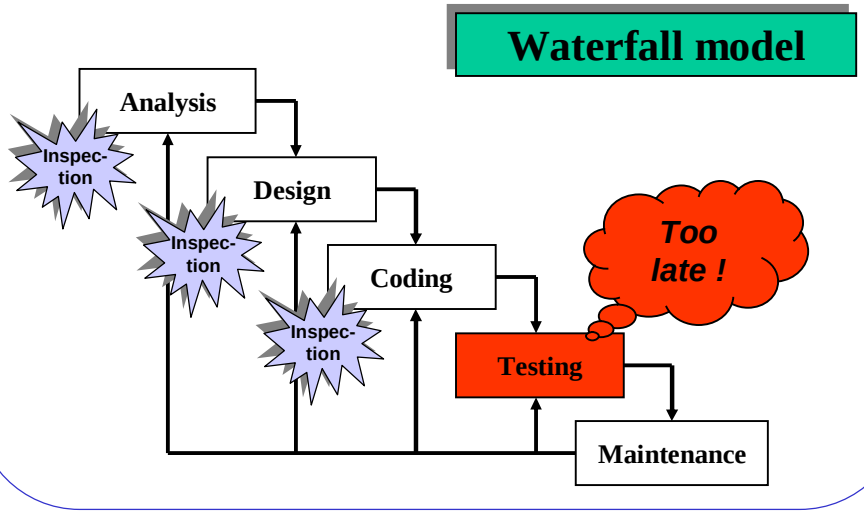
How to start with testing tools

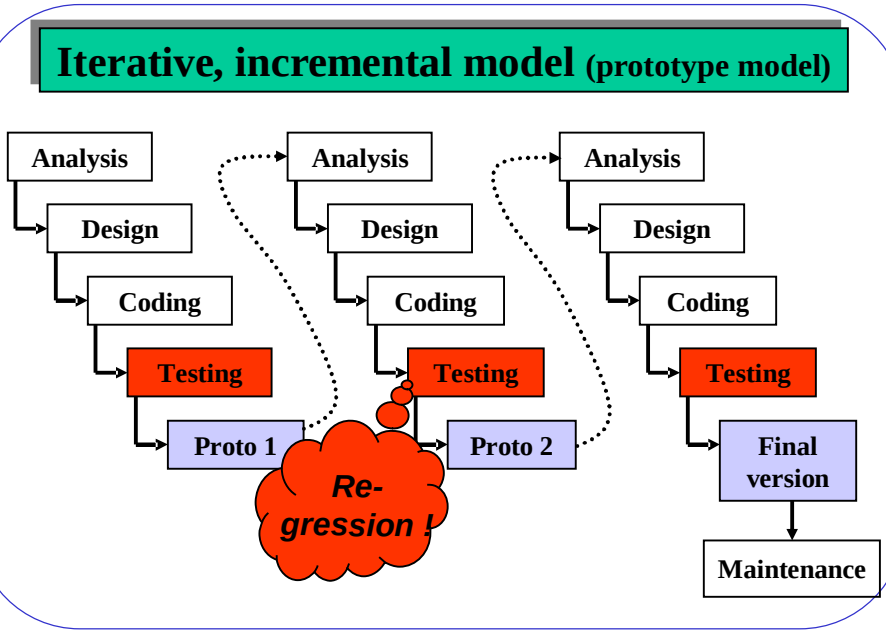
- Tool selection as a project
  - lots of commercial, super-expensive tools on the market
  - lots of marginal costs: licenses, training, testware maintenance, ...
- Selection, installation, and training before actual software development, to make test planning possible
- Pilot project!
  - in-house, in the operational environment
  - by own employees
  - with own target software
  - with own test data
  - with a realistic case
  - with specified goals for automation (effectiveness, quality, money, ...)

3.8. Testing organizations

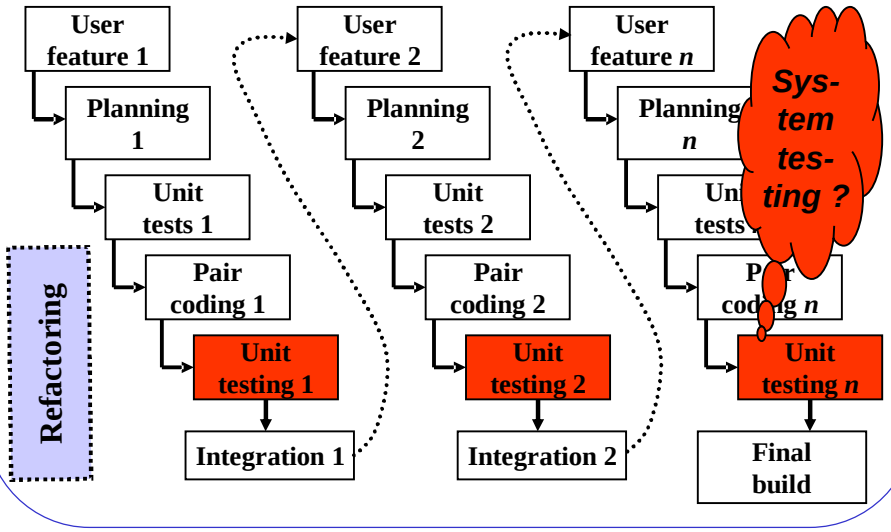
1. ***Testing is each person's responsibility***: the product developers are also responsible for testing their own code. Drawback: testing must be done independently; programmers are too biased by their own solutions and blind to their errors.
2. ***Testing is each group's responsibility***: the product developers within the project group test each other's code. Microsoft: "joint ownership and responsibility". Drawback: usually development efforts finally dominate over testing efforts.
3. ***Testing is performed by a dedicated resource***: there is a person in the project group who only concentrates on testing. Drawback: a person might be nominated who is less effective in development (and probably in testing as well).
4. ***Testing is performed by a test organization***: testing is a component of quality assurance with a dedicated team. Drawback: additional organizations and processes.

3.9. Testing in a software development process

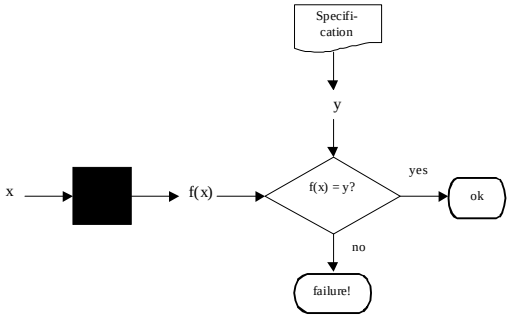




Lightweight process (Extreme Programming, XP)



4. Black-box testing



Principles

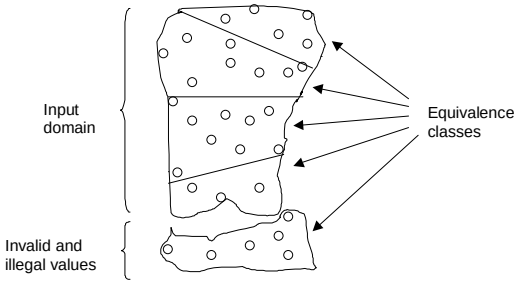
- Based on ***specifications*** and ***documents***
  - *requirements*
  - *technical plans, architectures*
  - *user manuals*
- Code not necessarily needed (while it certainly helps)
- General strategy, applies especially to integration testing, system testing, acceptance testing
- Can be assisted by a post-white-box testing phase, to obtain code coverage measures as indicators of testing quality

4.1. Domain partitioning: equivalence classes

System domain: set of all input values

Equivalence class: certain set of input values (subset of domain, subdomain)

***Equivalence classes*:**



§ each equivalence class represents a central property of the system

§ each value in an equivalence class makes the system behave "in the same manner"

- # in testing, ***each*** value reveals a *failure* or makes the system behave *ok*

§ each value activates (almost) the same execution path through the system

§ in black-box testing, based on system's specification and experience / intuition of tester

***Black-box testing hypothesis*:**

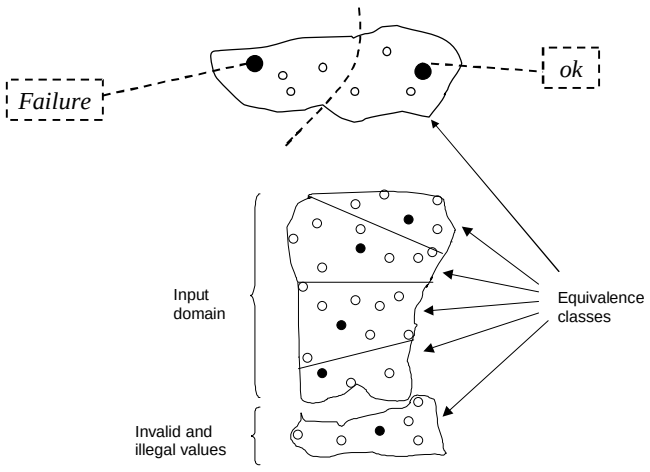
§ each value in an equivalence class results in correct execution when used as input to the system

OR

§ each value in an equivalence class results in failure when used as input to the system

§ for testing purposes, ***one*** representative input value from each equivalence class is enough !

§ in practice, the hypothesis does not hold universally, so the system shall be tested with *several* input values from each equivalence class



§ *range* of values ⇒ one valid and two invalid classes

"integer *x* shall be between 100 and 200" ⇒
{integer *x* | 100 ≤ *x* ≤ 200},
{integer *x* | *x* < 100},
{integer *x* | *x* > 200}

§ specific *value* within a range ⇒ one valid and two invalid equivalence classes

"value of integer *x* shall be 100" ⇒
{integer *x* | *x* = 100},
{integer *x* | *x* < 100},
{integer *x* | *x* > 100}

§ *set* of values ⇒ one valid and one invalid equivalence class

"weekday *x* shall be a working day" ⇒
x ∈ {Monday, Tuesday, Wednesday, Thursday, Friday},
x ∈ {Saturday, Sunday}

§ *Boolean* ⇒ one valid and one invalid equivalence class

"condition *x* shall be *true*" ⇒
x = *true*, *x* = *false*

§ one or several equivalence classes for *illegal* values, that is, for values that are incompatible with the type of the input parameter and therefore out of the parameter's domain

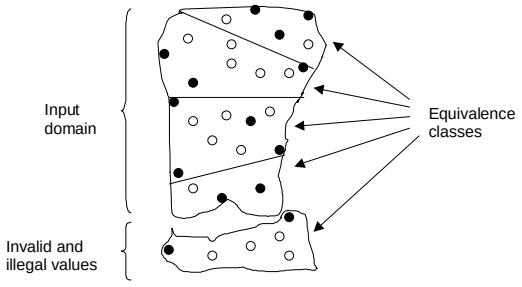
"integer values *x*" ⇒
{real-number *x*}, {character-string *x*}

§ if there is reason to believe that the system handles each valid/invalid/illegal input value differently, then each value shall generate an equivalence class

§ if there is reason to believe that the input values in an equivalence class are not processed in an identical manner by the system, the class shall be subdivided into smaller classes

4.2. Boundary analysis

Bugs often lurk at ***domain boundaries*** (verified by empirical studies on programming: most progamming errors are made with relational expressions <, >, <=, …)



Domain boundaries are generated by ***boundary conditions*** over the domain:

- *open boundaries*: generated by inequality operators (<, >)
- *closed boundaries*: generated by equality operators (=, ≤, ≥)

- *on point*: value that lies on a boundary
  - for open boundaries: the boundary value; for instance $x > \underline{0}$
- *off point*: value not on a boundary

- *1×1* (”one-by-one”) *domain testing strategy*: one on point and one off point for each domain boundary

Selection rules for on and off points:

- *open boundary*: one on point and one off point
  - on point: a value outside the domain ⇒ the condition is *false*
  - off point: a value inside the domain ⇒ the condition is *true*
- *closed boundary*: one on point and two off points (on both sides of the boundary, as close as possible)
  - on point: a value inside the domain ⇒ the condition is *true*
  - one off point: a value outside the domain ⇒ the condition is *false*
- nonscalar type: one on point and one off point
  - enumerations, Booleans, strings, complex numbers, …
  - on point: the condition is *true*
  - off point: the condition is *false*
  - the difference between on and off values should be minimized (for instance, for strings a single character difference)

§ *range* of values ⇒ two boundary conditions

"integer *x* shall be between 100 and 200" ⇒
{integer *x* | (*x* ≥ 100) ∧ (*x* ≤ 200)}

closed boundaries

– on points: 100, 200
– off points: 99, 101, 199, 201

§ strict inequality operator ⇒ open subdomain

"integer *x* shall be greater than 100" ⇒
{integer *x* | *x* > 100}

– on point: 100
– off point: 101

§ specific *value* ⇒ one (closed) boundary condition

"value of integer *x* shall be 100" ⇒ {integer *x* | *x* = 100}

– on point: 100
– off points: 99, 101

§ *set* of values ⇒ nonscalar type

"weekday *x* shall be a working day" ⇒
x ∈ {Monday, Tuesday, Wednesday, Thursday, Friday}

– on point: Friday, off point: Saturday

§ *Boolean* ⇒ nonscalar type

– on point: *true*, off point: *false*

4.3. The category-partition method

systematic black-box test design method based on equivalence partitioning of input

(1) Specification of input *categories*, "problem parameters"

Array sorting categories:

size of array
type of elements
maximum element value
minimum element value
position of maximum element in the array
position of minimum element in the array

(2) Division of categories into
choices = equivalence classes

Array sorting / choices for *size of array*:

size = 0
size = 1
$2 \leq size \leq 100$
size > 100
("*size* is illegal")

(3) *Test specification*:

§ A set of *test frames*: sets of choices, with each category contributing either zero or one choice.
§ A set of *test cases*: a single value from each of the choices in a test frame.

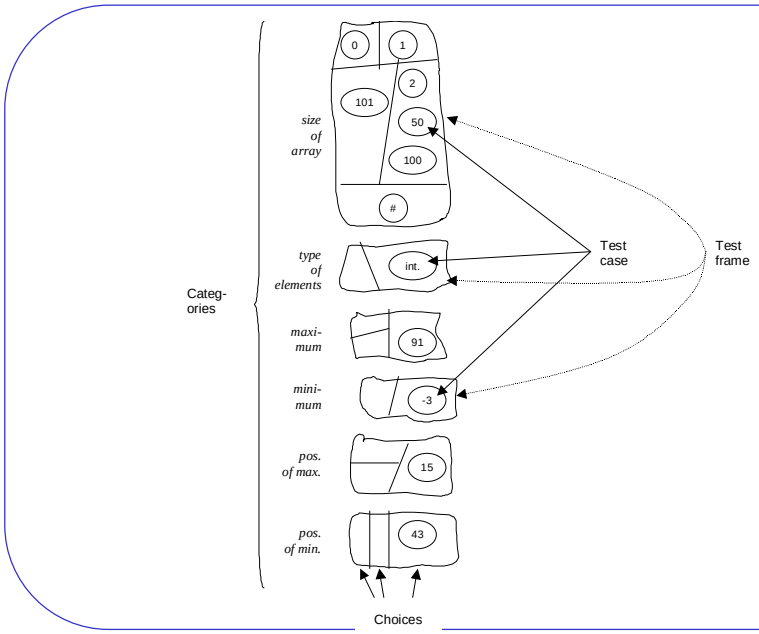
Array sorting / test case:

size of array = 50 (choice: 2 ≤ *size* ≤ 100)
type of elements = integer
maximum element value = 91
minimum element value = –3
position of maximum element in the array = 15
position of minimum element in the array = 43

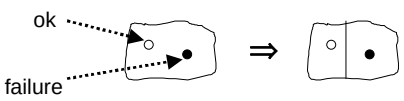
(4) Generation of test cases for the test frames into executable form (using a tool), combination into *test suites*.

(5) Storing the testware into a *test database*.

(6) Testing of the unit by the test cases, refinement of conflicting choices, maintenance of test database (using a tool).



Choice refinement:



4.4. System testing / user interface testing:

§ target: *operations* available at the (graphical) user interface

§ *parameters* of operations divided into equivalence classes

§ testing by all different *combinations* of equivalence classes (with one input value from each class)

§ testing of operation *sequences* (not independent)

§ based on user's manual

§ supported by tools (capture / replay)

Example:

***Find* (document, text, direction, match case)**

§ *document*: the current text file, subject to search

§ *text*: the character string to search for

§ *direction* (*down*, *up*): direction of the search with respect to current position of the cursor

§ *match case* (*yes*, *no*): whether or not the operation is case sensitive to letters

Equivalence classes:

§ *text*:
{strings with lower-case letters but without upper-case letters}
{strings with upper-case letters but without lower-case letters}
{strings with both upper-case and lower-case letters}
{strings with no letters}
{empty (illegal) strings}

§ *direction*: {*down*}, {*up*}

§ *match case*: {*yes*}, {*no*}

§ *document*: {text found}, {text not found}

Method: *exhaustive combination of equivalence classes*

text					*dir.*		*match*		*doc.*	
lc	*uc*	*luc*	*no-luc*	*ε*	*d*	*u*	*y*	*n*	*f*	*n-f*
x					x		x		x	
x					x		x			X
x						X	x		x	
x						x	x			X
	...									
	X					x	x			x
	x				x			x	x	
	...									
				x		x		x		x

***Number of* (*independent*) *combinations* = *Total number of tests* :**

$$E_1 * E_2 * \ldots * E_k$$

where E_i = number of equivalence classes for parameter i

Here: $5 * 2 * 2 * 2 = 40$

Note: some of the (invalid, illegal) combinations might be unexecutable, but that must be tested too!

Test case patterns (40):

§ *text*: lower-case, *direction*: down, *match case*: yes,
document: found **(1)**
§ *text*: lower-case, *direction*: down, *match case*: yes,
document: ***not found*** **(2)**
§ *text*: lower-case, *direction*: ***up***, *match case*: yes,
document: found **(3)**
§ *text*: lower-case, *direction*: up, *match case*: yes,
document: ***not found*** **(4)**
…
§ *text*: ***empty***, *direction*: up, *match case*: no,
document: not found **(40)**

Selection of test cases (40):

§ each pattern generates a test case
§ each equivalence class in a pattern is realized as an input value in the corresponding test case
§ in different test cases, different values are selected for the same equivalence class (better coverage)
§ boundary values are selected, when applicable
- for text, both short and long character strings
- for text, the whole character set

document		***text***	***direction***	***match case***
This **bea**utiful text	**(1)**	bea	down	yes
This beautiful text	**(2)**	beatles	down	yes
This **1bea**utiful text	**(3)**	1bea	***up***	yes
This 1Beautiful text	**(4)**	1bea	up	yes
This &**%1bEAu**tiful text	(5)	%1beau	down	no
This &%2beautiful text	(6)	%1beau	down	no
This **B**E utiful text	(7)	b	up	no
This BE utiful_text	(8)	beauti	up	no
This **BEA**UTIFUL text	(9)	BEA	down	yes
This_BEAUTIFUL text	(10)	BEAT	down	yes
THIS beautiFUL text	(11)	THIS	up	yes
THIS beatiful text	(12)	TH2S	up	yes
T**his** Beautiful Text	(13)	HIS	down	no
this %#& beautiful text	(14)	S	down	no
t**his %#&** beautiful text	(15)	HIS %#&	up	no
This %#&beautiful text	(16)	#& BE	up	no

document		*text*	*direction*	*match case*
This Beautiful **Text**	(17)	Text	down	yes
This Beautiful Text		Text	down	yes
TH**IS is** beautiful text		IS is	up	yes
This is beautiful text		IS is	up	yes
This **text 1**-99		ExT 1	down	no
This text 1 and text 2		eXt 1	down	no
This was beautiful_text		His Was Beauti	up	no
(This) (Was) (123text)	(24)	aS( )	up	no
123 one-two-three	(25)	123	down	yes
One-two-three 1-2-3		12-3	down	yes
This **&**007# mess		&	up	yes
This Bloody Mess		#%	up	yes
(This)_(was1) (was[**2**])		2]	down	no
0987654321!"#%&/*///		7654321#	down	no
1!2"3**#4$5%6&7/8(9)**0=oops		#4$5%6&7/8(9)	up	no
This %#&beautiful text	(32)	22	up	no
This is beautiful texT	(33)		down	yes
1 or two			down	yes
1 or two			up	yes
0K1+(8Those			up	yes
1 & 2			down	no
_			down	no
This %#&beautiful text			up	no
_	(40)		up	no

Example: print (file, copies, font, pagination)

Input parameters:

§ name of the file (must be provided)

§ -cn, where n is the number of copies ($1 \le n \le 100$);
default: n = 1

§ -fkm, where k indicates a font ($1 \le k \le 9$) and m indicates a mode (N for normal or B for **bold**);
defaults: k = 1, m = N

§ -np: no pagination (default: pagination shall be done)

Equivalence classes:

Related to file name:

1. Name of existing file given (*V*alid).
2. No file name given (*N*ot*V*alid).
3. Name of non-existing file given (*NV*).
4. "Name" does not follow the syntactic rules (*NV*).

Related to copies (-cn):

5. 1 ≤ n ≤ 100 (*V*).
6. Default: no n given (*V*).
7. n = 0 or n > 100 (*NV*).

Related to fonts (-fkm):

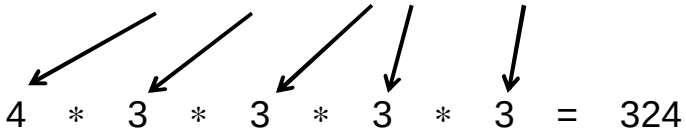
8. 1 ≤ k ≤ 9 (*V*).
9. Default: no k given (*V*).
10. m = N or m = B (*V*).
11. Default: no m given (*V*).
12. k = 0 or k > 9 (*NV*).
13. m other than N or B (*NV*).

Related to pagination (-np):

14. -np given (*V*).
15. -np not given (*V*).
16. Something else than -np given (*NV*). (This class covers also the other syntactically invalid -options.)

Number of exhaustive combinatory test cases:

print **file [-cn] [-f k m] [-np]**



This might be too many, so a method reducing the number of test cases is needed.

print **file [-cn] [-fkm] [-np]**

Optimizing principle:
- one test case for each *NV* equivalence class
- each equivalence class covered by *at least one* test case

(a)	-c5 -np	
(b)	xxyy -c3	(no file *xxyy* in directory)
(c)	#%$file5.3	
(d)	myfile -c0	(file *myfile* is in directory)
(e)	myfile -f100N	
(f)	myfile -f2H	
(g)	myfile -c5 -f1 -hjk	

Test cases / Eq. classes	(a)	(b)	(c)	(d)	(e)	(f)	(g)
1				√	√	√	√
2	X						
3		X					
4			X				
5	√	√					√
6			√		√	√	
7				X			
8						√	√
9	√	√	√	√			
10					√		
11	√	√	√	√			√
12					X		
13						X	
14	√						
15		√	√	√	√	√	
16							X

***Ok* : just 7 test cases**

However:

no actual task

with *valid*

parameters tested !!

Extending principle: combinations over the *number* of parameters

§ name of existing file always given

§ a test case where all the parameters are missing (0 present)

§ a test case for each individual parameter (1 present)

§ each parameter included in the set of pairs (2 present)

§ each parameter included in the set of triplets (3 present)

§ all the parameters given (4 present)

print **file [-cn] [-fkm] [-np]**

(h)	myfile	(none present)
(i)	myfile –c1	(n present)
(j)	myfile –f9	(k present)
(k)	myfile –fB	(m present)
(l)	myfile –np	(-np present)
(m)	myfile –f1N	(k, m present)
(n)	myfile –c100 –np	(n, -np present)
(o)	myfile –c50 –f5 –np	(n, k, -np present)
(p)	myfile –c1 –fB –np	(n, m, -np present)
(q)	myfile –c99 –f2N –np	(all present)

Test cases / Eq. classes	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)	(q)
1	√	√	√	√	√	√	√	√	√	√
2										
3										
4										
5		√					√	√	√	√
6	√		√	√	√	√				
7										
8			√			√		√		√
9	√	√		√	√		√		√	
10				√		√			√	√
11	√	√	√		√		√	√		
12										
13										
14					√		√	√	√	√
15	√	√	√	√		√				
16										

In total:

17 test cases

– 10 valid ones

– 7 invalid ones

User interface errors (preventing use of the system):

1. ***Functionality***. Something one reasonably expects the system to do is hard, confusing, or impossible.
– *Excessive functionality*: the system tries to do too much, and is therefore hard to learn and use.
– *Inadequacy for the task at hand*: a key feature isn't there at all, or is too restricted or too slow in which case the system cannot be used for real work. For instance, a database management system that takes eight hours to sort 100 records does provide the sorting feature but in a totally unusable form.
– *Missing function*: a major function is not available even though it is documented in a specification or user's manual, or is obviously desirable (such as "save" and "print" in a word processor).
– *Wrong function*: a function should do one thing (according to a specification or user's manual) but it does something else.
– *Functionality to be created by the user*: the system supplies all the basic capabilities but the actual functions to be used must be assembled from primitives by the user.

2. *Communication*. Information flow from the system to the user is somehow imperfect.
– *Missing information*: some relevant information is not available on the screen (or other interfacing device), examples being operating instructions, on-line help, activity and state information, and acknowledgement of finished operations.
– *Wrong, misleading, or confusing information*: the relevant information is available, but it cannot be trusted since it looks suspicious or has been erroneous in earlier phases of execution.
– *Display bugs*: the information is there, but in a wrong place or partly hidden.

3. *Command structure and entry*. The system organizes the available commands from the user to the system improperly.
– *Inconsistencies*: in different situations, the same command is presented in a different format (abbreviated, with different name, positioned differently, in different syntax, ...).
– *Excessive commands*: a command is provided to the user, even though it actually cannot be activated in that particular situation.
– *Missing or incomplete dialogs*: parameters of a command should be given to the system via a dialog at the graphical user interface, but the dialog is not available at all or it lacks some of the parameters. The dialog may also be inert by not accepting any keystrokes by the user.

4. *Missing commands*. A command is not available, even though it should be there .
– *Corrupted name*: the name of a command (provided in some kind of a menu) is simply mis-spelled, e.g., “Inster” instead of “Insert”.
– *Misleading name*: a command is not doing what it logically should do according to its name; e.g. the command “Save” might save the resources of the computer by immediately shutting it down (instead of storing the contents of a file in some secure place).
– *State transitions*: the user cannot reach every viable state of the system (commands cannot be stopped, execution cannot be aborted, ...).
– *Disaster prevention*: the system does not minimize the consequences of failures (no back-ups, no general undo mechanism, no incremental saves, ...).

5. *Performance*. The system is too slow, consumes too much memory, or severly limits the size of processed input.
– *Slow system*: an operation is not usable in practice because it takes too much time to execute (slow echoing of commands, no warning that an operation will take a long time, too long or short time-outs over data entries by the user, ...).
– *System out of memory*: the system cannot fulfil its task because it has already consumed all the memory available in the computer / device.
– *Insufficient user throughput*: the amount or quantity of data provided from the user as input to the system exceeds the system's processing capabilities (and there are no documented restrictions on the volume of input).

6. *Output*. The system output is erroneous or in a wrong format.
– *Incorrect output*: the system exhibits a failure.
– *Certain data missing*: the system does not provide all output data the user is interested in (according to a specification or user's manual).
– *Format incompatible*: the output data of the system (or a subsystem) should be given as input to a follow-up process, but the passing of data fails due to an incompatible format.
– *Layout misleading*: the output data is represented in a format that is not properly understood by the user (and the format cannot be dynamically customized).

5. White-box testing

$0 + 1 + 2 + \ldots + i, \quad i \in [0, 100]$:

```
read(i);
if ((i < 0) || (i > 100))
   error();
else
   { sum=0; x=0;
   while (x < i)
      { x=x+1;
      if (i==10) sum=1; else sum=sum+x; }
   print(sum); }
```


Black-box *test cases* (without looking at source code):

$i = -1$	OK
$i = 0$	OK
$i = 1$	OK
$i = 50$	OK
$i = 99$	OK
$i = 100$	OK
$i = 101$	OK

However:

$i = 10$ $\Rightarrow$ ***FAILURE!*** **(sum=1)**

***White-box testing principles*:**

§ details of source code analyzed
§ design of test cases on the basis of code structure

§ *execution path*: a certain sequence of program statements, executed when starting the program with a certain input (test case)
§ different test cases => different execution paths

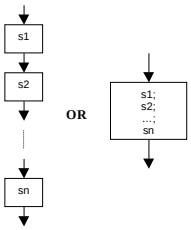
§ *control-flow testing*: based on the execution order of the statements
§ *data-flow testing*: based on the processing of the data during execution

§ (*control*) *flow graph*: abstraction of the program's control flow, in graphical form
§ *data-flow graph*: abstraction of the program's data flow (for a certain input variable), in graphical form; usually extension of control-flow graph

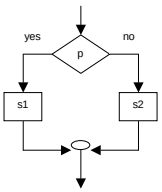
§ control-flow graph, data-flow graph automatically produced
§ test cases designed from the graphs

§ *coverage*: the relative amount of statements (and others) executed during testing, computed from control-flow / data-flow graph

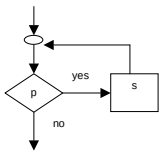
Flow-graph structures:



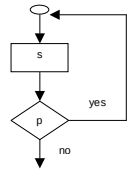
(a) *Statement sequence*:
s1; s2; …; sn



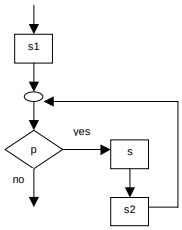
(b) *Conditional* (*if*) *statement*:
if (p) **then** s1 **else** s2



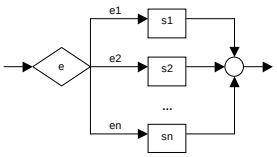
(c) *Loop* (*while-do*) *statement*:
while (p) **do** s



(d) *Loop* (*do-while*) *statement*:
do s **while** (p)



(e) *Iterative* (*for*) *statement*:
for (s1; p; s2) **do** s



(f) *Switch* (*case*) *statement*:
switch (e) {**case** e1: s1;
case e2: s2; …;

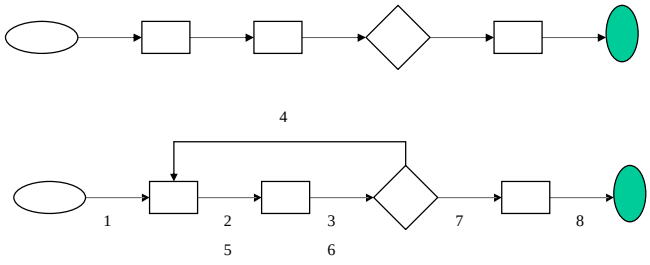
5.1. Control-flow testing

Coverage: how extensively the program has been (or will be) tested with a given set of test cases

§ the (relative) number of *nodes* (statements) in the flow graph executed during testing
§ the (relative) number of *edges* (control transitions) in the flow graph traversed during testing

§ ***statement coverage***: each node (statement) has to be executed at least once
§ ***branch coverage***: each edge (transition) has to be traversed at least once
§ a large number of variations of different coverage power
§ special target: ***loop testing***

Definition 5.1. An ***execution path*** is a sequence of nodes (and connecting edges) from the unique begin-node of the flow graph to the unique end-node of the graph.
[A certain execution of the underlying program.]
[May contain the same node several times: loops.]

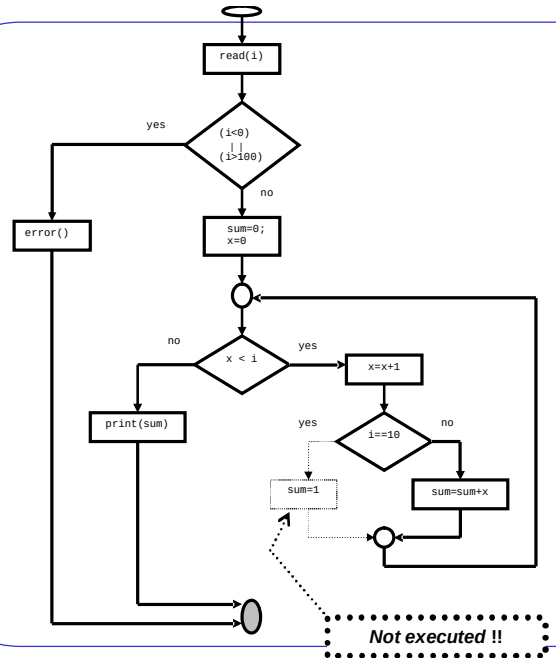


Definition 5.2. A set *P* of execution paths satisfies the ***statement coverage criterion*** if and only if for all *nodes n* in the flow graph, there is at least one path *p* in *P* such that *p* contains the node *n*.
[Each statement of the program is executed at least once during testing, by some test case.]

– criterion met => *complete* (100%) statement coverage
– criterion not met => partial statement coverage (< 100%)
– begin-node, end-node, junctions excluded
– complete coverage surprisingly hard to achieve in practice
– "dead code" / conditional compilation

Definition 5.3. A set *P* of execution paths satisfies the ***branch coverage criterion*** if and only if for all *edges e* in the flow graph, there is at least one path *p* in *P* such that *p* contains the edge *e*.
[Each control-flow branch / decision (*true / yes*, *false / no*) is taken at least once during testing, by some test case.]

– criterion met => *complete* (100%) branch coverage
– complete branch coverage => complete statement coverage (branch coverage *subsumes* statement coverage)
– usually more tests are needed for complete branch coverage than for complete statement coverage
– branch coverage is more extensive: the criterion is stronger than the statement coverage criterion
– criterion not met => partial branch coverage (< 100%)



i = -1

i = 0

i = 1

i = 50

i = 99

i = 100

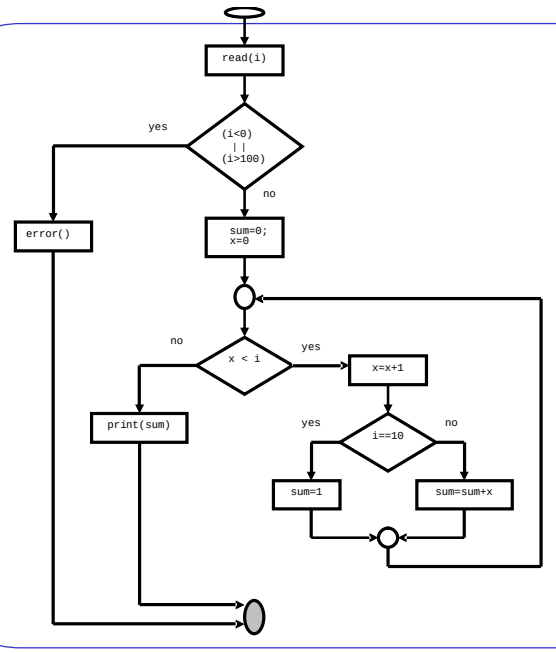
i = 101

Statement coverage:

9 / 10 = 90%

Branch coverage:

13 / 15 = 87%



i = -1

i = 1

i = 10

Statement coverage:

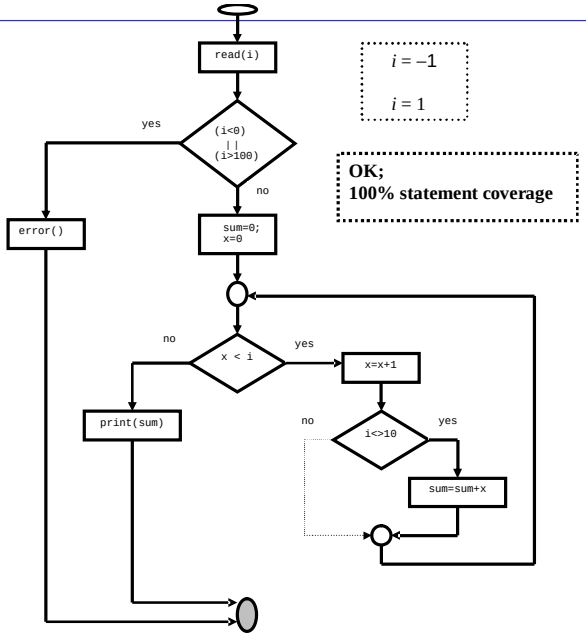
10 / 10 = 100%

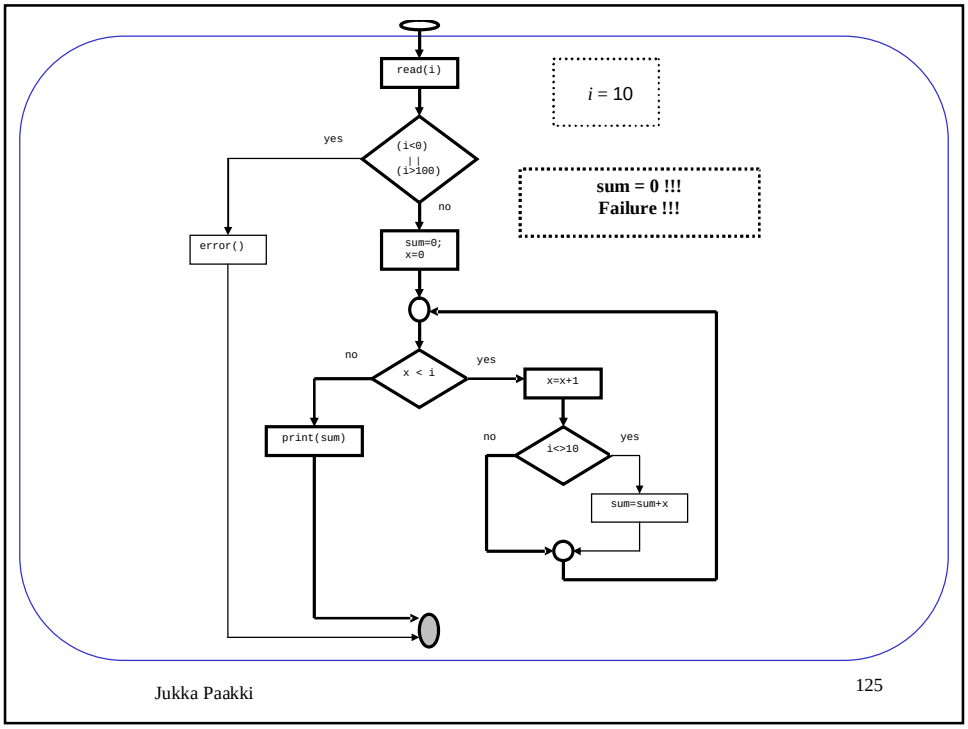
Branch coverage:

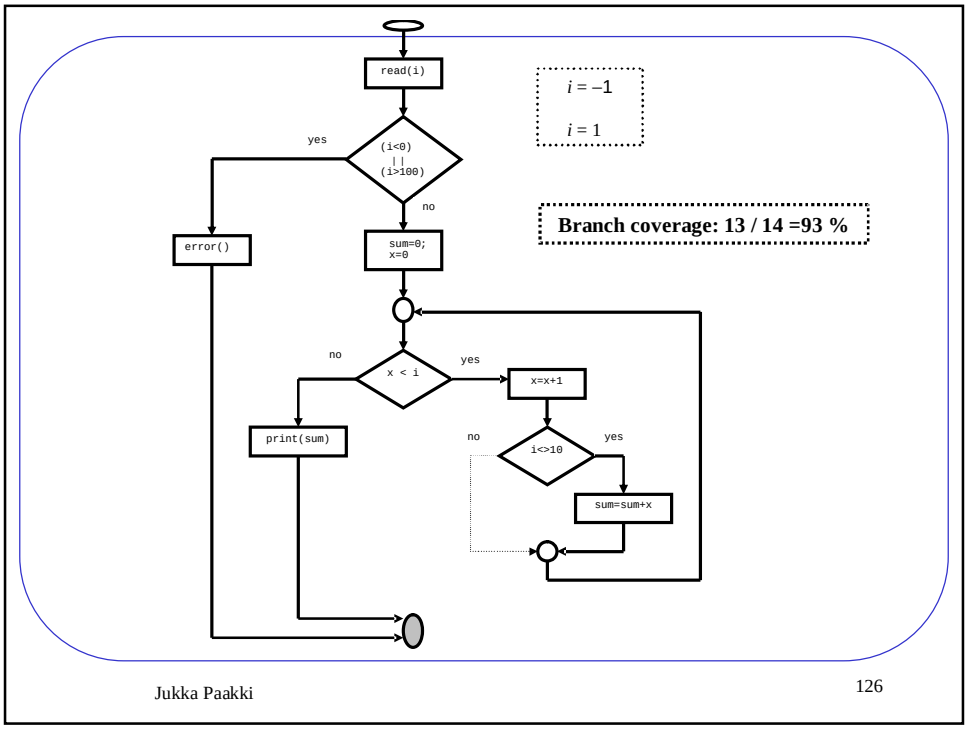
15 / 15 = 100%

***Statement coverage ≠ Branch coverage*:**

```
read(i);
if ((i < 0) || (i > 100)) error() else
{ sum=0; x=0;
while (x < i)
{ x=x+1; if (i <> 10) sum=sum+x; }
print(sum); }
```

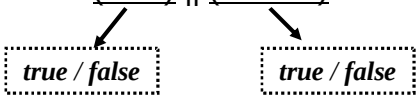




read(i)
i = 10
yes
(i<0)
||
(i>100)
no
error()
sum=0;
x=0
sum = 0 !!!
Failure !!!
no
x < i
yes
x=x+1
print(sum)
no
i<>10
yes
sum=sum+x


read(i)
i = –1
i = 1
yes
(i<0)
||
(i>100)
no
error()
sum=0;
x=0
Branch coverage: 13 / 14 =93 %
no
x < i
yes
x=x+1
print(sum)
no
i<>10
yes
sum=sum+x

Definition 5.4. A set *P* of execution paths satisfies the ***condition coverage criterion*** if and only if for every control node in the flow graph consisting of atomic predicates (c_1, c_2, …, c_n), c_i yields *true* (*yes*) when evaluated within a path p_1 in *P* and c_i yields *false* (*no*) when evaluated within a path p_2 in *P*, i = 1, 2, …, *n*.

– internal structure of composite control predicates taken into account: (i < 0) || (i > 100)



– each atomic predicate separately tested

Definition 5.5. A set *P* of execution paths satisfies the ***multicondition coverage criterion*** if and only if for every control node in the flow graph consisting of atomic predicates (c_1, c_2, …, c_n), each possible *combination* of their truth values (*true/yes*, *false/no*) is evaluated within at least one path *p* in *P*.

– stronger requirement than for condition coverage

– all the *combinations*

– for 2 atomic predicates: (*true*,*true*), (*true*,*false*), (*false*,*true*), (*false*,*false*)

– for 3 atomic predicates: 8 combinations, etc….

Definition 5.6. The ***path coverage criterion***.

Definition 5.7. The ***independent path coverage criterion***.

§ Path coverage the strongest criterion

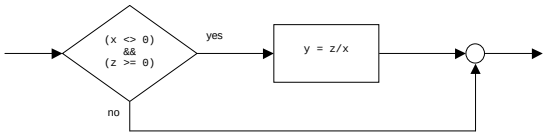
– usually impossible to reach

§ Independent path coverage stronger than branch coverage

– used also in complexity analysis of programs

***statement coverage vs. branch coverage vs. condition coverage vs. multicondition coverage*:**

```
if ((x <> 0) && (z >= 0)) y = z/x
```



§ Complete statement coverage: ($x = 1$, $z = 0$) **[1 test]**

§ Complete branch coverage: ($x = 1$, $z = 0$) for the *yes*-branch, ($x = 1$, $z = -1$) for the *no*-branch **[2 tests]**

§ Complete condition coverage: ($x = 0$, $z = 0$) for the combination (*false*, *true*), ($x = 1$, $z = -1$) for the combination (*true*, *false*) (*yes*-branch unexplored !) **[2 tests]**

§ Complete multicondition coverage: ($x = 1$, $z = 0$) for the combination (*true*, *true*), ($x = 1$, $z = -1$) for the combination (*true*, *false*), ($x = 0$, $z = 0$) for the combination (*false*, *true*), ($x = 0$, $z = -1$) for the combination (*false*, *false*) **[4 tests]**

Selection of test cases

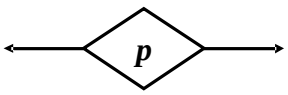
A certain code coverage criterion and percentage (e.g. 100% branch coverage) has been chosen $\Rightarrow$ one has to design test cases to reach the criterion.

1. Construct a flow graph for the program (with a white-box tool).
2. Choose the execution paths that satisfy the criterion.
3. For each execution path, design a test case (program input) that activates a traversal of the path.
4. Execute the tests (with the white-box tool that calculates the coverage).
5. If the required coverage has not been reached, return to step 3 to design additional test cases for those execution paths that have not been traversed yet during testing.

The process of designing a test case for a particular execution path is called ***path sensitization***:

- In general, path sensitization is undecidable: there is no algorithm that can find a suitable test case for each possible path.
- Symbolic execution and equation solving tools succeed in some cases.
- A heuristic: Begin with the control conditions of a branch at the *end* of the path. Select such variable values that will satisfy these conditions. Repeat this analysis for each prior branch in the path until you reach the entry node of the flow graph. Use the selected values of the input variables as the test case for the path.
- There may be infeasible paths that cannot be executed with any input, caused by short-circuit evaluation, contradictory or mutually exclusive control conditions, redundant control predicates, or dead code

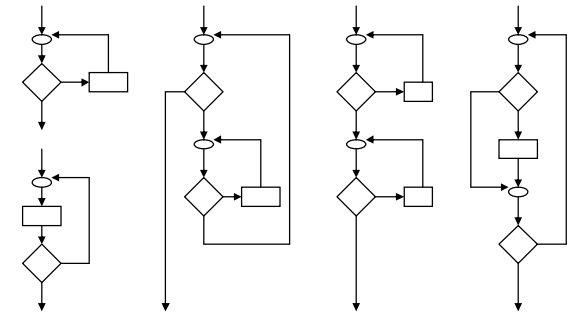
§ the crucial points in a flow-graph are those where the execution diverges, that is, the ***control predicates*** of branches



§ one has to find the input values such that when executing the program with the input, control branches into the desired direction and the predicate *p* obtains the corresponding value (*true / false*) or value combination

- *note* 1: *p* may depend on the input just indirectly
- *note* 2: it may not be possible to obtain all the required truth values for *p*: ((x == 1) && (x==2))

5.2. Loop testing



(a) Simple loops (b) Nested loops (c) Serial loops (d) Unstructured loops

Basic coverage-based white-box methods do not take into account the inherent iterative nature of loops

- Testing of simple loops: 0 iterations (no looping), *minimum* number of iterations (possibly 0), *minimum*+1 iterations, *typical* number of iterations, *maximum*-1 iterations, *maximum* number of iterations, *maximum*+1 iterations (should not be feasible)
  - note: loops with fixed iteration control may not be executable (testable) with all the suggested iteration patterns

    ```
    for ( j=0; j < 999; ++j ) { … }
    ```

- Testing of serial loops:
  - if there is no data-flow relationship between the loops, test them both as simple loops
  - if there is a data-flow relationship between the loops, test them as if the loops were nested
- Testing of unstructured (”spaghetti”) loops: test the loop with an equivalent simple / serial / nested loop as model
  - spaghetti code should be rewritten into structured form, for testing as well as for maintenance purposes

§ Testing of nested loops:

– There would be too many tests when repeating all the inner loop tests every time an outer loop is iterated, so:

1. The innermost loop is tested first using the simple-loop strategy. The other loops are iterated their minimum number of times.
2. Set up the looping conditions of the previously tested loop such that it will be iterated a suitable number of times (minimum, typical, or maximum).
3. Proceed to testing the outer loop which is nesting the previously tested one, using the simple-loop strategy. (The outer loops are iterated their minimum number of times, the inner loops are iterated their suitable number of times.)
4. Repeat the steps 2 and 3, until the outermost loop has been tested.
5. Set up a test that will iterate all loops their maximum number of times.

5.3. Data-flow testing

Events on data (variables):

§ ***d*** (*defined*): the variable *gets a value* by an assignment statement, initialization, input statement, …
§ ***k*** (*killed*): the variable gets undefined, e.g., by deallocation of its space. [Of minor importance.]
§ ***u*** (*used*): the value of the variable is referred to and used. If necessary, the uses can be classified further:
– ***c*** (*computation*): the value of the variable is used in *computation*, typically in the expression on the right-hand side of an assignment.
– ***p*** (*predicate*): the value of the variable is used in *a control predicate* for branching the execution flow, typically in an if-statement, while-statement, for-statement, or a switch-(case-)statement.

***Data-flow anomalies*:**

...; **x**:= 1; y:= y+10; read(z); z:= **x**+y+z;
d ***u***

§ *dd*: suspicious, but may be harmless: `x:= 1; x:= 2`.
§ *dk*: probably a *bug* (a data-flow "anomaly"); why define the variable without using it at all?
§ *du*: the normal case; the variable is defined and then used.
§ *kd*: normal situation; the variable is killed and then redefined.
§ *kk*: harmless, but probably a *bug*; why kill the variable twice?
§ *ku*: a *bug*; the variable is undefined, so its value cannot be used.

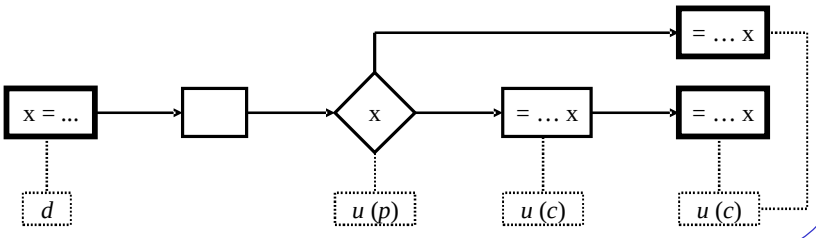
§ *ud*: normal situation (in imperative languages):
`y:= x+1; x:= 10` or `x:= x+1`.
§ *uk*: normal situation.
§ *uu*: normal situation: `y:= x+x`.
§ *–d*: the normal case.
§ *–k*: probably a *bug*; the variable is killed even though it does not exist.
§ *–u*: probably a *bug*; the variable is used without a value.
§ *d–*: suspicious; why give a value that will never be used?
§ *k–*: normal situation.
§ *u–*: usually the normal case. (However, if the variable is dynamic, it should also be killed and the space for it should be deallocated.)

***Data-flow testing methods*:**

§ *data-flow graph* with respect to a certain variable:
annotation of the control-flow graph
§ *annotation*: coding of the nodes with the symbols
d, *k*, *c*, *p* (*u*)
§ several forms of *coverage*: how extensively the sequences of *d*, *k*, *c*, *p* (*u*) are exercised during testing
§ stronger than control-flow methods:
more testing needed to reach a certain coverage

Definition 5.8. A *definition-clear path* with respect to a variable *x* is a path where no node contains a definition occurrence (*d*) of *x*.

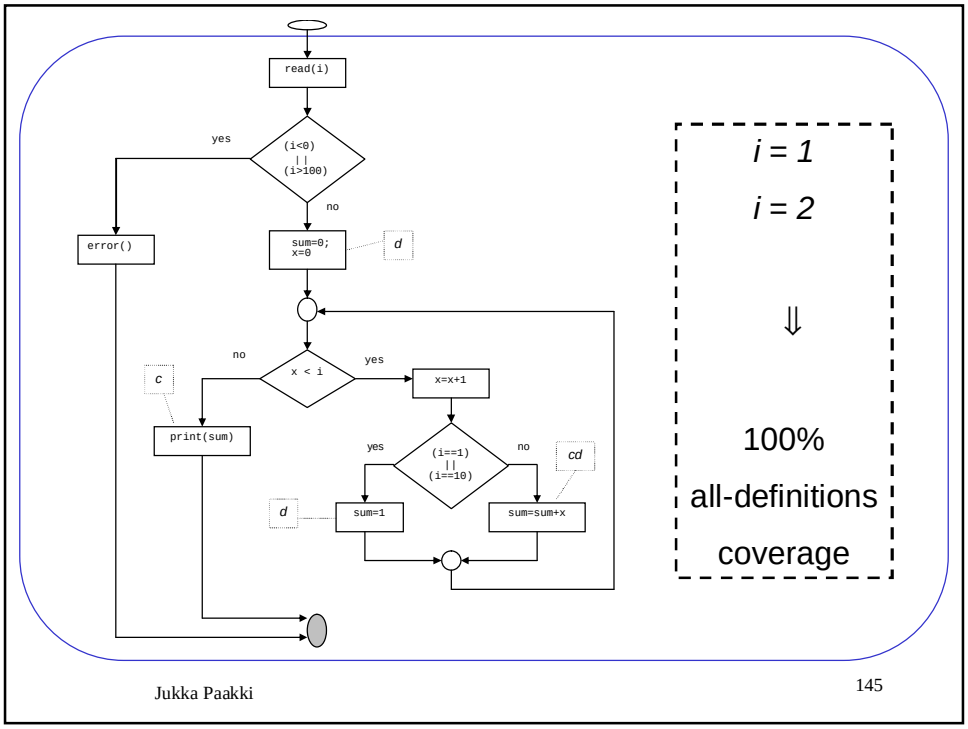
Definition 5.9. A definition occurrence (*d*) of a variable *x* at a node *n reaches* a use occurrence (*u*, *c*, *p*) of *x* at a node *v*, if and only if there is a path of nodes (n, w_1, w_2, …, w_m, v) from *n* to *v*, and (w_1, w_2, …, w_m) is definition-clear with respect to *x*.

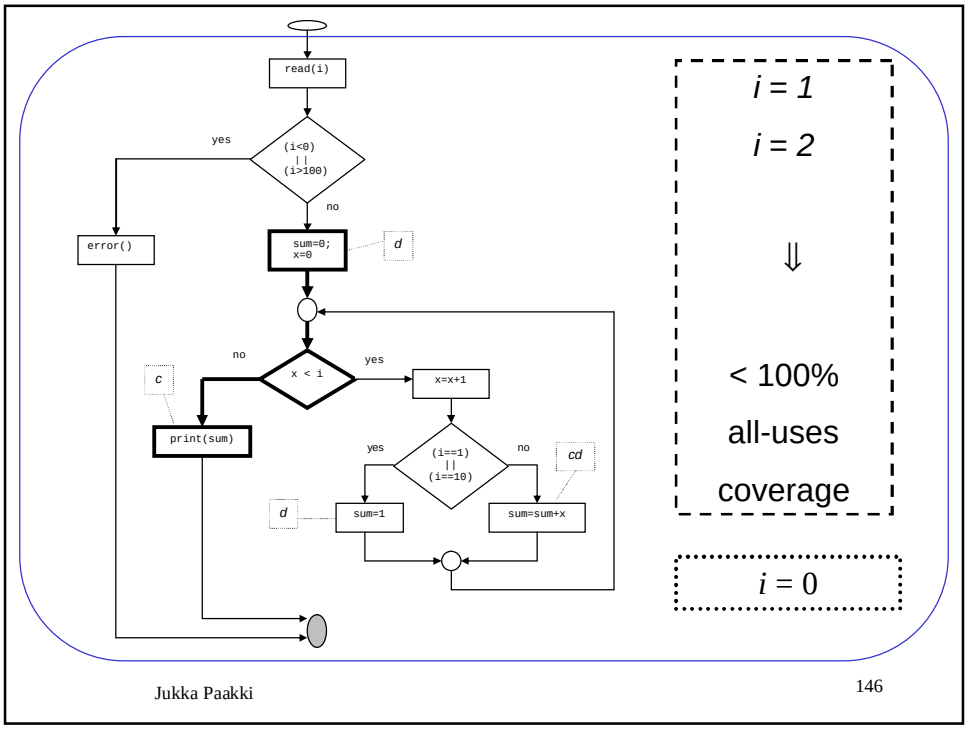


Definition 5.10. A set *P* of execution paths satisfies the ***all-definitions criterion*** (with respect to a variable *x*) if and only if for all definition occurrences (*d*) of *x*, there is at least one path in *P* that includes a subpath through which the definition of *x* reaches ***some*** use occurrence (*u*, *c*, *p*) of *x*.

Definition 5.11. A set *P* of execution paths satisfies the ***all-uses criterion*** (with respect to a variable *x*) if and only if for all definition occurrences (*d*) of *x* and ***all*** use occurrences (*u*, *c*, *p*) of *x* that the definition reaches, there is at least one path in *P* that includes a subpath through which that definition reaches the use.


```
read(i);
if ((i < 0) || (i > 100))
  error() else
  { sum=0; x=0;
  while (x < i)
     { x=x+1;
     if ((i == 1) || (i == 10)) sum=1
     else sum=sum+x; }
   print(sum); }
```

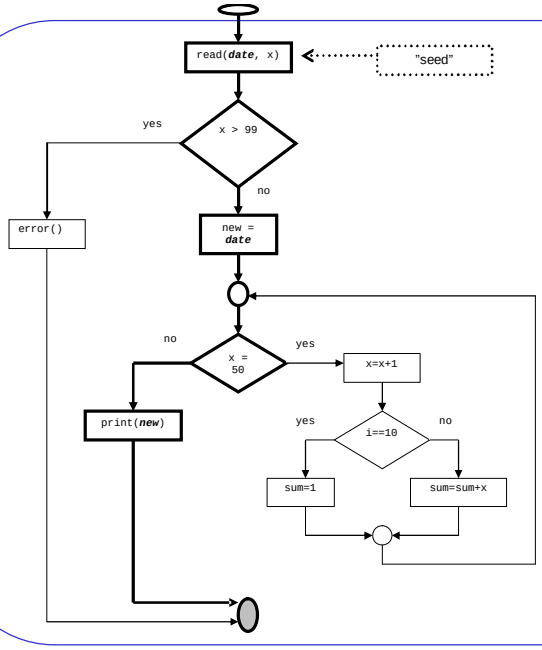

read(i)
yes
(i<0)
||
(i>100)
no
error()
sum=0;
x=0
d
no
x < i
yes
x=x+1
c
print(sum)
yes
(i==1)
||
(i==10)
no
cd
d
sum=1
sum=sum+x
i = 1
i = 2
⇓
100%
all-definitions
coverage


read(i)
yes
(i<0)
||
(i>100)
no
error()
sum=0;
x=0
d
no
x < i
yes
x=x+1
c
print(sum)
yes
(i==1)
||
(i==10)
no
cd
d
sum=1
sum=sum+x
i = 1
i = 2
⇓
< 100%
all-uses
coverage
i = 0

Definition 5.12. The ***all-du-paths*** criterion.

Definition 5.13. The ***all-p-uses / some-c-uses*** criterion.

Definition 5.14. The ***all-c-uses / some-p-uses*** criterion.

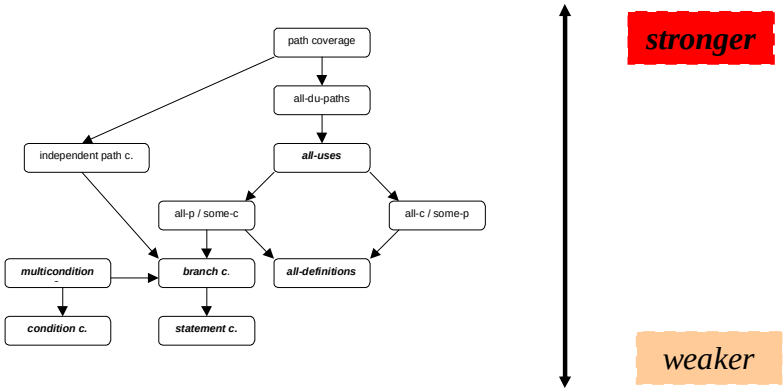


Technique:

automatic data-flow

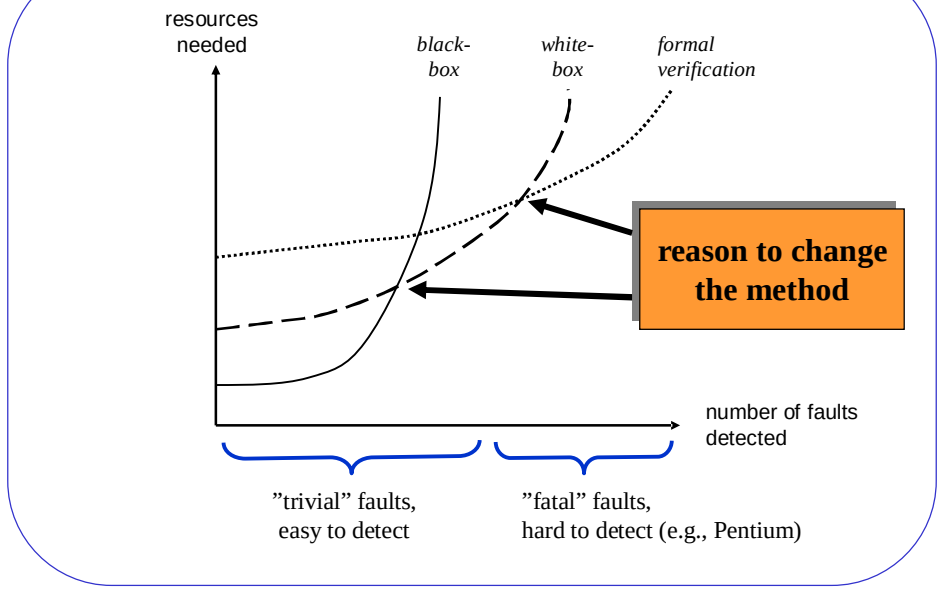
slicing

5.4. Subsumption among white-box methods



5.5. Main differences between *black-box* and *white-box*:

§ *white-box*: source code needed
§ *black-box*: test cases obtained directly by design
§ *white-box*: test cases obtained by separate code analysis
§ *black-box*: less systematic and "formal", more ad-hoc and based on intuition ⇒ technically easier, but more risky
§ *white-box*: strong theoretical background, automation possibilities (code coverage)
§ *black-box*: no standard notion for internal testing quality ("coverage") [input space / functionality / features ?]
§ *black-box*: more commercial testing tools
§ *black-box*: more common in industry



Jukka Paakki

151