System Technical and organizational means for the autonomous fulfillment of a task (Birolini) A system can consist of hardware, software, people (service and maintenance personnel) and logistic assistance **Technical System** System where influences by people and logistics are ignored. **Quality** Degree in which the inherent attributes of an entity fulfill quality requirements **Quality Requirement** Expectation or demand defined (by a customer) that is generally assumed or mandatory **Quality Characteristic** Property of an entity on the basis of which its quality is described and estimated, but which makes no statement about the degree of fulfillment of the characteristic **Quality characteristic** can be refined incrementally into partial characteristics • Inherent attribute of a process, product or a system that relates to a quality requirement. **Quality Measure** Allows to draw conclusions on the fulfillment of specific quality characteristics. MTTF is a quality measure of the quality characteristic Reliability. **Safety** • State where the danger of a personal or property damage is reduced to an acceptable value• Birolini defines safety as a measure for the ability of an item to endanger neither persons, property nor the environment **Safety analysis** aims at proving that the actual risk is below the acceptable risk **Technical Safety** Measure for the ability of a failure afflicted item to endanger neither persons, property nor the environment **Correctness** •Correctness has a binary character •A fault-free realization is correct •An artifact is correct if it is consistent to its specification •If no specification exists for an artifact, correctness is not defined **Completeness** • A system is functional complete, if all functions required in the specification are implemented. This concerns the treatment of normal cases as well as the interception of failure situations **Robustness** •Property to deliver an acceptable behavior also in exceptional situations •A correct system •Accordingly, robustness is rather a property of the specification than of the implementation •Robustness has a gradual character. **Reliability** •Part of the quality with regard to the behavior of an entity during or after given time periods with given working conditions •Measure for the ability of an item to remain functional, expressed by the probability that the required function is executed failure-free under given working conditions during a given time period. MTBF **Availability**-Measure for the ability of an item to be functional at a given time. **(MTBF/ (MTBF + MTTR))** **Failure** Inconsistent behavior w.r.t. specified behavior while running a system (happens dynamically during the execution) => Each failure has a timestamp e.g., **malfunctioning** **Fault, defect**: Statically existent cause of a failure, i.e., a **bug** (usually the consequence of an error made by the programmer) **Error**: Basic cause for the fault (e.g., **misunderstanding of a particular statement of the programming language**) **Accident** is an undesired event that causes death or injury of persons or harm to goods or to the environment **Hazard** is a state of a system and its environment where the occurrence of an accident depends only on influences that are not controllable by the system **Risk** is the combination of hazard probability and severity of the resulting accident **Acceptable Risk** is a level of risk that authorities or other bodies have defined as acceptable according to acceptance criteria **Influences among quality characteristics**: • **Safety– (-) – Availability**: In systems with stable safe state or fail safe state (e.g. trains) • **Availability – (undefined/+) – Safety**: Availability of safety related functions (e.g. heart pacemaker) has positive influence on safety•• **Safety – (undefined) – Reliability** •**Reliability – (+) – Safety**: Reliability of safety related functions (e.g. reverse thrust) • **Reliability – (+) – Availability**: Longer time without failure Better up-time/downtime ratio • **Availability – (-) – Reliability**: System might become unreliable if maintenance is not being done properly (short repair time). • **Efficiency – (-) – Safety**: When resources that improve safety have to be left out (e.g. redundant components), so that the system is more efficient. • **Safety – (-) – Efficiency**: When safety related functions considerably consume machine resources and computation time takes long. (e.g. plausibility checks) • **Efficiency – (-) – Reliability**: The lack of resources at runtime might lead to reliability problems (e.g. exception caused by stack overflow).

Hardware failures
- Commonly caused by manufacturing errors or wear (physical degradation)
- Traditionally, potential design faults within hardware are disregarded
- When the faulty component is substituted, its reliability becomes the initial value of this component
- In the ideal case, the system contains no hardware defects in the beginning and therefore shows no hardware failures
- The reliability of the system does not exceed its initial value through the substitution of components with new components

Software failures
- It is commonly assumed that the software contains defects, which cannot be detected immediately
- Software failures are a result of design errors that are contained in the product from the start and appear accidentally
- Occurrence heavily depends on operational profile (→ execution of faulty code)
- After error correction the system reliability exceeds its initial value (under the assumption that no additional faults are introduced)
- Faults that are introduced during debugging decrease reliability

Quality characteristics	Sub-characteristics
Functionality	Suitability, Accuracy, Interoperability, Compliance
Reliability	Maturity, Recoverability, Fault Tolerance
Usability	Learnability, Understandability, Operability
Efficiency	Time Behavior, Resource Utilization
Maintainability	Stability, Analyzability, Changeability, Testability
Portability	Installability, Replaceability, Adaptability

TRUE: •Correctness has a binary character • An artifact is not consistent to its spec, if it is not correct• System is affected by human• The environment and people can be part of a system • If a system is reliable then it is available• A correct system can have low robustness• Robustness is depending on the specification• Safety allows the presence of risk• If there are no defects the program is correct• A system with a low technical safety can be a tech system •High reliability achieves high safety •Safety can be measured •Every / Each Fault leads (always) to a Failure •Every failure should have a timestamp •An error can cause a failure •Errors can lead to faults• Every failure is caused by a fault• MTTR is a reliability measure• High reliability always leads to high availability •Equipment may be available but not reliable •Low Robustness don't handle failure condition. •Robust system need not be correct. •Correctness is the property of code. •System with low technical safety can be a technical system ***False:*** •A correct system is always safe •Correctness is always defined •Correct software always guarantees a safe system •It can always be decided whether an artifact is correct/not •An available system is (always)safe •When analyzing a system, people are never taken into a/c- •A system with correct specification is always available •A system can have a low robust- ness even if its correct •Robustness has a binary character (gradual char) •Robustness is a property only of the implementation •A system that is robust is always safe •A safe system is always available •Hazard always leads to Accident. (may or may not) •High availability always leads to high reliability •A technical system cannot influence the environment/people.

Chapter 3: Situation Analysis

Design methods: • Still widespread use of informal methods (text) • High interest in semi-formal methods (in particular UML) • Minor use of formal methods • Quality management: • Trend towards the certification of quality management processes (ISO 9001) • Stage of capability maturity model-based assessment methods (e.g. SPICE) **Quality assurance methods:** • Informal methods are frequently applied (testing, review techniques) • Formal methods (proofs) often fail concerning the complexity of the software and the properties of modern programming languages **Stochastic methods** are not widespread, but are increasingly applied in critical application areas in particular **Categories Quality assurance methods**: -Informal are frequently applied (testing, re- view techniques). Formal methods (proofs) often fail concerning the complexity of the software and the properties of modern programming languages. Stochastic methods are not widespread but are increasingly required in critical application areas in particular. **1-Informal Methods:** Methods based on plausibility which produce incomplete results (Testing, Inspection and review)- **2-Stochastic Methods:** Methods which produce statistically reliable, quantified results (Stochastic reliability analysis) **3-Formal Methods:** Methods which produce formally complete results on the basis of formal specifications (Formal verification techniques (Proofs)). **Quality Assurance Methods: Prognosis** •Systematic informal methods are widely used and are obligatory for many application areas where they are required by appropriate standards (**Function-oriented test** planning, Tool supported structural testing) •**Test support** is essential (e.g. **regression tests**) •**Static analyses** are additionally used (**Inspections** in early phases, Tool supported analyses of code in addition to the analyses performed by the compiler) **Test Phases:**•**Module/Unit test**: (Testing of the modules -Testing the correct function of a module w.r.t. the module specification.) •**Integration test** (Testing of the interaction of the mod- ules-Incremental assembly of the modules building the integrated system. Testing of their correct interaction.) •**System-/Acceptance test** (Testing of the functionality and efficiency of a software with regard to the requirements determined in the definition phase) • **Benefit** • Testing in different phases is the reduction of the respective complexity to a reasonable level. • Localization of faults • Improvement of test efficiency, reduction of costs • **Example of OO software: Module Test**: Class/Function test, **Integration test**: Interface test, **System test**: run entire system as a whole, determine correctness and efficiency **Test Phases diagram** (OOP Depiction) (Planning, Analysis, Design, Implementation || Module test, Integration Test, System Test, Field use). **Classification of the QA Methods:** (**Formal verification, Symbolic testing, Dynamic Testing, Static analysis.**) **1. Dynamic Test:** •Properties of dynamic testing**->** The executable program is provided with concrete input values and is executed **->** Program may be tested in the real environment-Never complete (it is not possible to test all possible inputs)-Correctness of the tested program cannot be proven.) **Pros:** -> widely used. -> Often unsystematically applied. **Cons:** > unsystematic -> Tests often not reproducible. -Diffuse activity (management difficulties).**2. Static Analysis** **Pros**-> No program execution is required. -> No input values are selected. -> Some static analyses can detect faults directly. **Cons:** -> The static analysis concentrates on particular partial aspects. ->It is no proof of correctness.) • **Sub-categories** ->Measurement (Metrics) -> Generation of diagrams and tables -> Data flow anomaly analysis -> Testing of programming conventions- Inspection and review techniques. **3.Formal Verification:** -> Formal verification uses mathematical techniques to prove the consistency between specification and implementation. -> A formal specification is necessary. **Pros:** -Verification may be almost completely automated (exception: e.g. finding loop invariants). **Cons:** -Requires preconditions which are often not fulfilled in practice

Chapter 4: Dynamic Testing

Goal of dynamic testing Creating test cases that are: •Representative •Fault sensitive •Distinct from each other (minimal redundancy) •Economic. **Structural Testing**: Evaluation of the adequacy and completeness of the test cases on the basis of the software structure. Determination of the correctness of the outputs based on the specification. **Pros:** Code structure is considered **Cons:** Forgotten (not implemented), but specified functions cannot be detected) •**Two approaches**: **Control flow testing and Data flow testing**). **Control flow testing:** **Statement Coverage:** - **C0-test** •The goal of the statement coverage is to execute each statement at least once, i.e., the execution of all nodes of the control flow graph. **Branch Coverage**: •Branch coverage aims at executing all branches of the program to be tested. This requires the execution of all edges of the control flow graph. **-C1-test. Minimal for unit testing** **Condition Coverage:** Composite decisions are tested from **left to right**. The evaluation of decisions stops when its logical value is known. This is referred to as incomplete evaluation of decisions. **Simple Condition Coverage:** •The simple condition coverage demands the test of all simple **conditions concerning true and false** •**Pros:** simple, low test costs. •**Cons** •Limited performance •In general (concerning the complete evaluation of decisions) it cannot be guaranteed that the simple condition coverage subsumes the branch coverage. **-If decisions are evaluated incompletely, the simple condition coverage subsumes branch coverage**. **Condition/Decision Coverage** •The condition/decision coverage guarantees a complete branch coverage in addition to a simple condition coverage •**Pros**: simple, low test costs, branch coverage is ensured •**Cons:** (Limited performance - Structure of decisions is not really considered). **Minimal Multiple Condition Coverage** •The minimal multiple condition coverage test demands that **besides the simple conditions and the decision also all composite conditions are tested against true and false** **Modified condition/decision coverage:** •The modified condition/decision coverage requires test cases which demonstrate that **every condition can influence the logical value of the overall decision independently of the other conditions** •The application of this method is required by the standard RTCA DO-178C for flight critical software (level A). Basically the method aims at a test as extensive as possible with justifiable test costs •The relation between the number of conditions and the required test cases is linear •**For the test of a decision with n conditions at least n+1test cases are required**. The **maximum number of test cases is 2n** •A complete modified condition/decision coverage causes a branch coverage on the object code level •But not every branch coverage test on the object code level causes a complete modified condition/decision coverage. **Multiple Condition Coverage: Requires full complete evaluation and incomplete evaluation from left to right, full table and test all value combinations of all the conditions.** **Pros** (Very extensive test - Subsumes the branch coverage test and all other condition coverage test techniques) • **Cons** (High test costs - Sometimes there exists no test data for certain combinations (e.g., because of incomplete evaluation of decisions or dependences between conditions) **Path Coverage** •Structured path test distinguished only paths that execute loop not more than k times. This avoids explosion of the number of paths caused by loops • The structured path test **with k=2 is called boundary interior coverage** •The boundary interior coverage differentiates the **three cases** **no loop execution, one loop execution and at least two loop executions** **Data Flow Testing:** Data flow testing is based on the data flow. The basis is the control flow graph enhanced by data flow attributes • Accesses to variables are assigned to one of the classes-> **write: Def use** -> **read: Predicate use (p-use) -> read: Computation use (c-use)** • **All defs-test: make sure all defs have been used in either c-use or p-use, check from the cfg all def node** • **All p-uses-test** • The all p-uses-test requires that every p-use that exists w.r.t. each definition is taken into account during testing, **check from the table all p-use**• **All c-uses-test** • The all c-uses-test requires that every c-use that exists w.r.t. each definition is taken into account during testing, **, check from the table all c-use**• **All c-uses / some p-uses-test resp. all p-uses / some c-uses-test** • If no c-uses resp. p-uses exist for some variable definitions, it is required that at least one p-use resp. c-use is tested• **All uses-test• All c-uses-test + All p-uses-test** **Functional Test (Specification-based Test):**•Determination of the adequacy and the completeness of the test cases as well as derivation of the test data and evaluation of the outputs based on the specification **Pros**: •Completeness w.r.t. the specification is checked. •The test cases are systematically drawn from the specification **Cons:** •Information represented by the code is discarded. **Equivalence Partitioning:** •The test cases for **valid equivalence classes** are generated by the selection of test data from as many valid equivalence classes as possible. •The test cases of **invalid equivalence classes** are generated by choice of test data from 1 invalid class with combination of the rest of data extracted from valid equivalence classes. **State-based Testing: Pros:**+ State-based tests can be used in unit and system testing. + It has widespread use particularly in technical applications such as industry automation, avionics, or the automotive industry. **Cons:** -In state charts of large systems, there tends to be an explosion in the number of states, leads to a considerable increase in transitions. **Transaction Flow Testing Pros:** + A good basis for generating test cases. It directly specifies possible test cases. **Cons:** -Sequence diagrams display only one out of many different options. **Decision Tables or Decision Trees: Pros:** + They guarantee a certain test-completeness by way of their methodical approach. **Cons:** -The size of this representation increases exponentially with the number of conditions. **Diversified Test:** Test of several software versions against each other. **Back-to-Back Test** -Implementation of 2, 3, or even more versions by independent programmers based on the same specifications-Evaluation of the outputs by automated comparison• Requires the multiple realization of software modules based on identical specifications• Economically applicable, if outstanding safety and/or reliability requirements exist, or an automatic evaluation of the outputs is desired or required. **Pros:** test execution (incl. checking of outputs) can be done automatically (saves time and money) **-Cons:** Multiple implementations is required. Faults occurring in all versions are not detected. **Mutations Test** In fact no test method but a possibility to evaluate the efficiency (error detection rate) of test methods. **Regression Test** Test of the present version against the previous version in order to identify undesired changes of the behavior.

Simple Condition Coverage: test of all simple conditions concerning true and false.
Condition/Decision Coverage: test of all simple conditions + overall decision concerning true and false
Minimal Multiple Condition Coverage: all simple conditions + overall decision + composite conditions concerning true and false.
Modified condition/decision coverage: requires test cases which demonstrate that every condition can influence the logical value of the overall decision independently of the other conditions
Multiple Condtn Coverage: test of all value combinations of the conditions

(A && B) || (C && D)

		A	B	C	D	A&B	C&D	||
1	1,2,5,6	F	-	F	-	F	F	F
2	3,7	F	-	T	F	F	F	F
3	4,8	F	-	T	T	F	T	T
4	9,10	T	F	F	-	F	F	F
5	11	T	F	T	F	F	F	F
6	12	T	F	T	T	F	T	T
7	13,14,15,16	T	T	-	-	T	-	T
Simple Condition		Com: (1, 16):Incom: (2,3,4,7)						
con/Dec coverg		Com: (1, 16):						
Minimal Multiple		Com:(1, 16) Incom: (2,3,5,7)						
Modified Condition		Com: (6,7,8,10,14)Incom(1,2,3,4,7)						

(A | | B) && (C | | D)

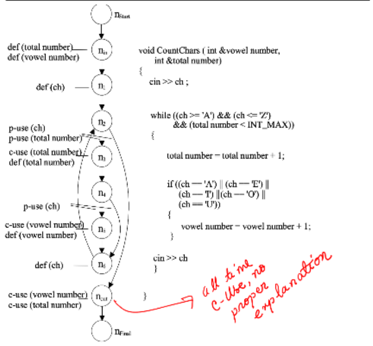
		A	B	C	D	A||B	C||D	&&
1	1,2,3,4	F	F	-	-	F	-	F
2	5	F	T	F	F	T	F	F
3	6	F	T	F	T	T	T	T
4	7,8	F	T	T	-	T	T	T
5	9,13	T	-	F	F	T	F	F
6	10,14	T	-	F	T	T	T	T
7	11,12,15,16	T	-	T	-	T	T	T
Simple Condition		Com: (1, 16):Incom: (1,2,3,7)						
con/Dec coverg		Com:(5,12)						
Minimal Multiple		Com:(1, 16) Incom: (1,2,6,7)						
Modified Condition		Com:(2,6,9,10,11) Incom(1,2,3,4,7)						

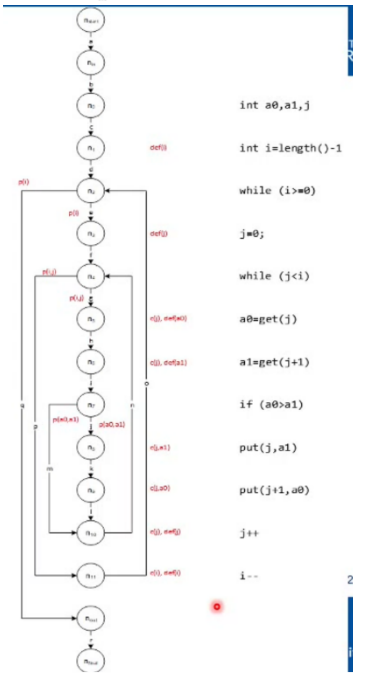
(A && B) | | (C | | D)

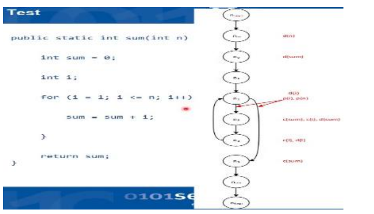
		A	B	C	D	A||B	C||D	||
1	1,5	F	-	F	F	F	F	F
2	2,6	F	-	F	T	F	T	T
3	3,4,7,8	F	-	T	-	F	T	T
4	9	T	F	F	F	F	F	F
5	10	T	F	F	T	F	T	T
6	11,12	T	F	T	-	F	T	T
7	13,14,15,16	T	T	-	-	T	-	T
Simple Condition		Com: (1, 16):Incom: (5,7,6,1)						
con/Dec coverg		Com: (1, 16):						
Minimal Multiple		Com:(1, 16) Incom: (5,7,6,1)						
Modified Condition		Com:(5,9,10,11,13) Incom(1,4,5,6,7)						

(A && B) && (C | | D)

		A	B	C	D	A&&B	C||D	&&
1	1,2,....,8	F	-	-	-	F	-	F
2	9,10,11,12	T	F	-	-	F	-	F
3	13	T	T	F	F	T	F	F
4	14	T	T	F	T	T	T	T
5	15,16	T	T	T	-	T	T	T
Simple Condition		Com: (1,16):Incom: (1,2,3,4,5)						
con/Dec coverg		Com:(1, 16) Incom: (1,2,3,4,5)						
Minimal Multiple		Com:(1, 16) Incom: (1,2,3,4,5)						
Modified Condition		Com:(9,6,13,14,15) Incom(1,2,3,4,5)						

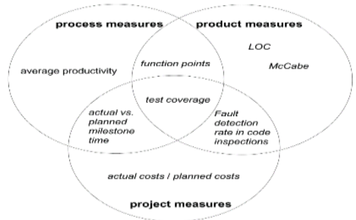






Chapter 5: Measurements

•**Measures quantify certain aspects** of software. **Application of Measures** • Control of software quality • Control of software complexity • Control of the software development process • Costs and time prediction • Costs and time tracing • Definition of standards • Early problem identification • Comparison and evaluation of products • Feedback concerning the introduction of new methods, techniques, tools.



Measure Types: **Product Measures** • Information about properties of a product (complexity, size) • Identification of critical product parts • Classification and comparison of products **Process Measures** • Information about properties of the software development process (productivity, failure costs) • Control of the proper execution of process steps **Project Measures** • Planning and tracking of a project. **Function point method involves product and process measures. Requirements of Measures** • **Simplicity** Is the result so simple that it can be interpreted easily? • **Suitability** Does the measure show an appropriate correlation to the desired property? • **Robustness** Is the value of the measure stable with regard to minor changes of the measured software? • **Timeliness** Is the measure available early enough? • **Processability** Is it possible to process the measures (e.g. statistically -> scale type)? **Reproducibility** **McCabe's cyclomatic number: e-n+2 e = number of edges of a CFG; n = number of nodes of a CFG; CFG = control flow graph** **Completely reproducible** • **Lines of Code (LOC) Count blank lines? Count comment lines?** **Completely reproducible, if specified appropriately** • **Function Points: manual evaluation of complexities required** **Not completely reproducible**• **Understandability**: **Not reproducible** **Measure Scales** **Nominal scale** • Free description of certain properties with labels • Inventory numbers of books of a library (DV 302, PH 002, CH 056) • Names of different requirements engineering methods (SA; SADT)**Ordinal scale** • Mapping of an ordered aspect of a property to an ordered number of measurements in such a way that the order is maintained• Mapping of the arrival of patients to the waiting numbers in a doctor's surgery. **Interval scale** • A scale which is still valid if transformations g(x) = ax + b, with a > 0 are applied to it • Temperature scales in degree Celsius or Fahrenheit **Rational scale** • A scale where measurements can be correlated (statements like double, half, three times as much make sense)• Length in meters (it is twice as far from a to b as from c to d) • Temperature in Kelvin **Absolute scale** • A scale which is the only possibility to measure the issue • Counting• Probabilities. **HouseNo:** value range: N, sometimes with letters, build rule: position, construction time, scale: ordinal. **Sea level of different sites**: value range: Z, Q, build rule: relative to 0-point height measure, scale: interval. **Amount of ducks on a lake**: Counting, absolute. **Weights of Martians on their planet**: Q, build rule: spring balance, scale: interval

Scale type	Operations	Transformations	Example
nominal	=, ≠	all unique	gender, job, ...
ordinal	=, ≠, <, >	strictly monotonic	marks, 1,2,3,4,5,6
interval	=, ≠, <, >, +, -	linear, y=ax+b	Temp in °C
rational	=, ≠, <, >, +, -, *, :	linear, y=ax	Distance in m
absolute	=, ≠, <, >, +, -, *, :	none	quantity

Determination of Scales: Weak Order, Ordinal

- A relation •≥ on a set A is called *order* if
  - a) ∀ x, y, z ∈ A: x •≥ y ∧ y •≥ z ⇒ x •≥ z (transitivity)
  - b) ∀ x ∈ A, ∃ y ∈ A: x •≥ y or y •≥ x (comparability)
  - Example: the relation "is ancestor of" on the set of persons
- An order is called *quasi order* if
  - c) ∀ x ∈ A: x •≥ x (reflexivity)
  - c implies b: every x is at least comparable to itself
  - Quasi orders can contain elements which cannot be ordered
  - Example: the identity "=" on every set
- A *quasi order* is called *half order* if
  - d) x •≥ y ∧ y •≥ x ⇒ x = y (anti-symmetry)
  - Half orders also can contain elements which cannot be ordered
  - Example: the relation „≥" on the set of integers
- A half order is called *linear* if
  - e) ∀ x, y ∈ A: x •≥ y or y •≥ x (connectivity, completeness)
  - Example: the relation "≥" on the set of integers
- Orders which fulfill the axioms a, c (and thus also b) and e, but not necessarily d, are called *weak order*
- In the following the empirical relation •≥ is considered. It is demanded that it generates a weak order on the set of the modules A, fulfilling the following axioms
  - axiom 1: reflexivity: a •≥ a, ∀ a ∈ A
  - axiom 2: transitivity: a •≥ b, b •≥ c ⇒ a •≥ c, ∀ a, b, c ∈ A (If the complexity of module a is greater equal the complexity of module b and the complexity of b is greater equal the complexity of c also the complexity of a is greater equal the complexity of c.)
  - axiom 3: connectivity (completeness): a •≥ b or b •≥ a, ∀ a, b ∈ A

a. We have A a set of software modules. We introduce a new binary empirical relation ¤ between two software modules on the set A. Which measurement axioms should be satisfied by ¤ to yield an ordinal scale?

Axiom 1: Reflexivity $a ¤ a, \forall a \in A$
Axiom 2: Transitivity $a ¤ b, b ¤ c \Rightarrow a ¤ c, \forall a, b, c \in A$
Axiom 3: Connectivity (Completeness) $a ¤ b$ or $b ¤ a, \forall a, b \in A$

•≥ *more complex or equally complex*
Reflexivity $a \bullet\geq a, \forall a \in A$
Transitivity $a \bullet\geq b, b \bullet\geq c \Rightarrow a \bullet\geq c, \forall a, b, c \in A$
Connectivity (Completeness) $a \bullet\geq b$ or $b \bullet\geq a, \forall a, b \in A$
Strict Weak Order

•> *more complex*
Reflexivity NOT SATICFIED $a \bullet> a, \forall a \in A$
Transitivity $a \bullet> b, b \bullet> c \Rightarrow a \bullet> c, \forall a, b, c \in A$
Connectivity (Completeness) $a \bullet> b$ or $b \bullet> a, \forall a, b \in A$
Anti-symmetry $a \bullet> b \wedge b \bullet> a \Rightarrow a \neq b$
Strict total Order

•≈ *equally complex*
Reflexivity $a \bullet\approx a, \forall a \in A$
Transitivity $a \bullet\approx b, b \bullet\approx c \Rightarrow a \bullet\approx c, \forall a, b, c \in A$
Symmetry $a \bullet\approx b \wedge b \bullet\approx a \Rightarrow a = b$
Completeness NOT SATICFIED $a \bullet\approx b$ or $b \bullet\approx a, \forall a, b \in A$
Equivalence relation

Determination of Scales: Rational

- A measure for A → ℝ which meets the requirements of the rational scale mentioned above exists when
  - (A, •≥) fulfills the axioms 1, 2, 3 (reflexivity, transitivity, connectivity)
  - axiom 4: a ° (b ° c) •≈ (a ° b) ° c, ∀ a, b, c ∈ A (associativity)
  - axiom 5: a •≥ b ⇔ a ° c •≥ b ° c ⇔ c ° a •≥ c ° b, ∀ a, b, c ∈ A (Monotony)
  - axiom 6: if c •> d, it is valid: ∀ a, b ∈ A, ∃ n ∈ ℕ, a ° nc •≥ b ° nd (archimedic axiom)

enumerable measure: Lines of Code **calculated measure**: MTTF **evaluated measure**: function points. **complexity measures**• Lines of Code • McCabe's cyclomatic number • Halstead's measures **quality measures** • MTTF (mean time to failure) or • Faults / LOC **Important Primary Data** **Quality-related** • Number, type and cause of faults • Number of problem messages • Number of changes **Cost-related** • Costs of fault corrections • Costs and time exposure (development and testing costs per process step) **Product-related** • Size of the product using an appropriate measure (LOC, pages, processes, number of entries in data dictionary, function points, no of modules) **Single measurement exercise 1.** Measure P is equal to no of atomic predicates in a software module. Atomic predicates are only present in the decision of a module. •**Type of measure is product measure on code level.** • **P can be used as ordinal scale because modules can be ordered according to count of atomic predicates.** •**P can be used as textual chaining of 2 modules, because chaining obtains relation for monotony, archimedic and associtivity. 2. Add** data access to new variable = 1 def + 1 c-use. **Add** already available data access to available variable = measure constant. **Add** new data access type to already available variable = 1 c-use increase. **(We can still apply ordinal scale but can't apply rational scale after textual chaining because measure could stay the same in case same variable used in both modules)** **Monotony condition** states that adding same two parts in given relation, order stays unchanged.

Monotony: x ≥ y ⇔ x ° z ≥ y ° z ⇔ z ° x ≥ z ° y , ∀ x, y, z ∈ A
Prove by counter example:

M(x)= 4/2= 2; M(y)= 4000/3000= 1,33; so x ≥ y
M(z)= 100/100= 1
M(x ° z)= 104/102= 1,02; M(y ° z)= 4100/3100= 1,32; so y ° z ≥ x ° z
which violates Monotony

Software measurement importance • Project planning and cost estimation • Flat management structures • Compliancy to certain software engineering standards • High capability maturity levels **The Halstead Measures** • Set of measures concerning different aspects, e.g., complexity, size, costs, etc. • Are all based on theoretical considerations • η= η1+ η2–**size of the vocabulary** • N=N1+ N2–**length of the implementation** • N= η1log2η1+ η2log2η2• V = N log2η-**Program volume** • D =1/L-**difficulty to implement algorithm**• E=V/L =V^2/V^*-**effort** **Operators**: math operations, function calls, procedure calls, keywords, e.g.: **while, for, if, +,-,*,=,>=,<=,(..), {..},int, string, return**; **Operands**: Variables, constants, jumps, labels, e.g.: int **text**; **i**++, **i**=0, **pi**=**3.14**; Convert.tochar()=**convert(operand), .tochar(operator)** zchn == "a"=**zchn=operand; "a"=operand.** **Live Variables** •Means variable is used, only declared variable is not live. **Definition**: a variable "lives" within a procedure from its first to its last reference **The Capability Maturity Model** classifies the maturity of a software development process using maturity levels. The model used by the SEI uses the following levels: **1-initial, 2-repeatable, 3-defined, 4-measured, 5-optimizing** **The attainment of the maturity levels 4 and 5 is possible** only with the existence and use of a measuring system which enables the following operations • Measuring of productivity and quality • Evaluation of projects on the basis of these measuring • Identification of deviations • Corrective actions in the case of deviations• Identification and control of project risks

LV: medium number of live variables = $\frac{\text{total number of live variables}}{\text{number of executable statements}}$

Chapter 6: DFA

• Rules for data flows • A value must not be assigned twice to a variable (**dd-anomaly**) • An undefined variable must not be referenced (**ur-anomaly**) • The value of a variable must not be deleted directly after the value has been assigned (**du-anomaly**)• These data flow anomalies can be detected by static analysis. **Rules for exercise:** **Without loop**:•**r**(refer the value), •**d**(write the value), •**u**(**n_start and n_end all variables will be undefined**) **example**: **amount = factor*weight, amount=d-use; factor, weight=r-use; return always r-use** **With loop** Has to **execute the loop 3 times for path**, n=0(no loop execution path), n=1(**1 time loop body execution**), n=2(**2 times loop body execution**), N.B. if there are **if conditions**, true and false value of if should also be considered in the path. Res=udr((d)**kr**)**n**ru(k=how many times condition of if statement is valid, k < n)

Chapter 7: Slicing

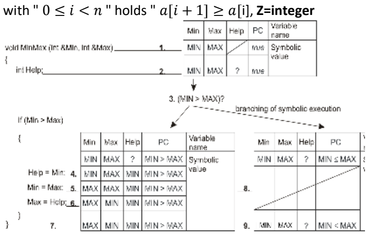
Slicing is related to the extensively automatic provision of information about interactions in a program under observation. Operation mode is comparable to manual fault finding (**debugging**) after the occurrence of a failure. Debugging is also a typical application area for automatic slicing. • A **program slice** in describes which instruction affects an observed value in which way. A slice will always be given in relation to an observed value at a certain position of the program. In a strict sense, this is the definition of so-called **backward slicing**. There is also **forward slicing**. **Static slicing** can be executed in the sense of static analysis. It is not necessary to execute the program or to make assumptions about the values of variables. This is critical if dependencies only result from the concrete value of a datum. Such situations exist, e.g., in conjunction with the application of pointers or fields. • A remedy is offered by so-called **dynamic slicing.** A drawback of all static procedures is that the information that is first generated at the execution time cannot be considered completely. **Exercise:-Options to generate faulty output:- Wrong data flow(straight line <-----)-for this, always search for predecessor, check for def use Wrong control flow(dashed line - - ->), search for successor, check for p-use**

Chapter 8: Reviews

Manual quality assurance in three variants **Review through sending documents to the review team members** • Fast, cheap, flexible, low performance **Structured walkthrough** • Medium use of resources and moderate performance **Fagan inspection** • Expensive and time consuming, but efficient and effective **Software inspection** • Manual quality control of a product • Small group of participants with defined roles • Aims at the detection of faults, not at finding the solutions • Requires a functioning development process • Executed as a formal process • Input and output criteria • Defined inspection phases • Skilled participants • Collection and analysis of inspection data including feedback to the inspection process • Fault documentation • Objectives for the results (e.g. Fault detection rates, inspection rate) **An inspection can be executed in every phase of a software development** **Reviews** • "Review" refers to methods which are no formal inspection. Partially, review is used in the literature as a generic term for all manual test methods (formal inspection included) • Often not only focused on the efficient detection of faults, but also as a means for • decision making • solving of conflicts (e.g. concerning design decisions) • exchange of information • brainstorming • Normally no formal procedure exists for the execution and the choice of the participants as well as their roles • Often no record and analysis of review data • Often no quantitative objectives **The main differences between reviews** respectively walkthroughs and formal software inspections are: • Inspections have the sole aim to detect faults efficiently and effectively • Inspections are done as a defined process **Why Software Inspections?** • Many quality characteristics – e.g. understandability, changeability, informational value of identifiers and comments – are testable only manually • Undetected faults from the definition and design phase later cause high consequential costs • As inspections are executed in a team, the knowledge base is enhanced • Implementation of the principle of external quality control • Delivery of high-quality results to the subsequent software development phase (milestone) • Responsibility for the quality is assigned to the whole team • Manual testing of products is a useful complement of tool supported tests • The compliance to standards is permanently monitored • Critical product components are detected early • Every successful inspection is a milestone in the project • Every member of the inspection team becomes acquainted with the work methods of his colleagues • As several persons inspect the products, the authors try to use an understandable style • Different products of the same author contain fewer defects from inspection to inspection • It turned out that functioning inspections are a very efficient means for quality assurance **Requirements for Inspections** • The required time has to be scheduled project planning • The participants have to be skilled w.r.t. inspections • The procedure of the inspections has to be written down and their observance has to be controlled• The project has to be done well-structured and controlled • There has to be a quality management process with defined quality objectives • The results of inspections must not be used in personnel evaluation • The period between registration and execution of an inspection has to be short,i.e., inspections are executed with high priority **Inspection Team** • **Moderator** • Accepted specialist with special training as moderator • Chairs meeting and controls that the inspection is executed according to the scheduled procedure • **Author (editor)** • Is responsible for the correction of faults detected during the inspection and normally has generated the product to be tested • The author is never the moderator, reader or recorder • **Reader** • Leads the inspection team through the session • Has to be able to describe illustratively the different parts of the work • **Recorder** • Notes and classifies all faults and supports the moderator with the making of the remaining reports • **Inspectors** • All members of the inspection team (also the moderator, author, reader, and recorder) are inspectors whose aim has to be the detection of faults • Further inspectors can be, e.g. • project members from the same project • consultants (standards!) • system specialists • **Size of the review team: 3 to 7 members** **Inspection Phases** • **Planning**: Organizational preparation • **Overview**: The author informs(**2-3 Hours, 500 NLoC** per hour without comments) • **Preparation**: Every inspector prepares(**125 NLoC per** hour) • **Inspection meeting** (2-3 Hours, 90 NLoC/Hour to 125 NLoC/Hour, Accepted, conditionally accepted, Reinspection Required)• **Rework**: Fault correction • **Follow-up**: Inspection of the fault corrections (If conditionally accepted -> Verification can be done by author and reader || if reinspection =>Conventional inspection meeting) **Ariana5 Failure**: break-down of flight controller resulting a mechanical destruction of rocket **Fault**: The integer conversion of the 64-bit floating point variable into a signed integer caused a data overflow **Error** blind reuse of software components of Ariane4, which have not been tested sufficiently within the new environment of Ariane5

Chapter 9: Formal Verification

Symbolic Test - Properties • The unit under test is executed with symbolic values by an interpreter • Runs in an artificial env •Gains complete information about the correctness for whole input areas • Sometimes can prove the correctness • Takes a position between dynamic test, static analysis and verification **Symbolic Test - Problems** • As symbolic results are always assigned to a program path, for programs with an infinite number of paths also the number of symbolic results is infinite• Problems caused by the data structures available e.g. arrays or pointers • If an array is accessed during the symbolic execution the array index is a symbolic term so that the number of possibilities to be considered increases very fast, as a result, it cannot be decided which concrete array element has to be accessed• Floating point variables also cause problems due to the discrepancy between the discrete arithmetic of a computer and the continuous character of real numbers• In the symbolic execution no approximation errors occur. An insertion of concrete values in the symbolic results produce absolute exact values **Formal Verification** • The verification demonstrates consistency between specification and implementation using proof techniques• A formal specification is required• The verification can prove the correctness **Proof techniques: 1.Assertion:** Is a statement about the state that is true at a specific point in a program. **Partial correctness:**If algorithm returns a result,it's correct **Total correctness** A result exists and algorithm terminates (one is always delivered). Total correctness implies partial. **Hoare calculus is used to show partial correctness. 1. x=√2;precondn: y≥0 ∧ y∈R; postcondn: z≥0∧x∈R∧x² =y 2. int** $a[] = \text{sort}(b[])$ **Precondition:** sizeof $(a[]) = \text{sizeof}(b[]) \wedge \forall i, n \in N''$ with " $0 \le i \le n$ " holds " $b[i] \in Z$ **Solution** **Postcondition**: $\forall i, n \in N$ with " $0 \le i \le n$ " holds " a[i],b[i]єZ $\wedge \forall i, n \in N$ with " $0 \le i < n$ " holds " $a[i+1] \ge a[i]$, **Z=integer**



Hoare-Calculus • A program is a formal description of behavior • It is possible to determine the behavior of a program by application of inference rules • If **S** is a program or a part of a program and **P** is the precondition before the execution of S and after the execution of S postcondition **Q** is valid on condition that S terminates, one writes• **P { S } Q Axioms • A0: P$_f$^{x} { x := f} P**

- A1: $\frac{P\ \{S\}\ Q,\ Q \supset R}{P\ \{S\}\ R}$ A postcondition Q can be replaced by a weaker Postcondition R
- A2: $\frac{Q\ \{S\}\ R,\ P \supset Q}{P\ \{S\}\ R}$ A precondition Q can be replaced by a stronger Precondition P
- A3: $\frac{P\ \{S_1\}\ Q,\ Q\ \{S_2\}\ R}{P\ \{S_1; S_2\}\ R}$ If P is precondition of S_1 wrt. Q and Q is precondition of S_2 wrt. R, than P is precondition of the Sequence S_1; S_2 wrt. R
- A4: $\frac{P \wedge B\ \{S\}\ P}{P\ \{\text{WHILE B DO S END}\}\ P \wedge \neg B}$ If P ∧ B is precondition of S wrt. P then P can be used as loop invariant for the loop with S as the body of the loop and B as the loop condition. P is precondition of the loop *WHILE B DO S END* wrt. P ∧ ¬ B

$n \geq 0$ *(PRE)* × $i = 0$ × $k = -1$; × $y = 0$
{ **Invariante** P}
while $(i < n)${
{ $P \wedge$ **loop condition B**}
$i = i + 1$
$k = k + 2$
$y = y + k$ **// to find loop invariant p**
$\{P\}$ **// execute loop 5 times, put values of all variables**
} **// if variable initialized, put that value on table**
$\{P \wedge \neg B\}$ // table
$y = n^2 (POST)$
// with $B = \{i < n\}$ and $P = \{i \le n \wedge k = 2i - 1 \wedge y = i^2\}$

```
public class Switch {
  private boolean mode;
  public Switch() {
    init();
  }
  private void init() {
    mode=true;
  }
  public boolean toggle(boolean mode) {
    boolean result;
    if((this.mode&&mode)||(!this.mode&&!mode))
      result=true;
    else
      result=false;
    if (this.mode)
      this.mode=!mode;
    else
      mode=mode;
    return result;
  }
}
```