Software Engineering — the establishment and use of sound engineering principles to obtain economically software that is reliable and works efficiently on real machines.
(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) study of approaches as in (1).
1) Capturing Requirements
2) Design
3) Implementation
4) (Code) Quality Assurance
5) (Software) Project Managment
(sd) project **sucessfull** iff **developer, customer, user** happy
quality metric —
(1) A quantitative measure of the degree to which an item possesses a given quality attribute.
(2) A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.
A Metric Should be: • **relevant** wrt. overall goals and needs• **plausible**: Good evidence that proband's valuations mand quality are related• **robust**: The valuation of a proband cannot be arbitrarily manipulated; antonym / opposite: subvertible• **available**: Valuations need to be in place when needed• **economical**: Cost of measuring needs to be in a good relation to gain• **comparable**: Some scales have incomparable values• **reproducible**: Multiple applications to the same proband yields the same valuation• **differentiated**: Sufficiently different valuations for sufficiently different probands
• **prescriptive**, i.e. stating a need or demand on not yet existing software.
• **descriptive**, i.e. stating a diagnosed or prognosed property of existing software.
• **base measure** — measure defined in terms of an attribute and the method for quantifying it.• **derived measure** — measure that is defined as a function of two or more values of base m

Nominal Scale
• nationality, gender, car manufacturer, geographic direction, train number, . . .
• programming language (S = {Java, C, . . . })
Ordinal Scale • > • CMMI scale (maturity levels 1 - 5) **Interval Scale**
→ There is a (natural) notion of difference
$\Delta : S \times S \to \mathbb{R}$, but no (natural) proportion and 0.
• time of check-in in revision control system
Rational Scale• runtime of a program for given inputs→ The (natural) zero induces a meaning for proportion m1/m 2 .
Absolute Scale• "average number of children per family: 1.203" – what is a 0.203-child?• number of known errors.**Subjective and Pseudo Metric**:• if no good objective metrics• e.g. we do not really measure maintainability; average-LOC is only interpreted as such • good correlation between proband quality and metric valuation for our usual probands
Goal-Question-Metric (GQM):
(i) Identify the goals relevant for project or organisation.(ii) From each goal, derive questions that need to be answered to see whether the goal is reached. (iii) For each question, choose (or develop) metrics that contribute to finding answers

Maximise profit, i.e. maximise difference between **benefit** and **cost** (cost = all disadvantages of a solution, **benefit** = all benefits of a solution)
Delphi method: 1) write down estimates 2) show estimates and explain 3) estimate again => then take e.g. median
Person-months ist eine Einheit für effort, sie gibt an wieviele Monate, wenn jeden Monat genau 152 Arbeitsstunden enthält, eine einzelne Person bräuchte, um ein spezifisches Projekt fertigzustellen
COCOMO: S = Esimtaed program size S in DSI = Delivered Source Instructions
Principles of (Architectural) Design.
Modularisation: split software into units / components of manageable size, provide well-defined interface. **Separation of Concerns:** each component should be responsible for a particular area of tasks, group data and operation on that data; functional aspects; functional vs. technical; functionality and interaction.
Information Hiding: the "need to know principle" / information hiding, users (e.g. other developers) need not necessarily know the algorithm and helper data which realise the component's interface. **Data Encapsulation:**offer operations to access component data, instead of accessing data (variables, files, etc.) directly.

T: decision table	r_1	···	r_n	else
description of condition c_1	$v_{1,1}$	···	$v_{1,n}$	
⋮	⋮	⋱	⋮	
description of condition c_m	$v_{m,1}$	···	$v_{m,n}$	
description of action a_1	$w_{1,1}$	···	$w_{1,n}$	$w_{1,e}$
⋮	⋮	⋱	⋮	⋮
description of action a_k	$w_{k,1}$	···	$w_{k,n}$	$w_{k,e}$

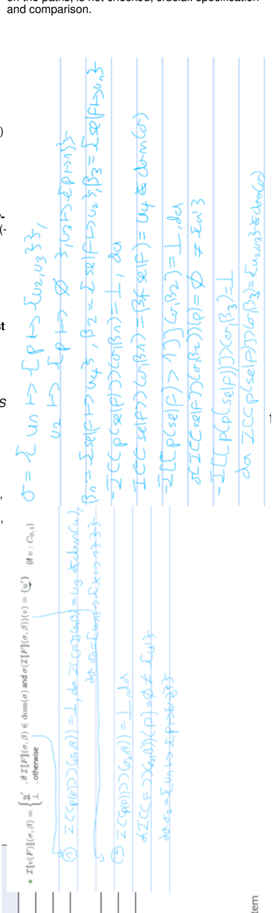
project – temporary activity characterized by • a start date,• specific objectives and constraints,• established responsibilities,• a budget and schedule and• a completion date.
project **successfull** iff all satisfied
risk — a problem, which did not occur yet, but on occurrence threatens important project goals or results. Whether it will occur, cannot be surely predicted.
iterative software development — software developed in multiple iterative steps Goal: each iterative step, beginning with the second, corrects and improves the existing system based on defects detected during usage. Each iterative steps includes the characteristic activities analyse, design, code, test. **incremental software development** — Total extension of a system realised in stages of expansion. The first stage is the core system. Each stage of expansion extends the existing system and is subject to a separate project. Providing a new stage of expansion typically includes an improvement of the old components **evolutionary software development** — an approach which includes evolutions of the developed software under the influence of practical/ field testing. New and changed requirements are considered by developing the software in sequential steps of evolution.
The Agile Manifesto: Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan
Agile Principles:
continous / sustainable delivery (satisfy the customer through early and continuous delivery of valuable software), **people** (self-organizing teams), **simplicity** (the art of maximizing the amount of work not done is essential), **changes** (Welcome changing requirements, even late in development), **retrospective** (Continuous attention to technical excellence and good design enhances agility)
Lastenheft (Requirements Specification)
Vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrages.
Pflichtenheft (Feature Specification) Vom Auftragnehmer erarbeitete Realisierungsvorgaben aufgrund der Umsetzung des vom Auftraggeber vorgegebenen Lastenhefts.
Possible causes for unsuccessful projects:
1) misunderstanding of requirements; contradicting requirements 2) non-scalable design; feature forgotten; designer misunderstood requirement 3) programmer misread design specification; simple programming mistake 4) wrongly conducted test; tester misunderstood requirement 5) wrong cost estimation; bad scheduling; team member was not aware of responsibilities
Spiral Model: iteratively address the currently highest risks. +: 1) know early if the project goal is unreachable. 2) gives a good feeling
Waterfall Model — Software devel-opment is seen as a sequence of activities coupled by results (documents). These activities can be conducted concurrently or iteratively.
Apart from that, the sequence of activities is fixed as (basically) analyse, specify, design, code, test, install, maintain.
Extreme Pro-gramming (XP):
XP Values: simplicity, feedback, communication, courage, respect
XP practices:
programming (test driven development, refactoring, simple design, pair programming), **team** (joint responsibility for code), **managment** (planning game, short release cycles)
Scrum:
Product Owner: maintains requirements in the product backlog, plans and decides which requirement(s) to realise in next sprint. representetive of the Customer
Scrum Team: members capable of developing autonomously,
decides how and how many requirements to realise in next sprint, distribution of tasks
Scrum Master: moderates daily scrum, responsible for keeping product backlog up-to-date, helps to conduct smcrum the rightTM way; **Name Origin: Rugby** get the ball in a **scrum**, then **sprint** to score.
product backlog: comprises all requirements to be realised, priority and effort estimation formmt requirements, collects tasks; **release plan:** based on initial version of product backlog, how many sprints, which major requirements in which sprint; **release-burndown report:** see sprint-burndown report; **sprint backlog:** requirements to be realised in next sprint, taken from product backlog, more precise estimations, daily update (tasks done, new tasks, new estimations); **sprint-burndown report:** completed/open tasks from sprint backlog, should decrease linearly, otherwise remove tasks from sprint backlog;
sprint report: which requirements (not) realised in last sprint, description of obstacles/problems during sprint; **daily scrum:** daily meeting, 15 min, discuss progress, synchronise day plan, document new obstacles); **sprint:** at most 30 days; **sprint review:** assess amount and quality of realisations, product owner accepts results; **sprint retrospective:** assess how well the scrum process was implemented, identify actions for improvement

V-Modell XT: role, activity, product=artefact, step=parts of activities, topic=parts of products, discipline=set of related products / activitimes, phase=project segment
Process model defines dependencies which artefacts needs to be available before starting which activity. Does not define that Activity A must be completed before Activity B
• **linear:** the strict Waterfall Model (no feedback between activities)
• **non-linear:** everything else (feedback between activities)
Prozessmodel erstellen:
• in Reihenfolge: Rollen, Aktivitäten, Artefakte (alle Pfeile gehen nur nach rechts)
• Im Diagramm steht jede Rolle für eine Teilmenge aus einem Pool von Leuten => Rollen kommen mehrfach vor
• Aktionen können auch mehrfach vorkommen, da man verschiedene Inputs und Outputs mischen soll
prototype — A preliminary type, form, or instance of a system that serves as a model for later stages or for the final, complete version of the system **prototyping** — A hardware and software development technique in which a preliminary version of part or all of the hardware or software is developed to permit user feedback, determine feasibility, or investigate timing or other issues in support of the development process
rapid prototyping — A type of prototyping in which emphasis is placed on developing prototypes early in the development process to permit early feedback and analysis in support of the development process.
Code & Fix: approach where coding or fixing in alternation with ad-hoc testing are the only consciously conducted activities
A **phase** is a continuous range of time in which certain works are carried out and completed. At the end of each phase, there is a **milestone**.
A phase is **successfully completed** if the criteria defined by the milestone are satisfied.
Phase Model: The projectm is planned by phases, delimited by well-defined milestones. Each phase is assigned a time/cost budget. No iteration of phases. Activities may span multiple phases. Makes controlling easier.
Process —
(1) A sequence of steps performed for a given purpose.
(2) To perform operations on data.
Describing Soft. Dev. Processes:
• **role** — has responsibilities and rights, needs skills and capabilities
• **artefact (or product)** — all products emerging during a development process. May have a state
activity — any processing of artefacts **decision point** — special case of activity
Measure quality of a process:
• **SPICE (Software Process Improvement and Capability Determination):** can be seen as a specification for process pseudo-metrics, 1) define considered process areas 2) assess each process for so-called process attributes 3) map results to maturity level; assessment conducted by specially trained assessors (subjective metrics)
• **CMMI (Capability Maturity Model Integration):** considers 5 process categories (project magmt., support, engineering, process mgmt.); each consisting of 5–7 process areas; each process area can be assigned a capability level (0: incomplete, 1: performed, 2: managed, 3: defined); capability levels can be aggregated to organisation's maturity level (1: initial, 2: managed, 3: defined, 4: quantitatively managed, 5: optimizing)
Requirement:
(1) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, orother formally imposed documents.(2) A documented representation of a condition or capability as in (1)
requirements analysis – (1) The process of studying user needs to arrive at a definition of system, hardware, or software requirements.
requirements specification,• i.e., a set of specified requirements,• is supposed to partition• the set of possible systems• into acceptable and non-acceptable(or correct and incorrect) systems.
requirements specification should be: • correct• complete• relevant• consistent, free of contradictions•neutral, abstract•traceable, comprehensible•testable, objective
representation and form of a requirements specification should be: •easily understandable •precise •easily maintainable •easily usable
expected total effort: alle PM's von Aktivities aufsummieren und dabei availability miteinbeziehen (e.g. 0.5PM per month)
expected minimum duration: Gantt Chat
functional requirement:
Every constraint on things which are observable in the sequences of the function S; timing is a functional requirement
non-functional requirements:
programming language, coding conventions, process model requirements, portability
hard requirement: es muss immer passieren
soft requirement: wenn es manchmal zu spät ist, ist es nicht schlimmt **dictionary:** comprises definitions and clarifications of terms that are relevant to the project and of which different people (in particular customer and developer) may have different understandings before agreeing on the dictionary. **formal method:** iff its techniques and tools can be explained in mathematics.

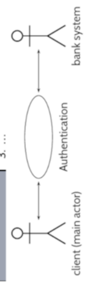
open: customer is aware of and able to explicitly communicate the requirement
semi-tacit: customer not aware of something being a requirement (obvious to the customerbut not considered relevant by the customer, not known to be relevant).
Task of the analyst: • ask what is (not) wanted, • establish precision, look out for contradictions • anticipate exceptions, difficulties, corner-cases • have technical background to know technical difficulties • communicate (formal) specification to customer • "test" own understanding by asking more questions. => Requirements Engineering is about seeing the absence of information.
specification — A document that specifies,• in a complete, precise, verifiable manmmner,the• requirements, design, behavior,or other characteristics of a system or component,and, often, the procedures for determining whether these provisions have been satisfied.
software requirements specification (SRS) — Documentation of the es-sential requirements of the software and its external interfaces.
Use Case — A sequence of interactions between an actor (or **actors**) and a **system** triggered by a specific actor, which produces a result for an actor.
User Story is a description of a piece of functionality that will be valuable to a user (or owner) of the software."Per user story, use one file card
• unique identifier (e.g. unique number),• priority (from 1 to 10) assigned by customer,• effort, estimated by developers,• back side of file card: (acceptance) test case(s)
+: easy to create, small units; close contact to customer; objective / testable: by fixing test cases early
-: may get difficult to keep overview over whole system to be developed→ maybe best suited for changes / extensions (after first iteration); not designed to cover non-functional requirements and restrictions; agile spirit: strong dependency on competent developers;
architecture— The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
design— (1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component. (2) The result of the process in (1).
software architecture— The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.
architectural description— A model – document, product or other artifact – to communicate and record a system's architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns.
system— A collection of components organized to accomplish a specific function or set of functions.
software system— A set of software units and their relations, if they together serve a common purpose. It usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.
component— One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components. **software component**— An architectural entity that (1) encapsulates a subset of the system's functionality and/ or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.
module— (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler (2) A logically separable part of a program. (3) A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.
interface— A boundary across which two independent entities meet and interact or communicate with each other.
interface (of component)— The boundary between two communicating components. The interface of a component provides the services of the component to the component's environment and/or requires services needed by the component from the requirement.
structure: The structure of something is the set of relations between its parts.
unstructured: Something not built from (recognisable) parts
model: Is an abstract, formal, mathematical representation or description of structure or behaviour of a (software) system.
Concrete or mental **image** (Abbild) of something or a concrete or mental **archetype** (Vorbild) for something.Three properties are constituent:(i) the **image attribute** (Abbildungsmerkmal), i.e. there is an entity(called original) whose image or archetype the model is,(ii) the **reduction attribute** (Verkürzungsmerkmal), i.e. only those attributes of the original that are relevant in the modelling context are represented,(iii) the **pragmatic attribute**,i.e. the model is built in a specific context for a specific purpose.

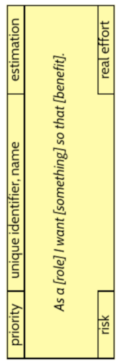
view — A representation of a whole system from the perspective of a related set ofconcerns
viewpoint — A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposesand audience for a view and the techniques for its creation and analysis
A **perspective** is determined by concerns and information needs: • team leader, e.g., needs to know which team is working on what component
4+1 View:
system view: How is the system under development integrated into (or seen by) its environment? Withwhich other systems (including users) does it interact how? **static view (~ developer view):** Components of thearchitecture, their interfaces and relations. Possibly:assignment of development, test, etc. onto teams.**dynamic view (~ process view):** how and when are components instantiatedand how do they work together at runtime. **deployment view (~ physical view):** How are component instances mapped onto infrastructure and hardware units?
reflective (or assertive): what should (or should not) be computed; **constructive:** how things are computed.
Point error: isolated input value triggers the error. **Range error:** multiple "neighbouring" inputs trigger the error. (For software, adjacent inputs may yield arbitrarily distant output values.)
execution trial—does the program run at all? **throw-away-test**—invent input and judge output on-the-fly (-> "rumprobieren") **systematic test**—somebody (not author!) derives test cases, defines input/soll, documents test
execution. More economic in long run.
unit test—a single program unit is tested (function, sub-routine, method, class, etc.) **module test**—a component is tested. **integration test** — the interplay between components is tested. **system test** — tests a whole system.
Ideal test case (*In, Soll*): **low redundancy**, i.e. it does not test what other test cases also test. **error sensitive**, i.e. has high probability to detect an error. **representative**, i.e. represent a whole class of inputs, (i.e., software *S* passes (*In, Soll*) if and only *S* behaves well for all *In'* from the class)
Statistical test/testing: Randomly (!) choose test cases *T1, . . . , Tn* for test suite *T*. Execute test suite *T*.
Software error habitat: Software errors (seem to) enjoy range boundaries, special cases of the problem, or of the programming language semantics, and complex implementations.
Glass-box testing: considers the control flow graph, defines coverage measures.
criteria for when to stop testing: **all** (previously) specified **test cases** have been executed with **negative result**. (Special case: All test cases resulting from a certain strategy, like maximal statement coverage have been executed.) testing effort time sums up to **x (hours, days, weeks)**. testing effort sums up to **y** (any other useful unit). **n errors** have been discovered. no error has been discovered during the last **z hours (days, weeks)** of testing. The average cost per error discovery **exceeds** a defined threshold **c**. Values for **x, y, n, z, c** are fixed based on experience, estimation, budget, etc.
Model-based Testing. Approach: **Location Coverage.** Check whether for each location of the model there is a corresponding configuration reachable in the software (needs to be observable somehow).Analogously: **Edge Coverage.** Check whether each edge of the model has corresponding behaviour in the software.
Quality Criteria on Architectures: testability architecture design should keep testing (or formal verification) in mind (buzzword "design for verification"), high locality of design units may make testing significantly easier (module testing), particular testing interfaces may improve testability (e.g. allow injection of user input not only via GUI; or provide particular log output for tests).
changeability, maintainability most systems that are used need to be changed or maintained, in particular when requirements change,risk assessment: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise, encapsulate). **portability** porting: adaptation to different platform (OS, hardware, infrastructure). systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change (→ introduce abstraction layer).
validation— Process evaluating a system or component during or at end of development process. Determine if satisfies specified requirements. Contrast with: **verification**.
verification—(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: **validation.** (2) Formal proof of program correctness.
Testing: Execution of test suites. **Review:** Review by human. **Formal Verficiation:** Überprüfung Korrektheit Abläufe durch Darstellung Abläufe in Formaler Sprache. (LSC)

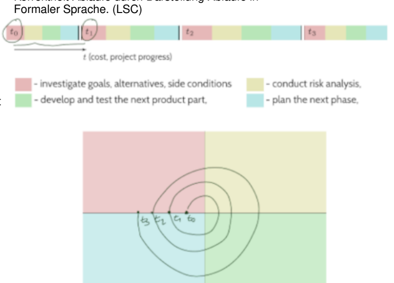
A test suite is a finite set of test cases {T1, . . . , Tn}. An execution of a test suite is a set of computation paths, such that there is at least one execution for each test case.
Software Examination In each examination, there are two paths frommm the specification to results:the **production path** (using model, source code, executable, etc.), and the **examination path** (using requirements specifications). A check can only discover errors on exactly one of the paths. If a difference is detected:examination result is positive. What is not on the paths, is not checked; crucial: specification and comparison.



name	
goal	
pre-condition	
post-condition	
post-cond. in exceptional case	
actors	
open questions	
normal case	1. … 2. … 3. …







COCOMO II: $E = 2.94 \cdot S^X \cdot M$

• **Program size**: $S = (1 + REVL) \cdot (S_{new} + S_{equiv})$ $\quad M = RELY \cdot DATA \cdot \cdots \cdot SCED$
• requirements volatility $REVL$:
e.g. if new requirements make 10% of code unusable, then $REVL = 0.1$
• S_{new}: estimated size minus size w of re-used code,
• $S_{equiv} = w/q$, if writing new code takes q-times the effort of re-use.

• **Scaling factors:** $X = \delta + \omega$, $\omega = 0.91$, $\delta = \frac{1}{100} \cdot (PREC + FLEX + RESL + TEAM + PMAT)$

COCOMO 81:
Basic COCOMO
• effort required: $E = a \cdot (S/kDSI)^b$ [PM (person-months)]
• time to develop: $T = c \cdot E^d$ [months]
• headcount: $H = E/T$ [FTE (full time employee)]
• productivity: $P = S/E$ [DSI per PM] (← use to che…

Intermediate COCOMO:
$E = a \cdot (kLOC)^b \cdot \prod_{i=1}^{15} C_i$ $\qquad E = M \cdot a \cdot (S/kDSI)^b$ [person-months]
(effort adjustment factor = product)
$M = RELY \cdot CPLX \cdot TIME \cdot ACAP \cdot PCAP \cdot LEXP \cdot TOOL \cdot SCED$
riskvalue = $p \cdot K$
p: probability of problem occurrence, K: cost in case of problem occurrence.

(empirische) p-Quantil: $x_p := \begin{cases} x_{(\lceil np \rceil)}, & \text{falls } np \notin \mathbb{N}, \\ \frac{1}{2}\left(x_{(np)} + x_{(np+1)}\right), & \text{falls } np \in \mathbb{N} \end{cases}$

1, 2, 3, **4**, **5**, 6, 8, 9
Median $= (4+5)\div 2 = \underline{4.5}$

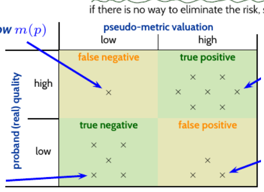
1, 3, 3, **6**, 7, 8, 9
Median = $\underline{6}$

	$=, \neq$	$<, >$ (with transitivity)	min, max	percen-tiles, e.g. median	Δ	propor-tion	natural 0 (zero)
nominal scale	✔	✘	✘	✘	✘	✘	✘
ordinal scale	✔	✔	✔	✔	✘	✘	✘
interval scale (with units)	✔	✔	✔	✔	✔	✘	✘
rational scale (with units)	✔	✔	✔	✔	✔	✔	✔
absolute scale	a rational scale where S comprises the key figures itself						

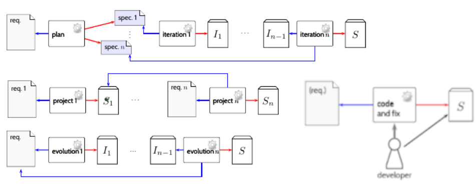
Definition. A **metric**[1] is a function
$$m : P \to S$$
that assigns to each **proband** $p \in P$ a **valuation** ("Bewertung") $m(p) \in S$.
We call S the **scale** of m.

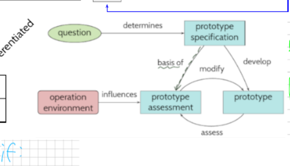
dimension	unit	measurement procedure
program size	LOC_{tot}	number of lines in total
net program size	LOC_{ne}	number of non-empty lines
code size	LOC_{pars}	number of lines with not only comments and non-printable
delivered program size	$DLOC_{tot}$, $DLOC_{ne}$, $DLOC_{pars}$	LOC of only that code which is delivered to the customer

Spiral Model: iteratively address the (currently) highest risk
Repeat until end of project (successful completion or failure):
(i) **determine** the set R of **risks** which are **threatening** the project;
if $R = \emptyset$, the project is successfully completed
(ii) **assign** each risk $r \in R$ a **risk value** $v(r)$
(iii) for the risk r_0 with the **highest risk value**, $r_0 = \max\{v(r) \mid r \in R\}$,
find a way to eliminate this risk, and go this way;
if there is no way to eliminate the risk, stop with project failure

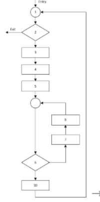


	plausible	robust	available	economical	comparable	reproducible	differentiated
Subjective Metrics	✔	✔	(✔)	✘	(✔)	(✘)	✔
Pseudo Metrics	(✘)	(✘)	✔	✔	✔	✔	✔





Control Flow Graph:

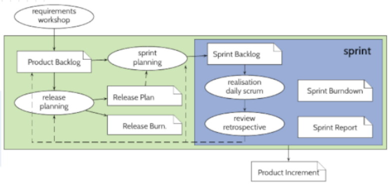


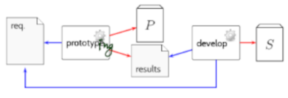
```
void insertionSort(int[] array) {
  for (int i = 2; i < array.length; i++) {
    tmp = array[i];
    array[0] = tmp;
    int j = i;
    while (j > 0 && tmp < array[j-1]) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = tmp;
  }
}
```

(McCabe Complexity)
• zählt #if + #while

Number of edges: $|E| = 11$
Number of nodes: $|V| = 6 + 2 + 2 = 10$
External connections: $p = 2$
$\to v(P) = 11 - 10 + 2 = 3$

Definition. [Cyclomatic Complexity [McCabe, 1976]]
Let $G = (V, E)$ be the Control Flow Graph of program P.
Then the cyclomatic complexity of P is defined as $v(P) = |E| - |V| + p$ where p is the number of entry or exit points.





Proposal: View software S as a **function**
$$S : i_1, i_2, i_3, \dots \mapsto o_0, o_1, o_2, \dots$$
which maps **sequences of inputs** to **sequences of outputs**.