Software Engineering II

0) AGILE & SCRUM

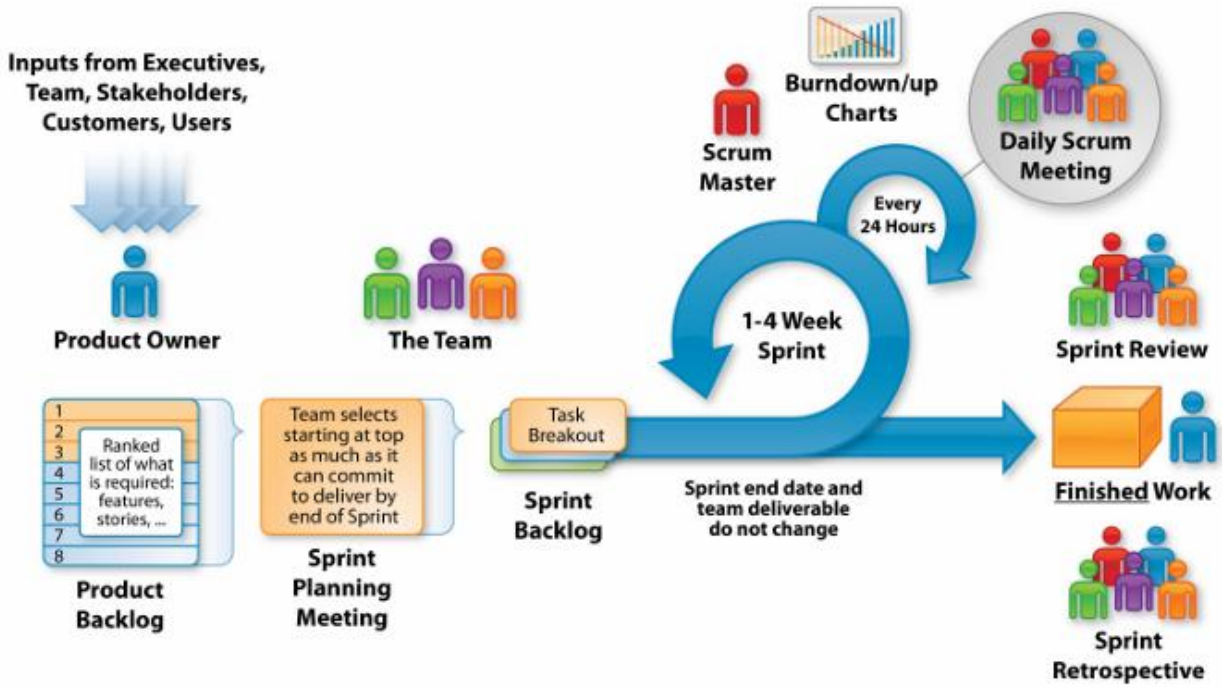
Engineering = Design + Construction (what we focus on: planning, monitoring, tools) + Operation.

AGILE Development:
- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan

The priority is satisfying the customer through early and continuous delivery of software; changing in the project is frequent, even late in development. Continuous face-to-face conversation. Simplicity is key. Best architectures, requirements and designs emerge from self-organizing teams; teams reflect on how to become more effective and then adjust their behavior accordingly. There must be always tight collaboration between devs and stakeholders.

Software SCRUM is the most used methodology: a team-based framework to develop complex systems and products; some principles are:
- test as you go (iterative and incremental approach to optimize predictability and control risk)
- deliver product early and often (feedback)
- document as you go (only as required)
- cross-functional teams
- TRANSPARENCY (process visible to whom is responsible)
- INSPECTION (on artifacts and goals to detect variances)
- ADAPTATION (to meet the goals)

Scrum process in a nutshell:



SCRUM ROLES:

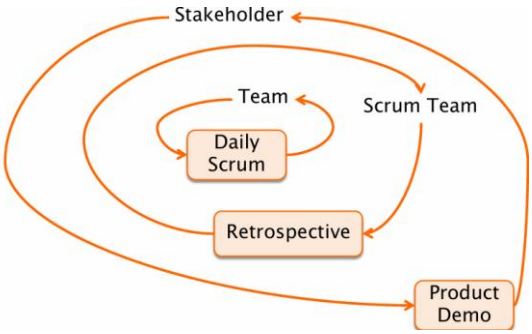
- **PRODUCT OWNER** (**P.O.**) = owns the product backlog; directs team toward most valuable work; represents business and customers/stakeholders. He makes sure that the needs of the customers and end-users are understood by the team, and is the keeper of the product vision (who is built for, why and how they will use it). He prioritizes stories
- **SCRUM MASTER** (**S.M.**) = acts as a coach to produce a self-organizing team (is a facilitator, not a "boss") [as the team becomes self-managing, the master steps back] and removes impediments for the team ("impediment bulldozer"). He is a scrum expert and advisor
- **TEAM MEMBER** (**T.M.**) = team is self-organizing; team member estimates the effort for the features. Team size is 5-9
  ⚠ SCRUM TEAM = owner + master + team; TEAM = team members (not owner and master)

EVENTS:

- **SPRINT** → basic iteration in the Scrum approach: produces a piece of working SW to be demonstrated and reviewed at the end of the sprint (1-4 weeks long)

Monday	Tuesday	Wednesday	Thursday	Friday
	Scrum (15m)	Scrum (15m)	Scrum (15m)	Scrum (15m)
Sprint Planning (2h)				
				Sprint Review (1/2 h)
		Story Time (1h)		Retrospective (1h)

  - Sprint Planning:
    - What will we do? Set of committed stories; Product Owner proposes story; Team Members decide whether commit
    - How will we do it? Decompose stories into tasks; max effort per task
    - Sprint Backlog = list of stories; related tasks
    - Estimation = task hours, task points, task count
    - Effort Estimation = to define a predictable schedule (relative vs absolute estimates); after a sprint is complete, "VELOCITY" (an estimate of stories per sprint) is done
    - Estimation Poker = every team member picks a card and show it to others (value = estimation effort for a task)
  - Daily Scrum = daily, small, brief to point out what has been done, what will be done, obstacles
  - Sprint Review = show off some piece of working SW to stakeholders, report on incomplete stories, transparency, stakeholders' feedback
  - Retrospective = gather data, generate insights, decide what to do, focusing on lessons learned (to apply them to next sprint), so the goal is to identify things to improve and define the relative action plan



⚠ Release (long-term) planning = fixed scope, fixed date or both

ARTIFACTS:

- **Product Backlog** (**PB**) = includes PBI (Product Backlog Items) [we will use User **Stories** as PBIs]. It's anything that will consume team resources; it's ever changing and property of the P.O.
  PBIs (Stories) must be "**INVEST**" (Independent, Negotiable, Valuable, Estimable, Small, Testable); PB must be "**DEEP**" (Detailed, Estimated, Emergent, Prioritized)
  A **STORY** is a description of a desired functionality from the view of a user; it's written by the P.O. and it's easy to turn into automated tests. The template is:

  As an *unregistered user*
  I want *to create a new account*
  So that *I can buy items*

  Aspects of a user story are: Card (essential description and estimation), Conversation (team-P.O-customer-stakeholders; several times done), Confirmation (satisfaction).
  If a story is too large, it's called EPIC and needs to be split into stories

- **Sprint Backlog** (**SB**) = detailed small tasks; includes committed stories (relative tasks) and **additional tasks** (team improvement, research, performance, security, bug fix). We have charts:
  - Burn down chart = residual work units, sprint (team), release (P.O)
  - Burn up chart = completed work units
  - Task board = to do, in progress, done ("*shippable*" = after code review, design review, refactoring, performance testing and unit testing)

→ YouTrack

YouTrack is a project management and team collaboration tool (available on cloud or on-premises; free up to 10 users). Main features are:

- task **Management** (+ issue tracking)
- team **Collaboration**
- **Planning**
- **Time Tracking**
- **Agile** boards

⚠ **SEE SLIDES** for all the information and **SETUP**

1) Project → PARTICIPIUM

⚠ PO can be contacted on Telegram; clarification on stories (using tag #story-id). Scrum Master can be contacted and tagged with #scrum-master

⚠ Do frequent scrum meeting; track the time used in the meeting on YouTrack

⚠ For each sprint → 6 members * 8h/week * 2 weeks = 96h

Deadlines and requirements – part 1

- Tue Nov 4 – Estimation of 1/3 of stories and first sprint planning
  - Youtrack and Github repositories on the shared sheet
- Thu Nov 13 – Demo 1
  - Estimations of at least 2/3 of the stories (at least 16)
  - License indication (name and/or link) on the shared sheet → Workshop #7
- Thu Nov 27 – Demo 2 – Release 1
  - The release consists in providing a Docker container with all the software
    - Plus instructions on how to start and use the system
    - Must contain the full deployment
  - Estimation of all available stories
  - Docker repository on the shared sheet
  - Docker container deployable by third parties

Deadlines and requirements – part 2

- Thu Dec 11 – Demo 3
  - Sonar cloud repository on the shared sheet → Workshop #8
  - Sonar cloud analyses performed and track of TD management
- Thu Jan 8 – Demo 4 – Release 2
  - Docker container deployable by third parties
  - Sonar cloud analyses performed and track of TD management
- Fri Jan 16 – Teaser video on YouTube (private)
  - Private link on the on the shared sheet

⚠ Info sul progetto nelle slides (app per cittadini per interagire con il comune al fine di segnalare eventuali problemi o cose; ispirati ad IRIS@Venice)

2) FREE SOFTWARE

With UNIX and GNU, the target was to have a PC with all free software (from OS on up).

Public domain = category of creative works that are unprotected by intellectual property law. Since these works cannot be owned, they are free for anyone to use, adapt, reproduce, or distribute for commercial and noncommercial purposes.

4 freedoms:
0) freedom to run the program as you wish, for any purpose
1) freedom to study how the program works, and change it so it does your computing as you wish
   [precondition = access to source code]
2) freedom to redistribute copies so you can help your neighbor
3) freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes
   [precondition = access to source code]

A software is **free software** if the user can **enjoy all the 4 freedoms** (quindi è legato alle freedom, non al fatto che sia gratis!!!). The 4 freedoms are possible thanks to a copyright license.

GNU GPL (General Public License) → 4 freedoms + copyleft obligation (***what is free, must remain free***) [copyleft allows users to modify and redistribute sw, provided that any derivative work shall maintain the same license].

⚠ Open source indicates slightly weaker criteria than those provided for free software

Permissive licenses (es. **MIT**, **Apache**…) allow to use, modify and distribute the sw with minimal or no restrictions

Linking → allows free/open-source code to be linked to proprietary code without extending the restrictions of open source to the entire project (es. **LGPL**)

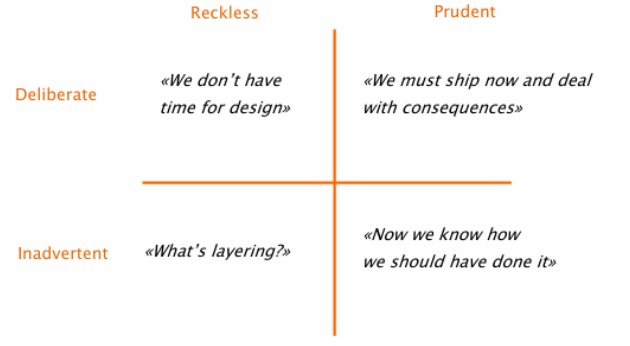
3) TECHNICAL DEBT (TD) + SONARCLOUD

TECHNICAL DEBT (TD) = imperfections in a software system that were caused by lack of time or by intentional choices and that run the risk of causing higher future maintenance cost. It's the **gap** between a perfect code and just make it working (perfect code = doc up-to-date, 100% test coverage, compliance with good programming practices, high modularity) [> gap, > long-term problems].

Example:

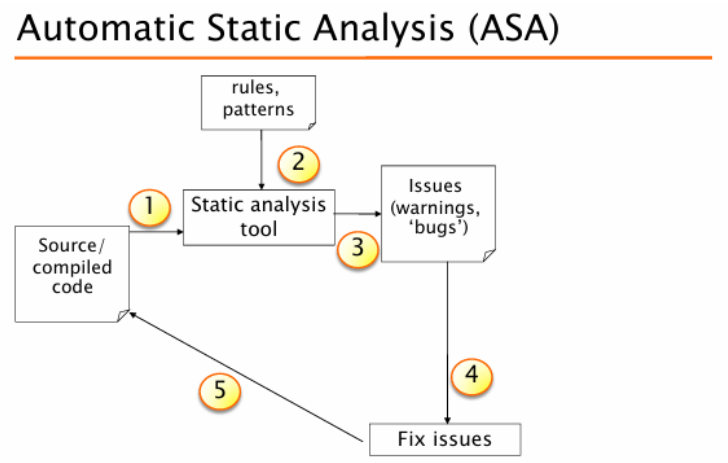
You take out a loan to buy a car:
- The cost of the car is 20.000 €
- You can spend now only 5.000 €
- The **principal** on the loan is 15.000 €
- The bank charges 5% **interest rate**
- The **interest expense** is 750 €

Technical Debt Quadrant



TD Types → defect debt (latent defects not yet fixed), design/architecture debt (bad organization of classes), documentation debt (outdated or incomplete documentation), testing debt (missing test cases)

TD causes → TECHNOLOGY, PROCESS, PEOPLE



Code Smell = disharmonies in code (bad practice)

DISHARMONIES

Identity disharmonies (identità delle classi/metodi)

- **God Class**: classe enorme che accentra la logica, ha tante responsabilità, usa molti dati di altre classi
  - Riconoscimento:
    - **ATFD** alto = usa molti attributi di altre classi
    - **WMC** alto = metodi molto complessi
    - **TCC** basso (< 1/3) = poca coesione interna
- **Feature Envy** (a livello di metodo): metodo che usa più dati di altre classi che della propria
  - ATFD alto, **LAA** (uso di attributi locali) basso, pochi **FDP** (pochi fornitori esterni ma molto usati)

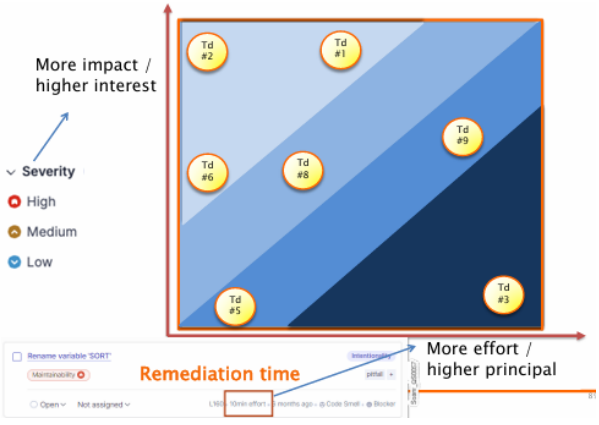
Collaboration disharmonies (collaborazioni fra classi/metodi)

- **Intensive Coupling**: un metodo chiama tantissimi metodi ma concentrati in poche classi → legame troppo forte con pochi collaboratori
- **Dispersed Coupling**: un metodo chiama molti metodi sparsi in tante classi → è "sparato" ovunque
- **Shotgun Surgery**: un metodo è chiamato da molti punti diversi: ogni modifica richiede tanti micro-cambi in molte classi

Classification disharmonies (gerarchie di classi)

- **Refused Parent Bequest**: sottoclasse che praticamente non usa (o rifiuta) ciò che eredita; ha identità propria
- **Tradition Breaker**: sottoclasse che introduce tanti servizi scollegati da quelli della superclasse, non la specializza davvero

Prioritizing TD items with **SonarQube**



4) AGILE CONTRACTS

Basic cost models:

- **Fixed cost** → cost defined for the whole project; requires well-defined scope, conditions and time; no surprise or flexibility for client. For the dev's side, there's a deadline and a fixed budget
- **Time & Materials** → cost defined for the unit of work (es. person-hour [**ph**]); doesn't require detailed upfront knowledge, but budget must be flexible

Contract key points:

- PROJECT **SCOPE** (and how much it can change)
- **DELIVERY CYCLE** (single or multiple iterations?)
- **PRICING** (budget, total cost and timing of payments)
- CHANGE **CONTROL** (how changes in scope are managed?)
- **ACCEPTANCE** (the definitive ok on the product or part of it; usually made using testing)
- **DELIVERABLES** (what exactly is expected [sw, documentation…]; intellectual property?)
- **TERMINATION**
- **WARRANTIES** & **LIABILITIES**

AGILE CONTRACTS → goal: successful project; foster system thinking (avoid silo mentality); involved lawyers should understand agile principles

Agile	Traditional
Time: Fixed	Time: Variable
Cost: Fixed	Cost: Variable
Scope: Variable	Scope: Fixed

So key pillars are fixed price work packages (sprints), early termination option, flexible changes, additional work always possible, ranged estimates (attenzione che non è vero che: agile non stima i requirements prima dello start; agile requirements devono per forza cambiare).

AGILE CONTRACT key points:

- PROJECT **SCOPE** = not an exact and unchanging project scope (target-cost and progressive)
- **DELIVERY CYCLE** = at end of each timeboxed iteration (sprint) deliver a deployable system
- **PRICING** = pay each iteration after acceptance (so every iteration has an agreed price)
- CHANGE **CONTROL** = change is at the core of agile (flexible scope and changes in relationship between parties [defined in contract])
- **ACCEPTANCE** = contract should define the framework for acceptance only
- **DELIVERABLES** = avoid list of deliverables (distract focus from working software); documentation may be useful for maintenance, not earlier
- **TERMINATION** = unique change option: the ideal termination model is to allow the customer to stop, without penalty, at the end of any iteration (early termination is a positive event in agile)
- **WARRANTIES** & **LIABILITIES** = early and frequent delivery mitigate liabilities (issues discovered soon and so fixed) [warranties are similar]

Contract models:

- **FPFS** (Fixed Price Fixed Scope) → easy to manage from customer, salesforce and manager; but worst for project success (lead to technical debt accumulation). Flexibility is limited (just replace existing requirements with new ones of equal effort; change priorities/order…)
- **VPVS** (Fixed Price Fixed Scope) → before every iteration customer and supplier agree on goal; customer exposure is limited; variations:
  - Capped Price Variable Scope = non-binding release backlog (allows flexibility)

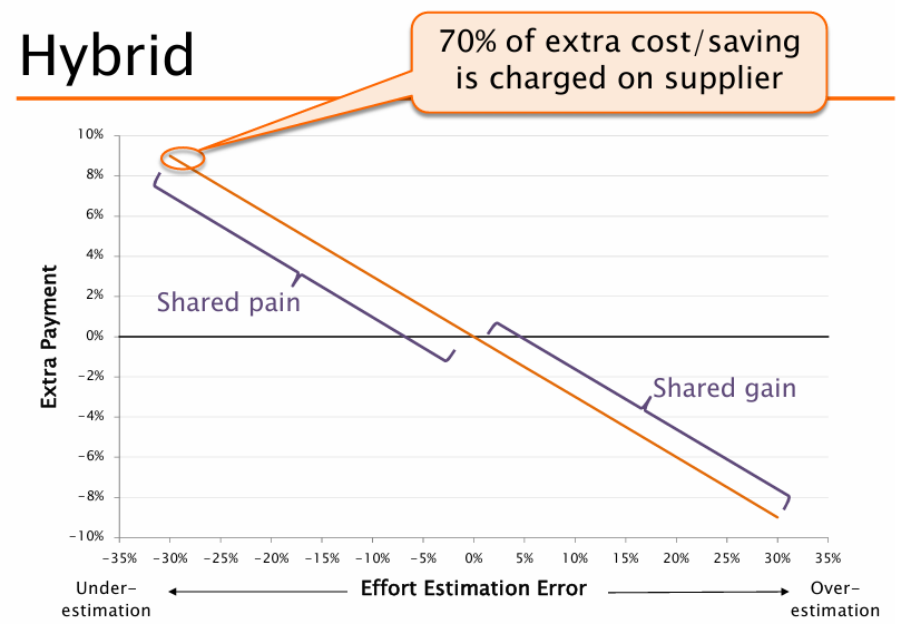
- Capped Price Partial-Fixed Scope
- Fixed Price Variable Scope

⚠ **Pricing schemes**:

- **Fixed** Price (*lump sum*)
- T&M (**Time & Materials**) → fit perfectly agile (iterative to solve issues)
- Fixed price **per iteration** (as fixed price, but smaller scope; but flexible)
- Fixed price **per UoW** (Unit of Work = running and tested features [working sw is the primary measure of progress]); issues is "business value vs size points" (which loose correlation and impact estimation)
- **Hybrid** (shared pain/gain) ↓
- Pay per use

⚠ Risk shifting and sharing → aligning motivations of both parties (sharing both customer and supplier; shifting is the risk on the party that is accountable: **requirement risks** in the hands of customer [what], **technical risks** in the hands of supplier [how])

- **Shared pain/gain**
  - Target cost = cost definition + actual cost tracking and sharing during project execution, to determine the adjustment (*Payment = Target Cost + Target Profit + Adjust*). Key elements: high transparency, collaboration for continuous improvement, test-driven acceptance…
  - Hybrid = price composed of price for effort (discounted) + price per UoW:



  - Multi-phase variable model = projects have a changing risk profile over time; multiphase adjust contract type to accommodate such change
  - Profit sharing

5) DEVOPS

DevOps = combinations of cultural philosophies, practices and tools that increases an organization's ability to deliver applications and services at high velocity. They aim to let work together the **DEV team** and the **OPS team** (operations). In practice, it's the union of people, process and products to enable continuous delivery of value to our end users.

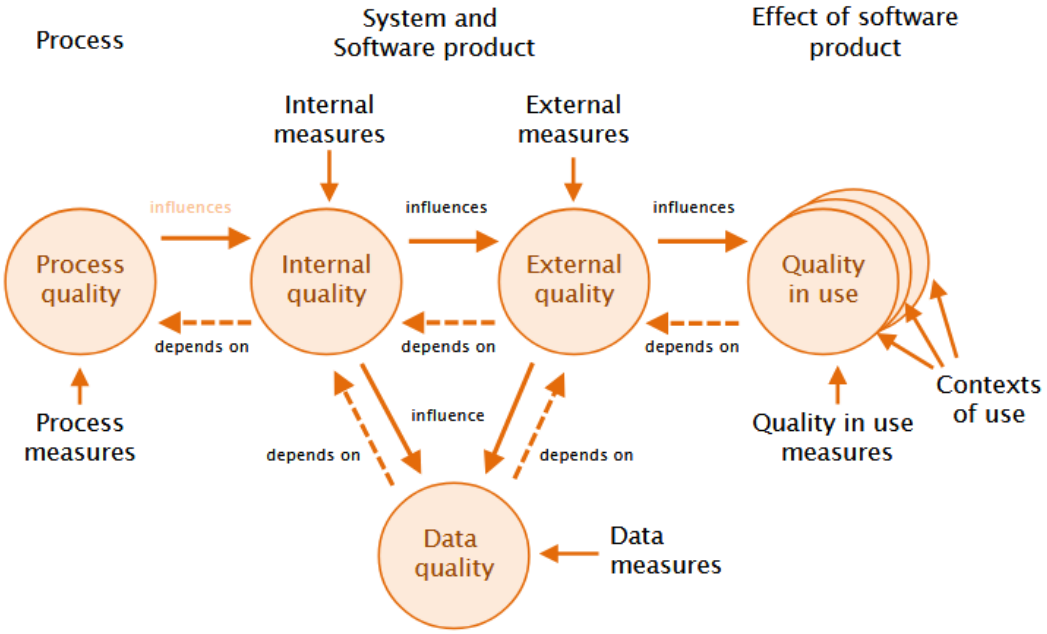
⚠ DevOps non è solo per startup o per progetti open-source, non rimpiazza Agile, è compatibile con la sicurezza e la compliance!

DevOps **principles** → **CAMS** (Culture, Automation, Measurement, Sharing); the 3 ways: system thinking, feedback loops, culture of experimentation

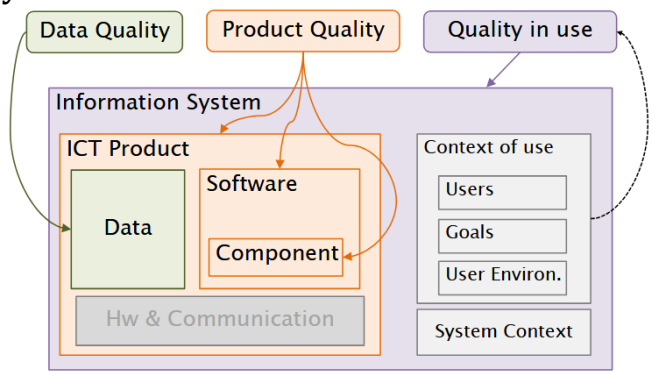
⚠ Dev e Ops hanno skill diverse e conoscenze diverse, che devono essere combinate per dare values allo end user (per questo combinati in DevOps, ma fanno cose diverse, ma collaborando Dev + Ops)

6) SOFTWARE PRODUCT QUALITY (ISO 25000)

ISO SQuaRE (Software product Quality Requirements and Evaluation):
- Internal Quality = values, formats, relation (internal of entity)
- External Quality = technological environment
- Quality in Use = context of use (*)
- Data Quality = data handled by the system



Target entities and quality models:

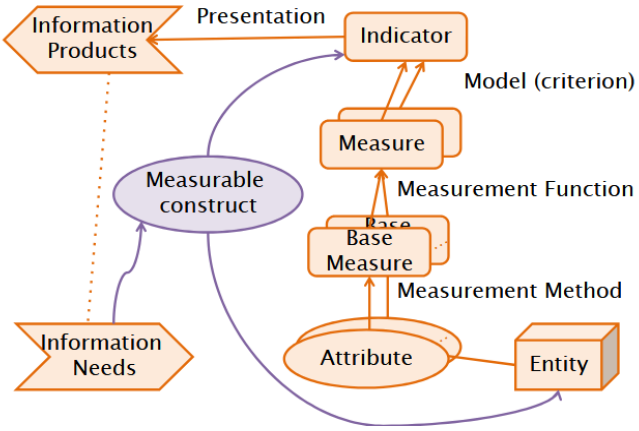


In particular the ISO 25000 family divides as:

- 2500X → Quality Management
- 2501X → Quality Model
- 2502X → Quality Measurement
- 2503X → Quality Requirements
- 2504X → Quality Requirements

MODEL STRUCTURE:

- **Characteristic** (main aspects [es. usability]):
  - Functional suitability = capability of a product to provide functions that meet stated needs of intended users (completeness, correctness …)
  - Performance efficiency = capability of a product to perform its functions within specified parameters and be efficient in the use of resources (time behavior, resource usage, capacity …)
  - Compatibility = capability of a product to exchange information with other products (interoperability …)
  - Interaction capability (Usability) = capability of a product to be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction (operability …)
  - Reliability = capability of a product to perform specified functions under specified conditions for a specified period of time (availability …)
  - Security = capability of a product to protect information and data (non-repudiation, integrity …)
  - Maintainability = capability of a product to be modified by the intended maintainers with effectiveness and efficiency (modularity …)
  - Flexibility = capability of a product to serve a different or expanded set of requirements or contexts of use (adaptability …)
- **Sub-Characteristic** (specific aspects [es. accessibility])
- **Measure** (measurement function to evaluate a characteristic) → construct = concept/topic of study (abstract); MEASURE = **operational construct**
- **Measure element** (base measure)



MEASUREMENT = empirical objective assignment of values to entities (product, process, resource) to characterize a specific attribute. Scale types can be nominal, ordinal, interval, ration, absolute (> richness, < simplicity se si va in ordine)

SMART indicator = Specific in purpose + Measurable + Achievable + Relevant + Timely

Interpretation of rating:

- between excellent and acceptable = min level of measure for providing opportunity
- between acceptable and unacceptable = min level of measure to avoid risk

Robustness = understandability + processing cost + significance + frequency + structuredness (***you get what you measure***)

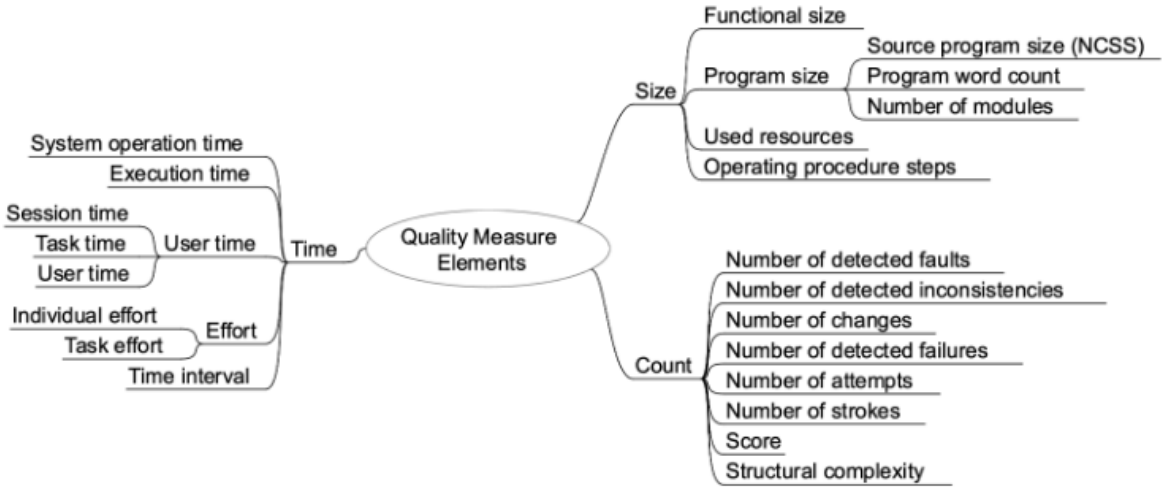
⚠ **Campbell's Law** = the more a metric counts for real decisions, the greater the pressure for corruption and the more it distorts the situation it's intended to monitor

⚠ **Goodhart's Law** = when a measure becomes a target, it ceases to be a good measure

BASE SOFTWARE MEASURES

- **Product metrics** → internal attributes (some easy to measure, other controversial and automated measurement); *quality assurance* = internal attributes can be measured during development to predict and control external ones
- **Process measures** → duration, effort, number of events and subjective measures
- **Resource metrics** → magnitude, cost, quality, *productivity* (amount of output/effort input = product/ process) [resource = input for sw development (personnel, materials, tools, methods)]

QMEs (Quality Measure Elements):



Vediamo questi 3 punti nel dettaglio:

1. **SIZE** → functional size: COSMIC FP measures functional size of a software counting how many movements of data occurs between user/system/archive to satisfy the requirements. Use Case Points Principles: UCP = (Actor types + Use case types) * Tech Complexity * Environmental factors
   ⚠ LOC (Lines Of Code) → most intuitive (count the n° of lines of code) [easy to understand, but hard to measure precisely]
2. **TIME**
3. **COUNT**

Mc Cabe Cyclomatic Complexity = complexity of the control flow (control flow is represented as a CFG; V(G) is the n° of base paths in G); if G is a strongly connected graphs, V(G) = #E - #N + 1 (a typical CFG is not strongly connected, unless we add an edge from final to initial node). It's well defined from a mathematical point of view, but focus on code complexity (strongly correlated with LOC)

Design metrics → $\boldsymbol{LCOM} = 1 - \frac{\sum mA_i}{m*a}$ where m = n° of methods in class, a = n° attributes in class, mA_i = n° of methods using attribute A_i (but design metrics lacks of validation)

Quality in Use model (*) [***old vs new***]: