1 Requirements

standards are described in IEEE Standard 830

1.1 *analysis*

analyzing, documenting, validating and managing software or system requirements.

1.2 *complete*

The requirement is fully stated in one place with no missing information.

1.3 *consistent*

The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.

1.4 *elicitation*

gathering requirements from end-users,customers and other stakeholders

1.5 *Entity-Relationship*

mostly used to describe business processes. For example relations between managers in companies

1.6 *functional*

What a system will do.
Authentication, Administrative functions or External Interfaces, Reporting Requirements or Historical Data

1.7 *generalizable-elements*

Classes,Association, Stereotypes, Signals and Use Cases

1.8 *natural language*

pros:

- almost everyone can understand them
- almost anyone can write them

cons:

- one requirement can be written in so many ways
- There is no easy way to modularise natural language requirements

1.9 *non-functional*

How a system will do something.
bandwidth, availability,backup, documentation or maintainability

1.10 *stakeholder(user-centered design)*

Definition. A stakeholder in the architecture of a system is an individual, team, organization, or classes thereof, having an interest in the realization of the system.
Primary stakeholder: people who is affected of the outcome of the project(users)
secondary:people who can affect the outcome of the project but is not themself affected of it(designers)
Direct stakeholder: Concerned with the day to day activities of the project(designers)
Indirect stakeholder: people affected by the end result(users)

1.11 *tracable*

The requirement meets all or part of a business need as stated by stakeholders and authoritatively documented.

1.12 *unambigious*

The requirement is concisely stated without recourse to technical jargon or acronyms

2 Design & Architecture

2.1 *architectural view*

- Logical view : The logic view of the functionality to the end-user
- Development view : The development view illustrates a system from a programmer's perspective
- Process view :Explains the system processes and how they communicate
- Physical view : The physical view depicts the system from a system engineer's point of view.

2.2 *Behavioral design patterns*

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

2.3 *client-server*

- two-tier,thin-client: -heavy load on server,-significant network traffic
- two-tier.fat-client:+distribute workload on clients,-needs to update software on server

- three-tier:+map each layer on separate hardware,+possibility for load-balancing

2.4 cohesion and coupling

Cohesion:How uniform and easy understanding the code in software is. High cohesion is prefered
Coupling: how much all the modules are entagled into another. Low coupling is prefered

2.5 Facade pattern

hide complexity of a system from the user

2.6 implementation view

Software packages, libraries, frameworks and classes

2.7 layered architecture

pros:

- reduced complexity
- easier to maintain code
- easier to add new functionality
- easier to test
- allows to reuse code

cons:

- restricting communication to adjacent layers
- keeping coupling between layers reduced.

2.8 observer pattern

Objects have observers which are notified if the object changes

2.9 pipe-and-filters

advantages:

- simplicity:easy to see the software flow.
- Maintenance and reuse
- concurrent Execution:Each filter can be implemented as a separate task and be executed in parallel with other filters.

disadvantages:

- Interactive transformations are difficult – Filters being independent entities designer has to think of each filter as providing a complete transformation of input data to output data.
- No filter cooperation.
- Performance – may force a lowest common denominator on data transmission

2.10 strategy pattern

In computer programming, the strategy pattern/policy pattern enables an algorithm's behavior to be selected at runtime.

- defines a family of algorithms
- encapsulates each algorithm
- makes the algorithms interchangeable within that family.

3 Testing

A fault(bug) leads to an error that can lead to a failure of the system

3.1 acceptance testing

is a test conducted to determine if the requirements of a specification or contract are met.

3.2 big-bang-testing

A system wide test that test all modules at once. Very time efficiant

3.3 bottom-up

Start from the bottom and write drivers to run test-cases on the target level. Drivers are normally easier to write than stubs. Can write better tests for the upper level afterwards.

3.4 code-coverage(white box testing)

Code-coverage means how much of the code in the project is being tested. For example a project with 100% code-coverage means that every single line of code have been tested.

3.5 configuration management

- Lock-modify-unlock: When you check something out you lock that object so it cant be modified.
- Centralized-modify-merge: When a conflict occur the system tries to merge the changes. If its not able to you can merge the chages manually.

- Decentralized-modify-merge: The merging is done off the system. You ahve to mail patches around and then one person manually patches it and commits it.

3.6 *continuous integration*

Everytime a developer commit a piece of code a build server builds the project and runs some tests to see that the integration with the new code works.

3.7 *daily build*

Normally an automated procces that build the project daily and runs some smoke tests.

3.8 *function testing*

bases its test cases on the specifications of the software component under test

3.9 *installation testing*

An installation test assures that the system is installed correctly and working at actual customer's hardware.

3.10 *performance test*

is generally executed to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload

3.11 *repo phrases*

- branch: create a new branch
- tag: a version that you want to preserve
- merge: merge to branches in on branch
- trunk:main body of project from start to finish

3.12 *sandwhich testing*

A target layer is defined in the middle of the program and testing is carried out from the top and bottom layers to meet at this target layer. top and bottom layers can be tested in parallel and can lower the need for stubs and drivers. Hard to select best target level.

3.13 *smoke test*

Very basic fast tests. For example does the program run?, does it open a windows?, does the main button do anything?

3.14 *test case*

every test-case needs test-case ID,inputs,expected outputs

3.15 *top-down*

Start from the top and write stubs for the lower lovels and work down. Might needs alot of stubs. Finds design errors early

4 Planning & Processes

4.1 *Agile methods*

agile manifesto:

- Individuals and interactions: in agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- Working software: working software will be more useful and welcome than just presenting documents to clients in meetings.
- Customer collaboration: requirements cannot be fully collected at the beginning of the software development cycle, therefore continuous customer or stakeholder involvement is very important.
- Responding to change: agile development is focused on quick responses to change and continuous development.

practices:

- adaptive planning
- evolutionary development
- early delivery
- continuous improvement
- encourages rapid and flexible response to change

pros:

- People and interactions are emphasized rather than process and tools.
- Working software is delivered frequently (weeks rather than months).
- Close, daily cooperation between business people and developers.

- Continuous attention to technical excellence and good design.
- Even late changes in requirements are welcomed

cons:

- In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.

when:

- when change is needed during the project
- when a project needs to start right away
- major decisions can be made during the development

when not:

- the development team is spread over the world. You need face to face communication
- if you dont have a project manager on site that can make quick decisions
- if people affected by the project need complete clarity on solutions before starting the project
- You have a fix deadline with a fixed set of requirements

4.2 delphi method

a panel of expert are asked anonymous how much time a task will take. Someone summaries the answers and the groups meets and discusses the different answer and converges towards the correct answer.

4.3 extreme programming(agile)

practises:

- Pair programming: means that all code is produced by two people programming on one task on one workstation
- User stories (planning): desribes features
- Small releases (building blocks): With XP, you develop and deliver the application in a series of small, frequently updated versions.
- Collective ownership: No one person owns or is responsible for individual code segments.
- Coding standard: All team members write code in the same way, using the same styles and formats.

4.4 four parameters

calender time,resources,features,quality

4.5 gantt chart

- task: for example ”fix bug nr 34”
- phase: for example ”design phase”
- milestone: for example ”network module finished”
- tollgate: for example ”design stop”

4.6 incremental model

The projects consists of alot of small builds which become greater with time with more features added each time.
pros:

- generate working software quickly
- easier to test and debug with small iterations
- customers can respond to each built

cons:

- requires good planning and design

- needs a clear and complete definition of the whole system so it can be broken down
- resulting cost might be greater
- addiotional functionality that is added might arise problems related to the system design

4.7 *iterative model*

no overhead

pros

- building and improving the product step by step. Hence we can track the defects at early stages
- we can get the reliable user feedback.
- ess time is spent on documenting and more time is given for designing

cons:

- Each phase of an iteration is rigid with no overlaps
- Costly system architecture or design issues may arise because not all requirements are gathered up front for the entire lifecycle

4.8 *project plan*

project description:

- Background to the project
- Relevant constraints (budget etc.)
- Project Goal
- Start and expected end date

project organization

- Roles
- Knowledge / skill
- Training
- Communication and reports

Time and Resource Plan

- Milestones
- Tollgates
- Deliverables
- Activities
- Resources

Risk Management

- Risks,Probability and Impact
- Mitigation and Contingency plan

4.9 *prototype model*

Remainds alot of incremental model. You build a prototype of the system to find out what is missing and what needs to be refind or added to it and you refine the prototype. Itś good for system with alot of end-user interaction such as online systems and web interfaces.
pros:

- users are involved in the development
- errors can be detected early
- missing functionality can be identified easily

cons:

- leads to implementation then repairing
- may increase complexity of system

4.10 *risk planning*

$$identifyrisk \to analyze \to plan \to monitor$$

Risk avoidance:
for example not taking on a project and thuse avoiding the risk that comes with it.
risk transfer: for example using a subcontractor for some part of the project that your company do not have the knowledge of.
risk acceptance: you accept that there are a risk and you put it on a watch list and take no further action.
risk mitigation: planning and taking actions to reduce the effects of the risks. For example don't start any war that might end in a terrorist attack.
Contingency plan: minimize the effects if the worst would happen. Don't let alot of the employes travel on the same plan because there might be a terrorist attack.

4.11 scrum

- burn down chart:is a graphical representation of work left to do versus time.
- daily scrum meeting: What have you done since yesterday? What are you planning to do today? Any impediments/stumbling blocks? Problems are the scrum master responability to resolve
- scrum master:being a facilitator, makes shure tho project moves forward
- product backlog: is a prioritized list of features
- sprint: short durations milestones
- sprint retrospective. 15-30 minuts to look at what is and what is not working

4.12 waterfall model

good with fix-rpice contracts

pros:

- This model is simple and easy to understand and use.
- It is easy to manage due to the rigidity of the model
- Waterfall model works well for smaller projects where requirements are very well understood.

cons:

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- No working software is produced until late during the life cycle.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

5 Quality factors

5.1 availability

(total time - down time) / total time. More os less how available the system is. A system with high availability seldom crashes and when it crashes id recovers fast.

5.2 correctness and functionality

Does the software have all the functionality stated in the specification?

5.3 cylomatic complexity

It directly measures the number of linearly independent paths through a program's source code.

5.4 depth in the inheritance tree

greater depth means greater complexity

5.5 flexability

possible to add/remove modules without breaking the system.

5.6 maintainability

How easy the software is to maintain meaning does the code follow any standars, are everything well documented an so on.

5.7 performance

How fast the software respons and how efficiently it uses ram and harddrive space. For example a software that does some easy calculations cannot use 16gb of ram.

5.8 portability

Can it run on several platforms?

5.9 reliability

Software reliability is the probability of the software components of producing incorrect output. Software should not wear out and continue to operate after a bad result.

5.10 scalability

Can run on newer machines(vertical scaling). Can run on several machines/-multi proccesing (horizontal scaling).

5.11 software audit

An independent examination of a software product, software process, or set of software processes to assess compliance with specifications, standards, contractual agreements, or other criteria

5.12 software inspection

peer review by trained individuals who look for defects using a well defined process.The goal of the inspection is to identify defects

roles

- Author: The person who created the work product being inspected.

- Moderator: This is the leader of the inspection. The moderator plans the inspection and coordinates it.

5.13 usability

How easy the system is to use. Mostly through UI:s

5.14 usability inspection

experts test the software without end-users

5.15 usability testing

users test the software and give feedback