0 Contents

1 **Software Quality** 1
- 1.1 Therac25 Rontgen overdosis (causes) 1
- 1.2 Rules for Good Quality 1
- 1.3 Requirements 1
  - 1.3.1 Requirements Overview 1
- 1.4 Overengineering and Underengineering 2
  - 1.4.1 Technical Depth 2
- 1.5 Software Aging 2
  - 1.5.1 Impact of complacency 2
  - 1.5.2 Tools for Quality 2
- 1.6 Maintainability of Code 2
  - 1.6.1 McCC 2

1 Software Quality

1.1 Therac25 Rontgen overdosis (causes)

- unknown, single developer
- reused parts by former employees without documentation
- Bad UI -> misuse
- bad documentation
- missing processes for identifying problems -> testing
- missing processes for solving issues after accidents happen
- self written real-time OS instead of proven standard
- Software only tested with simulator, not with real hardware

1.2 Rules for Good Quality

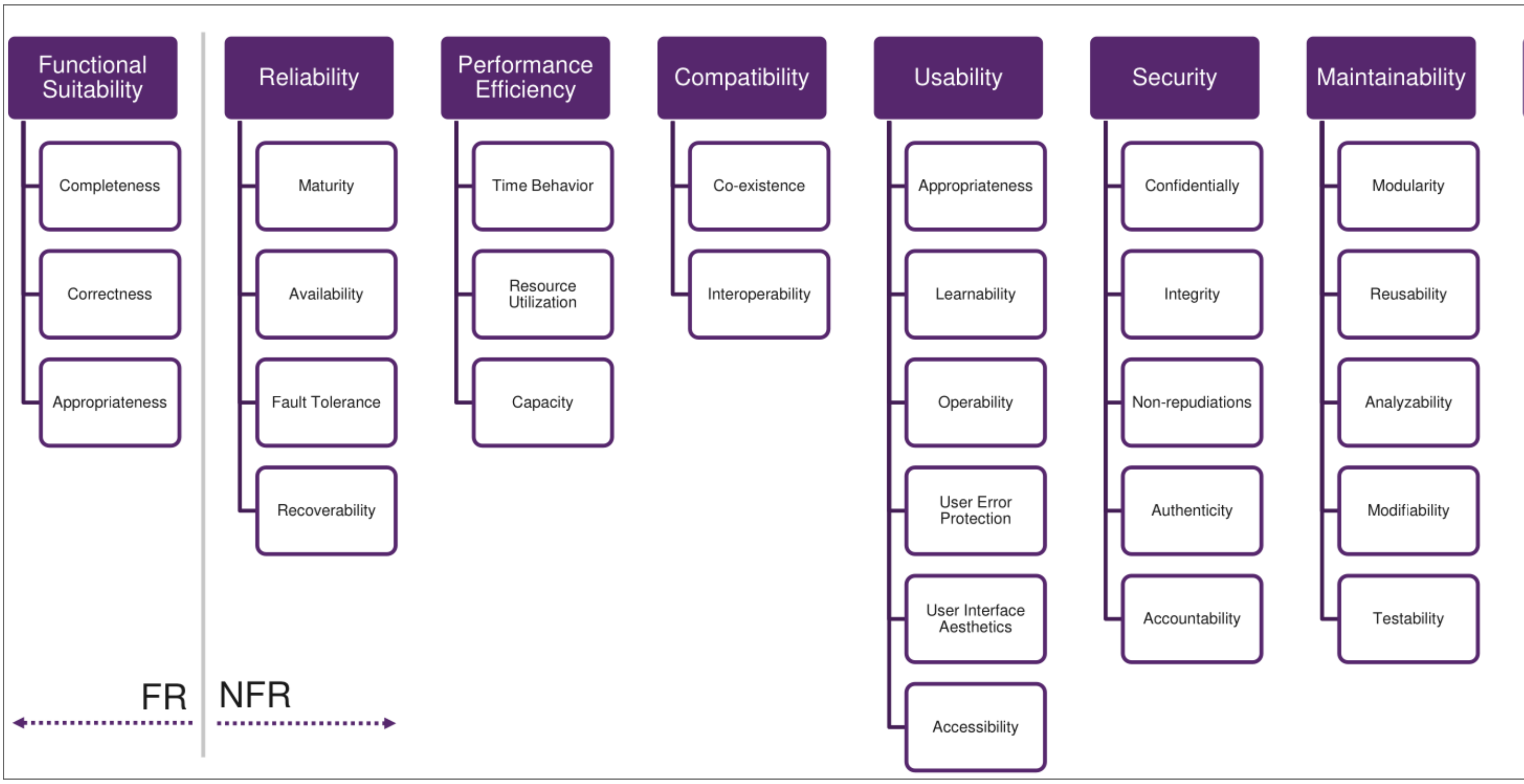
1. Conform to requirements
   Write and test according to the customers needs, a smartphone app may crash, but a plane definitely not!
2. Use multiple layers of quality control
   Multiple layers of control will have a better chance of elmiminating the risk -> swiss cheese model

1.3 Requirements

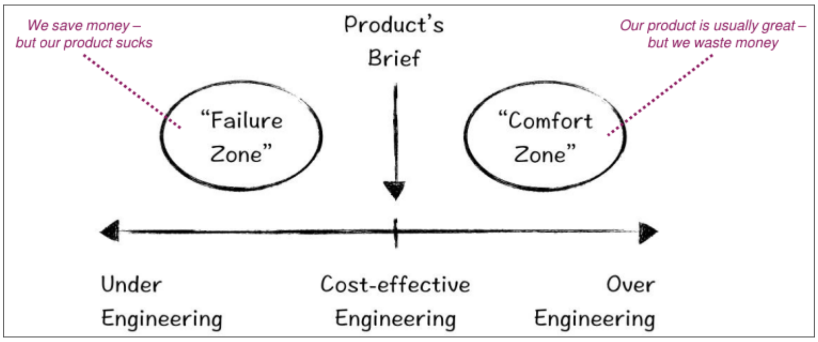
Functional:
- acquired together with the customer
- Userstories etc
- easy to verify -> customer says yes
- WHAT does the system do?

Non-Functional:
- Defined by the customer AND the developer
- Developer stories etc
- Harder to verify
- HOW does the system work? PSQL? GIN? BEVY?

1.3.1 Requirements Overview



1.4 Overengineering and Underengineering



The right side wastes money, while the left side saves money at the cost of reputation.

1.4.1 Technical Depth

This is essentially just a backlog of things that you should still do. Ex. Tests, or better documentation, or a cleaner implementation.
Technical depth will often accumulate and you will end up at the left side, -> underengineering.
It is important to keep track of this with developer stories and try to achieve *more overengineering, rather than being complacent*.

1.5 Software Aging

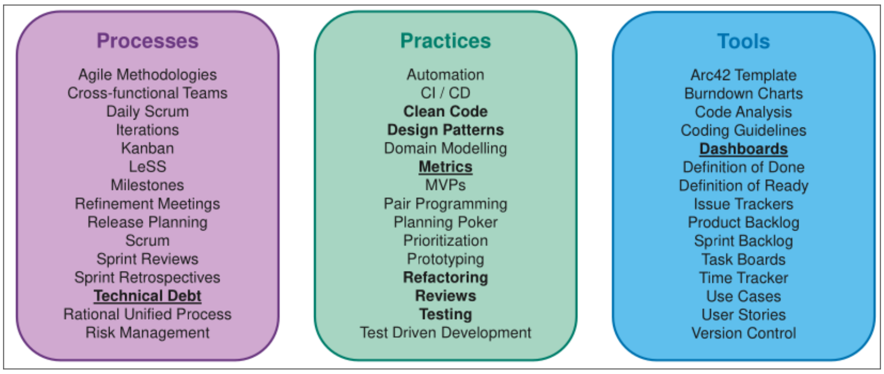
- Causes
  - lack of change in technology
  - update without thinking
- Costs
  - inability to keep up
  - reduced performance
  - decreasing reliability
- Remedy
  - stop the deterioration
  - design for success
  - plan ahead

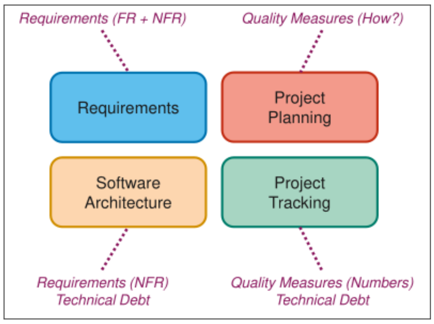
Important: Over time, the quality will pretty much always decline! -> wayland over Xorg!

1.5.1 Impact of complacency

If you start to accept bugs, it will likely lead to more acceptence of more bugs.
Eg. do not accept things such as not tested code, as it will likely reduce the quality of code in the long run!

1.5.2 Tools for Quality

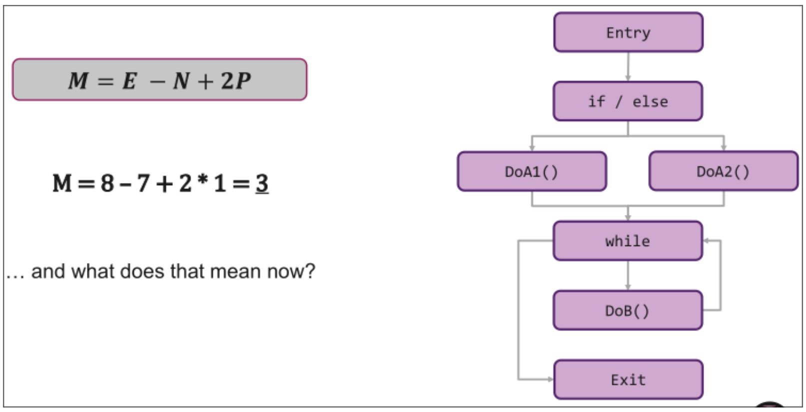


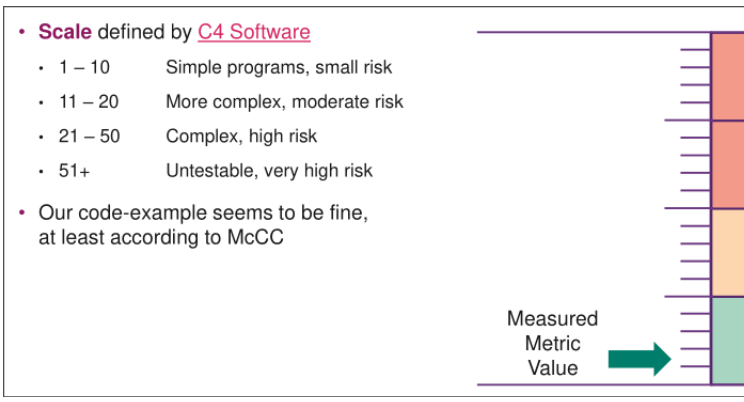


1.6 Maintainability of Code

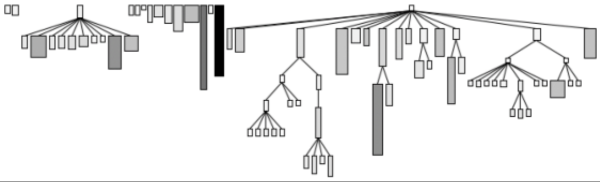
- POSIX function -> 1 functionality per function
- Code Coverage
- Code Smells LOC
- Lines of code CC
- Cyclomatic Complexity McCC

1.6.1 McCC





Do NOT calculate this by hand, only let a tool do this task for you.
There are also a lot of tools that visualize the metric in order to stand out more to human eyes.



Node	Class
Edge	Inheritance Relationship
Width	Number of Attributes
Height	Number of Methods
Color	Number of Lines of Code