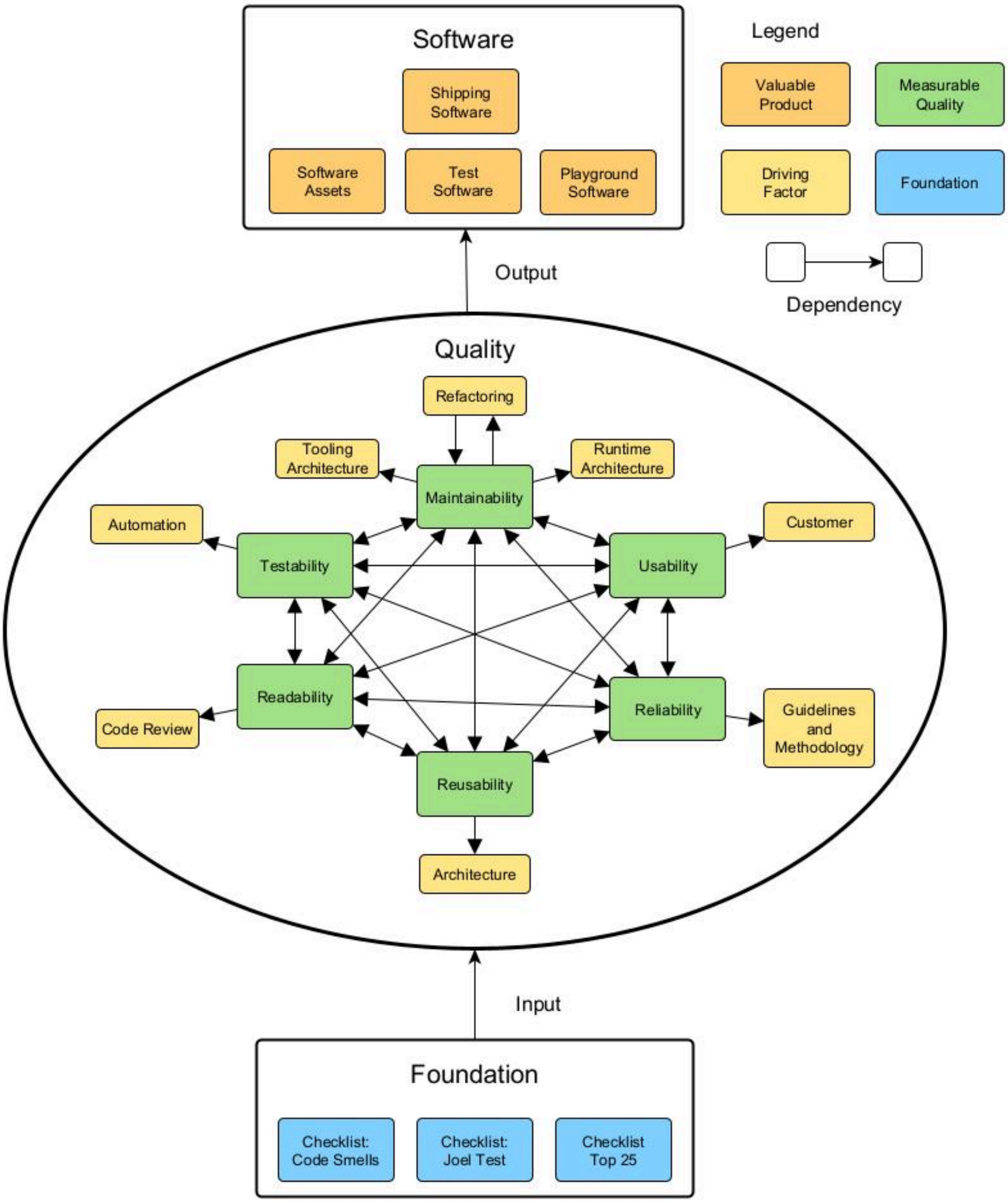
Software Maintainability Cheatsheet



Software

Shipping Software

The cost / time / quality - triangle of your sellable software product is determined by the *maintainability* of your software. Keeping maintainability and feature production are two different aspects of the development activity.

Software Assets

Any of your libraries, framworks or whatever software that is reusable, reduce R&D cost. Removed software is removed money.

Test Software

- The amount of test software is proportional to test automation.
- *Test automation reduce R&D cost*
- There are many free nice tools available for example gtest or catch2 for c++, robot framework for integration testing, AHK2 for gui test automation, ...
- Caveat: I disagree to the promotion of "80% coverage"- everything must be tested after all.

Playground Software

- Creeativity is driven by experiments
- Experimental software is the least organized

Quality

Measuring quality is the basis of understanding the role of R&D in regard to that and helps to understand the order of importance to launch the right programs for quality improvement.

Furthermore, the quality points mentioned here all depend on eacht other.

Maintainability

Rationale

Refactoring is mandatory to keep maintainability up. It is almost a natural process, that the maintainability of software deteriorates over time. You need high test depth and coverage to make that safe.

But there are barriers:

- Management drop it, disempowering developers.
- Feature driven CI/CD-Micromenagement makes it "a thing", i.e. it can't be as in Clean Code "The boy scout rule" as something that happens "naturally" all the time. Even variable renaming get's rejected.

Keeping refactoring up reduce R&D cost.

Runtime Architecture

If you use C++ and cmake, you have two big areas to care about: The structure of your source code, like what classes do you have and so on. Let's call this your **runtime architecture**.

Tooling Architecture

The organisation of your files, folders, libraries and so on, in your repository, is another kind of architecture, let's call it **tooling architecture**. Both need maintainance, especially for cmake, and if you have many buld options and different target plattforms. This part is maybe much easier if you program in rust, but you have it anyway. Even if you go for something like Cube Ide, which offers you something by making a lot of it automatic. But you still have to manually fiddle a lot of details, like include paths, and so on.

Measurement

Compare the logic complexity of a feature to the amount of work it needs to implement it:

Logic complexity:

- How much text and diagramming is needed to describe it ?
- How many conditions has it ?

Implementation complexity:

- How many lines of code to change ?
- How many files to change ?
- How many build steps ?
- How much "bouncing" ? (e.g. rejection of pull requests)
- How much additional work has to be done, e.g. documentation is extra (confluence vs. doxygen)

Keeping complexity low reduce R&D cost.

Testability

Rationale

Test driven is the modern industry standard of software development, for many reasons, e.g. compliance in regulated markets.

Test software should adhere to the same principles as shipped software. Don't orphanize it, don't theat is as second class software (i.e. lowering standards here).

Test software can provide extra added Value:

- Force developper to think about the user experience of his software.
- Give solid documentation by example.
- Makes refactoring more safe.
- Safeguards our Business.

Quality standard and added value of test software reduce R&D cost.

Measure

- How much do you cover with tests ?
- How much of your tests are automated ?
- How about the Maintainability, Usability, Reliability,

Readability

Rationale

To make software readable, you must performe code reviews. Only another person can tell if your software is readable, because your blind spot is that "you understand it", no matter what you write. **Code reviews must aim for readability.**

Measure

- Have a look at the famous WTF/Minute image.
- How much do you jump from one file to another when you read the code ?
- Does your Diagrams fit a visible area ?

Usability

Rationale

Usability is what your customer pays for.

Measure

Compare these error messages:

```
Error 0xA83C. Shutting down.
```

```
Operation Halted due to Maintenance Requirement.
Refill the water tank.
[ABORT] [INSTRUCTIONS]
```

Reliability

Rationale

Unpredictable errors usually stem from overcomplexity.

Unreliable software drives away your customer and risks damage to live, environment and business. One single tiny error at the wrong time in the wrong place can be devastating.

Bugs destroy your business.

There are a lot of questionable opinions regarding reliability.

- "It's not a bug, it's a feature"
- "Bugs are inevitable"
- "It's just a glitch"
- ...

My proposal: **Go for ZERO Bugs.**

Measure

- How long is your bug list over time ?
- It the length of your bug list increasing or decreasing ?
- Categorize Bugs by the identified causes.

Reusability

Rationale

If you can re-use a piece of software that cuts the R&D cost in half (ok, exaggerated, but you get the point).

Reusability reduce your R&D cost.

Measure

- How many times is your library/framework/module/... used ?
- How many times do you make a "new generation", starting from scratch ?

Foundation

A lot of work has been done by others in the attempt to solve that very fundamental problem of software maintainability. A lot of it is freely available these days. Some things are overcomplicated and/or questionable, e.g. "Maintainability Index". But generally rejecting this means you must invent the wheel again.

Applying literature about software maintainability reduces your R&D cost.

I like to offer Three examples here. This is based on my personal experience: Whenever i witnessed the downfall of software development, it was not primarily due to the lag of professionalism. I have seen supersmart software engineers, using all the tools and methods I can think of, and yet fail to keep up the procuctivity in the software department. When that happend, i always found gross violations and flagrant disregard of the following material.

The Joel Test

The Joel Test

These simple 12 questions disassemble your whole software developent department. And yet, no one knows it, or it makes people upset, or ther start an excuse marathon.

It is the emergency life support for your failing codebase.

It may hit hard, but it is as effective as it is hitting hard.

TOP 25

Dave B. Stewart: Twenty-Five Most Common Mistakes with Real-Time Software Development

This often copied and adapted list is an early source of the idea of working with a checklist to get everything covered.

Top 25

It is specifically for real time systems, so this is for the embedded systems and machine control.

Use another more suitable list of common mistakes as neccessary.

List of Code Smells

List of Code Smells)←This is an adopted list, the orignal can be foud in the book Clean Code by Robert C. Martin.

This was just the beginning of me starting to understand that there is more on making good software than my then hobby approach of constant improving by trial and error.

This list is long, apparently questionable, controversal and maybe outdated. Many altered copies exist.

I think is has several very sharp insights.

Table of Contents

- Software Maintainability Cheatsheet
- Software
  - Shipping Software
  - Software Assets
  - Test Software
  - Playground Software
- Quality
  - Maintainability
  - Testability
  - Readability
  - Usability
  - Reliability
  - Reusability
- Foundation
  - The Joel Test
  - TOP 25
  - List of Code Smells
- Table of Contents