



SBA
Research

Tackling Software Rot

Advancing from Programming to the Engineering of
(More) Sustainable Software



What Do They Have in Common?



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <arpa/inet.h>

void serveur1(portServ ports)
{
    int sockServ1, sockServ2, sockClient;
    struct sockaddr_in monAddr, addrClient, addrServ2;
    socklen_t lenAddrClient;

    if ((sockServ1 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Erreur socket");
        exit(1);
    }
    if ((sockServ2 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Erreur socket");
        exit(1);
    }

    bzero(&monAddr, sizeof(monAddr));
    monAddr.sin_family = AF_INET;
    monAddr.sin_port = htons(ports.port1);
    monAddr.sin_addr.s_addr = INADDR_ANY;
    bzero(&addrServ2, sizeof(addrServ2));
```

They Share Another Property – They Both Rot

Eiffel Tower riddled with rust and in need of repair, leaked reports say

Experts say €60m repaint before Paris hosts 2024 Olympics is only a cosmetic makeover



📷 The Eiffel Tower is the fourth most visited cultural site in France. Photograph: Lesdronographes/AP

When it was completed in 1889, the Eiffel Tower - Paris's Iron Lady - was expected to stand for 20 years before being dismantled. One hundred and thirty-three years on, the tower is still standing, less by design than through diligent maintenance.

Avoid Software Rot with Strategic Maintenance

by Mike Swieton



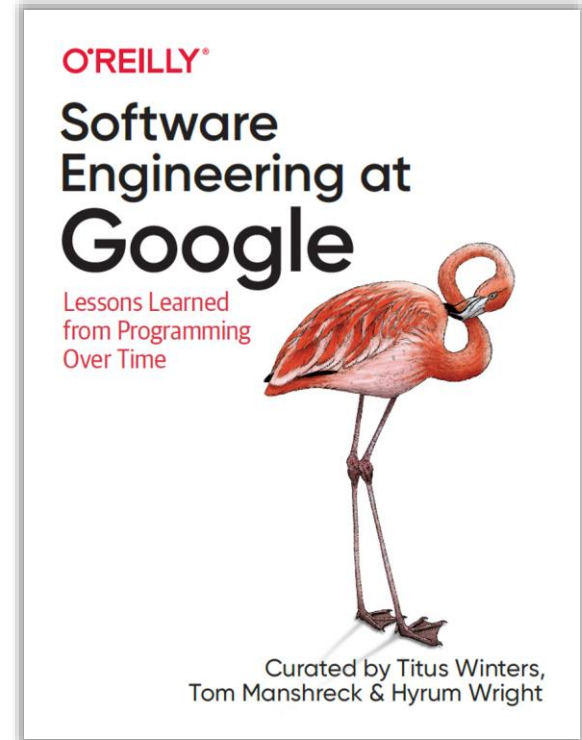
Software applications (like homes, cars, and nearly everything else) need maintenance. Even when the software itself doesn't change, the systems/devices it runs on and the larger software environment are always moving forward. This is called Software rot. The longer legacy software sits on the shelf, the harder it becomes to work on it. It may even stop working on its own one day. This surprises my clients frequently, so I wanted to talk a bit about why this happens and how to plan for the software maintenance effort.

How Could It Come so Far?

- Human tendency to **focus** on new shiny features and interesting projects – maintainability is unrewarding and boring (“no glory in maintenance”)
- **Economics** and wrong **incentives**
- **Short term thinking** – maintainability creates short time costs while benefits only materialize in the future (& offloading of responsibility – let maintenance be the problem of someone else)
- Human **psychology** – even if we know something makes sense and would benefit us (living healthy, exercising regularly) it is quite hard to resist immediate temptation (e.g. marshmallow experiment) and not to postpone “tedious” (long term beneficial) activities

So How Can We Approach This Problem?

- We should strive to **advance** from **programming** to the **engineering** of (more) **sustainable software**
- Disclaimer: No easy short cuts or fundamental answers – illuminating paths towards finding those answers
- Source:
[Online Version](#)
[Pdf Version \(via Wayback Machine\)](#)



Programming Vs. Software Engineering

- Two main distinctions: **life time** and **team size**
- **Programming:**
 - mainly about development
 - **single** person(or only a few), **short** (expected) **life time** of the software
 - e.g. software project for diploma thesis
- **Software engineering:**
 - deals with development, modification & maintenance
 - **team effort**, requirements regarding **life time** and **adaptability**, often involves trade offs
 - can be thought of as” **multi-person** development of **multi-version programs**” and “**programming integrated over time**”

What is Sustainable Software?

- Systems that **meet** the **needs** of the **present** without compromising the **ability** to **meet future needs**.

(John Butlin, Our common future. Oxford University Press, 1987, pp. 383)

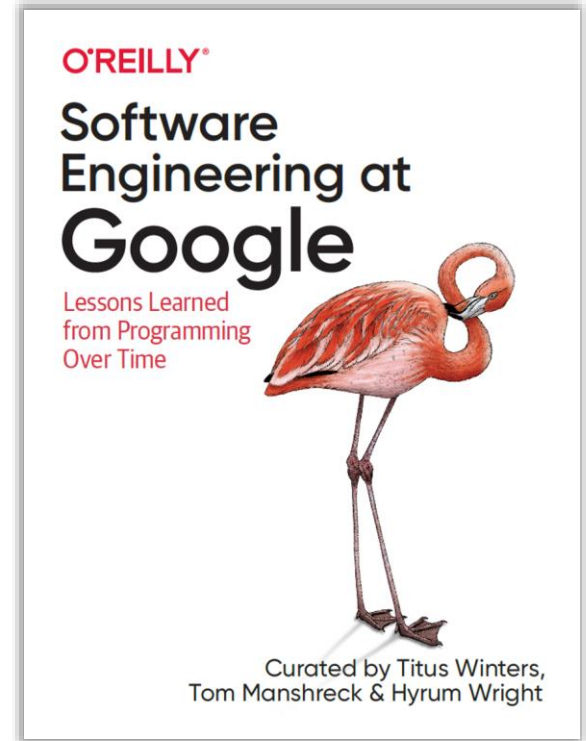
- Software is sustainable when, for the **expected life span** of the code, we are **capable** of **responding to changes** in dependencies, technology, or product requirements (without incurring intolerable cost).

We may choose to not change things, but we need to be capable.

(Software Engineering at Google)

Some Useful Concepts

1. Initial vs. sustained velocity
2. Style Guidelines & Readability Process
3. Code Review process
4. Treat Documentation as code
5. Code as Liability
6. Test automation
7. Dealing with Dependencies
8. Frequent Rewrites
9. Deprecation



1: Initial Vs. Sustained Velocity

- Natural tendency to **defer security, reliability & maintainability concerns** until some point in the future
 - May indeed **increase** your project's **velocity early** in the project's lifetime
 - Usually **slows you down** significantly **later** (and can come with substantial costs!)
 - This **trade-off** should be clearly understood
- Time initially invested in support of these goals is **not time lost** but time **saved from the future**
- Security, reliability & maintainability should be **embedded** in the team **culture**
- More details: [Building Secure & Reliable Systems](#)

2: Style Guidelines & Readability Process I

- Especially important in big environments where multiple different people will be confronted with written code over its lifetime
- Code **consistency**, **readability** and **understandability** are crucial for maintainability
- Code will be read over its lifetime many times more than written -> **optimize code** for the **reader** – not the author
- **Restrict usage** of error-prone, unusual, complicated or surprising constructs -> future team members might not understand than (even if the author does)
- External version available: <https://google.github.io/styleguide/>

2: Style Guidelines & Readability Process II

- Code changes have to undergo a **readability process**
- A person who is a certified readability expert of the respective language has to give his/her **approval**
- Try to **minimize effort** it takes to **comply** with the rules & automate rule enforcement (e.g. with static analysis tools, formatters)

3: Code Review Process

- **Each change** has to undergo this process before being committed
- Not part of a dedicated team – **everybody's responsibility**
- Three main steps which require approval
 - **Check for correctness** (incl. availability of tests) & comprehension
 - **Readability** review
 - **Owner** of the affected codebase part has to agree
- Creates additional effort and cost (short term)
- **Long term benefits** due to **knowledge transfer**, increased code **quality**, **maintainability** & consistency (+ saves time usually needed for debugging, trouble shooting etc.)

4: Treat Documentation as Code

- Documentation is critical for **productivity** & **maintainability** of the code
 - Main problems: either **not existing** (or fragmented) or **outdated/wrong**
- Reason: **few incentives** for creating documentation
 - > it's additional effort & benefits only materialize in the future (+ no glory in documentation)
- Solution: cultural change & writing documentation must be as frictionless as possible -> **integrate** it into the **tools** and **workflows** of **developers**
- [g3doc](#): framework for storing documentation next to code
 - Supports version control, reporting bugs, review process for documentation changes, submission of documentation updates together with the according code change etc.

5: Code as Liability

- **Code** itself **doesn't bring value** – the **functionality** it provides **brings value**
- **Code** itself **carries cost** (creation, maintenance, bugs introduced)
 - prefer simpler code which is easier to maintain
 - Focus on “functionality delivered” and not code produced
 - Don't try to measure developer productivity by counting “produced” LOC (will have adverse effects - [Goodhart's Law](#) and [The Tyranny of Metrics](#))
- **Inconsistent & duplicated** code **inhibits** changes & **maintenance** tasks
- Some of the best modifications to a codebase are deletions – **removing dead** or **obsolete code** benefits the **overall health of a codebase**
- For **new code** check: is this change necessary, does it improve the codebase?

6: Test Automation

- Code might be **modified dozens of times** over its **life time**
-> **writing tests** incurs short time cost but huge long term benefits
- Write code with testing in mind. **Invest** early on in “**testability**” – hard to “add” to an existing project
- **Make it easy to run tests** (automation)
- For **each change** (& bug fix) **test cases** must also be **provided** (otherwise change gets blocked)
- Write **clear tests** – if unclear and no documentation it might be removed in the future
- Requirements for tests (e.g. speed, determinism, reliability) and pitfalls (flaky tests) see: [All your Tests are Terrible – Tales from the Trenches](#)

7: Dealing with Dependencies

- One of the **biggest & unsolved challenges** in software engineering
- We not only have to deal with single dependencies but a whole **network of dependencies**
 - **Leap of faith – Inspection** of (all) dependencies rarely done
 - Problems like **transitive dependencies & diamond dependencies**
- Adding **external** dependencies comes with a (hidden) **cost**
 - in certain cases it can be more sustainable to develop it internally
 - Considerations for importing: [checklist](#) & [article](#)
- **Live at Head**: new & uncommon approach for dealing with (internal) dependencies (would be a paradigm change for OSS)
 - **Owner** of a **dependency** must make sure that an **update doesn't break** one of its **consumers, consumers** are **always** using the **latest version**

8: Frequent Rewrites

- Pretty **costly** but also some crucial benefits
- A few years old software usually designed around an older set of requirements (& technology) and is typically **not** designed in a way that is **optimal** for **current requirements**.
- Oftentimes it has **accumulated** a lot of **complexity** (& **technical debt**)
- [Rewriting code](#) **cuts away** unnecessary accumulated **complexity** that was addressing requirements which are no longer so important.
- Help to **ensure** that code is written using **modern technology** and **methodology**
- Way of **transferring knowledge** and a sense of ownership to newer team members

9: Deprecation of Code

- **All systems age** & need maintenance – gets continuously harder as systems move away from current technologies & platforms
- Already **think about deprecation** when **designing** a new system (e.g. nuclear power plant)
- Deprecation gets harder the longer a system is being used (due to [Hyrum's Law](#))
- **Dedicated team** should be responsible and support uses of the resp. component at moving away
- **Preventing backsliding** via static analysis tools and whitelists in the build system to ensure that no new dependencies are introduced to deprecated system

Further Interesting Concepts

- Storage of (almost) all source code in a [Monorepo](#)
- Trunk based development (especially due to scaling/efficiency)
- Reproducible Builds

Additional Literature

- Software Engineering at Google (Paper)
<https://arxiv.org/ftp/arxiv/papers/1702/1702.01715.pdf>
- Building Secure & Reliable Systems
https://static.googleusercontent.com/media/sre.google/en/static/pdf/building_secure_and_reliable_systems.pdf
- Preventing Software Rot
<https://software.rajivprab.com/2020/04/25/preventing-software-rot/>
- Milk or Wine: Does Software Security Improve with Age?
https://www.usenix.org/legacy/events/sec06/tech/full_papers/ozment/ozment.pdf
- They Write the Right Stuff (1996)
<https://www.fastcompany.com/28121/they-write-right-stuff>

Questions & Discussion




Philipp Reisinger

SBA Research

Floragasse 7, 1040 Wien

+43 660 543 62 74

preisinger@sba-research.org

 Bundesministerium
Klimaschutz, Umwelt,
Energie, Mobilität,
Innovation und Technologie

 Bundesministerium
Digitalisierung und
Wirtschaftsstandort



wirtschafts
agentur
wien
Ein Fonds der
Stadt Wien



FWF
Der Wissenschaftsfonds.

