DOI: 10.59647/978-617-520-936-3/1

A problem with modern software - the cost of underlying complexity of used dependencies.

Myroslav Shpak, student[1 i], (ORCID: 0009-0005-2372-8373)

[1]*The Kyiv National University of Construction and Architecture, Kiev, Ukraine.*

ANNOTATION

Thesis is to emphasize on how expensive can be the cost on dependencies or abstractions in modern software and underlying code bases that is getting harder to maintain over the years. Why abstractions, in most of the cases, are more hurtful than useful, and how to prevent maintenance, performance or security issues from occurring in your commercial or passion project? What are the downsides or upsides of simple software, and why the subject is relevant to everyone doing software development.

Keywords: Software complexity, dependency, maintenance, abstractions, over-engineering,

1. INTRODUCTION

Ever since computers got spread enough and the optimization of underlying technologies got more advanced, people have thought of many features or ideas they can implement in the future software, or change the ways in the process of making the software itself, since there was no previous limitations in computer performance or availability of the knowledge, way more people could do programming without years spend on studying mathematics, algorithms or computer science. However, little did they know or seemed to care, how easy it is to overcomplicate the problems, that have been already solved way before the marketing had a touch on the software development, and how horrible can be the state of software in the future.

2. THE TRUST IN THE THIRD-PARTY DEPENDENCIES

After Object-Oriented Programming got popular, due to marketing or promises it given to the developers of solving their need to program specific solutions, the idea of what software actually is, and how it should be done got a noticeably different compare to programming concepts that got established in their meanings beforehand in the 80s.

After that period a lot of software and computer technologies entered the public, so that every day user would have a better variety in development tools or a software to use. Many programming languages had been made over the years with the goal to simplify the programming experience with whatever the user would have in mind.

For regular programmers who wanted better portability or ease of use, many programming languages had been made: Python, Java, Lua, Clojure, just to name a few.

For the internet technologies getting wider spread in the society, more tools arrived to make everyone capable of creating their website: PHP, JavaScript, many Different User Interface frameworks and libraries.

However, an apparent issue had become obvious quite soon, the underlying complexity that majority of computer tech developers and users liked to ignore or accept as a for granted.

As it obvious to say, the simpler is an idea, the easier it is to understand and implement it. Even if the final goal sounds unreachable, as an example - creation of a video game, or a programming language compiler, the truth is that the entire way of making it come to life, is just trivial solving of primitive problems and writing simple algorithms, that everyone can study by themself in a spare time, due to one of the essential skill of a programmer - is being able to break huge complex problems and data structures, into simpler more intuitive one.

Sometimes the problem can get a bit harder to understand or learn, maybe the subject is too boring, or the time investment is too high, either way a good solution to that is a concept of the dependency.

Dependency can be anything that is required to assist your project, a library, some other program, a framework, etc. The idea is to trust the other more reliable source to have your problem solved for you, instead of wasting the time to do it yourself.

This is indeed very useful concept, everyone have used it in any way, via a standard library or maybe some other software they didn't want to make themself: tools like 'core-utils' that are present in any Linux distribution or other third-party program.

However, this has quite a drawback. By using a library the user, has to put trust into the developer(s) and maintainer(s) of a solution they wanna use. And that's exactly what hadn't bothered many of the modern software developers.

There is a bottom line of what could require a library or a framework, many subjects can be researched and done in the matter of hours, which can easily be performed by a newcomer in programming or software engineering, even with that in mind, many web technologies, game engines or other software infrastructures tend to use libraries that take just one or two functions to implement

The consequence of putting trust into external dependencies leads to layering of other people solutions in which the developer has no control over.

A great example of such layering is any of a modern Reactive user interface frameworks for the website creation.

React as for 2023 takes around 1.5 thousand dependencies and occupies over 360 Mb of free space. This is "needed" amount of carriage to just draw fancy buttons and track program state in the browser, that is already capable of creating, deleting, changing and indexing of any element type user might want.

For reference GTK4 (version 4.0), an UI library for any modern operating system on desktop, has source weight of 103Mb while having no more than a dozen of core operating system dependencies, and some external platform specific libraries like X lib, Xrendr, X11, Wayland, GDI, OpenGL, and a couple of others.

Even tho both of the examples are pretty big software examples, one can be used in a way bigger area, has virtually no performance issues or bugs and doesn't have any underlying abstraction layer besides the OpenGL, and Window creation

dependencies that are present only due to different ways of doing so in every operating system mentioned.

3. THE COMPLEXITY OVERHEAD

Despite many problems appearing to be complicated and have a need to come with a very complicated solution on the given problem, in reality most of the time, the actual solution tends to be simple and doesn't require a complex implementation from the software design or code part. Most complex implementations derive from the intent to make a library, a program or a framework to be generic, and capable of solving any given problem on the topic it builds for. As an example a C library Raylib has enough functionality to be used in most of the common cases in making games, graphical applications and yet this library has no support for multi-windowing, however there are plenty of ways of doing that by using sub-processing, or process-instancing. The workaround can be easily extended on the base library features. In previously mentioned library - GTK, multi-windowing takes noticeable chunk of code (20k lines of source and header files) that is almost as big as 25% of the entire Raylib implementation. The amount of useful code does not really excuse the added complexity for a feature that could be added when needed by a single developer.

There are many more examples of overengineering the software that can lead to unforeseen consequences. The paradigm of "The more moving parts there is in a system, the easier for it to break" can be seen in everywhere: in compiler designs, in modern web technologies ranging from website generators and frameworks finishing off with libraries and NPM modules, even in the game market, notable chunk of popular modern games are made with one of the 10 most popular game engines, despite that a lot of indie games could be done faster by being programmed from scratch.

Common pitfalls in making software usually comes from having prone to errors approach that will lead to the program unpredictable behavior, the practices that can easily lead to problematic design of the software are: Premature optimizations - the process of optimizing the software before having it fully designed, this not only can cause Undefined software behavior with an already complex solution, but also be very hard to catch and debug to determine the reasoning, usually it's a good practice to design software "the slow way" and only once everything works, make attempts on making it run fast. Partial development - it usually happens when the developer works on specific parts of the program way more than on the other, in the end, having only a specific part of it work well, while others are essential just placeholders. A commonly seen advice from highly experienced game and compiler developers, people who work only with complex pieces of technology, is to prototype software fast, and then iterate on the mistakes that had been made in the process. From previous statement the last common pitfall arise - Fear of refactoring. Refactoring may take a lot of time which could be spent on what's already is working, but usually for most projects out there, it doesn't take too much effort to iterate on what essentially has been done, thus redoing a poorly designed project can highly improve the speed, reliability and source code readability of the project to levels that could be reached previously. However, refactoring should only be done when needed, redoing the same work over and over again without a highly advised reason to do so.

4. HOW NOT TO REINVENT THE WHEEL, CONCLUSION.

Both of the topic might appear to contradict each other, if the software has to be simple to be predictable, how does it suppose to be fast? While the third-party dependencies are possibly unreliable, how should anyone do a full solution from the scratch, and make it do what we want it to do?

Both of mentioned aspects are what defines the actual hard part about software development, the task of the software engineer is to design such a solution that would satisfy the requirements, but also could be self-sustained. The dependencies are mostly unavoidable in the modern world due to each individual market or environment having their own standards, but absolutely possible to minimize the amount of used dependencies to manageable volume.

Usually it's a good idea to eliminate smaller dependencies, functionality of which can be easily added into the main project, also sometimes project may need to implement its own solution to have either performance or functionality, that can't be easily or concisely done in the used dependency.

If project might need to be compatible with some other technology, it's a usually wise to introduce compatibility overhead only when absolutely needed. This mindset can seriously cut off impulsive decision making, and help find more plausible approach for reaching the goal with lesser effort and software complexity.

In the end complexity in the software is required, but isn't as often as its current seen in many proprietary or open source projects, and the best way to solve it is to be very specific on what the project tries to solve, what are the goals and the taboos. Many examples of concise simple software present in the Open Source segment, were developers and maintainers show not only passioned work on their creations, but also serious responsibility on the quality of the product they deliver, even considering the work being done for free.

Literature references:

[1] Node module example called “is-even” as example of absurdity of modern dependencies, website. URL: https://www.npmjs.com/package/is-even?activeTab=code

[2] Package count as 2022 blog post on medium. URL: https://medium.com/frontendweb/find-how-many-packages-we-need-to-run-a-react-hello-world-app-695fbb755af7

[3] React dependency count reaction by an open source developer as for 2023: URL: https://youtu.be/XAGCULPO_DE?t=293

[4]GTK librarys source code. URL: https://gitlab.gnome.org/GNOME/gtk/-/tree/main/gtk?ref_type=heads

[5]Game engine usage frequency chart URL. : https://gamefromscratch.com/game-engine-popularity-in-2024/

[i] *The work was performed under the supervision of Yevheniia Shabala*