

DPI

De Programmatica *Ipsium*

DE PROGRAMMATICA IPSUM

Issue 047: Rust

August 1st, 2022

Table of Contents

| | |
|--|----|
| Issue 047: Rust | 5 |
| The Double Denim Of Software Engineering | 9 |
| The State Of Rust In 2022 | 13 |
| Michael Feathers | 19 |

Issue 047: Rust



August 1st, 2022

Welcome to the forty-seventh issue of *De Programmatica Ipsum*, dedicated to the subject of *Rust*. In this edition:

- Graham doubts the need to rewrite¹ all the wheels in Rust.
- Adrian observes the growth of Rust² in the past 15 months.
- In the Library section³, Graham reviews *Working Effectively with Legacy Code* by Michael Feathers⁴.

ISSUE 047: RUST

Enjoy this issue! Please subscribe to our free newsletter⁵ to stay updated about new releases, share the articles on social media, or contribute⁶ if you would like to support our work.

Cover photo by Jonathan Hanna⁷ on Unsplash⁸.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/the-double-denim-of-software-engineering/>
- ² <https://deprogrammaticaipsum.com/the-state-of-rust-in-2022/>
- ³ <https://deprogrammaticaipsum.com/category/library/>
- ⁴ <https://deprogrammaticaipsum.com/michael-feathers/>
- ⁵ <https://deprogrammaticaipsum.com/newsletter/>
- ⁶ <https://deprogrammaticaipsum.com/contribute/>
- ⁷ https://unsplash.com/@funnelhead?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ⁸ https://unsplash.com/s/photos/rust?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Double Denim Of Software Engineering



By Graham Lee

I am told that I am irresponsible. Why? Because I continue to use and maintain working software. I am told that my plan—to use my experience, along with the library of well-tested functions that I and my colleagues have built up over the decades—barely counts as a plan at all.

Ideas of what is “correct” in software development evolve¹ more like trends in lapel width and shoulder pad size in men’s suits than like best practice (or “state of the art”) in an engineering discipline. Firstly there is the practical coveralls of machine language, and then the stylish but expensive higher-level compilers. Subsequently, people like Backus, Naur, Perlis and their friends say that no, you are not cool.

If you want to look great and feel great you need some procedural vogue in your wardrobe. And so it goes. Procedural programming is the blue jeans of computing fashion: robust workwear for a narrow category of occupations that soon finds itself sold to every office clerk and football dad in the land.

Dijkstra tries to introduce a more formal variant: programming’s equivalent of a Canadian tuxedo—by suggesting that not only might programmers want to create procedures but that those procedures might ought to be correct. It does not catch on, except in a loud and opinionated niche subculture that—just like the double denim wearers of thrash metal—everybody else assumes is probably a Satanist. And so it goes.

Fads come and go but the dependable old blue jeans of procedural programming never quite go out of style, simply by never quite being in style anyway.

Alan Kay turns up at the end of the 1970s like some New Romantic-looking alien with his Smalltalk², and people pair his look with their comfortable blue jeans. Thus Java³ is the Spandau Ballet of programming: New Romo enough to be identifiable but mainstream enough not to worry your parents.

James Martin shows people how to open their own fourth-generation fashion houses, but that seems like hard work so they just carry on buying blue jeans.

RAD, codeless, functional: each tries and fails to become the new fashion paradigm. This decade, the school having its tilt at programming’s invincible windmill is type supremacy⁴. Type supremacy offers safety: write software the way we say and if the compiler likes it, it means it is not going to crash! What type supremacy actually delivers is safyness⁵: write software the way we say and *feel like* it might not crash so much.

Leading the charge for full-on ripping up the rulebooks, discarding the “square” denim your parents wear, and embracing the safety fashion as the new paradigm for reality is not so much the Rust community per se as their self-proclaimed paramilitary vanguard, the Rust Evangelism Strikeforce⁶. The RESF, or “jerk rs”, a sadly all-too-real subculture, believe that computers will not be safe and computing will not be realised upon this world of mortal humans until all of existing software is written in Rust⁷.

While others are saying that you should (almost) never rewrite your software⁸, the RESF would like you to rewrite your software *and everyone else's*. You wrote your web application in Rust? I mean, that is *okay...* but if you are still running it on that lame old insecure, unsafe OpenBSD written in lame old C then you (and your customers) only have yourself to blame.

The RESF ignore that the rest of society is perfectly comfortable in blue jeans and do not want to give them up—and gets a lot out of its blue jeans. Giving folks new implementation tools is a good way to get them to *get their implementations wrong*, when what they need to do is to understand a certain category of unsafe behaviours and ensure those behaviours are not realised. A plethora of tools can help us with that: type checkers yes, but also provers, testers, static analysers, symbolic executors, safe libraries, and more. For every Rust rewrite that is still in progress, there are a hundred JavaScript programmers extending their existing software using TypeScript⁹. A hundred C programmers adding scan-build and KLEE¹⁰ to their tooling. A hundred C++ programmers adopting concepts.

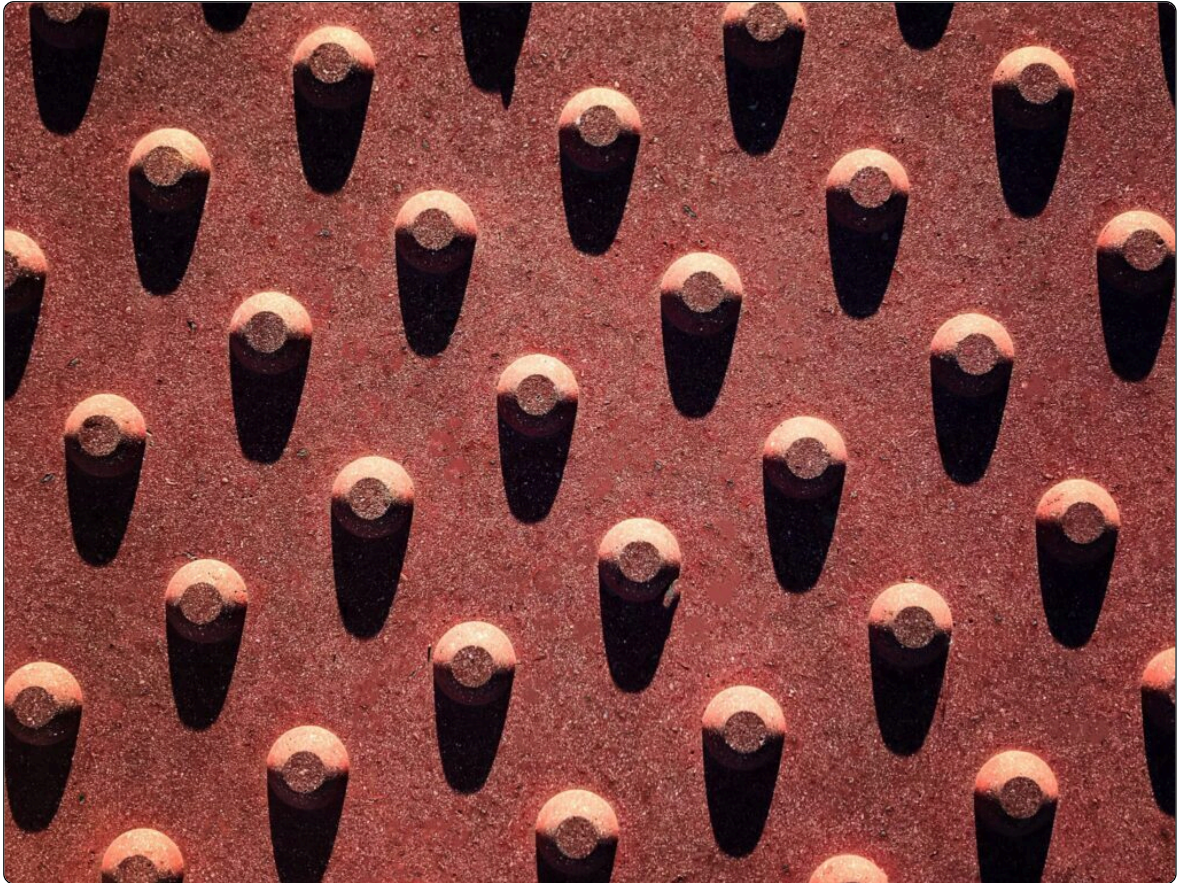
They still put their blue jeans on in the morning, but now they have both belt and braces to keep them up.

Cover photo by Ricardo Gomez Angel¹¹ on Unsplash¹².

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/what-is-behind-the-hype/>
- ² <https://deprogrammaticaipsum.com/what-smalltalk-was-not/>
- ³ <https://deprogrammaticaipsum.com/java-the-programmer-environment-that-has-it-all/>
- ⁴ <https://deprogrammaticaipsum.com/it-is-not-the-types-that-will-help-you/>
- ⁵ <https://blog.metaobject.com/2014/06/the-safyness-of-static-typing.html>
- ⁶ <https://classicprogrammerpaintings.com/post/682692428055134208/the-rust-evangelism-strike-force-howard-pyle>
- ⁷ <https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/>
- ⁸ <https://www.onstartups.com/tabid/3339/bid/2596/Why-You-Should-Almost-Never-Rewrite-Your-Software.aspx>
- ⁹ <https://deprogrammaticaipsum.com/where-does-microsoft-want-to-go-today/>
- ¹⁰ <https://klee.github.io/>
- ¹¹ https://unsplash.com/@rgaleriacom?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ¹² https://unsplash.com/s/photos/double-denim?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The State Of Rust In 2022



By Adrian Kosmaczewski

We published an article about Rust last year¹ in our edition about Modernism². Here we are back again in this field 15 months later, watching this language take the industry by storm and become an ever-increasing influence in history. This short article will provide an overview of Rust's most visible impacts, according to this author's biased eye.

Let us follow an upwards movement in the abstraction ladder for this review, starting at the bottom. On one side, Rust received knighthood status as the second

official language³ of the Linux Kernel. On the other, developers chose Rust to write the Redox⁴ operating system from scratch.

On top of such operating systems, we need userland programs. Developers are rewriting many of them in Rust⁵: `bat` instead of `cat`, `exa` instead of `ls`, `fd` instead of `find` and more⁶, sometimes with the same arguments and options, allowing adventurous users to `alias` right away. There is not only mere rewriting, though, as quite a bit of innovation is happening, and thankfully so. `Pijul`⁷ and `Jujutsu`⁸, for example, are not merely Git clones in the source code versioning market but new tools addressing shortcomings and borrowing inspiration from other projects like `Fossil`⁹.

Programmers need languages to create new software, and Rust is steadily becoming the new C for compiler and runtime (re)construction. `Artichoke`¹⁰ is Ruby made with Rust. `RustPython`¹¹ is Python made with Rust. `Deno`¹² is TypeScript and JavaScript made with Rust. And there are many more languages built with Rust¹³. There is even a book teaching you how to create your programming language¹⁴ with Rust and scripting languages you can embed¹⁵ inside your Rust apps or games. People are working on the interoperability of Rust with Python¹⁶ and many other programming languages.

We not only want CLI and TUI tools on our Rust-based operating systems but also need GUIs. `System76`, the maker of Linux laptops, is working in `COSMIC`, a desktop environment¹⁷ entirely written with Rust and based on `gtk-rs`¹⁸. The `Tauri`¹⁹ project recently reached version 1.0 and positions itself as a solid alternative to `Electron` for cross-platform GUI applications. But `Tauri` still requires web technologies on your desktop. What about the 100% native GUI app space? There is still a lot of work²⁰, with conflicting approaches and no clear winner yet. Some desktop apps like `1Password` run happily with Rust²¹ these days.

Full-stack and Cloud Native developers can use `Rome`²² to format, lint, and bundle CSS, TypeScript, HTML, and other languages used even higher on the abstraction ladder. They can use `Rocket`²³ as a replacement for Ruby on Rails or Django or `Actix`²⁴ and `Hyper`²⁵ for smaller web APIs. They can run WebAssembly in Kubernetes using `Krustlet`²⁶ or run their code in `Kata Containers`²⁷ much faster since it has been rewritten in Rust²⁸. Many benchmarks²⁹ show that Rust-based

web frameworks are among the fastest and most efficient available in the market today. Sadly Firefox fired the Servo³⁰ team in 2020; we might have had a fully-fledged Rust-based web browser by now.

Not all is green, though. While Rust is eminently great for apps, it is also great for malware, making it faster, smaller, and even harder to detect³¹.

Many companies are using Rust these days: apart from 1Password, mentioned above, there is Dropbox³², Google³³, and many more³⁴. Yet, there is still one practical roadblock to the adoption of Rust, which makes Go, C#, Crystal³⁵, and Dart³⁶ worthy opponents at this moment in various scenarios: a steep learning curve. Many programmers are baffled or downright confused by Rust's borrow checker, and even though Rust produces exceptionally efficient code, companies are skeptical³⁷ about their developers' productivity with it. To the point that Google argues that Carbon³⁸, its successor to C++, would be a better approach.

Thankfully, learning Rust is becoming easier every day. Developers can learn to read and write Rust code through rustlings³⁹. If that is not enough, Amos teaches you Rust in 30 minutes⁴⁰. The Rust Edu organization⁴¹ promotes it in the academic world, and Typst⁴² aims to replace LaTeX. Newsletters such as the Rust newsletter⁴³, RiB⁴⁴, Rust GameDev WG⁴⁵, and This Week in Rust⁴⁶ regularly reflect the innovations in this space directly in your inbox. There used to be another newsletter, RustTimes⁴⁷, but apparently, it is no longer in service. Last but not least, the same week this article hits the web, RustConf 2022⁴⁸ will take place. The Linux Foundation is teaching us how to write kernel modules in Rust⁴⁹.

Only time will tell whether developers will get used to the borrow checker, the same way they got used to garbage collection, runtime exceptions, functional programming, and many other seemingly "alien" concepts in the past 25 years.

In the meantime, we will watch with patience our industry rewrite all the wheels again, falling into the same traps and facing a new uncharted territory. Our craft has not yet reached sufficient maturity to understand that there is no silver bullet. There never was, and never will. As much as Rust seems like an excellent solution for many problems, it is not, and never will be, a replacement for our brains and our capacity to think.

ISSUE 047: RUST

Cover photo by Donald Giannatti⁵⁰ on Unsplash⁵¹.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/>
- ² <https://deprogrammaticaipsum.com/issue-32-modernism/>
- ³ <https://lkml.org/lkml/2021/12/6/461>
- ⁴ <https://www.redox-os.org/>
- ⁵ <https://zaiste.net/posts/shell-commands-rust/>
- ⁶ <https://zaiste.net/posts/shell-commands-rust/>
- ⁷ <https://pijul.org/>
- ⁸ <https://github.com/martinvonz/jj>
- ⁹ <https://fossil-scm.org/>
- ¹⁰ <https://www.artichokeruby.org/>
- ¹¹ <https://rustpython.github.io/>
- ¹² <https://deno.land/>
- ¹³ <https://github.com/alilleybrinker/langs-in-rust>
- ¹⁴ <https://createlang.rs/>
- ¹⁵ <https://www.boringcactus.com/2020/09/16/survey-of-rust-embeddable-scripting-languages.html>
- ¹⁶ <https://www.infoworld.com/article/3664124/how-to-use-rust-with-python-and-python-with-rust.html>
- ¹⁷ <https://blog.edfloreshz.dev/articles/linux/system76/rust-based-desktop-environment/>
- ¹⁸ <https://gtk-rs.org/>
- ¹⁹ <https://tauri.app/>
- ²⁰ <https://raphlinus.github.io/rust/gui/2022/07/15/next-dozen-guis.html>
- ²¹ <https://serokell.io/blog/rust-in-production-1password>
- ²² <https://rome.tools/>
- ²³ <https://rocket.rs/>
- ²⁴ <https://actix.rs/>
- ²⁵ <https://hyper.rs/>
- ²⁶ <https://krustlet.dev/>
- ²⁷ <https://katacontainers.io/>
- ²⁸ <https://www.zdnet.com/article/kata-containers-rewritten-in-rust-and-gets-a-major-speed-boost/>
- ²⁹ https://web-frameworks-benchmark.netlify.app/result?asc=0&l=rust&order_by=level512
- ³⁰ <https://servo.org/>
- ³¹ <https://www.zdnet.com/article/this-malware-has-been-rewritten-in-the-rust-programming-language-to-make-it-harder-to-spot/>
- ³² <https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine>
- ³³ <https://www.xda-developers.com/google-developing-android-rust/>
- ³⁴ <https://careerkarma.com/blog/companies-that-use-rust/>
- ³⁵ <https://akos.ma/blog/crystal-is-a-surprise/>

³⁶ <https://akos.ma/blog/dart-is-boring/>

³⁷ https://www.theregister.com/2021/11/30/aws_reinvent_rust/

³⁸ <https://9to5google.com/2022/07/19/carbon-programming-language-google-cpp/>

³⁹ <https://github.com/rust-lang/rustlings>

⁴⁰ <https://fasterthanli.me/articles/a-half-hour-to-learn-rust>

⁴¹ <https://rust-edu.org/>

⁴² <https://typst.app/>

⁴³ <https://discu.eu/weekly/rust/>

⁴⁴ <https://rustinblockchain.org/newsletters/>

⁴⁵ <https://gamedev.rs/>

⁴⁶ <https://this-week-in-rust.org/>

⁴⁷ <https://rusttimes.com/>

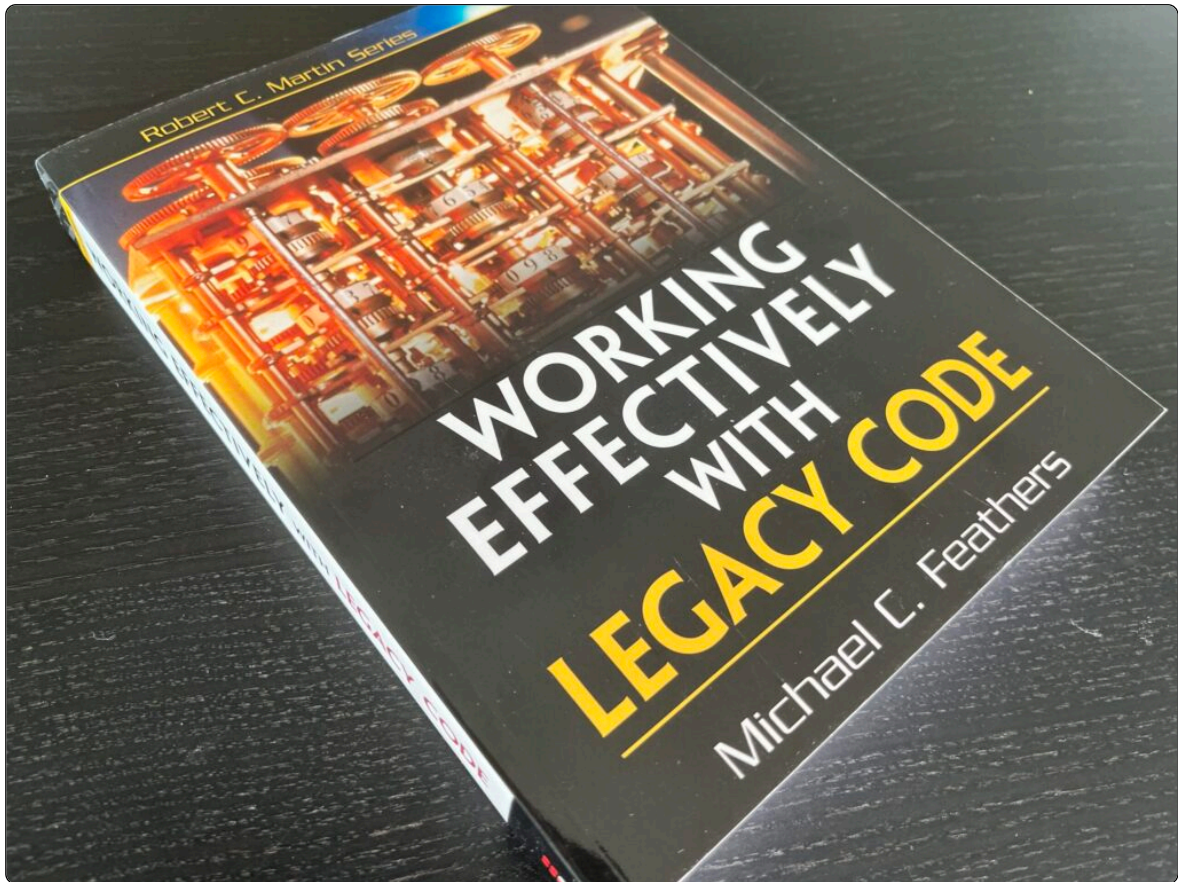
⁴⁸ <https://rustconf.com/>

⁴⁹ <https://linuxfoundation.org/webinars/writing-linux-kernel-modules-in-rust/>

⁵⁰ https://unsplash.com/@wizwow?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

⁵¹ https://unsplash.com/s/photos/state-of-rust?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Michael Feathers



By Graham Lee

Adrian previously discussed *Working Effectively with Legacy Code* when he talked about how to choose a programming language for your book¹. It deserves revisiting though, so here it is in the library section.

Quite simply, if you have not read this book yet, read it. If you have a colleague who has yet to read it, get them a copy. If someone asks you what one book to read about software engineering, it is this one. It is not *Code Complete, Second Edition*², nor is it *Clean Code*, nor any other book that claims to teach you how to get software right the first time around (you will not). It is not *Programming Rust*, nor *Programming*

Elixir, nor any other book that claims to teach you a technology that solves all of your problems (it will not).

All of your code will either be abandoned or become legacy code, so if you do not plan on failing you should learn how to work effectively with legacy code.

The central premise to the book is that legacy code is usually not enjoyed because it is insufficiently tested. And that the way to work with it is to identify few, hopefully not too invasive, changes that allow for parts of the software to be tested in isolation. Once those parts are under test, it becomes easier to make more, perhaps more significant, changes to the parts that are tested to subdivide into smaller testable units. After enough iterations of this you have a system whose behaviour is both described and stabilised by the tests and will be easier to work with.

Why bother with all of this? For the same reason we avoid rewrites whenever a new programming language comes along³: the existing software encapsulates both the current behaviour of the software system, and the *desired* behaviour: whatever the software is supposed to do, people have adapted to whatever it actually *does* and so that must be a starting point for any future work.

It is all too easy to say “oh the legacy code is really buggy, we should start from scratch” but those bugs are the way the system—not just the software, but the socio-technical system in which it is embedded—works. The ones that have been fixed are hard-fought lessons about what people want from this software. It may not be well-designed—but it may be. What gets called bad design might actually be good design from a few years ago: software design is fad-led, rather than engineering-led. But it may also be the case that a clean design is hiding under the weight of various patches and hot fixes. This is exactly the situation that *Working Effectively with Legacy Code* will let you take control of: fixing the software towards a clean design without having to let go of the good, valuable behaviour. And of course in this age of Agile®©™, we work to the principle that the primary measure of progress is working software. The legacy software already works.

So why do software engineers prefer to start from scratch? Partly it is because programming is a monetised hobby⁴: people enjoy writing software so they will find ways to do that for income. But it is also a matter of capability. A Masters-

level course in software engineering⁵ contains nothing about reading or adapting existing software; not even anything about buy-versus-build decisions. And most professional programmers are *not* trained software engineers. Without education or experience at reading, understanding, and modifying existing code, programmers see it as difficult, unnecessary effort. Why waste my time learning from person-centuries of experience at solving this problem, when I can type `cargo new` and have something that solves 5% of the problem badly in maybe a month or two?

And that is where this book comes in. It is your secret superpower. Learn how to work effectively with legacy code and you will be faster and more capable than almost all of the software writers working today, through this one weird trick: not writing most of the software you need.

Cover photo by Adrian Kosmaczewski.

REFERENCES

¹ <https://deprogrammaticaipsum.com/how-to-choose-a-programming-language-for-your-book/>

² <https://deprogrammaticaipsum.com/steve-mcconnell/>

³ <https://deprogrammaticaipsum.com/the-double-denim-of-software-engineering/>

⁴ <https://www.sicpers.info/podcast/episode-39-monetising-the-hobby/>

⁵ <https://www.cs.ox.ac.uk/softeng/courses/subjects.html>