Issue 010: Programming Paradigms

Graham Lee

July 1st, 2019

Welcome to the tenth issue of *De Programmatica Ipsum*, dedicated to the subject of *Programming Paradigms*. In this edition:

- Guido de Caso[1] shows us how our brains are shaped[2] by the programming paradigms we use.
- Graham[3] explains the the actual reason paradigms exist[4] in the first place.
- In this issue's subscriber-only article, Adrian[5] claims that in a Turing-complete language, the whole discussion about paradigms[6] is a distraction.

Enjoy this issue! Please let us know if you have any feedback[7] and get our free newsletter[8] to stay updated about new releases. If you want to support us, subscribe[9] for a month or a year, and let us know if you would like to write with us[10].

Cover photo by □□□ NG[11] on Unsplash[12].

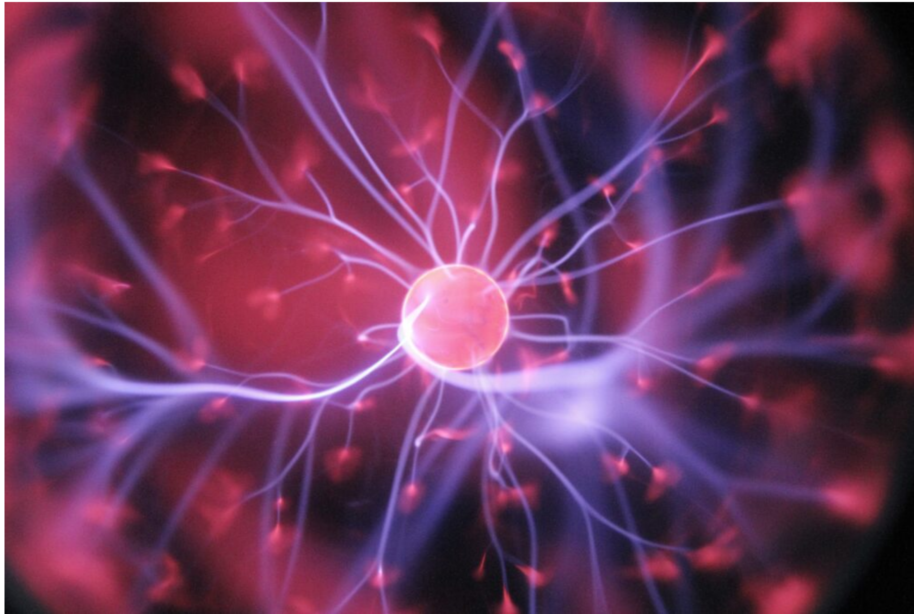
---

[1] https://deprogrammaticaipsum.com/user/gdecaso/
[2] https://deprogrammaticaipsum.com/on-the-influence-of-programming-language-paradigms/
[3] https://deprogrammaticaipsum.com/user/graham/
[4] https://deprogrammaticaipsum.com/in-which-thought-is-implied/
[5] https://deprogrammaticaipsum.com/user/akosmatr/
[6] https://deprogrammaticaipsum.com/alan-turing-wrote-object-oriented-code-in-c-and-ran-it-on-beam/
[7] https://deprogrammaticaipsum.com/feedback/
[8] https://deprogrammaticaipsum.com/newsletter/
[9] https://deprogrammaticaipsum.com/subscribe/
[10] https://deprogrammaticaipsum.com/write-with-us/
[11] https://unsplash.com/@danist07?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[12] https://unsplash.com/search/photos/building?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

On The Influence Of Programming Language Paradigms

Guido de Caso

July 1st, 2019



I have been recently been looped in rethinking the curriculum for my University's computer science degree. The current curriculum is strongly based on learning the fundamentals. A trait that allowed it to stay mostly relevant even 25 years after its introduction. Still, a quarter-century in our field is **a lot** of time, and revisiting the curriculum was due.

A very common question, when building or rebooting a curriculum in computer science, is *how* to introduce students into programming. What language should we start with? Should it be a simple bare-bones language or a feature-rich one? Should we use a real language or one that was designed to be easily taught and learnt? What paradigms should this first language favor?

In rethinking the curriculum, I was brought back to rethinking my own personal experience during those first University years, and the impact that curriculum design had on them as well as my professional career.

Leveling The Field

When I was pursuing my computer science degree fifteen years ago, my university introduced us to programming by learning Haskell. I know it may not seem the most intuitive option at first, but it turns out the experience was absolutely fantastic.

To begin with, and this may just be my personal opinion: Haskell is a lovely language.

Now, on to more objective territory: Haskell is a simple language. I typically equate the simplicity of something with the number of times that I find myself saying "except" or "but"

whenever I explain it. Haskell has very few rules. They may not be the most intuitive rules but they are indeed very few, and they are consistent: no exceptions. Haskell's syntax is elegant and clean.

Haskell is also a pure language. A "pure programming language" is one that is designed to enable development in one paradigm to its full extent. In the case of Haskell, it is a pure functional programming language. Typically languages fail to reach this level of purity. Indeed most mainstream programming languages are not pure, as they sacrifice this characteristic for other, more practical trade-offs. I am not arguing that a pure language is indeed better than an impure one as a general statement. But for the specific case of learning something, a pure language is indeed preferable.

Finally, Haskell was a fair language to our class. I mean fair in the sense of justice. Many of us had some experience with imperative programming before entering the university. However, most people did not. Had we started learning an imperative programming language like Pascal or even C this would have been unfair. Now, functional programming... that was something new for the entirety of the class.

I personally had had exposure to programming during high school: Logo, Pascal, C, Visual Basic and ASP. By the time I started college I was pretty sure that the first programming courses in University would be a breeze. Well... nothing quite far from the truth. I was soon faced with the stone cold fact that all my programming experience was in the realm of the imperative paradigm. I remember realizing how my apparent advantage dissolved into thin air right in front of me as soon as I started learning Haskell. New paradigm, new rules. All my experience was of zero use to me.

The Theory Of (Linguistic) Relativity

So let us take a closer look. I already had 3 or 4 years of programming experience and I learnt about 5 languages by that time. None of that counted. Why?

In the realm of natural languages there is a concept called *linguistic relativity*, also known as the Sapir–Whorf hypothesis. It states that the structure of a language affects its speakers' worldview.

A practical example of Sapir–Whorf in practice is how sexist vocabulary can influence career choices in a society. The terms "firemen" or "policemen" may not sound the most inviting to a woman as a career choice. While our language certainly affects our worldview, the inverse can also occur: our worldview can in turn affect our language. It is more common nowadays to use terms such as "firefighter" or "police officer" in an effort to remove gender biases.

Would this apply to programming languages too? In my case, it certainly did: my experience in imperative languages forged my worldview of what "programming" ment. When a language like Haskell was presented to me that was completely outside of my understanding. It seemed like it was not even programming... it looked more like math! My mind was super conflicted: "Was everything I learnt so far still valid? What is this new thing? What is going on?"

Functions Here, Functions There, Functions Everywhere

In time I came to love functional programming and I believe that it is one of the most influential trends in modern programming language design:

- JavaScript, one of the most pervasive languages out there, has had first-class support for functions since day one. Furthermore, JavaScript has since jumped out of the browser and evolved into a full-stack programming language that is used in native phone apps, server and even serverless applications. Asynchronous language idioms such as callbacks and promises became commonplace and are indeed built on top of functional programming concepts such as lambdas.
- Scala and Groovy are, without a doubt, two programming languages that revitalized the Java ecosystem. They were introduced in the early 2000s. Around that time Java was in version 5.0, generics where "the new thing" and Lambdas would not be around for another decade. Scala and Groovy could not be more different from each other; one is compiled the other is interpreted. One has an "academic" touch, the other one is pragmatic as it can be. However, both of these languages target the JVM and both of these incorporated functional programming elements that would then inform the Java language itself when these were incorporated in its version 8.
- Similarly, in the .NET community, C# incorporated the concept of lambdas well before Java. Furthermore, F# is an OCaml-inspired language which makes functional programming readily available to any .NET project.

So after learning Haskell and then using functional programming concepts extensively in other languages, my worldview of programming has certainly broadened. It used to be that imperative style was programming and that functional style was kind of like math. Now imperative programming looks kind of cumbersome to me and functional programming looks clean and stylish. Another example of linguistic relativity in practice.

Embracing Multiple Paradigms

I see paradigms as lenses that help us understand a problem under a different light. While I focused on functional programming here, there are other paradigms that are worth briefly discussing.

Logic programming is an interesting paradigm in which we state facts and rules about how a certain domain works and we ask whether certain other facts are derivable from those. It is a good lens to use whenever I need to think about a problem that involves multiple solutions.

Contrary to object-oriented programming, event-oriented programming focuses on state transitions instead of state itself. I find it to be a great tool in highly dynamic scenarios in which the mutation of state is the key aspect to model.

Finally, paradigms do not just apply to programming languages. As a devops practitioner I have seen a shift from bare metal to virtualization to containers to serverless. As an architect I have seen a shift from monolithic to microservices and back. As a leader in my organization I have seen a shift from waterfall to Scrum to Kanban to a mixture of those that made sense for us. While trends come and go, I find it key to keep an open mind and always try to apply the best tool for the job at hand.

Cover photo by Hal Gatewood[1] on Unsplash[2].

[1]https://unsplash.com/@halgatewood?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[2]https://unsplash.com/search/photos/psychology?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

In Which Thought Is Implied

Graham Lee

July 1st, 2019

Long-time payers of attention to my words will undoubtedly expect my article in the Programming Paradigms issue to be another rant about doing OOP correctly. That was the subject of my book, OOP the Easy Way[1]. I wrote about it in my M.Sc. dissertation[2]. In 2015, I gave a talk on the topic[3] saying that the reason OOP had failed was that it hadn't been tried.

> So was object-oriented programming the silver bullet? Is everybody an order of magnitude more productive because software engineering has saved us? Kindof hard to tell, because it's difficult to find an object-oriented program, even in 1995.

No, that's not a typo: I set the talk in 1995. So, time for another take-down of OOP as described, more complaints about how we've missed the point? Not so fast. I want to look at two sides of the same coin: the importance of a paradigm.

Everything Is Awesome!

Software engineers probably have as many problems now as we did back in 1995. Actually, one more problem: the price has dropped out of the software market (a predictable side effect

[1] https://leanpub.com/ooptheeasyway

[2] https://www.academia.edu/34882629/We_need_to_Small_talk_object-oriented_programming_with_graphical_code_browsers

[3] https://www.youtube.com/watch?v=_BbGxpiYFDg

of a lemon market[4]). On the NeXT, Lighthouse sold Diagram! for $499 (in 1995, so maybe $550 today). Today, OmniGraffle for iPad is $60, and definitely at the more expensive end of the market. A factor of ten reduction in cost is a good return for a factor of 1,000 increase in addressable market. Much software today is free and you can't make up for that with volume.

Money aside, writing software now is about as hard as writing software was in 1995. Whatever is going on, that's actually pretty impressive. It's impressive because our *expectations* for our software have increased astronomically in that time. The average computer user in 1995 had a beige box with a floppy drive, maybe a hard drive, and *exceptionally* a CD-ROM. It sat in their office, or maybe their home office, and was where *computering* got done. Many of these computers were not online and those that were used explicit online tools like Gopher, Netscape, or Email.

We expect modern software systems to remain up to date over a near-ubiquitous network link. They sync data between multiple devices and a network store in real time. Apps use multiple sensors and integrate data from many sources to improve the contextual relevance of their interfaces. And, in addition to their user-facing features, many apps perform surveillance, display advertising, engage in live testing of their users, and more.

The fact that we have increased our reach so much without drowning under the additional complexity truly is astounding.

Everything Is Awful!

Something must enable that growth in our aspirations and achievements. We find no single clear-cut cause: none of object-oriented programming, functional programming, reactive programming, or any other paradigms have clearly "won". What I mean by that programmers are not universally applying some programming paradigm in an identifiable way. You can see things called objects and classes all over the place without seeing a coherent philosophy behind how to make objects and classes or how to put them together. You can see things called functions (or worse, "lambdas"), and you'll see different things called functions in different places. They work in different ways.

Now a "paradigm" should be a conceptual framework, or pattern. When Object-Oriented Programming was still new, lots of programmers and architects expressed interest in a pattern language. The patterns community modeled itself after the community of practice in architecture formed around Christopher Alexander and his Pattern Language. In modern times many programmers know of the patterns language through a specific set of implementation patterns, published under the title *Design Patterns*. From either the book of that title or the newer *Head First Design Patterns*, developers learn about Singleton and Abstract Factory and resolve never to use either.

Which is a shame. If we want to share an idea of paradigm, we need to consistently describe it. The community comes together around shared practices by naming those practices and their motivations. By creating a pattern language. The language encapsulates a collection of related thoughts, and gives them a new name: a new, *higher-level thought* can be transmitted.

Everything Is Mediocre

To some extent, the software industry has indeed embraced patterns. We can identify some patterns in software business models, for example. "Ad-supported" means a lot more than

[4] https://deprogrammaticaipsum.com/the-various-meanings-of-quality/

supported by ads. “Software as a Service” usually implies Venture Capital backed software with a Free Tier and a Subscription Model yielding Recurring Revenue, storing data in Cloud Hostage. All of the capitalised phrases are themselves titles of patterns in the software business pattern language.

Maybe we have patterns and paradigms in all of the aspects of software construction except software construction. It’s interesting that while software-business phrases like “loot box” seem to have a stable meaning, software-engineering phrases do not. OOP, Agile, functional programming, free software, MVC, technical debt, refactoring, test-driven development. All of these ideas seem to go through a transition between description and practise[5] that obfuscates, dilutes or outright changes their meaning.

Everything Is Normally Distributed

Evidently we don’t *need* to all derive the same meaning from these phrases for the software industry to expand its reach and ambition. My question is whether we’ve scaled up just through bloody-minded stubbornness. Are we due some form of reckoning, where the bottom falls away from the market far enough that people notice just how ridiculous our salaries are and “right-size” their expenditure on software? Will such a crisis make us coalesce our ideas and distill our practices? Or are we collectively spending the right amount?

The answer, of course, is that this question makes no sense. There is no collective civilisation choosing to open its communal purse and spend an amount on software, which it then apportions to valuable and worthless exercises, efficiently or wastefully. Nor is there an invisible hand deftly guiding everyone to make the correct connections between available capabilities and required services. Instead, there are local people making local decisions with local information. Some of their choices are good and others bad. We avoid bad choices by doing what the paradigms were inviting us to do in the first place: to think.

Cover photo by Aaron Burden[6] on Unsplash[7].

[5] https://www.sicpers.info/2018/04/what-lenin-taught-me-about-software-movements/

[6] https://unsplash.com/@aaronburden?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[7] https://unsplash.com/search/photos/thought?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Alan Turing Wrote Object-Oriented Code In C And Ran It On BEAM

Adrian Kosmaczewski

July 1st, 2019

In his rare 1994 book "Object-Oriented Programming In C" Axel Tobias Schreiner[1] explains how to do inheritance, class methods, class hierarchies, and even how to raise exceptions using nothing else than pure, simple, pointer arithmetic-filled, ANSI C.

You do not believe me? You can read it online[2] for yourself, and I suggest you do. You can even find it in your preferred online bookstore if you look closely (Hint: ISBN 978-3446174269,) like Lulu[3], for example.

Reusability, Anyone?

The aforementioned book contains a terrible statement right at the very beginning:

> Only object-orientation permits code reuse between projects — although the idea of subroutines is as old as computers and good programmers always carried their toolkits and libraries with them.

(I can picture myself those developers in the first half of the 90's, carrying diskettes filled with useful code snippets from employer to employer, and watching new seasons of "The X Files" every evening at home.)

[1] https://www.cs.rit.edu/~ats/

[2] https://www.cs.rit.edu/~ats/books/ooc.pdf

[3] http://www.lulu.com/shop/axel-schreiner/object-oriented-programming-with-ansi-c/paperback/product-17561597.html

Let us give some slack to Herr Schreiner here: his book was written simultaneously to events he had no idea about. Roughly at the same time, somewhere in Silicon Valley, a team led by a certain James Gosling in a company named Sun was about to release the "Oak" project. Ultimately named Java, and decidedly marketed as an object-oriented environment, it borrowed some ideas from NeXT's Objective-C, like the `interface` keyword, and from C++, like `class`, and from a myriad other technologies.

In the following decades, however, the Java Virtual Machine has been shown to seamlessly support[4] functional, procedural, and even logic languages on top of it (yes, there are Prolog implementations for the JVM.)

History proved the quote above to be painfully wrong. Code reuse had more to do with component architectures, design patterns, levels of caffeine, decomposition in libraries or frameworks, and the actual and often underrated *wish to make things reusable,* than any particular programming paradigm.

And this happened because a Java Virtual Machine is a Turing-complete environment; as is the .NET Framework, or Erlang's BEAM[5], or Smalltalk-80, or those IBM z/VM[6] installations used to run those trusty mainframe COBOL programs written in 1965. They can run, by definition, anything that can be described by a Turing machine.

Just like the CPU in the computer you are using in this very moment to read these lines.

Reusability As A Paradigm Versus Paradigm Reusability

These days, we are using the offsprings of multiple programming paradigms having unprotected sex with one another in a thoughtful orgy. PHP, C#, Perl, C++ and even Visual Basic have all closures, lambdas or anonymous functions[7] now. F# and Scala can instantiate any class included in their corresponding vendor-provided frameworks. JavaScript implements functions as objects with a single method[8] `.call()`. Haskell comonads are actually objects[9]. Swift 1.0 implemented instance methods as curried functions[10].

But none of this is new. Smalltalk, arguably the precursor of object orientation, had `collect` and `select` methods which were the grandparents of our more common `map` and `filter` functional friends.

Right here, a lambda is an object. Over there, an object is a lambda. Further away, a method is a procedure. Another procedure is a function. And more often than not, a function is a closure.

In the Kingdom of Software, the classical taxonomy of programming languages into families, as promoted by Jean Sammet half a century ago, does not make sense today anymore. Much to the chagrin of right-wing programming language extremists, languages have bred with one another and have given birth to mutants and bastard children, some of which have lost all *purity* in exchange for a much needed *applicability*. Some of which have survived, some of which have faded away.

---

[4] https://en.wikipedia.org/wiki/List_of_JVM_languages
[5] https://en.wikipedia.org/wiki/BEAM_(Erlang_virtual_machine)
[6] https://en.wikipedia.org/wiki/VM_(operating_system)
[7] https://en.wikipedia.org/wiki/Anonymous_function
[8] https://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this/
[9] http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html
[10] https://oleb.net/blog/2014/07/swift-instance-methods-curried-functions/

Convenience And Egos

While it can be certainly a fun exercise to implement an Object-Relational Mapper or yet another React web framework in ANSI C, it begs the obvious question, "why?"

Of course we will not implement such a beast. Why would we? We can choose among thousands, even tens of thousands of languages today, most of them open source, and all bundled with libraries that make them suitable for a particular endeavour. They are all free, most of them hosted in Github, many of them actively maintained, and curated by long lists of luminaires in their `AUTHORS` files. These teams are continuously deciding about features to include or not. Most new popular languages created in the past decade (Go, Rust, Kotlin, Swift, to name a few) all support functional, procedural and object-oriented programming off-the-box.

Does it make sense to continue to talk about programming paradigms? It seems to the author of these lines, in any case, that the discussion has shifted to more actionable items, such as the strength of the type system, or the number of supported platforms. Maybe that is the more rational question to ask, the more open-minded discussion to have.

Or maybe, in the age of microservices and serverless, is it simply more important to be able to stick your preferred paradigm inside a container, inside a node, inside a Kubernetes deployment, inside a cloud somewhere else? Maybe, if your language can speak HTTP, can we follow Graham's advice[11] and reconsider microservices not as a new paradigm, but rather as yet another implementation of Object-Oriented programming, one finally done right?

There are too many programming paradigms[12] beyond the ones mentioned in this article. All of them, without exception, can be used to solve your problem as long as you use a Turing-complete language. The only valid statement that can help you choose one of them is the following: *cogito, ergo sum*.

Cover photo by Nick Fewings[13] on Unsplash[14].

[11] https://leanpub.com/ooptheeasyway
[12] https://en.wikipedia.org/wiki/Programming_paradigm
[13] https://unsplash.com/@jannerboy62?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[14] https://unsplash.com/search/photos/brain?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText