Linnéuniversitetet
Kalmar Växjö

Analysis of Programming Styles

How Object-Oriented Design Compares to Functional Programming

Report

Donatas Mockus

December 14, 2021

Semester: Autumn 2021
Discipline: Software Technology
Course code: 1DV510
Student ID: dm222qi@student.lnu.se

Abstract

Beyond the syntactic and pragmatic differences, the evolution of programming languages has brought along a cultural element to how code is expressed. Remarkably in the last decade, a shift in ideology has sparked a movement to find the perfect paradigm. The dominant explanation for this phenomenon is the newfound popularity of modern functional languages, the stagnant developments and archaic dogmas of pre-existing paradigms. The goal of this paper is to evaluate two of the most popular programming styles in use today – object-oriented and functional design. Besides the differences in implementation, a benchmark of a practical application is carried out and backed up by existing research to determine which of the two is better.

Keywords

Programming, Object-Oriented Programming, Functional Programming

Contents

1 Introduction **4**
1.1 Aim and Research Questions 4
1.2 Biases 4

2 Programming Paradigms **5**
2.1 Object-Oriented Programming 5
2.1.1 Criticism 6
2.1.2 Archetypical OOP 6
2.2 Functional Programming 7

3 Performance Analysis **8**

4 Results **9**

5 Discussion **9**

6 Conclusion **10**

Acronyms **11**

1 Introduction

A recent shift in paradigm in the software industry gave a fresh breath of life in functional programming (FP) language popularity. While still the dominant style, object-oriented programming (OOP) is increasingly garnering criticism for its strictly enforced semantics and over-complicated design, however, many of the presented arguments and examples are simply bad practices, poor implementations or in some cases not even OOP in the first place. Of course, each programming style has its advantages and drawbacks, fitting certain tasks and failing at others. It is therefore necessary to objectively evaluate the performance and readability of both object-oriented and functional programming.

1.1 Aim and Research Questions

The goal of this paper is to evaluate the performance and readability of OOP and functional programming styles. Furthermore, a meta-analysis of existing studies is taken into consideration to compare each style. The following research questions will help achieve this aim:

1. Why is object-oriented programming the dominant paradigm?
2. Which of the two programming styles is better?

1.2 Biases

To mitigate the effects of bias towards any one programming style, due care was taken when writing both practical and meta-analytical parts of this report. Code written and conclusions drawn in this paper are based on widely accepted opinions and facts, sets of ideals or empirical data, rather than any one person's individual experience.

2 Programming Paradigms

Paradigms in programming are simply styles; ways of structuring code by following a set of ideals with the intent to improve readability, runtime efficiency, development time or any combination of the three [1]. In its infancy, programming took an imperative approach, where the control flow of any given program was entirely explicit, instructions being executed one after another and affecting the global state of the computation [1]. While seemingly archaic, imperative programming is, fittingly, imperative to the way low-level assembly type languages, where such style is largely isolated to, work.

Following its footsteps, structured programming extends imperative programming by introducing lexical scopes – a block of code whereat local variables and functions reside. Straying away from affecting the global state in favour of smaller local states, structured programming also takes advantage of nested loops and subroutines for its control flow and avoiding "go to" statements [1]. This discipline is the lowest common denominator of all other higher-order programming styles that succeeded it, since OOP, FP, and all of their derivatives have some degree of intrinsic structure. Many early high-level languages embraced this type of style including Pascal, C, and Modula [1].

2.1 Object-Oriented Programming

Structured programming gave birth to the idea of encapsulating data and its behaviour into objects. The basis of OOP is passing messages to or between objects; the latter responding to the messages by performing operations, generally known as methods [1][2]. This approach has several main advantages:

- Highly structured design allow for a tidy compartmentalisation of data types. This is referred to as encapsulation, a concept where data and methods that modify it are bound together and abstracted away from external interference or misuse. A well structured code base enables simpler troubleshooting, as most logical errors can be narrowed down to a single object.
- Language extensibility enable user defined containers and data types.
- Inheritance is technique of generalising objects. It allows for hierarchical relationships between classes, sharing common behaviour and, by extension, re-use of the code. Inheritance is the foundation for polymorphism, the ability for an object's type to be inferred at runtime and assume its behaviour [2].

Object-oriented design had an advantage of being one of the first higher-order programming paradigms embraced by the early languages. Along with its intuitive control flow and powerful techniques, OOP became the dominant style still in use today [1].

2.1.1 Criticism

In spite of the presence in many contemporary programming languages, the very same advantages, that make OOP so powerful, are also its downfalls. Many of the critics of object-oriented design argue that the highly structured nature of OOP result in messy code [3]. In some respect they are right, while OOP does promotes structured programming, in larger code bases objects can become too broad and inheritance hierarchy impossible to keep track of. Meanwhile small programs seldom see the benefit of representing its data as objects.

Encapsulation can also be viewed as a disadvantage, because it abstracts away the inner workings of a method. In other words, there is no way to determine the purpose of a function by only seeing its return type, input parameters, and the name. Furthermore, polymorphism can indeed introduce uncertainty, although, somewhat ironically, functional programming attempts to emulate polymorphic behaviour too by allowing generic functions.

In summary, it appears that most of the criticism directed towards object-oriented programming actually stem from misuse and poor implementations. Just like any paradigm, OOP has its uses, and when applied correctly can be an incredibly powerful tool. An object should not support more than its most essential methods and data; misuses of encapsulation defeat the purpose of objects entirely, as well as lose the benefit of security; poor inheritance hierarchy inhibits effective use of polymorphism. Given this basis, a set of foundational statements can be established in order to define how object-oriented code can be written well.

2.1.2 Archetypical OOP

One of the biggest problems facing any programming style are the loose definitions of their sets of ideals. In other words, it can often be difficult to discern which paradigm a piece of code is following. Luckily, object-oriented programming was concretely defined by its pioneer Dr. Alan Kay in his 2003 email to Stefan Ram [4]. While initially designing the architecture in 1967, Dr. Kay broadly defined OOP with the following values: 1. Objects may only be able to communicate with messages; 2. Devoid of data; 3. Polymorphic. Later, he revised his definition as simply "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things." [4]

It is clear that messaging and encapsulation are intrinsic properties of OOP. Incidentally, good encapsulation results in effective messaging and vice versa. As discussed previously, encapsulation hides data by abstracting it away from external influence, thus encapsulation promotes decoupling [2].

In programming, coupling is the level of independence between objects. Fewer external dependencies class members have typically correlate with lower coupling [5]. In contrast, cohesion is the measure of how related class members are [5]. An object with member variables relating to similar data have high cohesion. In combination with low coupling, high cohesion is a sign of a well structured object, high readability, and maintainability.

Finally, drawing inspiration from Dr. Kay's definition, a set of OOP ideals can be established as follows:

1. An object may only contain data and methods that are intrinsically related. This raises cohesion.
2. Data should be abstracted away; only accessible and modifiable via getters and setters respectively. This lowers coupling.
3. Abstract methods should only be contained in the highest order class that is inherited by other classes using said method. This improves the readability of polymorphic objects.

2.2 Functional Programming

Functional programming preceded pure object-oriented design by fourteen years, taking root in the second oldest language still in use today – LISP [6]. While OOP was prefigured in the early 1960s by Modula, the modern principles, as discussed earlier in this report, were not established until the creation of Smalltalk by Dr. Alan Kay [2]. Furthermore, LISP sought inspiration from Modula, and then itself was the driving force behind Smalltalk [1]. This paints a clear picture that the earliest known high-level programming languages, and by extension their paradigms, were heavily inspired by each other.

That being said, functional programming takes a different approach to that of other styles. By discarding states and focusing solely on the control flow of a program, FP reaps the benefits of immutable data, minimal side effects, reusable and high order functions [1]. Such approach has the following advantages:

- Deterministic behaviour. Functional programming makes use of what are called pure functions which always produce the same output for a given input and do not depend on external states. This has the added benefit of making unit testing easier [6].
- Control flow is more intuitive. Since a given function is only concerned about its inputs and outputs, rather than the state of the wider program, FP avoids structural pitfalls seen in its object-oriented counterpart.
- Pure functions are inherently thread safe as they do not affect shared state of the program. This effectively eliminates race conditions, allowing for highly parallel applications.

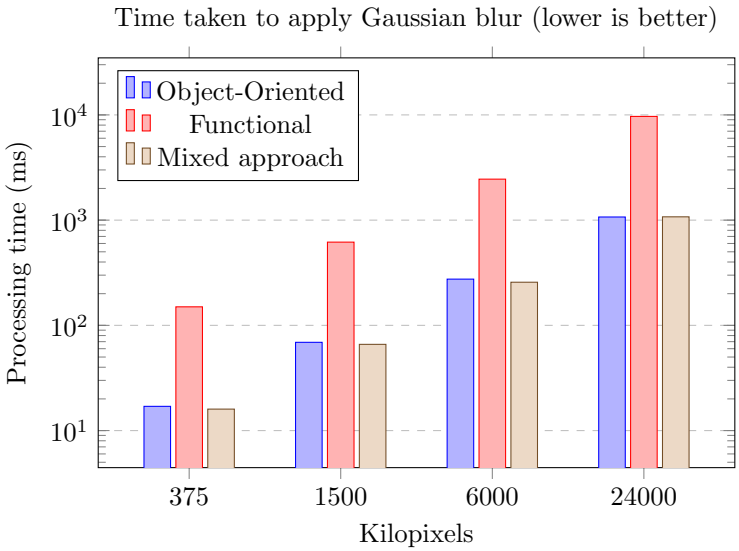
Following a familiar pattern, functional programming principles can also be viewed negatively. While criticism towards it is much more subdued, FP introduces some rather strange design choices such as favouring recursive functions over loops. In fact, while FP does indeed promote concise code it often comes at a cost of readability. Recursive functions are hard to read, high-order functions overcomplicate control flows, and worst of all, immutable data enforces the use of copy semantics, thus considerably inflating the memory usage of a program.

3 Performance Analysis

Choosing one style over another is often a matter of preference. There is not doubt disagreements about which paradigm should be followed will continue in spite of any arguments put forth by either side. Thankfully, empirical data can be collected by performing static analysis and measuring runtime performance. To achieve this, two programs, with identical functionality, were written following both functional and object-oriented guidelines. The application takes an image and applies a Gaussian blur filter.

The object-oriented approach uses a class to represent images as an array of pixels. The class methods are allowed to modify the object's state, uses ranged and primitive loops. On the other hand, the functional approach does not use any objects, all variables are constant, loops are replaced by recursive functions, and the use of C++ algorithm library is encouraged.

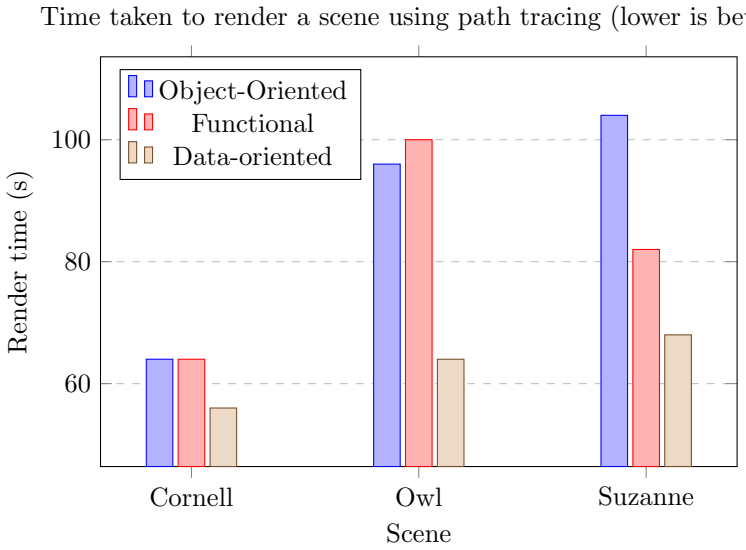
To test the program, an identical jpeg image was used at four different resolutions: original 4000×6000 and three more, each half the scale of the previous. Finally, time taken to execute both approaches was averaged over 100 runs.



Interpreting this data at face value, a clear, albeit premature, conclusion can be drawn that object-oriented approach is significantly faster. In fact, it is almost 10 times less efficient, which begs the question where functional programming falls short. In this particular case, it appears that the recursive nature of the functions is orders of magnitude slower than even naïvely implemented nested loops.

Taking a deeper look, existing research somewhat confirms the aforementioned results. In 2019, Matt Godbolt, the creator of compiler explorer, performed an identical analysis under slightly different conditions. Also using C++, he implemented a path tracer in three most popular programming paradigms: object-oriented, functional, and data-oriented design (DOD)

[7]. Godbolt's results show a similar trend: FP performing about the same, if not marginally worse, as OOP, however, both are outperformed by data-oriented design [7].



The one true conclusion that can be drawn, is that dogmatic approach to any programming style is detrimental to performance. Few languages are mono-paradigmatic, majority facilitating one or more approaches to suit programmers' and application needs, so it is naïve to blindly follow one set of ideals. Taking another look at the Gaussian blur example, it is clear that functional programming can indeed be faster than pure OOP or pure FP by simply embracing loops instead of recursion.

4 Results

The findings of this study, as well as existing research, do not indicate a clear best choice between the two programming paradigms. Both functional and object-oriented approaches have readability shortcomings and, in places, overcomplicated designs, assuming idealistic principles are followed. Nonetheless, they both offer similar performance, albeit functional programming suffers from higher memory overhead.

5 Discussion

Largely as a result of early languages taking inspiration from one another, object-oriented design became the defacto paradigm of choice. In spite of this, functional programming has seen somewhat of a resurgence in the recent years, with many industry professionals disavowing OOP. The movement has also inspired developers to reconsider their approach to coding and choose entirely different paradigms, distinct to that of functional or object-oriented. As seen previously,

data-oriented design outperforms both OOP and FP, despite it being least popular of the three. This also begs the question of what paradigms will be invented in the future and when OOP's reign will end.

In the meantime, it appears that neither object-oriented nor functional programming performs best in isolation. Besides pontifical beliefs, there seem to be no good arguments against mixing different styles, as long as the end result is well designed and performant code.

6 Conclusion

In summary, object-oriented and function programming styles offer a different approach to problem solving in computing. The former grouping data and its behaviour together; the latter focusing on the control flow of an application and immutability of data. The choice between these two styles should be left up to the individual programmer, however, more research is required to determine which is more performant.

Acronyms

DOD data-oriented design 8

FP functional programming 4, 5, 7, 9, 10

OOP object-oriented programming 4–7, 9, 10

References

[1] R. Toal. "Programming paradigms." (2008), [Online]. Available: `https://cs.lmu.edu/~ray/notes/paradigms/`. (accessed: December 14, 2021).

[2] D. J. Armstrong, "The quarks of object-oriented development," *Communications of the ACM*, vol. 49, no. 2, 2006.

[3] R. Fabian. "Data-oriented design." (2013), [Online]. Available: `https://www.dataorienteddesign.com/dodmain.pdf`. (accessed: December 14, 2021).

[4] S. Ram. "Dr. Alan Kay on the Meaning of "Object-Oriented Programming." (2004), [Online]. Available: `https://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en`. (accessed: December 14, 2021).

[5] A. J. Offutt, M. J. Harrold, and P. Kolte, "A software metric system for module coupling," *Journal of Systems and Software*, vol. 20, no. 3, 1993.

[6] D. A. Turner. "Some history of functional programming languages." (2012), [Online]. Available: `https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf`. (accessed: December 14, 2021).

[7] M. Godbolt. "Path tracng three ways." (2019), [Online]. Available: `https://github.com/mattgodbolt/pt-three-ways`. (accessed: December 14, 2021).