Reassessing Object-Oriented Programming: Structural Limitations and Implications for Scientific Computing

Dr. Ghazi CHERIF MD. MS.

ABSTRACT

Object-Oriented Programming (OOP) has been a dominant paradigm in software engineering for several decades and is widely used in languages such as C++, Java, and Python. OOP was promoted as a method for managing complexity in large software systems through encapsulation, inheritance, and polymorphism. However, significant criticism has emerged from researchers and practitioners who argue that object-oriented design often increases complexity rather than reducing it. Some critics have described the paradigm as a "trillion-dollar disaster" due to the economic cost associated with maintaining large object-oriented systems. This paper reviews major criticisms of object-oriented programming and examines their implications for scientific computing, particularly in biostatistics. We discuss problems related to excessive abstraction, shared mutable state, rigid inheritance hierarchies, and the mismatch between object-oriented modeling and data-oriented scientific workflows. We argue that although OOP provides certain organizational benefits, its structural characteristics may hinder reproducibility, transparency, and maintainability in scientific research software.

1. Introduction

Software has become a central component of modern scientific research. Disciplines such as biostatistics, epidemiology, and bioinformatics rely heavily on computational tools for data analysis, simulation, and statistical modeling. As datasets continue to grow in size and complexity, the reliability of scientific conclusions increasingly depends on the quality of the software used to analyze those datasets.

Object-Oriented Programming (OOP) emerged as one of the most influential programming paradigms in the late twentieth century. Languages such as C++, Java, and later Python adopted object-oriented constructs as a standard way to organize code. The central goal of OOP was to address the complexity of large software systems by structuring programs as collections of interacting objects.

Advocates of OOP argue that encapsulating data and behavior into objects improves modularity and allows developers to reuse code more effectively. Inheritance and polymorphism were designed to allow programs to evolve and extend functionality without rewriting existing implementations.

Despite these intended advantages, OOP has faced increasing criticism over the past two decades. A number of developers, researchers, and computer scientists have argued that the paradigm often leads to overly complex and difficult to maintain systems. In one well-known critique, object-oriented programming was described as a "trillion-dollar disaster," reflecting the cost associated

with maintaining enterprise systems built on object-oriented frameworks.

In the context of scientific computing, these concerns are particularly important. Scientific software must be transparent, reproducible, and easy to validate. If programming paradigms obscure the underlying computational logic, they may hinder the ability of researchers to reproduce and verify results.

This paper examines several major criticisms of object-oriented programming and evaluates their relevance for scientific computing in biostatistics and related fields.

2. Historical Background

The conceptual origins of object-oriented programming can be traced to the Simula programming language developed during the 1960s. Simula introduced the idea of modeling systems as collections of interacting objects that represent entities in the real world. This concept later influenced the design of languages such as Smalltalk and C++.

During the 1980s and 1990s, object-oriented programming gained widespread adoption. C++ introduced object-oriented features to the widely used C language, while Java was designed from the beginning around object-oriented principles. These languages became standard tools in enterprise software development and were widely taught in computer science education.

The core concepts of OOP include encapsulation, inheritance, and polymorphism. Encapsulation refers to the practice of combining data and the functions that operate on that data into a single unit called an object. Inheritance allows classes to derive behavior from parent classes, promoting code reuse. Polymorphism enables different objects to respond to the same interface in different ways.

These mechanisms were intended to make software systems easier to extend and maintain. However, critics argue that the same mechanisms often introduce additional layers of complexity.

3. Excessive Abstraction

One of the most common criticisms of object-oriented programming is that it encourages excessive abstraction. OOP frameworks frequently rely on complex class hierarchies, interfaces, and design patterns that can obscure the underlying logic of a program.

Developers working within large object-oriented frameworks may need to understand dozens or even hundreds of classes before they can determine how a particular function is implemented. Instead of simplifying software design, these abstraction layers can increase the cognitive burden placed on programmers.

Critics argue that programmers sometimes spend more time designing abstract class hierarchies than solving the actual computational problems at hand. Design patterns such as factories, visitors, and decorators can introduce substantial amounts of boilerplate code that provide little direct functional value.

This problem is particularly relevant for scientific computing. Researchers who attempt to reproduce a computational experiment must often read and understand the source code used in the

original analysis. If that code is embedded within a highly abstract object-oriented framework, understanding the implementation of a statistical method may require navigating large portions of the codebase.

In such situations, abstraction becomes an obstacle rather than a benefit.

4. Shared Mutable State

Another frequently cited criticism of object-oriented programming concerns the use of shared mutable state. Objects in object-oriented systems often maintain internal variables that change during program execution.

Although encapsulation hides the internal state of objects from other parts of the program, it does not eliminate the complexity associated with state changes. When multiple objects interact and modify shared data, reasoning about the behavior of the program becomes difficult.

The issue becomes particularly severe in concurrent systems. Many modern scientific applications rely on parallel processing to analyze large datasets or perform computational simulations. When multiple threads access and modify shared objects, developers must introduce synchronization mechanisms such as locks and mutexes.

These mechanisms can introduce additional problems, including race conditions and deadlocks. Debugging such issues is notoriously difficult and can require extensive testing and analysis.

For this reason, some critics argue that programming paradigms that emphasize immutable data structures may be better suited for parallel computation. Functional programming approaches, for example, often avoid mutable state altogether, which can simplify reasoning about concurrent programs.

5. Fragility of Inheritance Hierarchies

Inheritance is often promoted as a major advantage of object-oriented programming. By allowing classes to inherit behavior from parent classes, developers can reuse code and extend functionality.

However, inheritance can also introduce tight coupling between different components of a system. When a base class is modified, all subclasses that depend on it may be affected. This phenomenon is sometimes referred to as the “fragile base class” problem.

Deep inheritance hierarchies can make software systems difficult to maintain. Understanding the behavior of a single class may require examining multiple levels of parent classes. Small changes to a shared component can propagate through the system and cause unexpected failures.

In scientific software, where algorithms are often modified as research progresses, this rigidity can slow development. Researchers may find themselves constrained by the structure of existing class hierarchies rather than focusing on the implementation of new models or analyses.

6. Mismatch With Data-Oriented Scientific Workflows

Another criticism of object-oriented programming is that it models programs as collections of objects rather than emphasizing transformations applied to data.

Many scientific computations are inherently data-oriented. Statistical analysis typically involves a sequence of transformations applied to datasets. These transformations may include data cleaning, model fitting, parameter estimation, and visualization.

Such workflows are often naturally expressed as pipelines of operations. Functional or procedural programming paradigms can represent these pipelines in a direct and transparent manner.

In contrast, object-oriented design may require developers to introduce additional classes and abstractions that do not correspond directly to the underlying statistical operations. As a result, the conceptual structure of the program may diverge from the structure of the scientific analysis.

Languages commonly used in statistical computing, such as R, often rely heavily on functional programming concepts for precisely this reason.

7. Cultural and Educational Influences

The widespread adoption of object-oriented programming was influenced not only by technical considerations but also by educational and cultural factors. During the 1990s and early 2000s, many computer science curricula emphasized object-oriented design as the primary method for structuring programs.

As a result, generations of developers were trained to approach programming problems using object-oriented techniques. In some cases, this led to the application of OOP even in contexts where alternative paradigms might have been more appropriate.

Prominent figures in the computing community have expressed skepticism about the paradigm. Critics argue that an excessive focus on class hierarchies and design patterns can distract programmers from more fundamental concerns such as algorithm design and data structures.

8. Implications for Biostatistics

The criticisms of object-oriented programming have particular relevance for scientific disciplines such as biostatistics.

Scientific software must satisfy several critical requirements. It must be transparent so that other researchers can understand the implementation of statistical methods. It must be reproducible so that results can be verified independently. It must also be flexible enough to adapt to new datasets and research questions.

Highly abstract object-oriented frameworks may conflict with these goals. Deep class hierarchies can obscure the implementation of algorithms, making it difficult for researchers to verify correctness. Hidden mutable state within objects may introduce subtle errors that are difficult to detect.

Furthermore, many statistical workflows are inherently data-driven rather than object-driven. Expressing such workflows using object-oriented designs may introduce unnecessary complexity.

For these reasons, many scientific programming environments emphasize simpler design principles. Data analysis languages frequently rely on functional constructs and explicit data transformations rather than complex object hierarchies.

9. Balanced Perspective

Despite these criticisms, it would be inaccurate to conclude that object-oriented programming is universally harmful. OOP has enabled the development of many widely used software systems and provides useful tools for organizing large codebases.

Encapsulation can help developers manage complexity, and object-oriented interfaces can provide clear boundaries between different components of a system.

However, the key issue may be the uncritical application of object-oriented design. Modern programming languages increasingly support multiple paradigms, allowing developers to combine object-oriented, functional, and procedural approaches.

Such flexibility allows programmers to select the paradigm that best matches the structure of the problem being solved.

10. Conclusion

Object-oriented programming has played a major role in the evolution of software engineering. Its principles of encapsulation, inheritance, and polymorphism were intended to address the challenges of building and maintaining large software systems.

However, extensive criticism from both practitioners and researchers suggests that object-oriented programming can introduce significant structural problems. Excessive abstraction, mutable state, fragile inheritance hierarchies, and mismatches with data-oriented workflows may increase complexity rather than reduce it.

In scientific fields such as biostatistics, where transparency and reproducibility are essential, these issues deserve careful consideration. Researchers developing computational tools should evaluate whether object-oriented design genuinely improves clarity and maintainability or whether simpler paradigms might better support scientific goals.

Ultimately, effective scientific software depends not on adherence to a single programming paradigm but on thoughtful design choices that prioritize clarity, correctness, and reproducibility.

11. References

[1] Object-Oriented Programming: The Trillion Dollar Disaster.

`https://news.radio-t.com/post/object-oriented-programming-the-trillion-dollar-disaster`

[2] Strongest Criticisms of Object-Oriented Languages. Computer Science Stack Exchange.
https://cs.stackexchange.com/questions/171056/strongest-criticisms-of-object-oriented-languages

[3] Atwood, J. Linus Torvalds: Visual Basic Fan. Coding Horror.
https://blog.codinghorror.com/linus-torvalds-visual-basic-fan/

[4] Musial, K. Why OOP Is Bad and Possibly Disastrous.
https://news.ycombinator.com/item?id=6782539

[5] Hacker News discussion on object-oriented programming critiques.
https://news.ycombinator.com/item?id=6782539

[6] Thrawn01. Object-Oriented Programming Is Bad.
https://thrawn01.org/concepts/object-oriented-programming-is-bad

[7] Stallman, R. Essays on computing and programming philosophy.
https://stallman.org/stallman-computing.html