Evaluation Oriented Classes

by Sven Nilsen, 2025

In this paper, I introduce a framework for Object Oriented Programming (OOP) that is minimalistic in design and closer to functional programming in semantics than traditional OOP. This new framework fits objects models similar to those found in Dyon or Javascript, but is generally agnostic toward implementation. Two use cases of this framework are data modeling and LLM prompt engineering where LLM assists in writing code.

Structuring data efficiently for programming is a hard problem, since there are many soft constraints that span a variety of language biases. For example, how data is structured affects how people read and write code. People want to do their job without large delays and without mental or information overload, something that is difficult when the demands of software are increasing.

One of the programming pattern invented to overcome this problem, is Object Oriented Programming (OOP). While classes and inheritance dates back to Aristotle's philosophy (4th century BCE), the first record of anyone using this pattern in programming is from Simula Research (Norway 1961 – 1967), followed by Alan Kay's breakthroughs in system thinking onward from 1966, resulting in the programming language Smalltalk and mainstream OOP.

In this paper I will use a minimalistic version of OOP that constrains classes to optional single inheritance. There are no methods, only properties. All properties are public, meaning the data structure is transparent. Properties are read/write by default. Properties can be declared to be read-only in subclasses. There are no write-only properties. This means that reading from a property always succeeds, but writing to a property might not succeed:

<class> : some(<super class>)	Single inheritance
<class> : none()	Inheritance is optional
get(<object>, <property>) → <value>	Reading property always succeeds
set(<object>, <property>, <value>) → bool	Writing to a property might not succeed

In Evaluation Oriented Classes (EOC), there is an optional evaluation property per class:

eval <class> some(<property>)	Specifying an evaluation property
eval <class> none()	There is no evaluation property

EOC solves the design problem of data primitives versus classes. For example, a class `F64` can have a property `val : f64` where `f64` is a primitive (float precision of 64 bits). By the following:

```
eval F64 some(val)
```

The primitive `f64` is now integrated with the class `F64` through the `val` property.

In any context where only the `val` property of `F64` is used, it is possible to inline a `f64` constant. This means a programmer does not need to declare an instance of the class `F64` everywhere. This brings the semantics closer to functional programming where primitive data types are preferred.

The design of Evaluation Oriented Classes (EOC) is based on the theory of Avatar Extensions. In Avatar Extensions, there is a 0-avatar, which is like the primitive data type that gets extended with classes. A 1-avatar is like a class that wraps around a single primitive data type. Higher n-avatars are like classes with multiple properties.

The following example illustrates how close EOC is to functional programming (pseudo-code):

```
scalar { val: f64, eval val }
point { x: f64, y: f64, z: f64 }
line { from: point, to: point }
distance: scalar { ab: line, readonly val, eval val }
direction: point { ab: line, readonly x, readonly y, readonly z }
```

Here, a scalar itself is interchangeable with `f64`. The data of a scalar is completely transparent. However, a distance inherits from scalar and makes the `val` property read-only. This means, when trying to write to a scalar that actually is a distance, the process will not succeed.

Similarly, a direction inherits from point and makes the component `x, y, z` read-only. This means, when trying to write to a point that actually is a direction, the process will not succeed.

In functional programming, distance and direction of a line would be implemented as functions:

```
distance : line → f64
direction : line → point
```

The output of a function is inherently read-only in relation to the input. This has many benefits, among other things it is easier to understand large code bases. So, in functional programming, it is common to use immutability whenever possible. In Rust, data is immutable by default.

Similar to the types of functions in functional programming, EOC does not explain how the code is executed. Still, it offers data transparency, which makes reasoning about the use of the data structure easier. A property can be expressively linked to another object, such that changes over time are automatically propagated through the data structure. However, it can also be inlined to decouple dependencies, e.g. providing a snapshot of some data frozen in a moment of time.

One use case for EOC is as a tool for data modeling, as part of a process of planning or documentation, without requiring implementation to match the data model exactly. In this case, the power of EOC is that it naturally leans toward multiple uses of data, that can be complex to describe in detail. EOC can be used as a background material or overview used to understand a codebase.

Another use case for EOC is as a tool for LLM prompt engineering where LLM assists in writing code. LLMs can be very good solving specific problems when they are given good background material. The material does not need to match the code exactly, but is used to bias the LLM toward proper solutions. EOC is usually not sufficient on own to bias the LLM enough, but it can be part of a solution, in particular when LLM is writing functional or procedural code.

EOC is better suited to data models where transparency is common. For example, in Dyon and Javascript, objects are kind of like `HashMap` in Rust, where a property is a string used to look up a value. Classes of objects do not need to have a super-class, but can also function like informal duck-typing and by using enforced business logic while programming. It also fits with the paradigm of Path Semantics, where analysis and synthesis often is done externally instead of being integrated with the programming language. While planning is useful, it should not get in the way for coding.