Thinking Objects

I have recently gained some insights into object-oriented programming. In fact, one should not attach too much meaning to objects, or be constrained by the metaphors of "cats, dogs, animals, Aristotelian classification". A procedural perspective can help to understand that creating an object is equivalent to defining a bunch of variables, and the methods of the object are all the possible operations on these variables in the subsequent process. When reading code, instead of thinking about how to transform the parameters passed into a function, or what to calculate through them, one should pay attention to the fact that the methods of objects can change the state of objects, that is, the contents of objects. This is not to be ignored, and even the norm. This also leads to the difficulty of inferring what contents of objects are involved in the methods from the signatures of the methods, unlike pure functions that are locally understandable atoms.

This programming technique, without discussing the philosophy or the plain usage, is a kind of decomposition of long processes. Sometimes it is very useful, such as in the resource management and control parts of the process (opening/closing files, network/database connections, applying locks, etc.), sometimes it may not be so useful (such as the visitor pattern), and sometimes it may make the program harder to understand because of the fragmentation.

Of course, the features of encapsulation and access control, inheritance and runtime polymorphism, serialization and reflection, etc., are beyond my understanding here. We can think in reverse, in procedural programming, especially when there are many local variables, or when local variables involve resources and control, whether some variables with strong correlation are grouped together to model something or be responsible for some function.

If the object is complex and special (strongly related to the current program, without generality), or simply not designed but extracted from the operation process, or this group of variables (the contents of the object) represents a limited amount of resources, then it is likely that it has no mathematical properties (there are only three database connections in total, what to transform), and using functional thinking seems to have no gain. It is also difficult to do static analysis in this case.

The most powerful time of functional programming is when a large space is spanned by a finite number of orthogonal bases, and the algorithms we want can be easily implemented on this space. Object-oriented programming often talks about "encapsulation", but encapsulation should have no inevitable connection with inheritance and polymorphism. We think carefully, under what circumstances do we need encapsulation? In mathematics, does matrix need encapsulation? Does point need encapsulation? Does describing a triangle as a part enclosed by three straight lines need encapsulation? Without considering polymorphism, just talking about encapsulation, the parts that we do not want users to see are often related to the structure of the computer itself (the interface, protocol, cache, performance issues, etc.) rather than that are related to the problem. Implementation details are related to the carrier rather than the entity. In this scenario, I think it is appropriate to use encapsulation, and data structure is a typical example. Algorithm is sequential operation, circuit can also be very complex, but it is generally not called algorithm, but sequential logic.

What about polymorphism? Looking at it separately, it should only be used when the function selection is determined by the external input or other runtime results. However, sometimes the polymorphism of objects seems to be a requirement of some larger design pattern. In this case, it should have the problem of hindering inlining, but I have no experience. So it is still to use the abstract method according to the level of the problem.

I did not consider the type system problem here, that is, using inheritance to make sum type.