Evolution of Common Lisp Object System

Josef Matějka

01.02.2023

1 Introduction

Object-oriented programming was and still is one of the main paradigms used today. Languages like C#, Java, Javascript, Python, and many more are said to be OOP. They provide strong encapsulation of data - classes have private/protected/public methods and/or slots, users can derive new classes using inheritance, and they can overload methods in those new classes. It is usually said that the encapsulation of data and dynamic dispatch are regarded as fundamentals of OOP (for example see: `https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop`).

In this short article, we would like to look at the evolution of the different object-oriented systems for a language called Common Lisp. Lisp language was invented/discovered by John McCarthy in 1958, and in this regard, it is one of the oldest programming languages, the object system was added to the Common Lisp in 1986, therefore it is also much older than Java, C#, and Python. Although it is old, we believe it has some forgotten ideas people should know about.

2 Lisp and its dialects

Lisp is an umbrella term that includes many programming languages and dialects, since McCarthy's discovery many developers went and implemented their own version of Lisp[1]. In this article we will work with three Lisp languages first one will be MACLISP, then Interlisp and last we will talk about Common Lisp. For our purposes the differences between those languages are not important, therefore we often will call them just Lisp.

3 Object Oriented Programming

Before we start with Lisp we will look at a different programming language, which was developed by Alan Kay and his colleagues (Adele Goldberg and Dan Ingalls) in the 1970s in Xerox Palo Alto Research Center.[2] The language is called SMALLTALK, it is the first pure object-oriented language, it has brought the OOP paradigm to the public. The version SMALLTALK-80 was featured on the cover of Byte Magazine.

Alan Kay in his research has introduced six main ideas for SMALLTALK-72 (it was released in 1972), which can be regarded as basic rules for what it means object-oriented programming[2]. They are:

1. Everything is an object.
2. Objects communicate by sending and receiving messages (in terms of objects).
3. Objects have their own memory (in terms of objects).
4. Every object is an instance of a class (which must be an object).
5. The class holds the shared behavior for its instances (in the form of objects in a program list).
6. To eval a program list, control is passed to the first object and the remainder is treated as its message.[1]

The first rule simply gives regularity to the language. The second rule tells us how to work with objects, sending messages in SMALLTALK is similar to invoking the method of an object in modern OOP languages. Let us illustrate it in the following example:

```
15 between: 1 and: 16
```

We send a message between:and: with parameters 1 and 16 to the object 15, in C# we would send something like

```
15.BetweenAnd(1, 16);
```

The main reason for this rule is the regularity of syntax. Compared to C# there is no special syntax for instance creation, you simply send a message new to the appropriate class and the instance of the class is created. In SMALLTALK everything happens by sending messages, which makes the syntax regular and easy to understand.

The rules 3., 4. and 5. will be familiar to a modern OOP developer. Objects have slots/fields where some data can be stored, or loaded. Every object is an instance of a class - in SMALLTALK this means that also a class is an instance of another class, which is usually called a meta-class. Also, the instances of a single class behaving according to their class, therefore the class holds their behavior. The last rule introduces a lisp-like property to SMALLTALK[2], this allows for the evaluation of a list of objects. It was added by Alan Kay who was familiar with Lisp because he deemed it useful.

Interestingly encapsulation and inheritance are not listed them. SMALLTALK itself does not have any notion of private/protected and/or public methods, everything is visible to the developer. Also, let us note that the inheritance was in SMALLTALK added in a later version.

[1]This last rule was added for convenience, this is similar to what Lisp does. First element of a list is regarded as a function/object which will evaluate the rest of the items in the list.

4 FLAVORS

The first object-oriented extension was developed by Howard Cannon at the MIT Artificial Intelligence Laboratory for a lisp language called MACLISP. It was released in 1979, three years after SMALLTALK-76. We will look at the manual called Flavors: Message passing in the Lisp Machine written by Daniel Weinreb and Daniel Moon[3], the name already suggests that message passing is the main idea. This is close to Alan Kay's thinking which has famously said:

> The big idea is messaging. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

The manual lists modularity as the main benefit of OOP. They claim that if developers has at their disposal modular programming constructs and techniques it helps and encourages to write programs that are easy to use and easy to understand. Another benefit of an object is that it can hide its implementation from the developer and they can use it as a black box. As long as the developer follows the documented interface they can use the objects safely.[3]

Since Lisp is more of a hacker language, which is fully reflexive and does not hide anything from the developer, the manual gives emphasis on following the documented interfaces and avoiding using hacks (like using aref to get property from the object).

4.1 Code example

We will take the first code sample from the manual, where we describe the main ideas and functionality of FLAVORS. In this system there are no classes, what we usually call class is here under the name flavor. On top of the object hierarchy was vanilla flavor, mixins are usually described as mixing flavors of ice cream together.[3]

```
(defflavor ship (x-position y-position
x-velocity y-velocity mass)
()
:gettable-instance-variables)

(defmethod (ship :speed) ()
(sqrt (+ (^ x-velocity 2)
(^ y-velocity 2))))

(defmethod (ship :direction) ()
(atan y-velocity x-velocity))
```

We define flavor/class ship with slots x-position, y-position, x-velocity, y-velocity, and mass. The flavor has two methods: speed and direction. The symbol :gettable-instance-variables specified that the slot can be read, user can also specify :settable-instance-variable and :initable-instance-variables which allows for defining initial values for slots in the constructor.

Suppose we have :initable-instance-variables we will now show how the instance of flavor ship can be constructed:

```
(make-instance 'ship
':x-position 0.0
':y-position 3.0
':mass 3.5)
```

To send a message to the instance of ship we have to use the function funcall. If we have variable my-ship which is an instance of ship flavor we can send message :direction this way:

```
(funcall my-ship ':direction)
```

In MACLISP funcall has the same meaning as in Common Lisp, the first argument is a function that will be invoked, and the rest are the arguments[4]. This means that internally each object was callable/functional and the message was passed to it as the argument.

4.2 Mixing flavors

We have already mentioned so-called mixins, these special flavors served for extending the functionality of other flavors. These mixins can be viewed as an abstract class or interface in languages like C#, mixins also should not be instantiated but only used through inheritance[3].

Before we show the example we need to talk about inheritance and method combination. In modern OOP languages, the user can define a class, and declare some methods as virtual and those can be overloaded in the class descendants. In flavors all prime methods are virtual, therefore any prime method can be overloaded in the descendant flavor.

But Flavors object system can do much more. There are three types of methods, before, after, and primary methods. When we invoke a method in a flavor this happens:

1. All before methods are run.
2. Single primary method is run.
3. All after methods are run.

In the flavor definition user can define in which order will the methods be run (from closest parent to the furthest or the opposite) and also what happens with the results of those methods - they can be concatenated together, they can be aggregated or only the result of primary method can be returned[3].

Back to the example, which is taken from the article The structure of a programming language revolution[5] by Richard P. Gabriel. Suppose we want to create three mixins greeter, bachelor, and exclamation. All these mixins will have a method greet that we utilize, we also want to define a flavor person with name and surname slots and with the greet primary method. Lastly, we will create a flavor mgr-student which will also have method greet, when we invoke greet on the mgr-student we will get the string "Hello, bc. {name} {surname}".

```
; greeter mixin
(defflavor greeter ()
()
(:method-combination :base-flavor-first :greet))

; greeter before method
(defmethod (greeter :before :greet) ()
(format nil "Hello, "))

; person flavor
(defflavor person (name surname)
:initable-instance-variables
:gettable-instance-variables)

; this is primary method on person,
; it is overloaded in flavor mgr-student
(defmethod (person :greet)
(format nil "~A" (funcall self :name)))

; bachelor mixin
(defflavor bachelor ())

; bachelor before method
(defmethod (bachelor :before :greet) ()
(format nil "bc. "))

; exclamation mixin
(defflavor exclamation ())

; exclamation after method
(defmethod (exclamation :after :greet) ()
(format nil "!"))

; mgr-student inherits from person flavor
; and from mixins greeter, bachelor and
; exclamation
(defflavor mgr-student (greeter
bachelor
person
exclamation))
```

```
; when invoked, this primary method will call
; greeter before method
; bachelor before method
; this primary method
; exclamation after method
(defmethod (mgr-student :greet) ()
(format nil "~A ~A" (funcall self :name)
(funcall self :surname)))
```

4.3 New Flavors

In the year 1986 the extension Flavors was ported for the Common Lisp language by David Moon. The major change is in the method definition, same as in current CLOS user must define generic[2] using defgeneric macro and after that, they can create methods using defmethod macro, methods are no longer invoked using funcall macro, but they can be invoked directly[6].

The shift from "sending messages" to invoking generic methods is in our opinion important, although generic methods were used before New Flavors, their invocation was still interpreted semantically as sending a message. With generic methods in New Flavors we do not have to type funcall or send we can invoke any method just like any ordinary function in Lisp, which makes the interface for OOP more consistent with the whole language. Also, it removes the difference between the method of an object and a classic function, this allows users to think more functionally about their programs and gives the OOP programming functional flavor.

4.4 LOOPS

Another popular OOP extension for Lisp was released in 1983 for Interlisp language, its name is LOOPS and it was developed by Daniel G. Bobrow[7]. The manual tells us, that the LOOPS is a more ambitious project than LOOPS, since it adds object-oriented, data-oriented, and ruleset-oriented programming to Lisp. We will mostly focus on the object-oriented aspect and maybe other features that directly influenced CLOS (Common Lisp Object System).

It follows Alan Kay's ideas closely, for example, each class is also an object - we did not encounter this idea in Flavors. In LOOPS there is still the concept of sending messages to objects, this is done using the ← operator:

```
(← object message arg0 ... argn)
```

We create an instance of a class using the message New sent to the class. And we can create a new class by sending a message New to the meta-class. This shows how much the LOOPS followed the mantras "everything is an object" and "objects communicate by sending and receiving messages"[7].

[2]Generic method in Lisp is a method that provides dynamic dispatch according to the classes passed to the method.

To show the similarities between LOOPS and SMALLTALK, let us show a small example in which we create simple class named StudentEmployee which inherits from classes Student and Employee. We could start by:

```
(← ($ Class) New 'StudentEmployee '(Student Employee))
```

Here ($ Class) represents metaclass for classes. Or we could simply use macro DC:

```
(DC 'StudentEmployee '(Student Employee))
```

After that user could write (EC 'StudentEmployee) and according to the manual this command would open an editor looking like this:

```
[DEFCLASS StudentEmployee
(MetaClass Class Edited: (*   lc:   "18-Oct-82 14:26"))
(Supers Student Employee)
(InstanceVariables)
(Methods]
```

The user could edit and save the class, then it would be recompiled. Programming in LOOPS was interactive, whenever the user wanted to edit a class or method an editor opened with the method/class definition, after the user saved their changes the method/class would get recompiled[7]. This is similar to how the user usually works with System Browser in Smalltalk-80.

The inheritance and method combination is very close to how it works in modern OOP languages (and SMALLTALK). When we create a new class, we can override a method behavior, we can also invoke the definition in the parent using the operator ←Super or we could invoke the same method in all the ancestors using the operator ←SuperFringe. When we compare this to flavor we do not have a concept of :before or :after methods.

Also, we find it interesting that LOOPS had a concept of a getter and a setter as we know it from C#. We could for each slot define a special get method that would be invoked when we would access the slot and the same goes for the set method[7]. Let us note that this functionality was added to C# 6.0, in this sense LOOPS was ahead of time.

4.5 CommonLoops

CommonLoops was released in 1986 after the successful consolidation of Lisp dialects into one Common Lisp language. As we dived into CommonLoops we expected it to be just an extension of LOOPS rewritten for Common Lisp, but in reality, CommonLoops is something entirely different.

It is a Common Lisp Object-Oriented System that aims to provide compatibility with Lisp functional style, it aims to provide a powerful base for writing your own object-oriented languages and it should be easily portable to other lisp systems. The CommonLoops paper says that this system is a good base for implementing languages like Flavors, LOOPS, ObjectLisp, and others, this suggests that the CommonLoops aims to take the base of OOP and implement it in efficient and portable way[8].

Still, we do not call or invoke methods in CommonLoops but send messages, this is done using the macro called send like this:

```
(send object message arg0 ... argn)
```

Internally this call is translated into

```
(funcall (function-specified-by object) arg0 ... argn)
```

The definition of the method is completely different from LOOPS system and is more similar to Flavors, for example, we define method move for class block:

```
(defmeth move ((obj block) x y)
; ... implementation
)
```

Interestingly we can define multi-methods which are specialized not only by the class of receiving object but also by the class of some (or all) of its arguments, this means that we could define a multi-method move that would behave differently if x argument is an integer or complex number, same goes for argument y. Multi-method could look like this:

```
(defmeth move ((obj block) (x complex) (y complex))
; ... implementation
)
```

The algorithm is the following: we look into the class of the instance that received the message then we check against the defined method trying to find the one, that is mostly specialized to our call. Also, it is worth mentioning, that CommonLoops gives the ability to write user-defined method-lookup[8].

Classes are also defined similarly to Flavors, we use defstruct macro, and multiple inheritance is still allowed. In this sense it is pretty standard, the user can define from which meta-class this class inherits, can define slots (class and instance) and their accessors - for example, read, write, or both. Since defstruct is a macro that also exists in plain Common Lips it needed to be rewritten in order to accept also the class definition.

CommonLoops came up with a robust object hierarchy, that incorporated all built-in types of Common Lisp. Every class is derived from T (which is the true value in Common Lisp), and direct descendants of T are built-in types like number, sequences, and so on. Also the class object is a descendant of T. Most user-made classes are derived from a class called simply class which is a descendant of object.

In conclusion, CommonLoops should be viewed as a framework that can be utilized to write better OOP languages in Lisp. In the paper, it is mentioned that its abilities are strict super-set to New Flavors and users can even write New Flavors using CommonLoops. This level of extensibility fully aligns with lisp culture, where the user is given the ability to rewrite almost anything to their liking.

4.6 Common Lisp Object System

The work on Common Lisp Object System started in 1986. The process for ANSI standard took two years and the result is based mainly on CommonLoops and New Flavors. The main idea was to create a system that is:

1. simple to use,
2. functional at heart,
3. and extensible.[9]

Therefore the new system has generic methods from New Flavors, but also allows dynamic dispatch on multiple arguments (multi-methods from CommonLoops), it also supports method combination (:after, :before and others). The object hierarchy was taken from CommonLoops, this means that everything in Common Lisp is an object and we can define meta-classes that can define the behavior of derived classes - this is part of meta-object protocol[9].

4.7 Conclusion

We have seen the evolution of object-oriented programming in LISP, the functional nature of the language sharped also the OOP paradigm, where we focus more on generic functions than the objects itself. Also it is worth noting, that CLOS provides many features that are not seen in other OOP languages like method combination and multi-methods, which allow for dynamic dispatch on all of its arguments. Plus if user is not happy with the current CLOS they can extend or change it to their liking using the meta-object protocol.

References

[1] G. L. Steele and R. P. Gabriel, "The evolution of lisp," in *History of programming languages—II*, pp. 233–330, 1996.

[2] A. C. Kay, "The early history of smalltalk," in *History of programming languages—II*, pp. 511–598, 1996.

[3] D. Weinreb and D. Moon, "Flavors: Message passing in the lisp machine.," tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1980.

[4] D. A. Moon, *MACLISP reference manual.* Massachusetts Institute of Technology, 1974.

[5] R. P. Gabriel, "The structure of a programming language revolution," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 195–214, 2012.

[6] S. C. Lisp, "Language concepts," *Symbolics, August*, 1986.

[7] D. G. Bobrow and M. Stefik, *The LOOPS manual.* Knowledge Systems Area, Xerox Palo Alto Research Center, 1983.

[8] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "Commonloops: Merging lisp and object-oriented programming," *ACM Sigplan Notices*, vol. 21, no. 11, pp. 17–29, 1986.

[9] L. G. DeMichiel and R. P. Gabriel, "The common lisp object system: An overview," in *ECOOP'87 European Conference on Object-Oriented Programming: Paris, France, June 15–17, 1987 Proceedings*, pp. 151–170, Springer, 2000.