Describing Dynamic Organizational Structures through the Common Lisp Object System

Lennard Wolf
lennard.wolf@student.hpi.de

Hasso Plattner Institute
Prof.-Dr.-Helmert-Straße 2-3
14482 Potsdam
Germany

Abstract. Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System Describing Dynamic Organizational Structures through the Common Lisp Object System

My notations

– "term" : wording unsure

1 Introduction

– Topic / Domain
– Problem
– Contribution
– Outline (The following is a summary of each section. In Section...)

2 Problem

– To gain an overview of all the people within organizational structures, it is often useful to have the participants and their roles written down somewhere.
– say a system needs all participants to be objects
– their occupations would define their possible interactions, rights etc
– in many structures will not only the people come and leave, but they might also change their occupations
– example case university: professors can also be "CEOs", students can also be tutors etc

– how would it be possible to describe this within a software system, so that it is easily maintainable - meaning that individuals can change their jobs etc - and that it is easily extendable - meaning that new occupations can easily added and combined with already existing ones

3 Background

A "solid" solution to the problem introduced in Section 2 would be to use the *Common Lisp Object System* (CLOS), an extension to Common Lisp[1] providing "full" object-orientation. [1] However, to understand its utilization in such a problem domain as well as the reasoning behind that, some background information will be needed first.

In this Section we will hence start out by introducing the general ideas behind object-orientated programming languages, as well as the language Lisp, which is followed by a description of the three main concepts that make CLOS unique. An overview of the history of CLOS will be provided thereafter, so as to better understand its origins and historical context.

3.1 Object-Orientated Languages

A definition of *object-orientated* programming languages requires a preceding explanation of the terms *object*, *class*, and *inheritance*. *Objects* have a set of *attributes* that define its *state*, as well as a number of *messages*[2] that it can receive to evoke certain behavior. By sending such messages, objects can interact with one another. *Classes* are templates for objects, so that objects of the same class can have uniform interfaces and behavior. An object is hence an *instance* of a class. *Inheritance* opens up the possibility to create class hierarchies, so that classes can be specializations of others and *inherit* certain behavior while adding something unique. The inheriting class is called the *subclass*, the other the *superclass*.

An *object-orientated* programming language is one that has these three concepts "integrated". They give programmers the ability to model human language based conceptualizations of the real world, since these are also just seperations of phenomena into groups with common traits.

3.2 Lisp

Stemming from the term **Lis**t **P**rocessor, Lisp is a programming language in which *everything is a list*. This means that there is *no discernment between data and code*. Hence an expression such as `(plus 3 4)` is without context nothing more than a "meaningless" listing of the "meaningless" expressions `plus`, `3`, and `4`. An expression like that only gets its meaning from an *evaluation* by

[1] `https://common-lisp.net`, accessed: 2016-07-12
[2] In CLOS, these are referred to as callable `methods`.

the interpreter which considers it in a certain context. In such a context the expression `plus` could be a function to add the arguments it is given, but that is entirely arbitrary.

Common Lisp is a standardized version of Lisp which provides certain data types and operations. But this type system can be extended through the use of *macros*, which let developers give meaning to expressions. This feature of Lisp greatly facilitates the creation *Domain Specific Languages*. [2] They also form the basis of CLOS which in its entirety consists of 8 macros and 33 functions.

explain Macros in detail?

3.3 History of CLOS

In the late 1960s, the new concept of object-orientation started to become a topic of interest for researchers in computer science. In 1986, four major attempts, namely New Flavors[3], CommonLoops[4], Object Lisp (LMI), and Common Objects, were made to bring that concept to the popular and easily extendable programming language Lisp. [5] To create a standardized object system, researchers from Symbolics, Inc. and Xerox PARC met to combine their respective object systems, New Flavors and CommonLoops. [1] The Common Lisp Object System was the result, which had been two years in the making. [5] Important concepts, next to those described in the following *Main Concepts* passage, were *metaclasses*, of which regular classes are just instances, and *metaobjects*, which lay the foundation for the concept of objects. [6] The *metaobject protocol* in CLOS allows the system to be written in itself. [5] Furthermore, CLOS is portable to different LISP dialects. Examples are *EIEIO*[5] for Emacs Lisp and *SOS*[6] for Scheme.

3.4 Main Concepts

CLOS comprises concepts that are rather uncommon in modern object-orientated languages. This is, in the cases of *generic functions* and *method combination*, due to their specificity to the Lisp environment and, in the case of *multiple inheritance*, because it can cause many problems if employed in contexts that are inappropriate, which they more often than not are. [**?**]

We will thus start out if a short introduction of the new macros supplied by CLOS, which we will then need to explain the aforementioned concepts.

Syntax Basics The core of object-oriented programming lies with the creation of classes, corresponding methods, and the instantiation of objects. Listing 2

[3] Information on its predecessor Flavors can be found in [3].

[4] Information on CommonLoops can be found in [4].

[5] `https://www.gnu.org/software/emacs/manual/html_mono/eieio.html`, accessed: 2016-07-13

[6] `https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-sos/`, accessed: 2016-07-13

shows these basic operations in CLOS. Other macros exist[7] but are not relevant in this context.

```
;;; Generic form of class and method definition, as well as instantiation
(defclass NAME (SUPERCLASS1 ... SUPERCLASSn) (ATTRIBUTES))

(defmethod NAME (RECIPIENT1 ... RECIPIENTn) (CODE))

(make-instance 'CLASS :ATTR1 VAL1 ... :ATTRn VALn)


;;; Examples of class and method definition, as well as instantiation
(defclass teacher (person)              ; teacher is subclass of person
  ((subject :accessor teacher-subject   ; teacher has attribute subject
            :initarg :subject)))

(defmethod what-is-my-subject ((teach teacher))
  (format t "I am a teacher in ~d." (teacher-subject teach)))

(make-instance 'teacher :name "Mary" :subject "Maths")
```

Listing 1: The central macros provided by CLOS

Multiple Inheritance was first introduced by the predecessor of New Flavors, *Flavors*. It allows classes to have multiple superclasses, thereby inheriting methods and attributes of each (see Figure 1). The AthleticTeacher class defined by `(defclass athletic-teacher (teacher athlete) () )` would, if some inheritances from Teacher and Athlete are conflicting, prioritize the Teacher behavior, because it is listed first.

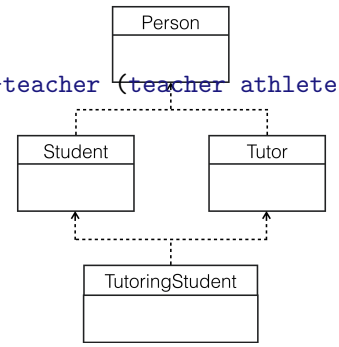

Figure 1: A UML class diagram with multiple inheritance

Method Combination is an important addition to multiple inheritance, because it lets developers blend behavior together. The keywords `:before`, `:after`, and `:around` allow the specification of certain behavior to be added to a called method. `(defmethod method :after () )` would run additional behavior after the primary behavior defined for `method` for all Teacher.

[7] The entire implementation of CLOS can be found at `http://norvig.com/paip/clos.lisp`, accessed: 2016-07-13

Generic Functions In Flavors, messages were sent to objects by calling the `send` function (`(send object :message)`). New Flavors however introduced the `(message object)` notation, which required `message` to be *generic*, meaning that it needed to be a globally defined interface (see Figure 2). By calling the generic function, different behavior will ensue, depending on the type of the receiver provided as argument. If methods are defined that have no corresponding generic function, CLOS will add it automatically at compile time.

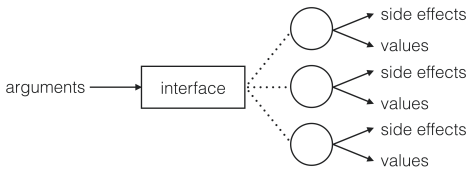

Fig. 2. The global interface provided by the generic function

4 Approach

- 3 concepts will be used
- maybe give function that lets user add classes

5 Implementation

- give code details for concepts from Approach

6 Evaluation

- DSL type declaration is easily maintainable
- using langs like Java (?) might be problematic, give example

7 Conclusion

Write me pl0x

```
(defgeneric who-am-i (person)
      (:documentation "Makes the person describe themselves."))

(defmethod who-am-i ((p person))
    (format t "My name is ~d and I am ~d years old.~%" (person-name p) (person-age p)))

(defmethod who-am-i ((p teacher))
    (format t "My name is ~d and I am ~d years old.~%" (person-name p) (person-age p))
    (format t "I am a teacher and my subject is ~d.~%" (teacher-subject p)))

(defmethod who-am-i :after ((p teacher))
    (format t "I am a teacher and my subject is ~d.~%" (teacher-subject p)))

(defmethod who-am-i :after ((p athlete))
    (format t "I am an athlete and my sport is ~d.~%" (athlete-sport p)))

(defmethod who-am-i :around ((p athletic-teacher))
    (format t "Oh, hi!~%" )
    (let ((result (call-next-method)))
      (format t "Gotta sprint back to class, bye-bye!~%")
      result))
```

Listing 2: The central macros provided by CLOS

References

[1] DeMichiel, L.G., Gabriel, R.P.: The common lisp object system: An overview. In: European Conference on Object-Oriented Programming, Springer (1987) 151–170

[2] Fowler, M.: Domain-specific languages. Addison-Wesley, Upper Saddle River, NJ (2011)

[3] Moon, D.A.: Object-oriented programming with flavors. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. OOPLSA '86, New York, NY, USA, ACM (1986) 1–8

[4] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: Commonloops: Merging lisp and object-oriented programming. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. OOPLSA '86, New York, NY, USA, ACM (1986) 17–29

[5] Steele Jr, G.L., Gabriel, R.P.: The evolution of lisp. ACM SIGPLAN Notices **28** (1993) 231–270

[6] Kiczales, G., Des Rivieres, J., Bobrow, D.G.: The art of the metaobject protocol. MIT press (1991)

[7] Abelson, H., Sussmann, G.J., Sussmann, J.: Structure and Interpretation of Computer Programs. 2nd edn. The MIT Press, Cambridge, Massachusetts (1996)