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

1 Introduction

Organizational structures often require maintainable and easily extendable descriptions of shareholders and their roles. The *Common Lisp Object System* (CLOS) with its concepts of *multiple inheritance*, *generic functions*, and *method combination* is a tool that facilitates the creation and perpetuation of such a register of members. This work presents an overview of CLOS' history as well as its basic syntax and core concepts by employing it for the creation of a dynamic member register for a university.

This work is structured as follows. Section 2 introduces the problem of dynamic shareholder lists for organizational structures. Section 3 introduces CLOS, its history, and core concepts. In Section 4, CLOS is applied to the given problem, first in theory, then in practice. Section 5 gives a short evaluation of the solution based on the given criteria, in Section 6, other possible solutions are discussed, and Section 7 concludes this work.

2 Problem

To achieve their goals, organizational structures require planning as well as control over processes and the network of shareholders [1, 42f.]. To gain the necessary overview, it is often useful to have a list of all members and their roles. In the United Kingdom[1] as well as other countries such a *register of members* is furthermore required by the legislation.

Since organizations are often dynamic, such a list should also be easily *maintainable*, meaning that participants can be both added or removed as well as change their role. It should also be *extendable* meaning that new roles can easily added and combined with already existing ones. Furthermore, computer programs used within the organization might have to interact with the list, so *accessibility* is another key requirement. These *evolution qualities* [2] would thus be the key requirements for any list describing the participants within an organizational structure.

3 Background

A solution to the problem introduced in Section 2 which would meet all the given requirements would be the employment of the Common Lisp Object System, an extension to Common Lisp[2] providing object-orientation [3]. 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-Oriented Languages

At its core, Object Orientation is a concept in computer science, after which a program works in such a way that it is made up of individual objects that can interact with one another.

– according to... oop is usually...
– give example of p.. oo

A definition of *object-oriented* 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* that it can receive to evoke certain behavior, defined by *methods*. 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*

[1] Companies Act 2006 `http://legislation.data.gov.uk/ukpga/2006/46/part/8/chapter/2/data.htm?wrap=true`, accessed: 2016-07-18
[2] `https://common-lisp.net`, accessed: 2016-07-12

certain behavior while adding something unique. The inheriting class is called the *subclass*, the other the *superclass*.

An *object-oriented* 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 separations 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 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 can be demonstrated by a trivial example. By defining a macro `(defmacro eight () (+ 3 5))`, a programmer would be able to simply write `eight` instead of the actual number whenever it is needed in the code. In the *macro expansion phase* during compilation, the compiler then replaces all occurrences of `eight` with `(+ 3 5)`, which, during evaluation, will turn into the desired number. This feature of the Lisp language enables programmers to interfere with evaluation orders, and also greatly facilitates the creation of *Domain Specific Languages* [4]. Furthermore, they form the basis of CLOS which in its entirety consists of 8 macros and 33 functions.

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 [7]. To create a standardized object system, researchers from Symbolics, Inc. and Xerox PARC met to combine their respective object systems, New Flavors and CommonLoops [3]. The Common Lisp Object System was the result, which had been two years in the making [7]. 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[8]. The *metaobject protocol* in CLOS

[3] Information on its predecessor Flavors can be found in [5].
[4] Information on CommonLoops can be found in [6].

allows the system to be written in itself [7]. 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-oriented 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 3 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)
```

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.

[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

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

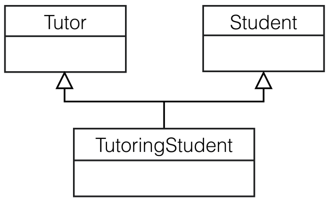

Fig. 1. A UML class diagram with multiple inheritance

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 ((teach teacher)) ()` would run additional behavior after the primary behavior defined for `method` for all Teacher.

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.

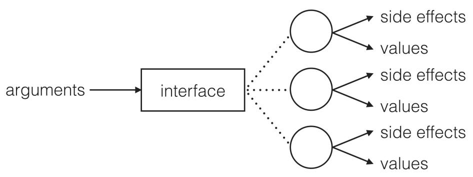

Fig. 2. The global interface provided by the generic function

4 Concept and Implementation

As discussed in Section 3.1, object-orientation is suitable for the description of real world states of affairs. Together with its concepts, CLOS thus lends itself to the creation and maintenance of descriptions of organizational structures.

In our solution, classes represent the roles, while an instance of such a class would be an individual member. This way, a Lisp REPL can be used for simple queries, while other Lisp-programs (such as human capital management software) can use the objects for further interaction. CLOS furthermore permits the storage of the objects in SQL database systems via object-relational mappers[8], so that non-Lisp programs can also have easy access. Nevertheless, this work focuses on a description of the arrangement of people within an organizational structure and an interface to query information on individuals and occupations.

A common organizational structure is the university. As new students and chairs come and go, new roles and corresponding individuals will have to be accommodated for. The following is an abstract description of a possible arrangement of roles which is visualized in Figure 3. Since all members have a certain set of common attributes such as name and adresse, an abstract class `Person` for all members to inherit from will have to exist. As it is possible that the university's chairman is also a professor, occupations need to be combinable. This is possible through multiple inheritance so that `ProfessorChairman` would simply be a subclass of both `Professor` and `Chairman`, as described in Section 3.4.

A distinction between students and staff would make sense, since the staff needs bank details for compensations. In reality however, this line is blurred by multiple inheritance, as a student might also be a research assistant.

When making queries via the Lisp REPL, a simple interface should make responses that should share a common format. And since the addition of new occupations should not require any alterations to such an interface, the method combination concept would prove its usefulness.

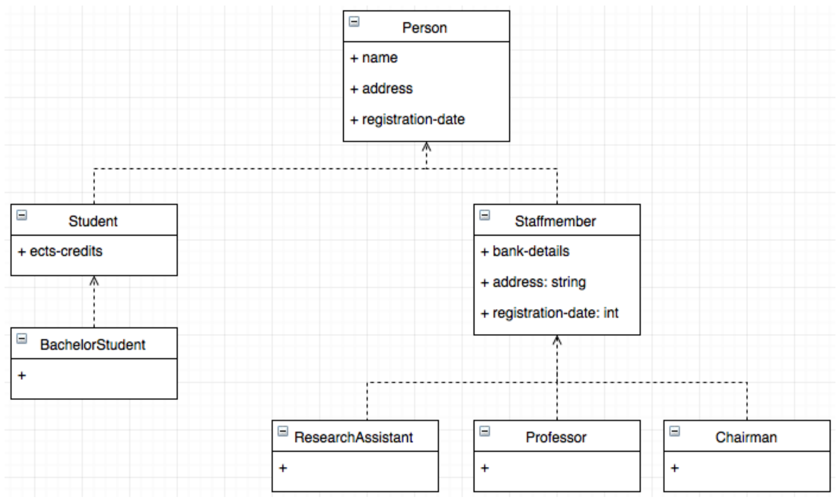

Fig. 3. Excerpt from a UML class diagram of a description of the occupations within a university WIP FIX ARROWS

[8] A list of object-relational mappers for CLOS can be found at `http://www.cliki.net/ORM`, missing from the list is Mito (`https://github.com/fukamachi/mito`), both accessed: 2016-07-18

– common classes first
– relationships?
– add chair class?
– function for creation of student in REPL?

```
(defclass person ()
  ((name :accessor person-name
         :initarg :name)
   (age :accessor person-age ;is the person-* accessor not kind of stupid?
        :initarg :age)
   (address :accessor person-address
         :initarg :address)
   (registration-date :accessor person-registration-date
         :initarg :registration-date)))

(defun make-person (name age adress registration-date)
  (make-instance 'person :name name :age age :address address
     :registration-date registration-date))
```

Listing 2: The basic person class WIP

```
(defclass staffmember (person)
  ((bank-details :accessor staffmember-bank-details
            :initarg :bank-details)))

(defun make-staffmember (name age subject adress registration-date bank-details)
  (make-instance 'staffmember :name name :age age :address address
     :registration-date registration-date :bank-details bank-details))

(defclass student (person)
 ((ects-points :accessor student-ects-points
           :initarg :ects-points)))

(defun make-student (name age subject adress registration-date ects-points)
 (make-instance 'student :name name :age age :address address
    :registration-date registration-date :ects-points ects-points))
```

Listing 3: The interface to access the data in a human readable manner WIP

5 Evaluation

In Section 2 we defined as main requirements the three evolution qualities *maintainability*, *extensibility*, and *accessibility*. In this Section we will discuss, whether and to what degree the solution given in Section 4 can satisfy them.

– talk about all requirements given in intro!!
– DSL type declaration is easily maintainable
– future work: relations between members, could be used as framework for any such structure; UI is easy

6 Related Work

– using langs like Java (?) might be problematic, give example mult inheritance: needs interfaces as shown in https://stackoverflow.com/questions/21824402/java-multiple-inheritance
– other examples of how it can be solved + vergleich

7 Conclusion

Write me

References

[1] Schmidt, G.: Einführung in die Organisation. Gabler Verlag, Wiesbaden (2000)

[2] Young, R.R.: Effective requirements practices. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)

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

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

[5] 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

[6] 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

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

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

APPENDIX

```
(some code here)
```

Listing 4: The implementation of the described system in full WIP