Describing Dynamic Organizational Structures with 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. CLOS is an object-oriented system for Common Lisp. In this work, we present its history, basic syntax, as well as its three main concepts of *multiple inheritance*, *method combination*, and *generic functions*. This is done by showing their application to the domain of descriptions of organizational structures. The university is used as example use case and the implementation of a maintainable, extendable, and accessible framework to describe its structure is presented.

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, a different solution using Java is 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. This *paradigm* can be implemented into a language in different ways. But next to such variations such as prototype-based programming, the quintessential Object Orientation style is class-based, according to [4], which we refer to in the following definition.

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* certain behavior while adding something unique. The inheriting class is called the *subclass*, the other the *superclass*. [4]

[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

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* [5]. 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 [8]. 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 [8]. 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[9]. The *metaobject protocol* in CLOS allows the system to be written in itself [8]. Furthermore, CLOS is portable to

[3] Information on its predecessor Flavors can be found in [6].
[4] Information on CommonLoops can be found in [7].

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 [10]. 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 1 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

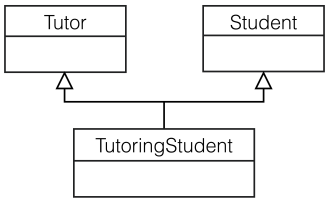

Fig. 1. A UML class diagram with multiple inheritance

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.

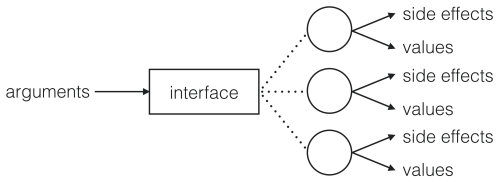

Fig. 2. The global interface provided by the generic function

4 Approach 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 of the organization. This way, a Lisp REPL can be used for simple queries, while other Lisp-programs (such as human capital management software) could use the objects for further interaction. CLOS

furthermore permits the storage of 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.

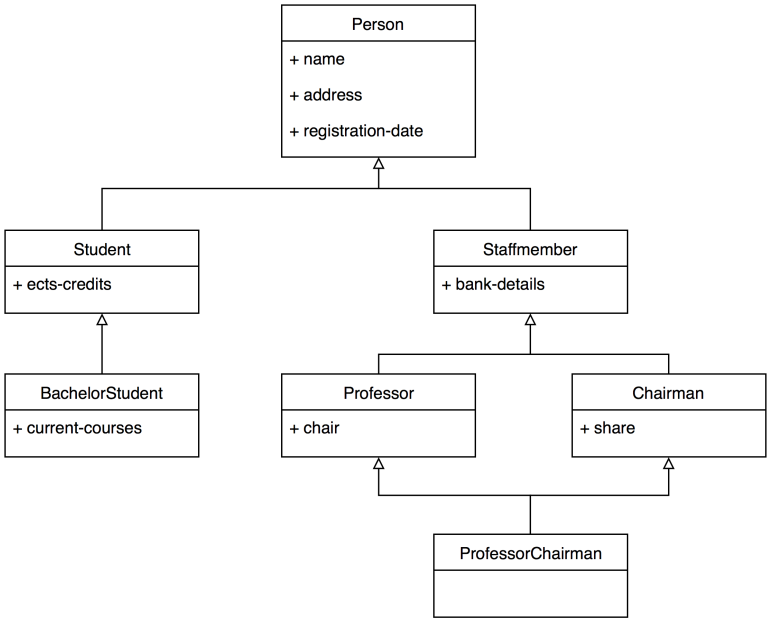

Fig. 3. Excerpt from a UML class diagram of a description of the occupations within a university

A common organizational structure is the university, which we will use to exemplify our approach to the general problem presented in Section 2. 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 address, an abstract class `person` for all members to inherit from will have to exist. Listing 2 shows a possible implementation of such a class. Even though `person` is an abstract class, it is still possible to instantiate it. However, since

[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

such a `person` object would not provide any meaning to the overall system, our approach omits any helper method in the form of `make-person` or the like.

```
(defclass person ()
  ((name :accessor name
         :initarg :name)
   (age :accessor age
        :initarg :age)
   (address :accessor address
         :initarg :address)
   (registration-date :accessor registration-date
         :initarg :registration-date)))
```

Listing 2: The basic `person` class

A distinction between students and staff members 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 at some point become a research assistant. A `staffmember` class would, just like `person`, again be abstract to facilitate the centralization of common attributes and behaviors. Such a distinction will obviously be unnecessary in other organizations, but it helps in making sense of the inherent structures. Listing 3 shows the definition of the `student` and `staffmember` classes, which both inherit from student. As `student` is not an abstract class, it is useful to create an easy-to-use `make-student` function, since the `make-instance` function can be felt to be rather tedious to work with.

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

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


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

Listing 3: The definition of the `student` and `staffmember` classes

As it is possible that a member of an organization can take on multiple roles within the structure, classes will need to be able to have multiple superclasses. An

example would be the university's chairman who also happens to be a professor. In our solution, this is possible through CLOS' concept of multiple inheritance. As shown in Listing 4, `professor-chairman` would simply be a subclass of both `professor` and `chairman`[9]. Here it is important to make sure that the order in which the superclasses are declared is in such a way that in the case of duplicate methods or attributes, the superclass named first is the one to have priority. If prioritization however is not desired, then the naming will have to be adjusted.

```
(defclass professor (staffmember)
  ((chair :accessor chair
          :initarg :chair)))

(defclass chairman (staffmember)
  ((share :accessor share
          :initarg :share)))

(defclass professor-chairman (professor chairman) () )
```

Listing 4: The definition of the `professor`, `chairman`, and `professor-chairman` classes

When making queries via a Lisp REPL[10], 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. As can be seen in Listing 5, the generic function is defined first via `defgeneric`, and then the individual implementations for each class are defined. With `:after`, each additional class can concatenate a particular own behavior to the general behavior of its superclass(es).

Figure 4 demonstrates a possible interaction with the framework for describing organizational structures.

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 extent the solution given in Section 4 can satisfy them.

Maintainability The `make-CLASSNAME` functions provide an easy to use interface to add new members to the system. The only issue here would be the

[9] The instantiation interfaces like `make-professor` have been omitted to declutter the code. In an actual implementation however they should exist as well.

[10] *Read-Evaluate-Print Loop*

```
(defgeneric get-info (person)
      (:documentation "Returns data on all attributes."))

(defmethod get-info ((p person))
    (format t "Name: ~d ~% Occupation: ~d ~% Age: ~d ~%
     Address: ~d ~% Date of Registration: ~d ~% "
     (name p) (class-of p) (age p) (address p) (registration-date p)))

(defmethod get-info :after ((s student))
    (format t "ECTS Points: ~d ~% " (ects-points s)))

(defmethod get-info :after ((s staffmember))
    (format t "Bank Details: ~d ~% " (bank-details s)))

(defmethod get-info :after ((p professor))
    (format t "Chair: ~d ~% " (chair p)))

(defmethod get-info :after ((c chairman))
    (format t "Share: ~d ~% " (share c)))
```

Listing 5: The functions for human readable output WIP

yet non-existent man-page to exactly know which parameters a class has and in what order they are supposed to be designed. A UI would make this a lot easier. However, the `age` attribute for example is not useful in such a system, since the value will inevitably change over time. To keep it maintainable, a `date-of-birth` attribute would be more appropriate.

Extensibility As CLOS makes it easy to add new classes and combine them, new types of occupations within an organizational structure can be easily added. The concept of inheritance in object-oriented languages facilitates especially. The method combination furthermore facilitated the creation of human readable output functions, which can be concatenated for any additional attribute of a new class.

Accessibility Since CLOS is standardized, any Lisp program can make use of the proposed framework. Objects can be reused in other programs and though the help of object-relational mappers, other non-Lisp programs can also access the information within the system.

Our solution shows that CLOS as example of Object Orientation lends itself very well to the description of real world states-of-affairs. It is a powerful extension of the Lisp programming language and gave a good impression of what was to expect from object-oriented languages to come.

```
[1]> (defvar *Max* (make-student "Max Mustermann" 22 "Beispiel Strasse 1" 21222011 0))
*MAX*
[2]> (get-info *Max*)
Name: Max Mustermann
 Occupation: #<STANDARD-CLASS STUDENT>
 Age: 22
 Adress: Beispiel Straße 1
 Date of Registration: 21222011
 ECTS Points: 0

NIL
[3]> (defvar *Maria* (make-professor-chairman "Maria Musterfrau" 52 "Schoene Strasse 2" 21222010 "IBAN:1234" "Software Architecture" "50%"))
*MARIA*
[4]> (get-info *Maria*)
Name: Maria Musterfrau
 Occupation: #<STANDARD-CLASS PROFESSOR-CHAIRMAN>
 Age: 52
 Adress: Schöne Straße 2
 Date of Registration: 21222010
 Bank Details: IBAN:1234
 Share: 50%
 Chair: Software Architecture

NIL
```

Fig. 4. A screenshot of a possible interaction with the framework provided by our solution

6 Related Work

Today, Object Orientation is a very popular programming concept. Therefore, a plethora of different object-oriented languages exist which could solve the presented problem in their own way. In this Section we will discuss the example of Java, a particularly popular object-oriented programming language.

```
public interface ProfessorInterface {
}

public class Professor implements ProfessorInterface{
}

public interface ChairmanInterface {
}

public class ProfessorChairman implements ProfessorInterface, ChairmanInterface{
}
```

Listing 6: An example of multiple inheritance using Java interfaces

Java does not have multiple inheritance integrated as directly as CLOS does, meaning that because of the diamond problem unwanted behavior can arise during compilation[11]. The combination of classes would thus need to be solved in a different manner. Instead, Java *Interfaces* are recommended to be used to achieve a similar effect. An interface is a sort of protocol that can be used by

[11] `https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html`, accessed: 2016-07-28

classes. Such an interface could be used as an abstract class, and so multiple interfaces could be implemented for a class that is a combination of others.

However, such a solution can be a lot less intuitive for the programmer trying to extend the system, since adding a new occupation to the system would require to create a new class and a interface. Furthermore, it could be unclear what sort of behavior would have to be defined in the either the class or the interface. The elegant solution of method combination in CLOS would also not be present in Java.

7 Conclusion

In this work we have discussed the object-oriented Lisp system CLOS by applying it to the problem of member registers of organizations, based on the example of a university.

First, an introduction to the concepts behind CLOS was given, as well as a description of its history, and its three main ideas, namely *multiple inheritance*, *method combination*, and *generic functions*, were presented. Based on the qualities of *maintainability*, *extensibility*, and *accessibility*, the solution was evaluated and the example of Java as alternative tool was discussed.

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] Wegner, P.: Dimensions of object-based language design. Volume 22 of OOPLSA '87., New York, NY, USA, ACM (1987)

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

[6] 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

[7] 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

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

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

[10] Cargill, T.A.: The case against multiple inheritance in c++. In: The evolution of C++, MIT Press (1993) 101–109

APPENDIX

```
(defclass person ()
  ((name :accessor name
         :initarg :name)
   (age :accessor age
        :initarg :age)
   (address :accessor address
         :initarg :address)
   (registration-date :accessor registration-date
         :initarg :registration-date)))

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

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

(defclass professor (staffmember)
  ((chair :accessor chair
            :initarg :chair)))

(defclass chairman (staffmember)
  ((share :accessor share
            :initarg :share)))

(defclass professor-chairman (professor chairman) () )
```

Listing 7: The implementation of the classes

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

(defun make-professor (name age address registration-date bank-details chair)
  (make-instance 'professor :name name :age age :address address
    :registration-date registration-date :bank-details bank-details :chair chair))

(defun make-chairman (name age address registration-date bank-details share)
  (make-instance 'chairman :name name :age age :address address
    :registration-date registration-date :bank-details bank-details :share share))

(defun make-professor-chairman (name age address registration-date bank-details chair share)
  (make-instance 'professor-chairman :name name :age age :address address
    :registration-date registration-date :bank-details bank-details :chair chair :share share))
```

Listing 8: The instantiation functions

```
(defgeneric get-info (person)
      (:documentation "Returns data on all attributes."))

(defmethod get-info ((p person))
    (format t "Name: ~d ~% Occupation: ~d ~% Age: ~d ~%
     Address: ~d ~% Date of Registration: ~d ~% "
     (name p) (class-of p) (age p) (address p) (registration-date p)))

(defmethod get-info :after ((s student))
    (format t "ECTS Points: ~d ~% " (ects-points s)))

(defmethod get-info :after ((s staffmember))
    (format t "Bank Details: ~d ~% " (bank-details s)))

(defmethod get-info :after ((p professor))
    (format t "Chair: ~d ~% " (chair p)))

(defmethod get-info :after ((c chairman))
    (format t "Share: ~d ~% " (share c)))
```

Listing 9: The implementation of the human readable interface

```
(defvar *Max*
   (make-student "Max Mustermann" 22 "Beispiel Strasse 1" 21222011 0))

(defvar *Maria*
   (make-professor-chairman "Maria Musterfrau" 52 "Schoene Strasse 2"
    21222010 "IBAN:1234" "Software Architecture" "50%"))

(get-info *Max*)
(format t "~%--------------~%~%")
(get-info *Maria*)
```

Listing 10: Examples to test the framework