Comparison Object Oriented Programming in Objective-C with GO

Thomas Martin Randl

Fakultät für Informatik

WS 2020/21

Abstract

ToDo

Keywords: Go, Objective-C, Object Oriented Programming

Table of Contents

List of Abbreviations

Go	Programming Language Go
OOP	Object Oriented Programming

1. Introduction

2. History of OOP

The first approach for Object Oriented Programming (OOP) was more than 40 years ago with the programming language Simula 67. However the first language to be really successful with this new approach in programming was Smalltalk. These languages fullfilled the basics needed for OOP. With a slow and steady development OOP became a more and more used concept in new developed programming languages [KOP09].

Since the teaching of programming languages like Java took place in a wide range of university classes, OOP became more and more relevant to companies. This lies in the fact that graduates from all over the world tend to know how to use at least one OOP language. The fact that more and more companies made use of OOP languages in their software solutions pushed their popularity even further.

Nowadays OOP is the most spread concept between all programming languages. Besides their teaching all over the globe their popularity can be derived from the fact that problems from the real world can easily be transfered into the programming world. It is simple to imagine a button in a graphical interface to be an object with certain properties. Even non OOP languages nowadays tend to use object similar approaches as you can see with the programming language Go in Chapter 5. Object Oriented Programming in Go [KOP09]. Sometimes this works out well, but the OOP approach turns out to be the better way in most of this cases.

3. Object Oriented Programming

As we learned in chapter 2. History of OOP, OOP is a widely spread concept in modern programming languages. It takes proceedings from the real world into a programming context. This is because of the fact that you can declare every instance from the real world as an object. These objects on the other hand can be put in context and interact with each other. These traits of OOP are called classification and inheritance. With that in mind every object needs his own class with its own methods to interact with each other [KOP09]. In the following chapter we will learn how a class in OOP is structured and what base concepts OOP uses to set classes in context with each other and how they are visibile outside of the object. The extend to which a programming language uses these concepts defines how much of an OOP language it is.

3.1 Classification

Classification is a technique used to classify objects due to their skills. During the process objects are structured (chapter 3.3 Inheritance) and the skills are assigned to the objects class.

A class describes and implements a new type of object. Objects in OOP are derived from that class. Every class consists of several attributes and methods. An object kann be created using the class as kind of a datatype with parameters. The added parameters are assigned to the attributes in the constructor of the class. The created object interacts independently from other objects with the environment using the methods provided by the base class.

The methods of a class are declared in an interface. This can be used by other classes to interact with the classes objects using these methods [KOP09].

3.2 Encapsulation

Encapsulation is used for hiding the objects information from the environent. It is uesd to protect the attributes values of the object from manipulation. This is done by declaring the attributes visibility. If an attribute is declared as not visible from outside the object it can only be accessed by using methods declared in the inteface of the objects class. Methods can also be declared in different visibility states depending on the programming language.

An encapsulated object can only be interacted with via the methods declared in the interface. Every attribute or method not mentioned in the interface is neither accessible nor visible from outside [KOP09].

3.3 Inheritance

As in chapter 3.1 Classification mentioned, Objects in OOP are structured by their skills and abilities. In OOP every Object is derived from a parent object. The resulting child object is able to use the functionality of the parent and its own. This property is called inheritance.

With the use of inheritance it is possible to reuse the previous defined structures or to make use of an abstract object which can be specified in the child object. This helps the programmer to reduce the complexitiy and effort of the programm.

Especially the use of an abstract class can be very useful. This is the case if a program uses different object that contain several equal attributes. In this case the abstract class defines the base attributes and the child objects inherit them and implement their functionality for them [KOP09].

3.4 Polymorphism

A synonym for polimorphism is diversity. Taking this into the context of programming languages, polymorphism is the approach to accept and return values of more than one datatype.

OOP uses this functionality with its inheritance (chapter 3.3 Inheritance). This allows the use of different objects as parameters and/or return values.

3.5 Persistence

Persistence stands for the lifetime an object exists in the program after it is created. There are some different approaches depending on the used programming language. While in C++ the user is responsible for deleting the created objects after their use expired, languages like Java use a so called "garbage collector".

Having this in mind, languages with a garbage collector are for no use in safety related software, because a fast reaction to a problem can not be guaranteeed if the garbage collector interrupts the programm at the exact moment of a emergency. An automatic memory management on the other hand is much less bug prone due to the lower complexity [KOP09].

4. Object Oriented Programming in Objective-C

5. Object Oriented Programming in Go

The programming language Go was introduced by Google in 2009. It has been develoed since by a team at google and a lot of other contributors from the open source community. The BSD style license it was released with allowed the community the further development to this day .

Its initial cause was to create a language that is more accessible and save than C/C++ in terms of syntax, compile time and functionality. The focus was to develop an easier solution for scalable network services and cloud computing. Whilst Go differs in many ways from the C programming language, its roots with this language are preserved in the fact, that it still uses C like pointers. But they do not support pointer arithmetic which is because of the fact, that Go puts his focus on fast compiling [GOL20].

Go does a balancing act between velocity and accessibility. Its purpose is to deliver a solution which is faster than competing languages like python [WSP20] and more accessible than the really fast

languages C and C++. Sadly Go does not fit well with GUI development or the development of embedded systems [COP20].

In terms of OOP even Google is not sure whether or not Go is an OOP language. The total abscence of some features, which are discussed later in this chapter, could lead to the conclussion that Go simply is no OOP language. But at least it can be argued, that Go allows an OOP like style of programming [IGO18]. The degree to which Go differs from classic programming languages like C++ or Java will be discussed in the following chapters.

5.1 Classification with Go

Go does not provide a classic syntax for creating a class. Go does not provide classes. To achive a classification Go uses structs similar to structs in the C programming language. The following code snipped shows how a class is implemented in Go.

Example for a class in Go

```
1 # This is the type containing the attributes of the class ClassA
2 type ClassA struct {
3     color color.Color
4     name  string
5 }
6
7 # This is a method of class ClassA
8 func (c ClassA) SayName() {
9     fmt.Println(name);
10 }
```

This states out that a struct is a user defined type that can hold a list of attributes. In combination with functions using the struct as base, as shown in the snippet, Go is able to offer similar functionality than other languages using classes [COP20].

5.2 Encapsulation with Go

Classic OOP languages use keywords like "protected", "private" and "public" to encapsulate attributes and methods of their classes. Go does offer a different approach.

Go encapsulates on package level by differentiating between lower or upper case on the first letter of the method, type or interface name.

Encapsulating possibilities in Go

```
1 # This is a public struct due to the capital letter
2 type ClassA struct {}
3
4 #This is a private struct due to the lower case letter
5 type classB struct { x, y float64 }
6
7 #This is a private method due ot the lower case letter
8 func (c *ClassA) string() string {
9     return fmt.Println("I can only be called inside my package")
10 }
11
12 # This is a public method due to the capital letter
13 func (c *ClassA) Draw() {
14     fmt.Println("I can be called from outside my package")
15 }
```

As is seen in the above code snippet one can make some methods of a struct private and others public. Even structs can be declared private. This allows the base concept of encapsulation even if it is not as convenient like in other OOP languages as mentioned before. Calling the encapsulating functions public or private is not completely correct, since it is more of a controll mechanism to allow the export of a function, type or interface [IGO18].

If you declare a method with a parameter in front of the method name you can conveniently call it like for example "myStruct.myFunction()". The Go compiler compiles it like a normal parameter after the function name anyway. So this is just syntactic sugar which gives you the illusion to write and call a real OOP method [IGO18].

5.3 Inheritance with Go

The Go programming language does not implement inheritance in any way. This beeing said Go has is own way to map relationships between types. Therefore it uses composition instead of inheritance.

Languages like Java and C++ require the programmer to know the relationships of his objects at first implementation. Additonaly multiple inheritance can get very complex very fast. Therefore the developers of Go made the choice to implement a lightweight alternative, which allows the implementation of relationships between types [FAQ20].

Inheritance alternative approach in Go

```
# Class A
type ClassA struct { x, y float64 }

# Class B extends ClassA
type ClassB struct {
    c     ClassA  # ClassB embedding a classA instance extends ClassA with the
    num   int     # attributes of ClassB
}
```

That beeing said we do not speak so much of ClassA inherits from ClassB as ClassA extends ClassB by embedding a ClassB instance in the struct of ClassA. Go also supports overriding functions. Overloading a function like in Java however is not possible with Go [COP20].

5.4 Polymorphism with Go

Since Go does not support inheritance but follows the principle of composition there is a different approach to polymorphism than usual. To achieve the treatment of different objects uniformely it implements interfaces implicitly [IGO18]. This leads to a very intuitive approach as can be seen in the following code snipped.

Inheritance alternative approach in Go

```
# Interface for the ploymorphism methods
type getAttributes interface {
    getArea() float64
}

type square struct {
    length float64
}

type rectangle struct {
    width  float64
    height float64
}
# The implementation of getArea() for square
func (s square) getArea() float64 {
    return s.length * 2
}
# The implementation of getArea() for rectangle
func (r rectangle) getArea() float64 {
    return r.width * r.height
}
# The function to call to get the output for all objects with a getArea() method
func calcArea(a getAttributes) {
    fmt.Println("The area of the attribute is:", a.getArea())
}

func main() {
    s := square{length: 4}
    r := rectangle{width: 2, height: 3}
    calcArea(s) # The area of the attribute is: 8
    calcArea(r) # The area of the attribute is: 6
}
```

In this example code there are two structs that implenent the same function getArea(). Both have different approaches since rectangle and square areas are calculated differently. The interface getAttributes is basically a named collection for all method signatures defined by the interface. This interface allows the call of methods contained by it and therefore we can call the calcArea function on square and rectangle and the respective methods are called. This means that Go delivers a very intuitive way of polymorphism.

5.5 Persistence with Go

As mentioned in chapter 3.5 Persistence OOP languages often use a garbage collector, which is responsible for deleting unused objects. Go also makes use of such a garbage collection. This results in the fact that the programmer does not need to care about deleting objects. On the other hand is

Go no fit for programming a system with high safety requirements.

6. Differences between Go and Objective-C regarding OOP

6. Summary

References

COP20 Johannes Weigend, Concepts of Programming Languages, called 2020-12-03, https://github.com/jweigend/concepts-of-programming-languages

DEP15 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software, 2015, Mitp

FAQ20 Website, golang.org, called 2020-12-03, https://golang.org/doc/faq#Is_Go_an_object-oriented_language

GOL20 Website, golang.org, called 2020-12-03, https://golang.org/

KOP09 Arnd Poetzsch-Heffter, Konzepte objektorientierter Programmierung, Rev. 2, 2009, Springer

IGO18 Website, medium.com, called on 2019-12-03, https://medium.com/gophersland/gopher-vs-object-oriented-golang-4fa62b88c701

OC209 Stephen G. Kochan, Objective-C 2.0, 2009, Addison-Wesley

OOT15 Suad Alagic, Object-Oriented Technology, 2015, Springer

OOA15 Brahma Dathan, Sarnath Ramnath, Object-Oriented Analysis, Design and Implementation - An Integrated Approach, Rev. 2, 2015, Springer

POL20 Website, duden.de, called 2020-12-02, https://www.duden.de/rechtschreibung/Polymorphismus

WSP20 Lucas Lukac, getstream.io, called 2020-12-03, https://getstream.io/blog/switched-python-go/

List of Figures