Compare Object Oriented Programming in Go with Eiffel - Concepts of Programming Languages WS 18/19

Johann Neuhauser

Table of Contents

Introduction

Go programming language

Go is a compilable programming language that supports concurrency and has automatic garbage collection. Go was initially designed by Robert Griesemer, Rob Pike and Ken Thompson and is developed by employees of Google Inc.

Go was developed out of dissatisfaction with existing software development languages such as C, C++ or Java in the context of today's computer systems, especially with regard to scalable network services, cluster and cloud computing. Go has far fewer keywords than ANSI C (25 in Go vs. 32 in C). One of the problems Go wants to solve is very long compile times in large C++ projects.

Key characteristics

- Very fast compilation times
- UTF-8 only strings
- Pointer arithmetics
- Garbage collection
- Static type system with runtime support
- Static linking
- Package system
- Designed for concurrency with goroutines and channels

Hello World

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

Eiffel programming language

Eiffel is a compilable, universal and object oriented programming language designed by Dr. Bertrand Meyer and his company Interactive Software Engineering Inc. as alternative to C++. It was originally conceived in 1985 only for use by it´s own company, but after attracting considerable public attention in early 1986, it was released in late 1986 as a commercial product.

Eiffel is based on a set of principles as design by contract, the uniform access principle, command–query separation, the don't repeat yourself principle, the open/closed principle, and

option–operand separation. Many concepts introduced in Eiffel by Dr. Bertrand Meyer later found their way into Java, C# and other popular programming languages.

Key characteristics

- Design by contract tightly integrated with other language constructs
- Garbage collection
- Inheritance, including multiple inheritance, renaming, redefinition and other mechanisms intended to make inheritance safe
- All types, including basic types such as INTEGER, are class based
- Static type system
- Void safety against calls on null references through the attached-types mechanism
- Once routines for object sharing and decentralized initialization

Hello World

```
class
    HELLO_WORLD
create
    make
feature
    make
        do
            print ("Hello, world!%N")
        end
end
```

Class model

In terms of type theory, a class is an implementation of a concrete data structure and collection of related programs. The Go and Eiffel programming language are very different when it comes to classes. The following chapter tries to point out the main characteristics of the class model in Go and Eiffel. The most important is explained briefly in words and then shown by a simple data class, called "person", which holds the name as string and the age as a integer. In addition to show the usage of the class, a small example is implemented.

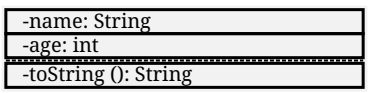

Figure 1. UML representation of a class

Output of example code

```
Max Mustermann (33 years)
Max Mustermann (34 years)
```

Go programming language

Go doesn't require an explicit class definition, instead a "class" is implicitly defined by providing a set of "methods" which operate on a common type. The type may be a struct or any other user-defined type. Therefore Go also doesn't require a constructor, setter or accessor for a "data class".

main.go (data class and usage)

```
package main

import "fmt"

type Person struct { ①
    name string ②
    age int
}

func (p Person) String() string { ③
    return fmt.Sprintf("%s (%d years)", p.name, p.age)
}

func main() {
    p := Person{"Max Mustermann", 33} ④
    fmt.Println(p) ⑤
    p.age = 34 ⑥
    fmt.Println(p)
}
```

① Define the public "data class" / user type **Person**

② Define attribute **name** with type **string**

③ Implement the Stringer interface for the user type **Person** for textual type representation

④ Create a **Person** object **p** without a explicit constructor

⑤ Print textual representation of **Person** object through Stringer interface

⑥ Assign new value to **Person** attribute **age** without a explicit setter procedure

Eiffel programming language

Eiffel is a class only based programming language in which everything has to be wrapped in a class. An Eiffel class consists of some headers, optional features and an end delimiter. Every class in Eiffel inherits implicit from the ANY class which is very similar to the Object class in Java.

Headers

The headers define the type of the class and optionally the explicit inheritance with the keyword "inherit" and the constructor routines with the keyword "create". The class type "class" is a normal class from which an object can be created. The "deferred class" is an abstract class from which no object can be created and contains at least one signature of an unimplemented routine.

Features

The definition of attributes and routines are wrapped in feature clauses. Every class can have multiple feature clauses. The designer of a class controls which clients are allowed to call their features. Feature declarations are organized into groups according to their visibility. Each group has a header that lists the classes in curly braces that can use the following features. A client can call a feature only if the client belongs to one of the classes in the list. If the header does not contain a list, any client can call the features. If the list is empty or contains the word NONE, the features are for internal use only.

Attributes

The definition of attributes requires only a name, followed by a colon and a class/type name. If attributes are visible to clients, they can only be accessed and not assigned from outside the class by default. To make it possible to assign values from outside the class, there must be a setter procedure that has one argument of the assigned type and is assigned to the attribute with the keyword "assign".

Routines

The signature of routines is almost similar to those of the attributes, but they have a argument list in parentheses and a body introduced by "do" and delimited by "end" or instead are marked with deferred as unimplemented. The list of arguments and the return type of routines are optional and can be omitted. If a return type is defined, there is always a routine local attribute which holds the return value. The value returned by a function is the value that is in "Result" after the function ends. "Result" is automatically declared and initialized at the beginning of the function and can be changed as often as required.

person.e (data class)

```
class PERSON ①
inherit
    ANY
        redefine
            out ②
        end
create
    make ③
feature {NONE} ④
    make (a_name: STRING; a_age: INTEGER)
        do
            set_name (a_name)
            set_age (a_age)
        end
feature
    name: STRING assign set_name ⑤
    age: INTEGER assign set_age
    set_name (a_name: STRING) ⑥
        do
            name := a_name
        end
    set_age (a_age: INTEGER)
        do
            age := a_age
        end
    out: STRING ⑦
        do
            Result := name.out + " (" + age.out + " years)"
        end
invariant
    non_negative_age: age >= 0 ⑧
end
```

① Define the custom data class **PERSON**

② Announce that the **out** function of **ANY** is overridden by **PERSON**

③ Define the **make** procedure as a constructor of **PERSON**

④ Constructors should be private in Eiffel because they are only needed for object construction

⑤ Define attribute **name** with type **STRING** and assign the setter procedure **set_name**

⑥ Define procedure **set_name** with the argument **a_name** with type **STRING**

⑦ Redefine function **out** with return value type **STRING** which produces a textual representation of a object

⑧ Ensure that attribute **age** becomes never a negative number (class global assertion)

application.e (usage)

```
class
    APPLICATION
create
    make
feature {NONE}
    make
        local
            p: PERSON ①
        do
            create p.make ("Max Mustermann", 33) ②
            print (p.out + "%N") ③
            p.age := 34 ④
            print (p.out + "%N")
        end
end
```

① Declaration of local variable **p** of type **PERSON**

② Create a **PERSON** object **p** with constructor **make**

③ Print textual representation of **PERSON** object **p** with **out** function

④ Assign new value to **PERSON** attribute **age** through assigned setter **set_age**

Composition

This chapter discusses the object-oriented programming principle of "object composition", where a data type is composed of other data types and is also referred as a "has a" relationship. This relationship of different data types is explained with the data class "Customer" which includes the data type "Address".

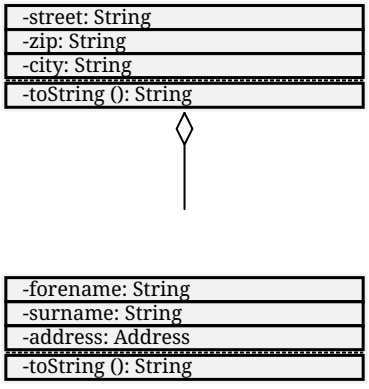

Figure 2. UML representation of object composition

Output of example code

```
Max Mustermann, Musterstrasse 1, 12345 Musterstadt
```

Go programming language

Composition in Go can be achieved by embedding one struct into another struct with a chosen name. Beside the "has one" relationship, it is also possible to implement a very simple "has more" relation with a slice which is a self extending implementation of an array. For more complex scenarios, where a simple slice is not enough, there are lists or maps in the Go libraries.

address.go (data class)

```
package main

import "fmt"

type Address struct {
    street string
    zip string
    city string
}

func (a Address) String() string {
    return fmt.Sprintf("%s, %s %s", a.street, a.zip, a.city)
}
```

customer.go (composed data class)

```
package main

import "fmt"

type Customer struct {
    forename string
    surname string
    address Address ①
}

func (c Customer) String() string {
    return fmt.Sprintf("%s %s, %s", c.forename, c.surname, c.address) ②
}
```

① Embed data type **Address** as **address** in data type **Customer**

② Generate textual representation of **Customer** with the use of the **String()** method of **Address**

main.go (usage)

```
package main

import "fmt"

func main() {
    my_address := Address{"Musterstraße 1", "12345", "Musterstadt"}
    my_customer := Customer{"Max", "Mustermann", my_address}
    fmt.Println(my_customer)
}
```

Eiffel programming language

In the Eiffel programming language, composition is achieved by adding an attribute with the desired data class as the type. Because all data types in Eiffel are class-based, each new data class with at least one attribute is a compound data type. To achieve a "has more" relationship, it is necessary to use a list, set, or map implementation which are provided by the rich Eiffel libraries.

address.e (data class)

```
class ADDRESS
inherit
    ANY
        redefine
            out
        end
create
    make
feature {NONE}
    make(a_street, a_zip, a_city: STRING)
        do
            set_street (a_street)
            set_zip (a_zip)
            set_city (a_city)
        end
feature
    street: STRING assign set_street
    zip: STRING assign set_zip
    city: STRING assign set_city
    set_street (a_street: STRING)
        do
            street := a_street
        end
    set_zip (a_zip: STRING)
        do
            zip := a_zip
        end
    set_city (a_city: STRING)
        do
            city := a_city
        end
    out: STRING
        do
            Result := street.out + ", " + zip.out + " " + city.out
        end
end
```

customer.e (composed data class)

```
class CUSTOMER
inherit
    ANY
        redefine
            out
        end
create
    make
feature {NONE}
    make (a_forename, a_surname: STRING; a_address: ADDRESS)
        do
            set_forename (a_forename)
            set_surname (a_surname)
            set_address (a_address)
        end
feature
    forename: STRING assign set_forename
    surname: STRING assign set_surname
    address: ADDRESS assign set_address ①
    set_forename (a_forename: STRING)
        do
            forename := a_forename
        end
    set_surname (a_surname: STRING)
        do
            surname := a_surname
        end
    set_address (a_address: ADDRESS)
        do
            address := a_address
        end
    out: STRING
        do
            Result := forename.out + " " + surname.out + ", " + address.out ②
        end
end
```

① Use data type **ADDRESS** as attribute **address** in data type **CUSTOMER**

② Generate textual representation of **CUSTOMER** with the use of the **out** function of **ADDRESS**

application.e (usage)

```
class
    APPLICATION
create
    make
feature {NONE}
    make
        local
            my_customer: CUSTOMER
            my_address: ADDRESS
        do
            create my_address.make ("Musterstrasse 1", "12345", "Musterstadt")
            create my_customer.make ("Max", "Mustermann", my_address)
            print (my_customer.out + "%N")
        end
end
```

Inheritance

Inheritance in object-oriented programming is the principle of class hierarchy. It is the ability of an object or class to inherit the attributes, behavior, and functionality of another object or class. Parent classes can have child classes which can have the same properties of the parent classes and can define new attributes, behaviors and functionality of their own. Additionally it's possible that child classes override functionality or behavior of their parent classes. The relationship between a parent and a child class is often described as a "is a" relationship. This chapter uses the data class "Employee" which is inherited from the data class "Person" with an additional employee number attached.

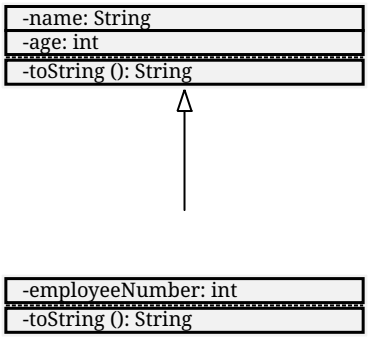

Figure 3. UML representation of class inheritance

Output of example code

```
12346: Thomas Bauer (25 years)
12346: Thomas Huber (25 years)
```

Go programming language

person.go (data class)

```
package main

import "fmt"

type Person struct {
    name string
    age  int
}

func (p Person) String() string {
    return fmt.Sprintf("%s (%d years)", p.name, p.age)
}
```

employee.go (inherited data class)

```
package main

import "fmt"

type Employee struct {
    Person ①
    employeeNumber int
}

func (e Employee) String() string {
    return fmt.Sprintf("%d: %s", e.employeeNumber, e.Person) ②
}
```

① Inherit all properties from **Person** for "class" **Employee**

② Generate textual representation of **Employee** with the use of the **String()** method of **Person**

main.go (specialized data class)

```
package main

import "fmt"

func main() {
    e := Employee{Person{"Thomas Bauer", 25}, 12346} ①
    fmt.Println(e)
    e.name = "Thomas Huber" ②
    fmt.Println(e)
}
```

① Create a **Employee** "object" with a embedded **Person** "object"

② Assign another name to the **Employee** "object"

Eiffel programming language

person.e (data class)

```
class PERSON
inherit
    ANY
        redefine
            out
        end
create
    make
feature {NONE}
    make (a_name: STRING; a_age: INTEGER)
        do
            set_name (a_name)
            set_age (a_age)
        end
feature
    name: STRING assign set_name
    age: INTEGER assign set_age
    set_name (a_name: STRING)
        do
            name := a_name
        end
    set_age (a_age: INTEGER)
        do
            age := a_age
        end
    out: STRING
        do
            Result := name.out + " (" + age.out + " years)"
        end
invariant
    non_negative_age: age >= 0
end
```

employee.e (inherited data class)

```
class
    EMPLOYEE
inherit
    PERSON ①
        rename ②
            make as person_make,
            out as person_out
        end
create
    make
feature {NONE}
    make (a_name: STRING; a_age, a_employee_number: INTEGER)
        do
            person_make (a_name, a_age) ③
            set_employee_number (a_employee_number)
        end
feature
    employee_number: INTEGER assign set_employee_number
    set_employee_number (a_employee_number: INTEGER)
        do
            employee_number := a_employee_number
        end
    out: STRING
        do
            Result := employee_number.out + ": " + person_out ④
        end
end
```

① Inherit everything from class **PERSON** for new class **EMPLOYEE**

② Rename inherited class routines **make** and **out** as we define them new in class **EMPLOYEE** and want to use the inherited routines internally in the new class

③ Use renamed constructor **make** of class **PERSON** in constructor **make** of class **EMPLOYEE**

④ Generate textual representation of **EMPLOYEE** with the use of the renamed **out** function of **PERSON**

application.e (usage)

```
class
    APPLICATION
create
    make
feature {NONE}
    make
        local
            e: EMPLOYEE
        do
            create e.make ("Thomas Bauer", 25, 12346) ①
            print (e.out + "%N")
            e.name := "Thomas Huber" ②
            print (e.out + "%N")
        end
end
```

① Create a object **e** of class **EMPLOYEE** which is inherited from class **PERSON**

② Assign another name to the **EMPLOYEE** object

Polymorphism

The polymorphism of object-oriented programming is a property that always occurs in connection with inheritance and interfaces. A routine is polymorphic if it has the same signature in different classes but is re-implemented.

This chapter uses a class point and a class circle as an example to demonstrate the needed steps to implement polymorphism in the particular programming language. The circle class inherits all properties from the point class and adds only the required radius. In addition, we would like to check the radius at the time of object construction.

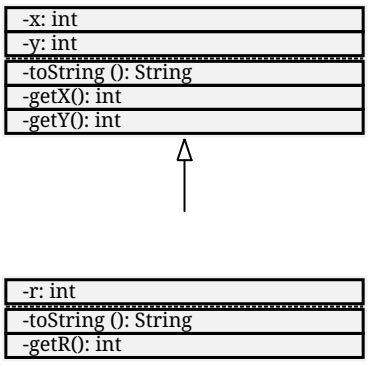

Figure 4. UML representation of polymorphism

Output of example code

```
Point:  x = 0   y = 0
Circle: x = 0   y = 0   r = 0
Point:  x = 10  y = 15
Circle: x = 20  y = 25  r = 5
Circle: x = 30  y = 35  r = 10
Circle: x = 20  y = 25  r = 35
```

Go programming language

Non default object construction

To support checks at "object" construction time for arguments or the construction of objects apart from their default initialized values, it’s required to explicit define constructors for our user defined data type. Because Go uses struct’s for their "data classes", it’s not possible to prohibit the "object" construction without a defined constructor.

Polymorphic operations

For polymorphic operations on different data types, generic methods for the polymorphic operations must be defined via an interface. It’s not possible that we create a circle object on a point variable as in other object-oriented programming languages. We have to use a user defined interface type as variable which can hold all types which implements this specific interface.

point.go

```
package main

import "fmt"

type Point struct {
    x, y int
}

type IPoint interface { ①
    String() string
    GetX() int
    GetY() int
}

func (p *Point) String() string { ②
    return fmt.Sprintf("Point:\tx = %d\ty = %d", p.x, p.y)
}

func (p *Point) GetX() int { ②
    return p.x
}

func (p *Point) GetY() int { ②
    return p.y
}

func NewPoint(x, y int) *Point { ③
    return &Point{x, y}
}

func NewPointOrigin() *Point { ③
    return &Point{}
}
```

① Generic interface which declares the needed methods that must be implemented for a data type to be compatible with the **IPoint** interface

② Needed method to cope with the **IPoint** interface requirements

③ Very simple constructor that is only implemented for demonstration

circle.go

```
package main

import "fmt"

type Circle struct {
    Point
    r int
}

type ICircle interface {
    IPoint ①
    GetR() int ②
}

func (c *Circle) String() string { ③
    return fmt.Sprintf("Circle:\tx = %d\ty = %d\tr = %d", c.x, c.y, c.r)
}

func (c *Circle) GetR() int { ④
    return c.r
}

func NewCircle(x, y, r int) *Circle { ⑤
    if r < 0 {
        panic("Negative radius for circle not allowed")
    }
    return &Circle{Point{x, y}, r}
}

func NewCircleOrigin() *Circle {
    return &Circle{}
}

func NewCircleFromPoint(g *IPoint, r int) *Circle { ⑥
    if r < 0 {
        panic("Negative radius for circle not allowed")
    }
    return &Circle{Point{(*g).GetX(), (*g).GetY()}, r}
}
```

① Inherit all interface requirements from **IPoint**

② Extend **IPoint** interface requirements for **ICircle**

③ Override String() method from "class" **Point**

④ Needed method to cope with the **ICircle** interface requirements

⑤ Very simple constructor that checks the radius argument

⑥ Constructor that creates a **Circle** "object" from a "object" that implements the **IPoint** interface

main.go

```
package main

import "fmt"

func main() {
    var myPoint IPoint
    var myCircle ICircle

    myPoint = NewPointOrigin() ①
    fmt.Println(myPoint)

    myPoint = NewCircleOrigin() ②
    fmt.Println(myPoint)

    myPoint = NewPoint(10, 15) ③
    fmt.Println(myPoint)

    myPoint = NewCircle(20, 25, 5) ④
    fmt.Println(myPoint)

    myCircle = NewCircle(30, 35, 10) ⑤
    fmt.Println(myCircle)

    myCircle = NewCircleFromPoint(&myPoint, 35) ⑥
    fmt.Println(myCircle)
}
```

① Create a Point "object" on a IPoint variable with constructor "NewPointOrigin"

② Create a Circle "object" on a IPoint variable with constructor "NewCircleOrigin"

③ Create a Point "object" on a IPoint variable with constructor "NewPoint"

④ Create a Circle "object" on a IPoint variable with constructor "NewCircle"

⑤ Create a Circle "object" on a ICircle variable with constructor "NewCircle"

⑥ Create a Circle "object" on a ICircle variable with constructor "NewCircleFromPoint" and last IPoint "object" which is a Circle as argument

Eiffel programming language

Eiffel supports the most popular object-oriented programming principles very well and therefore is the implementation of polymorphism a breeze without any restrictions. We can use a generic interface that is inherited from every class on which the polymorphic operation should be executed, or just a common base class.

point.e

```
class
    POINT
inherit
    ANY
        redefine
            out
        end
create
    make, make_origin
feature {NONE}
    make (a_x, a_y: INTEGER)
        do
            set_x (a_x)
            set_y (a_y)
        end
    make_origin
        do ①
        end
feature
    x: INTEGER assign set_x
    y: INTEGER assign set_y
    set_x (a_x: INTEGER)
        do
            x := a_x
        end
    set_y (a_y: INTEGER)
        do
            y := a_y
        end
    out: STRING
        do
            Result := "Point:%Tx = " + x.out + "%Ty = " + y.out
        end
end
```

① Nothing to do because INTEGER is automatically initialized with zero

circle.e

```
class
    CIRCLE
inherit
    POINT
        rename ①
            make as point_make
        redefine ②
            make_origin,
            out
        end
create
    make, make_origin, make_from_point
feature {NONE}
    make (a_x, a_y, a_r: INTEGER)
        require
            non_negative_radius_argument: a_r >= 0 ③
        do
            point_make (a_x, a_y) ④
            set_r (a_r)
        end
    make_origin
        do
        end
    make_from_point (a_p: POINT; a_r: INTEGER)
        require
            non_negative_radius_argument: a_r >= 0
        do
            set_x (a_p.x)
            set_y (a_p.y)
            set_r (a_r)
        end
feature
    r: INTEGER assign set_r
    set_r (a_r: INTEGER)
        require
            non_negative_radius_argument: a_r >= 0
        do
            r := a_r
        end
    out: STRING
        do
            Result := "Circle:%Tx = " + x.out + "%Ty = " + y.out + "%Tr = " + r.out
        end
invariant
    non_negative_radius: r >= 0 ⑤
end
```

① Rename inherited class routines **make** as we define it new in class **CIRCLE** and want to use the inherited routine internally in the new class

② Announce that we override the inherited class routines **make_origin** and **out** in our new class **CIRCLE**

③ Ensure that attribute **radius** becomes never a negative number (routine local assertion)

④ Use renamed constructor **make** of class **POINT** in constructor **make** of class **CIRCLE**

⑤ Ensure that attribute **radius** becomes never a negative number (class global assertion)

application.e

```
class
    APPLICATION
create
    make
feature {NONE}
    make
        local
            my_point: POINT
            my_circle: CIRCLE
        do
            create my_point.make_origin ①
            print (my_point.out + "%N")

            create {CIRCLE} my_point.make_origin ②
            print (my_point.out + "%N")

            create my_point.make (10, 15) ③
            print (my_point.out + "%N")

            create {CIRCLE} my_point.make (20, 25, 5) ④
            print (my_point.out + "%N")

            create my_circle.make (30, 35, 10) ⑤
            print (my_circle.out + "%N")

            create my_circle.make_from_point (my_point, 35) ⑥
            print (my_circle.out + "%N")
        end
end
```

① Create a POINT object on a POINT variable with constructor "make_origin"

② Create a CIRCLE object on a POINT variable with constructor "make_origin"

③ Create a POINT object on a POINT variable with constructor "make"

④ Create a CIRCLE object on a POINT variable with constructor "make"

⑤ Create a CIRCLE object on a CIRCLE variable with constructor "make"

⑥ Create a CIRCLE object on a CIRCLE variable with constructor "make_from_point" and last POINT object which is a CIRCLE as argument

Conclusion

In summary, it can be said that object-oriented programming is also possible in Go, but due to the lack of many concepts that improve the robustness of a program, it is more suitable for experienced programmers. On the other hand, Eiffel is a nice programming language to build safe and robust programs. But you quickly realize that a lot of training is needed here. In addition, the compile time already suffers from the complexity of the language even with simple programs.

References

- [edo] https://www.eiffel.org/documentation (2019-01-05)
- [gdo] https://golang.org/doc (2019-01-05)
- [spe] Simon Parker. Eiffel for beginners. 1999. https://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm (2019-01-05)
- [gpi] http://www.golangpatterns.info (2019-01-05)
- [rcw] http://rosettacode.org/wiki (2019-01-05)
- [wpe] https://en.wikipedia.org/wiki/Eiffel_(programming_language) (2019-01-05)
- [wpg] https://en.wikipedia.org/wiki/Go_(programming_language) (2019-01-05)