Object-Oriented Programming - A Short History

nprenet@bsu.edu

CS431, Spring 2022

Contents

Discussion: functional turtles

Week #6: Assignment vs. Eval contrasted two models of computations:

- the **evaluation model**, herited from Church's lambda-calculus, where computation is a top-down application of a function (Lisp, 1958 → Clojure, 2007)
- the **imperative model**, herited from Türing, where any computation can be modeled by a tape/register machine (Fortran, 1957 → Go, 2009)

Object-oriented (OO) programming pertains to the second model. This programming style is characterized by

- **encapsulation**
- **message passing**
- **dynamic binding** (the program's behavior may change at runtime)

To make clear what is at stake, consider this piece of Haskell code, that creates a `turtle` "object", with an initial position on a board:

```
turtle name (x,y) = \msg -> msg name (x,y)

Prelude> myTurtle = turtle "Clelia" (0,0)
Prelude> :t myTurtle
myTurtle :: (Num a, Num b) => ([Char] -> (a, b) -> t2) -> t2
```

Although `myTurtle` is a function, not a value, it stores some pieces of data, that we can retrieve, by sending the relevant message:

```
getName t = t (\ n (x,y) -> n)
getPosition t = t (\ n (x,y) -> (x,y))
```

```
Prelude> getName myTurtle
"Clelia"
Prelude> getPosition myTurtle
(0,0)
```

We can simulate the turtle's moves over the board:

```
Prelude> myTurtleAfterMove1 = move myTurtle (1,-1)   -- (1,-1)
Prelude> getName myTurtleAfterMove1
"Clelia"
Prelude> getPosition myTurtleAfterMove1
(1,-1)
```

Even though `myTurtleAfterMove1` shares its name property with `myTurtle`, it is a different object. Keeping track of state after a sequence of moves requires some bookkeeping:

```
Prelude> myTurtleAfterTwoMoreMoves= move (move myTurtleAfterMove1 (4,3)) (1,2)
Prelude> getPosition myTurtleAfterTwoMoreMoves
(5,2)
```

We are facing the limits of what we can do with pure functions. Although our functional turtle can receive messages and hide data, it has no ability to *change* its state: ultimately we have to share data through top-level definitions. There is no late-binding either, but note that those limitations are intrinsic to Haskell's design: a number of functional languages *do* provide assignment and late-binding (Lisp, Javascript, ...).

The Essence of OOP

The term "object-oriented programming" (OOP) was coined by Alan Kay in 1967 and inspired by

- Sutherland's Sketchpad application (1963): objects representing graphical images, with dynamic simulation obtained through encapsulation and inheritance
- Simula I (1965) and Simula 67: inheritance and virtual methods (to be defined later)

→ SmallTalk (Kay, Ingalls, Goldberg[1] 1972) embodies Kay's core idea: data sharing across the program is replaced by object interactions, where data structures and procedures are combined and encapsulated into little computers that communicate through messages. SmallTalk featured what Kay considered the defining characteristics of OO:

- encapsulating state by isolating other components from local state changes
- loosely-coupled components through messaging (instead of direct assignment)
- runtime-adaptability by late-binding, that allows for adding/replacing components at runtime

Notably, inheritance was not part of SmallTalk '72. This is worth an explanation, but we need first to illustrate the concepts just mentioned.

Encapsulation and message-passing

State encapsulation and message-passing go hand-in-hand, and may be implemented with any language that supports assignment, such as Javascript.

[1] Adele Goldberg, to be precise: she is worth a special mention, because women were far and few in the rarefied world of AI and language development.

Stateful turtle, the functional way

```
function makeTurtle ( name, x, y ){
   return function( msg ){
      switch (msg){
         case "walk":
            return (d) => {
                  let [xoff, yoff] = direction[d]
                  x += xoff
                  y += xoff
               }
         case "getName":
            return () => name // a function that returns the name
         case "getPosition":
            return () => [x,y] // a function that returns the position
         default:
            return undefined
      }
   }
}
```

```
> const t = makeTurtle("test1", 0, 0)
> t("walk")('South')
> t("getPosition")())
[0,1]
```

Comments:

- a **factory function** returns another function, not an object: the `dispatch` function *is* the component; however, while the Haskell turtles were pure functions, the dispatch shown here function captures its parameters into the local state, that can be mutated through assignment.
- parameters (`name`, `x`, `y`) are captured by the nested functions (closure), and can be mutated the same way as local variables ("fields")
- the dispatch function serves as a "gateway" to the component's inner data fields, that are otherwise hidden

Stateful turtle, with object

```
function makeTurtleObject ({name, x=0, y=0}={}){
   return {
      get name(){ return name }, // read accessor (without write counterpart)
      x,                         // <--> name cannot be changed
      y,
      walk: function(d){
         let [xoff, yoff] = direction[d]
         this.x += xoff
         this.y += yoff
      },
      getPosition: function(){ return [this.x, this.y] },
      getName: function(){ return this.name }
   }
}
```

Comments:

- the turtle factory function returns a Javascript `object` type, with explicit properties (`x`, `y`, `walk`, ...)

- properties may be implemented through static fields (`x`, `y`, ...) or accessors (`name()`), that allow for monitoring access to the inner data
- message passing made easier through methods and dot notation. E.g.:

```
> var  t = makeTurtleObject({name: "Clelia", x: 0, y: 0})
> t.walk( 'South' )
> t.getPosition()
[0,1]
> t.name = 'john' // no effect!
'john'
> [t.name, t.x, t.y]
['Clelia', 0, 1]
```

Stateful turtle, hybrid

```
function makeTurtleHybrid ({name, x=0, y=0}={}){
   return {
      walk: function(d){
         let [xoff, yoff] = direction[d]
         x += xoff
         y += yoff
      },
      getPosition: function(){ return [x, y] },
      getName: function(){ return name }
   }
}
```

Comments:

- the factory function returns an object that has explicit properties only for the procedures
- state is stored in the local variables captured by the closure, and can only be accessed through the appropriate methods. E.g.

```
> var  t = makeTurtleHybrid({name: "Clelia", x: 0, y: 0})
> t.walk( 'South' )
> t.getPosition()
[0,1]
> [t.name, t.x, t.y]
[undefined, undefined, undefined]
> t.getName()
'Clelia'
```

What we have learned about encapsulation

- No special treatment for function/objects/data.
- Objects and methods make encapsulation easier, but they are not essential.
- Classes and `new` are not needed, but factory functions definitively are, if we want to initialize several components from the same mold.

Late binding

The terms refers to the fact that the *concrete* type of an object may not be known until runtime:

- in languages with dynamic type checking, late binding is the rule (Python, Lisp, Javascript)
- in languages with static type checking (C#, Java):
  - the **declared type**, or interface, is known at compile time

  - the **concrete type**, or implementation, may not be known until runtime, thus allowing for substituting a component for another, if they both share their interface

```go
type Turtle interface {
    Move(d Direction)
}

type Walker struct {
    name string
    Position
}

type Flyer struct {
    name string
    fuel float32
    Position
}

func (t *Walker) Move(d Direction) {
    if d%2 == 0 { // limit to N,E,S,W
        offset := offsetFromDirection(d)
        t.x += offset.x
        t.y += offset.y
    }
}

func (t *Flyer) Move(d Direction) {
    offset := offsetFromDirection(d)
    t.x += offset.x
    t.y += offset.y
    t.fuel -= 0.5
}

func main() {

    var t1 Turtle
    t1 = &Walker{name: "test1"} // Position is automatically initialized by Go
    t1.Move(South)

    var t2 Turtle
    t2 = &Flyer{name: "test2", fuel: 2}
    t2.Move(Southwest)
}
```

What about inheritance?

Inheritance was very much part of OO early history: see Sketchpad, Simula 67, and even SmallTalk, that eventually introduced inheritance in 1976. All OO languages that came afterward include the notions of classes and subclasses: C++, Java, C#, Python, and Ruby, a direct descendant of SmallTalk. The concept has been popularized enough so that many programmers think it is *the* defining feature of OO programming. Let us do a brief review of the notion, using Python for brevity's sake.

Subclassing (a Python example)

```
class Turtle:

    def __init__(self, name, x=0, y=0):
        self.x = x
        self.y = y
        self.name = name

    def getPosition(self):
        return (self.x, self.y)

    def walk (self, d):
        if d in ['North', 'East', 'South', 'West']:
            xoff, yoff = direction[d]
            self.x += xoff
            self.y += yoff

    def __str__(self):
        return "name: {}, position: {}".format( self.name, self.getPosition())

class FlyingTurtle( Turtle ):

    def __init__(self, name, x=0, y=0, fuel=2):
        super().__init__(name, x, y)
        self.fuel = fuel

    def fly(self, d ):
        xoff, yoff = direction[d]
        self.x += xoff
        self.y += yoff
        self.fuel -= 0.5

    def getFuel(self):
        return self.fuel

    def __str__(self)
        return super().__str__() + ", fuel: {}".format(self.getFuel())
```

Comments:

- every object created by the `Turtle( ... )` constructor is an *instance* of the class, which is a *copy* of all characteristics described by the class:[2]

  ```
  >>> myTurtle = Turtle("Clelia")
  >>> myTurtle
  <Turtle object at 0x7fba95829700>
  >>> print(myTurtle)
  name: Clelia, position: (0, 0)
  ```

- in most OO languages, an object cannot modify its class, but in both SmallTalk and Ruby, classes *are* objects too, thus allowing for dynamic modification of classes.
- The `FlyingTurtle` class *extends* the `Turtle` class: this means that the `FlyingTurtle` class has its own *copy* of all properties defined in the `Turtle` class, as well as its own properties.

[2]A constructor is just a factory function (see above), usually with syntactic hints for the compiler (`new` in SmallTalk, Java, C#, Ruby); in Python, the constructor makes a backend call to the initialization function `__init__()`.

- A subclass may contain its own definition of methods that have already been defined by the superclass. E.g. A `FlyingTurtle` object has its own version of the `__str__()` function; it keeps a copy of the original method too, that can be referred to with `super()`.
- **OO polymorphism** has two aspects:
  - **virtual methods**, i.e. methods defined in a superclass that are designed to be overriden by subclasses; such methods are declared at compile time, but their concrete implementation may not be known until runtime: then the appropriate function is selected through *dynamic dispatch.*
  - any method can reference a method at a higher level of the inheritance hierarchy (e.g. `super`)

Inheritance issues

The benefits of inheritance seem obvious at face value:

- it is an easy and elegant way to avoid code duplication, when used appropriately
- it is one way to implement two symmetric aspects of **abtraction**: **generalization** and **specialization**. Dynamic dispatch allows for operating on objects based on what we know about their supertype, not their concrete type: it makes easier to break the code into modules.

But you need to consider the issues that come with it:

- inheritance relies on a domain model with *is-a* relations between entities. E.g. `FlyingTurtle` *is-a* `Turtle`, `Car` *is-a* `Vehicle`... More often than not, **inheritance breaks down in the real world**, where many concrete entities act in different capacities, depending on the context. E.g. A given student may be both a BSU `Student` and a BSU `Employee`;[3] multiple inheritance of concrete types is not the right answer: it comes with intractable problems,[4] which explain why most languages (Java, C#) disallow it.[5]
- **tight coupling**: all child classes depend on the *implementation* of the parent class; we avoid code duplication at the expense of a monolithic base class! A change in the base class may break all subclasses.
- if a use case does not fit the extension of an existing class, developers often rewrite and tweak some existing types, thus unwittingly duplicating code.
- even though the overuse of inheritance is mostly a beginner's habit, the added complexity that comes with big OO class hierarchies (Java APIs!) does not favor lean code; overengineered classes are hard to maintain; developers get into the habit of modeling every problem in terms of "things" (with categories and subcategories), when many problems have a more obvious, leaner, procedural solution.

Favor composition over inheritance

The book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, Vlissides, 1995) is a milestone of the OO literature, and was for a time required reading for any serious software engineering class in the country. It even earned a nickname (*Gang of Four*, or *GoF*), and, given its association with the OO-craze of the 1990s-2010s, as well as its generous use of class diagrams, we could almost forget that the authors were *not* advocating for more complexity.[6] Therefore, it is not coincidental that its contains the very answer to our problem:

> *Favor object composition over class inheritance.* (GoF, p. 20).

Most benefits of class inheritance can be obtained by **composing objects**.

- code reuse through extension and/or composition of existing types
- polymorphism through interfaces and/or composition of concrete types

[3]It is not an accident that exposition of inheritance in books usually draws on the same worn-out example Car-Vehicle: it is indeed hard to come by with examples that do not seem as contrived as this one, the `FlyingTurtle` being no exception.

[4]If a single type inherits properties of the same name from two different supertypes, which one to choose at runtime?

[5]C++ being a notable exception, and the case at hand, for issues related to multiple inheritance.

[6]The book was aiming at providing generic solutions to recurrent problems, using the languages favored by the industry (C++/Java) at the times; some misread it as a set of rigid prescriptions.

This will be the subject of our next lecture, that will illustrate these concepts with a language that does not implement inheritance: Go.