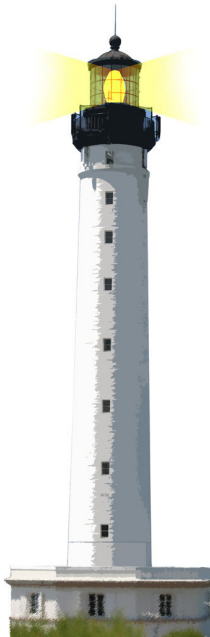
Polymorphic objects

support for software evolution

S. Ducasse



Pharo

http://www.pharo.org

Goals

- Polymorphic objects are key for evolution
- What ’s up in statically typed languages?
- Why do we need interfaces?

Coding against an API

'if it looks like a duck and quacks like a duck, it is a duck'

- In dynamicallty-typed languages, your objects do not have to be from the same hierarchy to be able to work with others
- They have to understand the messages that are needed for a good interaction for their role
- Related to the Adapter Design Pattern

Simple Example

```
Shape (draw)
  Rectangle (draw)
  Square (draw)
  Circle (draw)
```

```
Canvas >> display
  shapes do: [ :s | s draw ]
```

Adding Rhombus: Possibility one

If you can subclass Shape

```
Shape (draw)
  Rectangle (draw)
  Square (draw)
  Circle (draw)
  Rhombus (draw)
```

Adding Rhombus: Possibility two

If you cannot subclass Shape for any reason

```
Shape (draw)
  Rectangle (draw)
  Square (draw)
  Circle (draw)

Rhombus (draw)
```

Rhombus should implement the method draw to be able to play nicely with Canvas

```
Canvas >> display
  shapes do: [ :s | s draw ]
```

Step back

- Coding against an API
- Producing polymorphic objects (substituable objects) is KEY for evolution
- This is free in dynamically-typed languages

What about statically-typed?

Static types can get in your way

```
Shape s = new Shape();
```

- s can **only** contain instances of Shapes and its subclasses
- So if we cannot define Rhombus as subclass of Shape, it will not work because there is no type relationship between Rhombus and Shape

```
class Rhombus extend Object {...draw() {...} ...}
Shape s = new Rhombus()
>>>> Does not compile
```

Interface concept

- Group of method signatures
  - may contain default methods and more depending on their flavor
- Used by the type checker to check subtype relationships
- Support the evolution manipulation of instances of classes not in subtype relation (i.e. not in the same hierachy)

Example

```
interface IShape {
  draw();
}
```

```
class Shape extend Object implements IShape { ... }
```

```
class Canvas {
  ... display (){
      ArrayList<IShape> shapes = new ArrayList<IShape>() ...}
  ...}
```

class Rhombus Implements IShape

```
class Rhombus extend Object implements IShape {
  ... draw() { ... } ...}
```

So we can use Rhombus in Canvas because it implements the IShape interface expected by Canvas

Classes - Interfaces

- A class must implement the methods mentioned in the interface
- A class can implement many interfaces
- An interface can be composed out of multiple interfaces

Interfaces: step back

A nice mechanism for statically-checked languages

- defines what is expected
- lets the system evolve

When you use a class as a type:

- You freeze the possible instances
- You will only be able to have instances of type or subtypes

When you use an interface as a type:

- You will be able to use any instance of classes implementing the interface

Interfaces and nominal types

- Nominal types means that only the name if the type is considered (not its methods)
- Pay attention two interfaces with different names but the same contents are NOT compatible
- You will not be able to substitute instances of a class using one interface by instances of another class using another interface with the same contents

Conclusion

Polymorphism and interfaces support evolution

- Focusing on APIs is better for evolution than typing relationship

A course by

S. Ducasse, G. Polito, and Pablo Tesone