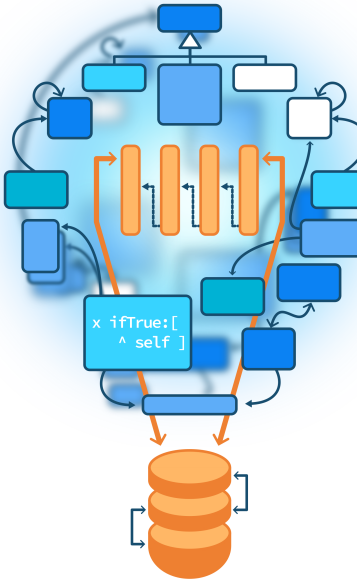
Polymorphic objects

Support for software evolution

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone



Pharo

Goals

- Polymorphic objects are key to software evolution
- What about them in statically typed languages?
  - why do we need interfaces in statically typed languages?

Simple Example

```
Shape (draw)
  Circle (draw)
  Rectangle (draw)
  Triangle (draw)
```

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

How to support rhombus?

Solution 1: subclassing Shape

Shape (draw)
- Circle (draw)
- Rectangle (draw)
- Triangle (draw)
- Rhombus (draw)

Solution 2: disjoint class

What happen if you cannot subclass Shape?

```
Shape (draw)
   Circle (draw)
   Rectangle (draw)
   Triangle (draw)

Rhombus (draw)
```

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

Polymorphic objects

Rhombus instances are polymorphic to shape objects even if Rhombus is not a subclass of Shape

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

Step back

Producing polymorphic objects (substituable objects) is KEY to software evolution.
In dynamically-typed languages:

- Objects do not have to be from the same hierarchy to work together
- Objects should understand the messages that are needed to play their role
  - e.g Rhombus **implements** draw
- **Duck typing**
  - *If it walks like a duck and it quacks like a duck, then it is a duck*

What about statically-typed languages?

Static types can get in your way:

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

- s can **only** contains instances of Shape or its subclasses
- if we cannot define Rhombus as a subclass of Shape (e.g. final class), it will not work because there is no subtype relationship between Rhombus and Shape

```
class Rhombus extend Object {...draw() {...} ...}
Shape s = new Rhombus()
> compilation error
```

Interface concept

An interface:

- has a name
- defines a type
- have one or more super-types
- contains a group of method signatures
- may contain default methods

Why interfaces?

- allow developpers to define subtypes out of class hierarchies
- are used by the type checker to check subtype relationships
- support evolution

Solution 3: with an interface

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

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

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

Solution 3: Rhombus implements IShape

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

The Rhombus class:

- inherits from Object
- implements IShape expected by Canvas

Rhombus and Shapes instances are subtypes of IShape and compatible with Canvas

Classes and 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

- Typing a variable using a class restricts the possible values of that variable to instances that class or one of its subclasses

```
Shape shape;
Collection<Shape> shapes;
```

- Interfaces are nice mechanism for statically-typed languages to define what is expected without restricting evolution

```
IShape shape;
Collection<IShape> shapes;
```

Interfaces and nominal types

Interfaces define “nominal types” (different from duck typing)

- type compatibility is only based on the name of the type
- two interfaces with different names but the same contents are NOT compatible
- instances of a class using one interface CANNOT be substituted by instances of another class using another interface with the same content

Conclusion

- Polymorphic objects are key to support software evolution
- Code against an API
  - Focusing on APIs is better for evolution than typing relationship
- In dynamically-typed languages, polymorphism is free
- In statically-typed languages, interfaces are key to create polymorphic objects not restricted to a specific class hierarchy
- Related to the Adapter Design Pattern

Produced as part of the course on http://www.fun-mooc.fr

Advanced Object-Oriented Design and Development with Pharo

A course by
S.Ducasse, L. Fabresse, G. Polito, and P. Tesone

Inria
LearningLab

2023