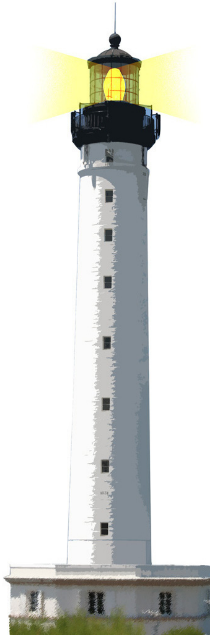
Advanced Object-Oriented Design

Subclassing vs. Subtyping

S. Ducasse



http://www.pharo.org

Goal

- What is the relation between the **API** of class and its **subclasses**?
- What is the relation between the **API** of class and its **clients**?
- What is **subtyping** & **subclassing**?
- Which one should we favor for good OO design?

Subtyping/subclassing and type systems

- You **can** use subtyping and subclassing in **dynamically-typed** languages!
- You **can** use subtyping and subclassing in **statically-typed** languages!

The compiler type checker :

- does not check such a point
- just checks that we can put **squares** into **square** shapes

What do you think about?

- Dictionary **subclass of** Set
- Dictionary **subclass of** HashedCollection

Subclassing

Dictionary subclass of Set

- a Dictionary is a set of bindings (associations Key->Value)
- Reuses (**abuse**) the implementation of Set
- Adds / removes extra messages (at:put:) to the default Set API

Subtyping

Dictionary is a subclass of HashedCollection

- Specializes HashedCollection (a collection of objects with a hash)
- Adds messages related to Dictionary (at:IfPresent:, ...)

Let us study a simple example

How to implement a Stack?

```
>>> s push: 12.
>>> s push: 24.
>>> s top
>>> s pop
24
>>> s isEmpty
false
```

- Using OrderedCollection (an ordered list of items) ?

Stack as subclass of OrderedCollection

```
OrderedCollection << Stack
```

```
Stack >> pop
  ^ self removeFirst

Stack >> push: anObject
  self addFirst: anObject

Stack >> top
  ^ self first
```

We get Stack»size, Stack»includes:, Stack»do:, Stack»collect: for free.

Wait!

- What do we do with the **rest** of the OrderedCollection API?
- a Stack **is not** an OrderedCollection!
- In a client program we cannot replace an OrderedCollection by a Stack

Wait!

```
OrderedCollection new addLast: anObject
```

```
Stack new addLast: anObject
```

Some messages do not make sense on Stack

```
Stack new last
```

We could cancel some operations

```
Stack >> removeFirst
  self error
```

And get a convoluted pop?

Remember:

```
Stack >> pop
  ^ self removeFirst
```

Jumping over cancelled operation :(

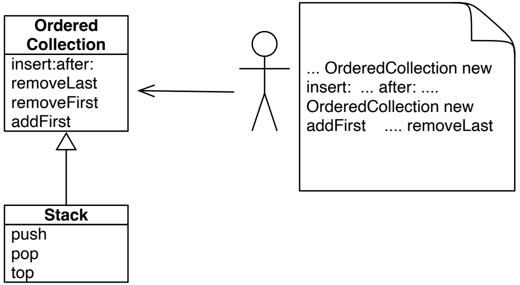
```
Stack >> pop
  ^ super removeFirst
```

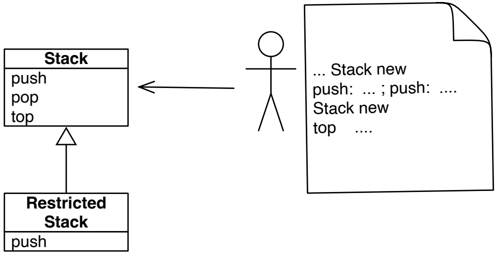
- Ugly
- Complexify the solution
- Complexify the evolution

Stepping back

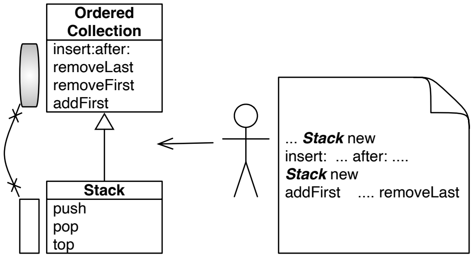
- There is not a **simple relationship** between `Stack` and `OrderedCollection` APIs.
- `Stack` interface is not an **extension** nor a **subset** of `OrderedCollection` interface.
- Compare with `RestrictedStack` a subclass of `Stack`
    - `RestrictedStack` interface is an *extension* of `Stack` interface

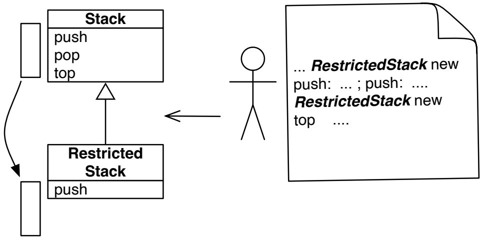
Compare the two uses





Compare the two replacements





Back to Stack

```
Object << Stack
  slots: {#elements}
```

```
Stack >> push: anElement
  elements addFirst: anElement
```

```
Stack >> pop
  ^ element ifNotEmpty: [ element removeFirst ]
```

Subclassing inheritance

- Inheritance for code reuse
- Subclass reuses code from superclass, but as a **different** specification
- It cannot be used everywhere its superclass is used. Usually overrides a lot of code

Cons:

- **Lowers** understanding
- **Hampers** future evolution
- **Forces** strange code

Subtyping Inheritance

- **Reuse** of specifications (generic code)
- A subclass **refines** superclass specifications
- A program that works with Numbers should 'work' with Fractions
- A program that works with Collections should 'work' with Arrays
- (We are not talking about behavioral subtyping)

Subclasses must not cancel methods

```
Stack >> removeFirst
  self error
```

This is a sign for bad design decision

- Cheap
- But you will pay later

Superclass/subclass

Usually the more generic definition should be in superclass

- RestrictedStack **subclass of** Stack **is a bad idea**

Better

- Stack **subclass of** AbstractStack
- RestrictedStack **subclass of** AbstractStack

Inheritance and polymorphism

- Polymorphism works best with **conforming/substituable** interfaces
- Subtyping inheritance creates families of classes with **similar interfaces**
  - An abstract class describes an interface fulfilled by its subclasses
- Inheritance helps software reuse by creating **polymorphic objects**
- Other classes implementing the same interface can also **be substituable**

Subtyping support

- We only have one `extend` or `subclass:` construct in PL
- Still you can express a **subtype** or **subclass** relationship between a class and its subclass.
- Subclassing/subtyping is not related to static typing

Conclusion

- Subtyping is about program specification **reuse**
- Subtyping is about to create **family of classes sharing common API**
- **Avoid** subclassing: it is a bad idea

A course by

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