Subclassing vs. Subtyping

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

- Discuss the relation between the API of a class and its subclasses
- Discuss the relation between the API of a class and its clients
- Compare subtyping & subclassing
- Impact on design
- Subtyping is good even in dynamically-typed languages
Example 1

class Poem extends LinkedList
{
    ...
}

What do you think about it?

- Yes, we can write this code
- What do you think of it? Does it make sense?

A poem API

- is addWord(word), isAlexandrin(), isHaiku(), ...
- should not contain addBeforeLink(aLinkOrObject, otherLink) (that is part of LinkedList)
Another example

class Stack extends LinkedList
{
  ...
}

What do you think about it?

- Yes, we can write this code.
- What do you think of it? Does it make sense?

A Stack API

- is pop(), push(el), top(), isEmpty()
- should not contain LinkedList methods.
Subclassing

The two previous examples are examples of subclassing, e.g., a subclass does not have an API in relation with its superclass. It reuses the superclass code.
Did you notice previous code snippets were in Java tiny syntax... because:

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

The compiler’s type checker does not check such a point

- It just checks that we can put **squares** into **squares**
Let us study a simple example

Basic Stack:

```plaintext
>>> s push: 12.
>>> s push: 24.
>>> s top
24
>>> s pop
24
>>> s isEmpty
false
```
Stack as subclass of OrderedCollection

OrderedCollection << Stack

Stack >> pop
  ^ self removeFirst

Stack >> push: anObject
  self addFirst: anObject

Stack >> top
  ^ self first

We get size, includes:, do:, collect: for free.
Wait!

- What do we do with the rest of the OrderedCollection API?
- Our Stack also understands: add:beforeIndex:, addAllFirstUnlessAlreadyPresent:, join:...
- A Stack is not an OrderedCollection!
- In a client program we cannot replace an OrderedCollection by a Stack
Some messages that make sense on the class `OrderedCollection` do not make sense on the class `Stack`

- `OrderedCollection new add: newObject beforeIndex: index`
- `OrderedCollection new add: newObject ; removeFirst`
We could cancel some operations

Stack >> removeFirst
self error
And get a convoluted pop?

Remember:

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

Jumping over cancelled operation :( 

```plaintext
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.
Imagine CountingStack

CountingStack >> pop
operations := operations + 1.
^ super pop

CountingStack >> push: anElement
operations := operations + 1.
^ super push: anElement
Compare the two uses

Ordered Collection
insert:after: removeLast removeFirst addFirst

Stack
push pop top

OrderedCollection new
insert: ... after: .... OrderedCollection new addFirst .... removeLast

Stack
push pop top

Counting Stack
push pop

Stack new
push: ... ; push: .... Stack new top ....
Compare the two replacements

- **Ordered Collection**
  - insert:after:
  - removeLast
  - removeFirst
  - addFirst

- **Stack**
  - push
  - pop
  - top

- **Counting Stack**
  - push
  - pop
  - top

- ... Stack new
  - insert: ... after: ....
  - Stack new
  - addFirst .... removeLast

- ... CountingStack new
  - push: ... ; push: ....
  - CountingStack new
  - top ....
Better use composition! A Stack holds a collection of elements

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 of code

**Cons:**
- **Lowers** understanding
- **Hampers** future evolution
- **Forces** strange code
Subtyping inheritance

- **Reuse** of specifications: interface inheritance
- 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
Subclasses must not cancel methods

Stack >> removeFirst
  self error

This is a sign of a bad design decision

- Cheap
- But you will pay later
RestrictedStack

Imagine that we have a stack where we can only push elements smaller than the top elements

```plaintext
push: anElement
    self top < anElement
    ifTrue: [^ self ]
    super push: anElement
```

What is the good superclass?

- **Stack** Probably.
- It would be better if the client handles this behavior, but maybe it is not mandatory or possible.
- A subclass does not have to make sure that the client program works (this is behavioral subtyping)
About Liskov Substitution Principle (LSP)

’if for each object o1 of type S there is another object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.’ Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23,5 (May 1988)

- LSP is about behavioral typing (about the same behavior)
- Most of the time when you define subclass to change behavior
- By definition, a subclass often exhibits a slightly different behavior than its superclass
- Therefore LSP looks useless in such a context.
Inheritance and polymorphism

- Polymorphism works best with **conforming/substitutable** interfaces
- Subtyping inheritance creates **families** of classes with **similar interfaces**
  - An abstract class describes an interface fulfilled by its subclasses
- Subtyping inheritance helps software reuse by creating **polymorphic objects**
- Now classes in different hierarchies implementing the same interface can also be **substitutable**
We only have one `extend` or subclass: construct in programming language.

Still you can express a `subtype` or `subclass` relationship between a class and its subclass.

Subclassing/subtyping is not related to static typing.
Conclusion

- Subclassing is about program specification **reuse**
- Subtyping is about creating **family of classes sharing common API**
- **Avoid** subclassing: it is a bad idea when you want subtyping
Advanced Object-Oriented Design and Development with Pharo

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

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
https://creativecommons.org/licenses/by-nc-nd/3.0/fr/