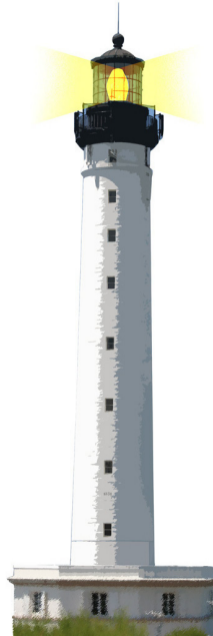


Message Sends are Plans for Reuse

S. Ducasse and G. Polito



<http://www.pharo.org>



About this lecture

- Related to sending a message is making a choice lecture
- Relevant to any object-oriented language
- Another essential aspect of object-oriented design



What you will learn

- Message sends are hooks for subclasses
- Message sends are places where subclasses code can be invoked



Let us start to reflect

Anecdotes

- *I like big methods because I can see all the code*
- *I do not like small methods*

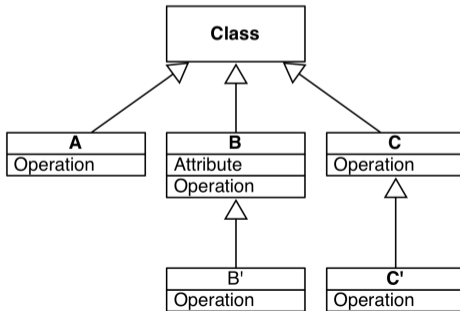
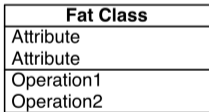
Questions

- Why large methods leads to *under-optimal* design?
- Why writing small methods is a sign of good design?



Remember...

- a message send makes a choice
- a class hierarchy defines the choices
- self **always represents the receiver**
- method lookup starts in the **class of the receiver** (except for super)



An example

Imagine

```
Node >> setWindowWithRatioForDisplay
| defaultNodeSize |
defaultNodeSize := mainCoordinate / maximizeViewRatio.
self window add:
  (UINode new
   with: bandwidth * 55 / defaultWindowSize).
previousNodeSize := defaultNodeSize.
```

What if we want to change the `defaultNodeSize` formula in a subclass

Duplication as a bad solution

Duplicate the code in a subclass

Node subclass: OurSpecificNode

...

```
OurSpecificNode >> setWindowWithRatioForDisplay
| defaultNodeSize |
defaultNodeSize :=
  (mainCoordinate / maximizeViewRatio) + 10.
self window add:
  (UINode new
   with: bandwidth * 55 / defaultWindowSize).
previousNodeSize := defaultNodeSize.
```



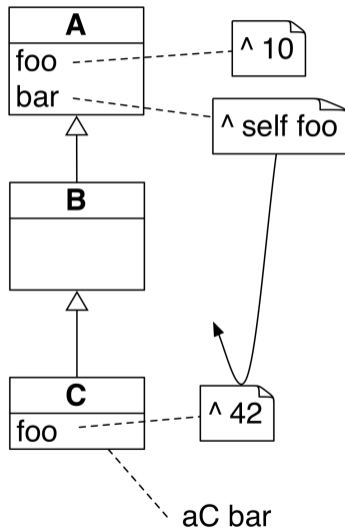
Avoid duplication

- Duplication is not a good practice:
 - duplication copies bugs
 - changing one copy requires changing others
- Note that in Java-like languages, using `private` attributes makes duplication in subclasses impossible



A much better solution

- Define small methods
- Send messages
- Subclasses can override such methods



We can refactor this

```
Node >> setWindowWithRatioForDisplay
| defaultNodeSize |
defaultNodeSize := (mainCoordinate / maximizeViewRatio).
self window add:
  (UINode new
   with: bandWidth * 55 / defaultWindowSize).
previousNodeSize := defaultNodeSize.
```

Into ...

```
Node >> setWindowWithRatioForDisplay
| defaultNodeSize |
defaultNodeSize := self ratio.
self window add:
  (UINode new
   with: bandwidth * 55 / defaultWindowSize).
previousNodeSize := defaultNodeSize.
```

```
Node >> ratio
^ mainCoordinate / maximizeViewRatio
```



Subclasses reuse superclass logic

```
Node >> ratio  
  ^ mainCoordinate / maximizeViewRatio
```

A subclass can refine the behavior

```
OurSpecificNode >> ratio  
  ^ super ratio + 10
```



Another step

```
Node >> setWindowWithRatioForDisplay
| defaultNodeSize |
defaultNodeSize := self ratio.
self window add:
  (UINode new
   with: bandwidth * 55 / defaultWindowSize).
previousNodeSize := defaultNodeSize.
```

What if we want have a different UINode.



Another step

We can also extract the `UINode` instantiation.

```
Node >> setWindowWithRatioForDisplay  
  | defaultNodeSize |  
  defaultNodeSize := self ratio.  
  self window add: self uiNode.  
  previousNodeSize := defaultNodeSize.
```

```
Node >> uiNode  
  ^ UINode new  
    with: bandwidth * 55 / defaultWindowSize
```

Do not hardcode class use

```
Node >> uiNode  
  ^ UINode new  
    with: bandwidth * 55 / defaultWindowSize
```

Define methods returning classes

```
Node >> uiNode  
  ^ self uiNodeClass new  
    with: bandWidth * 55 / defaultWindowSize.
```

```
Node >> uiNodeClass  
  ^ UINode
```



Many small messages

Small messages are a sign of good design, because:

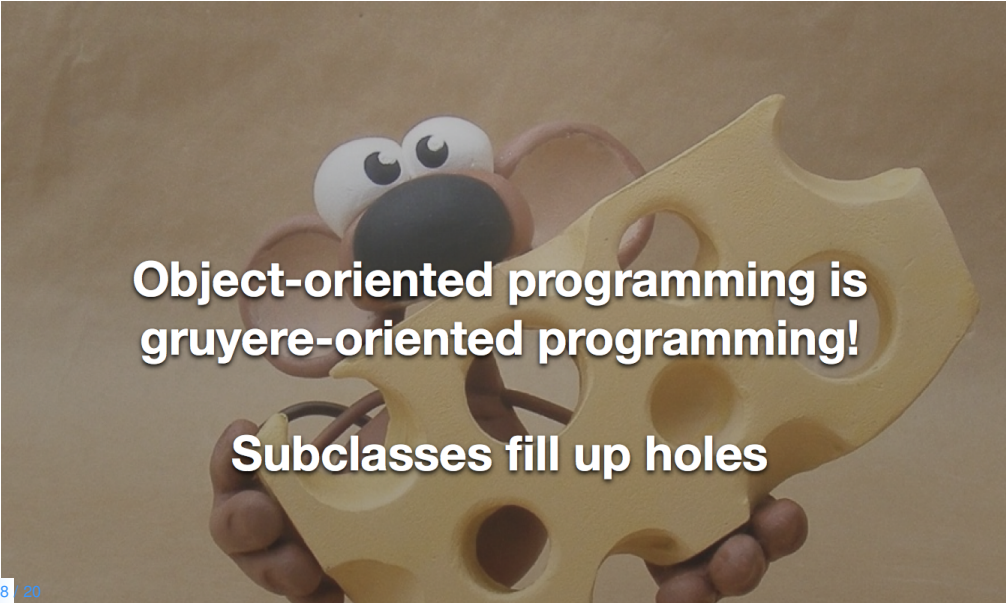
- each little method is a potential **hook** for customization
- gives name to expressions

Some developers complain about all these small methods

- But small methods are good
- Give meaningful name to methods
- Do not need to read all method definitions



Gruyere-oriented programming

A cartoon bee with large white eyes and a black nose is holding a large wedge of Gruyere cheese. The cheese has several holes of various sizes. The bee is positioned behind the cheese, with its hands visible at the bottom corners. The background is a plain, light brown surface.

**Object-oriented programming is
gruyere-oriented programming!**

Subclasses fill up holes

Conclusion

- Code can be reused and refined in subclasses
- Sending a message in a class defines a **hook**:
 - i.e., a place where subclasses can **inject variations**
- Prefer small methods because:
 - this gives names to expressions
 - this gives freedom to subclasses



A course by

S. Ducasse, G. Polito, and Pablo 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/>