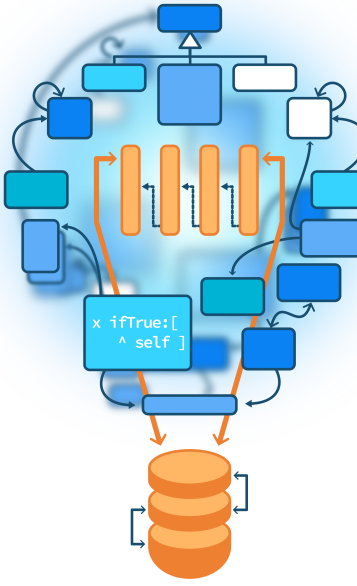
Message Sends are Plans for Reuse

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



Pharo

http://www.pharo.org

About this lecture

- Related to:
  - 'Sending a message is making a choice' and
  - self semantics
- 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 code of subclasses can be invoked

Let’s start thinking

Anecdotes

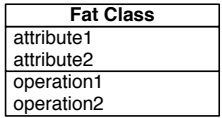
- *I like big methods because I can see all the code*
- *I do not like small methods*

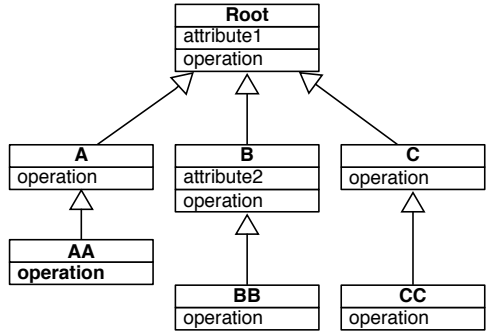
Questions

- Why large methods lead 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 possible choices
- self **always represents the receiver**
- Method lookup starts in the **class of the receiver** (except for super)





An example

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

What are the possible solutions to change the defaultNodeSize formula in a subclass?

Bad solution: duplication

Duplicate the code in a subclass

```
Node << #NodeWithMargins
  ...
```

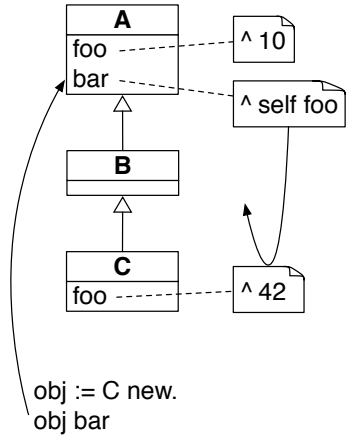
```
NodeWithMargins >> setWindowWithRatioForDisplay
  | defaultNodeSize |
  defaultNodeSize := (mainCoordinate / maximizeViewRatio) + 10.
  self window add: (UINode new with: bandWidth * 55 / defaultWindowSize).
  previousNodeSize := defaultNodeSize.
```

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

Essence of a better solution

- Define small methods for each aspects
- Send messages to self
- Subclasses can override such methods



Applying it to our example

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 can now reuse the superclass logic

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

A subclass can redefine this behavior into:

```
NodeWithMargins >> ratio
  ^ super ratio + 10
```

- In general there is no real need to invoke super ratio, but in our example this is better
- defaultNodeSize is computed when we execute:

```
NodeWithMargins new setWindowWithRatioForDisplay
```

Another step

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

How to use a different UINode in subclasses?

Another step: same solution applied

Extract the UINode instantiation into a separate method.

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

```
Node >> createUINode
  ^ UINode new with: bandWidth * 55 / defaultWindowSize
```

Improvement: do not hardcode class use

Refactor this:

```
Node >> createUINode
  ^ UINode new with: bandWidth * 55 / defaultWindowSize
```

into:

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

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

- Subclasses can change UI node class
- Good practice to define methods that return classes
- BTW, easy in Pharo because classes are regular objects!

Many take-aways

Small methods are a sign of good design, because:

- they give a **name** to expressions
- they encapsulate complexity (no need to read all method definitions) if their name is meaningful
- they ease testing
- they support self-send messages
- self-send messages are potential **hooks** for extensibility in subclasses (redefinition)

Emmental-oriented programming

Object-oriented programming is Emmental-oriented programming!
Subclasses fill up the holes

Conclusion

- Code can be reused and refined in subclasses
- Sending a message to `self` in a class defines a **hook**:
  - i.e. a place where subclasses can **inject variations**
- Prefer **small** methods because:
  - it gives names to expressions
  - each message to a small method is an extensibility point for subclasses

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