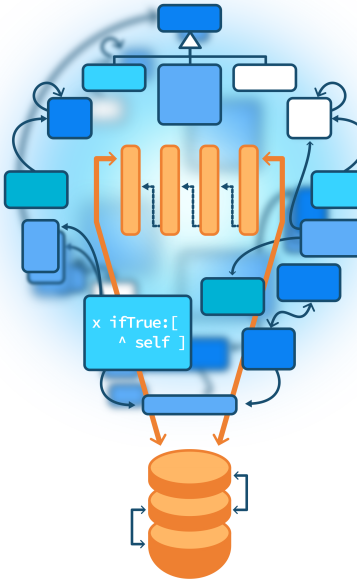
Methods: the elementary unit of reuse

Obvious but important

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



Pharo

Executing a method is reusing its code

Obvious but it is always good to hear it again

- Defining a method enriches the API of an object
- Calling a method is the first level of reuse

Case Study

```
PRTree >> inspectionPresenter
  <inspectorPresentationOrder: 35 title: 'PillarTree'>
  ^ SpTreePresenter new
    roots: { self };
    children: [ :aNode | aNode children ];
    display: [ :each |
      String streamContents: [ :stream |
        stream nextPutAll: each class name.
        each class = PRHeader ifTrue: [
          stream
            nextPutAll: '( level ';
            nextPutAll: each level asString;
            nextPutAll: ' )' ] ] ];
    yourself
```

Case Study: client side complexitiy

```
...
        String streamContents: [ :stream |
            stream nextPutAll: each class name.
            each class = PRHeader ifTrue: [
                stream
                    nextPutAll: '( level ';
                    nextPutAll: each level asString;
                    nextPutAll: ' )' ].
...
```

Why the client of a document is forced to define this behavior?

Better define two methods

```
PRObject>>displayStringOn: stream
  stream nextPutAll: self class name
```

```
PRHeader>>displayStringOn: stream
  super displayStringOn: stream.
   stream
        nextPutAll: '( level ';
        nextPutAll: self level asString;
        nextPutAll: ' )'
```

see Hook and Template Lecture!

And send a message

Sending a message will call a method (reuse its code)!

```
PRTree>>inspectionPresenter
  <inspectorPresentationOrder: 35 title: 'PillarTree'>
  ^ SpTreePresenter new
    roots: { self };
    children: [ :aNode | aNode children ];
    display: [ :each |
      String streamContents: [ :stream |
          each displayStringOn: stream ] ];
   yourself
```

Another example: logic repetition

```
...
stream := WriteStream on: (String new: 1000).
#(1 2 3) printOn: stream.
stream contents
```

```
...
stream := WriteStream on: (String new: 1000).
... printOn: stream.
stream contents
```

streamContents: to the rescue

```
String streamContents: [:stream | #(1 2 3) printOn: stream ]
```

- Encapsulates string creation
- Optimized
- Hides details
- Encapsulates termination

Encapsulate actions using blocks

```
SequenceableCollection class >> streamContents: blockWithArg
  ^ self new: 100 streamContents: blockWithArg
```

```
SequenceableCollection class >> new: newSize streamContents: blockWithArg

  | stream |
  stream := WriteStream on: (self streamSpecies new: newSize).
  blockWithArg value: stream.
  "If the write position of stream is at the end of the internal buffer of stream (
      originalContents),
  we can return it directly instead of making a copy as contents would do"
  ^ stream position = stream originalContents size
    ifTrue: [ stream originalContents ]
    ifFalse: [ stream contents ]
```

Another example of action encapsulation

```
'tintin' asFileReference readStreamDo: [:s | s next... ]
```

```
AbstractFileReference>> readStreamDo: aBlock
  | stream |
  stream := self readStream.
  ^ [ aBlock value: stream ]
    ensure: [ stream close ]
```

- Initialize
- and gracefully terminates

Stepping back

- Avoid spreading knowledge in clients
- Avoid duplication of logic in clients
- Encapsulate logic in the API
- Blocks (closure) helps building powerful API
  - but don’t abuse them! (see Blocks vs Objects lecture)

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