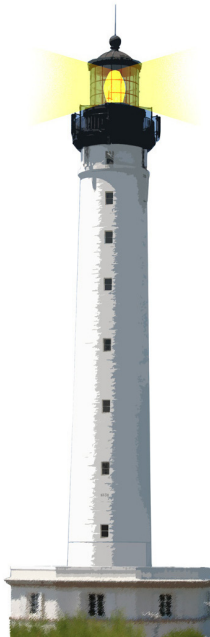
About Fluid API

The case of the class definition

S. Ducasse



Goal

- Think modular
- Looking at the class definition printer
- Fluid API

Fluid API

```
x foo: aString bar: anotherString babar: aThirdString
```

becomes

```
x
  foo: aString;
  bar: anotherString;
  babar: aThirdString
```

or

```
x
  babar: aThirdString;
  foo: aString
```

or

```
x foo: aString
```

Context: class definition

- How to support the **evolution** of class definition
- Supporting: package, various formats, slots
- Without parameter **explosion**

Historically: a class definition in ST-80

```
ArrayedCollection variableSubclass: #Array
  instanceVariableNames: ''''
  classVariableNames: ''''
  poolDictionaries: ''''
  category: ''Collections−Sequenceable−Base'''
```

Pharo up to Pharo 90

- Avoids poolDictionaries: when not used, but still...
- Supports package

```
ArrayedCollection variableSubclass: #Array
  instanceVariableNames: ''''
  classVariableNames: ''''
  package: ''Collections−Sequenceable−Base''
```

```
Object subclass: #Point
  instanceVariableNames: ''x y''
  classVariableNames: ''''
  package: ''Kernel−BasicObjects'''
```

Method parameter explosion

Modern class definition should support:

- Packages, tags, slots,
- New kind of subclasses (ephemerons,....)

Challenge: How to support new information without method parameter combinatorial explosion?

Using Fluid API

- Using a fluid API (cascade instead of mandatory parameters)
- Only the necessary parameters
- Composable
- Extensible

Fluid class definition in Pharo

```
ArrayedCollection << #Array
  layout: VariableLayout;
  tag: 'Base';
  package: 'Collections-Sequenceable'
```

```
Object class << Point class
  package: 'Kernel'
```

- Modular
- Compact
- Extensible

Support complex slots

Previous Pharo class definition did not support well slots

```
SpAbstractWidgetPresenter << #SpDiffPresenter
  slots: {
      #showOptions => SpObservableSlot .
      #showOnlyDestination => SpObservableSlot .
      #showOnlySource => SpObservableSlot .
      #contextClass => SpObservableSlot .
      #leftLabel => SpObservableSlot .
      #leftText => SpObservableSlot .
      #rightLabel => SpObservableSlot .
      #rightText => SpObservableSlot };
    tag: 'Widgets';
    package: 'Spec2-Core'
```

Supporting traits

```
Trait << #TTranscript
  package: 'Transcript-Core-Traits'
```

Builder API roles

- a **starting**/**creator** message: to create a kind of accumulator (an holder of arguments)
  - this one can be omitted when the builder is already created by the framework
- some configuration/setter objects
- a **closing** message: to perform an action **once** the arguments are passed around

Two specific roles

```
SpAbstractWidgetPresenter << #SpDiffPresenter
  ...
  ...
  package: 'Spec2-Core'
```

- << creates a class builder
- package: tells the builder to create and install the configured class

In Seaside

with: is a closing message

```
html heading
  level: 3;
  with: 'A third level heading'.
```

```
html paragraph with: 'Hello world.'.
html orderedList with: [
  html listItem: 'Item 1'.
  html listItem: 'Item 2' ].
```

Analysis of Fluid API

Pros:

- Handles combinatorial explosion
- Handles optional argument

Cons:

- The message order can be important
- There is a need for a closing message (package:)

Conclusion

- Three kinds of objects
  - container creator
  - setters
  - closer
- Fluid API is nice when we face many optional/exclusive parameters
- Still always think about the builder and closing message.

A course by

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