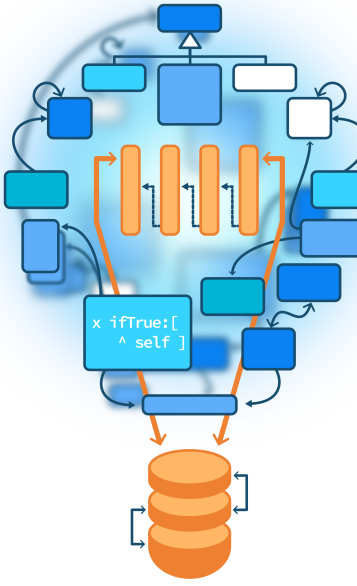
About Fluid APIs

The case of the class definition

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



Pharo

Goals

- Think about fixed and large parameter-driven APIs
- Fluid API?
- Think modular
- Study the class definition

Numberous mandatory parameter APIs

```
DockingBarMorph >>
add: wString icon: aForm help: hString subMenu: aMorph action: anAction
  keyText: aText
```

as invoked in

```
add: wString icon: aForm help: hString subMenu: aMorph
  "Append the given submenu with the given label."

  d add: wString icon: aForm help: hString subMenu: aMorph action: nil keyText: nil
```

- May force users to specify all parameters (often with nil)

Handling optional parameters may blow up API

Turning

```
add: wString icon: aForm help: hString subMenu: aMorph
  "Append the given submenu with the given label."
  d add: wString icon: aForm help: hString subMenu: aMorph action: nil keyText: nil
```

into

```
add: wString icon: aForm help: hString subMenu: aMorph
  "Append the given submenu with the given label."

  d add: wString icon: aForm help: hString subMenu: aMorph
```

Requires

```
add: w icon: f help: h subMenu: m
add: w icon: f help: h subMenu: m action: a
add: w icon: f help: h subMenu: m action: a keyText: t
```

- May result in multiple methods to handle all the cases of optional value

What is a fluid API?

```
x add: wString icon: aForm help: hString
```

becomes

```
x
  add: wString ;
  icon: aForm ;
  help: hString
```

or

```
x
  icon: aForm ;
  add: wString
```

or

```
x help: hString
```

Analysis

- **No need** to pass default value around
- **No combinatorial** parameters explosion
- Users focus on the message **they need**
- Requires good (and modular) initialization
- Should pay attention to dependencies between parameter

Case study: 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
- Supports package

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

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

Method parameter explosion

Modern class definition should support:

- Packages, tags, slots,
- Various instance format: Word, Byte, Variable, Indexed....
- New kind of format: ephemerons,....

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

Using a Fluid API

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

But

- Needs a builder
- (optional) default values

Fluid class definition in Pharo

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

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

- « returns a class builder
- Other messages configure it
- Modular
- Compact
- Extensible

Support well complex 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'
```

Support for 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** messages
- A **closing** message: to perform an action **once** the arguments are passed around

Roles in class definition

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

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

Case study2: 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
- Forces initialization with good default values
  - default values are not passed around via extra parameters but initialized in the configured object

Cons:

- The message order can be important
  - use configuration messages with more than one argument
- There is a need for a closing message (`package:`)

Conclusion

- Three kinds of objects
  - container creator
  - setters
  - closer
- A fluid API is nice when we face many optional/exclusive parameters
- See lectures on builder idioms

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