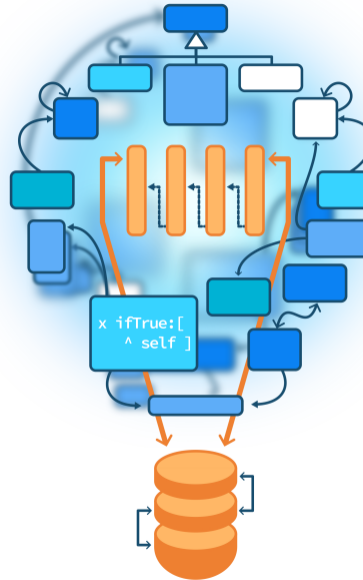


# Delegation of actions and accumulator

Form validation as an example

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



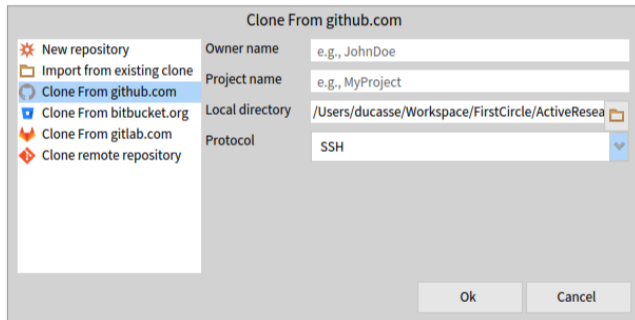
# Objectives

- Think about objects
- Think about structure traversal
- Look at objects as accumulators



# The case: Validation

- We want to validate UI forms
- Nested components may want to validate **or not** their contents
  - at input field or just at the pane level



The image shows a dialog box titled "Clone From github.com". On the left, there is a list of options: "New repository", "Import from existing clone", "Clone From github.com" (highlighted), "Clone From bitbucket.org", "Clone From gitlab.com", and "Clone remote repository". On the right, there are four form fields: "Owner name" with the placeholder "e.g., JohnDoe", "Project name" with the placeholder "e.g., MyProject", "Local directory" with the path "/Users/ducasse/Workspace/FirstCircle/ActiveResea" and a folder icon, and "Protocol" with the value "SSH" and a dropdown arrow. At the bottom right, there are "Ok" and "Cancel" buttons.

# Questions

- How can we navigate a tree of instances (widgets)?
- Where children can decide to be skipped?
- What do we report?



# Validation

What should return the validation?

- Yes/no?
- Specific objects with semantics
  - e.g. filepath is isValid?



# A formular: A tree of instances



# A first design

- Any presenter can validate its contents
- Per default does nothing

```
SpPresenter >> isValid  
^ true
```

```
SpPresenter >> report  
^ OkReport new
```



# A given item can refine it

```
MyFilePathPresenter >> isValid  
  ^ self inputField isEndingBy: 'git'
```

```
MyFilePathPresenter >> report  
  ^ WrongFileEndingReport new expecting: 'git' ; for: self path
```





# A first design: Container

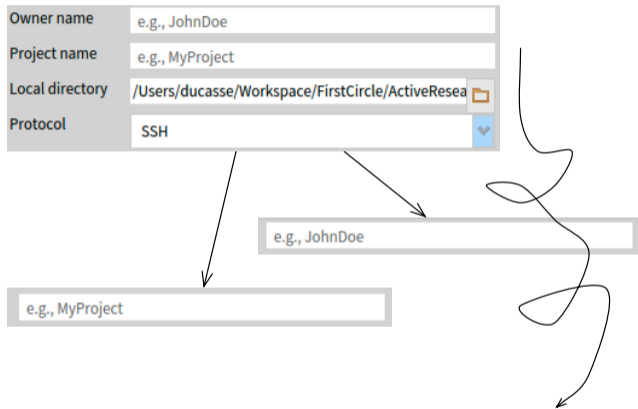
A container defines the semantics of collection for its children

```
SpOptionPresenter >> isValid  
  ^ self children allSatisfy: [:each | each isValid ]
```

```
SpOptionPresenter >> report  
  | report |  
  report := SpValidationReport new.  
  self children do: [ :childPresenter |  
    childPresenter isValid ifFalse: [ report add: childPresenter report ]].  
  ^ report
```



# Flow's first design

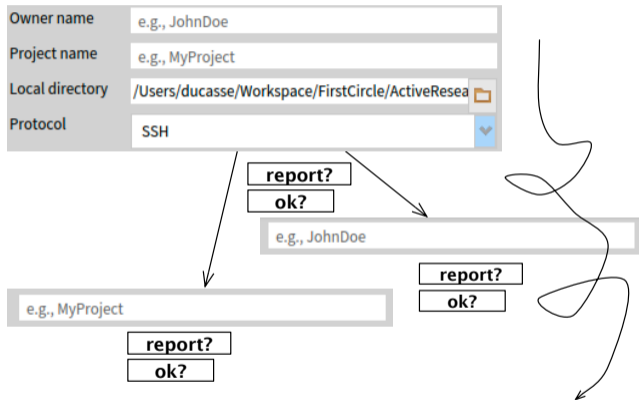


# Analysis

- To have a report we need to know if the validation failed or not
- Should `isValid` return a report?
- If `isValid` returns a report then we have to return an ok report for anybody

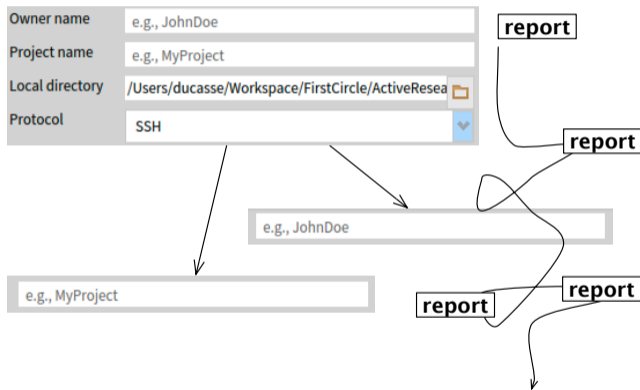


# Flow's first design



# Second design: provide an accumulator

Pass around a basket and let any sub instance decides if it wants to participate



# Second design: default

By default do not add to the report

```
SpPresenter >> validateInto: aReport  
^ self
```

## Second design: Containers and leaves

Each validating subcomponent

- gets the responsibility to fill up the report
- can bring its information to the report

```
MyFilePathPresenter >> validateInto: aReport  
  ^ aReport add: (WrongFileEndingReport new expecting: 'git'; for: self path)
```

```
SpOptionPresenter >> validateInto: aReport
```

```
  self children do: [ :presenter | presenter validateInto: aReport ].  
  ^ aReport
```



# Conclusion

- Question interrogative forms
- Let the object decides if it wants to join a process but passing a container
- You may also have some double dispatch between the report and the container
- Explore design





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



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/>