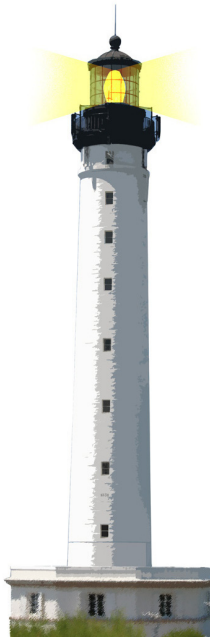
Reifying and delegating behavior

A case study

S. Ducasse



Pharo

http://www.pharo.org

Goals

- Study a concrete case
- Study introduction of a new object
- Show that delegation creates dispatch spaces
- Modular in addition

Case study: introducing .md file in Pillar

- Pillar used .pillar file containing Pillar text,
- Now there is .md file containing Microdown text

Pillar's .pillar file management

- How to get a parsed document?
- We ask the parser!
- The parser turns pillar file into a document tree
- There is ONLY one parser associated to a document
- The idea is to avoid to hardcode everywhere PRParser
- Worked for trying different versions of PetitParser parser

```
PRDocument parser parseFile: aFileReference
```

Case study: Pillar supports .pillar

```
PRAbstractOutputDocument >> buildOn: aPRProject

  | parsedDocument transformedDocument writtenFile |
  parsedDocument := self parseInputFile: file.
  parsedDocument properties: (self metadataConfigurationForDocument:
      parsedDocument).
  transformedDocument := self transformDocument: parsedDocument.
  writtenFile := self writeDocument: transformedDocument.
  self postWriteTransform: writtenFile.
  ^ PRSuccess new.
```

```
PRAbstractOutputDocument >> parseInputFile: anInputFile
  ^ PRDocument parser parse: anInputFile file
```

Problems

- PRDocument parser looks like a disguised global variable
- Only one syntax
- Checks for the file extension are hardcoded
- Difficult to know if a file is part of a project (books)
- Access to project configuration (user option) is cumbersome

Solution: Introduce InputDocument

First step:

- Instead of manipulating files, manipulate InputDocument objects
- InputDocument wrap files and more information (file extension, parser...)

Step 1: Introduce InputDocument

```
PRBuilAllStrategy >> filesToBuildOn: aProject

  ^ children flatCollect: [ :each |
    each allChildren
      select: [ :file | file isFile and: [ file extension = 'pillar' ] ]
      thenCollect: [ :file |
        PRInputDocument new
          project: aProject;
          file: file;
          yourself ] ]
```

```
PRInputDocument >> parser
  file extension = 'pillar'
    ifTrue: [ ^ PRDocument parser ].
  self error: 'No parser for document extension: ', file extension
```

First step

- We do not distribute responsibility

```
... select: [ :file | file isFile and: [ file extension = 'pillar' ] ]
```

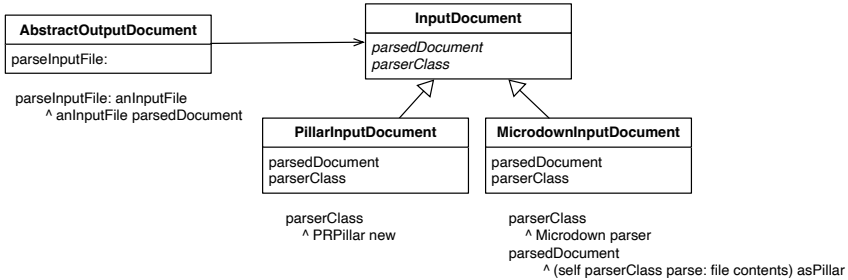
- Not modular!

Support for .mic/.md files

- Now Pillar compilation chain should accept .md file
- Different syntax!
- Different parser!

Refining InputDocument into a simple hierarchy

- Different classes
- Move behavior to such classes




```
PRInputDocument << #PRPillarInputDocument
  package: 'Pillar-ExporterCore'
```

InputFile is now responsible for its parser

```
PRAbstractOutputDocument >> parseInputFile: anInputFile
  ^ anInputFile parsedDocument
```

```
PRPillarInputDocument >> parsedDocument
  ^ self parserClass parse: file contents

PRPillarInputDocument >> parserClass
  ^ PRDocument parser
```

```
PRMicrodownInputDocument >> parsedDocument
  ^ (self parserClass parse: file contents) asPillar

PRMicrodownInputDocument >> parserClass
  ^ Microdown parser
```

Delegating extension checks

```
PRPillarInputDocument >> doesHandleExtension: anExtension
  ^ anExtension = 'pillar'
```

```
PRMicrodownInputDocument >> doesHandleExtension: anExtension
  ^ anExtension = 'mic'
```

Registration mechanism to support modularity

- We need to create objects of the right kind
- We use a registration mechanism, so that input documents can declare their existence

```
PRInputDocument class >> inputClassForFile: aFile

  ^ self subclasses
      detect: [ :each | each doesHandleExtension: aFile extension ]
      ifNone: [ PRNoInputDocument  ]
```

- Registration could be better (check corresponding lecture)

Creating the right kind of InputDocument objects

Now we are ready to create the adequate InputDocument objects

```
PRBuilAllStrategy >> filesToBuildOn: aProject

  ^ files collect: [ :file |
    (PRInputDocument inputClassForFile: file asFileReference) new
      project: aProject;
      file: (aProject baseDirectory resolve: file);
      yourself ]
```

Conclusion

- Turning implicit into an object
- Turning one object into objects of different but polymorphic classes
- Defining polymorphic behavior to be able to delegate
- Using registration to create modular design

A course by

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