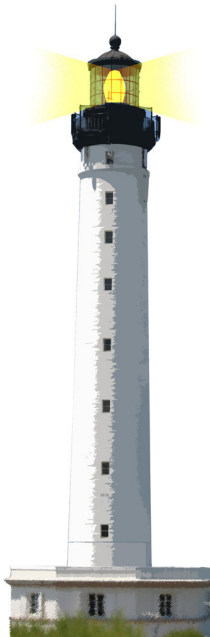
Reification and delegation

Microdown in Pillar case study

S. Ducasse



Pharo

http://www.pharo.org

Goals

- Creating specific objects supports delegation
- Delegation creates **dispatch** spaces (Strategy Design Pattern)
- Study a concrete case: Pillar handing now Microdown format
- Think about modularity

Case study: introducing .md file in Pillar

Existing

- Pillar is a document compilation chain to produce books, slides, sites
- Pillar used .pillar file containing Pillar text

New requirement: Now Pillar should handle .md files containing Microdown text

- How to support this new requirement?

Pillar's .pillar file management

How to get a document parsed?

- We ask the parser!
- The parser turns pillar file into a document tree
- There is **only one** parser associated to a document

```
PRDocument parser parseFile: aFileReference
```

- It avoids to hardcode PRParser everywhere
- Worked to substittute different versions of PetitParser parser

Case study: Pillar supports .pillar

```
PRAbstractOutputDocument >> buildOn: aPRProject

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

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

Problems

PRDocument parser implications

- There is only one parser
- Only one syntax

Other limits

- Checks for the file extension are hardcoded
- File does not know its project (book,...)
- Access to project configuration (user option) is cumbersome

At the end we do not have the possibility to distinguish between a .pillar and .md file

Solution: Introduce InputDocument

First step:

- Instead of manipulating files, manipulate InputDocument **objects**
- InputDocument wraps 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
```

Not much chagnes but still

- We do not distribute responsibility

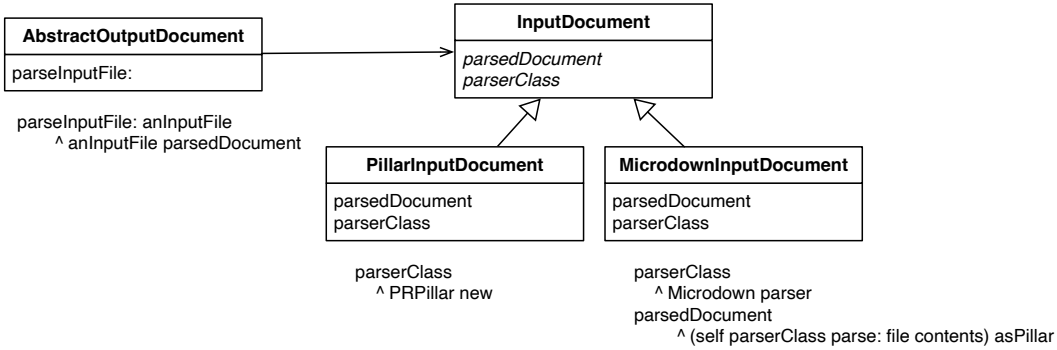
```
... select: [ :file | file isFile and: [ file extension = 'pillar' ] ]
```

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



InputFile is now responsible for its parser

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

becomes

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

Each subclass defines specific behavior

```
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

- Need to create objects of the right kind (PRMicrodownInputDocument or PRPillarInputDocument)
- 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  ]
```

- Note: 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
- Create dispatch spaces
- Using registration to create modular design

A course by

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