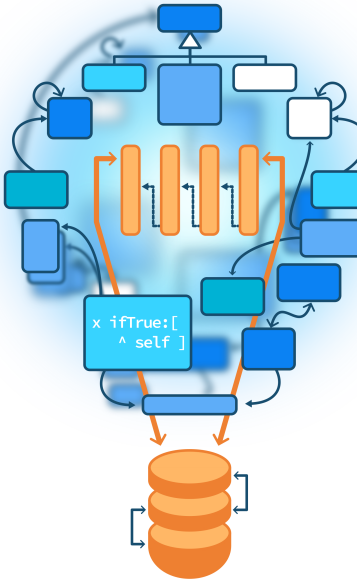
Reification and delegation

A case study: Microdown in Pillar

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



Goals

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

Case study: introducing .md file in Pillar

Existing:

- Pillar is a document compilation chain: it produces books, slides, sites
- Pillar used .pillar file containing Pillar text

New requirement:

- 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?

- Ask the parser
- The parser turns a pillar file into a document tree
- There was one parser associated to the class PRDocument

```
PRDocument parser parseFile: aFileReference
```

- Avoids to hardcode PRParser everywhere
- Worked to substitute 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
```

Challenges

PRDocument parser implications:

- There is **only one** parser
- Only one syntax

Other limits:

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

We cannot distinguish between a .pillar and .md file

Glimpse at a solution

- Define an object representing a specific inputs (.pillar, .md)
- Each input will know its format and corresponding parser

Now we get one parser per document type

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

- When a file ends by .pillar
- Create an instance of PRInputDocument

```
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 ] ]
```

Step 1: Introduce InputDocument

A PRInputDocument knows its parser

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

A little chicken step

We did not distribute responsibility **yet**

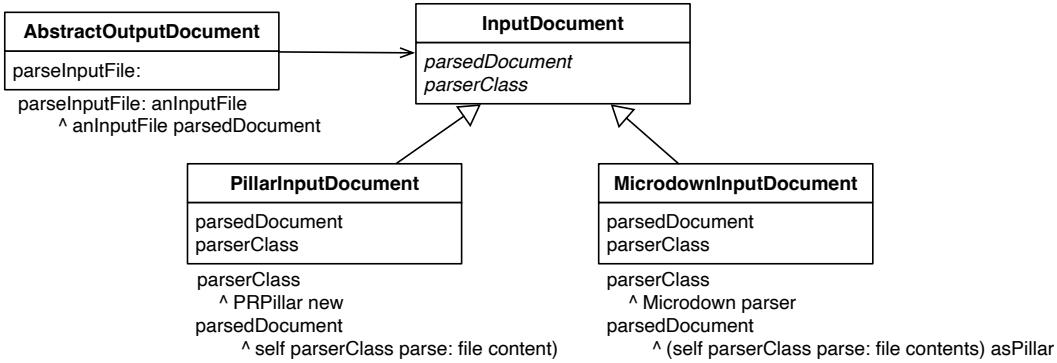
```
... select: [ :file | file isFile and: [ file extension = 'pillar' ] ]
```

Support for .md files

- Pillar compilation chain should accept .md files
- Different syntax
- Different parser

Refining InputDocument into a simple hierarchy

- Different classes
- Move behavior to such classes



InputFile objects are responsible for their 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

Each input document handles its extension

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

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

What if we want optional format

- We can parse .md and .pillar
- How to make them optional?

Registration mechanism to support modularity

- Use a registration mechanism, so that new input document kinds can declare their existence

```
PRInputDocument class >> inputClassForFile: aFile

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

- Note: the registration could be better (see corresponding Lectures)

Creating the right kind of InputDocument objects

Create the adequate InputDocument objects

```
PRBuilAllStrategy >> filesToBuildOn: aProject

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

Step back

- Turn **implicit into an object**
- Specialize one object into **objects of different but polymorphic** classes
- Define **polymorphic behavior** to be able to delegate
- Create dispatch spaces
- Use registration to obtain a modular design

Conclusion

- Define objects and their own behavior
- Delegate to such objects
- Think about place to create such objects
- **Tell do not ask** help you to spot place for variations
- Read Transform Conditionals of **Object-Oriented Reengineering Patterns** Book

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