Extensibility

George Fourtounis

Dept. of Informatics and Telecommunications
University of Athens

The Expression Problem

(`http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`)

We have:

- data of different kinds
- some operations we can perform on the data

Example:

- text files, PDF files
- print file description, print file extension

How easy is it to:

- add a new kind of data (that supports the operations)?
- add a new operation (that can be done differently on each kind of data)?

"Easy" = without recompiling existing code, without magic (e.g. casts)

Object-oriented with classes (Java, C++)

```
abstract class File {
  abstract void description();
  abstract void extension();
}
class PlainText extends File {
  void description() {
    System.out.println("Plain text file.");
  }
  void extension() {
    System.out.println(".txt");
  }
}
class PdfDocument extends File {
  void description() {
    System.out.println("PDF document.");
  }
  void extension() {
    System.out.println(".pdf");
  }
}
```

(Detail: instead of abstract classes, we could use interfaces.)

Object-oriented with classes (Java, C++)

Easy to add a new kind of data: just add a new subclass.

```
class PngImage extends File {
  void description() {
    System.out.println("PNG image.");
  }
  void extension() {
    System.out.println(".png");
  }
}
```

Object-oriented with classes (Java, C++)

Difficult to add a new operation: it must be added to the superclass and all subclasses must be modified to support it.

```
abstract class File {
  abstract void openWith();                          // added
}
class PlainText extends File {
  ...
  void openWith() {                                  // added
    System.out.println("Opens with a text editor.");
  }
  ...
}
class PdfDocument extends File {
  ...
  void openWith() {                                  // added
    System.out.println("Opens with a PDF viewer.");
  }
  ...
}
```

Pattern matching on values (Haskell, ML, F#)

Define a data type with cases for all different values:

```
data File = PlainText | PdfDocument
```

Define functions that do case-analysis:

```
description(f) =
  case f of
    PlainText   -> putStrLn "Plain text file."
    PdfDocument -> putStrLn "PDF document."

extension(f) =
  case f of
    PlainText   -> putStrLn ".txt"
    PdfDocument -> putStrLn ".pdf"
```

Pattern matching (Haskell, ML, F#)

Easy to add a new operation:

```
openWith(f) =
  case f of
    PlainText    -> putStrLn "Opens with a text editor."
    PdfDocument  -> putStrLn "Opens with a PDF viewer."
```

Pattern matching (Haskell, ML, F#)

Difficult to add a new kind of data, we have to modify both the data type and all its functions:

```
data File = ... | PngImage                    {* added *}

description(f) =
  case f of
    ...
    PngImage -> putStrLn "PNG image."         {* added *}

extension(f) =
  case f of
    ...
    PngImage -> putStrLn ".png"               {* added *}
```

Some languages have both OO classes and pattern matching (e.g. Scala).

There are unsafe ways around the problem: casts, reflection.

Some languages have both OO classes and pattern matching (e.g. Scala).

There are unsafe ways around the problem: casts, reflection.

Another solution: use a design pattern.

Rewrite using the Visitor pattern

```
abstract class FileVisitor {
  abstract void visit(PlainText f);
  abstract void visit(PdfDocument f);
}
class DescriptionVisitor extends FileVisitor {
  void visit(PlainText f) {
    System.out.println("Plain text file.");
  }
  void visit(PdfDocument f) {
    System.out.println("PDF document.");
  }
}  // similar: ExtensionVisitor
abstract class File {
  abstract void accept(FileVisitor visitor);
}
class PlainText extends File {
  void accept(FileVisitor v) { v.visit(this); }
}
class PdfDocument extends File {
  void accept(FileVisitor v) { v.visit(this); }
}
```

Using the Visitor pattern

A Visitor does double dispatch (= 2 calls with virtual method resolution):

```
File txt = new PlainText();
FileVisitor v = new DescriptionVisitor();
txt.accept(v);
```

Adding a new operation is now easy

```
class OpenWithVisitor extends FileVisitor {
  void visit(PlainText f) {
    System.out.println("Opens with a text editor.");
  }
  void visit(PdfDocument f) {
    System.out.println("Opens with a PDF viewer.");
  }
}
```

Use it, same as before:

```
File txt = new PlainText();
FileVisitor v = new OpenWithVisitor();
txt.accept(v);
```

But adding a new subclass of `File` is less easy

```
class PngImage extends File {
  void accept(FileVisitor v) { v.visit(this); }
}
```

All visitors must be modified to know about this new class:

```
class DescriptionVisitor extends FileVisitor {
  ...
  void visit(PngImage f) {
    System.out.println("PNG image.");
  }
}

class ExtensionVisitor extends FileVisitor {
  ...
  void visit(PngImage f) {
    System.out.println(".png");
  }
}
```

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context
- can work around limitations of the language and add expressive power

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context
- can work around limitations of the language and add expressive power
- can have its own disadvantages (so don't apply it blindly)

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context
- can work around limitations of the language and add expressive power
- can have its own disadvantages (so don't apply it blindly)
- can be more verbose compared to the same feature provided by a language as a built-in

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context
- can work around limitations of the language and add expressive power
- can have its own disadvantages (so don't apply it blindly)
- can be more verbose compared to the same feature provided by a language as a built-in
- can improve safety

Remarks

From our experience with the Visitor, a design pattern:

- is a code idiom, usable in a specific context
- can work around limitations of the language and add expressive power
- can have its own disadvantages (so don't apply it blindly)
- can be more verbose compared to the same feature provided by a language as a built-in
- can improve safety

In the next slides, we'll see more design patterns.