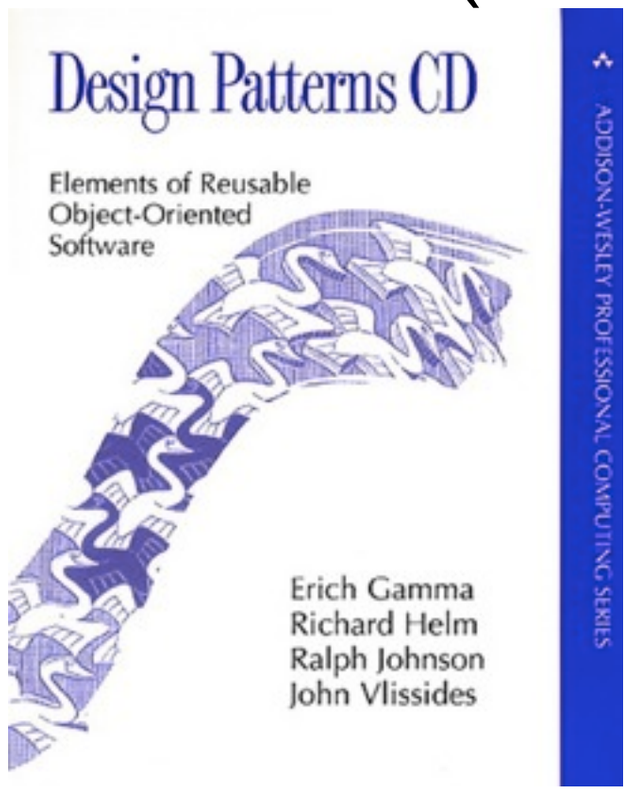
Design Patterns

S. Rosmorduc

Design Patterns

- Models of reusable architectures
- Concepts from «real-life» architectures (of buildings)
- Classical book



Why

- Object oriented architectures are difficult to design
- Some ‘micro-architectural’ designs are often useful
- Those designs are called «patterns»
- They provide a **common vocabulary**

Before patterns
a few simple rules

- Don’t repeat yourself: a responsability should be encoded in **one** place.
- A class represents a «simple» **concept** and represents it well.
- methods should do one thing, and have few parameters.

«code for interfaces, not for implementation»

- a class «A» should normally only «know» interfaces, not other classes :
    - instance variables should be interfaces
        - parameters should be interfaces...
- thus, any class can be replaced by a mock implementation : good for testing
- encourages low coupling between classes.

Patterns and their uses

- «cooking» recipes
- common vocabulary
- structural elements

Pattern description

- Pattern name: allows use of pattern for describing architecture.
- Motivation: what problems are solved
- Description and sample implementation
- Consequences: pros and cons, guideline for choosing to use this pattern.

A few examples of patterns

Singleton

- Motivation: some objects must exist only in one instance (example: printers spooler)
  - we want to be able to retrieve only one instance of those objects
  - we want to do it easily

Singleton: description

the only instance is stored in a **static** variable. In java, it can be directly initialized

```
public class PrinterSpooler {
  private static PrinterSpooler instance=
                 new PrinterSpooler();
  private PrinterSpooler() {....}
  public static PrinterSpooler getInstance() {
     return instance;
  }
}
```

constructors are **private**

to get the instance, one calls
PrinterSpooler a= PrinterSpooler.getInstance()

Singletons: examples

- java libraries: lots of singletons or variants of singletons.

```
Console c= System.console();

Calendar rightNow = Calendar.getInstance();

Runtime runtime= Runtime.getRuntime()
```

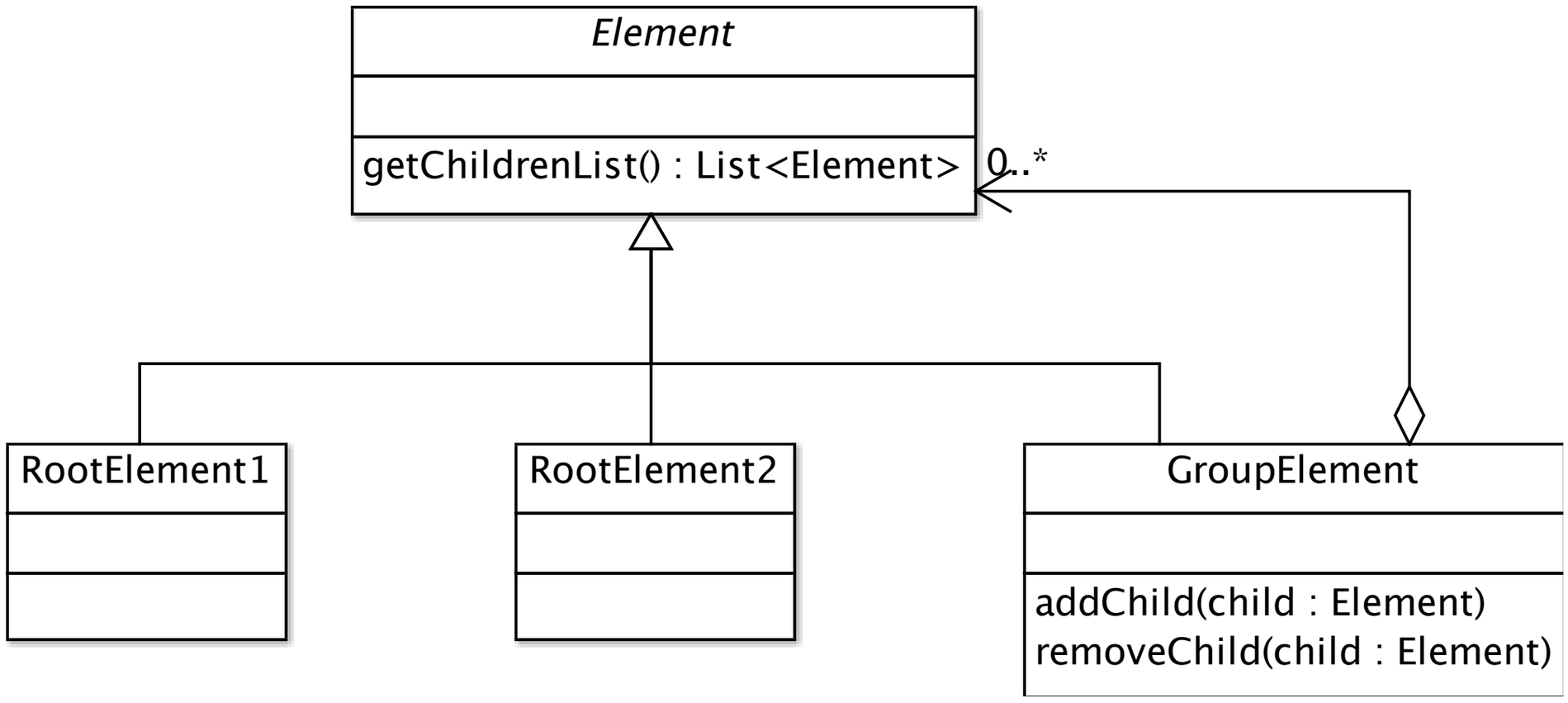
Singleton: consequences

- Basically, a «sophisticated» global variable.
- easy to do
- **Bad** for **testing** (can't be replaced by something else)
- better solutions: repository (JNDI) or *dependency injection*.

Composite

- Motivation: manipulate elements, as in text or graphical processors, where some elements can be groups of other elements.
- Idea: the «group-like» elements should hold a list of children elements

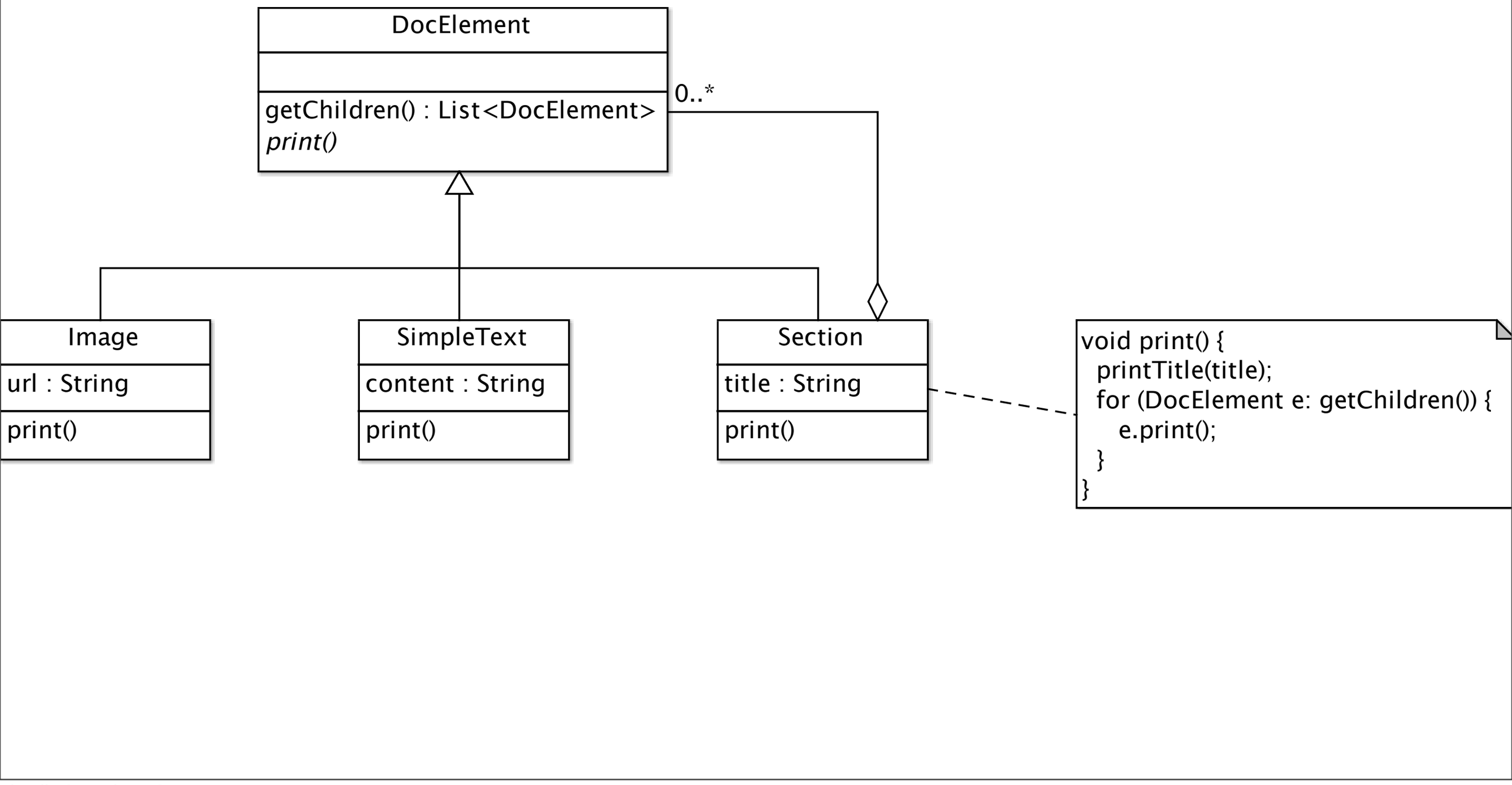
Composite implementation



Composite

- Note that we placed «getChildrenList()» on Element: simple navigation. All elements have a children list, which might be empty.
- Some implementations (including the original one) put addChild and removeChild in Element. Not safe.

Example : structured document

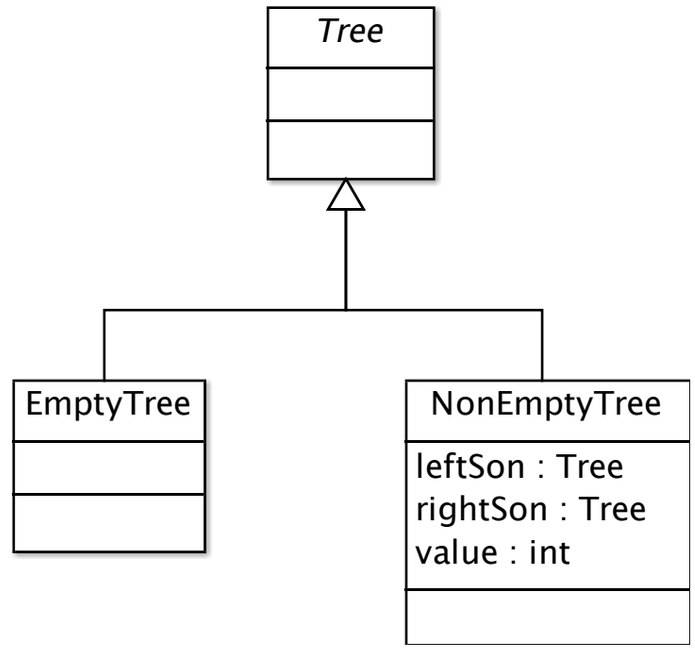


Visitor

- Motivation: if lots of operations can be done on a class hierarchy, one might end with overly «heavy» classes, which are then difficult to reuse. We would like to take non-fundamental operations away from the hierarchy implementation.

Visitor (Motivation)

- The tree class: many possible methods, height, sumOfValues, contains...

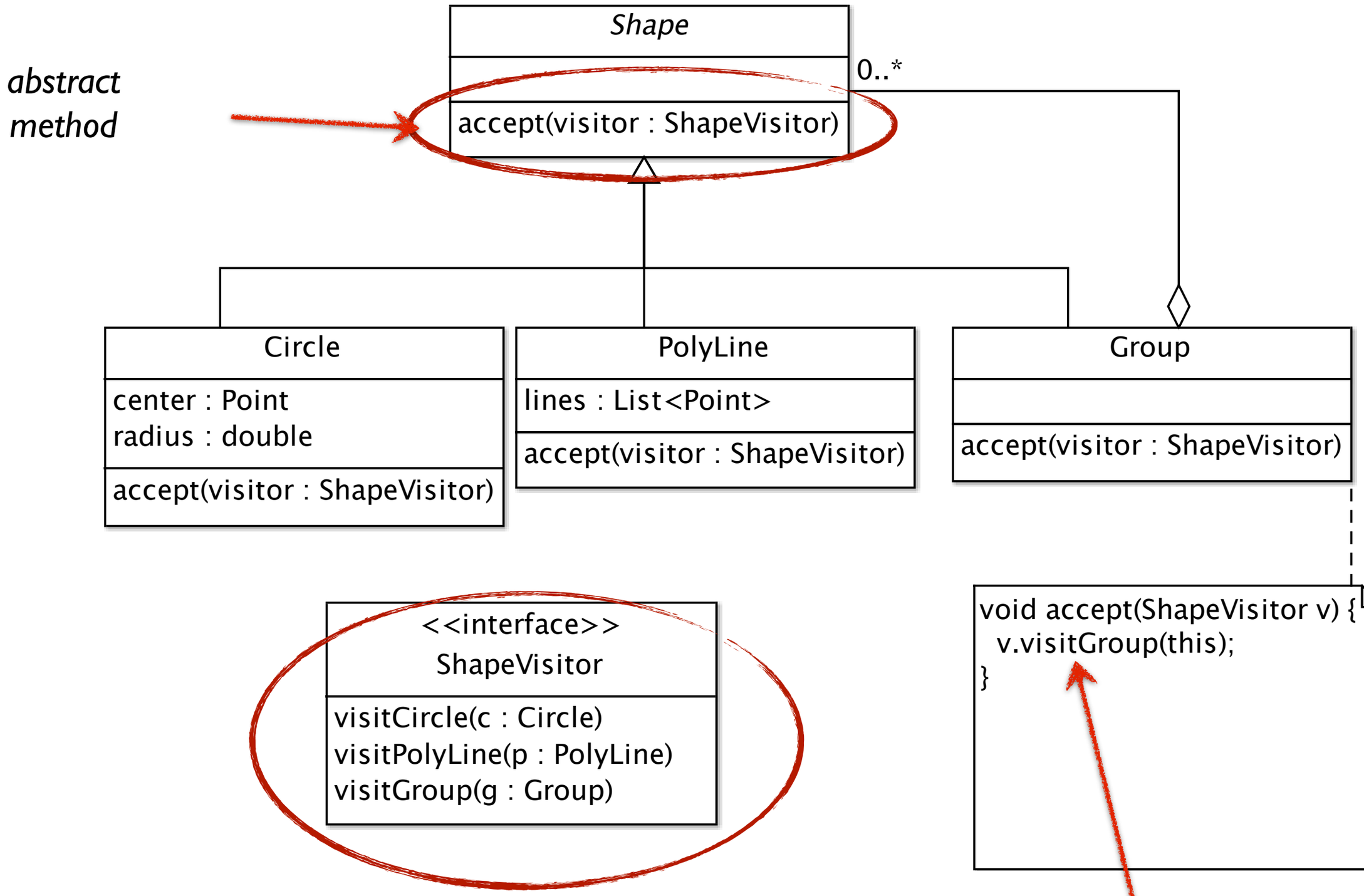


- how to add a new operation? do we always need to modify the whole class hierarchy?
- **we want a way to provide new manipulation without changing those classes.**

Visitor (Motivation)

- a **bad** solution : using instanceof :

```
public static int sum(Tree t)  {
  if (t instanceof EmptyTree) {
    return 0;
  } else {
    NonEmptyTree t1= (NonEmptyTree)t;
    return t1.getValue()+
              sum(t1.getLeftSon())+
              sum(t1.getRightSon());
  }
}
```



Visitor : implementation

calls the correct visitor method for this type...

Visitor

```
abstract class Tree {
  public abstract void accept(TreeVisitor v);
}
```

```
interface TreeVisitor {
  void visitEmptyTree(EmptyTree t);
  void visitNonEmptyTree(NonEmptyTree t);
}
```

```
class EmptyTree {
  public void accept(TreeVisitor v) {
     v.visitEmptyTree(this);
  }
}
```

Visitor

```
class NonEmptyTree {
  Tree left, right;
  int value;
  .....
  public void accept(TreeVisitor v) {
     v.visitNonEmptyTree(this);
  }
}
```

Visitor

```
class SumVisitor implements TreeVisitor {
  int result= 0;
  public void visitEmptyTree(EmptyTree t) {
  }
  public void visitNonEmptyTree(NonEmptyTree t) {
      result+= t.getValue();
      t.getLeft().accept(this);
      t.getRight().accept(this);
  }
  public int getResult() {
     return result;
  }
}
```

Using the visitor...

```
Tree t=.....;
SumVisitor v= new SumVisitor();
t.accept(v);
int result= v.getResult();
```

Reusable visitors...

```
abstract class InfixVisitor implements TreeVisitor {
   public final void visitNonEmptyTree(NonEmptyTree t) {
      t.getLeft().accept(this);
      processValue(t.getValue());
      t.getRight().accept(this);
   }
   public abstract void processValue(int value);
}
```

Sum visitor, revised

```
class SumVisitor extends InfixVisitor {
  int result= 0;
  public void processValue(int value) {
     result+= value;
  }
  public int getResult() {
    return result;
  }
}
```

Visitor: discussion

- fancy Object Oriented «switch» on a class hierarchy
- allows to extends functionnalities without changing the classes.
- possible use of inheritance to create generic visitors
- very often used with composites
- a way to simulate the functionnalities of some languages like Caml, which have «pattern matching» on values

Patterns as architectural guides

- Find the problems in the code which can be solved using a known pattern
- program using the said pattern use the «standard» names (e.g. «visit», «accept», «getInstance()» when possible
- beware of over-engineering (*YAGNI* : «**Y**ou **a**in't **g**onna **n**eed **i**t»)
- Refactoring to pattern

Pattern types (original book)

- **Creation** : pattern for creating new objects
- **Structural** : how classes and objects are connected together
- **Behavioural** : how classes interact, and which object gets which responsability

All patterns

- **Creational Patterns**
  - abstract factory, builder, factory Method, prototype, singleton
- **Structural Patterns**
  - adapter, bridge, composite, decorator, facade, flyweight, proxy
- **Behavioural patterns**
  - chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor.

A few more patterns

- Factory method
- Abstract factory
- Decorator
- Command
- Facade

Bibliography

- Eric Gamma et al. «Design Patterns, Elements of Reusable Object-Oriented Software»
- Martin Fowler, «Patterns of Enterprise Application Architecture»
- Joshua Kerievsky, «Refactoring to Patterns»
- http://en.wikipedia.org/wiki/Software_design_pattern : good starting point, with a list of patterns

Exercice 1

- Identify the patterns used in the following java classes
  - org.w3c.dom.Node and Element
  - LineNumberReader: which pattern : Bridge, Mediator, Decorator, Command ?
  - BorderFactory : is it AbstractFactory or Factory Method ?

Exercice

- Propose a (simple) system for *representing* HTML pages. We want to cover at least the following tags: p, h1, em, ul, li and img.
- Use a *visitor* to create an actual web page from the previous representation