Design of Software Systems (Ontwerp van SoftwareSystemen)

3 Design Patterns

Roel Wuyts
2015-2016

Warm-up Exercise

We need a design to be able to print different kinds of text (ASCII and PostScript) on different kinds of printers (ASCIIPrinter and PSPrinter).

ASCIIPrinters can only print ASCII text,
but PostscriptPrinters can print Postscript text as well as ASCIIText, after internally converting ASCIIText to Postscript text.

New types of printers and texts will be added in the near future.

Alexander’s patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without doing it the same way twice”

- Alexander uses this as part of the solution to capture the “quality without a name”

Illustrating Recurring Patterns...

Beijing (China)

Brussels (Belgium)

Zurich (Switzerland)

Singapore (Thailand)

Osio (China)

Alert!

Do not overreact seeing all these patterns!

Do not apply too many patterns!

Look at the trade-offs!

Most patterns makes systems more complex!

- but address a certain need.

As always: do good modeling.

- First start your design and note problems or difficulties,
- then propose multiple potential solutions with different trade-offs,
  - potentially using patterns,
- then take motivated decision

Design Patterns

A Design Pattern is a *pattern* that captures a solution to a *recurring design problem*

- It is **not** a pattern for implementation problems
- It is **not** a ready-made solution that has to be applied
  - It's still up to you !
  - You can simply use the pattern for inspiration,
  - Or only apply part of the design pattern
  - Remember the basic OO design principles and use them to weigh your design

Adapt to suit taste, allergies, nr. of people, available ingredients, …

Ingredients

- 8 endives, intact but cored
- 4 tablespoons butter
- 8 slices low-sodium ham
- 2 tablespoons all-purpose flour
- 2 cups whole milk
- Freshly cracked black pepper, for seasoning
- 1/8 teaspoon freshly grated nutmeg
- 8 ounces Gruyere cheese, grated

Directions

Special Equipment: Steaming basket

Serving Suggestion: French baguette, sliced

Preheat oven to 350 to 375 degrees F.

In a large pot fitted with a steaming basket, bring 1-inch of water to a boil. Place the endives in the basket, cover, and let cook until very soft, about 10 to 15 minutes. Transfer to a colander and let drain, pushing down on the endives with a clean kitchen towel until as much of the water as possible has been expelled. Do not mush the endives!

In a large frying pan over medium-high heat, melt 2 tablespoons of the butter. When the foam has subsided, add the endives and cook, turning occasionally, until brown and caramelized on all surfaces. Remove from heat and wrap each endive in 1 slice of the ham. Set aside.

In a small saucepan over medium heat, melt the remaining 2 tablespoons of butter. When the foam has subsided whisk in the flour and cook 1 minute, being careful not to brown the flour. Whisk in the milk in a slow, steady stream. Bring to a boil whisking constantly, then reduce heat to medium-low and simmer until thickened, about 8 minutes. Season with a generous amount of pepper and the nutmeg.

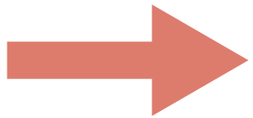
Spread about 1 cup of the sauce over the bottom of a 9 by 13-inch glass or ceramic baking dish, then arrange the ham-wrapped endives on top in a single layer. Cover with the remaining sauce and the cheese. Bake until the cheese is melted and the sauce is bubbling, about 30 minutes. Turn on the broiler, transfer the pan to the top rack, and broil until the cheese has patches of golden goodness - about 2 minutes. Serve hot with generous amounts of sauce and baguette slices.

Recipe courtesy of Amy Finley

Design Patterns

Example:

- "We are implementing a drawing application. The application allows the user to draw several kinds of figures (circles, squares, lines, polymorphs, bezier splines). It also allows to group these figures (and ungroup them later). Groups can then be moved around and are treated like any other figure."



Look at *Composite* Design Pattern

Pattern structure

A design pattern is a kind of blueprint

Consists of different parts

- All of these parts make up the pattern!
- When we talk about the pattern we therefore mean all of these parts together
  - not only the class diagram...

Tip: remember this for the exam – know your complete pattern

Why Patterns?

Smart

- Elegant solutions that a novice would not think of

Generic

- Independent on specific system type, language
  - Although biased towards statically-typed class-based OO languages (C++, Java, …)

Well-proven

- Successfully tested in several systems

Simple

- Combine them for more complex solutions

GoF Design Pattern Book

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang-of-Four (GoF))

Book is still very relevant today but (depending on edition):

- Original book uses OMT notation (analogous to UML)
- illustrations are in C++
  - Principles valid across OO languages!
  - Versions of book exists with illustrations in Java, …

GoF Design Pattern Book

23 Design Patterns

Classification

- according to purpose
- according to problems they solve (p. 24-25)
- according to degrees of freedom (table 1.2, p. 30)

Goal is to make it easy to find a pattern for your problem

Design Pattern Relationships

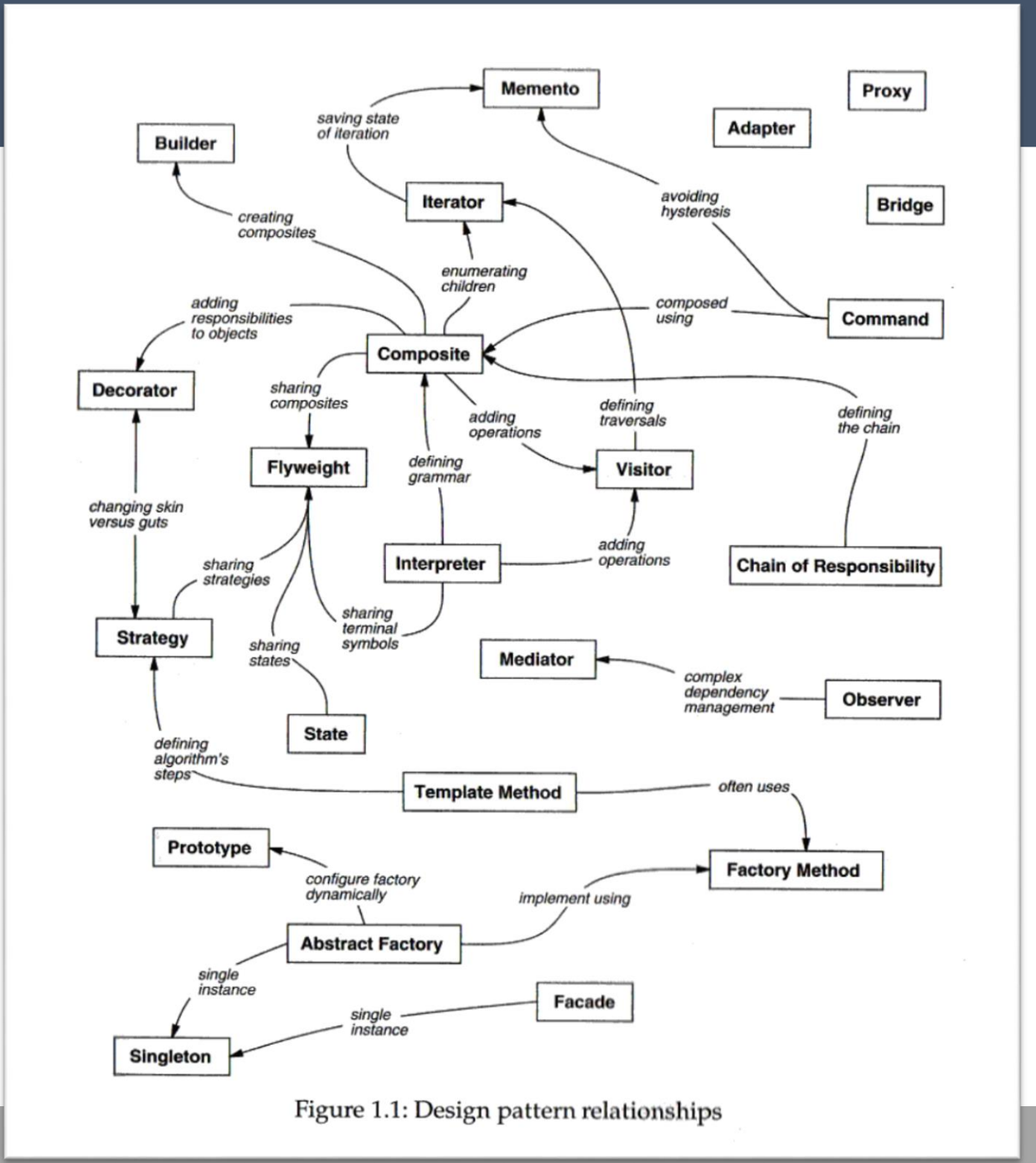


Figure 1.1: Design pattern relationships

Visitor

Visitor

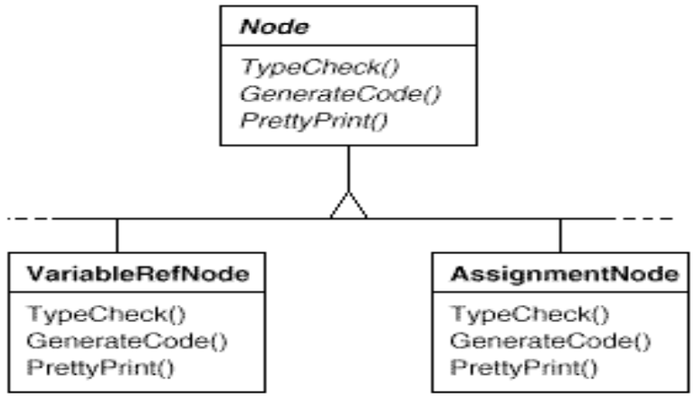
Category

- Behavioral

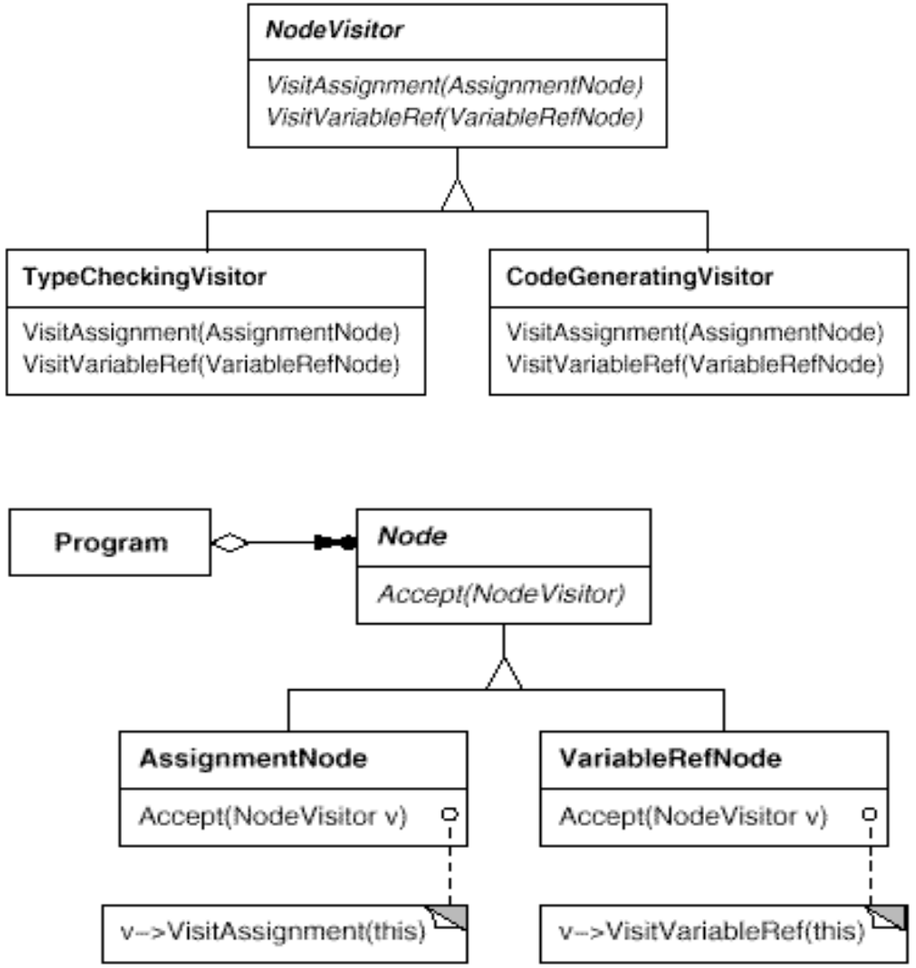
Intent

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation



Motivation (cont)



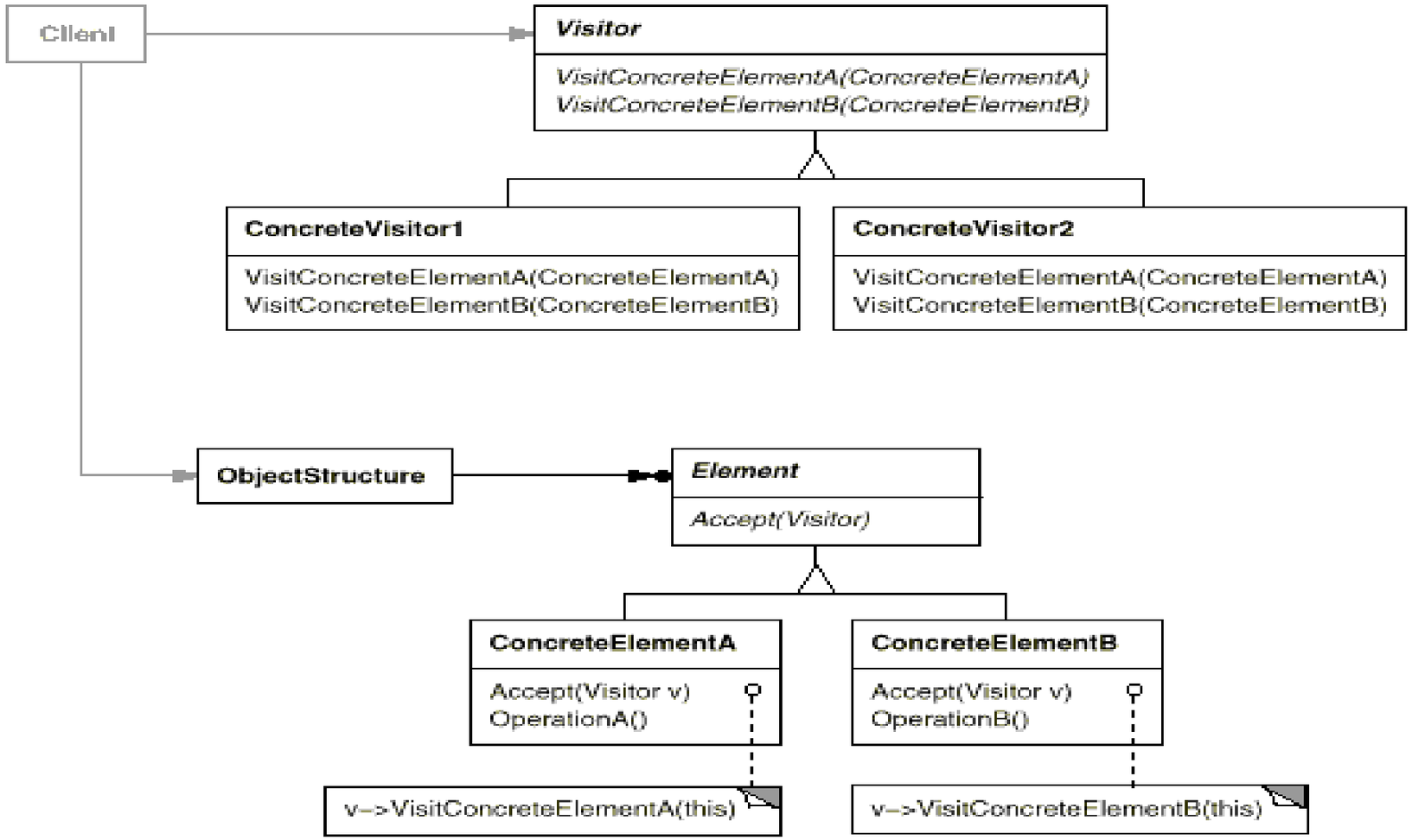
Applicability

An object structure contains many classes of objects with differing interfaces and you want to perform operations on these objects that depend on their concrete classes.

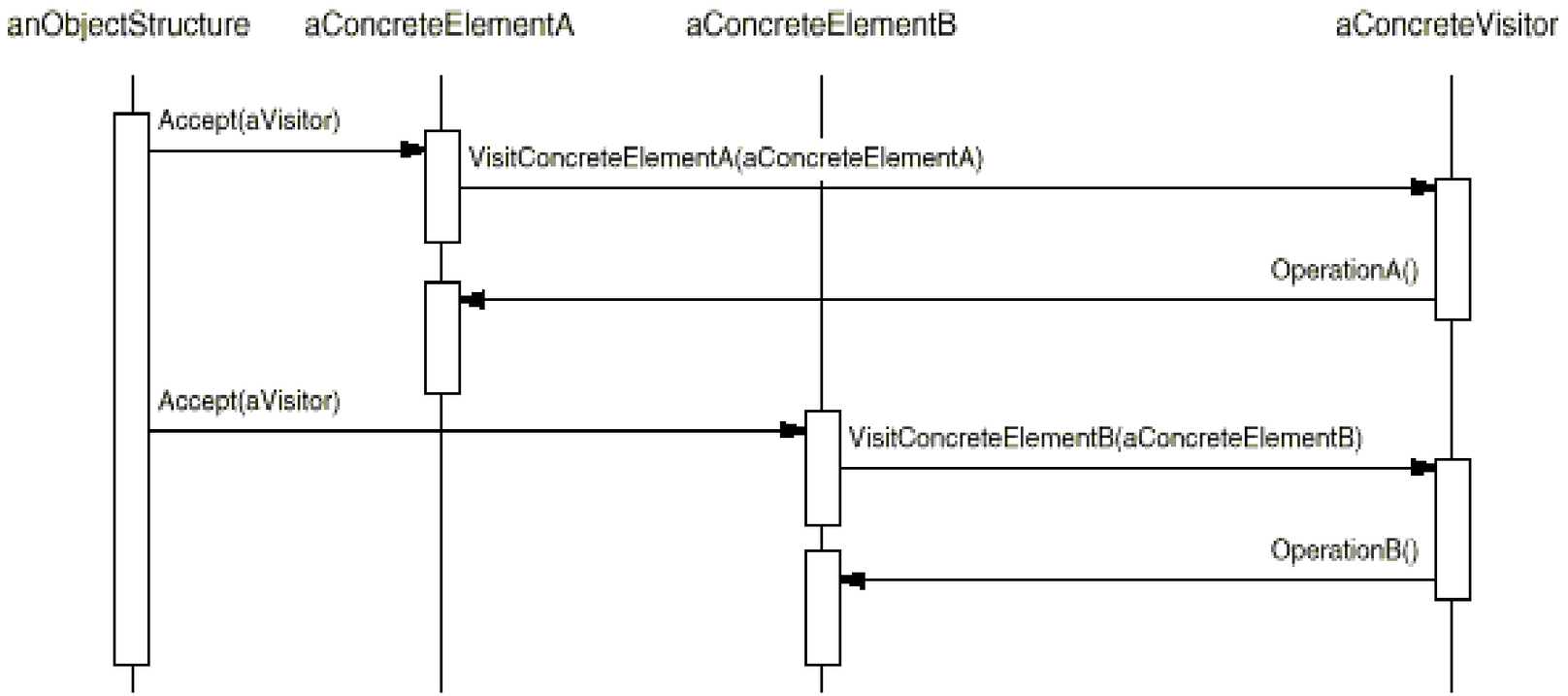
Many distinct and unrelated operations need to be performed on objects in an object structure an you want to avoid “polluting” their classes with these operations.

The classes defining the object structure rarely change but you often want to define new operations over the structure.

Structure



Sequence



Cfr. *Double Dispatch* - this is key to this pattern in order to link concrete elements and concrete visitors !

Participants

Visitor

- Declares a Visit operation for each class of ConcreteElement in the object structure.
- The operations name and signature identifies the class that sends the Visit request.

ConcreteVisitor

- Implements each operation declared by Visitor.
- Each operation implements a fragment of the algorithm for the corresponding class of object in the object structure.
- Provides the context for the algorithm and stores its state (often accumulating results during traversal).

...

Participants (cont)

Element

- Defines an accept operation that takes a Visitor as an argument.

ConcreteElement

- Implements an accept operation that takes a visitor as an argument.

ObjectStructure

- Can enumerate its elements.
- May provide a high-level interface to allow the visitor to visit its elements.
- May either be a Composite or a collection such as a list or set.

Collaborations

A client that uses the visitor pattern must create a ConcreteVisitor object and then traverse the object structure visiting each element with the Visitor.

When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation.

Consequences

Makes adding new operations easy.

- a new operation is defined by adding a new visitor (in contrast, when you spread functionality over many classes each class must be changed to define the new operation).

Gathers related operations and separates unrelated ones.

- related behavior is localised in the visitor and not spread over the classes defining the object structure.

Consequences (cont)

Adding new ConcreteElement classes is hard.

- each new ConcreteElement gives rise to a new abstract operation in Visitor and a corresponding implementation in each ConcreteVisitor.

Allows visiting across class hierarchies.

- an iterator can also visit the elements of an object structure as it traverses them and calls operations on them but all elements of the object structure then need to have a common parent. Visitor does not have this restriction.

Consequences (cont)

Accumulating state

- Visitor can accumulate state as it proceeds with the traversal. Without a visitor this state must be passed as an extra parameter of handled in global variables.

Breaking encapsulation

- Visitor’s approach assumes that the ConcreteElement interface is powerful enough to allow the visitors to do their job. As a result the pattern ofthen forces to provide public operations that access an element’s internal state which may compromise its encapsulation.

Known Uses

In the Smalltalk-80 compiler.

In 3D-graphics: when three-dimensional scenes are represented as a hierarchy of nodes, the Visitor pattern can be used to perform different actions on those nodes.

Visitor Pattern

So, we've covered the visitor pattern as found in the book

- Are we done?

Decisions, decisions …

visit(OperationA a)

visit(OperationB b)

vs

visitOperationA(OperationA a)

visitOperationB(OperationB b)

Short Feature...

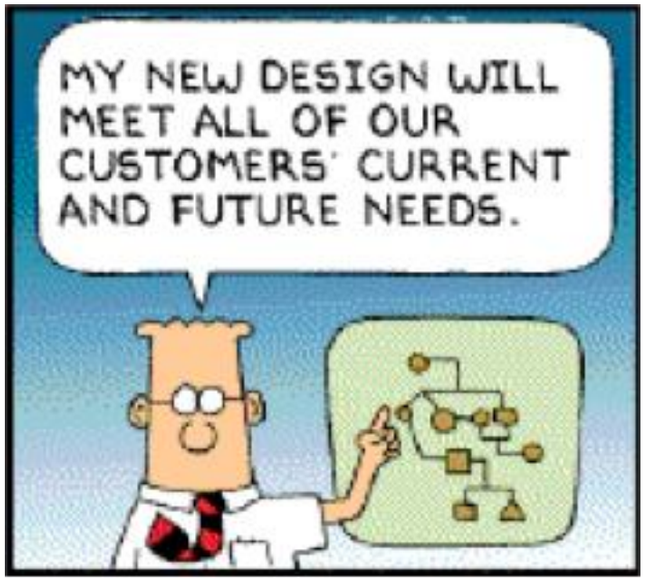
What is the result of the following expression?

```
class A {
      public void m(A a) { System.out.println("1"); }
}

class B extends A {
      public void m(B b) { System.out.println("2"); }
      public void m(A a) { System.out.println("3"); }
}


B b = new B();
A a = b;
a.m(b);
```

Main Feature...



Visiting all Elements in the CDT Parsetree

```
public abstract class ASTVisitor {

        public int visit(IASTTranslationUnit tu)                          { return PROCESS_CONTINUE; }

        public int visit(IASTName name)                                   { return PROCESS_CONTINUE; }

        public int visit(IASTDeclaration declaration)                     { return PROCESS_CONTINUE; }

        public int visit(IASTInitializer initializer)                     { return PROCESS_CONTINUE; }

        public int visit(IASTParameterDeclaration parameterDeclaration)   { return PROCESS_CONTINUE; }

        public int visit(IASTDeclarator declarator)                       { return PROCESS_CONTINUE; }

        public int visit(IASTDeclSpecifier declSpec)                      { return PROCESS_CONTINUE; }

        public int visit(IASTExpression expression)                       { return PROCESS_CONTINUE; }

        public int visit(IASTStatement statement)                         { return PROCESS_CONTINUE; }

        public int visit(IASTTypeId typeId)                               { return PROCESS_CONTINUE; }

        public int visit(IASTEnumerator enumerator)                       { return PROCESS_CONTINUE; }

        public int visit( IASTProblem problem )                           { return PROCESS_CONTINUE; }

}
```

To Arms! The Short Feature is Attacking the Main Feature

Advanced Visitor Discussions

When looking more closely at the visitor and its implementation, we can discuss a number of things in more detail:

- Who controls the traversal?
- What is the granularity of the visit methods?
- Does there have to be a one-on-one correspondence between Element classes and visit methods ?
- …

Composite

Composite

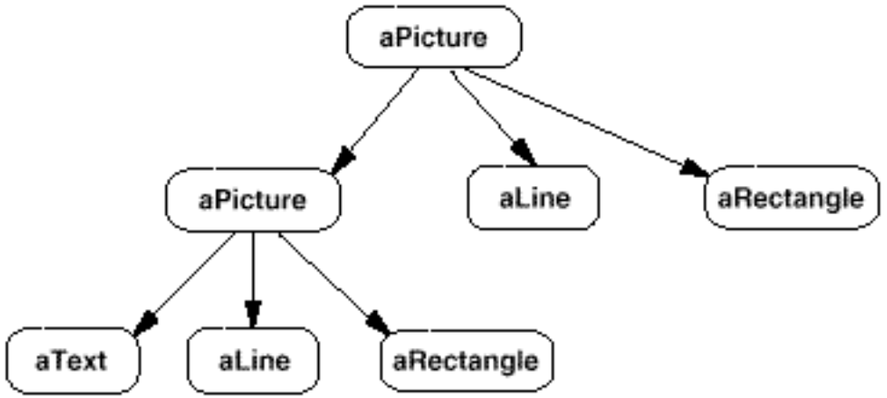
Category

- Structural

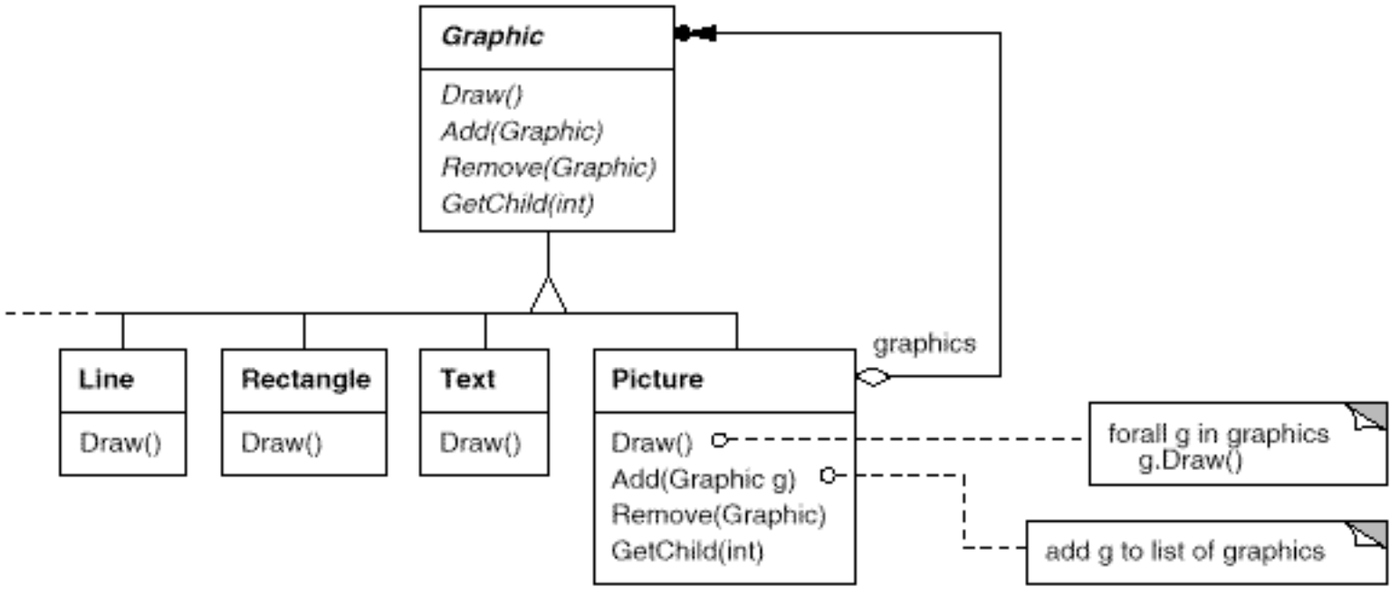
Intent

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation



Motivation (cont)

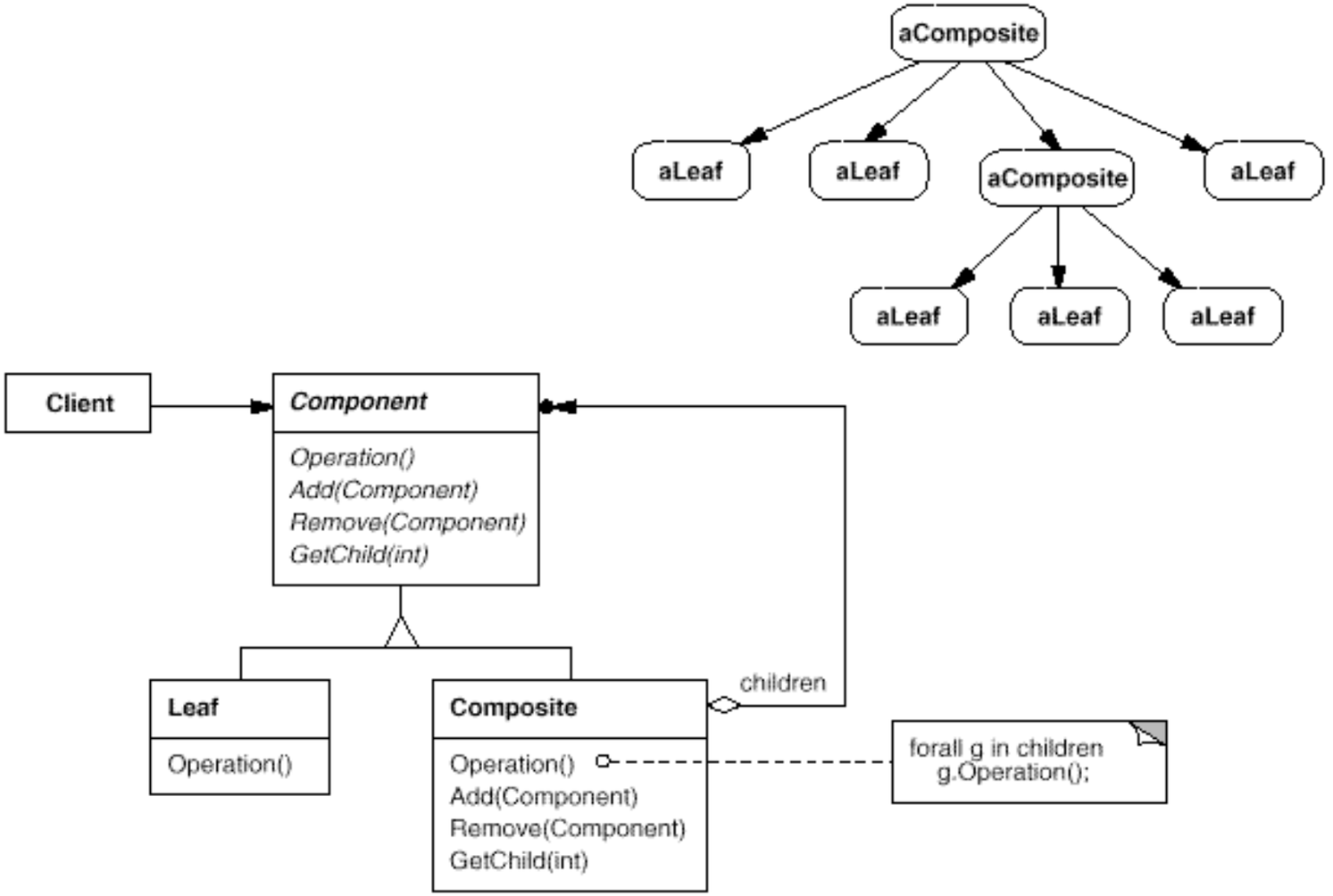


Applicability

Use the Composite Pattern when:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure



Participants

Component

- Declares the interface for objects in the composition.
- Implements default behaviour for the interface common to all classes, as appropriate.
- Declares an interface for accessing and managing its child components.

Leaf

- Represents leaf objects in the composition. A leaf has no children.
- Defines behaviour for primitive objects in the composition.

Participants (cont)

Composite

- defines behaviour for components having children.
- stores child components.
- implements child-related operations in the Component interface.

Client

- manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. Leaves handle the requests directly. Composites forward requests to its child components.

Consequences

Defines class hierarchies consisting of primitive and composite objects.

Makes the client simple. Composite and primitive objects are treated uniformly (no cases).

Eases the creation of new kinds of components.

Can make your design overly general.

Known Uses

Can be found in almost all object oriented systems.

The original View class in Smalltalk Model / View / Controller was a composite.

Questions

How does the Composite pattern help to consolidate system-wide conditional logic?

Would you use the composite pattern if you did not have a part-whole hierarchy? In other words, if only a few objects have children and almost everything else in your collection is a leaf (a leaf that has no children), would you still use the composite pattern to model these objects?

Patterns Catalogue

- Command
- Decorator
- Strategy
- Factory Method
- Abstract Factory
- Singleton
- Proxy
- Adapter
- Observer
- Chain of Responsibility
- FlyWeight
- Facade

See Design pattern books,
Or the patterns reference on website

Wrap-up

Architectures

"can't be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed." (Alexander)

- patterns describe such building blocks
- applying them implicitly changes the overall structure (architecture)
- whether it is on classes, components, or people

Conclusion

Can you answer this?

- How does Strategy improve coupling and cohesion?
- Does Abstract Factory says the same than the Creator GRASP Pattern?
- Can you give examples of patterns that can be used together ?
- When does it make sense to combine the Iterator and the Composite Pattern ?

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.