Ontwerp van SoftwareSystemen

4 Design Patterns

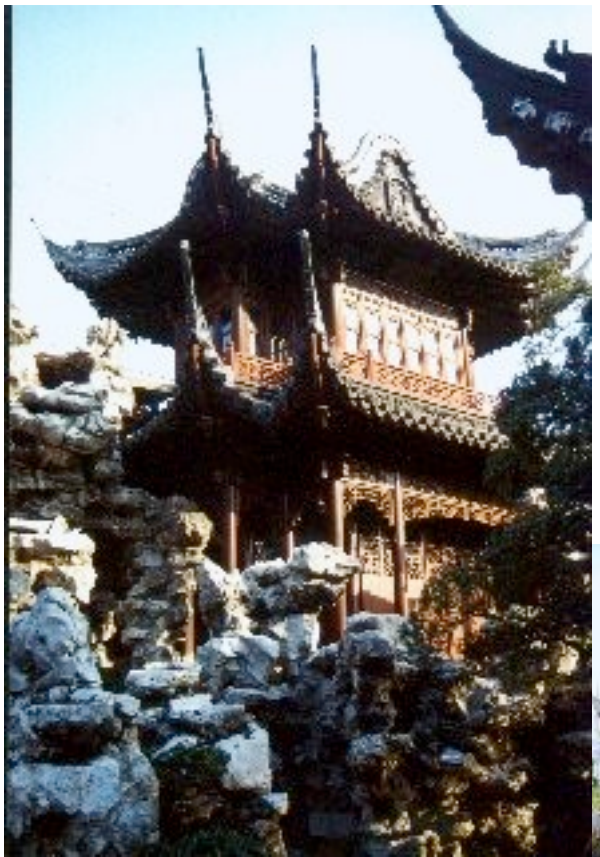
Roel Wuyts

OSS 2013-2014

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...



Alert!

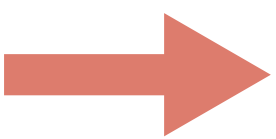
- Do not overreact seeing all these patterns!
- Do not apply too many patterns!
- Look at the trade-offs!
- Most patterns makes systems more complex!
  - but address a certain need.
- As always: do good modeling.
  - then see whether patterns can help,
  - and where.

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
    - cfr Rational Modeler, where patterns are available as preconstructed class diagrams, even though in literature the class diagrams are to illustrate the pattern!

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

Patterns in Software Design

- A design pattern is a description of communicating objects and classes that are customized to solve a general design problem in a particular context.

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...

Why Patterns?

- Smart
  - Elegant solutions that a novice would not think of
- Generic
  - Independent on specific system type, language
    - Allthough slightly biased towards C++
- 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:
  - uses OMT notation (analogous to UML)
  - illustrations are in C++
    - Principles valid across OO languages!

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)

Classification according to purpose

	Creational	Structural	Behavioral
Class (static)	Factory Method	Class Adapter	Interpreter Template Method
Object (dynamic)	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Classification according to problems

- Dependence on specific classes:
  - => Abstract Factory, Factory Method, Prototype
- Dependence op specific operations:
  - => Chain of Responsibility, Command
- Dependence on hardware and/or software platforms:
  - => Abstract Factory, Bridge
- Dependence on object representation or implementation:
  - => Abstract Factory, Bridge, Memento, Proxy
- Dependence on algorithms:
  - => Builder, Iterator, Strategy, Template Method, Visitor
- Tight Coupling:
  - => Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer
- Problems with enhancing functionality through subclassing:
  - => Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Impossibility of easily changing classes:
  - => Adapter, Decorator, Visitor

Classification according to problems

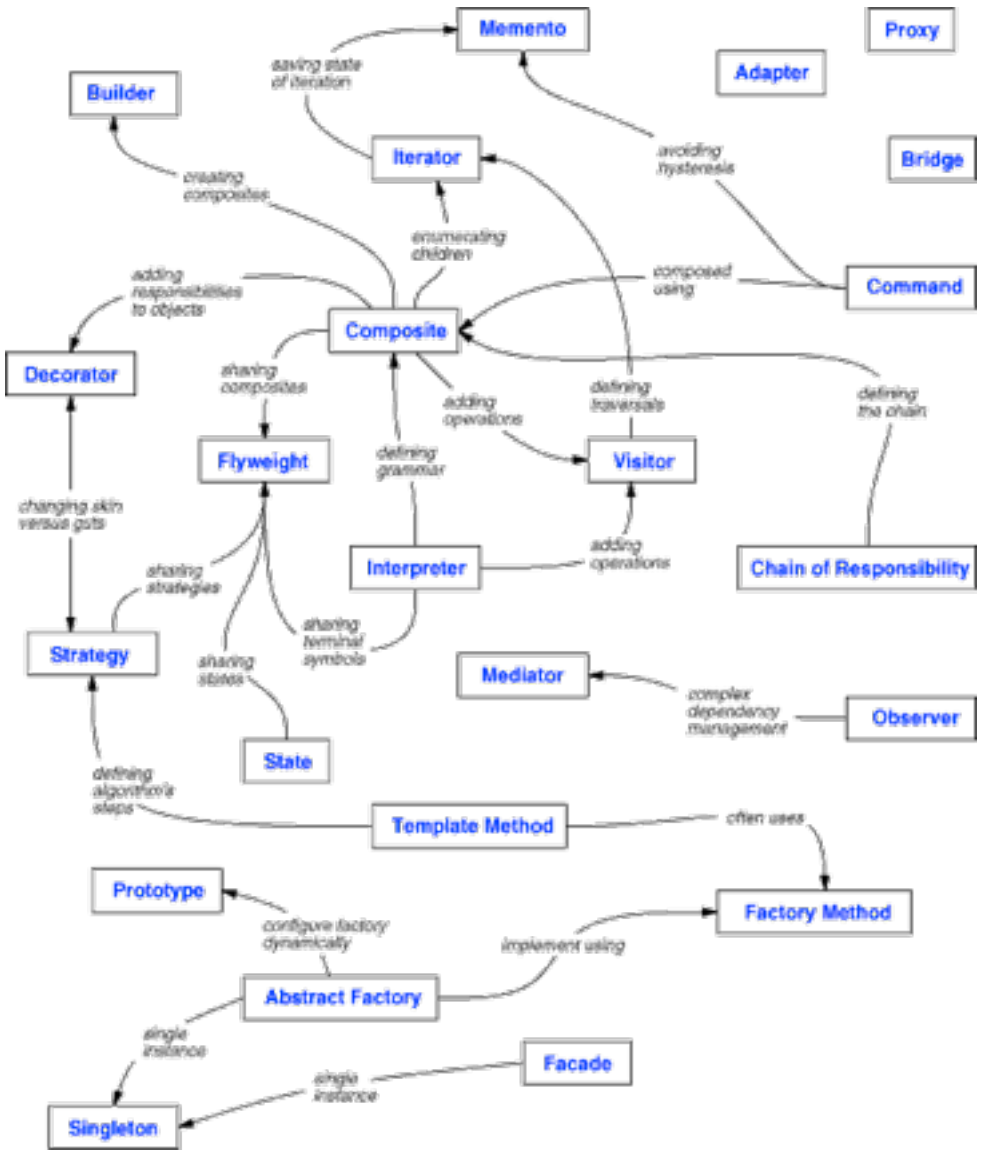
- Different patterns are typically applicable
- Ex.: dependance on algorithms
  - Algorithms are source of evolution (extend, replace, optimize, …)
  - Classes depending on algorithms are therefore unstable
  - So algorithms amenable to change have to be encapsulated
- Design patterns that can help do this:
  - Builder, Iterator, Strategy, Template Method, Visitor

Classification according to degrees of freedom

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location

Purpose	Design Pattern	Aspect(s) That Can Vary
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Design Pattern Relationships

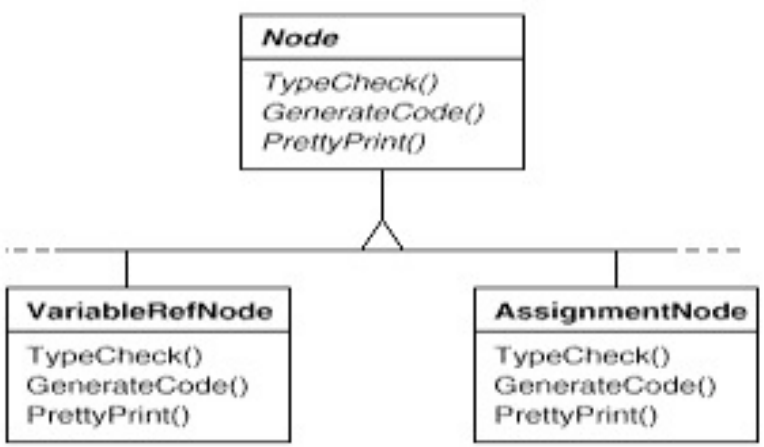


Example Pattern: Visitor

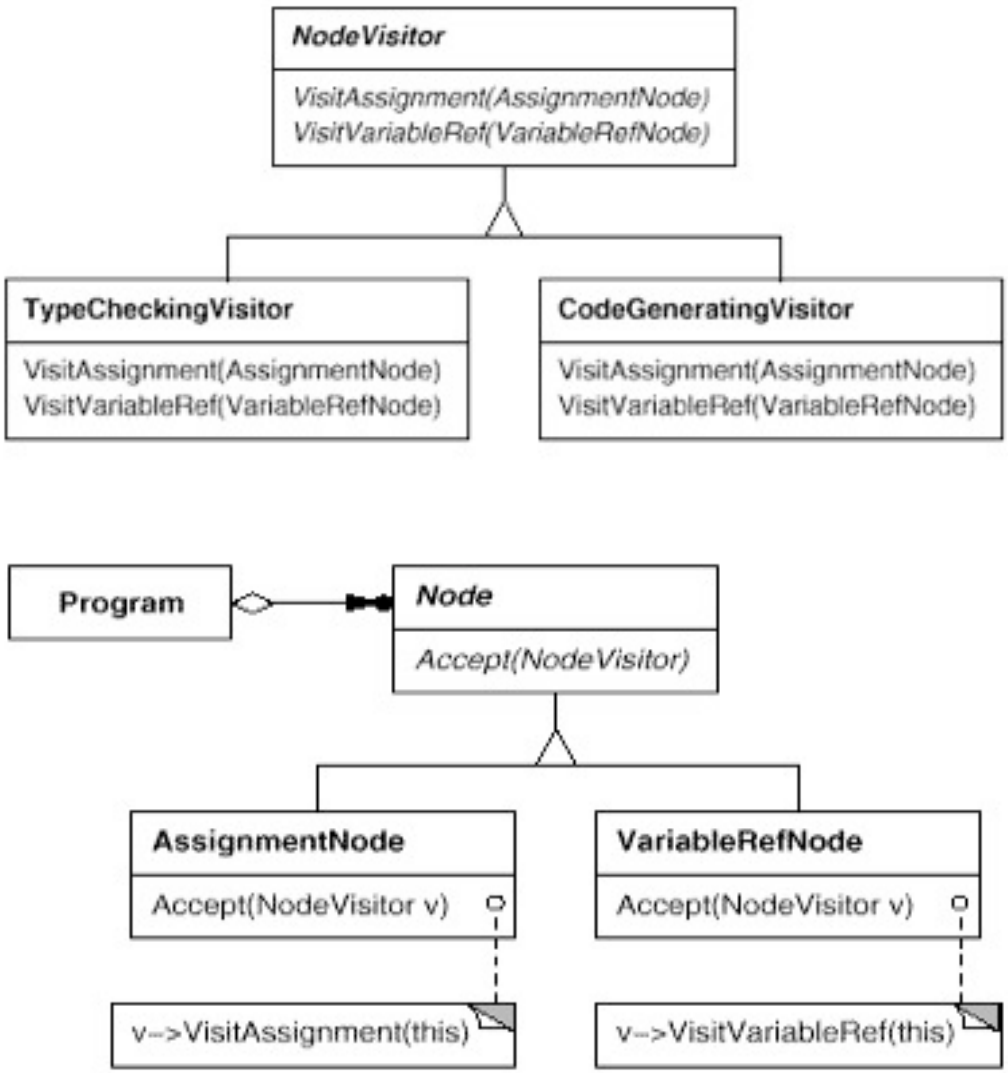
Visitor

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)

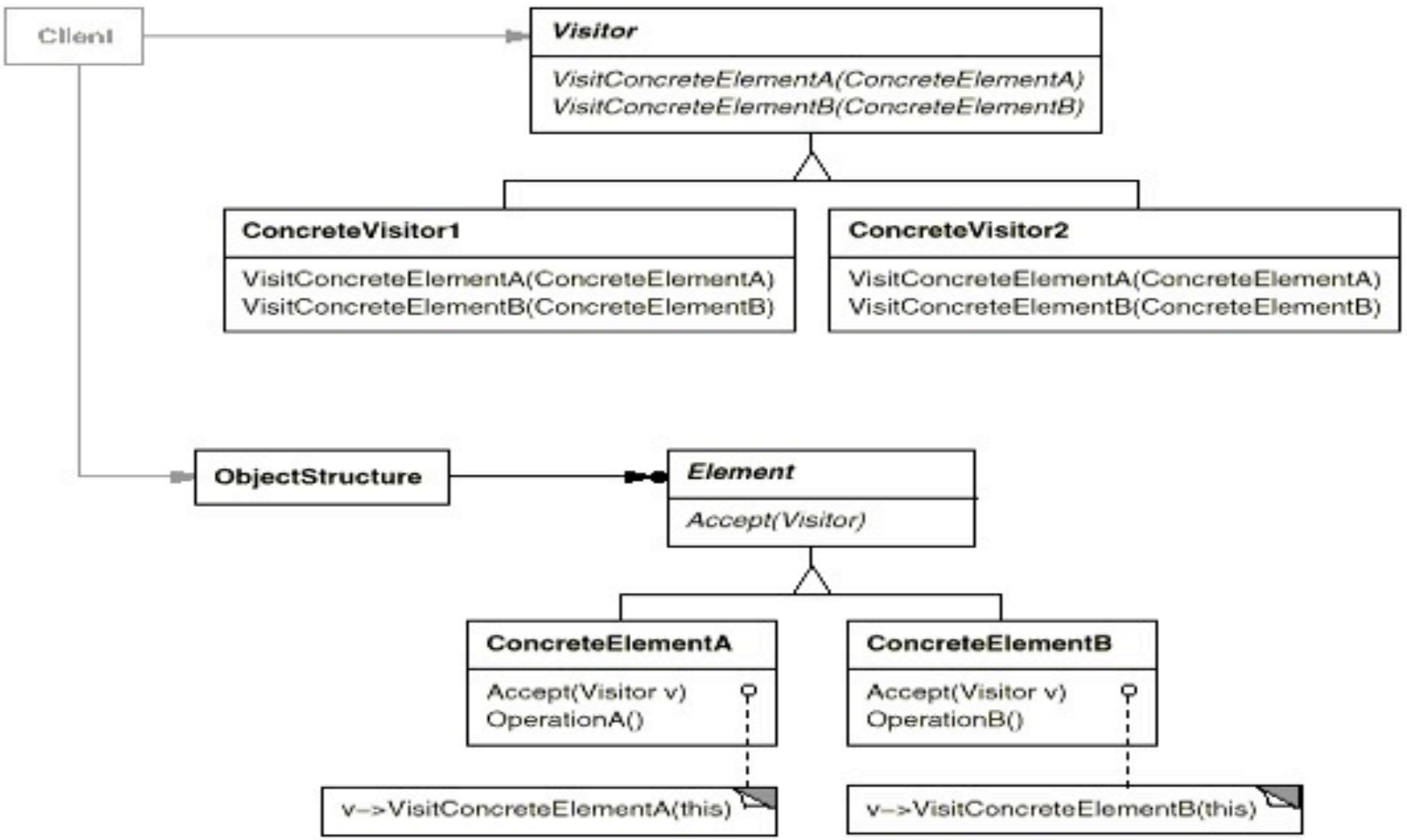


Applicability

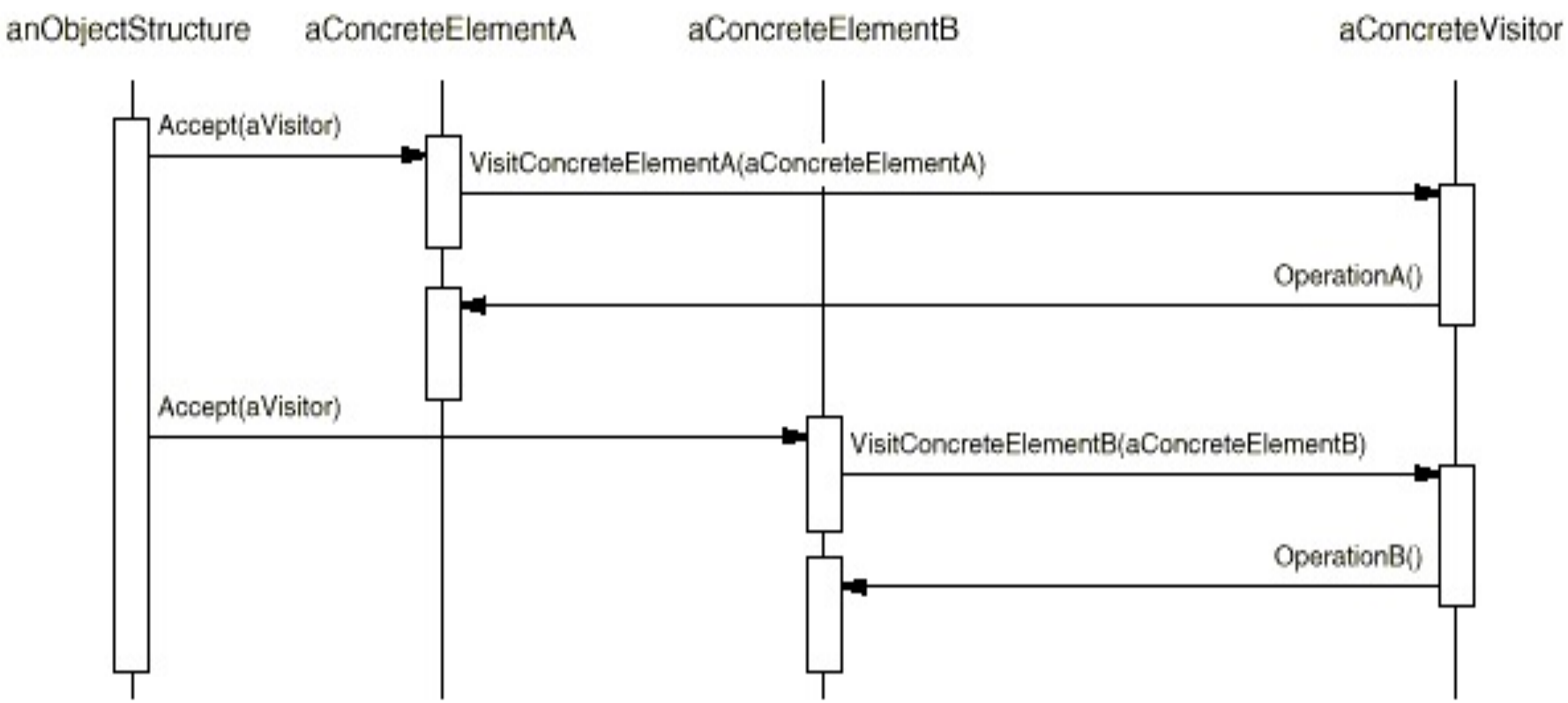
- An object structure contains many classes of objects with differing interfaces and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure an you want to avoid “polluting” 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

- Structure



Sequence



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

Participants (cont)

- 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.

Example Code

```
abstract class Equipment {
    String name;

    public String name(){
     return name;
    }

    abstract int power();
    abstract int netPrice();
    abstract void add(Equipment e);
    abstract void remove(Equipment e);

    abstract void accept(EquipmentVisitor v);

    protected Equipment(String n){
     name = n;
    }
}
```

Example Code

```
abstract class EquipmentVisitor {

    public abstract void visitFloppydisk(Floppydisk f);

    public abstract void visitCard(Card c);

    public abstract void visitChassis(Chassis c);

    public abstract void visitBus(Bus b);

    public abstract void visitCabinet(Cabinet c);

}
```

Example Code

```
public class Floppydisk extends Equipment {
    public Floppydisk(String name){
     super(name);
    }

    public int power(){
     return 60;
    }

    public int netPrice(){
     return 50;
    }

    public void accept(EquipmentVisitor v){
     v.visitFloppydisk(this);
    }

    public void add(Equipment e){}
    public void remove(Equipment e){}
}
```

Example Code

```
public class Cabinet extends CompositeEquipment {

    public Cabinet(String name){
     super(name);
    }

    public int power(){
     return 0;
    }

    public int netPrice(){
     return 60;
    }

    public void accept(EquipmentVisitor v){
     Iterator i = createIterator();
     while(i.hasNext()){
           Equipment e = (Equipment) i.next();
           e.accept(v);
     }
     v.visitCabinet(this);
    }
}
```

Example Code

```
public class Main {
    public static void main(String[] args) {
     Cabinet cabinet = new Cabinet("PC Cabinet");
     Chassis chassis = new Chassis("PC Chassis");
     cabinet.add(chassis);
     Bus bus = new Bus("MCA bus");
     bus.add(new Card("NetworkCard"));
     chassis.add(bus);
     chassis.add(new Floppydisk("3.5 Floppy"));
     PricingVisitor p = new PricingVisitor();
     cabinet.accept(p);
     InventoryVisitor i = new InventoryVisitor();
     cabinet.accept(i);

     System.out.println("The computer contains: ");
     i.getInventory();
     System.out.println("The price is " + p.getTotal());
    }
}
```

Example Code

```
The computer contains:

NetworkCard

MCA bus

3.5 Floppy

PC Chassis

PC Cabinet

The price is 350
```

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?

visit(OperationA a)

visit(OperationB b)

vs

visitOperationA(OperationA a)

visitOperationB(OperationB b)

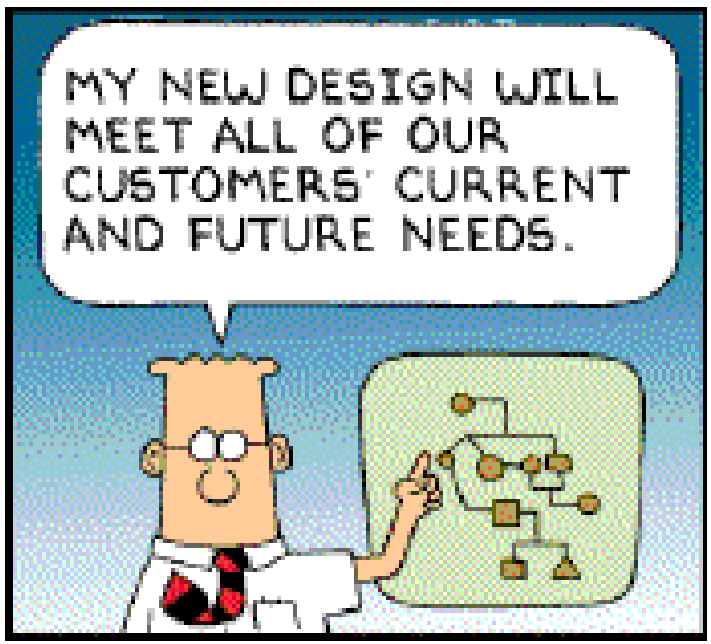
Short Feature...

What is the result of the following expression ?

```
class A {
    void m(A a) { println("1"); }
}
class B extends A {
    void m(B b) { println("2"); }
    void m(A a) { 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!

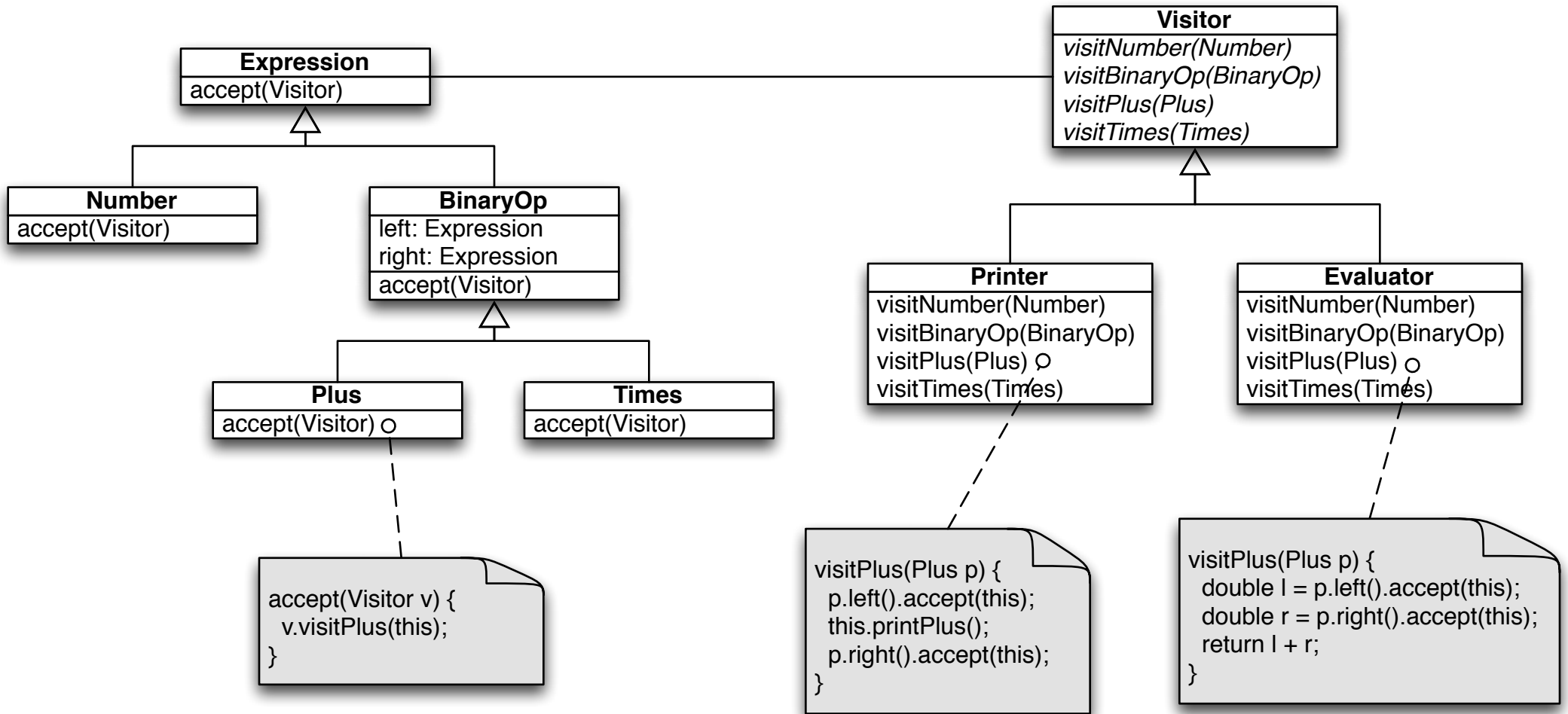
Advanced Visitor Discussions

- 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?
  - Implementation tricks

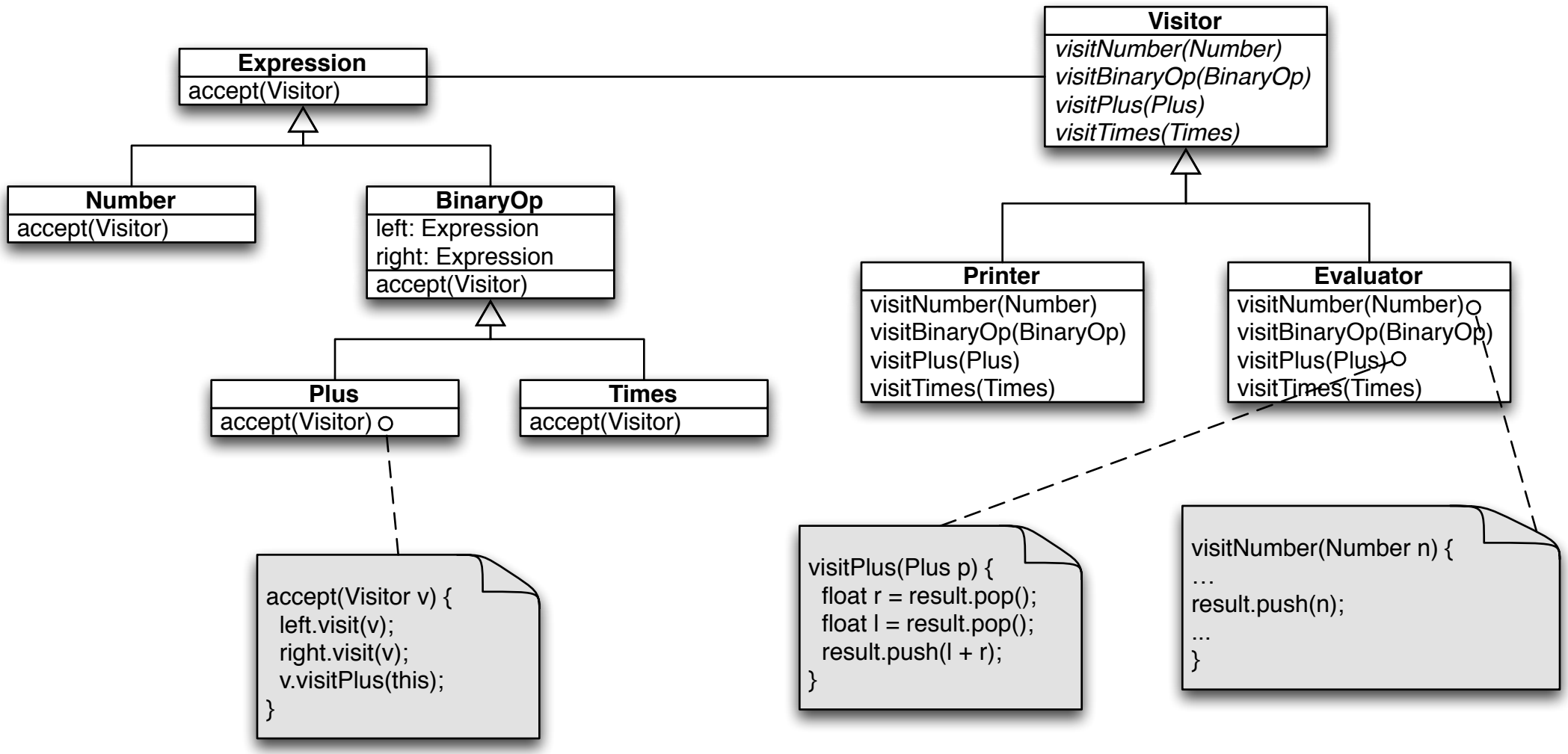
Controlling the traversal

- Somewhere in the visitor, items are traversed.
- Different places where the traversal can be implemented:
  - in the visitor
  - on the items hierarchy

Traversal on the Visitor



Traversel on the items

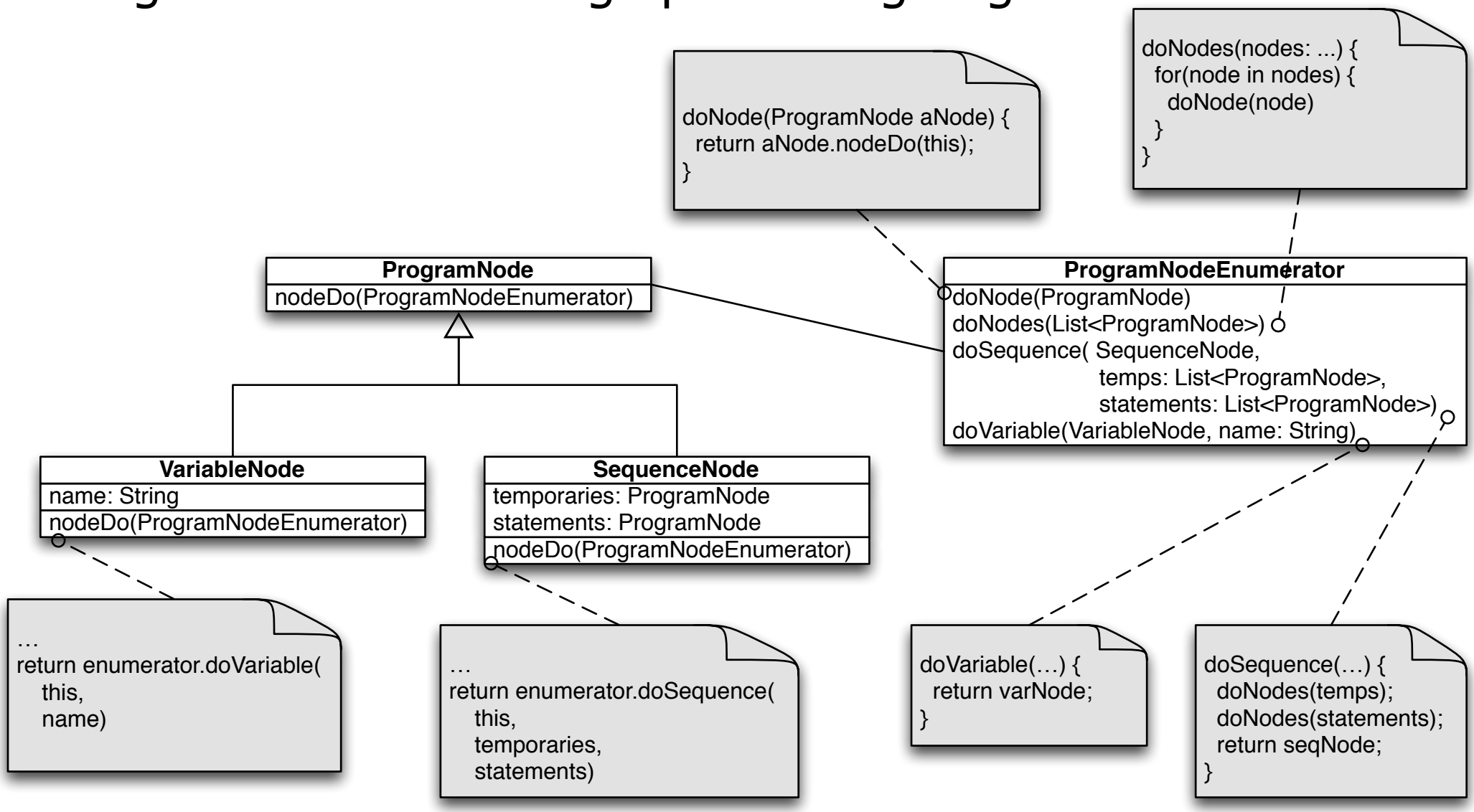


Granularity of Visit Methods

- Sometimes you can pass context information with the visit methods
- So visitors have more information for implementing their operations

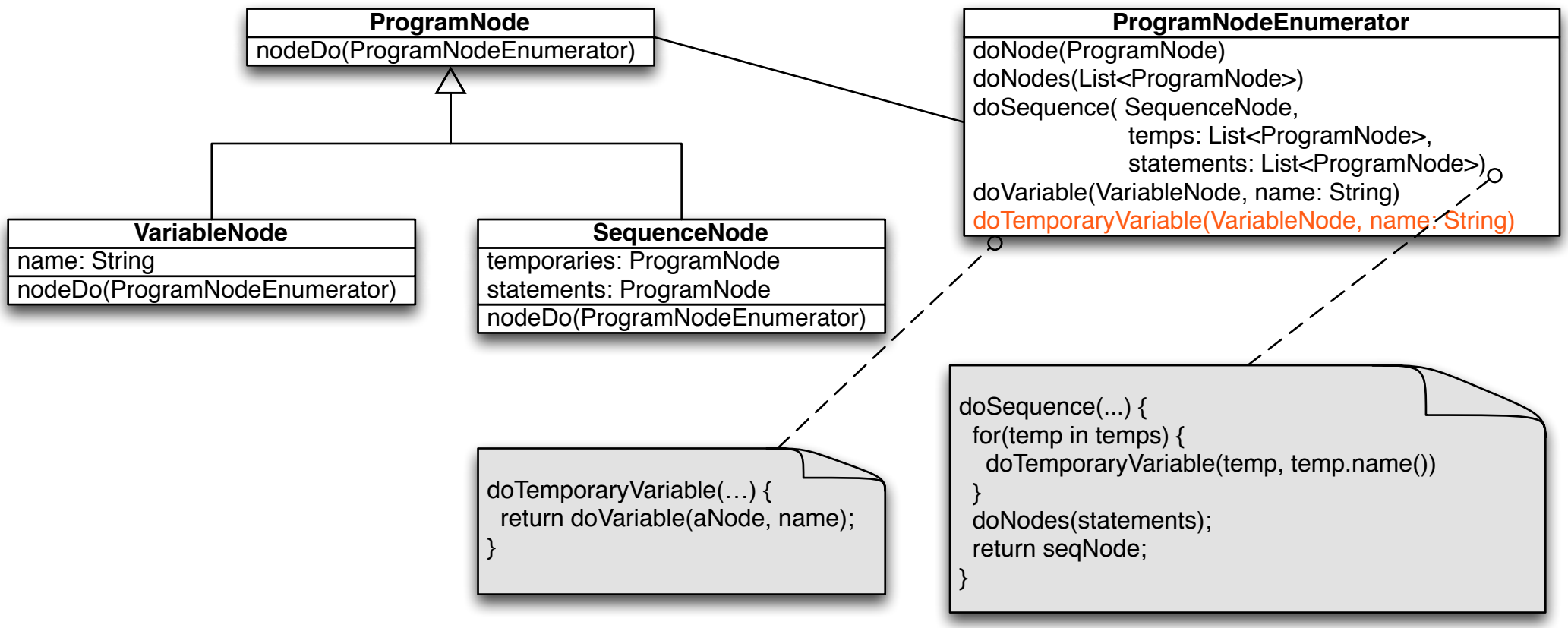
Granularity of Visit Methods

- Regular case: nothing special is going on



Refined granularity

- More semantics in the visitor
  - no TemporaryVariableNode, but specific visit method (cfr. Pure Fabrication)

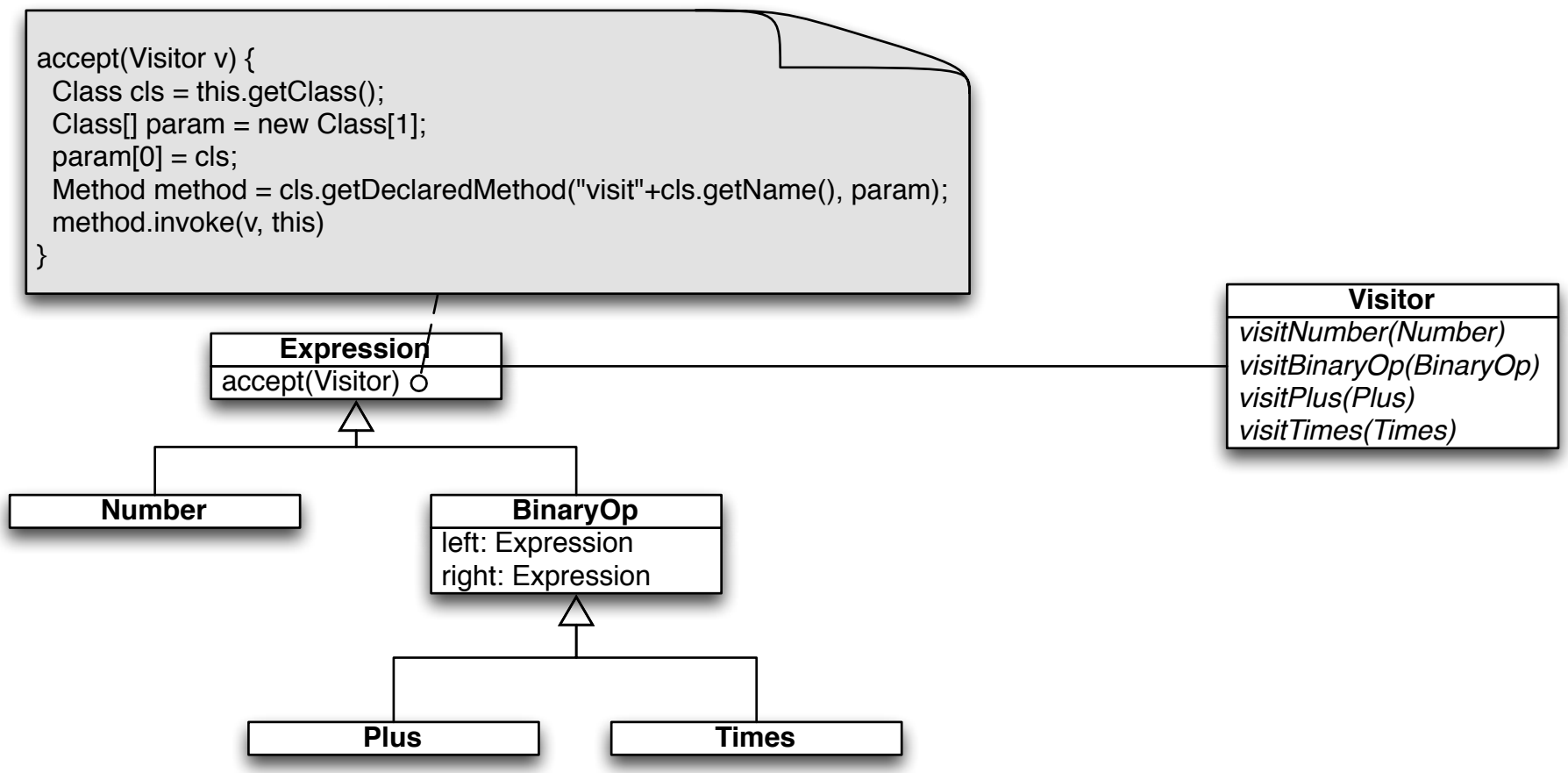


Implementation tricks

- You can implement it as we have shown before.
- But notice the general structure of the methods!
- This can be taken as advantage:
  - code can be generated for a visitor.
  - the method can be performed/invoked
- But take care:
  - only works when there is a full correspondence.
  - can make the code hard to understand.

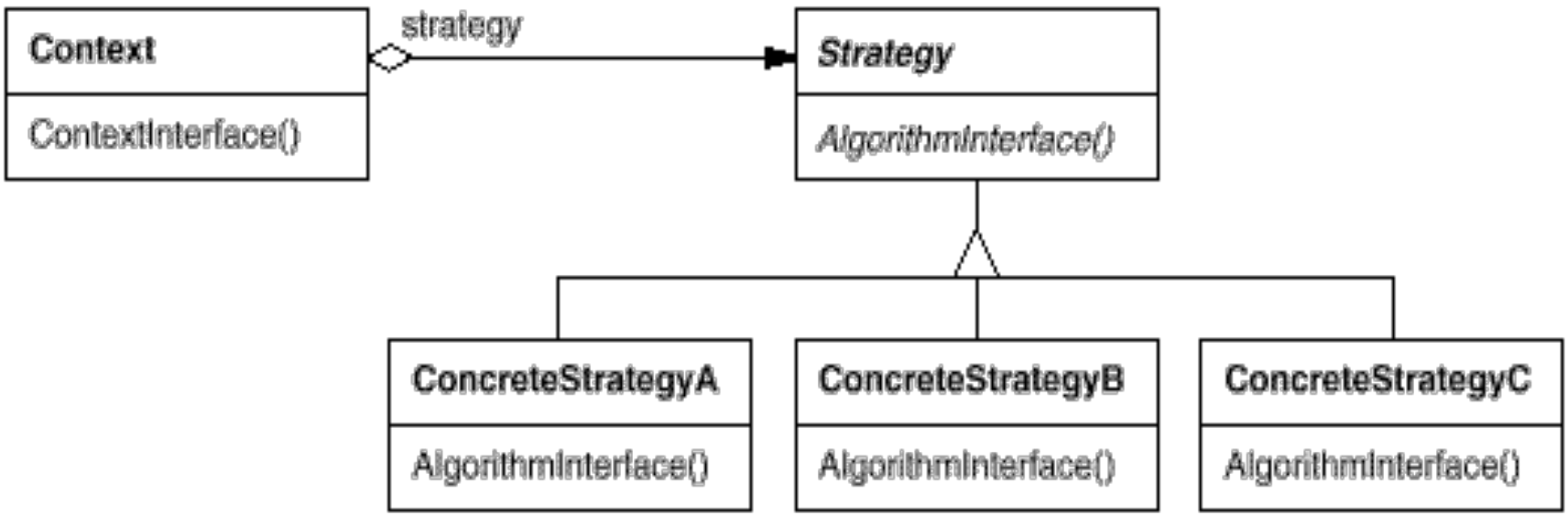
Using Reflection (*invoke*)

- Uses reflection to implement accept() only once
  - instead of hardcoding the bodies of accept methods over and over again



Strategy

Het Strategy (Policy) Patroon (315)



- Doel: onafhankelijkheid van algoritmen door inkapseling
  - => variatie van algoritme mogelijk
- Implementatie-overwegingen:
  - koppeling Context – Strategy:
    - data als parameters naar Strategy doorgeven
    - Context als parameter naar of als verwijzing vanuit Strategy

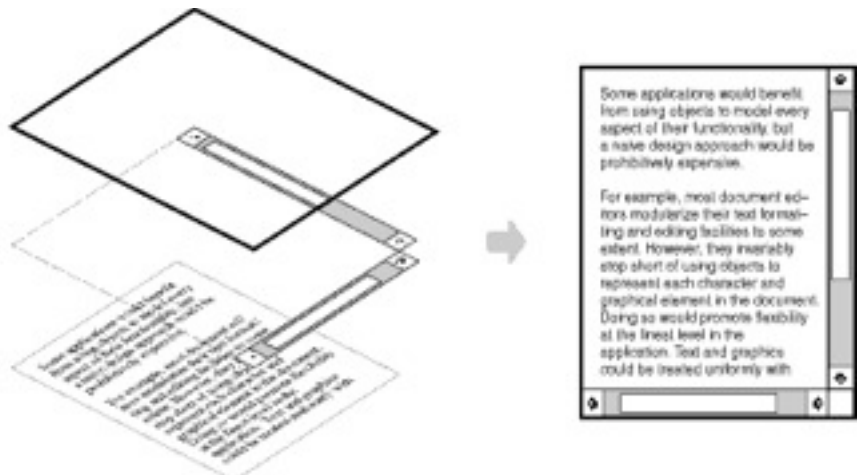
Het Strategy Patroon: gevolgen

- Familie van verwante algoritmen mogelijk. Er is wel een factorizatie van gezamenlijke functionaliteit nodig in de abstracte klasse
- Een alternatief voor subclassing. Subclassing op algoritme-niveau in plaats van op context-niveau
- Vermijdt conditionele statements, switch zit nu in verbonden strategy object
- Verschillende implementaties vaan eenzelfde gedrag mogelijk (bv. tijd/geheugen afweging)
- Oproeper moet wel op de hoogte zijn van verschillende mogelijke strategieen, en moet er misschien zelfs een instellen
- Communicatie-overhead tussen Context en Strategie
  => alle mogelijke bruikbare info moet worden doorgegeven maar is misschien overbodig
- Toenemend aantal objecten in je systeem => ev. Strategie als stateless object ontwerpen die dan gedeeld kan worden door de verschillende context-objecten (zie Flyweight)

Decorator

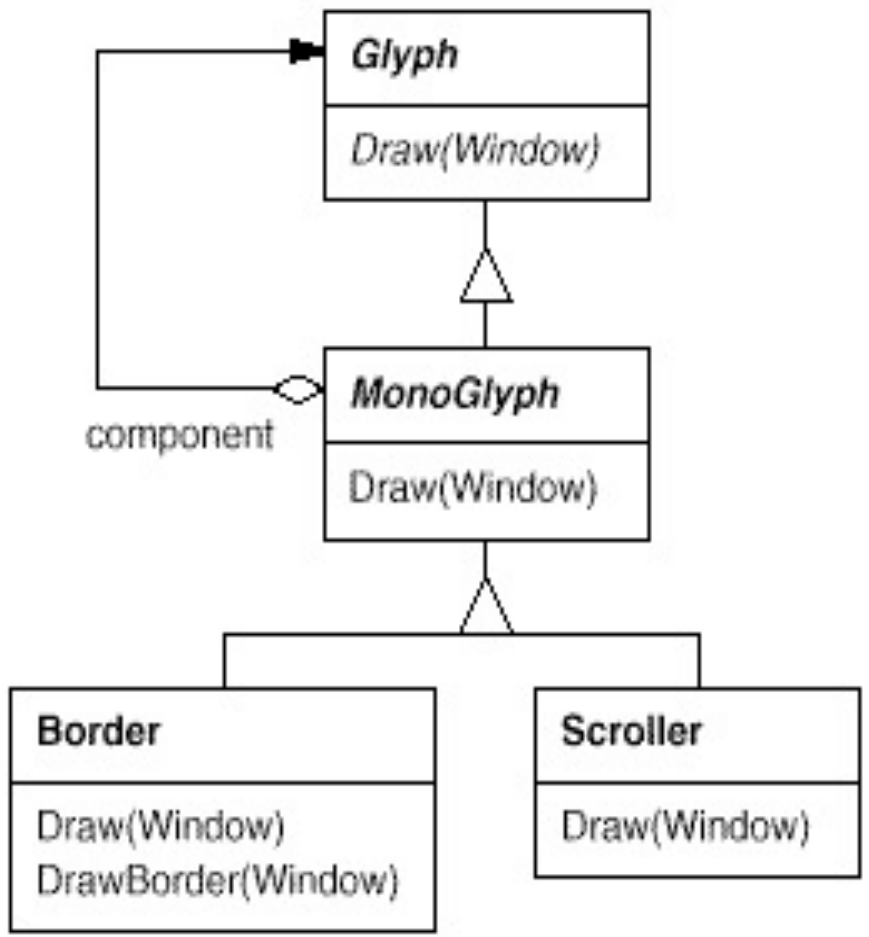
3. Verfraaien van UI Lexi

- toevoegen van kader, scroll bars, …
  - gemakkelijk dynamisch te verwijderen en te combineren
  - transparantie bij gebruik UI objecten
- oplossing door overerving:
  - combinatorische explosie
  - statische keuze
- oplossing door objectcompositie:
  - dynamische keuze
  - Glyph bevat Border of Border bevat Glyph ?



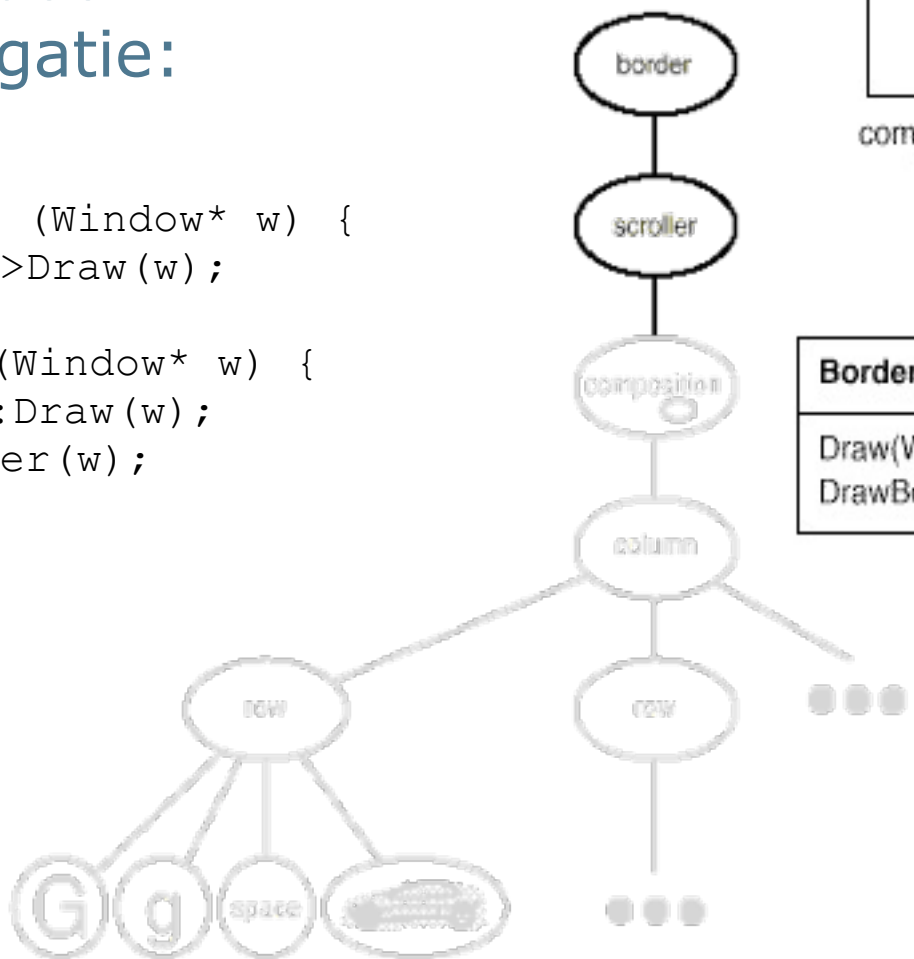
Verfraaien van UI Lexi

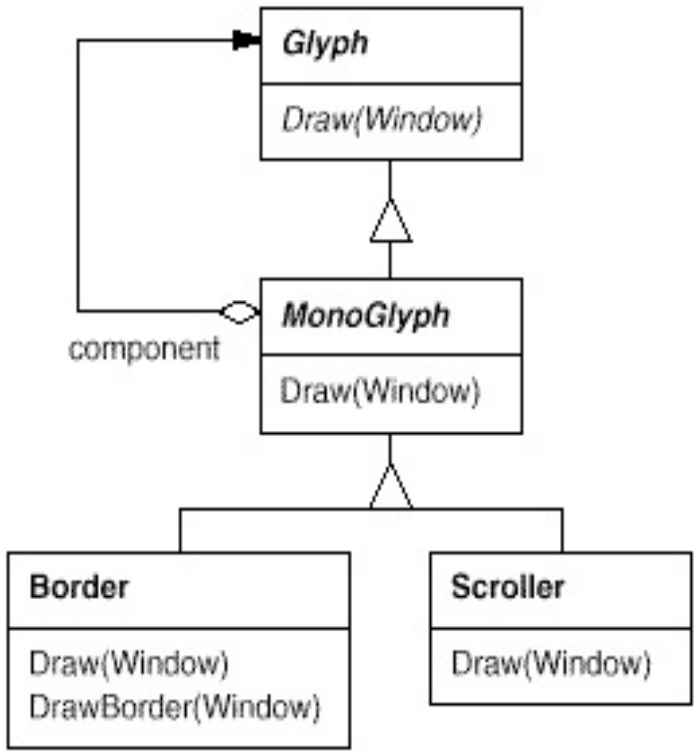
- Ontwerp van Border klasse:
  - hebben uitzicht, dus subklasse van Glyph
  - Border kan als gewone Glyph behandeld worden
- concept van Transparante Verpakking
  - enkelvoudige component
  - compatibele interfaces



- Dynamische configuratie
- Uitvoering door berichtdelegatie:

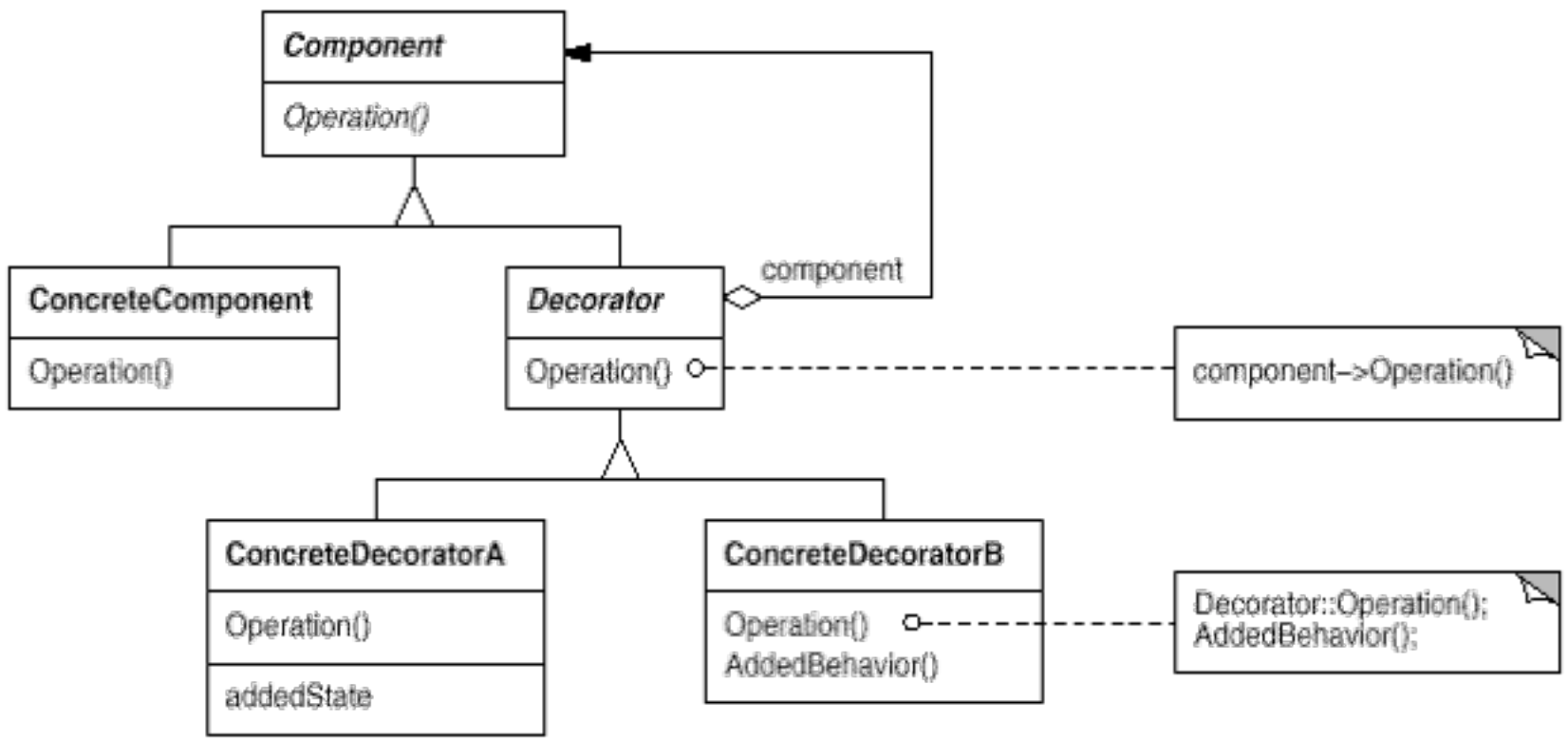
```
void MonoGlyph::Draw (Window* w) {
        _component->Draw(w);
                }
 void Border::Draw (Window* w) {
        MonoGlyph::Draw(w);
          DrawBorder(w);
                }
```





Het Decorator (Wrapper) Patroon (175)

- Doel:
  - dynamisch en transparant verantwoordelijkheden aan individuele objecten toevoegen
  - alternatief voor extensie door overerving

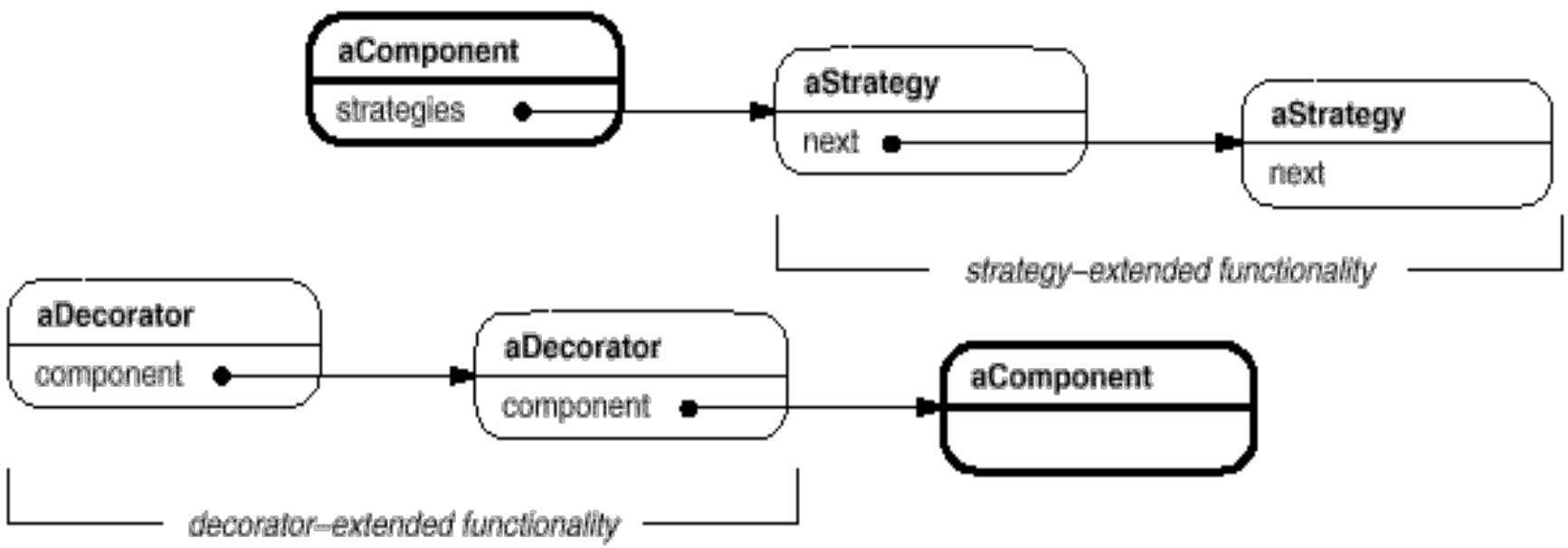


Het Decorator Patroon: gevolgen

- Kenmerken:
  - biedt meer flexibiliteit dan (statische) overerving
  - “pay as you go”: kleine functionele toevoegingen ipv alles-in-een
  - gedecoreerd object heeft andere identiteit
  - Nadeel: overvloed aan kleine objecten
- Implementatie-overwegingen:
  - houdt Component klasse lichtgewicht
  - abstracte Decorator alleen nodig bij meerdere verantwoordelijkheden

Het Decorator Patroon

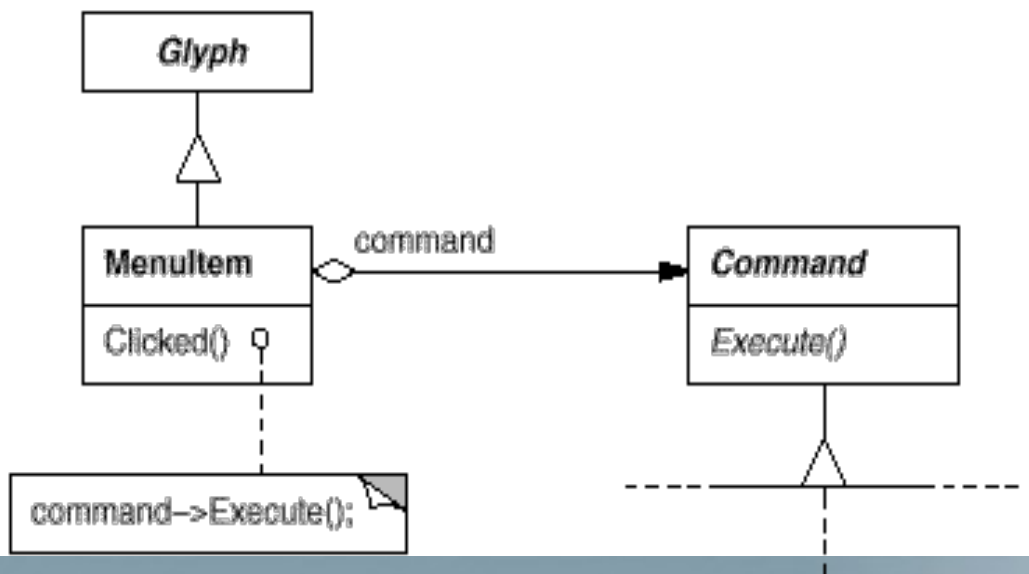
- Implementatie-overwegingen:
  - Decorator of Strategy?
    - Beiden passen gedrag object aan
    - Strategy moet gekend zijn door Component, maar kan eigen interface hebben
    - Strategy te verkiezen bij zwaargewicht Component klasse



Command

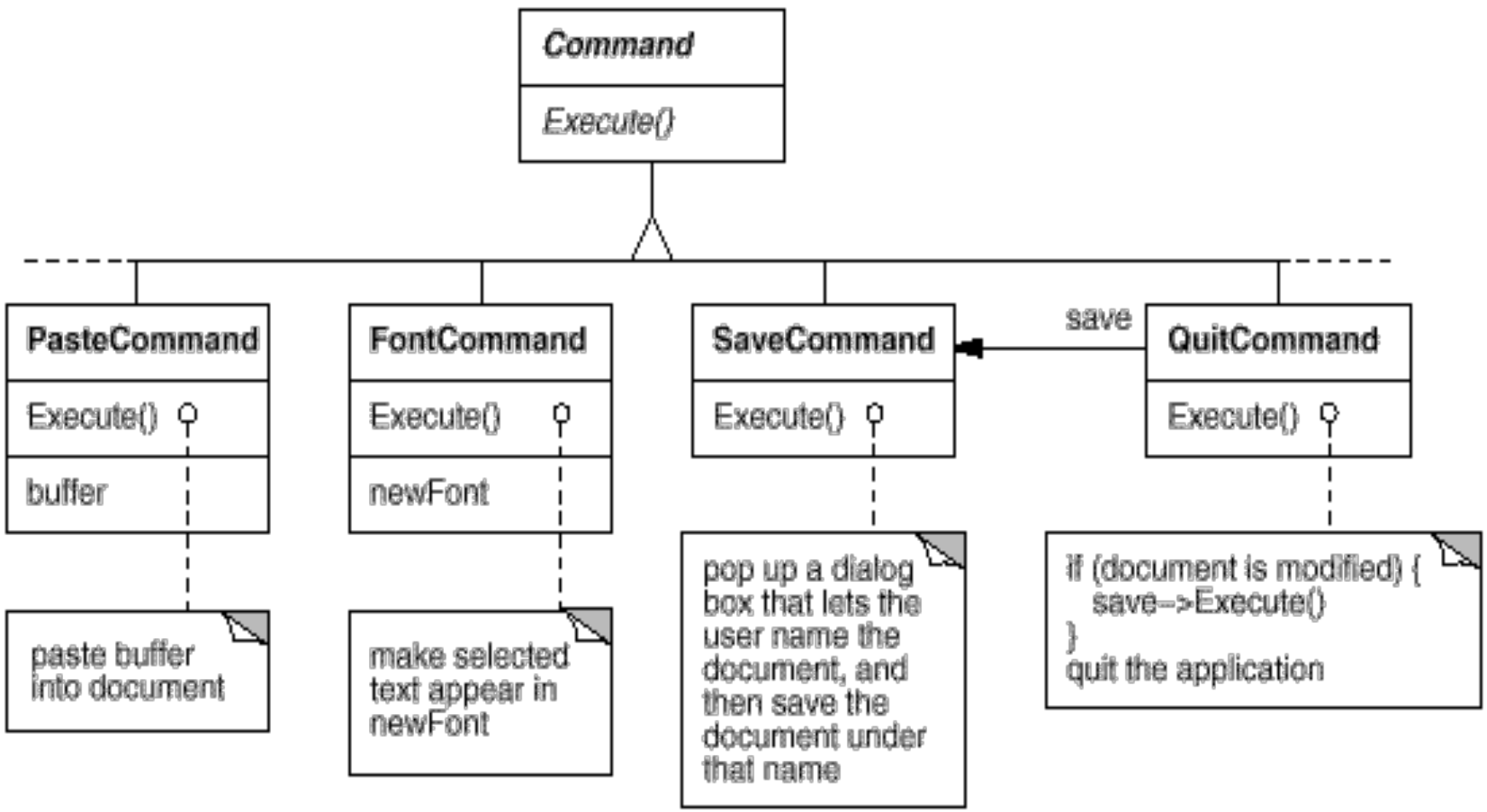
6. Gebruikersoperaties Lexi

- Doelstellingen:
  - scheiding operatie van user interface
    - één operatie kan op verschillende manieren aangegeven worden
    - anders hoge koppeling tussen UI klassen en applicatie
  - undo en redo van operaties ondersteunen
- Oplossing: definitie van Command klasse



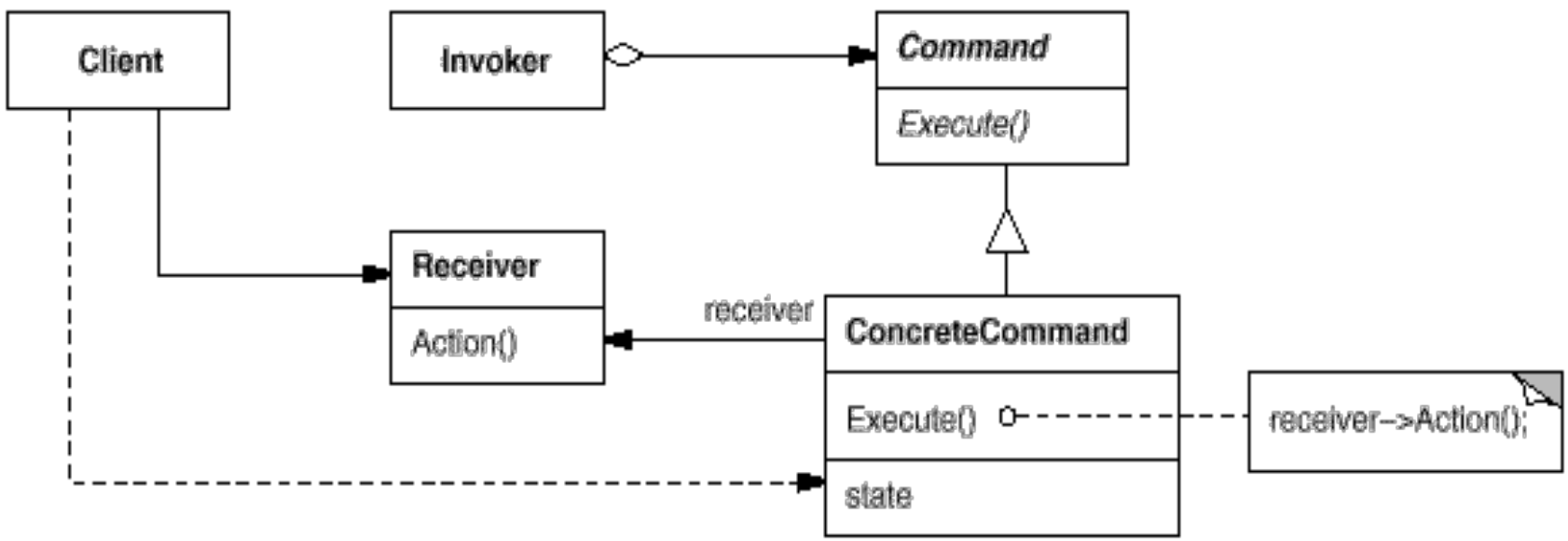
Gebruikersoperaties Lexi

- Elke concrete Command klasse bevat informatie voor en implementatie van operatie



Het Command Patroon (233)

- Doel:
  - van operaties eerste-orde objecten maken om deze te kunnen manipuleren (parameterizatie, queueing, logging, undoing, ...)
- Structuur:



Het Command Patroon: gevolgen

- Ontkoppeling oproeper en uitvoerder
- Commando's als first-class entiteiten die kunnen gemanipuleerd en uitgebreid worden
- Commando's kunnen worden gegroepeerd in samengestelde commando's
- Eenvoudig om nieuwe commando's te kunnen toevoegen
  => geen uitbreiding van basisklasse nodig

Patterns Catalogue

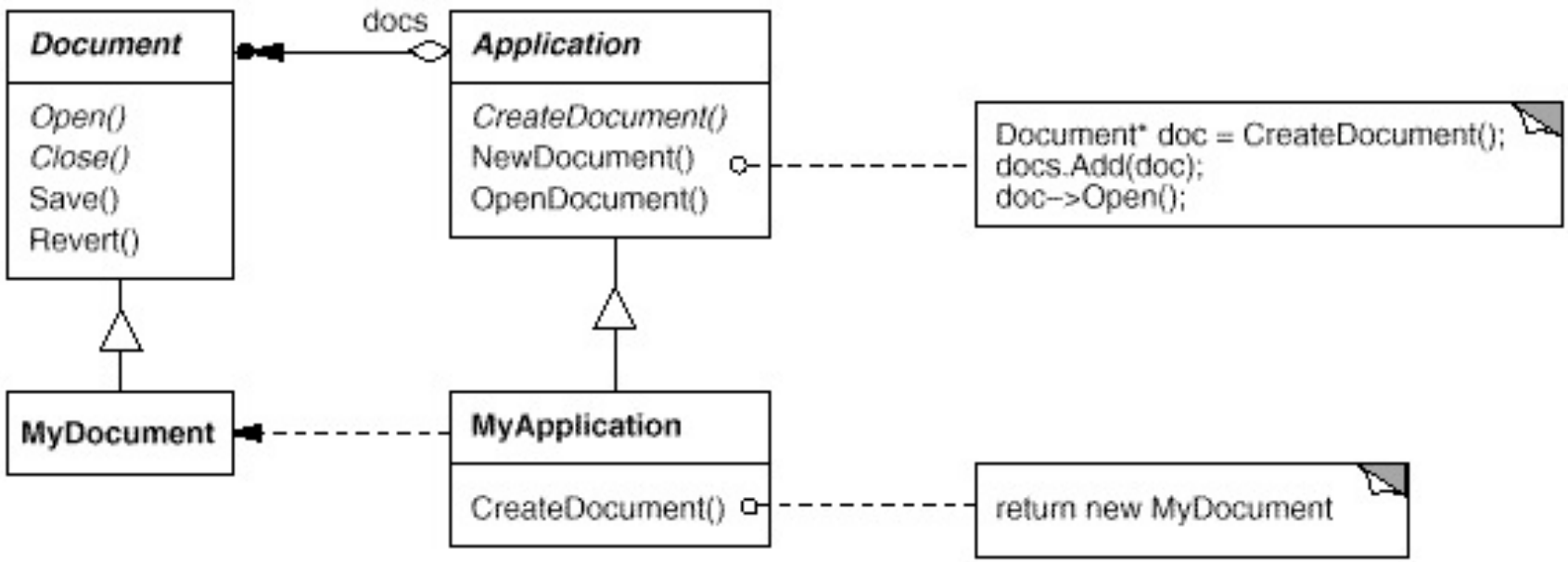
- Factory Method
- Composite
- Abstract Factory
- Singleton
- Proxy
- Adapter
- Observer
- Chain of Responsibility
- FlyWeight
- Facade

Factory Method

Factory Method

- Category
  - Creational
- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Motivation
  - When frameworks or toolkits use abstract classes to define and maintain relationships between objects and are responsible for creating the objects as well.

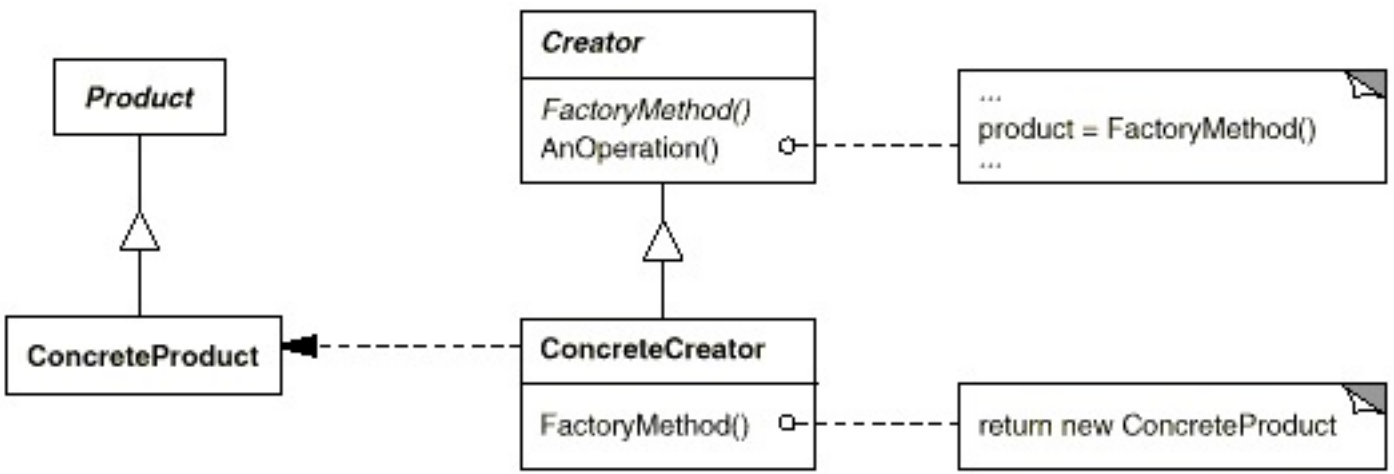
Motivation (cont)



Applicability

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure


Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
Product
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

Participants

- **Product**
  - Defines the interface of objects the factory method creates.
- **ConcreteProduct**
  - Implements the Product interface.
- **Creator**
  - Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - They call the factory method to create a Product object.
- **ConcreteCreator**
  - Overrides the factory method to return an instance of a ConcreteProduct.

Collaboration

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

- Eliminates the need to bind application specific classes into your code.
- Clients might have to subclass the Creator class just to create a particular ConcreteProduct object.
- Provides hooks for subclasses
  - the factory method gives subclasses a hook for providing an extended version of an object.
- Connects parallel class hierarchies
  - a client can use factory methods to create a parallel class hierarchy (parallel class hierarchies appear when objects delegate part of their responsibilities to another class).

Example Code

```
public interface Creator {

    public Fruit createFruit(String type);

}


public class GoodFruitCreator implements Creator{

    public Fruit createFruit(String type) {

    if (type == "apple"){

     return new Apple();

    }

    else return new Orange();

    }

}
```

Example Code

```
public class RottenFruitCreator implements Creator{

    public Fruit createFruit(String type){

    if (type == "apple"){

     return new RottenApple();

    }

    else return new RottenOrange();

    }

}
```

Example Code

```
abstract class Fruit {
    String type="";
    public String getType(){
     return type;
    }
}
public class Apple extends Fruit {
    Apple(){
     type = "apple";
    }
}
public class Orange extends Fruit {
    Orange(){
     type = "orange";
    }
}
```

Example Code

```
public class RottenApple extends Fruit {

    RottenApple(){

     type = "rottenapple";

    }

}



public class RottenOrange extends Fruit {

    RottenOrange(){

     type = "rottenorange";

    }

}
```

Example Code

```
public class FruitShop {

    Creator c;

    public FruitShop(Creator creator){

     c = creator;

    }

    public void getFruit(String type){

     Fruit f = c.createFruit(type);

     System.out.println("You get a(n) " + f.getType());

    }

}
```

Example Code

```
public class Main {
    public static void main(String[] args){
    FruitShop goodShop  = new FruitShop(new GoodFruitCreator());
    FruitShop badShop = new FruitShop(new RottenFruitCreator());
    goodShop.getFruit("apple");
    goodShop.getFruit("orange");
    badShop.getFruit("apple");
    badShop.getFruit("orange");
    }
}
```

Console:

```
    You get a(n) apple
    You get a(n) orange
    You get a(n) rottenapple
    You get a(n) rottenorange
```

Known Uses

- Toolkits and frameworks
- Class View in Smalltalk-80
  - contains a defaultController method which is a Factory Method.
- Class Behavior in Smalltalk-80
  - contains a parserClass method which also is a factory method.
- Could also be used to generated an appropriate type of proxy when an object requests a reference to an object. Factory Method makes it easy to replace the default proxy with another one.

Questions

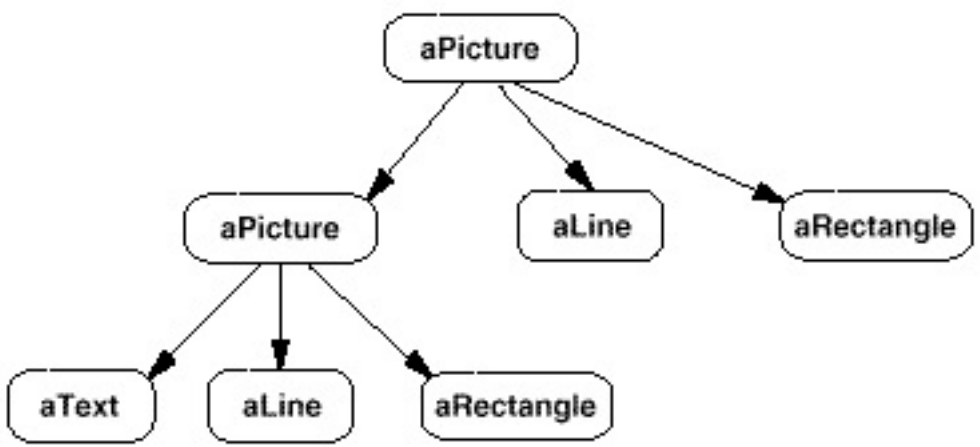
- How does this pattern promote loosely coupled code?
- Does the following code fragment implement the factory method pattern?

```
public class XMLReaderFactory {
    // This method returns an instance of a class
    // that implements the XMLReader interface.
    // The specific class it creates and returns is
    // based on a system property.

    public static XMLReader createXMLReader();

}

public interface XMLReader {
    public void setContentHandler(ContentHandler handler):
    public void parse(InputStream is);

}
```

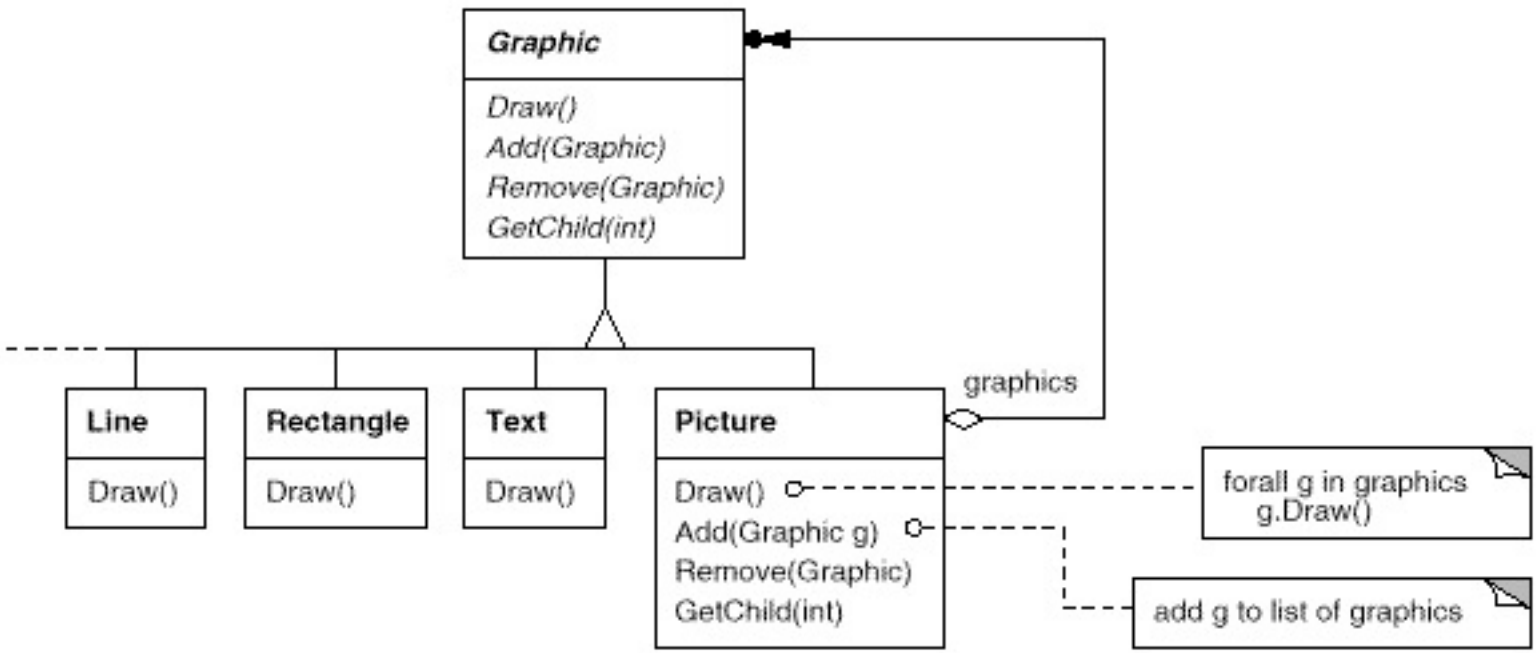
Composite

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



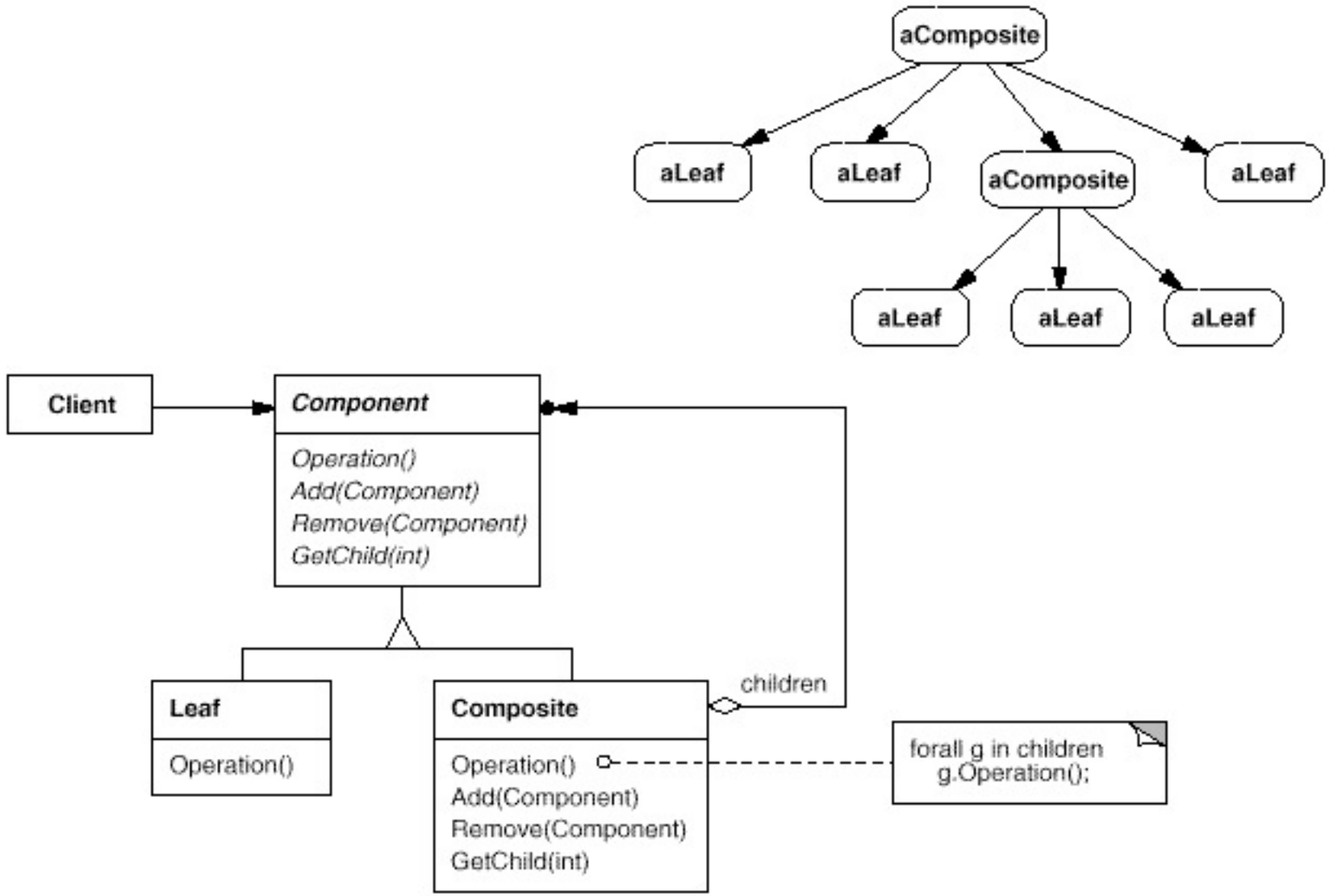
Motivation (cont)



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.

Example Code

```
abstract class Equipment {
    String name;
    public String name(){
     return name;
    }
    abstract int power();
    abstract int netPrice();
    abstract void add(Equipment e);
    abstract void remove(Equipment e);
    public Iterator createIterator(){
     return new NullIterator();
    }
    protected Equipment(String n){
     name = n;
    }
}
```

Example Code

```
abstract class CompositeEquipment extends Equipment {
    ArrayList l;
    public CompositeEquipment(String name){
     super(name);
     l = new ArrayList();
    }
    abstract int power();
    public int netPrice(){
     Iterator i = createIterator();
     int total = 0;
     while(i.hasNext()){
           Equipment e = (Equipment)i.next();
           total += e.netPrice();
     }
     return total;
    }
    public void add(Equipment e){
     l.add(e);
    }
//...
```

Example Code

```
//...


    public void remove(Equipment e){
     l.remove(e);
    }
    public Iterator createIterator(){
     return l.listIterator();
    }
}
```

Example Code

```
public class NullIterator implements Iterator {

    public boolean hasNext(){

     return false;

    }

    public Object next() throws NoSuchElementException{

     throw new NoSuchElementException();

    }

    public void remove() throws IllegalStateException{

     throw new IllegalStateException();

    }

}
```

Example Code

```
public class Cabinet extends CompositeEquipment {

    public Cabinet(String name){

     super(name);

    }

    public int power(){

     return 0;

    }

    public int netPrice(){

     int total = 60 + super.netPrice();

     return total;

    }

}
```

Example Code

```
public class Chassis extends CompositeEquipment {

    public Chassis(String name){

     super(name);

    }

    public int power(){

     return 0;

    }

    public int netPrice(){

     int total = 40 + super.netPrice();

     return total;

    }

}
```

Example Code

```
public class Bus extends CompositeEquipment {

    public Bus (String name){

     super(name);

    }

    public int power(){

     return 40;

    }

    public int netPrice(){

     int total = 100 + super.netPrice();

     return total;

    }

}
```

Example Code

```
public class Card extends Equipment{

    public Card(String name){

     super(name);

    }

    public int power(){

     return 60;

    }

    public int netPrice(){

     return 100;

    }

    public void add(Equipment e){}

    public void remove(Equipment e){}

}
```

Example Code

```
public class Floppydisk extends Equipment {

    public Floppydisk(String name){

     super(name);

    }

    public int power(){

     return 60;

    }

    public int netPrice(){

     return 50;

    }

    public void add(Equipment e){}

    public void remove(Equipment e){}

}
```

Example Code

```
public class Main {
    public static void main(String[] args) {
     Cabinet cabinet = new Cabinet("PC Cabinet");
     Chassis chassis = new Chassis("PC Chassis");
     cabinet.add(chassis);
     Bus bus = new Bus("MCA bus");
     bus.add(new Card("NetworkCard"));
     chassis.add(bus);
     chassis.add(new Floppydisk("3.5 Floppy"));
     System.out.println("The price is " + cabinet.netPrice());
    }
}
```

Console:

```
    The price is 350
```

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?

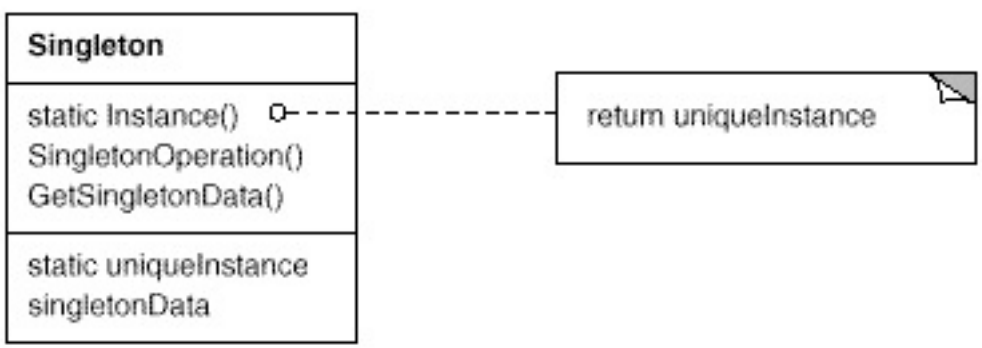
Singleton

Singleton

- Category
  - Creational
- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation
  - There should be only one instance.
  - For example, many printers, but only one printspooler.
  - Using a global variable containing the single instance?
    - Cannot ensure no other instances are created.
  - Let the class control single instance.

Applicability and Structure

- Use Singleton pattern when
  - There must be exactly one instance of al class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
- Structure



Participants and Collaborations

- Participants
  - Singleton
    - Defines an instance operation that lets clients access its unique instance. Instance is a class operation that will either return or create and return the sole instance.
    - May be responsible for creating its own unique instance.
- Collaborations
  - Clients access a Singleton solely through Singleton's instance operation.

Consequences

- Controlled access to sole instance.
  - Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
- Reduced name space.
  - The Singleton pattern is an improvement over global variables that store sole instances.

Consequences (cont)

- Permits refinement of operations and representation.
  - The Singleton class may be subclassed, an application can be configured with an instance of the class you need at runtime.
- Permits a variable number of instances.
  - The same approach can be used to control the number of instances that can exist in an application, only the operation that grants access to the instance(s) must be provided.
- More flexible than class operations.

Example Code

```
public class MazeFactory {

    private static MazeFactory instance = null;

    public static MazeFactory getInstance(){

     if (instance == null){

           instance = new MazeFactory();

     }

     return instance;

    }

    private MazeFactory();


    // rest of the interface

    // ...

}
```

Example Code

```
public class Main {

    public static void main(String[] args) {

     MazeGame gmg = new MazeGame();

     //MazeFactory factory = new MazeFactory();

     MazeFactory factory = MazeFactory.getInstance();

     Maze mz = gmg.createMaze(factory);

     }

}
```

Known Uses

- Every time you want to limit the creation of additional object after the instantiation of the first one. This is usefull to limit memory usage when multiple objects are not necessary.

Questions

- What is the difference with a global variable?
- Gamma (one of the authors of the book on Design Patterns) recently pointed out that he was very unhapy with this pattern. More specifically he claims that it usually indicates bad design. Can you imagine what he thinks so ?

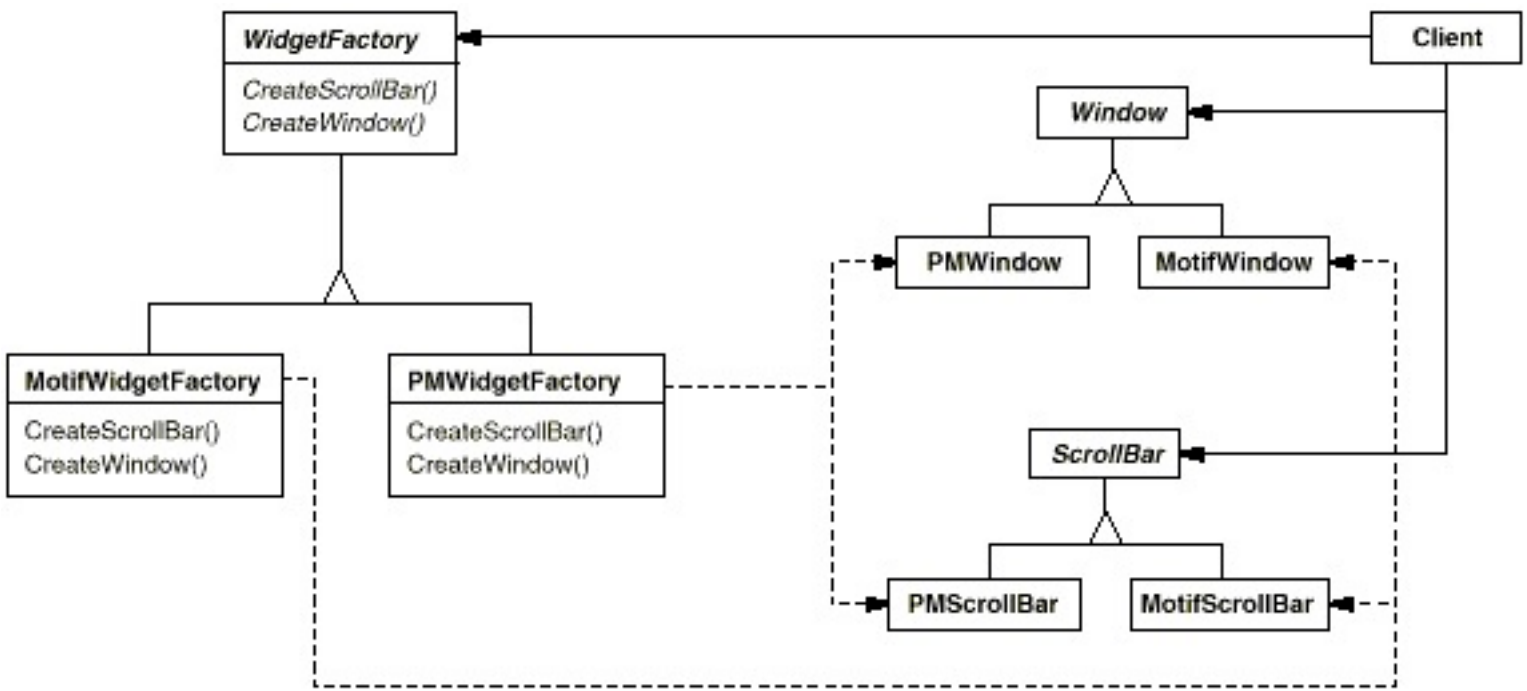
Abstract Factory

Abstract Factory

- Category
  - Creational
- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Motivation
  - User interface toolkit for multiple look-and-feel standards.
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation (cont)

- Motivation (cont)

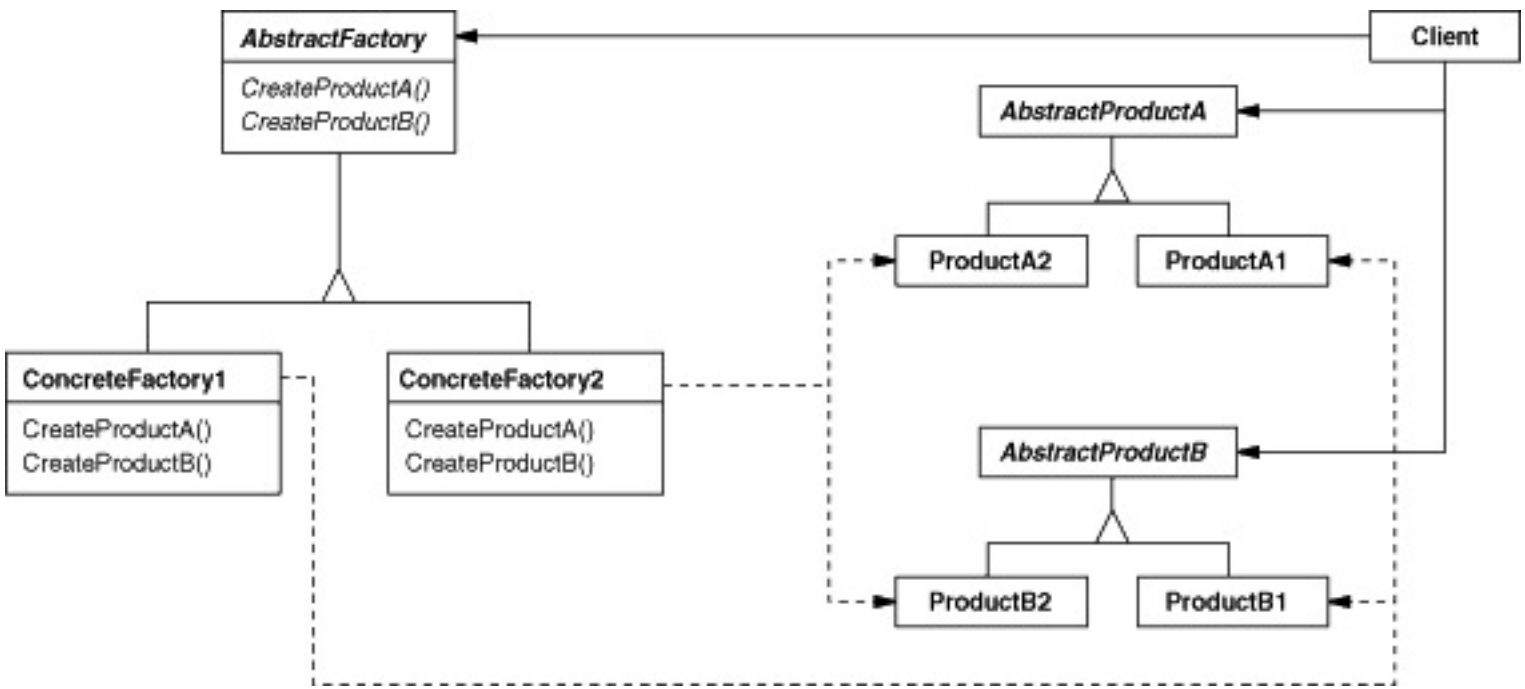


Applicability

- Use the Abstract Factory pattern when
  - a system should be independent of how its products are created, composed and represented.
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure

- Structure



Participants

- **AbstractFactory**
  - Declares an interface for operations that create abstract product objects.
- **ConcreteFactory**
  - Implements the operations to create concrete product objects.
- **AbstractProduct**
  - Declares an interface for a type of product object.

Participants (cont)

- **ConcreteProduct**
  - Defines a product object to be created by the corresponding concrete factory.
  - Implements the AbstractProduct interface.
- **Client**
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of ConcreteFactory is created at run-time. This concrete factory creates products having a particular implementation.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

- It isolates concrete classes.
  - The abstract factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes.
  - Product class names are isolated in the implementation of the concrete factory and do not appear in the client code.
- It makes exchanging product families easy.
  - The concrete factory appears only one in the application – that is, where it is instantiate – to it is easy to replace.

Consequences (cont)

- It promotes consistency among products.
  - When products of one family are designed to work together, it is important for an application to use objects from one family only.
  - The abstract factory makes this easy to enforce.
- Supporting new kinds of products is difficult.
  - Because the abstract factory interface fixes the set of products that can be created, it is not easy to add new products.
  - This would require extending the factory interface which involves extending changing the abstract factory and all its subclasses.

Example Code

```
public abstract class MapSite {
    public abstract void enter();
}

public class Room extends MapSite{
    int roomNumber;
    MapSite[] sides = new MapSite[4];
    public Room(int roomNo){
     roomNumber = roomNo;
    }
    public MapSite getSide(int direction){
     return sides[direction];
    }
    public void setSide(int direction, MapSite site){
     sides[direction] = site;
    }
    public void enter(){};
}
```

Example Code

```
public class Wall extends MapSite {
    public void enter(){};
}

public class Door extends MapSite {
    Room roomOne;
    Room roomTwo;
    boolean isOpen;
    public Door (Room room1, Room room2){
     roomOne = room1;
     roomTwo = room2;
    }
    public Room otherSideFrom(Room room){}
    public void enter(){};
}
```

Example Code

```
public class Maze {
 //...
    public Maze(){
    //...
    }
    public void addRoom(Room room){
    //...
    }
    public Room roomNo(int roomNbr){
    //...
    }
}

public class Direction {
    static int north = 0;
    static int south = 1;
    static int east = 2;
    static int west = 3;
}
```

Example Code

```
//This is a bad implementation, imagine you want to have other Mazes!
public class BadMazeGame {
    public Maze createMaze(){
     Maze newMaze = new Maze();
     Room r1 = new Room(1);
     Room r2 = new Room(2);
     Door theDoor = new Door(r1, r2);
     newMaze.addRoom(r1);
     newMaze.addRoom(r2);
     r1.setSide(Direction.north, new Wall());
     r1.setSide(Direction.east, theDoor);
     r1.setSide(Direction.south, new Wall());
     r1.setSide(Direction.west, new Wall());
     r2.setSide(Direction.north, new Wall());
     r2.setSide(Direction.east, new Wall());
     r2.setSide(Direction.south, new Wall());
     r2.setSide(Direction.west, theDoor);
     return newMaze;
     }
}
```

Example Code

```
public abstract class MazeFactory {
    public Maze makeMaze(){
     return new Maze();
    }
    public Wall makeWall(){
     return new Wall();
    }
    public Room makeRoom(int n){
     return new Room(n);
    }
    public Door makeDoor(Room r1, Room r2){
     return new Door(r1, r2);
    }
}

public class RegularMazeFactory extends MazeFactory {}
```

Example Code

```
public class GoodMazeGame {
    public Maze createMaze(MazeFactory factory){
     Maze newMaze = factory.makeMaze();
     Room r1 = factory.makeRoom(1);
     Room r2 = factory.makeRoom(2);
     Door theDoor = factory.makeDoor(r1, r2);
     newMaze.addRoom(r1);
     newMaze.addRoom(r2);
     r1.setSide(Direction.north, factory.makeWall());
     r1.setSide(Direction.east, theDoor);
     r1.setSide(Direction.south, factory.makeWall());
     r1.setSide(Direction.west, factory.makeWall());
     r2.setSide(Direction.north, factory.makeWall());
     r2.setSide(Direction.east, factory.makeWall());
     r2.setSide(Direction.south, factory.makeWall());
     r2.setSide(Direction.west, theDoor);
     return newMaze;
     }
}
```

Example Code

```
public class Main {

    public static void main(String[] args) {

     BadMazeGame bmg = new BadMazeGame();

     Maze mz1 = bmg.createMaze();



     GoodMazeGame gmg = new GoodMazeGame();

     RegularMazeFactory factory = new RegularMazeFactory();

     Maze mz2 = gmg.createMaze(factory);

    }

}
```

Known Uses

- Usually used in toolkits for generating look-and-feel specific user interface objects.
- Also used to achieve portability across different window systems.

Questions

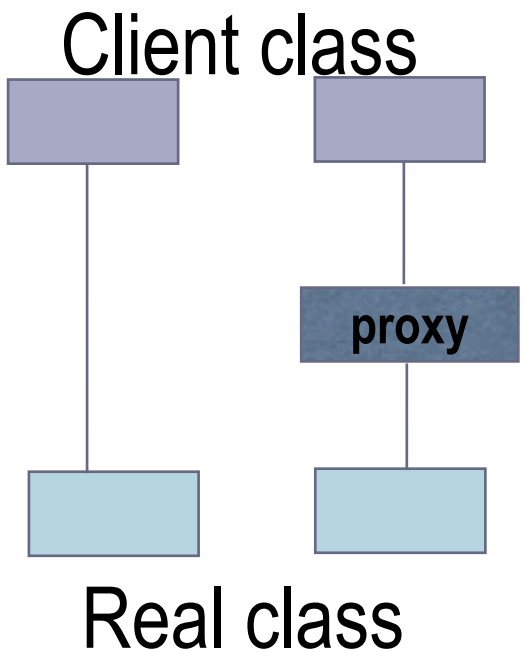
- Describe the working of the abstract factory pattern with your own words.
- What pattern(s) is (are) often used together with the abstract factory pattern?

Proxy

Proxy

- Category
  - Structural
- Intent
  - Provide a surrogate or placeholder for another object to control access to it.
- Motivation
  - Defer the full cost of the creation and initialisation of an object until we actually need it.
  - For example: a document with lots of graphical objects can be expensive to create, but opening it should be fast.
  - A proxy could act as a stand-in for the real objects.

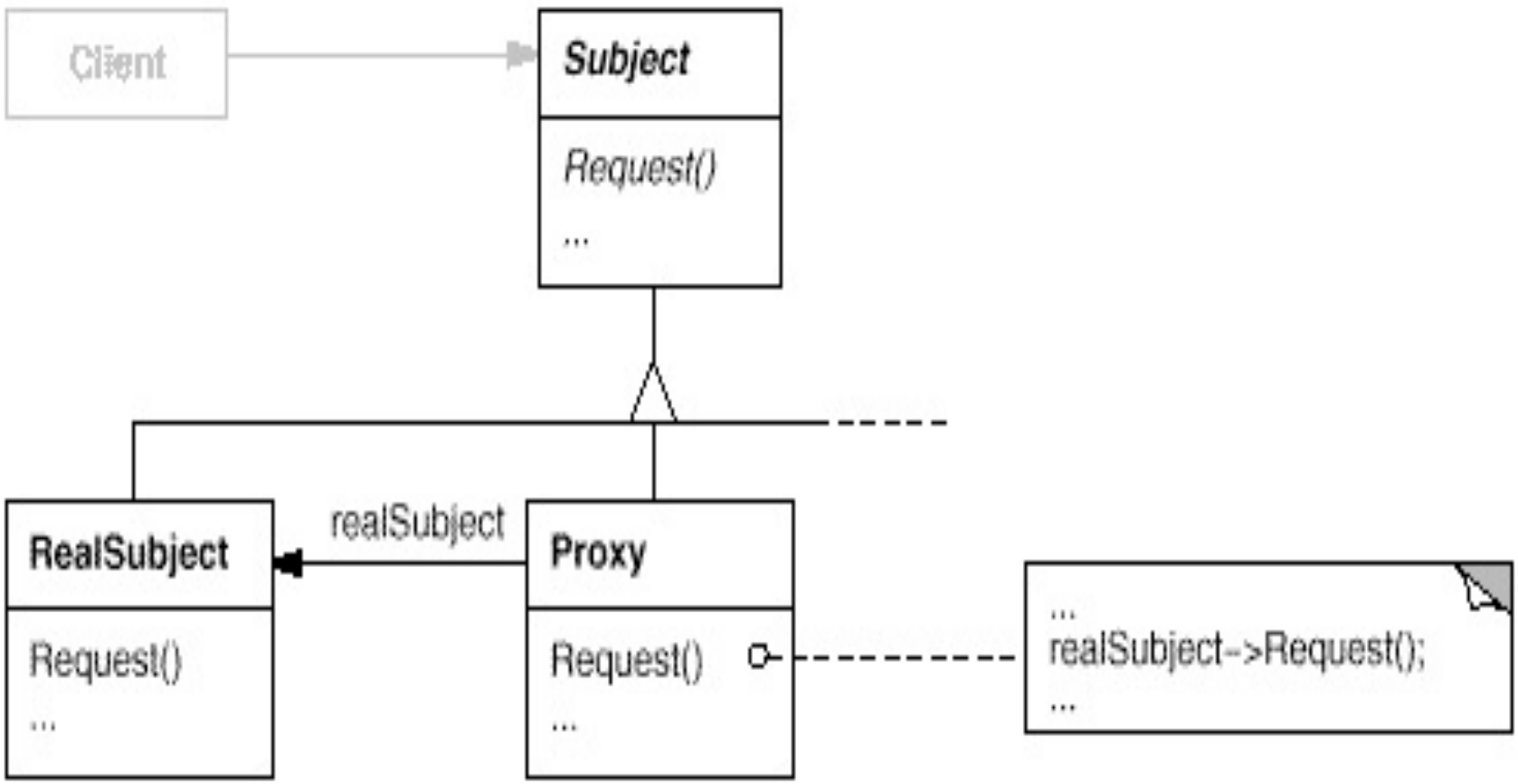
Kinds of Proxies



- a **remote proxy** provides a local representative for an object in a different address space.
- a **virtual proxy** creates expensive objects on demand.
- a **protection proxy** controls access to the original object and are useful when objects have different access rights.
- a **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed: e.g. counting references, loading a persistent object when it is first referenced, locking the real object, ...

Structure

- Structure



Participants

- Proxy
  - Maintains a reference that lets the proxy access the real subject.
  - Provides an interface identical to the Subject's so that a proxy can be substituted for the real subject.
  - Controls access to the real subject and may be responsible for creating and deleting it.
  - **Remote proxies** are responsible for encoding a request and its arguments and for sending the request to the real subject in the other address space.
  - **Virtual proxies** may cache information about the real subject so that they can postpone accessing it.
  - **Protection proxies** check that the caller has the access permission to perform a request.

Participants (cont) and Collaboration

- Participants (cont)
  - Subject
    - Defines a common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
  - RealSubject
    - Defines the real object that the proxy represents.
- Collaboration
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of Proxy.

Consequences

- The Proxy pattern introduces a level of indirection when accessing an object. This indirection has many uses:
  - A remote proxy can hide the fact that the object resides in a different address space.
  - A virtual proxy can perform optimisations.
  - Both protection proxies and smart pointers allow additional housekeeping.

Consequences (cont)

- The proxy patterns can be used to implement “copy-on-write”.
  - To avoid unnecessary copying of large objects the real subject is referenced counted.
  - Each copy requests increments this counter but only when a clients requests an operation that modifies the subject the proxy actually copies it.

Example Code

```
public interface IExampleClass {
    public void method1();
    public void method2();
    public void print();
}
public class ExampleClass implements IExampleClass {
    private String name;
    public ExampleClass(String n){
     name = n;
    }
    public void method1(){
     System.out.println(name + " executed ExampleClass method1");
    }
    public void method2(){
     System.out.println(name + " executed ExampleClass method2");
    }
    public void print(){
     System.out.println("My name is " + name);
    }
}
```

Example Code

```
public class ExampleClassProxy implements IExampleClass{
    private String name;
    private ExampleClass eClass = null;
    public ExampleClassProxy(String n){
     this.name = n;
    }
    public void method1(){
     getInstance().method1();
    }
    public void method2(){
     getInstance().method2();
    }
    public void print(){
     System.out.println("My name is " + name);
    }
    //...
```

Example Code

```
//...


    public IExampleClass getInstance(){
     if (eClass == null){
            eClass = new ExampleClass(name);
            System.out.println("Created the ExampleClass");
     }
     return eClass;
    }
}
```

Example Class

```
public class Main {

    public static void main(String[] args) {

     IExampleClass e = new ExampleClassProxy("Andy");

     e.print();

     e.method1();

     e.method2();

     }

}
```

Known Uses

- Encapsulators can be implemented as proxies.
- They are often used to represent local representatives for distributed objects.
- They have been used in textbuilding tools to enhance performance.

Questions

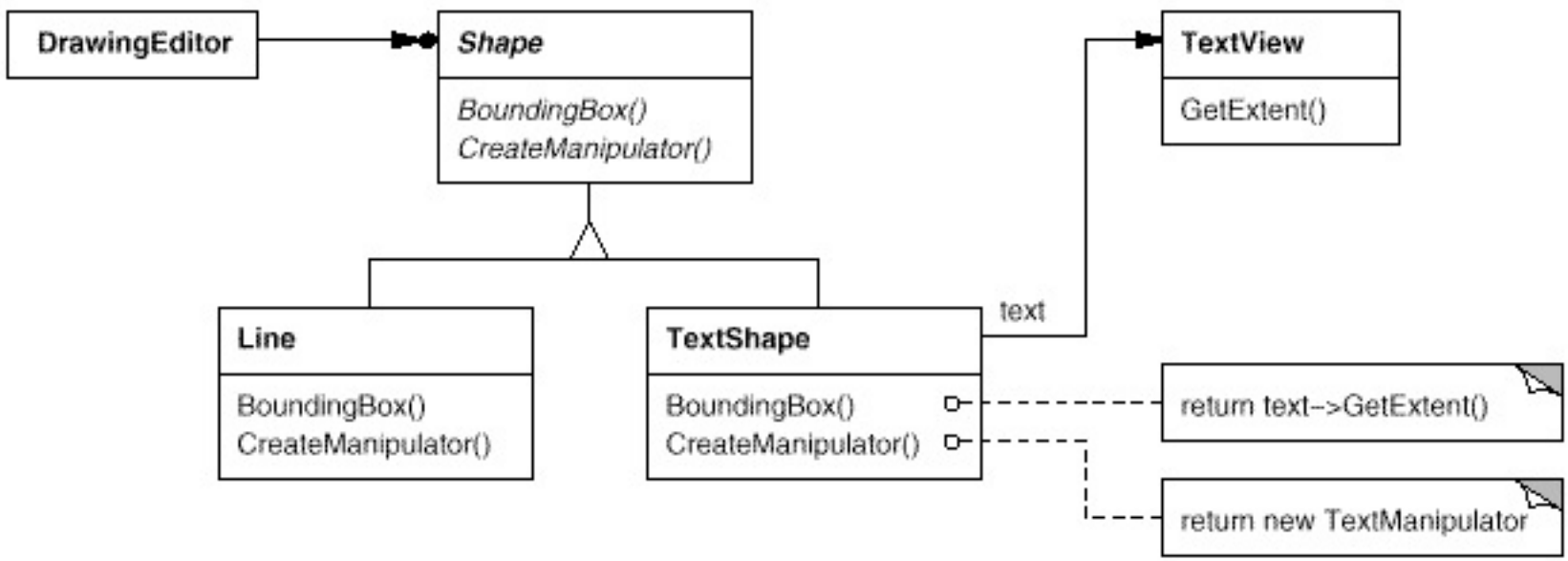
- If a Proxy is used to instantiate an object only when it is absolutely needed, does the Proxy simplify code?

Adapter

Adapter

- Category
  - Structural
- Intent
  - Convert the interface of a class into another interface clients expect. Lets classes with incompatible interfaces work together.
- Motivation
  - Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires.
  - A drawing editor has one abstraction for lines and textboxes, but textbox has a different interface and implementation.

Motivation (cont)

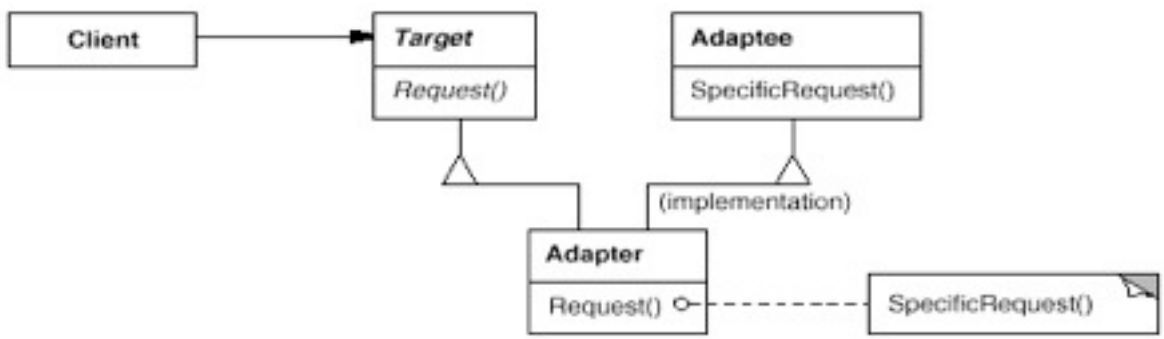


Applicability

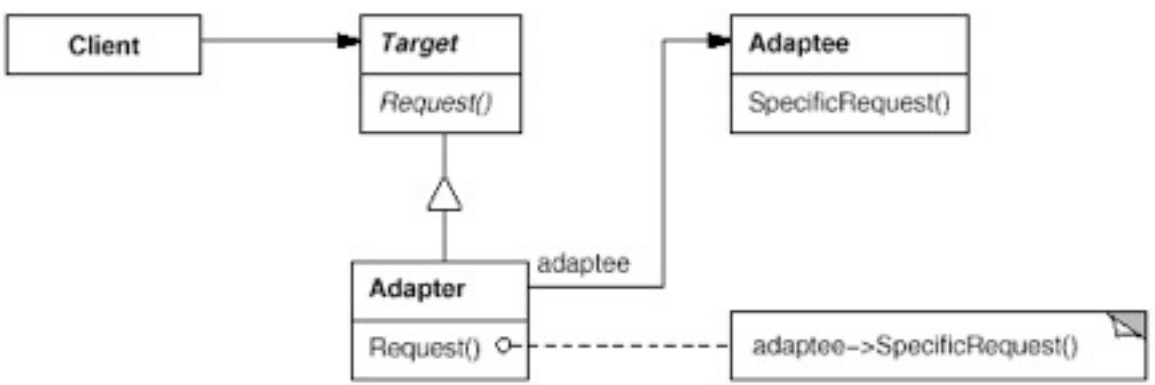
- Use Adapter when
  - You want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, which do not necessarily have compatible interfaces.
  - (object adapter only) You need to use several existing subclasses, but it's impractival to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure

- Structure
  - Class adapter



  - Object



Participants and Collaborations

- Participants
  - **Target**
    - Defines the domain-specific interface that Client uses.
  - **Client**
    - Collaborates with objects conforming to the Target interface.
  - **Adaptee**
    - Defines an existing interface that needs adapting.
  - **Adapter**
    - Adapts the interface of Adaptee to the Target interface.
- Collaborations
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

- How much adapting does Adapter do?
  - Ranges from simple interface conversion to supporting an entirely different set of operations.
- Pluggable adapters.
  - By building interface adaption into a class, it becomes more reusable because it does not assume the same interface to be used by other classes.
- Using two-way adapters to provide transparency.
  - An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency.

Example Code

```
// has a boundingbox for its bounderies
// uses a Manipulator to animate a Shape when a user manipulates it

interface Shape {
    public void boundingBox(Point bottomLeft, Point topRight);
    public Manipulator createManipulator();
}

// has origin, height and width instead
// has no Manipulator

public class TextView {
    public void getOrigin (Coord x, Coord y){};
    public void getExtent (Coord width, Coord height){};
    public boolean isEmpty (){return true;};
}
```

Example Code

```
// this is an example of a class-adaptor

public class CTextShape extends TextView implements Shape {
    //convert one interface to another
    public void boundingBox(Point bottomLeft, Point topRight){
     Coord bottom = new Coord();
     Coord left = new Coord();
     Coord width = new Coord();
     Coord height = new Coord();
     getOrigin(bottom, left);
     getExtent(width, height);
     bottomLeft = new Point(bottom, left);
     topRight = new Point(new Coord(bottom.value + height.value),
              new Coord(left.value + width.value));
    };
    //direct forwarding
    public boolean isEmpty(){return super.isEmpty();};
    //assume TextManipulator exists
    public Manipulator createManipulator(){
     return new TextManipulator(this);
    };
}
```

Example

```
// this is an example of an object-adaptor
public class OTextShape implements Shape {
    TextView text;
    OTextShape(TextView t){
     text = t;
    }
    public void boundingBox(Point bottomLeft, Point topRight){
     Coord bottom = new Coord();
     Coord left = new Coord();
     Coord width = new Coord();
     Coord height = new Coord();
     text.getOrigin(bottom, left);
     text.getExtent(width, height);
     bottomLeft = new Point(bottom, left);
     topRight = new Point(new Coord(bottom.value + height.value),                new
    Coord(left.value + width.value));
    };
    public boolean isEmpty(){
     return text.isEmpty();};
    public Manipulator createManipulator(){
     return new TextManipulator(this);
    };
}
```

Questions

- Would you ever create an Adapter that has the same interface as the object which it adapts? Would your Adapter then be a Proxy?

Observer

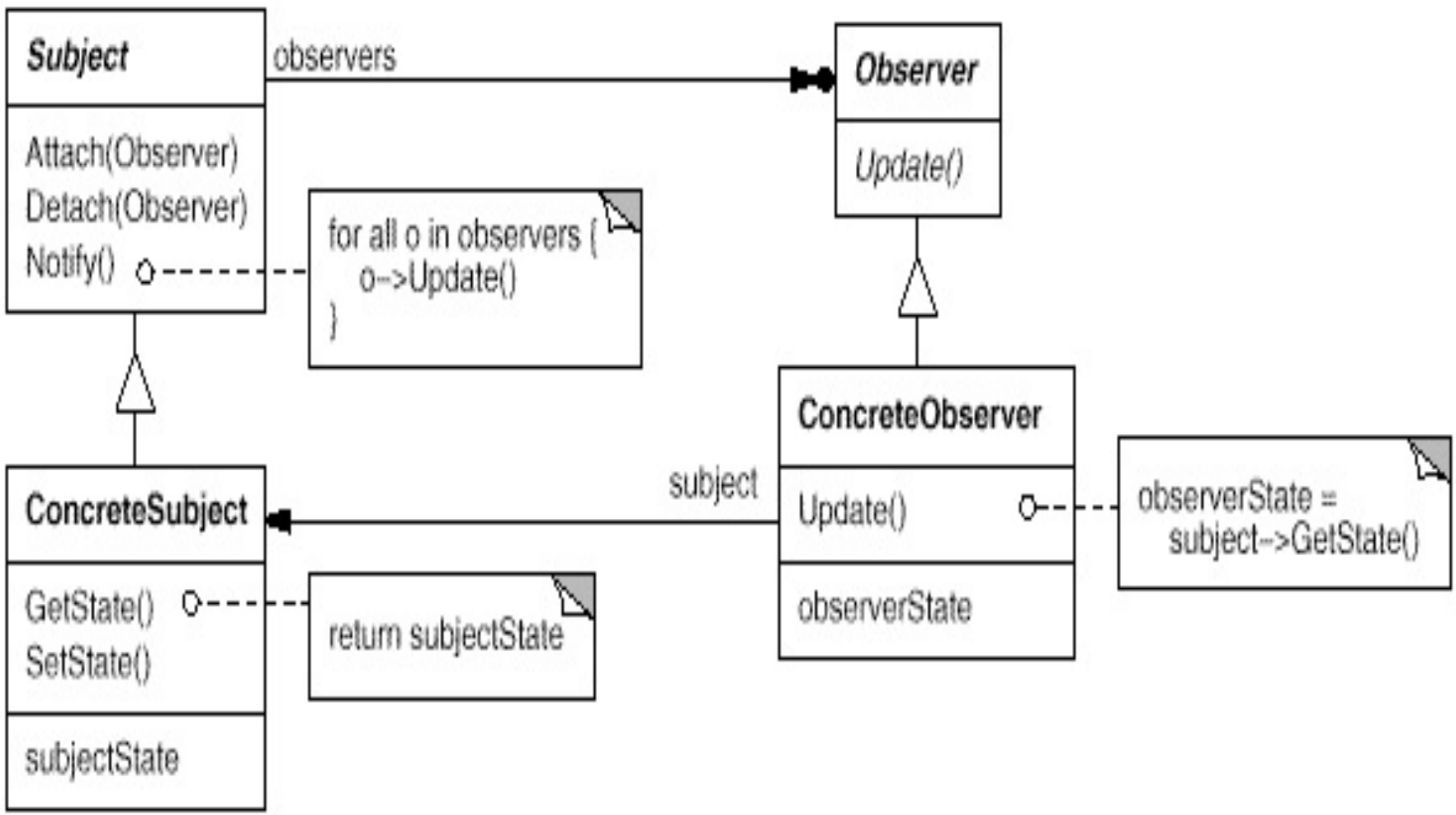
Observer

- Category
  - Behavioral
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.
- Motivation
  - different types of GUI elements depicting the same application data.
  - different windows showing different views on the same application model.

Applicability

- When an abstraction has two aspects, one dependant on the other. Encapsulating these aspects in seperate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you do not want these objects tightly coupled.

Structure



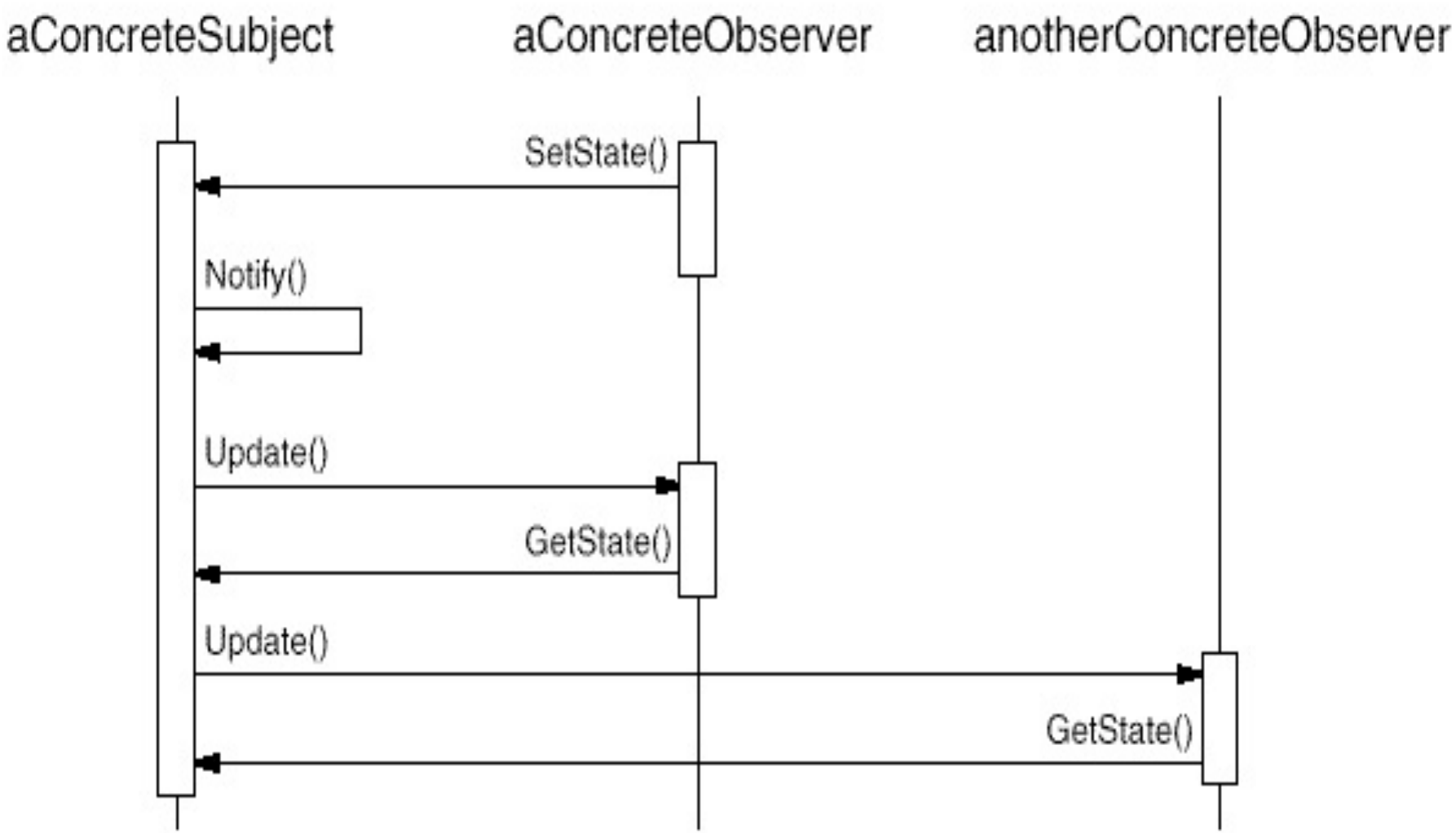
Participants

- Subject
  - knows its observers. Any number of Observer objects may observe an object.
  - provides an interface for attaching and detaching Observers.
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.

Participants (cont)

- ConcreteSubject
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistant with the subject's.
  - implements the Observer updating interface.

Collaborations



Consequences

- Abstract and minimal coupling between Subject and Observer.
  - The subject does not know the concrete class of any observer. Concrete subject and concrete observer classes can be reused independently.
- Support for broadcast communication.
  - The notification a subject sends does not need to specify a receiver, it will broadcast to all interested parties.

Consequences (cont)

- Unexpected updates.
  - Observers don’t have knowledge about each other’s presence, a small operation may cause a cascade of updates.

Example Code

```
public class Subject {
    private List observers = new LinkedList();
    protected Subject();
    void attach(Observer o){
     observers.add(o);
    };
    void detach(Observer o){
     observers.remove(o);
    };
    void notifyObservers(){
     ListIterator i = observers.listIterator(0);
     while(i.hasNext()){
           ((Observer) i.next()).update(this);
     }
    };
}
```

Example Code

```
public abstract class Observer {
    abstract void update(Subject changedSubject);
    protected Observer();
}

public class ClockTimer extends Subject {
    int hour = 0;
    int minutes = 0;
    int seconds = 0;
    int getHour(){
     return hour;
    }
    int getMinutes(){
     return minutes;
    }
    int getSeconds(){
     return seconds;
    }
//…
```

Example Code

```
void tick(){
    //updating the time
    if(seconds == 59){
    if (minutes == 59){
          if (hour == 23){
          hour = 0;
          minutes = 0;
          seconds = 0;
          }
          else {
             seconds = 0;
             minutes = 0;
             hour = hour + 1;
          }
    }
    else {
          seconds = 0;
          minutes = minutes + 1;
    }
    }
    else seconds = seconds + 1;
    notifyObservers();
    }
}
```

Example Code

```
public class ClockDisplay extends Observer{
    private ClockTimer subject;
    private String clocktype;

    ClockDisplay(ClockTimer c, String type){
     subject = c;
     subject.attach(this);
     clocktype = type;
    }

    void update(Subject changedSubject){
     if (changedSubject == subject){
           displayTime();
     }
    }

    void displayTime(){
     System.out.println(clocktype + "-->" + subject.getHour()+ ":"          +
    subject.getMinutes()+ ":" + subject.getSeconds());
    }
}
```

Example Code

```
//if not interrupted, will continue for 24 hours
public class Main {
    public static void main(String[] args) {
     ClockTimer c = new ClockTimer();
     ClockDisplay d1 = new ClockDisplay(c, "Digital");
     ClockDisplay d2 = new ClockDisplay(c, "Analog");
     int count = 0;
     while (count < (60*60*24)){
           c.tick();
           try {
              Thread.sleep(1000);
           } catch (InterruptedException e) {
              e.printStackTrace();
           }
           count = count + 1;
     }
    }
}
```

```
Digital-->0:0:1
Analog-->0:0:1
Digital-->0:0:2
Analog-->0:0:2
…
```

Known Uses

- Best known use is Smalltalk Model/View/Controller.

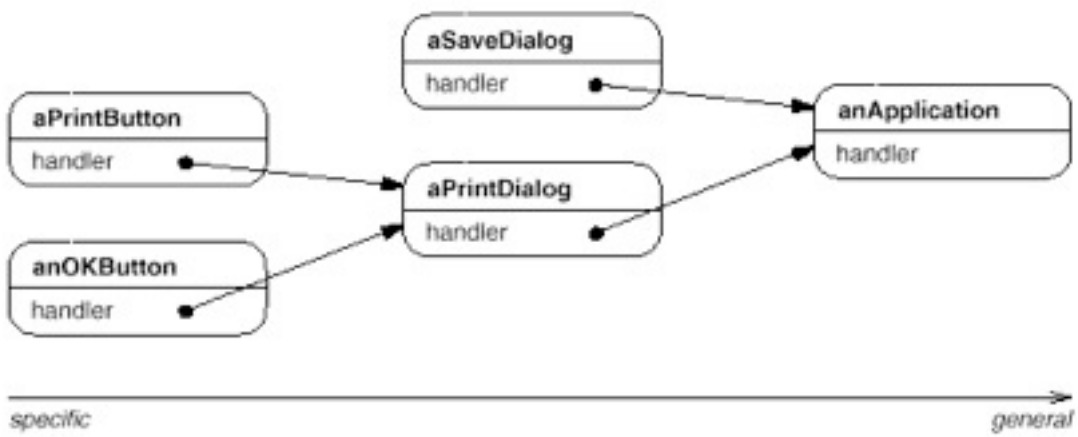
Questions

- There are two methods for propagating data to observers with the Observer design pattern: the push model and the pull model. Why would one model be preferable over the other? What are the trade-offs of each model?
- In what real-world system can we expect encounter the Observer pattern quite often?

Chain of Responsibility

Chain of Responsibility

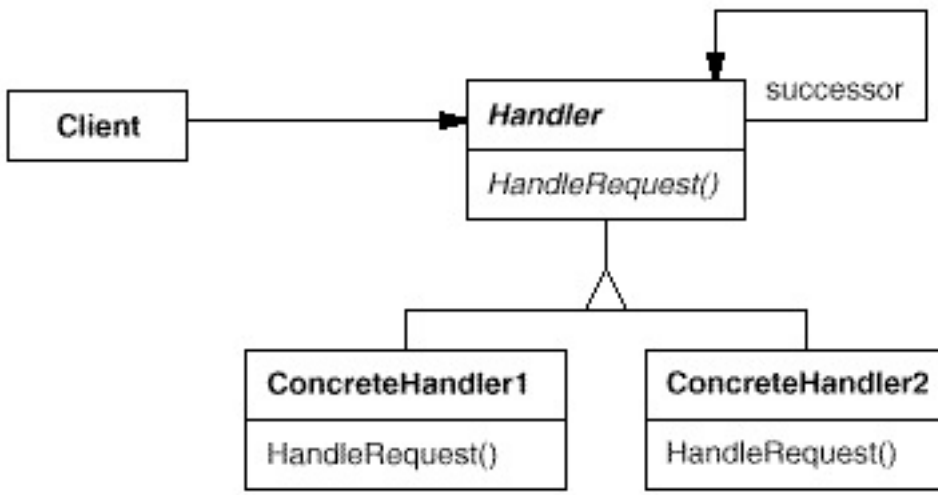
- Category
  - Behavioral
- Intent
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Motivation

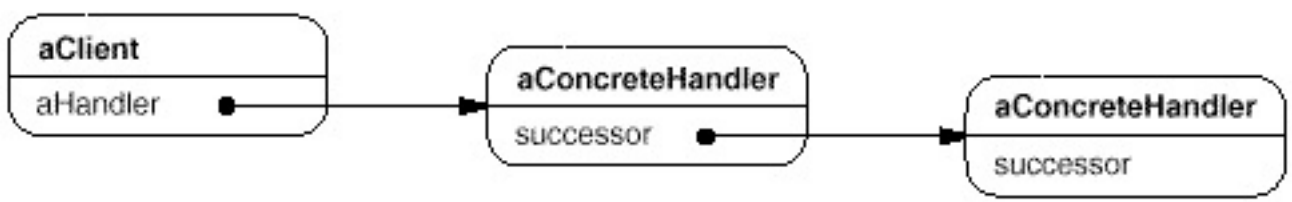


Applicability

- Use Chain of Responsibility when
  - more than one object may handle a request, and the handler is not known a priori.
  - you want to issue a request to one of several objects without specifying the receiver explicitly.
  - the set of objects that can handle a request should be specified dynamically.

Structure


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

Participants

- Handler
  - defines an interface for handling objects.
  - (optional) implements the successor link.
- ConcreteHandler
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so, otherwise it forwards the request to its successor.
- Client
  - initiates the request to a ConcreteHandler object on the chain.

Collaborations and Consequences

- Collaborations
  - When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility to handle it.
- Consequences
  - Reduced Coupling
    - The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled appropriately.

Consequences (cont)

- Added flexibility in assigning responsibilities to objects.
  - You can add or change responsibilities for handling a request by adding or changing the chain at runtime.
- Receipt is not guaranteed.
  - Since a request has no implicit receiver, there is no guarantee that it will be handled, it could fall of the end of the chain without being handled.

Example Code

```
public class Topic {

    static int NO_HELP_TOPIC = -1;

    static int PRINT_TOPIC = 1;

    static int PAPER_ORIENTATION_TOPIC = 2;

    static int APPLICATION_TOPIC = 3;

}
```

Example Code

```
public class Helphandler {
    private Helphandler successor = null;
    private int topic = Topic.NO_HELP_TOPIC;
    protected Helphandler();
    public Helphandler(Helphandler h, int topicValue){
     if(h != null){
           successor = h;
     }
     topic = topicValue;
    }
    boolean hasHelp(){
     return topic != Topic.NO_HELP_TOPIC;
    }
    void setHandler(Helphandler h, int topicValue){
     successor = h;
     topic = topicValue;
    }
    void handleHelp(){
     if (successor != null){
           successor.handleHelp();
     }
    }
}
```

Example Code

```
public class Widget extends Helphandler {
    private Widget parent;
    protected Widget();
    protected Widget(Widget w, int topicValue){
     super(w, topicValue);
     parent = w;
     }
}
public class Button extends Widget{
    public Button (Widget d, int topicValue){
     super(d, topicValue);
     }
    public void handleHelp(){
     if (hasHelp()){
           System.out.println("Button displays topic.");
     }
    else {
     super.handleHelp();
     }
```

Example Code

```
public class Dialog extends Widget{
    public Dialog(Helphandler h, int topicValue){
     setHandler(h, topicValue);
    }
    public void handleHelp(){
     if (hasHelp()){
           System.out.println("Dialog displays topic.");
     }
     else {
           super.handleHelp();
     }
    }
}
```

Example Code

```
public class Application extends Helphandler{

    public Application (int topicValue){

     super(null, topicValue);

    }

    public void handleHelp(){

     System.out.println("Application displays all possible topics.");

    }

}
```

Example Code

```
public class Main {
    public static void main(String[] args) {
     Application a = new Application(Topic.APPLICATION_TOPIC);
     Dialog d1 = new Dialog(a, Topic.PRINT_TOPIC);
     Button b1 = new Button(d1, Topic.PAPER_ORIENTATION_TOPIC);
     b1.handleHelp();
     Button b2 = new Button(d1, Topic.NO_HELP_TOPIC);
     b2.handleHelp();
     Dialog d2 = new Dialog(a, Topic.NO_HELP_TOPIC);
     Button b3 = new Button(d2, Topic.NO_HELP_TOPIC);
     b3.handleHelp();
    }
}
```

Button displays topic.

Dialog displays topic.

Application displays all possible topics.

Known Uses

- Different class libraries use this pattern, giving different names to handlers, e.g. when a user clicks on a mouse button, an event gets generated and passed along the chain.
- Is also used in graphical systems, where a graphical object propagates the request for an update to its enclosing container object, because that object has more information about its context.

Questions

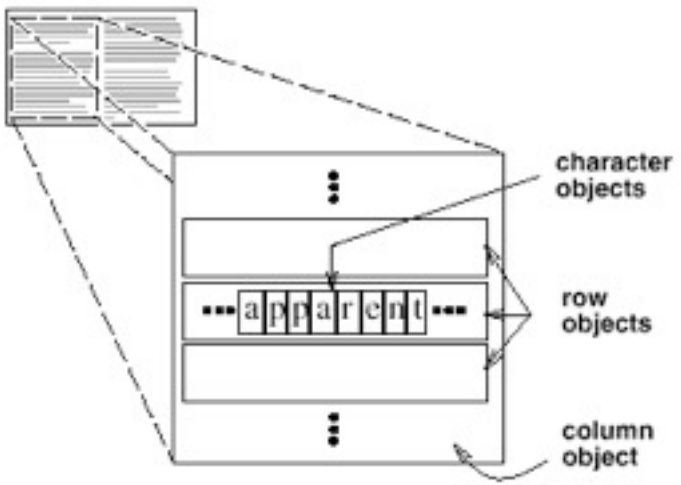
- What pattern(s) would you use in combination with the Chain of Responsibility? Why?

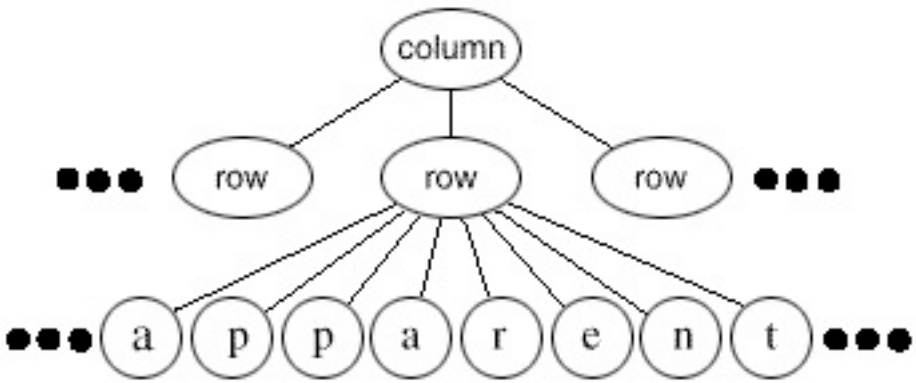
Flyweight

Flyweight

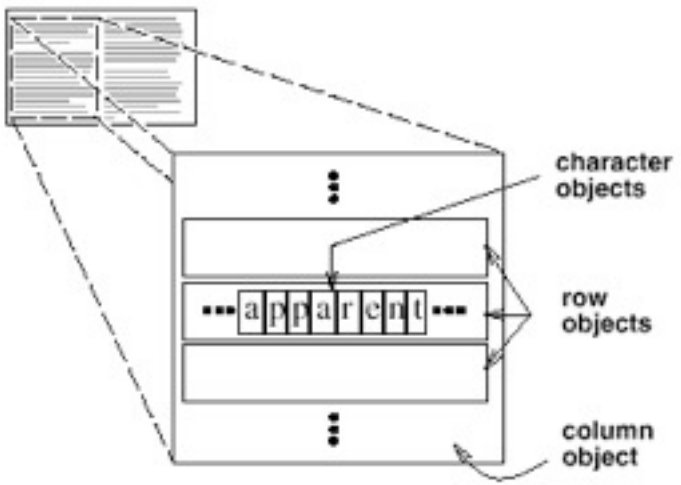
- Category
  - Structural
- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Motivation
  - Some applications benefit from using objects in their design but a naive implementation is prohibitively expensive because of the large number of objects.
  - For example a document editor uses an object for each character in the text.

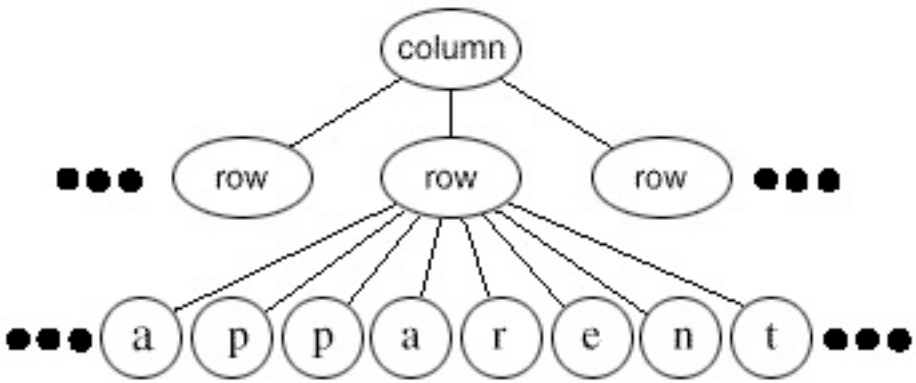
Motivation (cont)

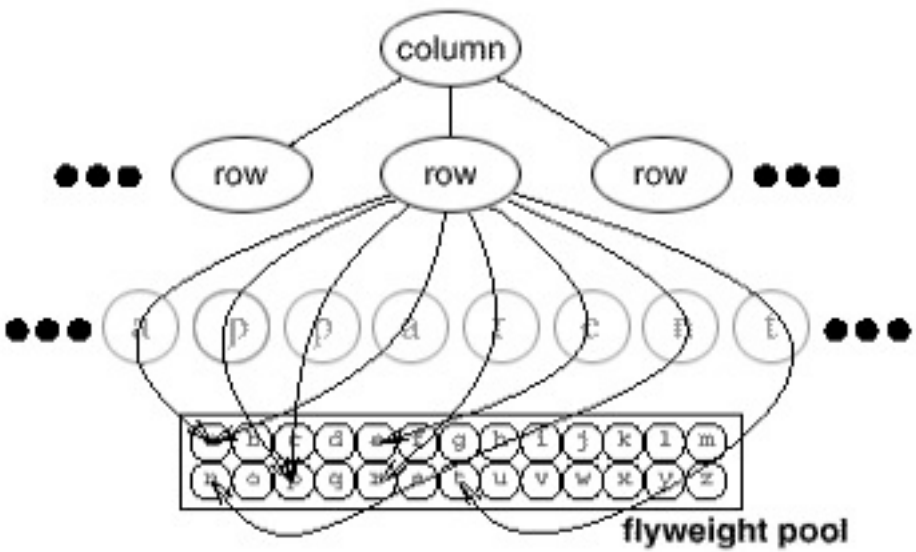




Motivation (cont)



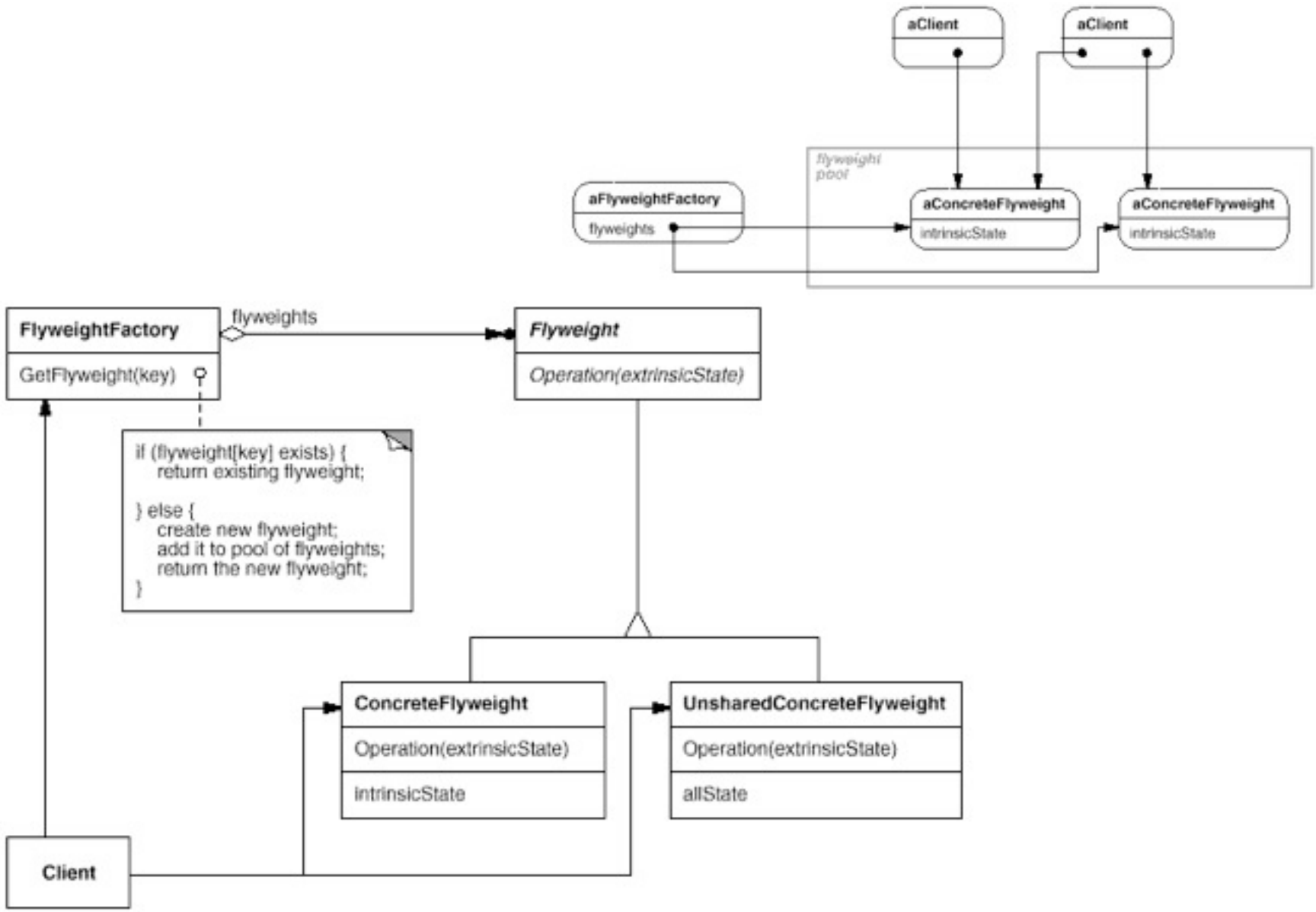




Applicability

- Apply the Flyweight pattern when **all** of the following are true:
  - An application uses a large number of objects.
  - Storage cost is high because of the quantity of objects.
  - Most objects can be made extrinsic.
  - Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed.
  - The application does not depend on object identity.

Structure


aClient
aClient
flyweight pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState
FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Participants

- Flyweight
  - Declares an interface through which flyweights can receive and act upon extrinsic state.
- Concrete Flyweight
  - Implements the flyweight interface and adds storage for intrinsic state.
  - A concrete flyweight object must be shareable, i.e. state must be intrinsic.
- Unshared Concrete Flyweight
  - Not all flyweights subclasses need to be shared, unshared concrete flyweight objects have concrete flyweight objects at some level in the flyweight object structure.

Participants (cont)

- Flyweight Factory
  - Creates and manages flyweight objects.
  - Ensures that flyweights are shared properly; when a client requests a flyweight the flyweight factory supplies an existing one from the pool or creates one and adds it to the pool.
- Client
  - Mainrains a reference to flyweight(s).
  - Computes or stores the extrinsic state of flyweight(s).

Collaborations

- State that a flyweight needs to function must be characterised as either intrinsic or extrinsic. Intrinsic state is stored in the concrete flyweight object; extrinsic state is stored or computed by client objects. Clients pass this state to the flyweight when invoking operations.
- Clients should not instantiate concrete flyweights directly. Clients must obtain concrete flyweight objects exclusively from the flyweight factory object to enshure that they are shared properly.

Consequences

- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state.
- The increase in run-time cost are offset by storage savings which increase
  - as more flyweights are shared.
  - as the amount of intrinsic state is considerable.
  - as the amount of extrinsic state is considerable but can be computed.

Consequences (cont)

- The flyweight pattern is often combined with the composite pattern to build a graph with shared leaf nodes. Because of the sharing, leaf nodes cannot store their parent which has a major impact on how the objects in the hierarchy communicate.

Example Code

- We will see a nice example of a Flyweight in the exercises ;-)

Known Uses

- Has been used in e.g. document editors. When first introduced in such an editor, only the style and and character code of the characters were intrinsic, while the position of the characters was extrinsic. This made the program very fast. In a document containing 180.000 characters, only 480 character objects had to be allocated.
- Can also be used to abstract the look and feel of layouts. Only the objects of the flyweight pool have to be replaced to change a complete layout.

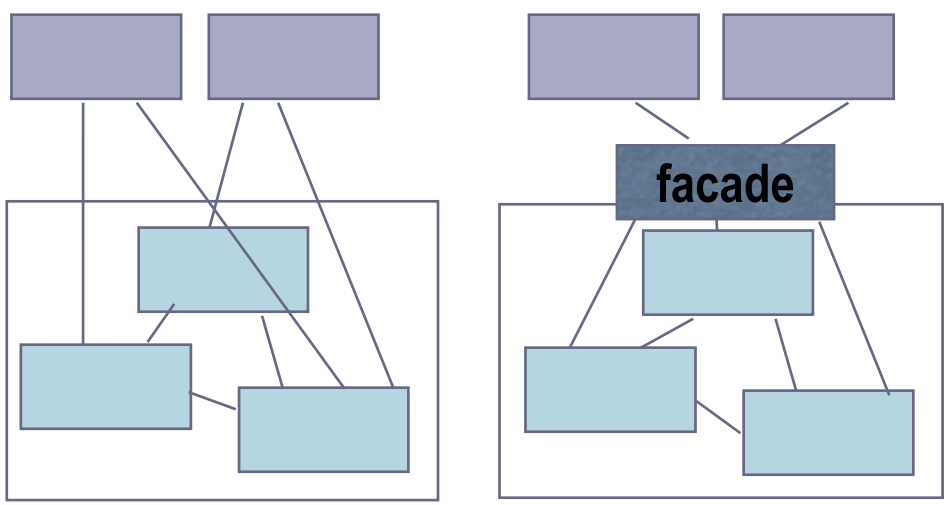
Questions

- Give a non-GUI example of a flyweight.
- What is the minimum configuration for using flyweight? do you need to be working with thousands of objects, hundreds, tens?
- Suppose you have to implement a texteditor. The text of the texteditor consists of lines and characters on the lines.

Facade

Facade

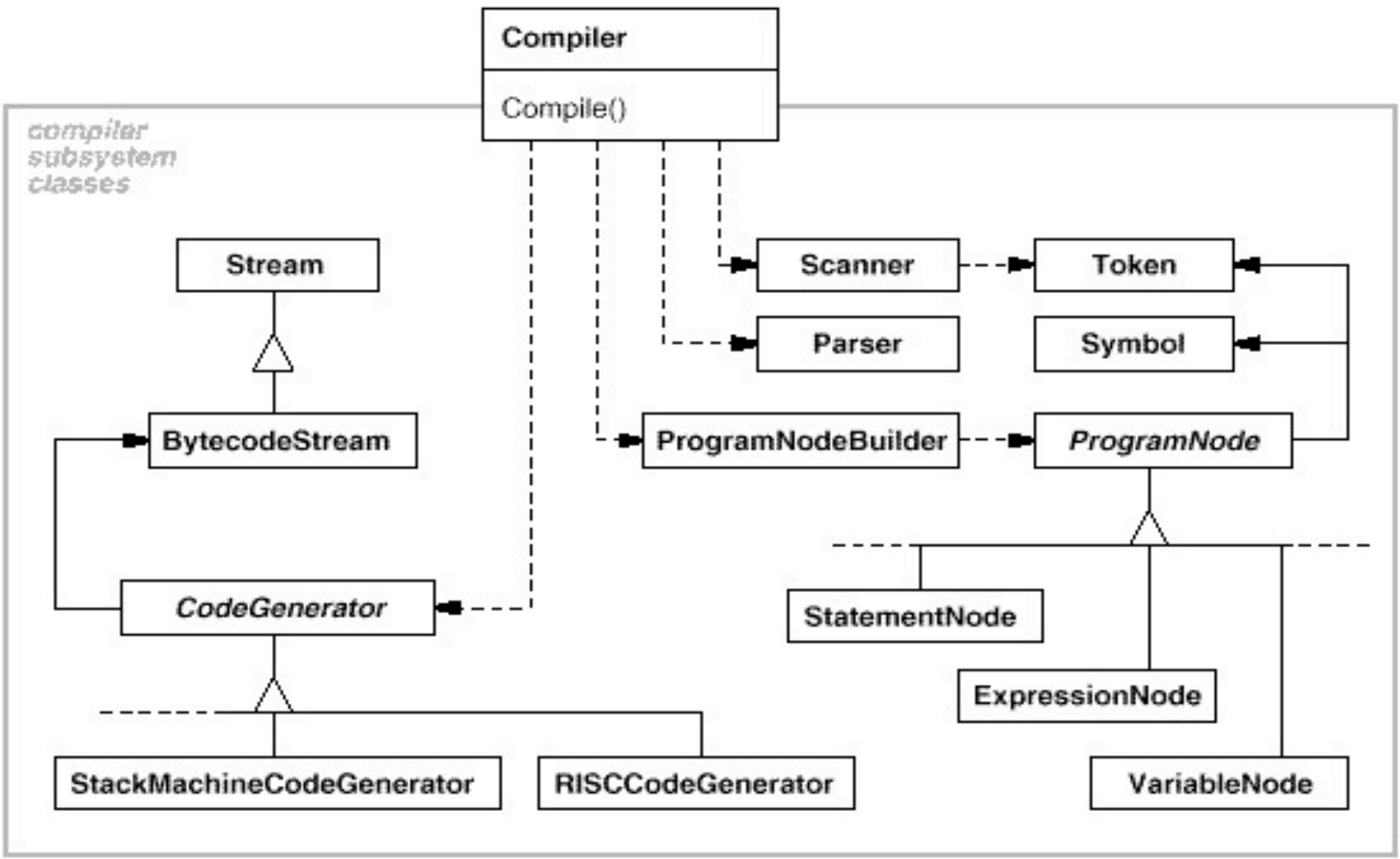
- Category
  - Structural
- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Motivation



Motivation (cont)

- Provide a simple interface to a complex subsystem.
- Decouple a subsystem from clients and other subsystems.
- Create layered subsystems by providing an interface to each subsystem level.

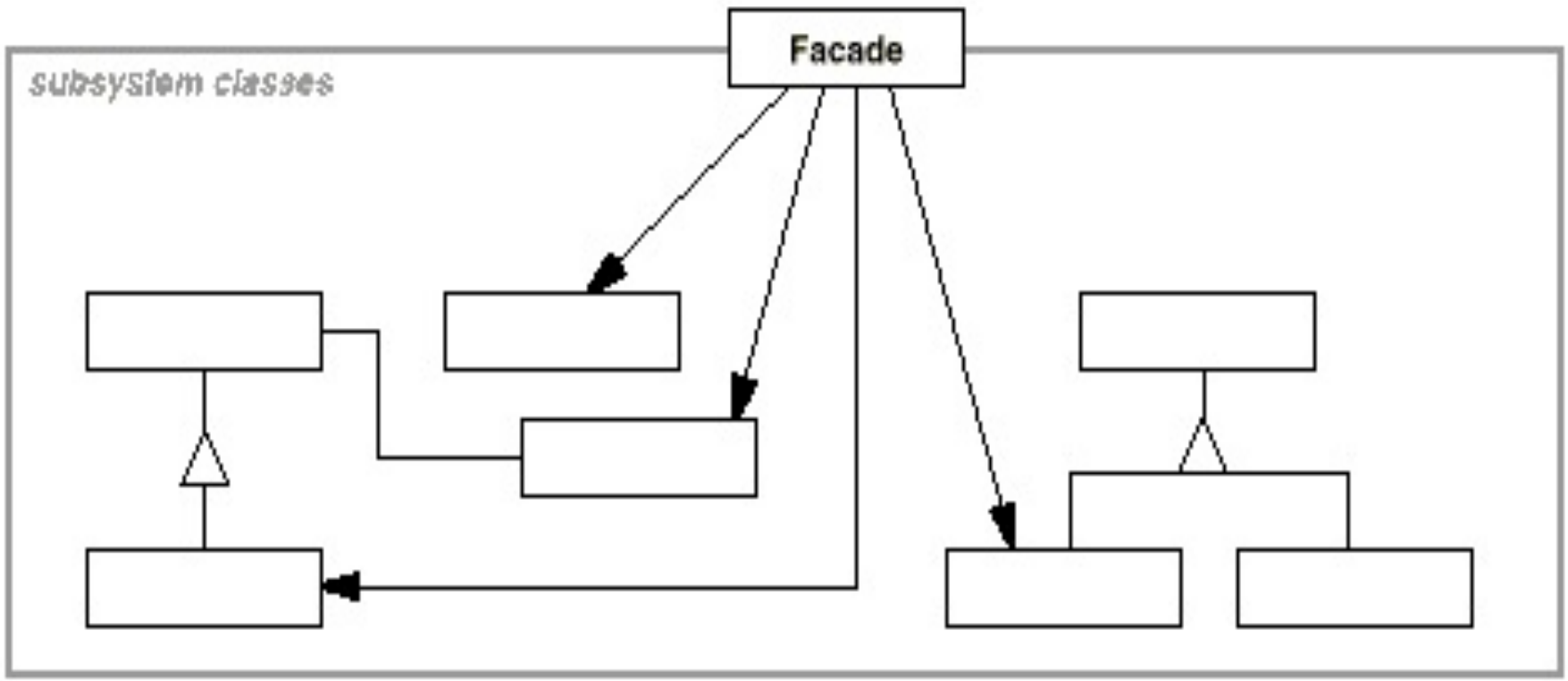
Example



Applicability

- Use the Facade pattern when
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
  - you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants

- **Facade**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

- The Facade pattern offers the following benefits:
  - It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
  - It promotes weak coupling between the subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
  - It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Example Code

```
public class Compiler {

    public void Compile(InputStream input, OutputStream output){

     Scanner scanner = new Scanner(input);

     ProgramNodeBuilder builder;

     Parser parser;

           parser.parse(scanner, builder);

           RISCCodeGenerator generator = new RISCCodeGenerator(output);

     ProgramNode parseTree = builder.GetRootNode();

     parseTree.traverse(generator);

     }

}
```

Known Uses

- We have seen the compiler example, but this pattern can be used for other complicated frameworks as well.

Questions

- Describe the differences between Facade and Adapter.
- How complex must a sub-system be in order to justify using a facade?
- What are the additional uses of a facade with respect to an organization of designers and developers with varying abilities? What are the political ramifications?

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 ?