Design patterns

Valentina Presutti
courtesy of Paolo Ciancarini

Agenda

- What are design patterns?
- Catalogues of patterns
- Languages of patterns
- Two case studies: design with patterns

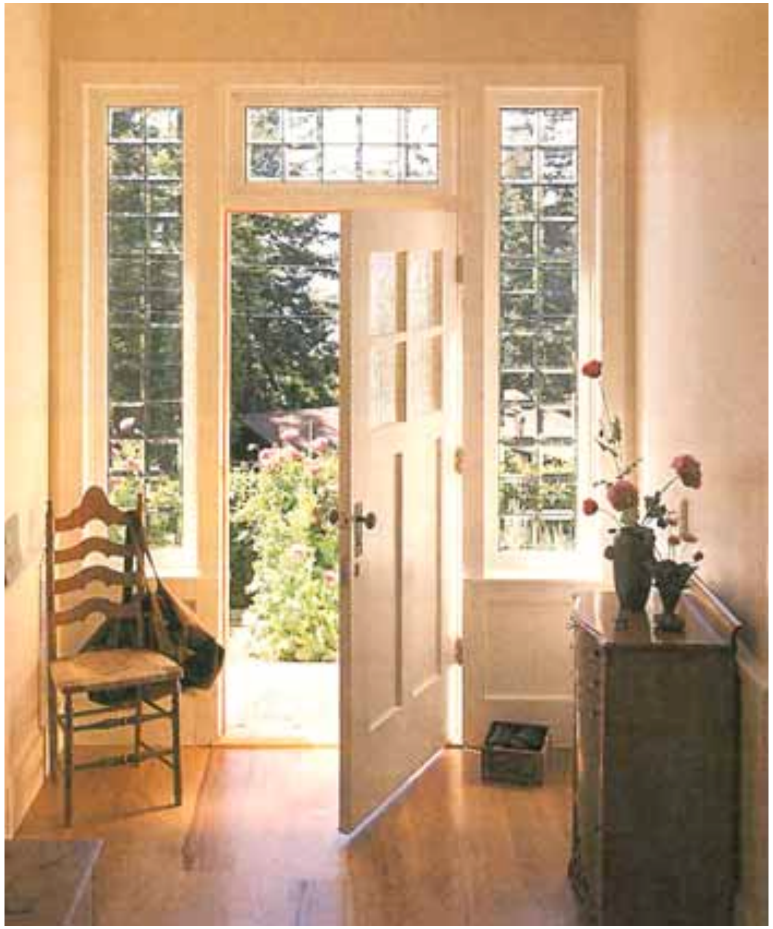
Software Architectures

- A **Software Architecture** provides a fundamental description of a system, detailing the components that make up the system and the meaningful collaborations among those components, including the data and control flows of the system
- The discipline of **Software Architecture** attempts to provide a sound basis for analysis, decision making, and risk assessment of both design and performance

Architectural archetypes

- The design of a software architecture can be driven by other architectures
- We call an abstract reusable architecture and its behavior an “architectural archetype”
- On a lower scale, reusable micro-architectures are called “**design patterns**”

Patterns in Architecture



- Does this room interior make you feel happy?
- Why?
  - Light (direction)
  - Proportions
  - Symmetry
  - Furniture
  - And more…

Christopher Alexander

- The architect Alexander introduced the concept of a **pattern language** in his book *The Timeless Way of Building*.
- His notion of patterns as best practice approaches to common design problems has been adopted by the software community
- He introduces the concept of a pattern language, as a related set of patterns that together provide a vocabulary for designing within a certain context or problem domain.
- In the book *A Pattern Language*, Alexander introduces a specific pattern language that includes sub-languages for designing towns (targeted to planners) and buildings (targeted to architects), and a sub-language for construction (targeted to builders).
- Examples of Alexander's patterns are:
  - **Town patterns** Ring roads, night life, and row houses
  - **Building patterns** Roof garden, indoor sunlight, and alcoves
  - **Construction patterns** Good materials, column connection, and half-inch trim

What is a Design Pattern?

A description of a recurrent problem and of the core of possible solutions

In short, a solution for a typical design problem

Discuss

What are typical software design problems?

Typical design problems

- This object should inform all its clients of its changes of state
- This class should create a unique object, namel it should remain unique in the system
- These high level objects should not know all those low level objects, but still they should be able to pass some data
- This object should change its behavior “dynamically”

Typical design problems

- What design issues can I reuse?
- Which design vocabulary can I use?
- How can I say that I found a good design solution?
- How can I simplify a design?

Design Patterns

Design patterns are reusable ("template") designs that can be used in a variety of ways in different systems. They are appropriate in situations where classes are likely to be reused in a system that evolves over time

a) Name. [Some pattern names have become standard software design terminology]

b) Problem description. Describes when the pattern might be used, often in terms of modifiability and extensibility

c) Solution. Expressed in terms of classes and interfaces

d) Consequences. Trade-offs and alternatives

Design patterns

- A pattern is a proven solution to a problem in a context
- Alexander says each pattern is a three-part rule which expresses a relation between a certain **context**, a **problem**, and a **solution**
- i.e Patterns = (*problem*, *solution*) pairs in a *context*

- In the field of software design, design patterns represent proven solutions to problems that arise when developing software within a particular context

Background

- In 1987 Cunningham and Beck worked with Smalltalk and found some patterns when designing GUIs
- Concept popularized in a book by Gamma, Helm, Johnson and Vlissides (The “Gang of four”, Go4): they were working on *frameworks* (E++, Unidraw, HotDraw)
- Design patterns use a consistent documentation approach and are often organized as *creational*, *structural* or *behavioral*
- Design patterns are applied at different levels such as frameworks, subsystems, and architectural archetypes

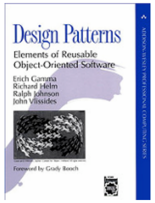
The Gang of Four

Ralph, Erich, Richard, and John at OOPSLA 1994

Why do designers need patterns?

- Reusing design knowledge
  - Problems are not always unique. Reusing existing experience might be useful
  - Patterns give us hints to “where to look for problems”
- Establish a common terminology
  - Easier to say, "We need a Façade here“
- Provide a higher level prospective
  - Frees us from dealing with the details too early
- Patterns are a “design reference”

Evolution of Design Patterns

Christopher Alexander *The Timeless Way of Building* *A Pattern Language: Towns, Buildings, Construction*	Architecture	1970’
Gang of Four (GoF) *Design Patterns: Elements of Reusable Object-Oriented Software* 	Object Oriented Software Design	1995’
Many Authors	Other Areas: Middleware, HCI, Management, Education, …	2000’

Categorizing Patterns

Patterns represent expert solutions to recurring problems in a context and thus have been captured at many levels of abstraction and in numerous domains. Some categories are:

- Design
- Architectural
- Analysis
- Creational
- Structural
- Behavioral

Structure of a design pattern*

a) Pattern Name and Classification

b) Intent
   - a short statement about what the pattern does

c) Motivation
   - A scenario that illustrates where the pattern would be useful

d) Applicability
   - Situations where the pattern can be used

*According to GoF

Structure of a design pattern

e) Structure
   - A ***graphical*** representation of the pattern

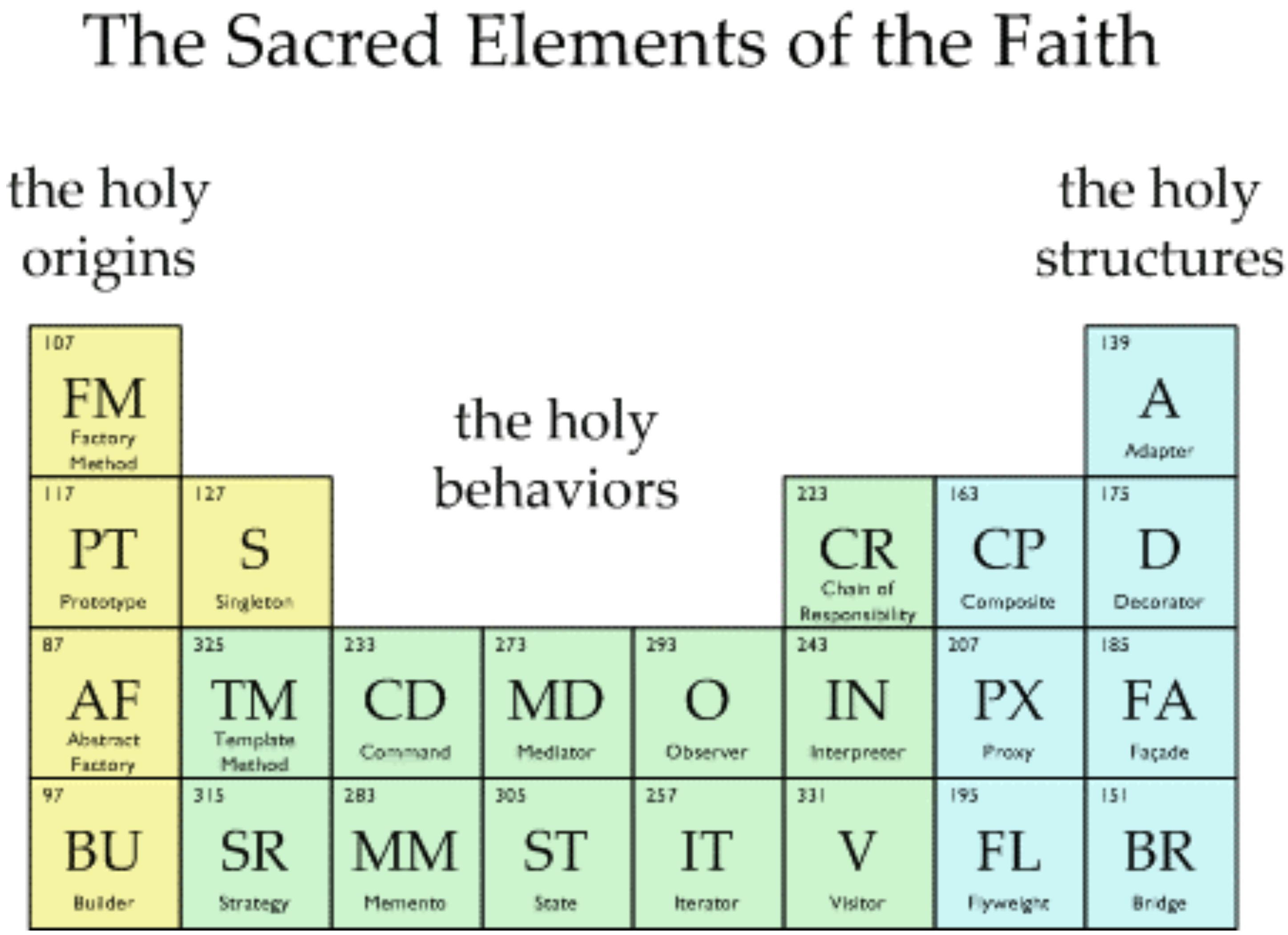
- Participants
   - The classes and objects participating in the pattern

e) Collaborations
   - How to do the participants interact to carry out their **responsibilities**?

f) Consequences
   - What are the pros and cons of using the pattern?

g) Implementation
   - Hints and techniques for implementing the pattern

Classification of GoF Patterns



The Sacred Elements of the Faith

the holy origins

the holy behaviors

the holy structures

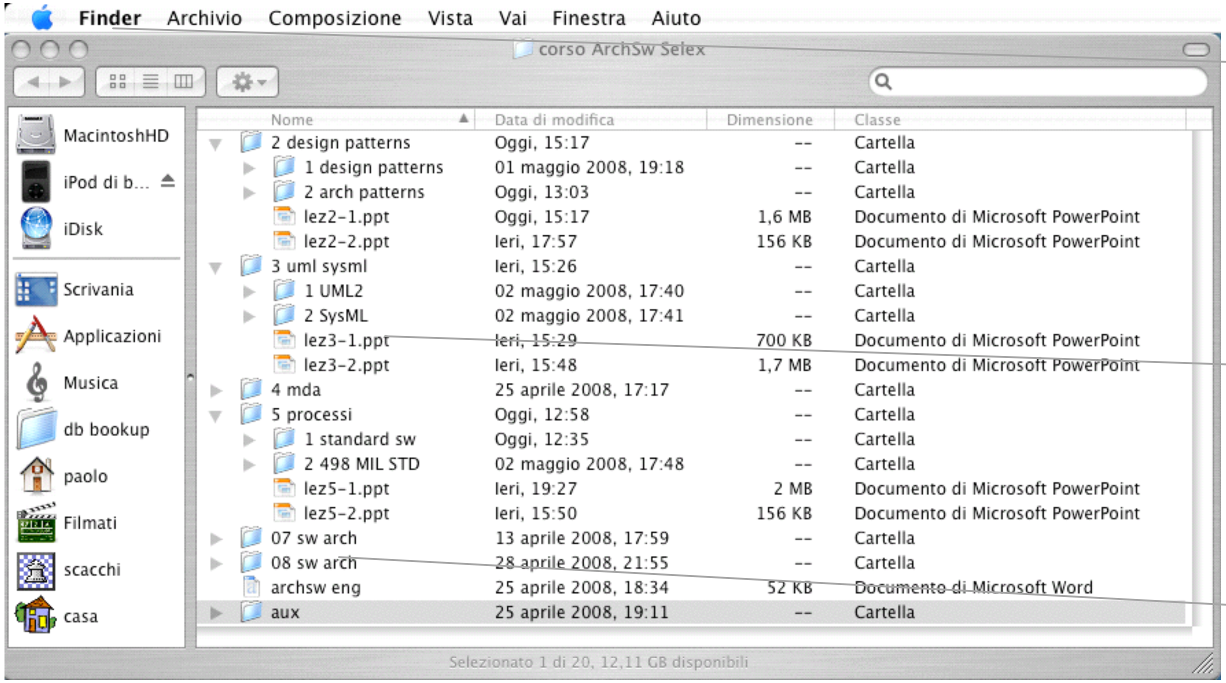
107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				223 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

home.earthlink.net/~huston2/dp/patterns.html

Main types of patterns

- **Creational**: the basic form of object creation could result in design problems or added complexity to the design. These patterns control object creation
- **Behavioral**: identify common communication patterns between objects and realize them increasing flexibility in communication
- **Structural**: ease the design by identifying simple ways to realize relationships between entities

Exercise



corso ArchSw Selex

Nome	Data di modifica	Dimensione	Classe
2 design patterns	Oggi, 15:17	--	Cartella
1 design patterns	01 maggio 2008, 19:18	--	Cartella
2 arch patterns	Oggi, 13:03	--	Cartella
lez2-1.ppt	Oggi, 15:17	1,6 MB	Documento di Microsoft PowerPoint
lez2-2.ppt	Ieri, 17:57	156 KB	Documento di Microsoft PowerPoint
3 uml sysml	Ieri, 15:26	--	Cartella
1 UML2	02 maggio 2008, 17:40	--	Cartella
2 SysML	02 maggio 2008, 17:41	--	Cartella
lez3-1.ppt	Ieri, 15:29	700 KB	Documento di Microsoft PowerPoint
lez3-2.ppt	Ieri, 15:48	1,7 MB	Documento di Microsoft PowerPoint
4 mda	25 aprile 2008, 17:17	--	Cartella
5 processi	Oggi, 12:58	--	Cartella
1 standard sw	Oggi, 12:35	--	Cartella
2 498 MIL STD	02 maggio 2008, 17:48	--	Cartella
lez5-1.ppt	Ieri, 19:27	2 MB	Documento di Microsoft PowerPoint
lez5-2.ppt	Ieri, 15:50	156 KB	Documento di Microsoft PowerPoint
07 sw arch	13 aprile 2008, 17:59	--	Cartella
08 sw arch	28 aprile 2008, 21:55	--	Cartella
archsw eng	25 aprile 2008, 18:34	52 KB	Documento di Microsoft Word
aux	25 aprile 2008, 19:11	--	Cartella

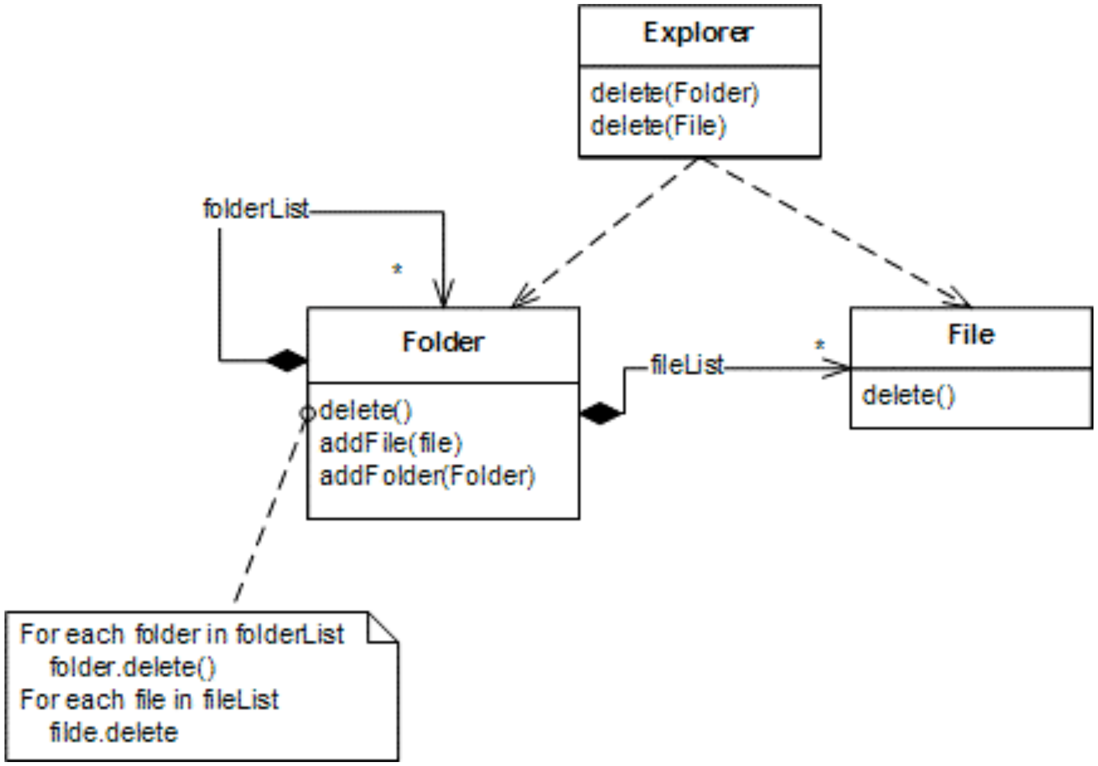
Selezionato 1 di 20, 12,11 GB disponibili

Client

Files

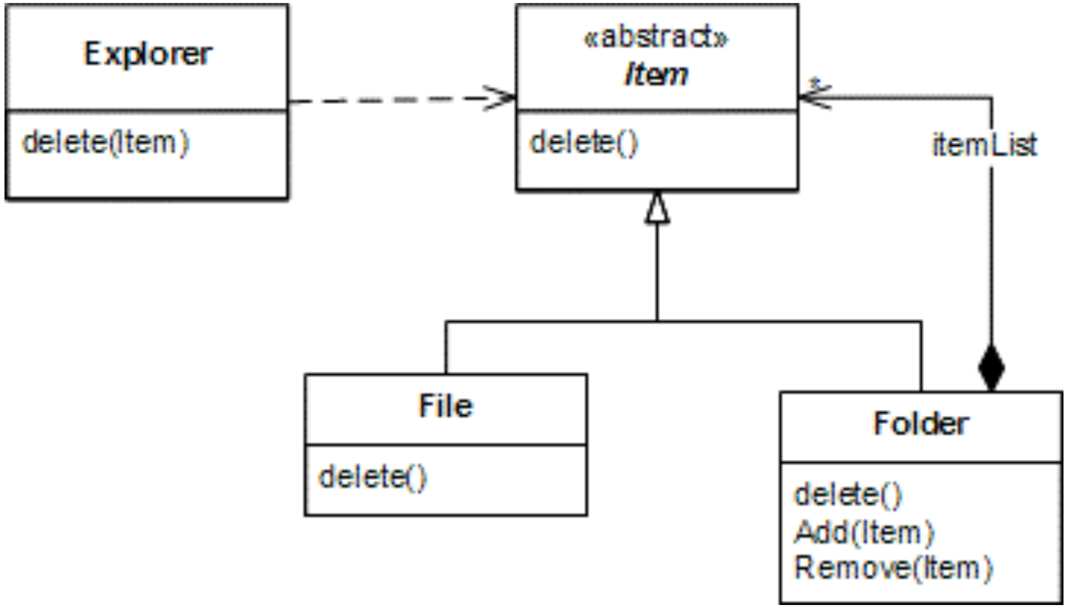
Folders

A Solution?

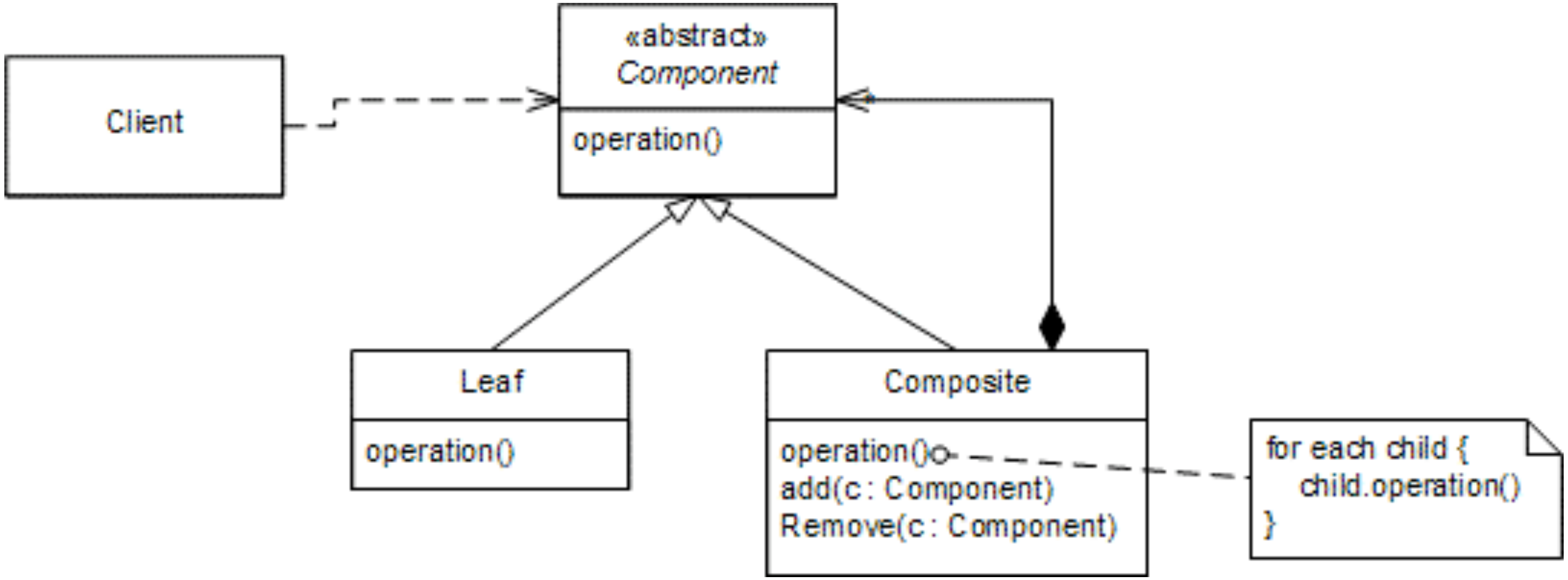


- Folder:
  - For each action (delete, display, copy etc), there is special treatment for files and folders.
- Explorer:
  - Each type of object is manipulated separately
- Scalability:
  - What if there are more types of elements (disks, CD, USB…)

A better solution



Composite: Structure



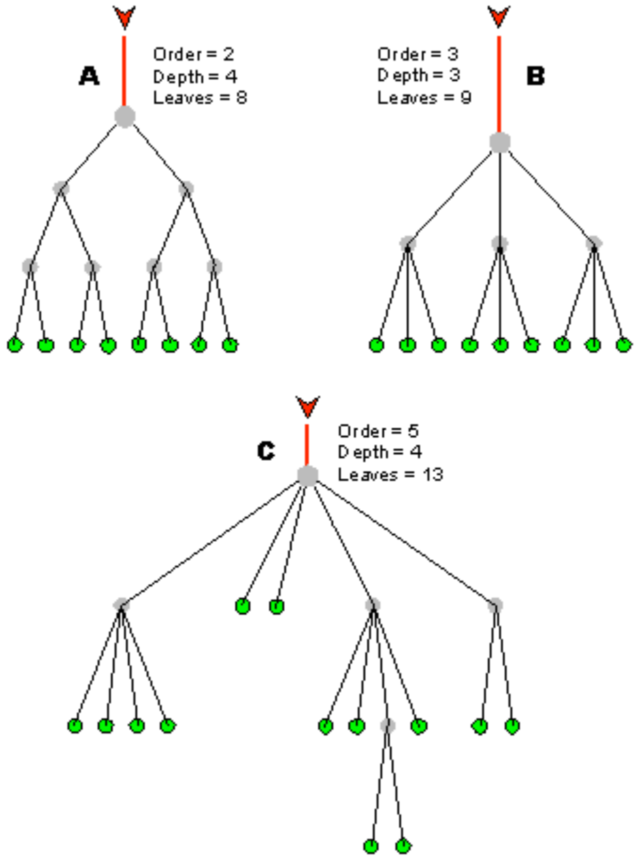
- An abstract base class (*Component*) specifies the uniform behavior
- Primitive (leaf) and Composite classes can be subclassed
- Composite manages *components* uniformly, using *add* and *remove*

Composite: Consequences

- Good
  - It makes it easy to add new kinds of components
  - It makes clients simpler. Clients can treat composite structure and individual objects uniformly
- Bad
  - It makes it harder to restrict the type of components of a composite. Sometimes you want a composite to have only certain components.

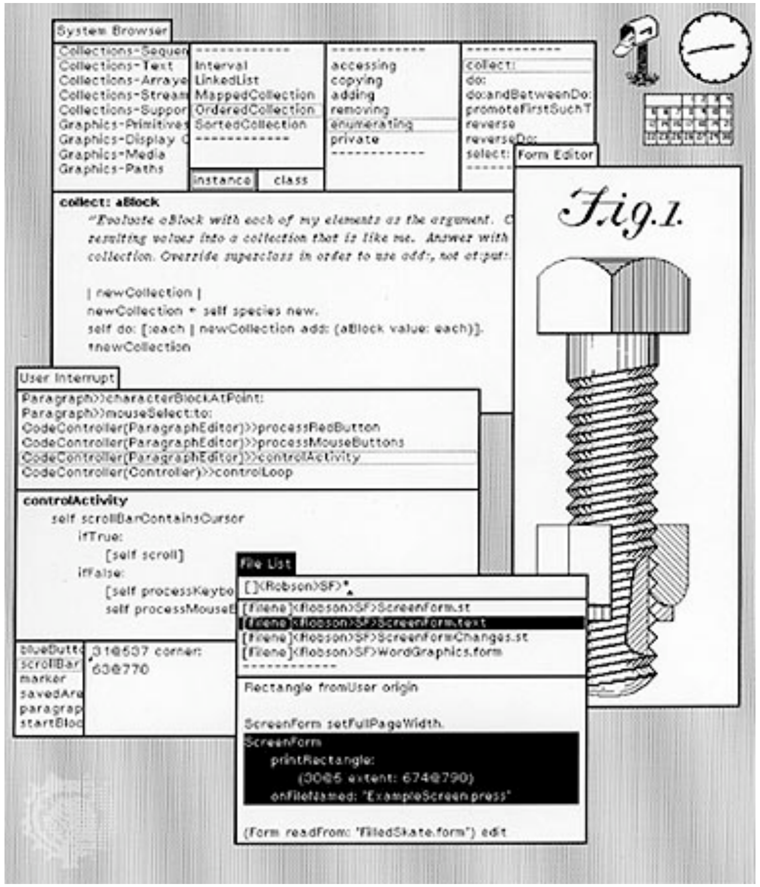
Composite: When to Use?

- Tree structures that represent part-whole hierarchies
- Uniform access



Composite: Known Uses

Graphical User Interface



```
collect: aBlock
    "Evaluate aBlock with each of my elements as the argument. C
    resulting values into a collection that is like me. Answer with
    collection. Override superclass in order to use add:, not at:put:.

    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
            [self processKeybo
            self processMouseB
```

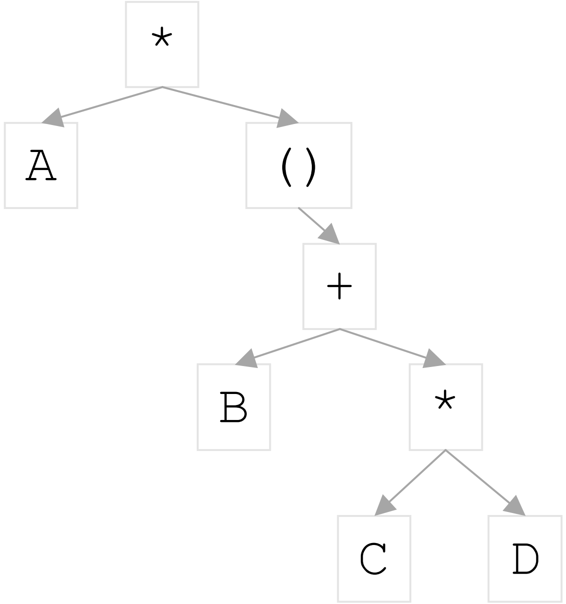
```
31@537 corner:
63@770
```

```
[]<Robson>SF>*
[Filene]<Robson>SF>ScreenForm.st
[Filene]<Robson>SF>ScreenForm.text
[Filene]<Robson>SF>ScreenFormChanges.st
[Filene]<Robson>SF>WordGraphics.form
------------
```

```
Rectangle fromUser origin

ScreenForm setFullPageWidth.
ScreenForm
    printRectangle:
        (30@5 extent: 674@790)
    onFileNamed: 'ExampleScreen.press'

(Form readFrom: 'FilledSkate.form') edit
```

Arithmetic Expressions

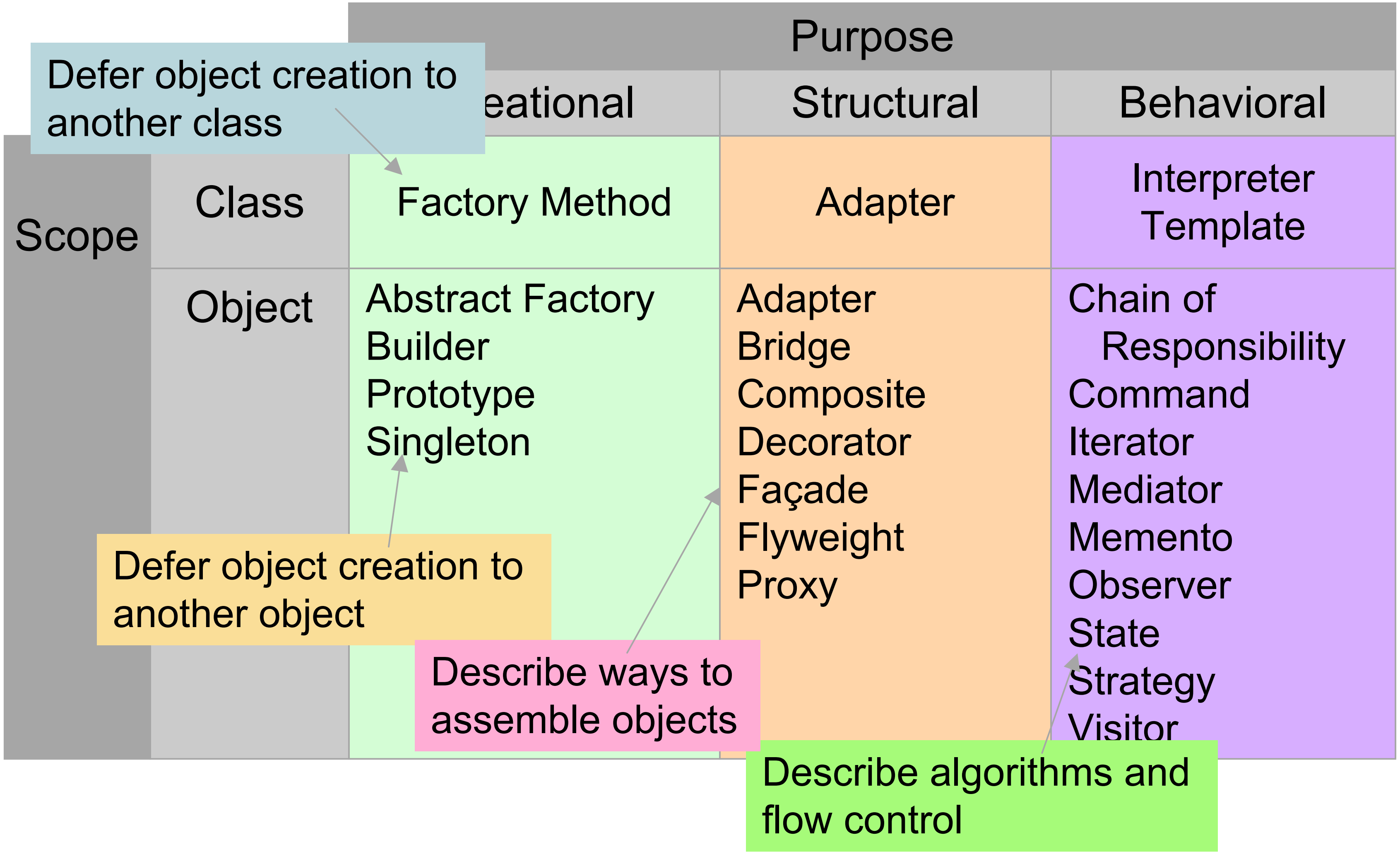
A * (B + (C * D))



Other Structural Patterns

- **Adapter** - Match interfaces of different classes
- **Bridge** - Separates an object’s interface from its implementation
- **Decorator** - Add responsibilities to objects dynamically
- **Flyweight** - A fine-grained instance used for efficient sharing
- **Proxy** - An object representing another object

The GoF pattern language



		Purpose		
		eational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Defer object creation to another class

Defer object creation to another object

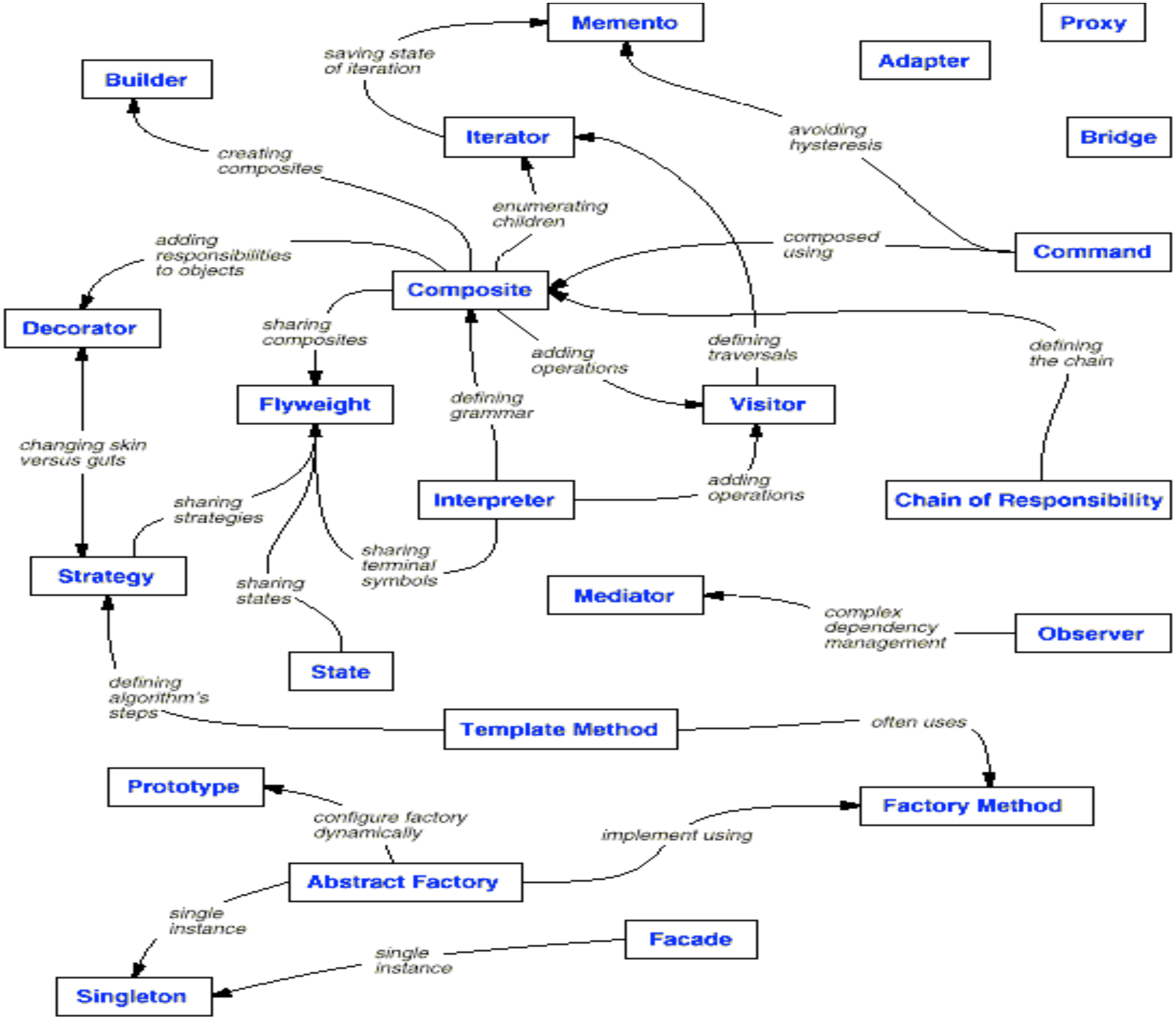
Describe ways to assemble objects

Describe algorithms and flow control

Some patterns and their use

Pattern Name	Use
Adapter	Convert the interface of one class into another interface clients expect. Adapter allows classes to work together that otherwise can’t because of incompatible interfaces.
Proxy	Provide a surrogate or placeholder for another object.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly and let one vary its interaction independently
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents will be notified and updated automatically.
Template	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Pattern relationships

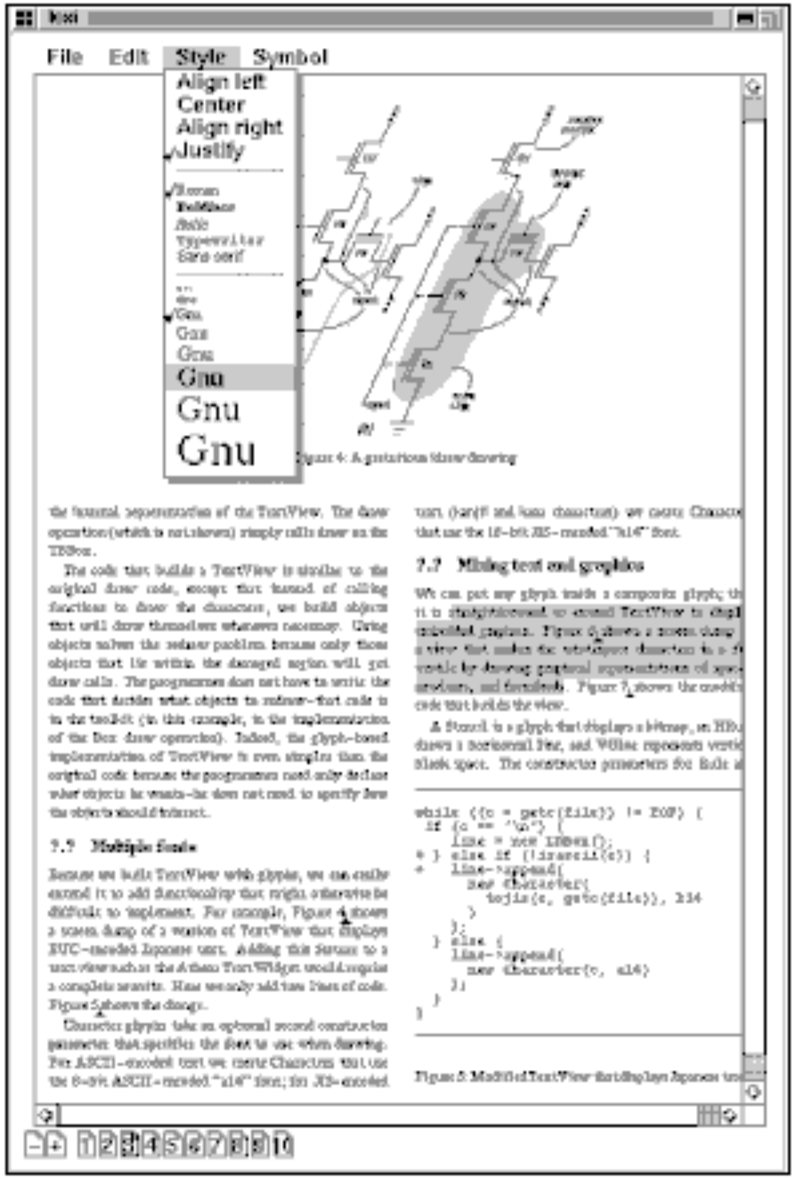


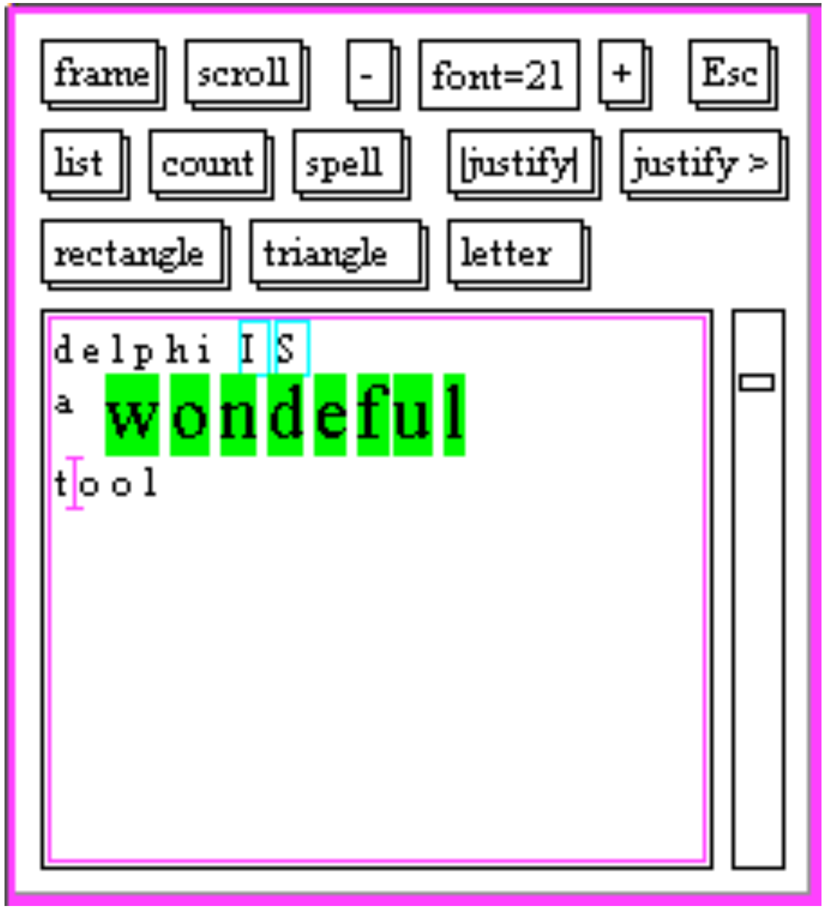
Case study 1

Text editor

- (from the GoF book)
- Design an editor for documents
- Documents include text and graphics
- GUI with multiple windows
- Several operations on texts: spelling, searching, etc.

Editor snapshots



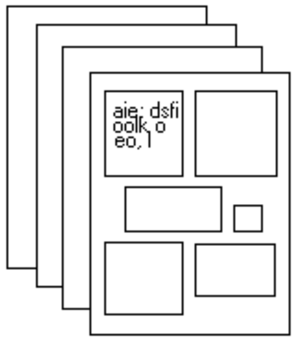


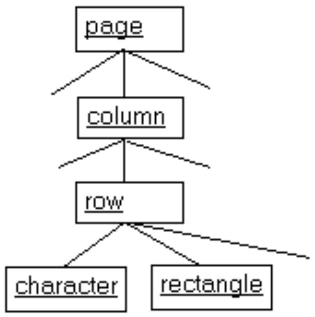
www.felix-colibri.com/papers/design_patterns/the_lexi_editor

Structure of a document

Documents have a logical structure

- Document: sequence of pages
- Page: sequence of columns
- Column: sequence of rows
- Row: sequence of gliphs
- Gliph: primitive element: char, picture, line, scrollbar, etc.





page

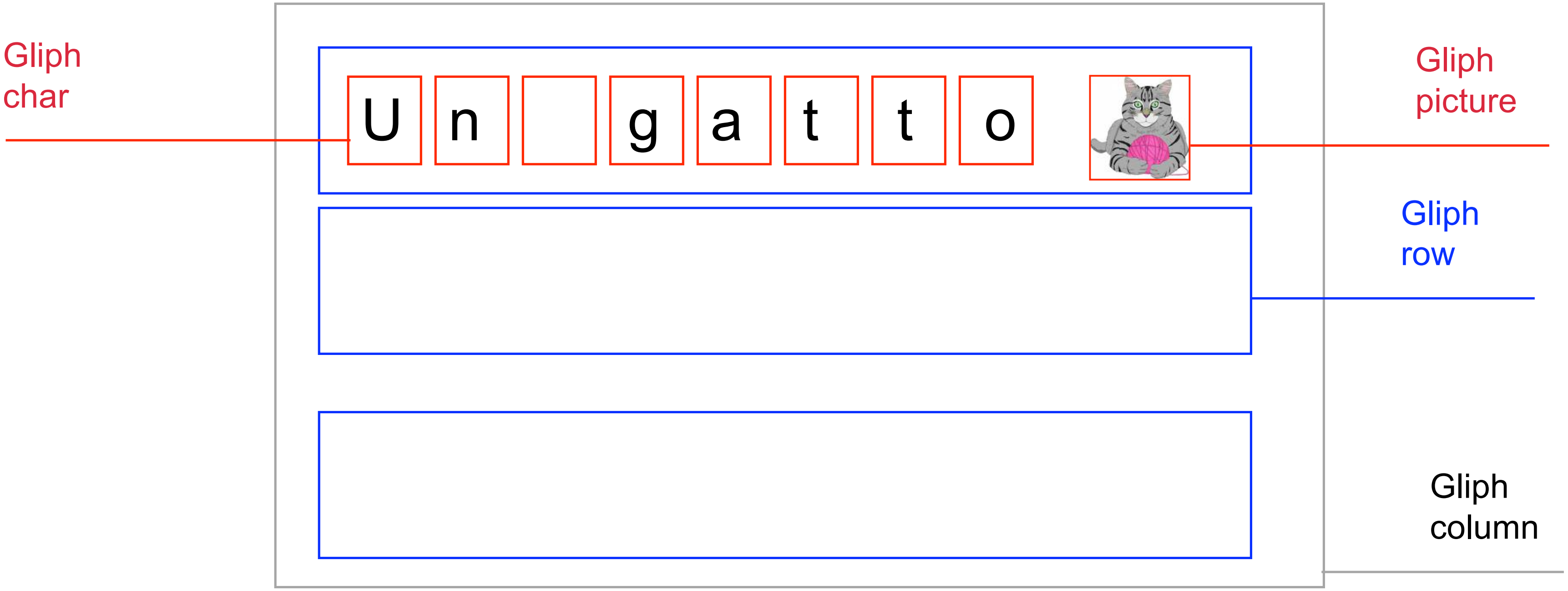
column

row

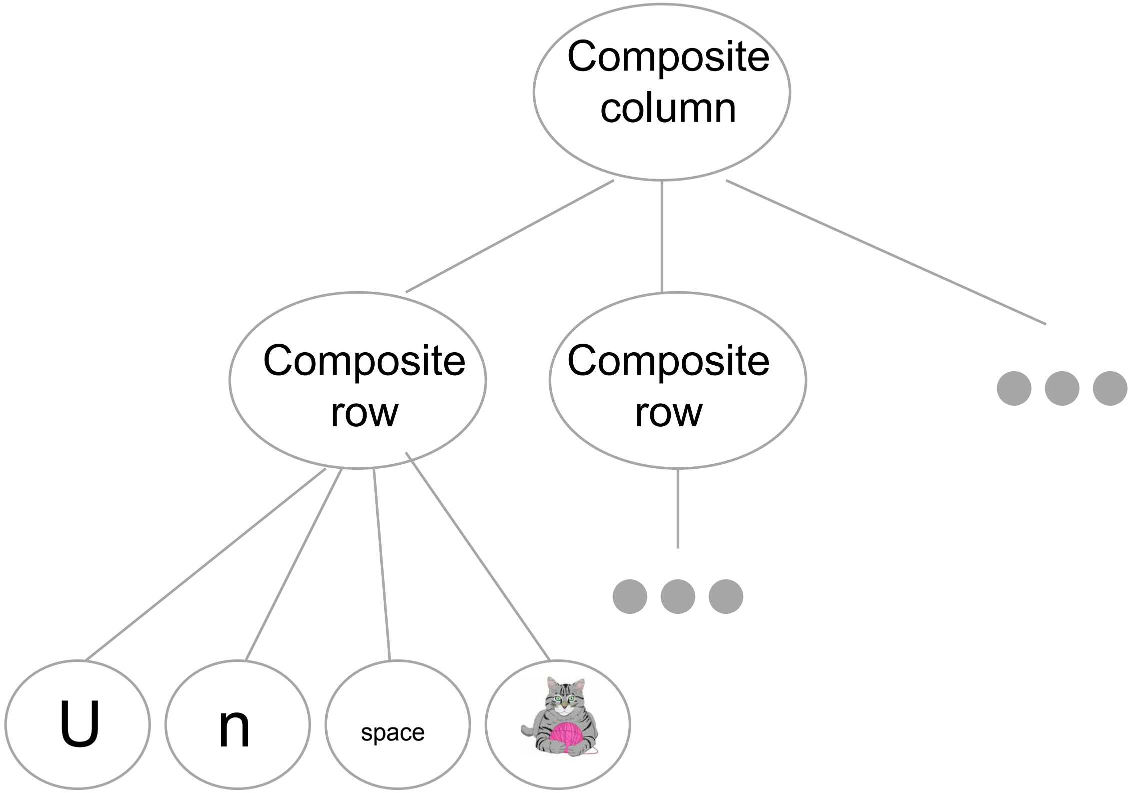
character

rectangle

Logical structure of pages



Representing the structure of a document by composites



Design alternatives

1. Different classes for each primitive element: char, line, column, page; and for gliphs like circle, square, etc.
2. Only one abstract class for a generic gliph, with unique interface to implement in different ways

Java code

```
Class Glyph{
  List children = new LinkedList();
  Int ox,oy,width,height;

  Void draw(){
    For (g:children) g.draw();
  }

  Void insert(Glyph g){
      children.add(g);
  }

  Boolean intersects(int x, int y){
    Return (x>= ox) && (x<ox+width)
    && (y>= oy) && (y<oy+height);
  }
}
```

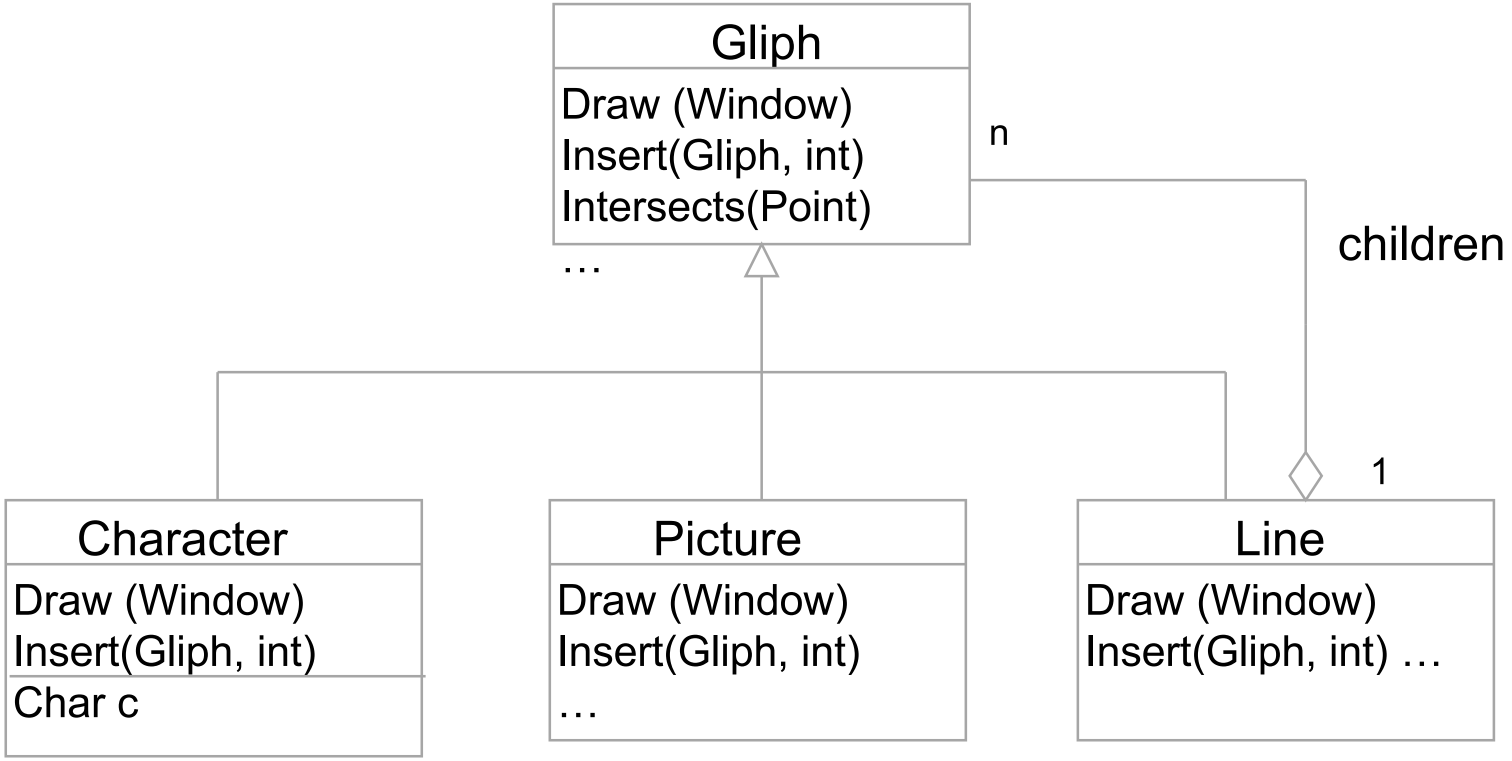
```
class Character extends Glyph {
    char c;

    public Character(char c) {
        this.c = c;
    }

    void draw() {
        …
    }
}
```

Java code

```
class Picture extends Glyph
   {
   File pictureFile;
   public Picture(File
               pictureFile) {
        this.pictureFile =
   pictureFile;
   }
   void draw() {
   // draw picture
   }
}
```

```
class Line extends
   Glyph {
   char c;
   public Line() {}
   // inherits draw, …
}
```

Diagram



Gliph

Draw (Window)
Insert(Gliph, int)
Intersects(Point)
…

n

children

1

Character

Draw (Window)
Insert(Gliph, int)

Char c

Picture

Draw (Window)
Insert(Gliph, int)
…

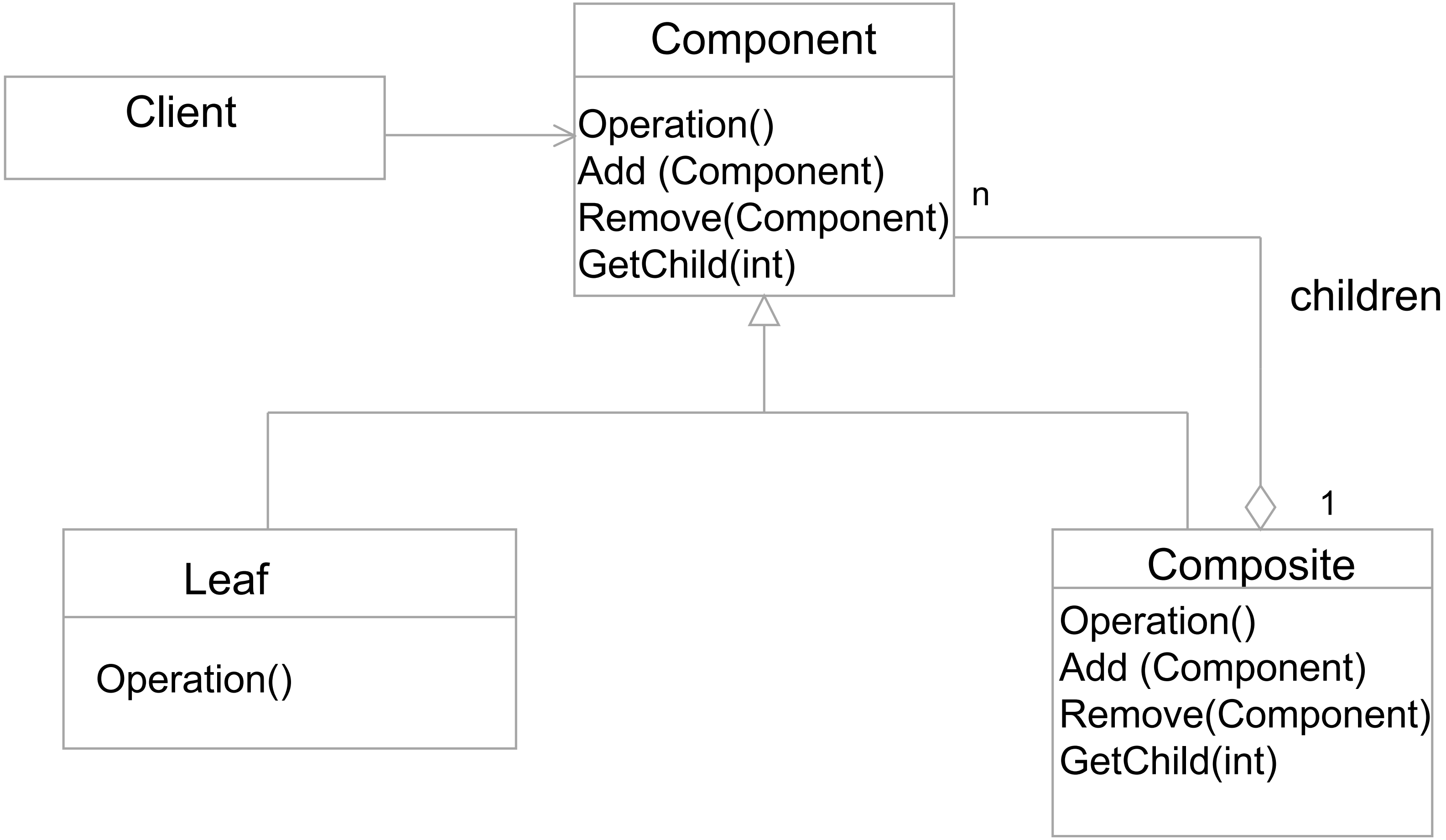
Line

Draw (Window)
Insert(Gliph, int) …

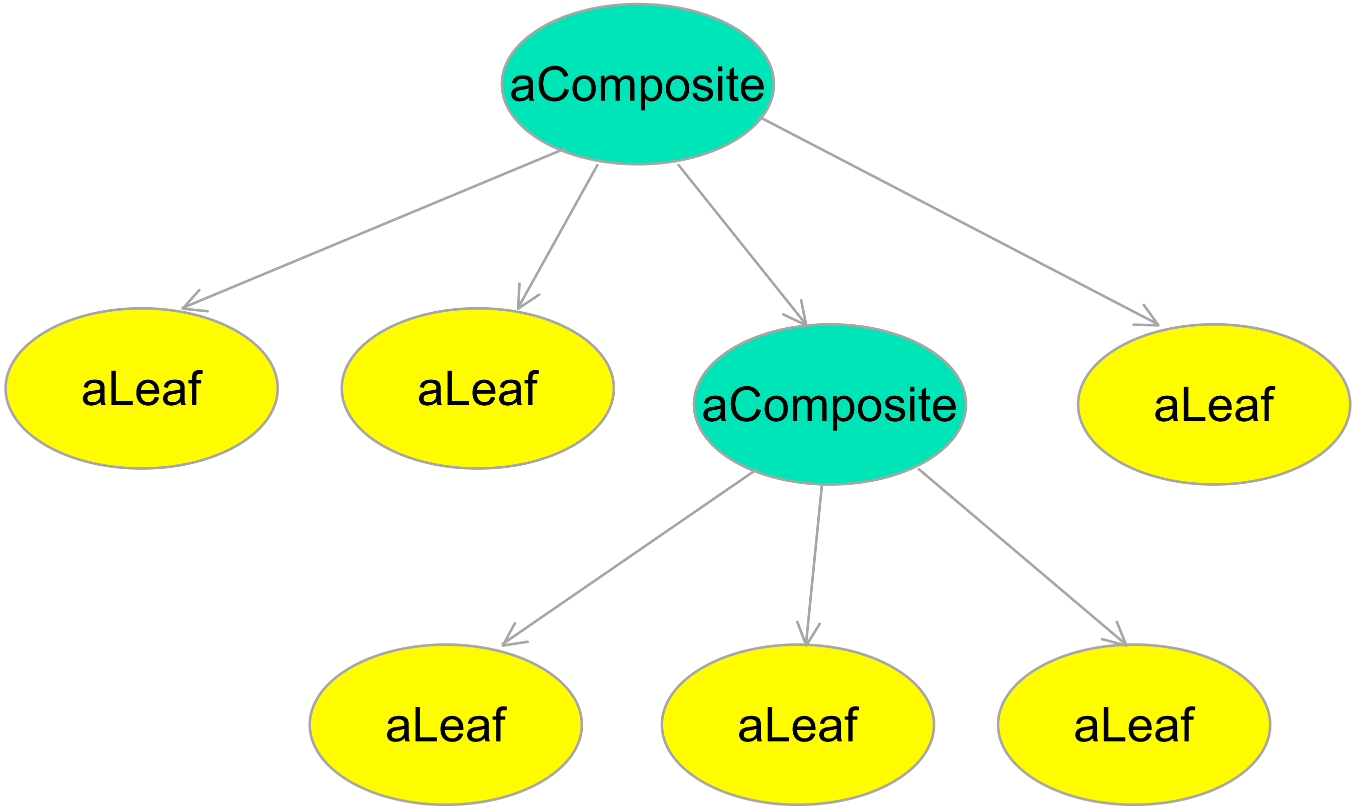
Pattern

- This is the pattern “**composite**”
  - Useful for “tree structures”, “recursive structures”, etc.
- Apply to any hierarchy
  - Leaves and nodes have the same behavior
  - Unique interface

Structure: composite pattern



Abstract data structure “Composite”

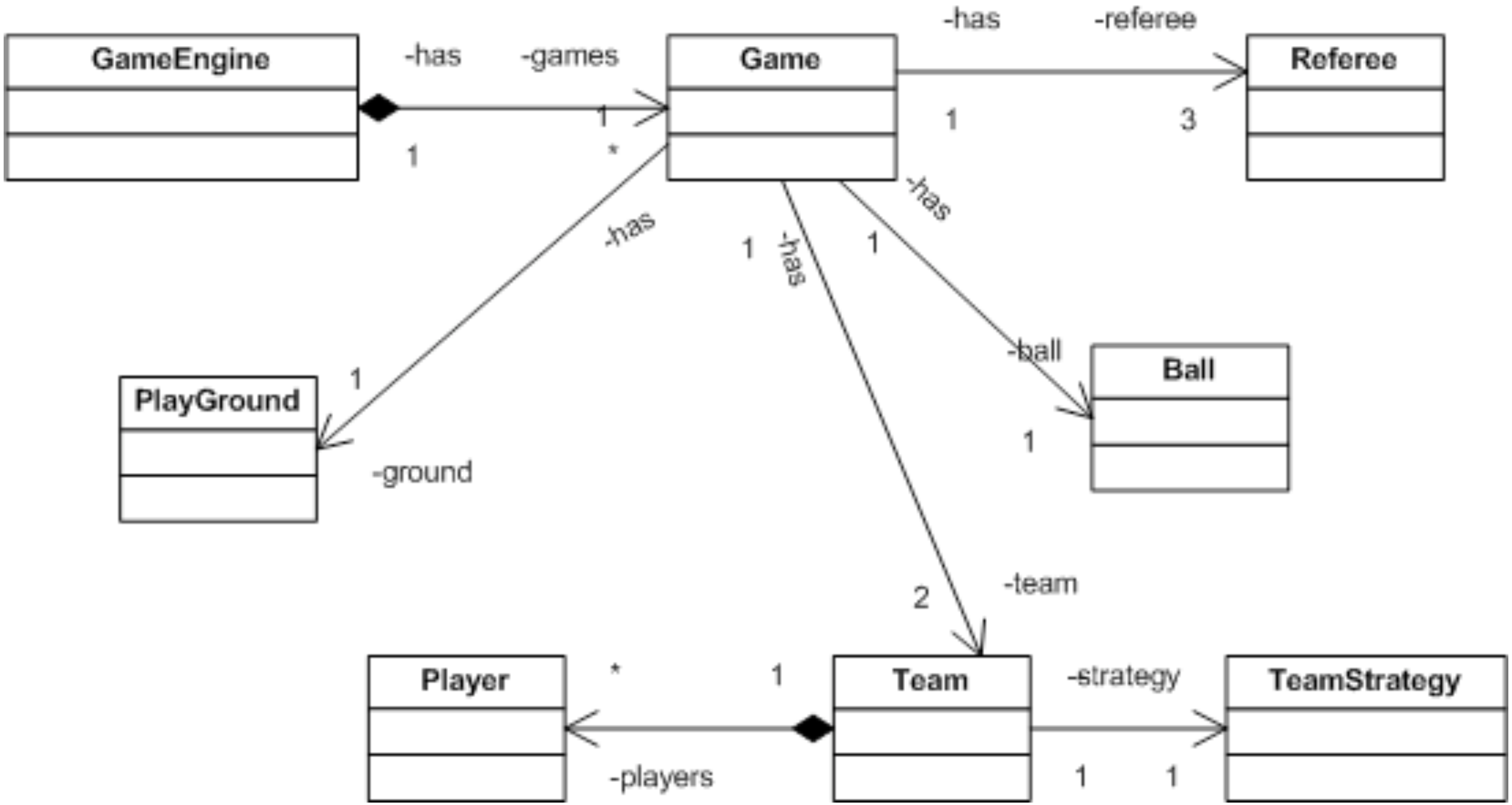


Case study 2

A soccer game problem

- A user operates a game in the following sequence
  - Start the game
  - Select two teams
  - Add or remove players to/from a team
  - Pick a play ground
  - Start a match
- The system may have a number of PlayGrounds in it, some Teams etc.
  - Player, who plays soccer
  - Team, with various players in it
  - Ball, which is handled by various players.
  - PlayGround, where a match takes place
  - Referee, to control the game
- Also, you may need some logical objects in your game engine, like
  - Game, which defines a football game, including two teams, a ball, a referee, a playground etc
  - GameEngine to simulate a number of games at a time.
  - TeamStrategy, to decide a team's strategy while playing

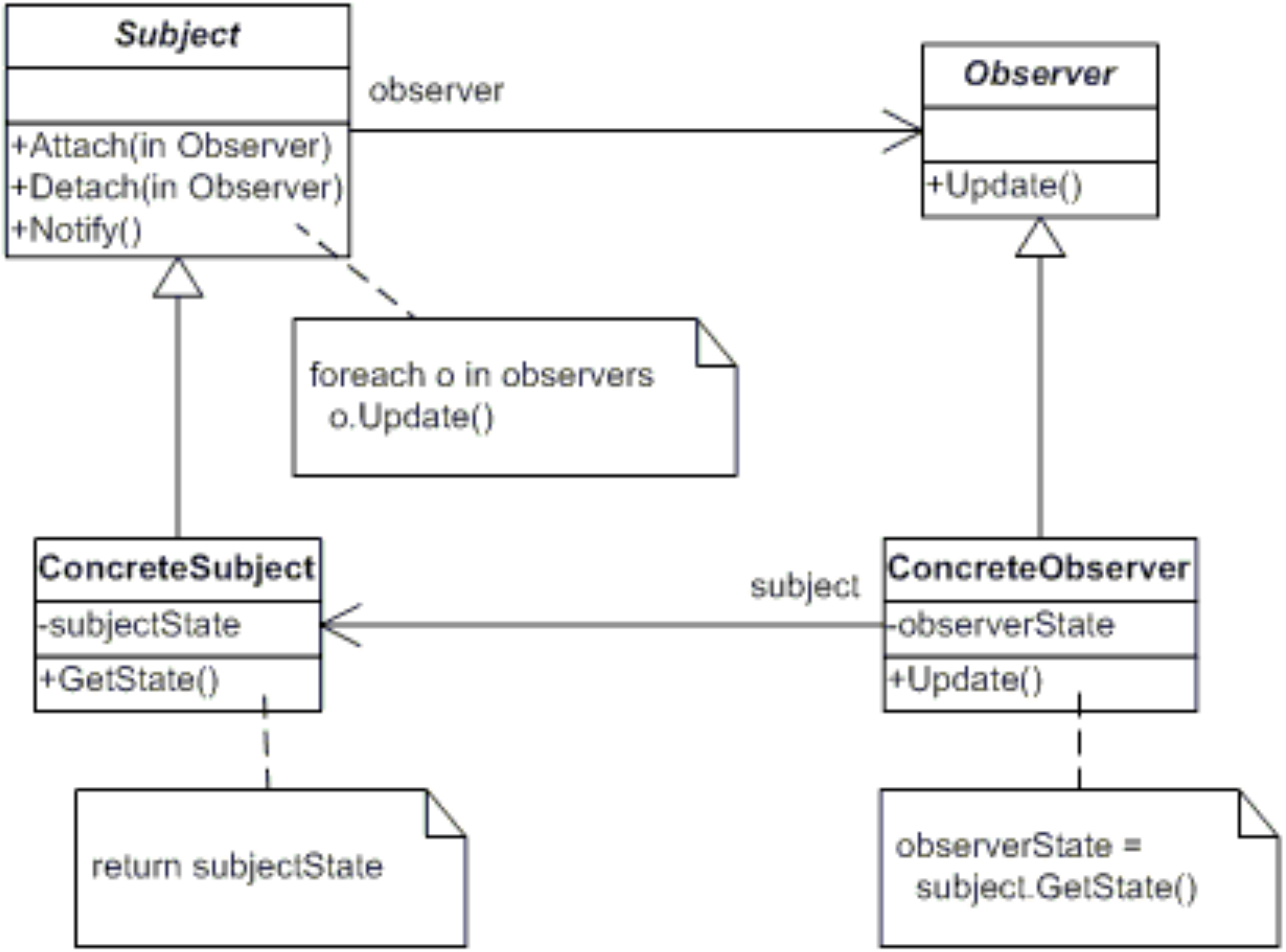
A soccer game engine



Design problem

- *"When the position of a ball changes, all the players and the referee should be notified"*
- *Problem Generalized: "When an object (in this case, the ball) changes, all its dependents (in this case, the players) are notified and updated automatically."*
- If we take the GOF patterns we find that we can apply the 'Observer' pattern to solve the problem
- **Observer Pattern**: Define a one-to-many dependency between objects, so that when one object changes its state, all its clients are notified and updated automatically

Observer pattern



Observer pattern

- This pattern manages a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically
- One or more objects (called *Observers* or *Listeners*) are registered (or register themselves) to observe an event that may be raised by the observed object (the *Subject*)
- The object that may raise an event generally maintains a collection of the *Observers*

Observer: subject participant

- **Subject**: abstract class providing an interface for attaching and detaching Observers
  - It holds a private list of *Observers*
  - It contains these methods:
    - **Attach**: it adds a new observer to the list of observers observing the subject
    - **Detach**: it removes an existing observer from the list of observers observing the subject
    - **Notify**: it notifies each observer by calling the notify() method in the Observer, when a change occurs

Observer: concretesubject participant

- **ConcreteSubject**: class providing the state of interest to observers
  - It sends a notification to all *Observers* by calling the *Notify* method in its superclass (i.e, in the *Subject* class).
  - It contains this method:
    - **GetState**: it returns the state of the *Subject*

Observer: observer participant

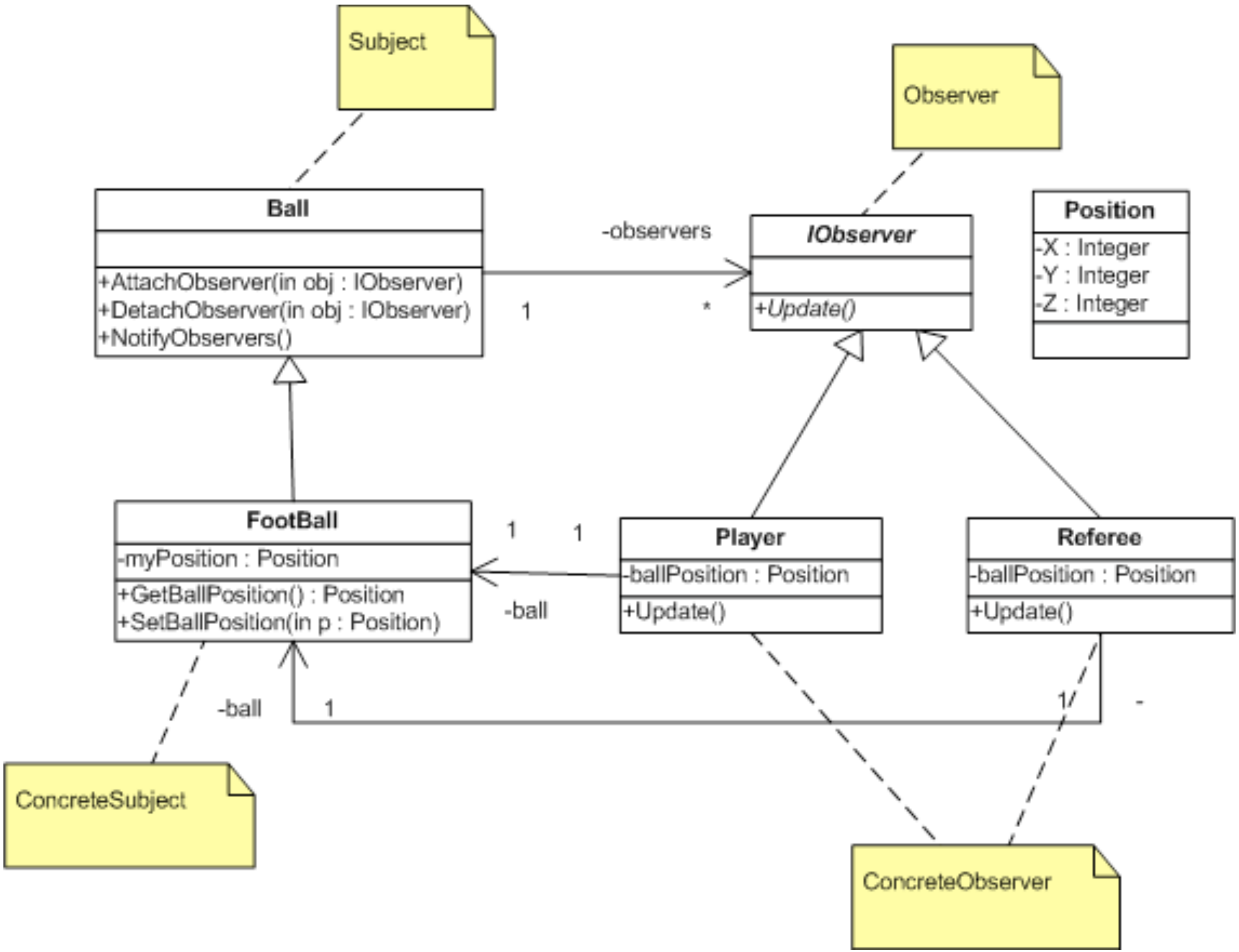
- **Observer**: class defining an updating interface for all *Observers* to receive update notification from the subject
  - It is used as an abstract class to implement concrete *Observers*
  - It contains this method:
    - **Notify**: abstract method, to be overridden by concrete *Observers*

Observer: concreteobserver participant

- **ConcreteObserver**: class maintaining a reference to *ConcreteSubject* to receive the state of the *Subject* when a notification event is received
  - It contains this method:
    - **Notify**: This is the overridden method in the concrete class
      - When it is called by the *Subject*, the *ConcreteObserver* calls the *GetState* method of the *Subject* to update the information it has about the *Subject*'s state
  - When the event is raised each *Observer* receives a callback
  - This may be either
    - a *Notify()* virtual method of *Observer*
    - a function pointer (more generally a function object or "functor") passed as an argument to the listener registration method
  - Each concrete observer implements the *Notify* method and as a consequence it defines its own behavior when the notification occurs

Observer usages

- The typical usages of this pattern:
  - Listening for an external event (such as a user action)
  - Listening for changes of the value of an object property
  - In a mailing list, every time an event happens (a new product, a gathering, etc.) a message is sent to the people subscribed to the list
- It is often associated with the Model-View-Controller (MVC) architectural pattern
  - In MVC, the *Observer* is used to create a loose coupling between the model and the view
  - A modification in the model triggers the notification of model observers which are actually the views


Subject
Observer
Ball
+AttachObserver(in obj : IObserver)
+DetachObserver(in obj : IObserver)
+NotifyObservers()
-observers
1
*
IObserver
+Update()
Position
-X : Integer
-Y : Integer
-Z : Integer
FootBall
-myPosition : Position
+GetBallPosition() : Position
+SetBallPosition(in p : Position)
1
1
-ball
Player
-ballPosition : Position
+Update()
Referee
-ballPosition : Position
+Update()
-ball
1
1
-
ConcreteSubject
ConcreteObserver

Patterns vs “Design”

- Patterns *are* design
  - Patterns transcend the “identify classes and associations” approach to design
  - Learn to recognize patterns in the *problem* space and translate to the solution
- Patterns can capture OO design principles within a specific domain
- Patterns provide structure to “design”

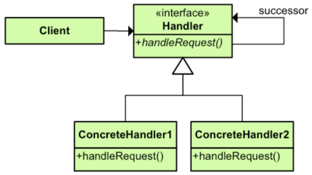
Patterns vs Frameworks

- An application framework is a standard reference structure for writing applications in a given operating system. Eg.: MFC for Windows or Cocoa for MacOS
- Patterns are at a level lower than frameworks
- Frameworks typically employ many patterns:
  - Factory
  - Strategy
  - Composite
  - Observer
- Patterns are the “plumbing” of a framework

Patterns vs Architecture

- Design Patterns (GoF) represent a lower level of system structure than a “software architecture”
- Patterns can be applied to architecture:
  - Mowbray and Malveau (CORBA)
  - Buschmann *et al*
  - Schmidt *et al*
- Architectural patterns focus on middleware. They are good at capturing:
  - Concurrency
  - Distribution
  - Synchronization

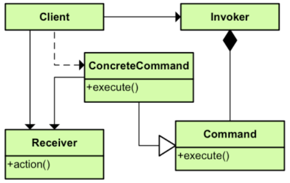
C	Abstract Factory
S	Adapter
S	Bridge
C	Builder
B	Chain of Responsibility
B	Command
S	Composite
S	Decorator
S	Facade
C	Factory Method
S	Flyweight
B	Interpreter
B	Iterator
B	Mediator
B	Memento
C	Prototype
S	Proxy
B	Observer
C	Singleton
B	State
B	Strategy
B	Template Method
B	Visitor



Chain of Responsibility

Type: Behavioral

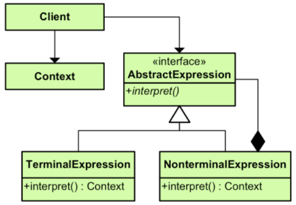
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Command

Type: Behavioral

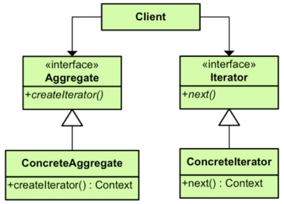
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Interpreter

Type: Behavioral

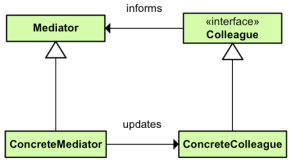
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Mediator

Type: Behavioral

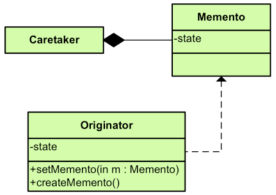
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

Memento

Type: Behavioral

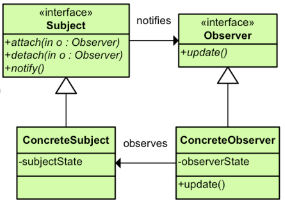
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Observer

Type: Behavioral

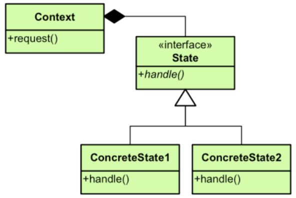
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



State

Type: Behavioral

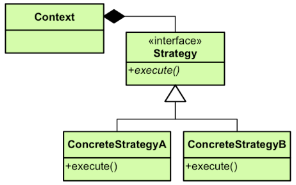
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Strategy

Type: Behavioral

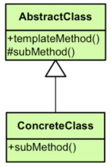
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Template Method

Type: Behavioral

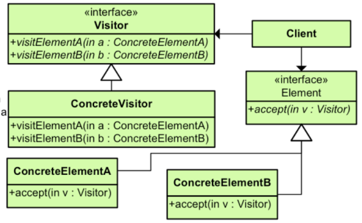
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Summary

- Design Patterns (GoF) provide a foundation for a deeper understanding of:
  - Modularity issues
  - Object-Oriented design
  - Software Architecture
- Understanding patterns can take some time
  - Re-reading them over time helps
  - Applying them in designs helps

Questions

- What is a design pattern?
- What is a pattern language?
- Which pattern exploits a recursive structure?
- When it is useful to use the pattern Singleton?
- When it is useful to use the pattern Observer?

References

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

- B. Bruegge and A. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd ed., Prentice Hall, 2004.

Useful sites

- A repository of patterns

  `hillside.net/patterns/`
- The original patterns for architecture by Alexander

  `www.enumerable.com/dev/index.htm`

  `www.patternlanguage.com`
- A course from UMBC University

  `www.research.umbc.edu/~tarr/dp/fall00/cs491.html`
- The Design Patterns Java Companion (online book)

  `www.patterndepot.com/put/8/JavaPatterns.htm`
- A Site dedicated to Design Patterns

  `home.earthlink.net/~huston2/dp/patterns.html`
- `www.research.ibm.com/designpatterns/publications.htm`

Useful sites

- Quiz online on GoF patterns
  `home.earthlink.net/~huston2/dp/patterns_quiz.html`
- A Pattern Language for HCI Design,
  `www.mit.edu/~jtidwell/common_ground_onefile.html`
- Design patterns for .NET:
  `www.dofactory.com/Patterns/Patterns.aspx`
- www.felix-colibri.com/papers/design_patterns/the_lexi_editor/the_lexi_editor.html
- Patterns for Concurrent, Parallel, and Distributed Systems
  `www.cs.wustl.edu/~schmidt/patterns-ace.html`
- J2EE Patterns Catalog
  `java.sun.com/blueprints/patterns/catalog.html`
- Articles on E++ pattern language
  `www.javaworld.com/jw-04-2001/jw-0420-eplus.html`

Tools

- `www.patternbox.com` Eclipse plug-in, pattern editor
- `dpatoolkit.sourceforge.net`
- Rational Software Modeler by IBM

Questions?