**INSTITUTE OF AERONAUTICAL ENGINEERING**
**(Autonomous)**
Dundigal, Hyderabad -500 043

**COMPUTER SCIENCE AND ENGINEERING**

**IV B. Tech I Semester**

# DESIGN PATTERNS

Prepared by: Mr. C.PRAVEEN KUMAR
Mr. R.M.NOORULLAH
Mr. M.RAKESH
Ms. J.HAREESHA

# DESIGN PATTERNS

IV B.Tech. I SEMESTER
UNIT 1
PPT SLIDES

**TEXT BOOKS:**

1. Design Pattern by Erich Gamma, Pearson Education
2. Pattern's in JAVA Vol-I BY Mark Grand, Wiley DreamTech
3. Pattern's in JAVA Vol-II BY Mark Grand, Wiley DreamTech
4. JAVA Enterprise Design Patterns Vol-III Mark Grand, Wiley Dream Tech
5. Head First Design Patterns By Eric Freeman-Oreilly-spd..
6. Design Patterns Explained By Alan Shalloway,Pearson Education

# What is a Design Pattern?

- Each pattern Describes a problem which occurs over and over again in our environment ,and then describes the core of the problem
- Novelists, playwrights and other writers rarely invent new stories.
- Often ideas are reused, such as the “Tragic Hero” from Hamlet or Macbeth.
- Designers reuse solutions also, preferably the “good” ones
  - Experience is what makes one an ‘expert’
- Problems are addressed without rediscovering solutions from scratch.
  - “My wheel is rounder”

## Design Patterns

- Design Patterns are the best solutions for there occurring problems in the application programming environment.
- Nearly a universal standard
- Responsible for design pattern analysis in other areas, including GUIs.
- Mainly used in Object Oriented programming

## Design Pattern Elements

1. Pattern Name

   Handle used to describe the design problem

   Increases vocabulary

   Eases design discussions

   Evaluation without implementation details

## Design Pattern Elements

2. Problem

   Describes when to apply a pattern

   May include conditions for the pattern to be applicable

   Symptoms of an inflexible design or limitation

## Design Pattern Elements

3. Solution

   Describes elements for the design

   Includes relationships, responsibilities, and collaborations

   Does not describe concrete designs or implementations

   A pattern is more of a template

## Design Pattern Elements

4. Consequences

   Results and Trade Offs

   Critical for design pattern evaluation

   Often space and time trade offs

   Language strengths and limitations

   (Broken into benefits and drawbacks for this discussion)

Design patterns can be subjective.

- One person's pattern may be another person's primitive building block.

The focus of the selected design patterns are:

- Object and class communication
- Customized to solve a general design problem
- Solution is context specific

## Design patterns in Smalltalk MVC

- The Model/View/Controller triad of classes is used to build user interfaces in Smalltalk-80

- MVC consists of three kinds of objects.
- M->>MODEL is the Application object.
- V->>View is the screen presentation.
- C->>Controller is the way the user interface reacts to user input.

MVC decouples   to increase flexibility and reuse.

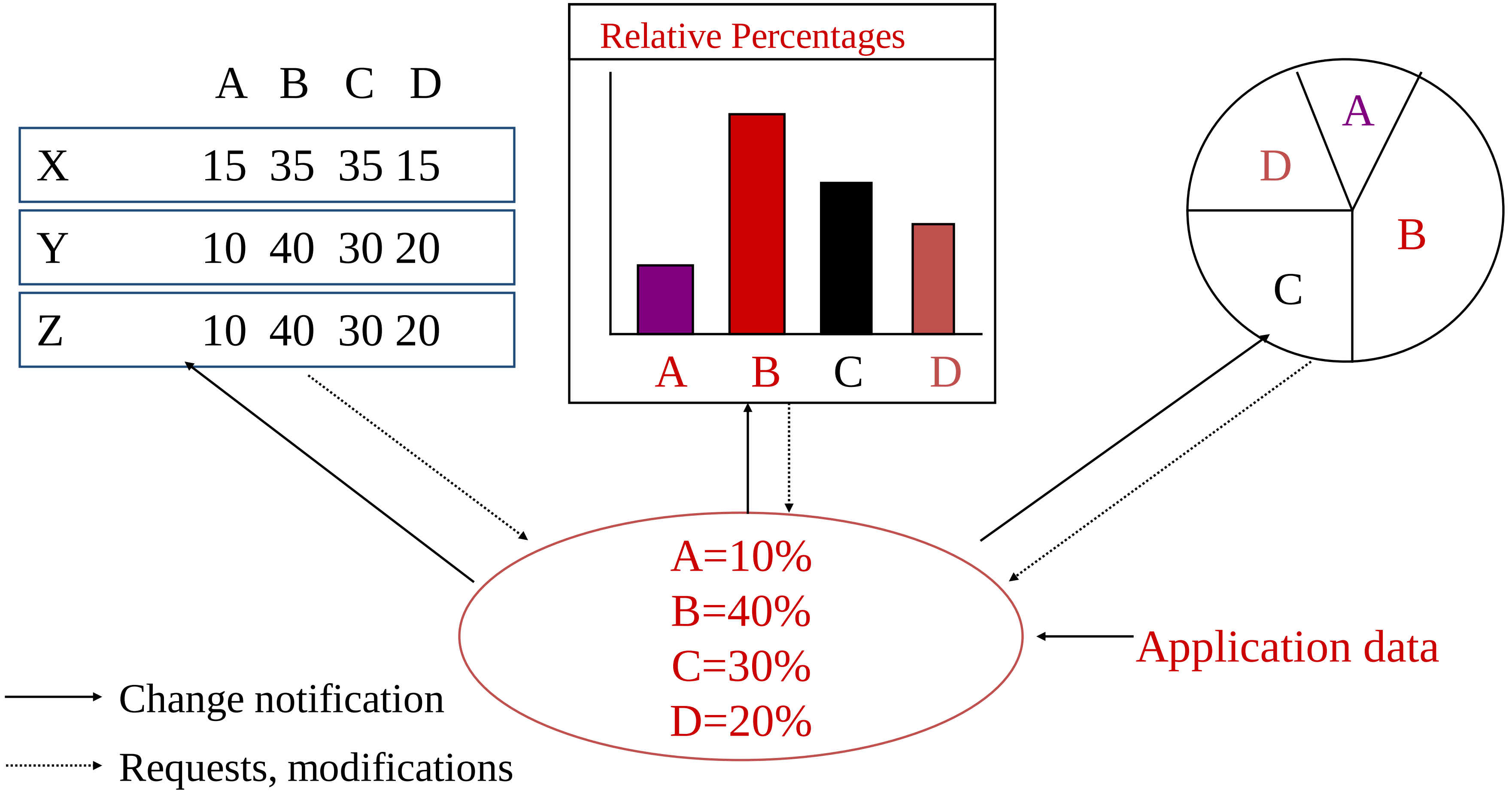
## Design patterns in Smalltalk MVC

- MVC decouples views and models by establishing a subscribe/notify protocol between them.
- A view must ensure that its appearance must reflects the state of the model.
- Whenever the model's data changes, the model notifies views that depends on it.
- You can also create new views for a model without Rewriting it.

# Design Patterns in Smalltalk MVC

- The below diagram shows a model and three views.
- The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways.
- The model communicates with it's values change, and the views communicate with the model to access these values.
- Feature of MVC is that views can be nested.
- Easy to maintain and enhancement.

# Design Patterns in Smalltalk MVC



## Describing Design Patterns

- Graphical notations ,while important and useful, aren’t sufficient.
- They capture the end product of the design process as relationships between classes and objects.
- By using a consistent format we describe the design pattern .
- Each pattern is divided into sections according to the following template.

# Describing Design Patterns

Pattern Name and Classification:

- it conveys the essence of the pattern succinctly good name is vital, because it will become part of design vocabulary.

Intent: What does the design pattern do?

- What is it’s rational and intend?
- What particular design issue or problem does it address?

Also Known As: Other well-known names for the pattern

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

- The scenario will help understand the more abstract description of the pattern that follows.

# Describing Design Patterns

Applicability:

- Applicability: What are the situations in which the design patterns can be applied?
- What are example of the poor designs that the pattern can address?
- How can recognize situations?
- Structure: Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique(OMT).
- Participants: The classes and/or objects participating in the design pattern and their responsibilities.

Structure:

- ➢ Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique(OMT).
- ➢ Participants: The classes and/or objects participating in the design pattern and their responsibilities.

## Describing Design Patterns

Collaborations:

- How the participants collaborate to carry out their responsibilities.

Consequences:

- How does the pattern support its objectives?
- What are the trade-offs and result of using the pattern ?
- What aspect of the system structure does it let vary independently?

Implementation:

- What Pitfalls, Hints, Techniques should be aware of when implementing the pattern ?
- Are there language-specific issues?

## Describing Design Patterns

Sample Code:

- ➢ Code fragments that illustrate how might implement the pattern in C++ or Smalltalk.

Known Uses:

- ➢ Examples of the pattern found in real systems.

Related Patterns:

- ➢ What design patterns are closely related to this one? What are the imp differences?
- ➢ With Which other patterns should this one be used?

## The Catalog of Design Pattern

- Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Adapter: Convert the interface of a class into another interface clients expect.
- Bridge: Decouple an abstraction from its implementation so that two can vary independently.

# The Catalog of Design Pattern

- Builder: Separates the construction of the complex object from its representation so that the same constriction process can create different representations.

- Chain of Responsibility: Avoid coupling the sender of a request to it's 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 objects handles it.

## The Catalog of Design Pattern

- Command: Encapsulate a request as an object, thereby letting parameterize clients with different request, queue or log requests, and support undoable operations.

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

## The Catalog of Design Pattern

- Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing  for extending functionality.

- Façade: Provide a unified interface to a set of interfaces  in a subsystem's Facade defines a higher-level interface that makes the subsystem easier to use.

## The Catalog of Design Pattern

- Factory Method: Defines an interface for creating an object ,but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.

- Interpreter: Given a language, defining a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.

## The Catalog of Design Pattern

- Iterator: Provide a way to access the element of an aggregate object sequentially without exposing its underlying representation.

- Mediator: Define an object that encapsulate how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and let's you very their interaction independently.

# The Catalog of Design Pattern

- Memento: Without violating encapsulation, capture and externalize an object’s internal state so that object can be restored to this state later.

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

## The Catalog of Design Pattern

- Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Proxy: Provide a surrogate or placeholder for another object to control access to it.
- Singleton: Ensure a class has only one instance, and provide a point of access to it.
- State: Allow an object to alter its behavior when its internal state changes. the object will appear to change its class.

# The Catalog of Design Pattern

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

- Template Method: Define the Skelton of an operation, deferring some steps to subclasses. Template method subclasses redefine certain steps of an algorithm without changing the algorithms structure.

## The Catalog of Design Pattern

- Visitor: 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.

# Organizing the Catalog

- Two criteria
  - ❖**Purpose**: what a pattern does

    *Creational*: The process of object creation

    *Structural*: The composition of classes or objects

    *Behavioral*: Characterize the ways in which classes or objects interact and distribute responsibility

  - ❖**Scope**: whether the pattern applies primarily to *classes* or to *objects*

## Organizing the Catalog

| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

## How Design Patterns Solve Design Problems

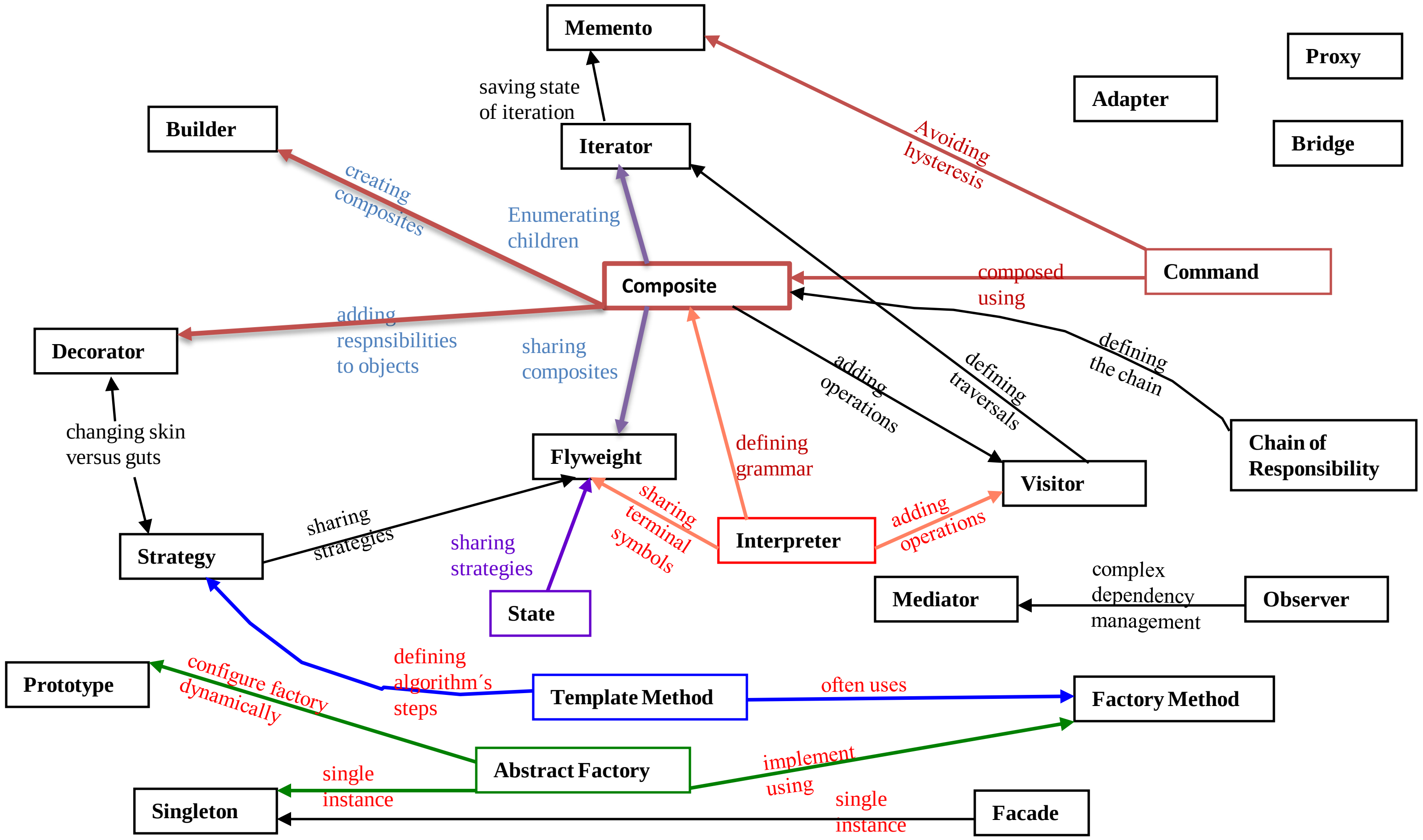
- Finding Appropriate Objects
  - Decomposing a system into objects is the hard part
  - Object Oriented Designs often end up with classes with no counterparts in real world (low-level classes like arrays)
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows
  - Design patterns identify less-obvious abstractions

## How Design Patterns Solve Design Problems

- Determining Object Granularity
  - Objects can vary tremendously in size and number
  - **Facade pattern** describes how to represent subsystems as objects
  - **Flyweight pattern** describes how to support huge numbers of objects

# Design Pattern relationship

- Mapping



## Specifying Object Interfaces

- Interface:
  - Set of all signatures defined by an object’s operations
  - Any request matching a signature in the objects interface may be sent to the object
  - Interfaces may contain other interfaces as subsets
- Type:
  - Denotes a particular interfaces
  - An object may have many types
  - Widely different object may share a type
  - Objects of the same type need only share parts of their interfaces
  - A **subtype** contains the interface of its **super type**
- Dynamic binding, polymorphism

## Specifying Object Interfaces

- An object’s implementation is defined by its ***class***
- The class specifies the object’s internal data and defines the operations the object can perform
- Objects is created by ***instantiating*** a class
  - an object = an *instance* of a class
- ***Class inheritance***
  - parent class and subclass

## Specifying Object Implementations (cont.)

- ***Abstract class*** versus ***concrete class***
  - abstract operations
- ***Override*** an operation
- Class versus type
  - An object's *class* defines how the object is implemented
  - An object's *type* only refers to its interface
  - An object can have many types, and objects of different classes can have the same type

## Specifying Object Implementations (cont.)

- Class versus Interface Inheritance
  - class inheritance defines an object's implementation in terms of another object's implementation (code and representation sharing)
  - interface inheritance (or subtyping) describes when an object can be used in place of another
- Many of the design patterns depend on this distinction

## Specifying Object Implementations (cont.)

- **Programming to an Interface, not an Implementation**
- Benefits
  - clients remain unaware of the specific types of objects they use
  - clients remain unaware of the classes that implement these objects

## Program to an interface, not an Implementation

- Manipulate objects solely in terms of interfaces defined by abstract classes!

- Benefits:
  1. Clients remain unaware of the specific types of objects they use.
  2. Clients remain unaware of the classes that implement the objects.
     Clients only know about abstract class(es) defining the interfaces

  - Do not declare variables to be instances of particular concrete classes
  - Use creational patterns to create actual objects.

## Favor object composition over class inheritance

- **White-box** reuse:
  - Reuse by sub classing (class inheritance)
  - Internals of parent classes are often visible to subclasses
  - works statically, compile-time approach
  - Inheritance breaks encapsulation
- **Black-box** reuse:
  - Reuse by object composition
  - Requires objects to have well-defined interfaces
  - No internal details of objects are visible

## Putting Reuse Mechanisms to Work

- ***Inheritance*** versus ***Composition***
- ***Delegation***
- Inheritance versus ***Parameterized Types***

## Inheritance versus Composition

- Two most common techniques for reuse
  - class inheritance
    - white-box reuse
  - ***object composition***
    - black-box reuse
- Class inheritance
  - advantages
    - static, straightforward to use
    - make the implementations being reuse more easily

## Inheritance versus Composition (cont.)

- Class inheritance (cont.)
  - disadvantages
    - the implementations inherited can’t be changed at run time
    - parent classes often define at least part of their subclasses’ physical representation
      - breaks encapsulation
    - implementation dependencies can cause problems when you’re trying to reuse a subclass

## Inheritance versus Composition (cont.)

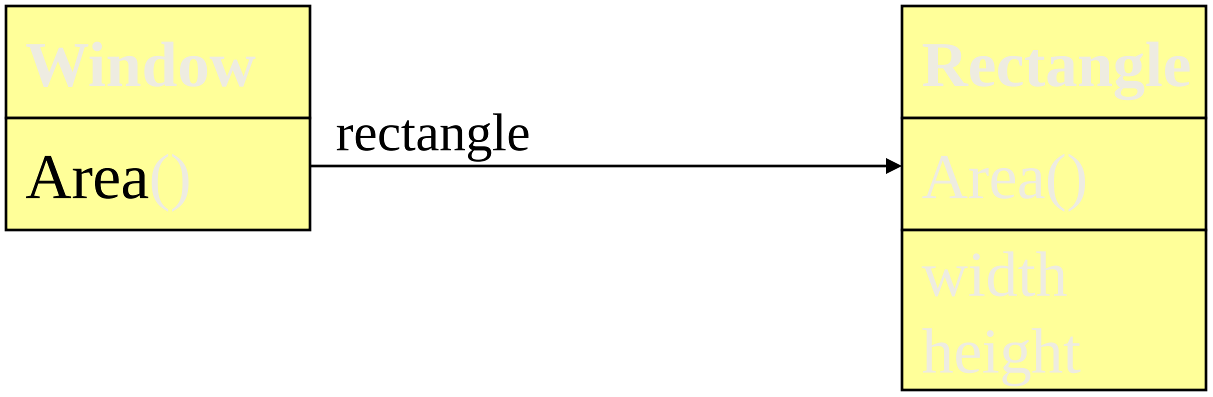
- Object composition
  - dynamic at run time
  - composition requires objects to respect each others ‘ interfaces
    - but does not break encapsulation
  - any object can be replaced at run time
  - Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task

## Inheritance versus Composition (cont.)

- Object composition (cont.)
  - class and class hierarchies will remain small
  - but will have more objects
- ***Favor object composition over class inheritance***
- Inheritance and object composition should work together

## Delegation

- Two objects are involved in handling a request: a receiving object delegates operations to its ***delegate***



## Delegation (cont.)

- Makes it easy to compose behaviors at run-time and to change the way they're composed
- Disadvantage:dynamic, highly parameterized software is harder to understand than more static software
- Delegation is a good design choice only when it simplifies more than it complicates
- Delegation is an extreme example of object composition

## Inheritance versus Parameterized Types

- Let you define a type without specifying all the other types it uses, the unspecified types are supplied as parameters at the point of use
- Parameterized types, generics, or templates
- Parameterized types give us a third way to compose behavior in object-oriented systems

## Inheritance versus Parameterized Types (cont.)

- Three ways to compose
  - ***object composition*** lets you change the behavior being composed at run-time, but it requires indirection and can be less efficient
  - ***inheritance*** lets you provide default implementations for operations and lets subclasses override them
  - ***parameterized types*** let you change the types that a class can use

## Relating Run-Time and Compile-Time Structures

- An object-oriented program’s ***run-time structure*** often bears little resemblance to its ***code structure***
- The code structure is frozen at compile-time
- A program’s run-time structure consists of rapidly changing networks of communicating objects
- ***aggregation*** versus ***acquaintance*** (***association***)
  - *part-of* versus *knows of*

## Relating Run-Time and Compile-Time Structures

- The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms
- The system’s run-time structure must be imposed more by the designer than the language

## Designing for Change

- A design that doesn’t take change into account risks major redesign in the future
- Design patterns help you avoid this by ensuring that a system can change in specific ways
  - each design pattern lets some aspect of system structure vary independently of other aspects

## Common Causes of Redesign

- Creating an object by specifying a class explicitly
- Dependence on specific operations
- Dependence on hardware and software platform
- Dependence on object representations or implementations
- Algorithmic dependencies

## Common Causes of Redesign (cont.)

- Tight coupling
- Extending functionality by subclassing
- Inability to alter classes conveniently

## Design for Change (cont.)

- Design patterns in application programs
  – Design patterns that reduce dependencies can increase internal reuse
  – Design patterns also make an application more maintainable when they’re used to limit platform dependencies and to layer a system

## Design for Change (cont.)

- Design patterns in toolkits
  - A ***toolkit*** is a set of related and reusable classes designed to provide useful, general-purpose functionality
  - Toolkits emphasize code reuse. They are the object-oriented equivalent of subroutine libraries
  - Toolkit design is arguably harder than application design

## Design for Change (cont.)

- Design patterns in framework
  - A ***framework*** is a set of cooperating classes that make up a reusable design for a specific class of software
  - You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework
  - The framework dictates the *architecture* of your application

## Design for Change (cont.)

- Design patterns in framework (cont.)
  - Frameworks emphasize *design reuse* over code reuse
  - When you use a *toolkit*, you write the main body of the application and call the code you want to reuse. When you use a *framework*, you reuse the main body and write the code ***it*** calls.
  - Advantages: build an application faster, easier to maintain, and more consistent to their users

## Design for Change (cont.)

- Design patterns in framework (cont.)
  - Mature frameworks usually incorporate several design patterns
  - People who know the patterns gain insight into the framework faster
  - differences between framework and design pattern
    - design patterns are more abstract than frameworks
    - design patterns are smaller architectural elements than frameworks
    - design patterns are less specialized than frameworks

## How To Select a Design Pattern

- Consider how design patterns solve design Problems.
- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a Cause of redesign.
- Consider what should be variable in your design.

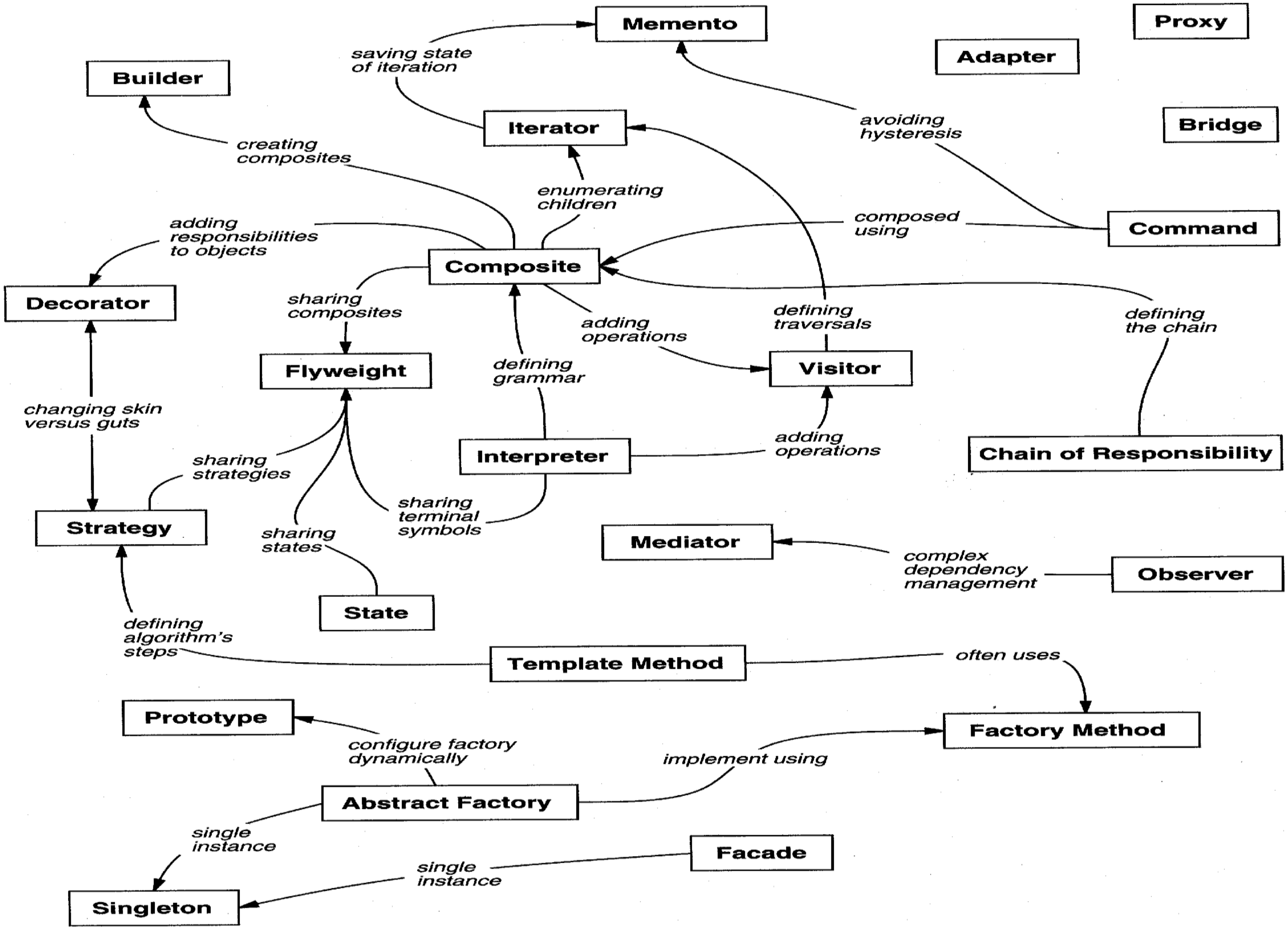
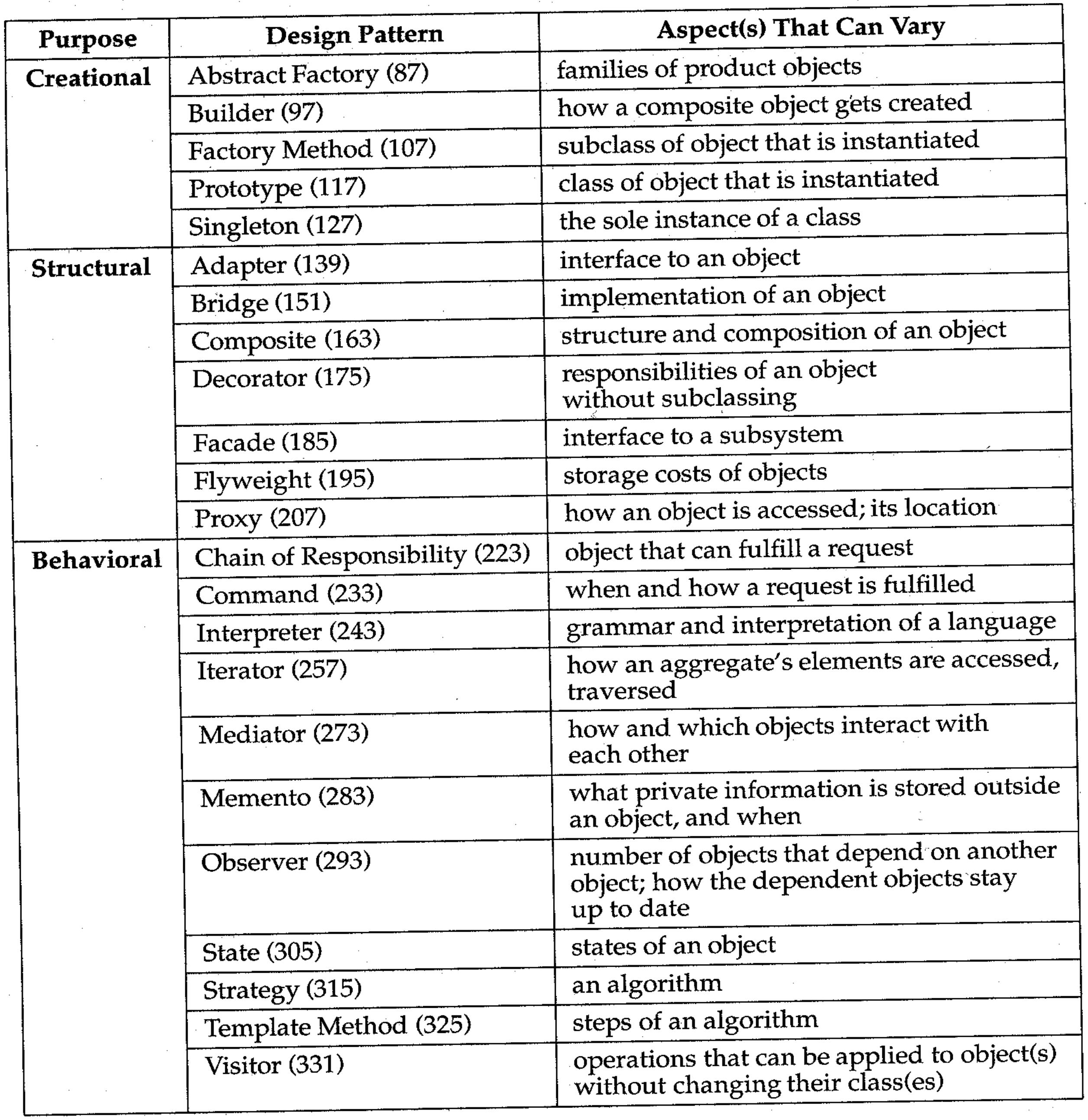


Figure 1.1: Design pattern relationships

## How To Use a Design Pattern

- Read the pattern once through for an overview.
- Go Back and study the Structure, Participants ,and Collaborations sections.
- Look At the Sample Code section to see a concrete

  Example of the pattern in code.
- Choose names for pattern participants that are meaningful in the application context.
- Define the classes.
- Define App’n-specific names for operations in the Pattern
- Implement the operations to carry out responsibilities in the pattern.



| 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 |
| **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) |

Table 1.2: Design aspects that design patterns let you vary

Design Aspects that design patterns let you vary

## Online resources

- Pattern FAQ
  - http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html
- Basic patterns
  - http://exciton.cs.oberlin.edu/javaresources/DesignPatterns/default.htm
- Patterns home page
  - http://hillside.net/patterns/

# Unit -II

## A Case Study: Designing a Document Editor

| S.NO. | TOPIC | PPT Slides | |
|---|---|---|---|
| 1 | Document structure | L1 | 4 – 13 |
| 2 | Formatting | L2 | 14 – 20 |
| 3 | Embellishment | L3 | 21 – 25 |
| 4 | Multiple look & feels | L4 | 26 – 30 |
| 5 | Multiple window systems | L5 | 31 – 35 |
| 6 | User operations L6 | 36 – 46 | |
| 7 | Spelling checking & hyphenation | L7 | 47 – 60 |
| 8 | Concluding Remarks | L8 | 61 – 61 |
| 9 | Pattern References | L8 | 62 – 67 |

# Design Problems:

- seven problems in Lexis's design:

Document Structure:

- ✓ The choice of internal representation for the document affects nearly every aspect of Lexis's design. All editing , formatting, displaying, and textual analysis will require traversing the representation.

Formatting:

- ✓ How does Lexi actually arrange text and graphics into lines and columns?
- ✓ What objects are responsible for carrying out different formatting policies?
- ✓ How do these policies interact with the document’s internal representation?

Embellishing the user interface:

Lexis user interface include scroll bar, borders and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexis user interface evolves.

Supporting multiple look-and-feel standards:
Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

Supporting multiple window systems:
Different look-and-fell standards are usually implemented on different window system. Lexi's design should be independent of the window system as possible.

User Operations:
User control Lexi through various interfaces, including buttons and pull-down menus. The functionality beyond these interfaces is scattered throughout the objects in the application.

Spelling checking and hyphenation.:
How does Lexi support analytical operations checking for misspelled words and determining hyphenation  points? How can we minimize the number of classes we have to modify to add a new analytical operation?

## Part II: Application: Document Editor (Lexi)

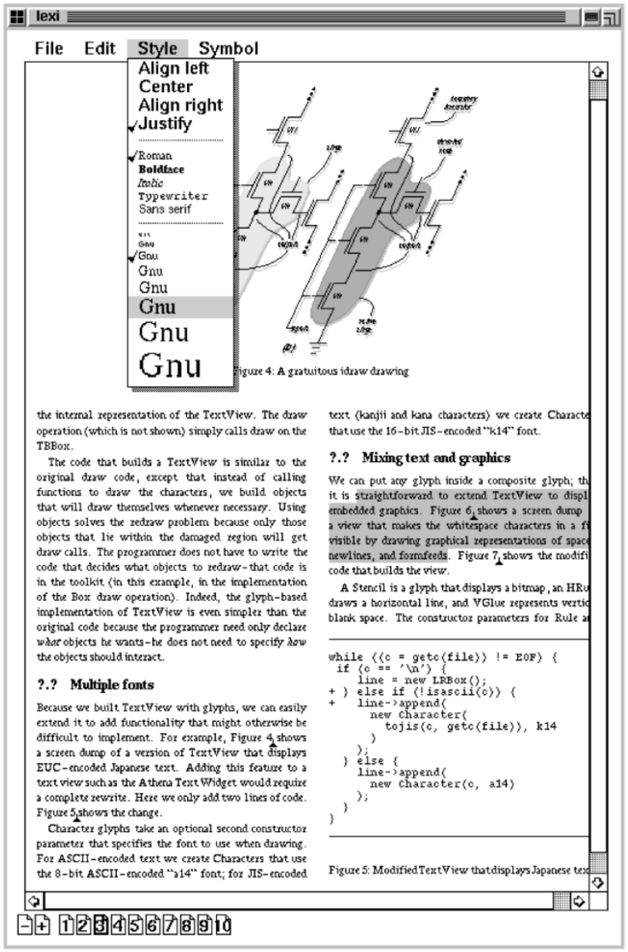


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw – that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare *what* objects he wants – he does not need to specify *how* the objects should interact.

**?.? Multiple fonts**

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded text (kanjii and kana characters) we create Characte that use the 16-bit JIS-encoded "k14" font.

**?.? Mixing text and graphics**

We can put any glyph inside a composite glyph; th it is straightforward to extend TextView to displ embedded graphics. Figure 6 shows a screen dump a view that makes the whitespace characters in a f visible by drawing graphical representations of space newlines, and formfeeds. Figure 7 shows the modifi code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu draws a horizontal line, and VGlue represents vertic blank space. The constructor parameters for Rule a

```
while ((c = getc(file)) != EOF) {
 if (c == '\n') {
     line = new LRBox();
+ } else if (!isascii(c)) {
+     line->append(
       new Character(
         tojis(c, getc(file)), k14
       )
     );
  } else {
     line->append(
       new Character(c, a14)
     );
  }
}
```

Figure 5: Modified TextView that displays Japanese tex

### 7 Design Problems

1. Document structure
2. Formatting
3. Embellishment
4. Multiple look & feels
5. Multiple window systems
6. User operations
7. Spelling checking & hyphenation

## Document Structure

Goals:

- present document's visual aspects
- drawing, hit detection, alignment
- support physical structure (e.g., lines, columns)

Constraints/forces:

- treat text & graphics uniformly
- no distinction between one & many

## Document Structure

- The internal representation for a document
- The internal representation should support
  - maintaining the document's physical structure
  - generating and presenting the document visually
  - mapping positions on the display to elements in the internal representations

## Document Structure (cont.)

- Some constraints
  - we should treat text and graphics uniformly
  - our implementation shouldn’t have to distinguish between single elements and groups of elements in the internal representation
- Recursive Composition
  - a common way to represent hierarchically structured information

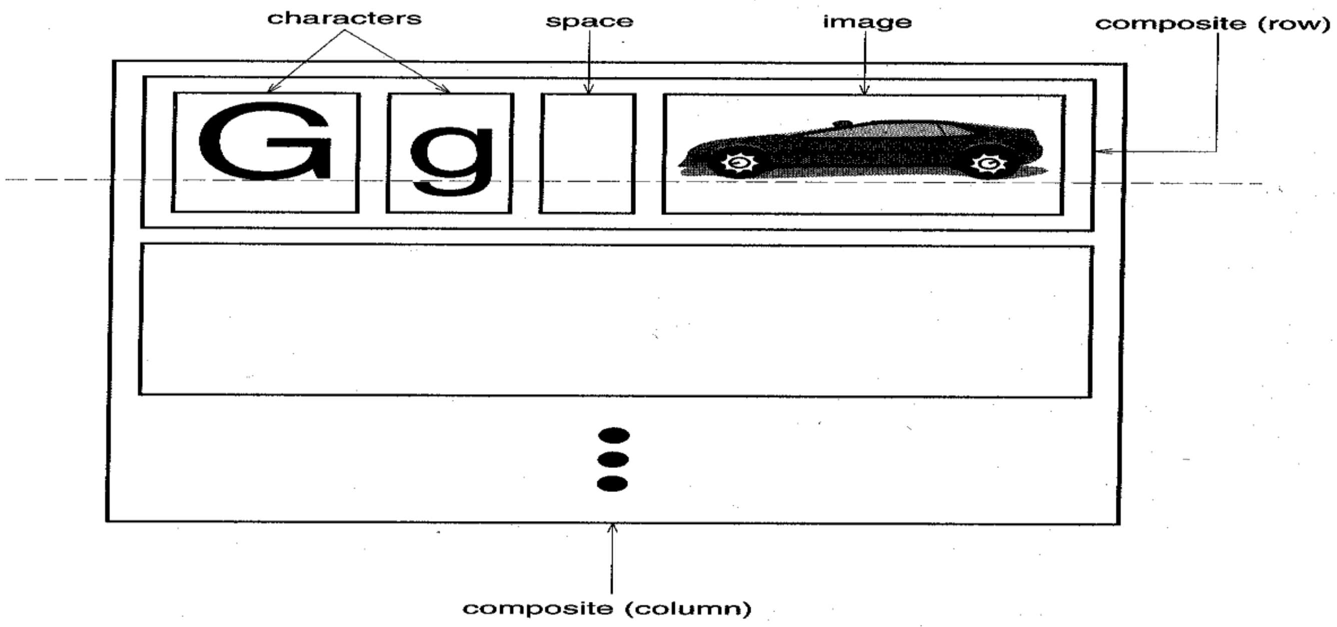


Figure 2.2: Recursive composition of text and graphics

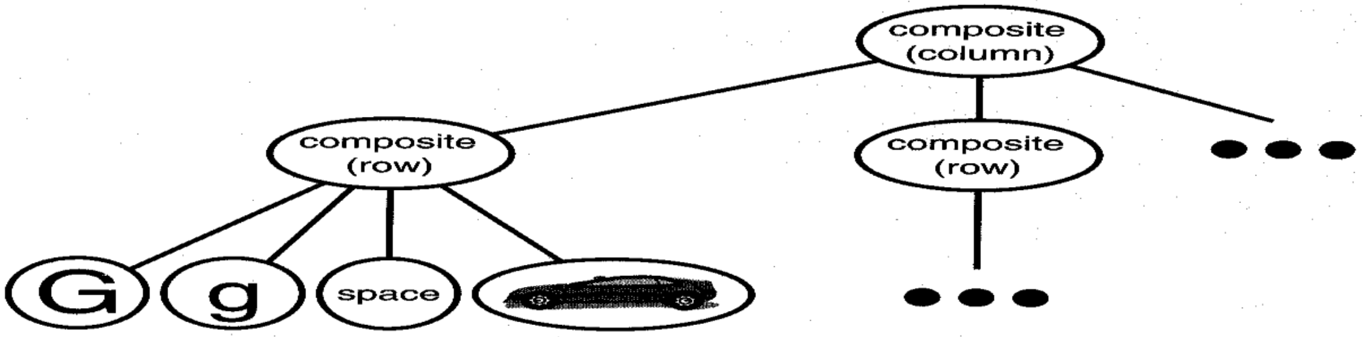


Figure 2.3: Object structure for recursive composition of text and graphics

## Document Structure (cont.)

- Glyphs
  - – an abstract class for all objects that can appear in a document structure
  - – three basic responsibilities, they know
    - how to draw themselves, what space they occupy, and their children and parent
- Composite Pattern
  - – captures the essence of recursive composition in object-oriented terms

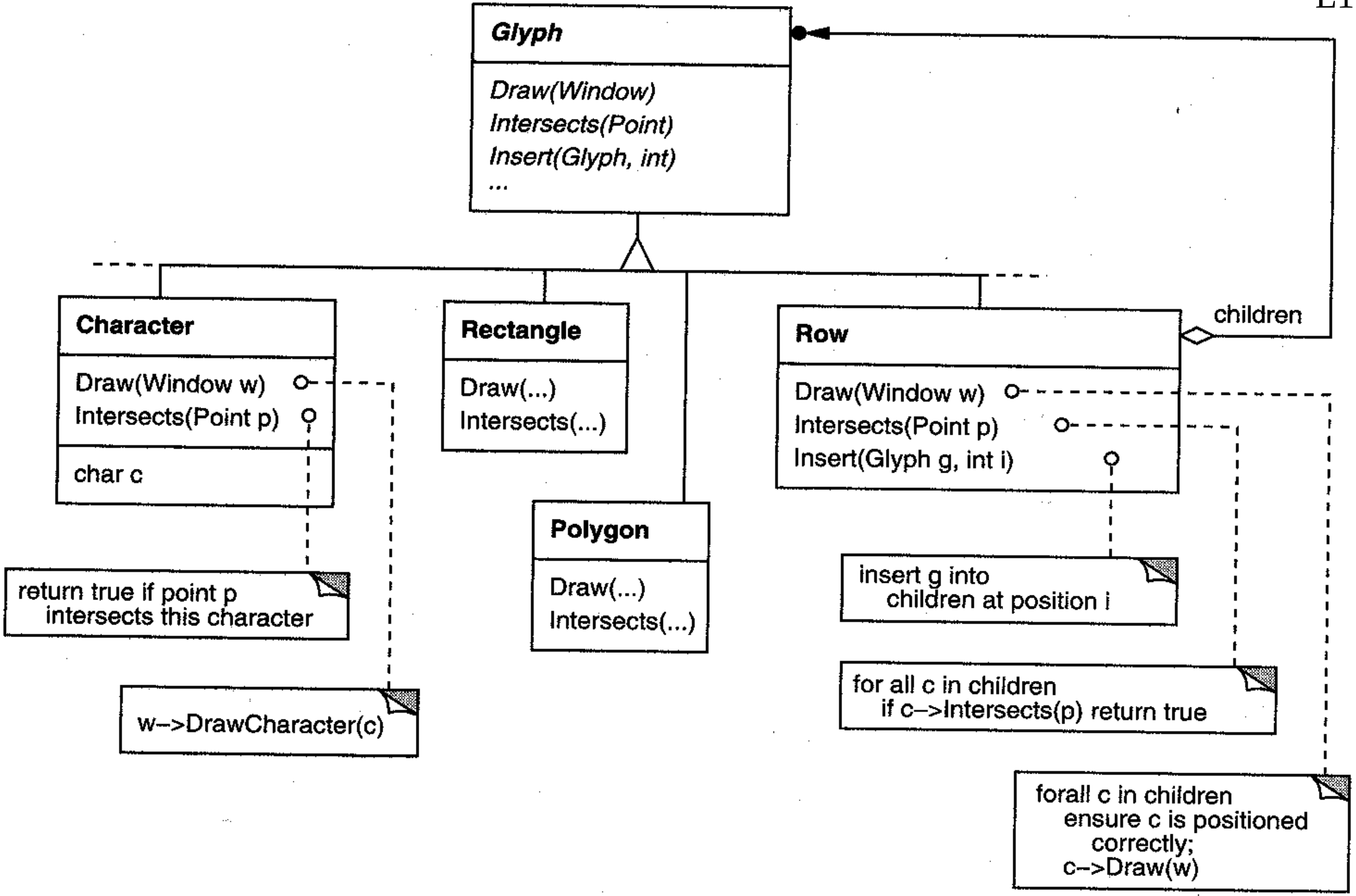


Figure 2.4: Partial Glyph class hierarchy

| Responsibility | Operations |
|---|---|
| appearance | `virtual void Draw(Window*)`<br>`virtual void Bounds(Rect&)` |
| hit detection | `virtual bool Intersects(const Point&)` |
| structure | `virtual void Insert(Glyph*, int)`<br>`virtual void Remove(Glyph*)`<br>`virtual Glyph* Child(int)`<br>`virtual Glyph* Parent()` |

Table 2.1: Basic glyph interface

## Formatting

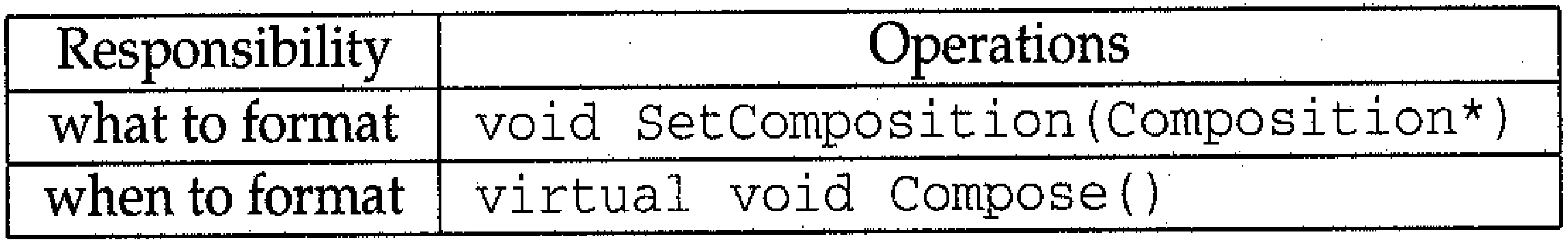
- A structure that corresponds to a properly formatted document
- Representation and formatting are distinct
  - the ability to capture the document's physical structure doesn't tell us how to arrive at a particular structure
- here, we'll restrict "formatting" to mean breaking a collection of glyphs in to lines

## Formatting (cont.)

- Encapsulating the formatting algorithm
  - keep formatting algorithms completely independent of the document structure
  - make it is easy to change the formatting algorithm
  - We'll define a separate class hierarchy for objects that encapsulate formatting algorithms

## Formatting (cont.)

- Compositor and Composition
  - We'll define a ***Compositor*** class for objects that can encapsulate a formatting algorithm
  - The glyphs Compositor formats are the children of a special Glyph subclass called ***Composition***
  - When the composition needs formatting, it calls its compositor's *Compose* operation
  - Each Compositor subclass can implement a different line breaking algorithm



| Responsibility | Operations |
|---|---|
| what to format | `void SetComposition(Composition*)` |
| when to format | `virtual void Compose()` |

Table 2.2: Basic compositor interface

## Formatting (cont.)

- Compositor and Composition (cont.)
  - The Compositor-Composition class split ensures a strong *separation* between code that supports the document’s physical structure and the code for different formatting algorithms
- Strategy pattern
  - intent: encapsulating an algorithm in an object
  - Compositors are strategies. A composition is the context for a compositor strategy

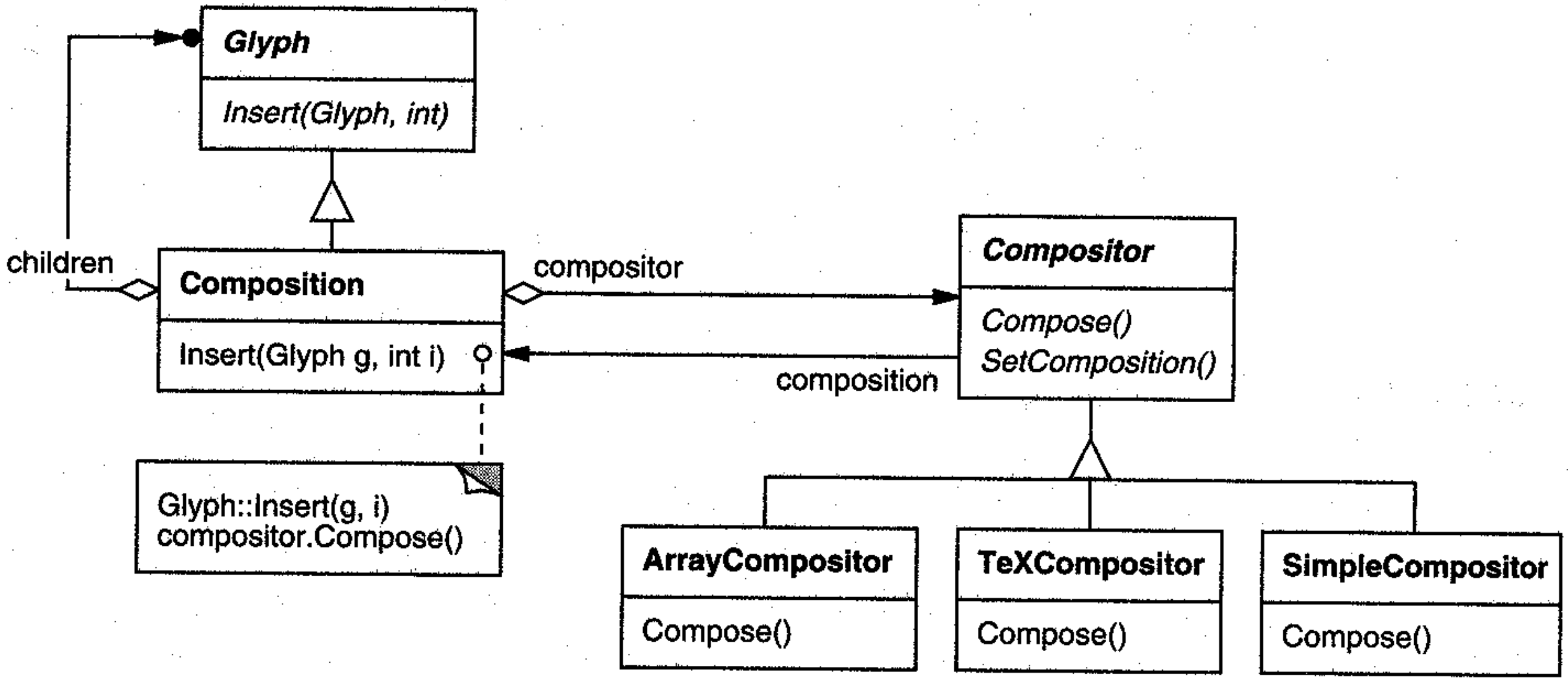


Figure 2.5: Composition and Compositor class relationships

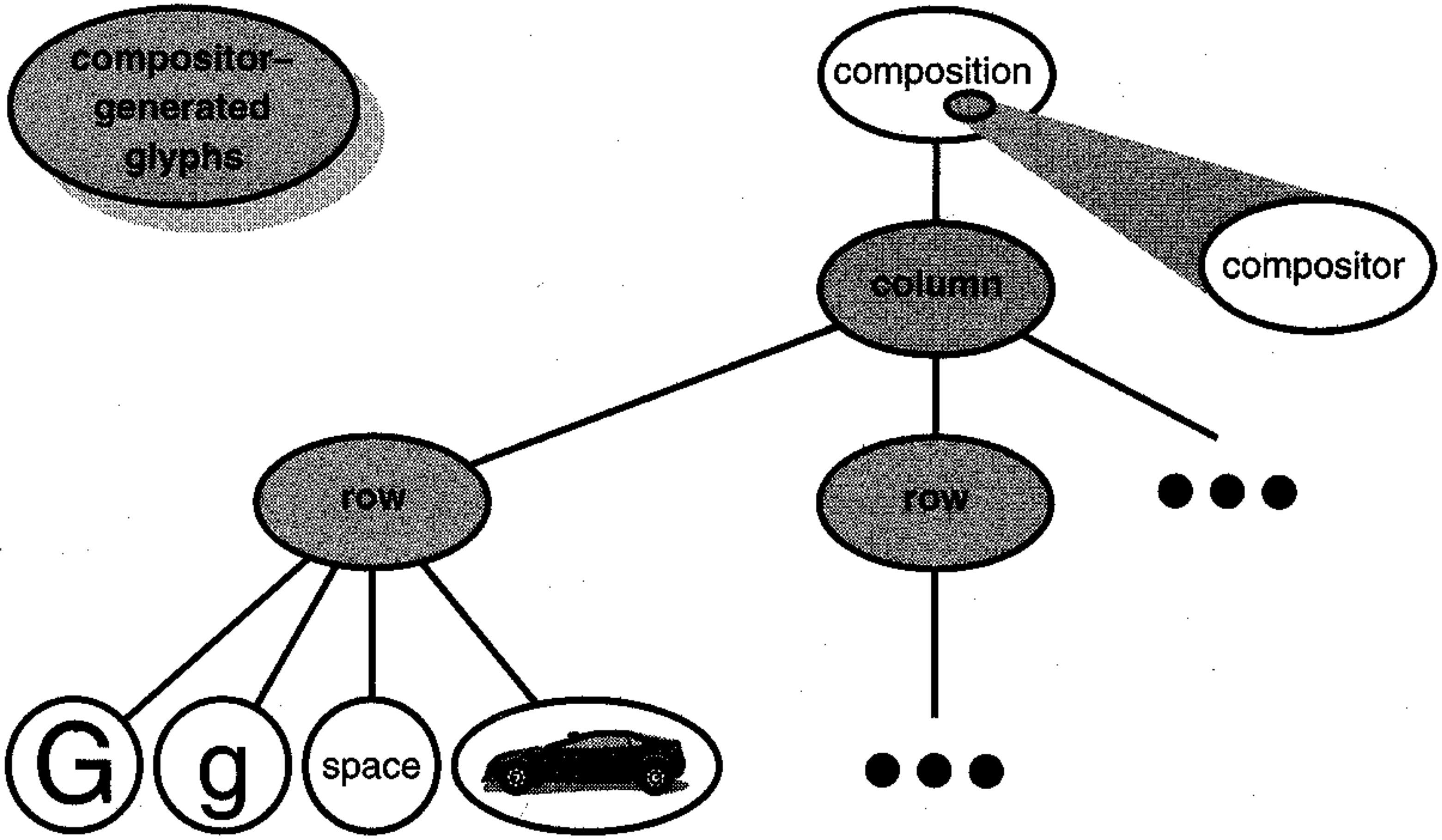


Figure 2.6: Object structure reflecting compositor-directed linebreaking

## Embellishing the User Interface

- Considering adds a ***border*** around the text editing area and ***scrollbars*** that let the user view the different parts of the page here
- Transparent Enclosure
  - *inheritance-based* approach will result in some problems
    - Composition, ScollableComposition, BorderedScrollableComposition, …
  - *object composition* offers a potentially more workable and flexible extension mechanism

## Embellishing the User Interface (cont.)

- Transparent enclosure (cont.)
  - object composition (cont.)
    - Border and Scroller should be a subclass of Glyph
  - two notions
    - single-child (single-component) composition
    - compatible interfaces

## Embellishing the User Interface (cont.)

- Monoglyph
  - We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs
  - the class, Monoglyph
- Decorator Pattern
  - captures class and object relationships that support embellishment by transparent enclosure

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

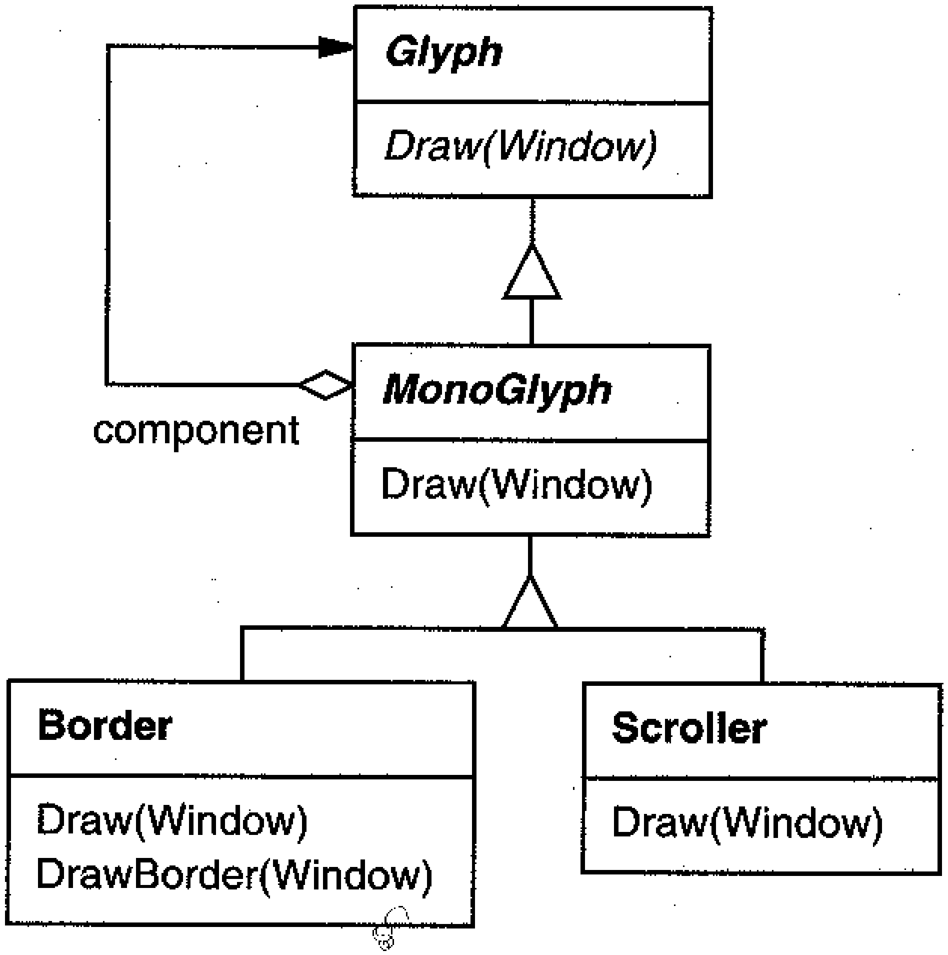


Figure 2.7: MonoGlyph class relationships

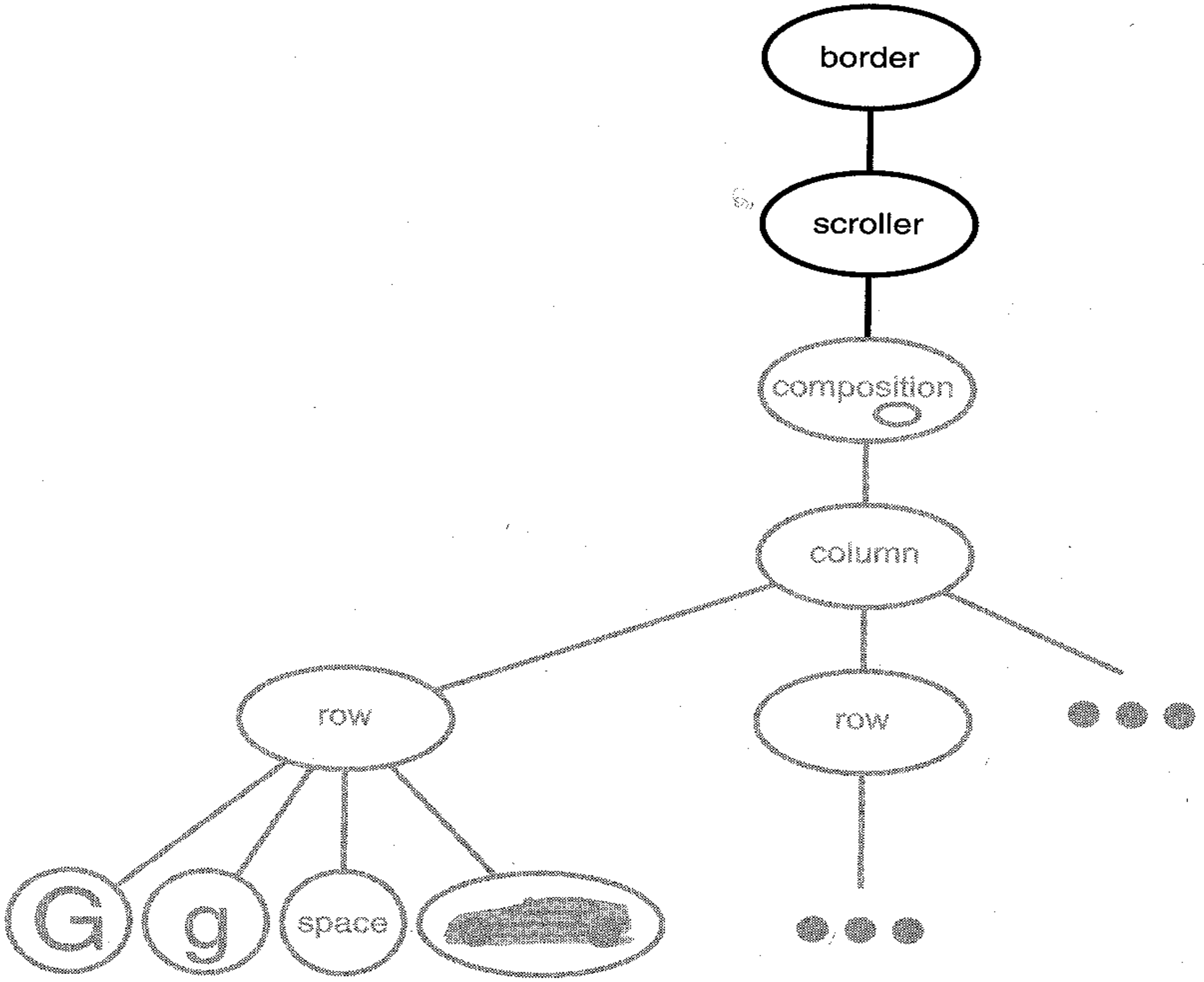


Figure 2.8: Embellished object structure

## Supporting Multiple Look-and-Feel Standards

- Design to support the look-and-feel changing at run-time
- Abstracting Object Creation
  - widgets
  - two sets of widget glyph classes for this purpose
    - a set of abstract glyph subclasses for each category of widget glyph (e.g., ScrollBar)
    - a set of concrete subclasses for each abstract subclass that implement different look-and-feel standards (e.g., MotifScrollBar and PMScrollBar)

## Supporting Multiple Look-and-Feel Standards (cont.)

- Abstracting Object Creation (cont.)
  - Lexi needs a way to determine the look-and-feel standard being targeted
  - We must avoid making explicit constructor calls
  - We must also be able to replace an entire widget set easily
  - We can achieve both by *abstracting the process of object creation*

## Supporting Multiple Look-and-Feel Standards (cont.)

- Factories and Product Classes
  - ***Factories*** create ***product*** objects
  - The example
- Abstract Factory Pattern
  - capture how to create families of related product objects without instantiating classes *directly*

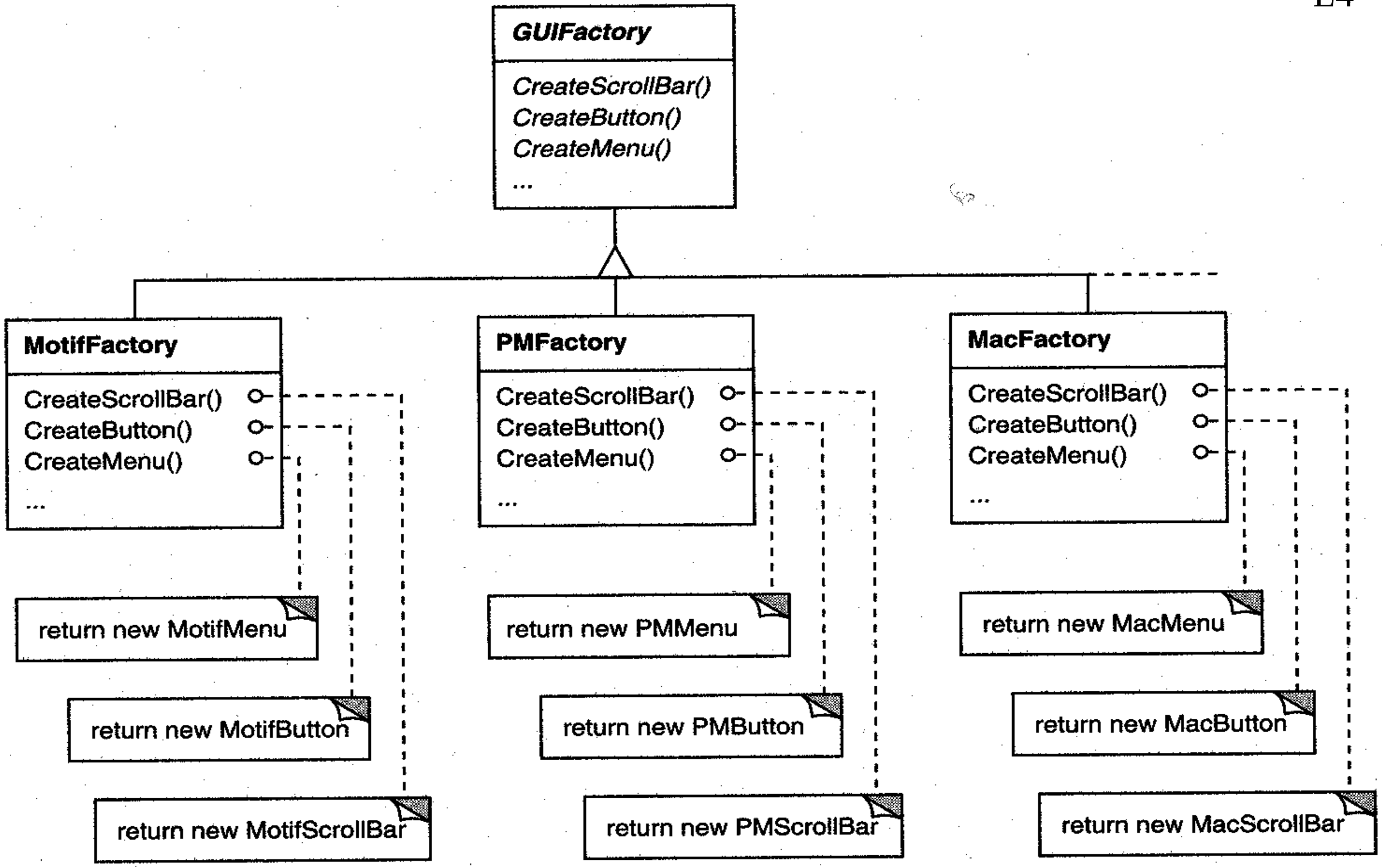


Figure 2.9: GUIFactory class hierarchy

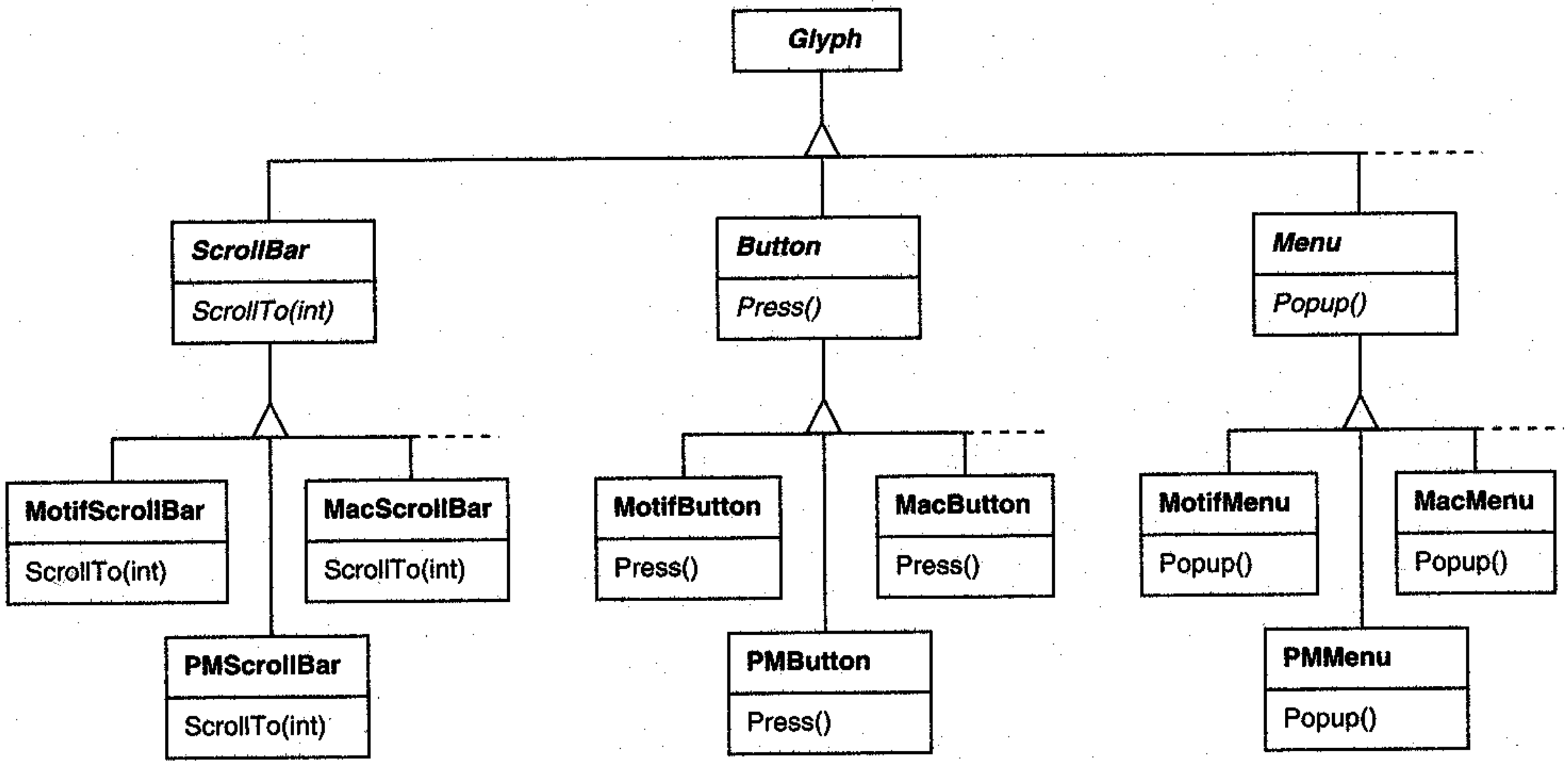


Figure 2.10: Abstract product classes and concrete subclasses

## Supporting Multiple Window Systems

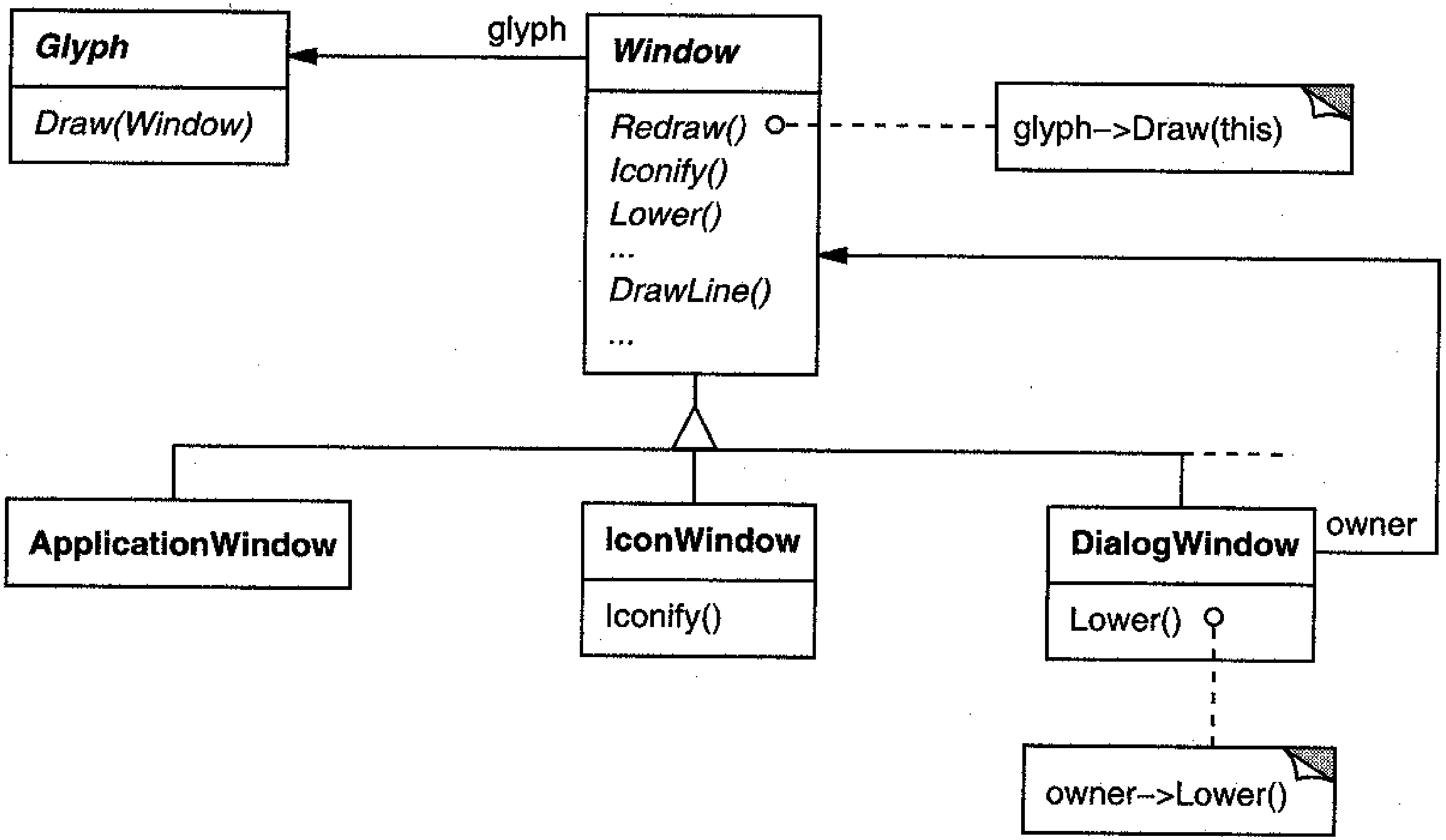
- We'd like Lexi to run on many existing window systems having different programming interfaces
- Can we use an Abstract Factory?
  - As the different programming interfaces on these existing window systems, the Abstract Factory pattern doesn ‘t work
  - We need a uniform set of windowing abstractions that lets us take different window system impelementations and slide any one of them under a common interface

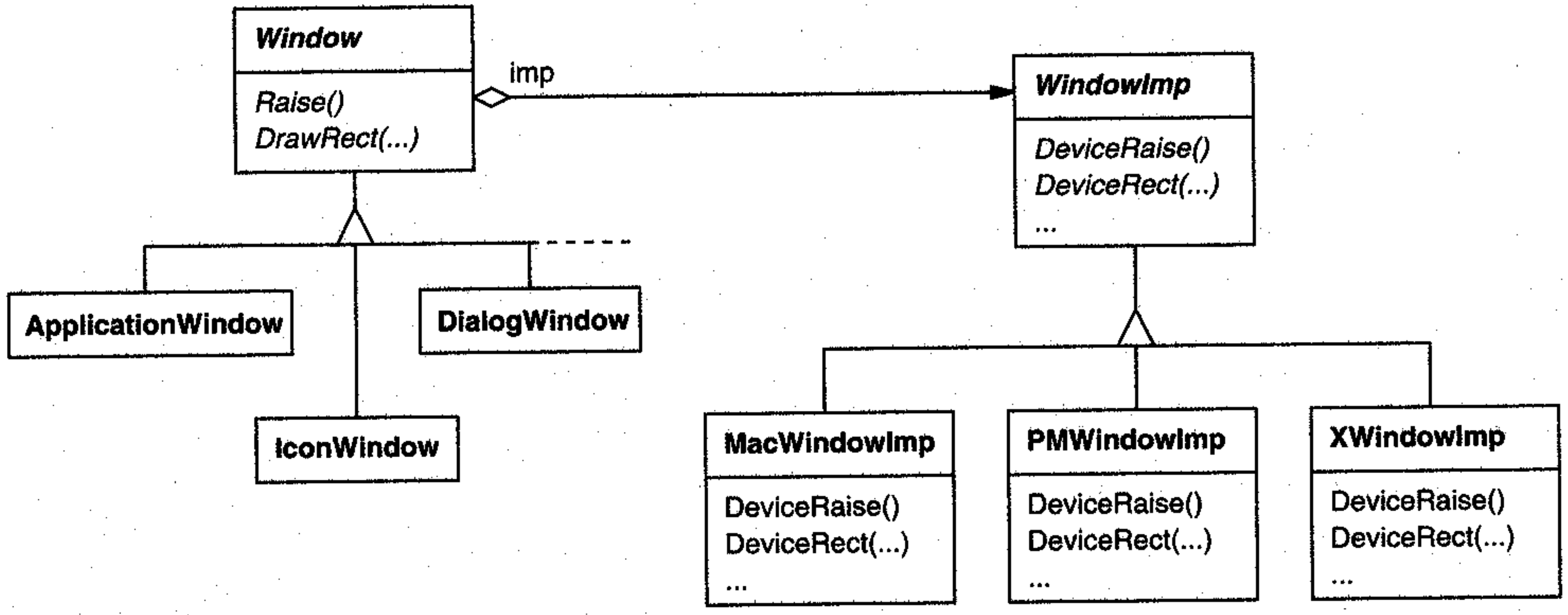
## Supporting Multiple Window Systems (cont.)

- Encapsulating Implementation Dependencies
  - The Window class interface encapsulates the things windows tend to do across window systems
  - The Window class is an abstract class
  - Where does the implementation live?
- Window and WindowImp
- Bridge Pattern
  - to allow separate class hierarchies to work together even as they evolve independently

| Responsibility | Operations |
|---|---|
| window management | `virtual void Redraw()`<br>`virtual void Raise()`<br>`virtual void Lower()`<br>`virtual void Iconify()`<br>`virtual void Deiconify()`<br>`...` |
| graphics | `virtual void DrawLine(...)`<br>`virtual void DrawRect(...)`<br>`virtual void DrawPolygon(...)`<br>`virtual void DrawText(...)`<br>`...` |

Table 2.3: Window class interface


Glyph
Draw(Window)
glyph
Window
Redraw()
Iconify()
Lower()
...
DrawLine()
...
glyph->Draw(this)
ApplicationWindow
IconWindow
Iconify()
DialogWindow
Lower()
owner
owner->Lower()


Window
Raise()
DrawRect(...)
imp
WindowImp
DeviceRaise()
DeviceRect(...)
...
ApplicationWindow
IconWindow
DialogWindow
MacWindowImp
DeviceRaise()
DeviceRect(...)
...
PMWindowImp
DeviceRaise()
DeviceRect(...)
...
XWindowImp
DeviceRaise()
DeviceRect(...)
...

## User Operations

- Requirements
  - Lexi provides different user interfaces for the operations it supported
  - These operations are implemented in many different classes
  - Lexi supports undo and redo
- The challenge is to come up with a simple and extensible mechanism that satisfies all of these needs

## User Operations (cont.)

- Encapsulating a Request
  - We could parameterize MenuItem with a *function* to call, but that's not a complete solution
    - it doesn't address the undo/redo problem
    - it's hard to associate state with a function
    - functions are hard to extent, and it's hard to reuse part of them
  - We should parameterize MenuItems with an ***object***, not a function

## User Operations (cont.)

- Command Class and Subclasses
  - The Command abstract class consists of a single abstract operation called “Execute”
  - MenuItem can store a Command object that encapsulates a request
  - When a user choose a particular menu item, the MenuItem simply calls Execute on its Command object to carry out the request

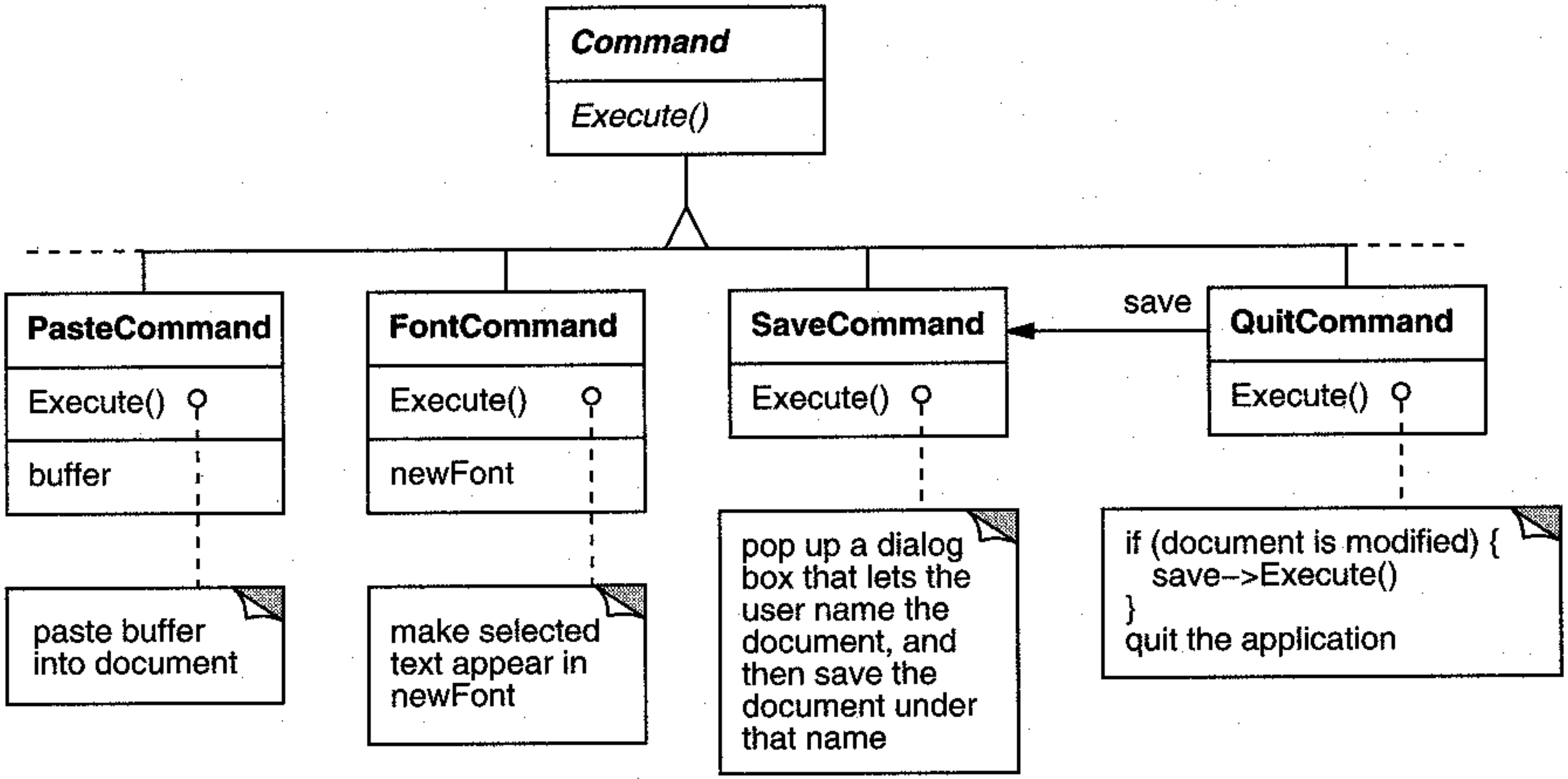

Command
Execute()
PasteCommand
Execute()
buffer
paste buffer into document
FontCommand
Execute()
newFont
make selected text appear in newFont
SaveCommand
Execute()
pop up a dialog box that lets the user name the document, and then save the document under that name
save
QuitCommand
Execute()
if (document is modified) {
save->Execute()
}
quit the application

Figure 2.11: Partial Command class hierarchy

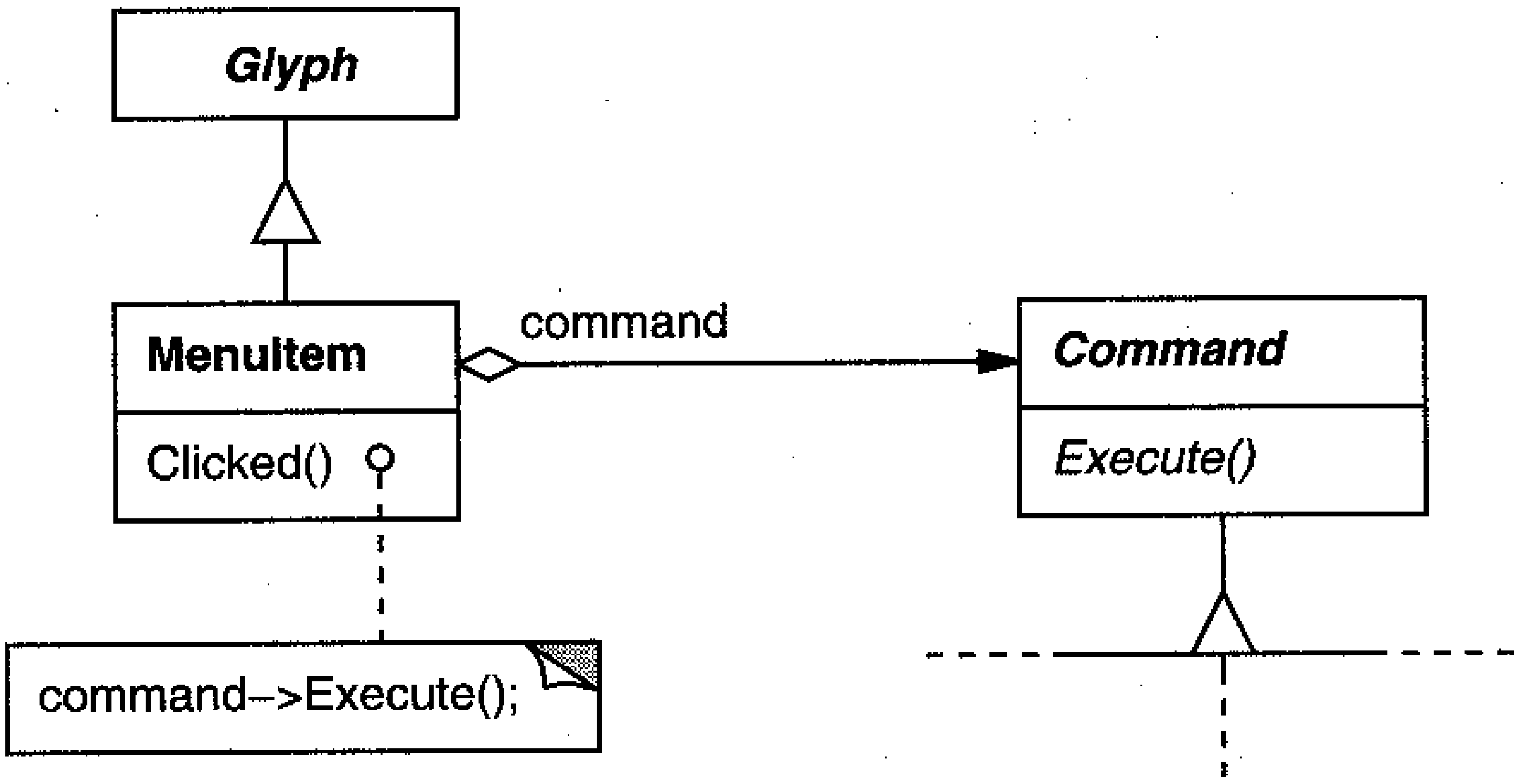
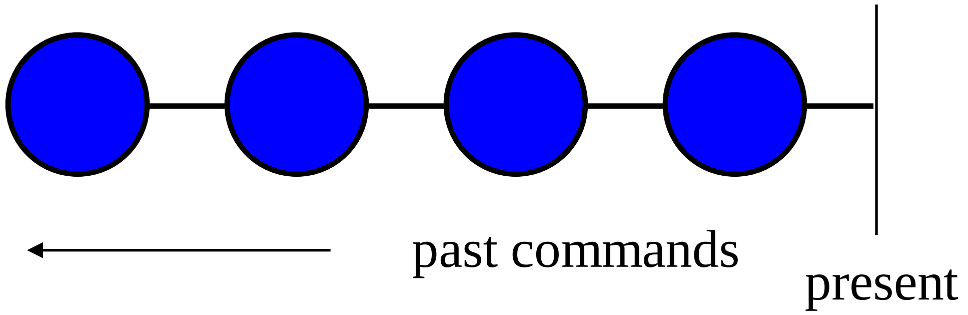


Figure 2.12: MenuItem-Command relationship

## User Operations (cont.)

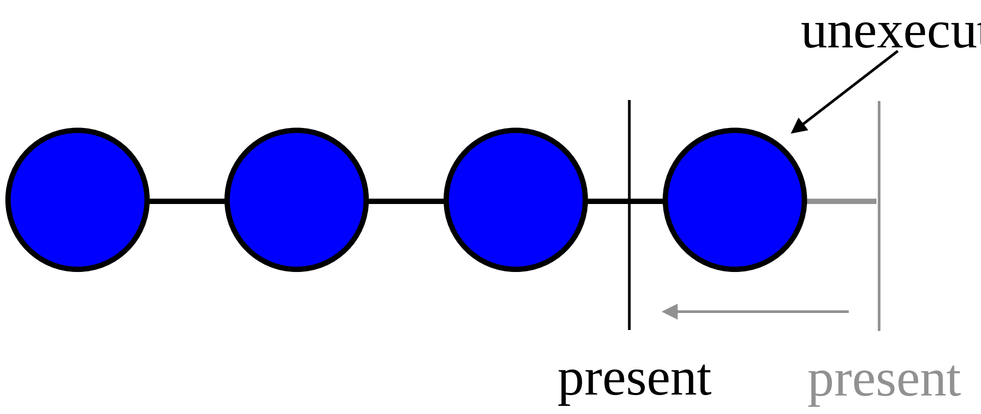
- Undoability
  - To undo and redo commands, we add an *Unexecute* operation to Command's interface
  - A concrete Command would store the state of the Command for Unexecute
  - Reversible operation returns a Boolean value to determine if a command is undoable
- Command History
  - a list of commands that have been executed

# Implementing a Command History



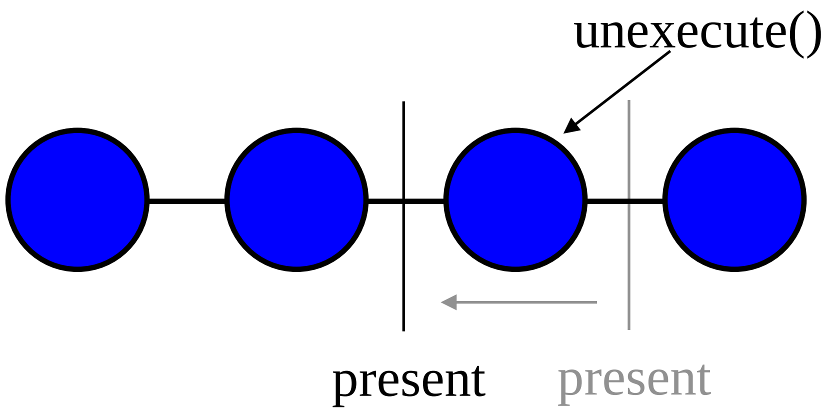
- The command history can be seen as a list of past commands commands
- As new commands are executed they are added to the front of the history

## Undoing the Last Command



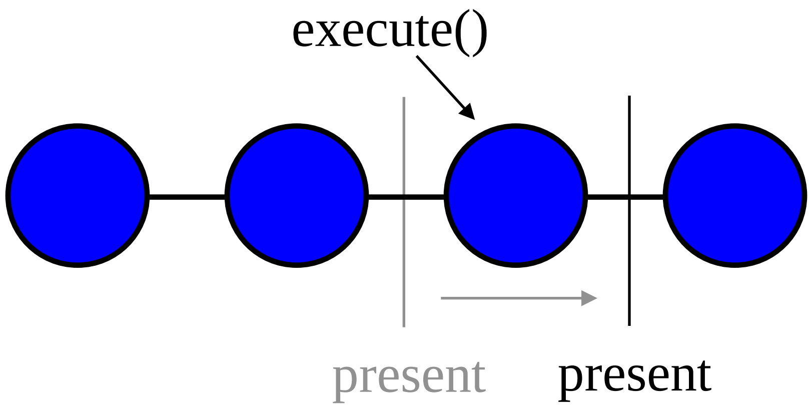
- To undo a command, unexecute() is called on the command on the front of the list
- The "present" position is moved past the last command

## Undoing the Previous Command



- To undo the previous command, unexecute() is called on the next command in the history
- The present pointer is moved to point before that command

## Redoing the Next Command



- To redo the command that was just undone, execute() is called on that command
- The present pointer is moved up past that command

## The Command Pattern

- Encapsulate a request as an object
- The Command Patterns lets you
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations
- Also Known As: Action, Transaction
- Covered on pg. 233 in the book

## Spelling Checking & Hyphenation

Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

Constraints/forces:

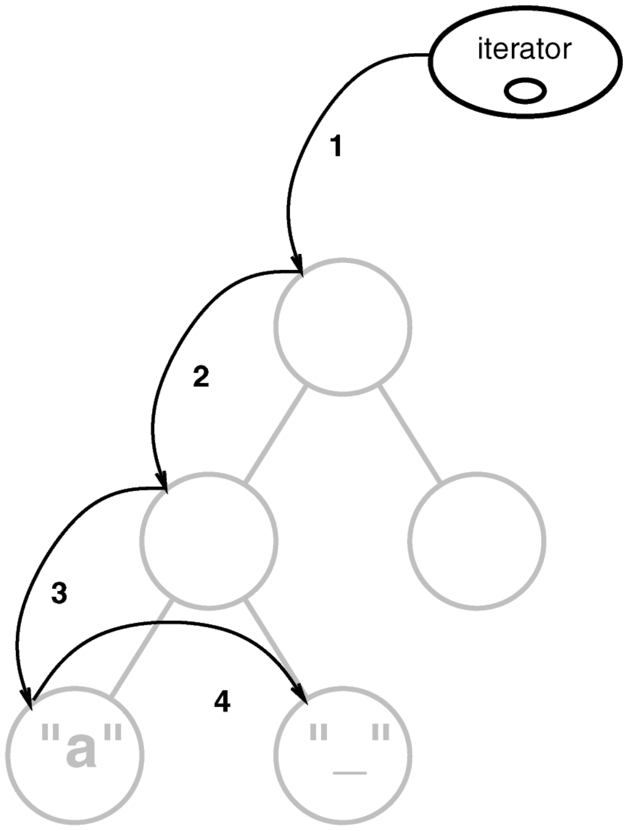
- support multiple algorithms
- don’t tightly couple algorithms with document structure

## Spelling Checking & Hyphenation (cont'd)

## Solution: Encapsulate Traversal

**Iterator**

- encapsulates a traversal algorithm without exposing representation details to callers
- uses Glyph's child enumeration operation
- This is an example of a "preorder iterator"



## Spelling Checking & Hyphenation (cont'd)

## ITERATOR object behavioral

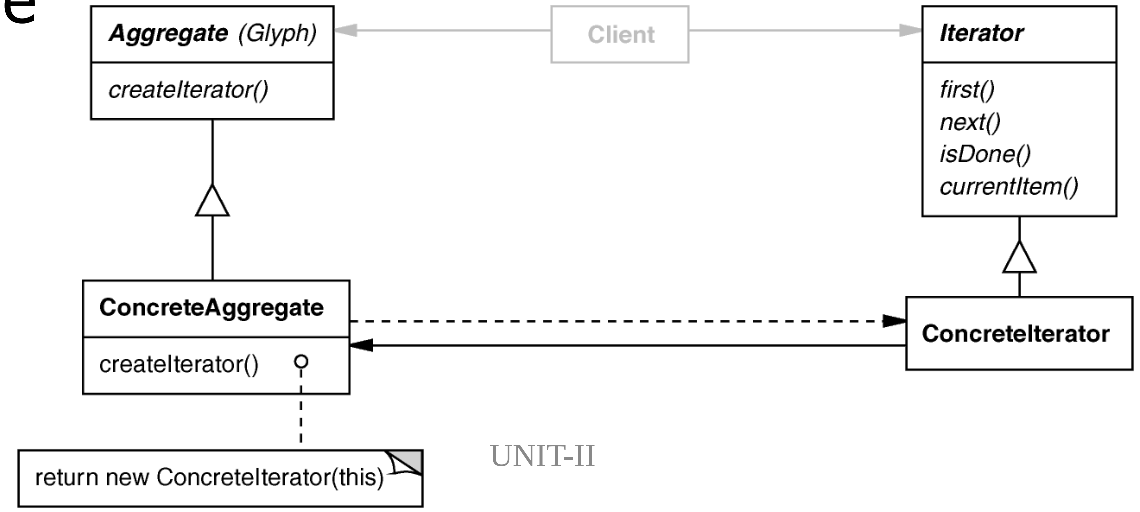
### Intent

access elements of a container without exposing its representation

### Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

### Structure



## Spelling Checking & Hyphenation (cont'd)

## ITERATOR (cont'd) object behavioral

Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {
   vector<string> args;
   for (int i = 0; i < argc; i++)
               args.push_back (string (argv[i]));
   for (vector<string>::iterator i (args.begin ());
        i != args.end ();
        i++)
     cout << *i;
   cout << endl;
   return 0;
}
```

The same iterator pattern can be applied to any STL container!

```
for (Glyph::iterator  i = glyphs.begin ();
     i != glyphs.end ();
     i++)
   ...
```

## Spelling Checking & Hyphenation (cont'd)

## ITERATOR (cont'd) object behavioral

### Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
− additional communication overhead between iterator & aggregate

### Implementation

– internal versus external iterators
– violating the object structure's encapsulation
– robust iterators
– synchronization overhead in multi-threaded programs
– batching in distributed & concurrent programs

### Known Uses

– C++ STL iterators
– JDK Enumeration, Iterator
– Unidraw Iterator

## Spelling Checking & Hyphenation (cont'd)

## Visitor

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's **accept(Visitor)** at each node
- **accept()** calls back on visitor (a form of "static polymorphism" based on method overloading by type)

```
void Character::accept (Visitor &v) { v.visit (*this); }

class Visitor {
public:
   virtual void visit (Character &);
   virtual void visit (Rectangle &);
   virtual void visit (Row &);
   // etc. for all relevant Glyph subclasses
};
```

## Spelling Checking & Hyphenation (cont'd)

## SpellingCheckerVisitor

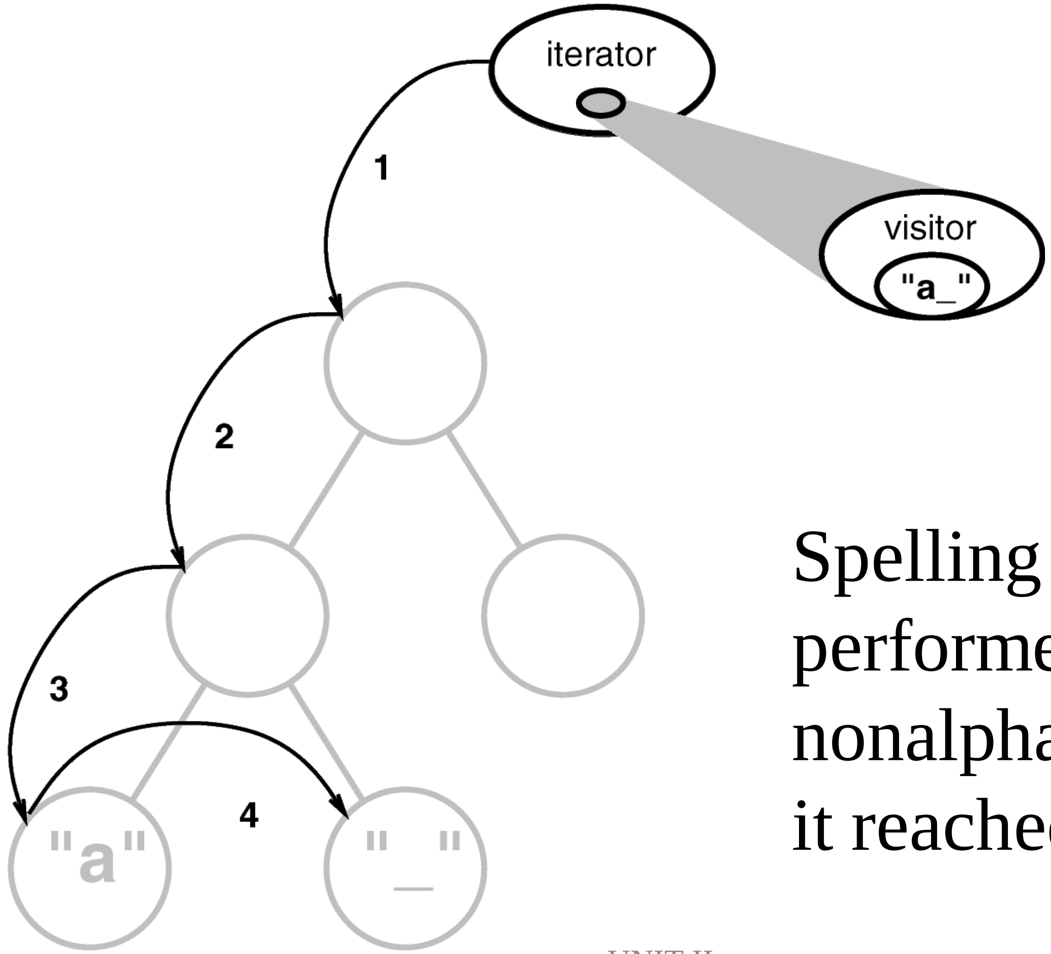
- gets character code from each character glyph

  Can define **getCharCode()** operation just on **Character()** class
- checks words accumulated from character glyphs
- combine with **PreorderIterator**

```
class SpellCheckerVisitor : public Visitor {
public:
   virtual void visit (Character &);
   virtual void visit (Rectangle &);
   virtual void visit (Row &);
   // etc. for all relevant Glyph subclasses
Private:
    std::string accumulator_;
};
```

## Spelling Checking & Hyphenation (cont'd)

## Accumulating Words

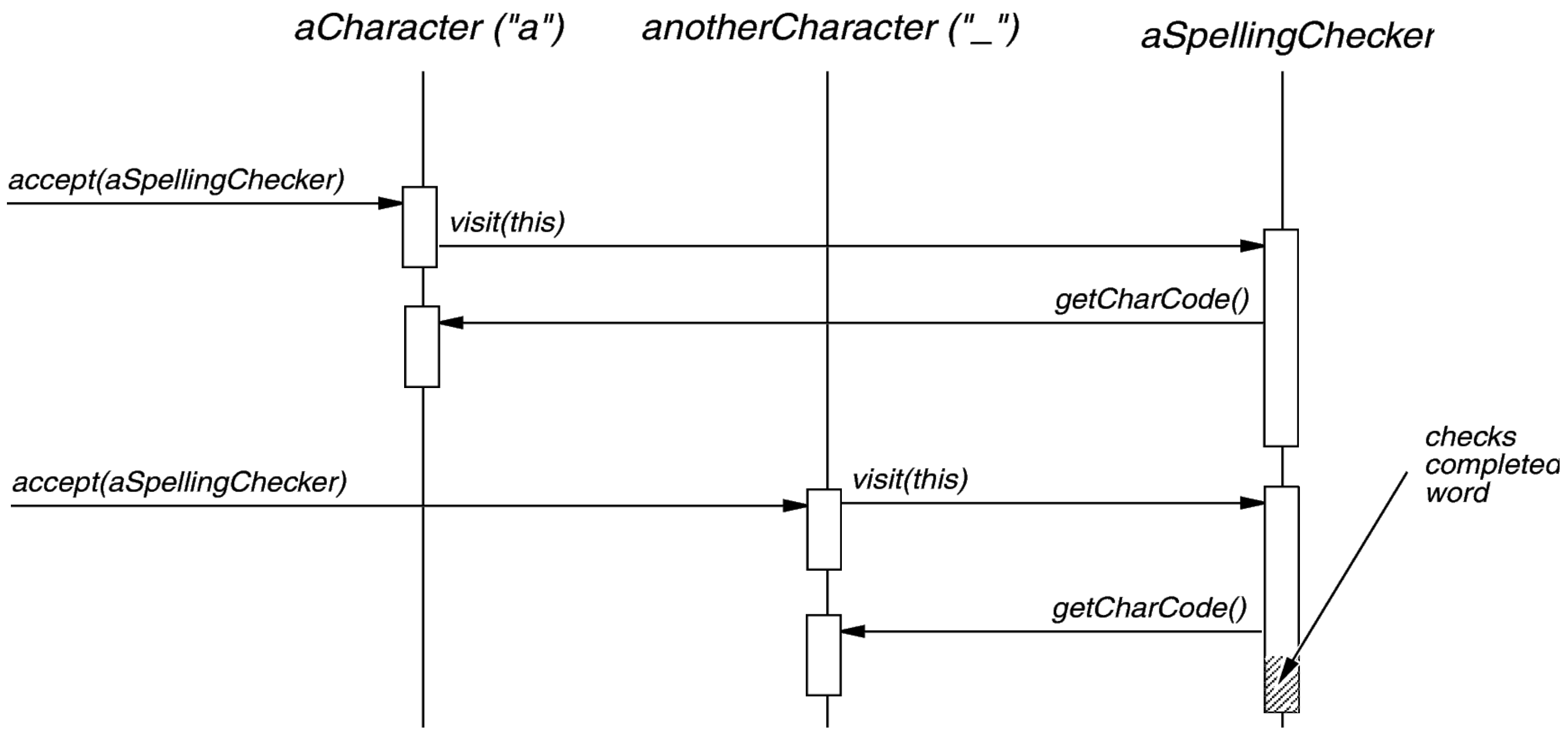


Spelling check performed when a nonalphabetic character it reached

Spelling Checking & Hyphenation (cont'd)

## Interaction Diagram

- The iterator controls the order in which accept() is called on each glyph in the composition
- accept() then "visits" the glyph to perform the desired action
- The Visitor can be sub-classed to implement various desired actions



## Spelling Checking & Hyphenation (cont’d)
## HyphenationVisitor

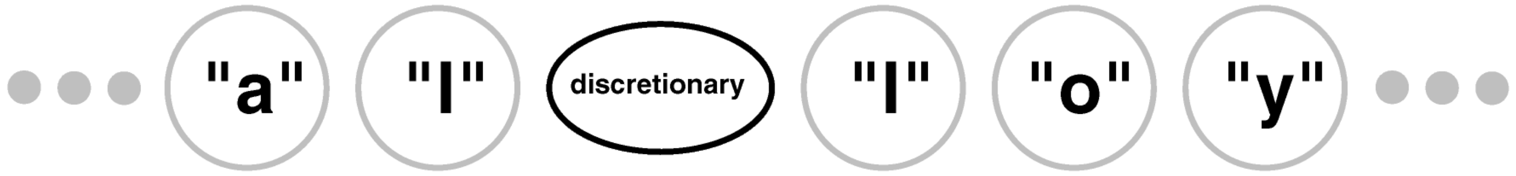
- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

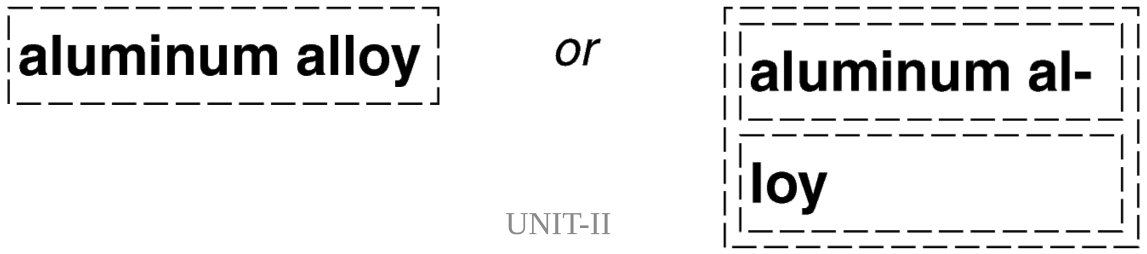
```
class HyphenationVisitor : public Visitor {
public:
    void visit (Character &);
    void visit (Rectangle &);
    void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

Spelling Checking & Hyphenation (cont’d)

## **Discretionary** Glyph

- looks like a hyphen when at end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks





## Spelling Checking & Hyphenation (cont'd)

**VISITOR** object behavioral

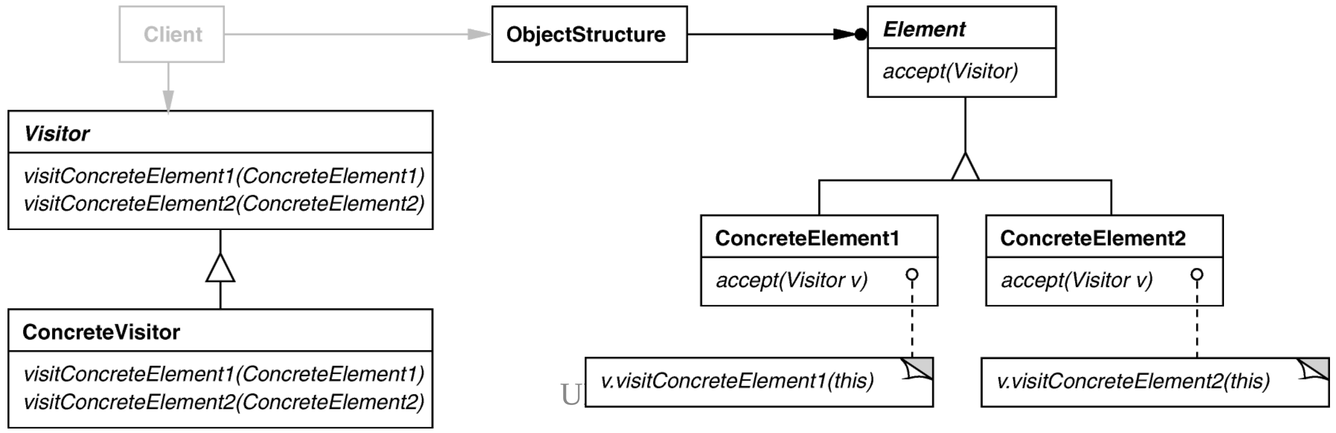
### Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

### Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

### Structure


Client
ObjectStructure
Element
accept(Visitor)
Visitor
visitConcreteElement1(ConcreteElement1)
visitConcreteElement2(ConcreteElement2)
ConcreteVisitor
visitConcreteElement1(ConcreteElement1)
visitConcreteElement2(ConcreteElement2)
ConcreteElement1
accept(Visitor v)
ConcreteElement2
accept(Visitor v)
v.visitConcreteElement1(this)
v.visitConcreteElement2(this)

## Spelling Checking & Hyphenation (cont'd)

### VISITOR (cont'd) object behavioral

```
SpellCheckerVisitor spell_check_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (spell_check_visitor);
}

HyphenationVisitor hyphenation_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (hyphenation_visitor);
}
```

## Spelling Checking & Hyphenation (cont'd)

## VISITOR (cont'd) object behavioral

### Consequences

+ flexibility: visitor & object structure are independent
+ localized functionality
− circular dependency between Visitor & Element interfaces
− Visitor brittle to new ConcreteElement classes

### Implementation

– double dispatch
– general interface to elements of object structure

### Known Uses

– ProgramNodeEnumerator in Smalltalk-80 compiler
– IRIS Inventor scene rendering
– TAO IDL compiler to handle different backends

Part III: Wrap-Up

## Concluding Remarks

- design reuse
- uniform design vocabulary
- understanding, restructuring, & team communication
- provides the basis for automation
- a “new” way to think about design

# Pattern References

Books

Timeless Way of Building, Alexander, ISBN 0-19-502402-8

A Pattern Language, Alexander, 0-19-501-919-9

Design Patterns, Gamma, et al., 0-201-63361-2 CD version 0-201-63498-8

Pattern-Oriented Software Architecture, Vol. 1, Buschmann, et al., 0-471-95869-7

Pattern-Oriented Software Architecture, Vol. 2, Schmidt, et al., 0-471-60695-2

Pattern-Oriented Software Architecture, Vol. 3, Jain & Kircher, 0-470-84525-2

Pattern-Oriented Software Architecture, Vol. 4, Buschmann, et al., 0-470-05902-8

# Pattern References (cont'd)

**More Books**

*Analysis Patterns*, Fowler; 0-201-89542-0

*Concurrent Programming in Java, 2nd ed.*, Lea, 0-201-31009-0

*Pattern Languages of Program Design*

Vol. 1, Coplien, et al., eds., ISBN 0-201-60734-4

Vol. 2, Vlissides, et al., eds., 0-201-89527-7

Vol. 3, Martin, et al., eds., 0-201-31011-2

Vol. 4, Harrison, et al., eds., 0-201-43304-4

Vol. 5, Manolescu, et al., eds., 0-321-32194-4

*AntiPatterns*, Brown, et al., 0-471-19713-0

*Applying UML & Patterns, 2nd ed.*, Larman, 0-13-092569-1

*Pattern Hatching*, Vlissides, 0-201-43293-5

*The Pattern Almanac 2000*, Rising, 0-201-61567-3

## Pattern References (cont’d)

**Even More Books**

Small Memory Software, Noble & Weir, 0-201-59607-5

Microsoft Visual Basic Design Patterns, Stamatakis, 1-572-31957-7

Smalltalk Best Practice Patterns, Beck; 0-13-476904-X

The Design Patterns Smalltalk Companion, Alpert, et al., 0-201-18462-1

Modern C++ Design, Alexandrescu, ISBN 0-201-70431-5

Building Parsers with Java, Metsker, 0-201-71962-2

Core J2EE Patterns, Alur, et al., 0-130-64884-1

Design Patterns Explained, Shalloway & Trott, 0-201-71594-5

The Joy of Patterns, Goldfedder, 0-201-65759-7

The Manager Pool, Olson & Stimmel, 0-201-72583-5

## Pattern References (cont'd)

**Early Papers**

"Object-Oriented Patterns," P. Coad; Comm. of the ACM, 9/92

"Documenting Frameworks using Patterns," R. Johnson; OOPSLA '92

"Design Patterns: Abstraction & Reuse of Object-Oriented Design," Gamma, Helm, Johnson, Vlissides, ECOOP '93

**Articles**

Java Report, Java Pro, JOOP, Dr. Dobb's Journal, Java Developers Journal, C++ Report

## Pattern-Oriented Conferences

**PLoP 2007**: Pattern Languages of Programs
October 2007, Collocated with OOPSLA

**EuroPLoP 2007**, July 2007, Kloster Irsee, Germany

…

See hillside.net/conferences/ for
up-to-the-minute info.

## Mailing Lists

patterns@cs.uiuc.edu: present & refine patterns

patterns-discussion@cs.uiuc.edu: general discussion

gang-of-4-patterns@cs.uiuc.edu: discussion on *Design Patterns*

siemens-patterns@cs.uiuc.edu: discussion on
*Pattern-Oriented Software Architecture*

ui-patterns@cs.uiuc.edu: discussion on user interface patterns

business-patterns@cs.uiuc.edu: discussion on patterns for business processes

ipc-patterns@cs.uiuc.edu: discussion on patterns for distributed systems

See http://hillside.net/patterns/mailing.htm for an up-to-date list.

## unit-2 part-2

| S.NO. | TOPIC | PPT Slides | |
|---|---|---|---|
| 1 | Creational Pattern Part-I Introduction | L1 | 4 – 8 |
| 2 | Abstract Factory | L2 | 9 – 28 |
| 3 | Builder | L3 | 29 – 39 |
| 4 | Factory Method | L4 | 40 – 47 |
| 5 | Prototype | L5 | 48 – 54 |
| 6 | Singleton | L6 | 55 – 66 |
| 7 | Repeated key points for Structural Patterns (Intent, Motivation, Also Known As ……………) | | |
| 8 | (discussion of Creational patterns) Review | | |

## Creational Patterns

- Abstracts instantiation process
- Makes system independent of how its objects are
  - created
  - composed
  - represented
- Encapsulates knowledge about which concrete classes the system uses
- Hides how instances of these classes are created and put together

## Creational Patterns

- Abstract the instantiation process
  - Make a system independent of how objects are created, composed, and represented
- Important if systems evolve to depend more on object composition than on class inheritance
  - Emphasis shifts from hardcoding fixed sets of behaviors towards a smaller set of composable fundamental behaviors
- Encapsulate knowledge about concrete classes a system uses
- Hide how instances of classes are created and put together

## What are creational patterns?

- Design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- Make a system independent of the way in which objects are created, composed and represented
- Recurring themes:
  - Encapsulate knowledge about which concrete classes the system uses (so we can change them easily later)
  - Hide how instances of these classes are created and put together (so we can change it easily later)

## Benefits of creational patterns

- Creational patterns let you program to an interface defined by an abstract class
- That lets you configure a system with "product" objects that vary widely in structure and functionality
- Example: GUI systems
  - InterViews GUI class library
  - Multiple look-and-feels
  - Abstract Factories for different screen components

## Benefits of creational patterns

- Generic instantiation – Objects are instantiated without having to identify a specific class type in client code (Abstract Factory, Factory)
- Simplicity – Make instantiation easier: callers do not have to write long complex code to instantiate and set up an object (Builder, Prototype pattern)
- Creation constraints – Creational patterns can put bounds on *who* can create objects, *how* they are created, and *when* they are created

## Abstract Factory Pattern

## Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes

## ABSTRACT FACTORY (Object Creational)

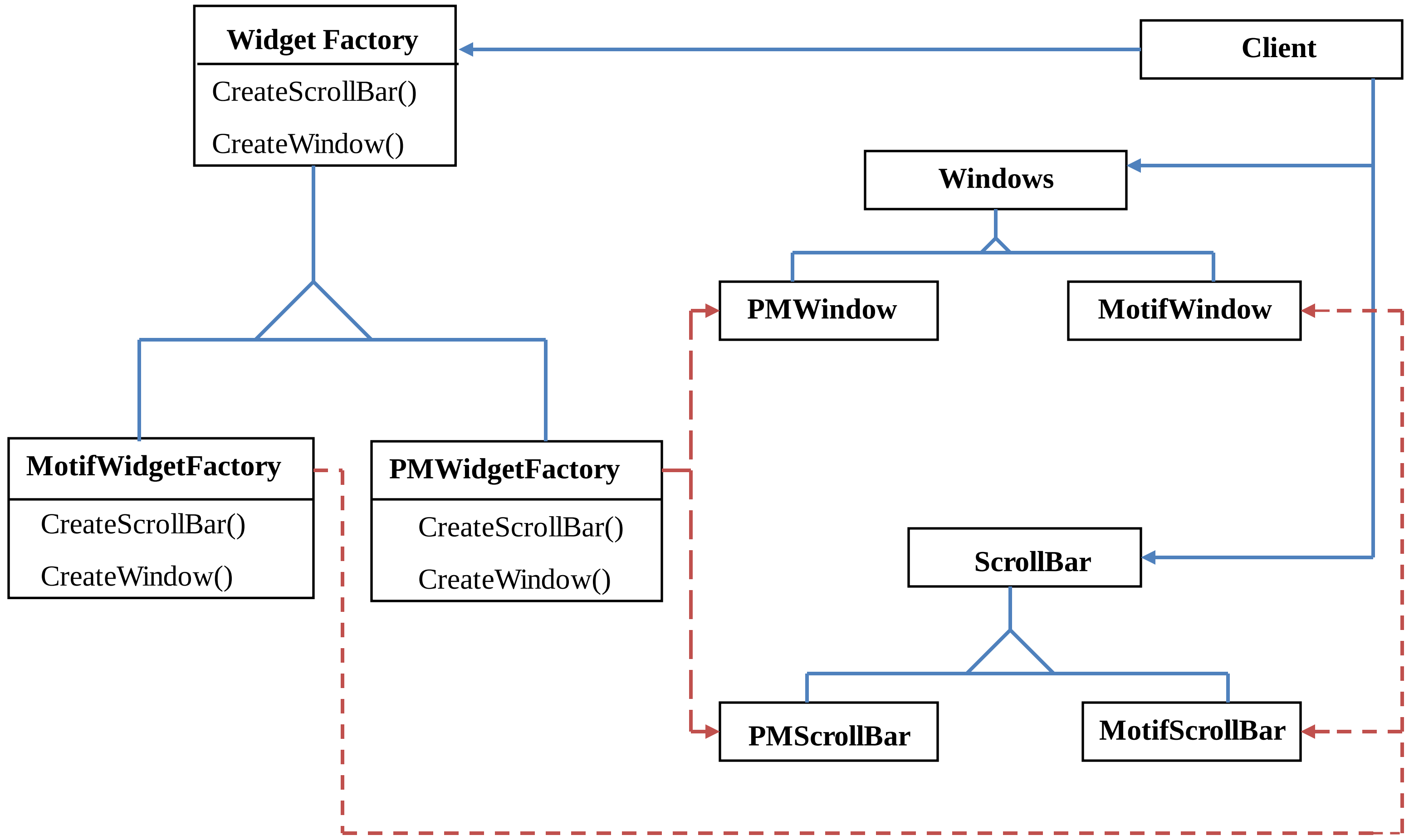
- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

- **Also Known As:** Kit.

## Motivation

- Motivation:
  - User interface toolkit supports multiple look-and-feel standards
(Motif, Presentation Manager)
  - Different appearances and behaviors for UI widgets
  - Apps should not hard-code its widgets

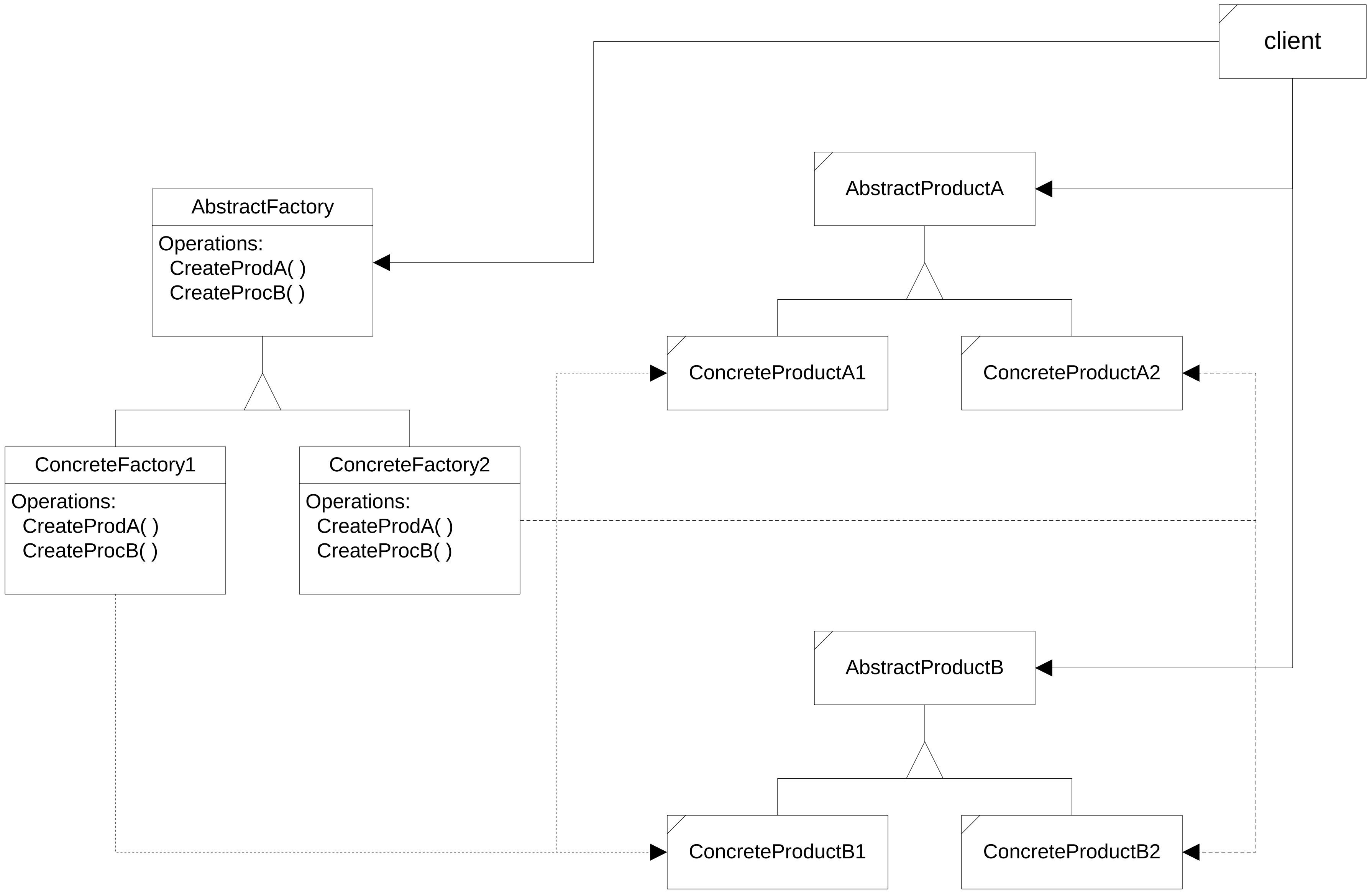
# ABSTRACT FACTORY Motivation



## Solution:

- Solution:
  - Abstract Widget Factory class
  - Interfaces for creating each basic kind of widget
  - Abstract class for each kind of widgets,
  - Concrete classes implement specific look-and-feel.

# Abstract Factory Structure



## 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 produces
- 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

## ABSTRACT FACTORY Participants

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

## • ABSTRACT FACTORY Participants(cont..)

- Concrete Product:

  •Defines a product object to be created by concrete factory

  •Implements the abstract product interface

- Client:

  •Uses only interfaces declared by Abstract Factory and AbstractProduct classes

## Collaborators

- Usually only one ConcreteFactory instance is used fo an activation, matched to a specific application context. It builds a specific product family for client use -- the client doesn't care which family is used -- it simply needs the services appropriate for the current context.

- The client may use the AbstractFactory interface to initiate creation, or some other agent may use the AbstractFactory on the client's behalf.

## Presentation Remark

- Here, we often use a sequence diagram (event-trace) to show the dynamic interactions between participants.

- For the Abstract Factory Pattern, the dynamic interaction is simple, and a sequence diagram would not add much new information.

## Consequences

- The Abstract Factory Pattern has the following benefits:
  - It isolates concrete classes from the client.
    - You use the Abstract Factory to control the classes of objects the client creates.
    - Product names are isolated in the implementation of the ConcreteFactory, clients use the instances through their abstract interfaces.
  - Exchanging product families is easy.
    - None of the client code breaks because the abstract interfaces don’t change.
    - Because the abstract factory creates a complete family of products, the whole product family changes when the concrete factory is changed.

## Consequences

- More benefits of the Abstract Factory Pattern
  - It supports the imposition of constraints on product families, e.g., always use A1 and B1 together, otherwise use A2 and B2 together.

## Consequences

- The Abstract Factory pattern has the following liability:
  - Adding new kinds of products to existing factory is difficult.
    - Adding a new product requires extending the abstract interface which implies that all of its derived concrete classes also must change.
    - Essentially everything must change to support and use the new product family
      - abstract factory interface is extended
      - derived concrete factories must implement the extensions

## Implementation

- Concrete factories are often implemented as <u>singletons</u>.

- Creating the products
  - Concrete factory usually use the <u>factory method</u>.
    - simple
    - new concrete factory is required for each product family

  - alternately concrete factory can be implemented using <u>prototype</u>.
    - only one is needed for all families of products
    - product classes now have special requirements - they participate in the creation

# Implementation

- Concrete factories are often implemented as <u>singletons</u>.

- Creating the products
  - Concrete factory usually use the <u>factory method</u>.
    - simple
    - new concrete factory is required for each product family

  - alternately concrete factory can be implemented using <u>prototype</u>.
    - only one is needed for all families of products
    - product classes now have special requirements - they participate in the creation

## Implementation

- Defining extensible factories by using create function with an argument
  - only one virtual create function is needed for the AbstractFactory interface
  - all products created by a factory must have the same base class or be able to be safely coerced to a given type
  - it is difficult to implement subclass specific operations

## Know Uses

- Interviews
  - used to generate “look and feel” for specific user interface objects
  - uses the Kit suffix to denote AbstractFactory classes, e.g., WidgetKit and DialogKit.
  - also includes a layoutKit that generates different composite objects depending on the needs of the current context

ET++

- another windowing library that uses the AbstractFactory to achieve portability across different window systems (X Windows and SunView).

- COM – Microsoft’s Component Object Model

## Related Patterns

- Factory Method -- a “virtual” constructor
- Prototype -- asks products to clone themselves
- Singleton -- allows creation of only a single instance

## Code Examples

- Skeleton Example
  - Abstract Factory Structure
  - Skeleton Code
- Neural Net Example
  - Neural Net Physical Structure
  - Neural Net Logical Structure
  - Simulated Neural Net Example

# BUILDER (Object Creational)

- Intent:

  Separate the construction of a complex object from its representation so that the same construction process can create different representations

- Motivation:
  - RTF reader should be able to convert RTF to many text format
  - Adding new conversions without modifying the reader should be easy

- Solution:
  - Configure RTFReader class with a Text Converter object
  - Subclasses of Text Converter specialize in different conversions and formats
  - TextWidgetConverter will produce a complex UI object and lets the user see and edit the text

## Implementation

- Concrete factories are often implemented as singletons.

- Creating the products
  - Concrete factory usually use the factory method.
    - simple
    - new concrete factory is required for each product family

  - alternately concrete factory can be implemented using prototype.
    - only one is needed for all families of products
    - product classes now have special requirements - they participate in the creation

## Implementation

- Defining extensible factories by using create function with an argument
  - only one virtual create function is needed for the AbstractFactory interface
  - all products created by a factory must have the same base class or be able to be safely coerced to a given type
  - it is difficult to implement subclass specific operations

## Know Uses

- Interviews
  - – used to generate "look and feel" for specific user interface objects
  - – uses the Kit suffix to denote AbstractFactory classes, e.g., WidgetKit and DialogKit.
  - also includes a layoutKit that generates different composite objects depending on the needs of the current context

  ET++

  - – another windowing library that uses the AbstractFactory to achieve portability across different window systems (X Windows and SunView).

- COM – Microsoft's Component Object Model

## Related Patterns

- Factory Method -- a “virtual” constructor
- Prototype -- asks products to clone themselves
- Singleton -- allows creation of only a single instance

# Code Examples

- Skeleton Example
  - Abstract Factory Structure
  - Skeleton Code
- Neural Net Example
  - Neural Net Physical Structure
  - Neural Net Logical Structure
  - Simulated Neural Net Example

# BUILDER (Object Creational)

- Intent:

  Separate the construction of a complex object from its representation so that the same construction process can create different representations

- Motivation:
  - RTF reader should be able to convert RTF to many text format
  - Adding new conversions without modifying the reader should be easy

- Solution:
  - •Configure RTFReader class with a Text Converter object
  - •Subclasses of Text Converter specialize in different conversions and formats
  - •TextWidgetConverter will produce a complex UI object and lets the user see and edit the text

## Why do we use Builder?

- Common manner to Create an Instance
  - ***Constructor!***
  - Each Parts determined by Parameter of the Constructor

```
public class Room {
    private int area;
    private int windows;
    public String purpose;

    Room() {
    }

    Room(int newArea, int newWindows, String newPurpose){
        area = newArea;
        windows = newWindows;
        purpose = newPurpose;
    }
}
```

There are Only 2 different ways to Create an Instance part-by-part.

## Why do we use Builder?

- In the previous example,
  - We can either determine all the arguments or determine nothing and just construct. We can't determine arguments partially.
  - We can't control whole process to Create an instance.
  - ***Restriction of ways to Create an Object***

    ☞ ***Bad Abstraction & Flexibility***

## Discussion

- Uses Of Builder
  - Parsing Program(RTF converter)
  - GUI

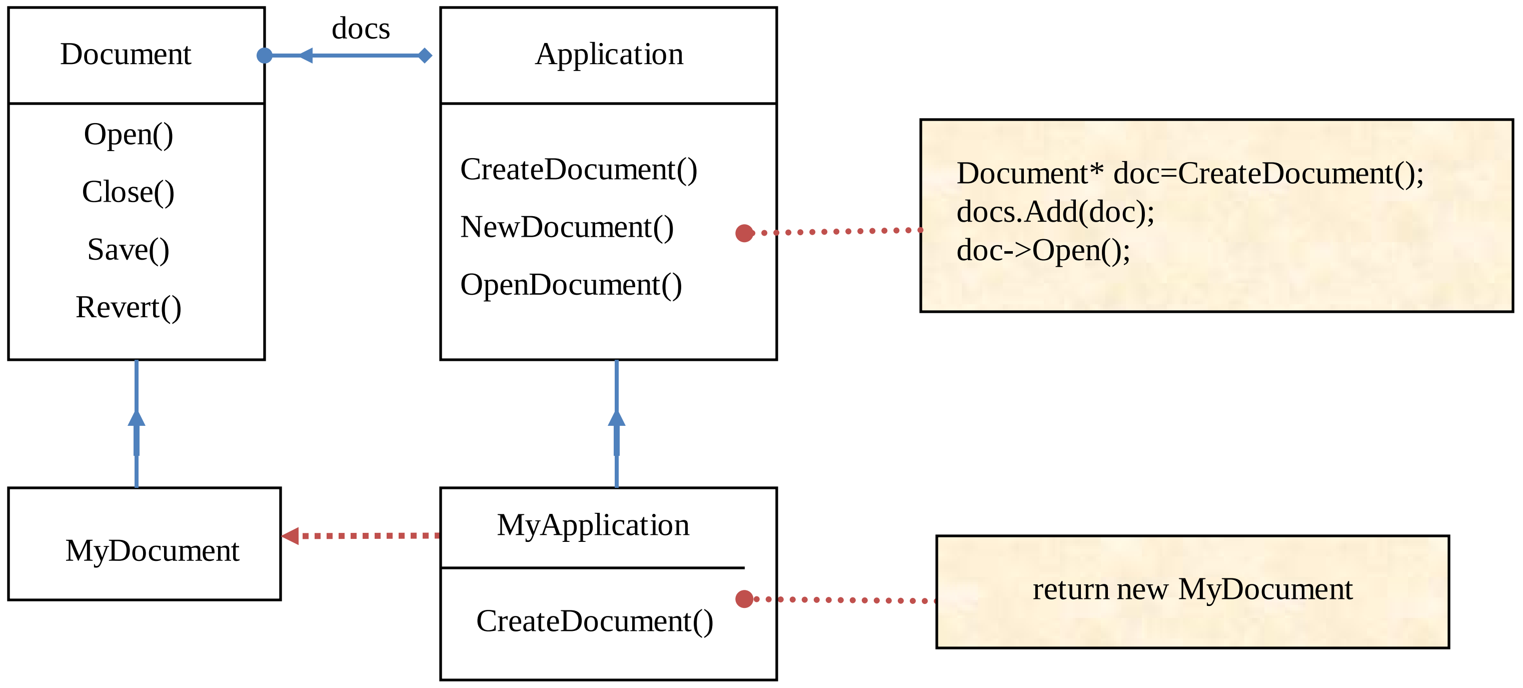
# FACTORY METHOD (Class Creational)

- 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:
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate

## Motivation:

- Motivation: Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework
- Also Known As: Virtual Constructor

# FACTORY METHOD
## Motivation



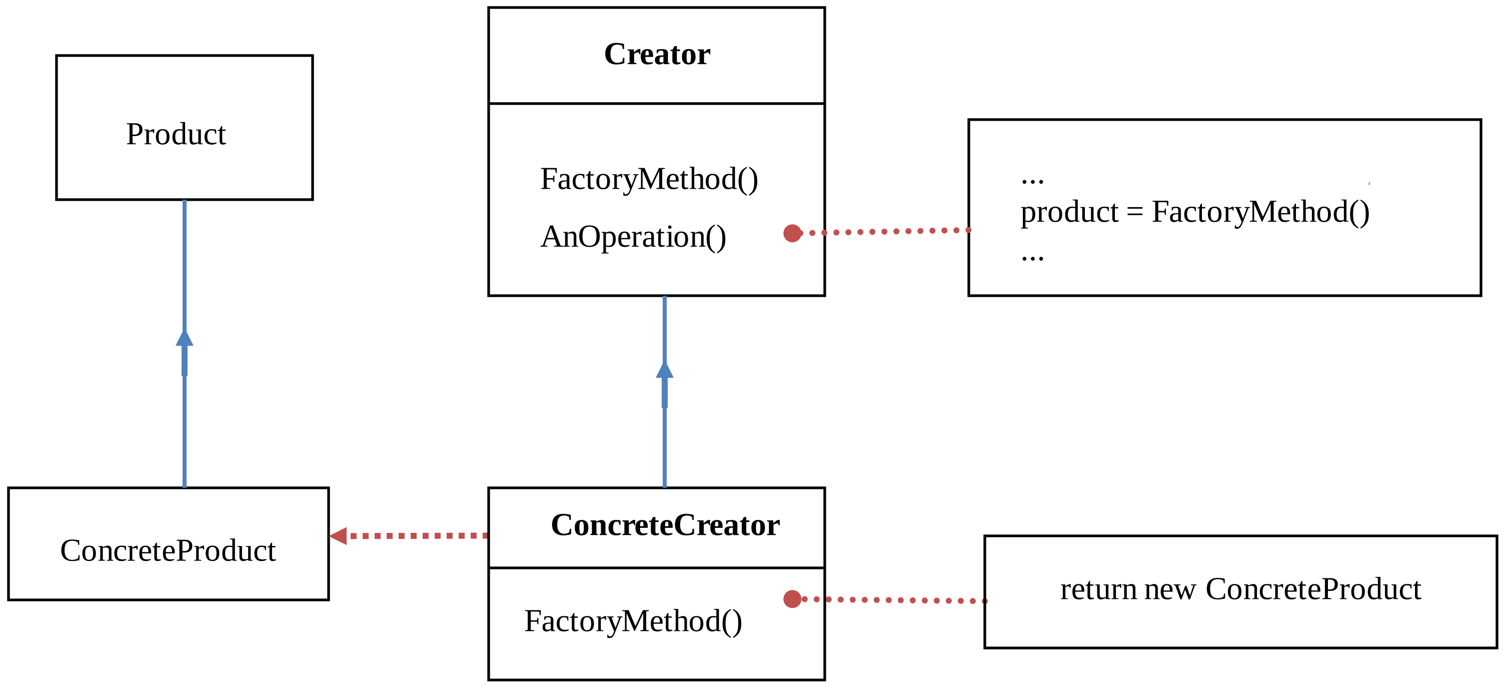
```
Document* doc=CreateDocument();
docs.Add(doc);
doc->Open();
```

```
return new MyDocument
```

## 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.

# FACTORY METHOD
## Structure

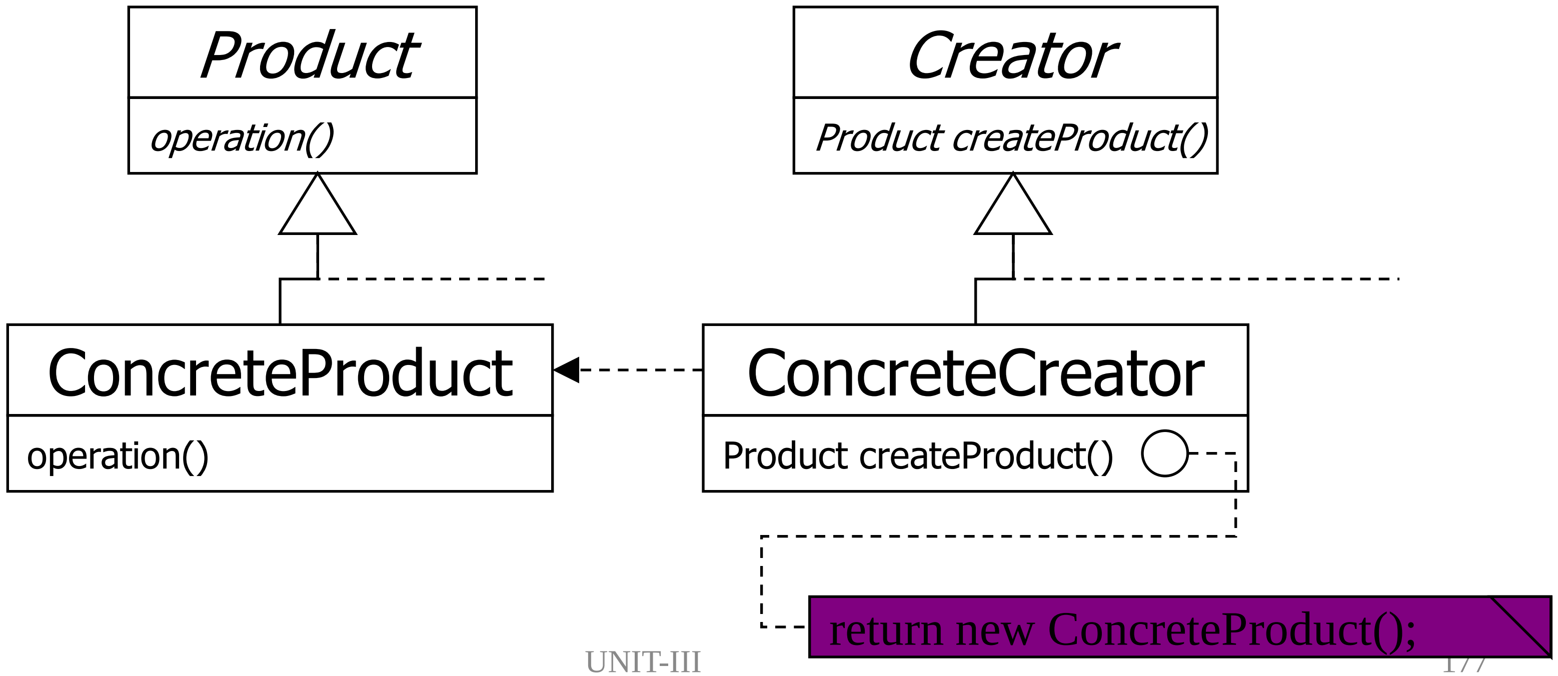


# Participants

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type product
  - May contain a default implementation of the factory method
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.

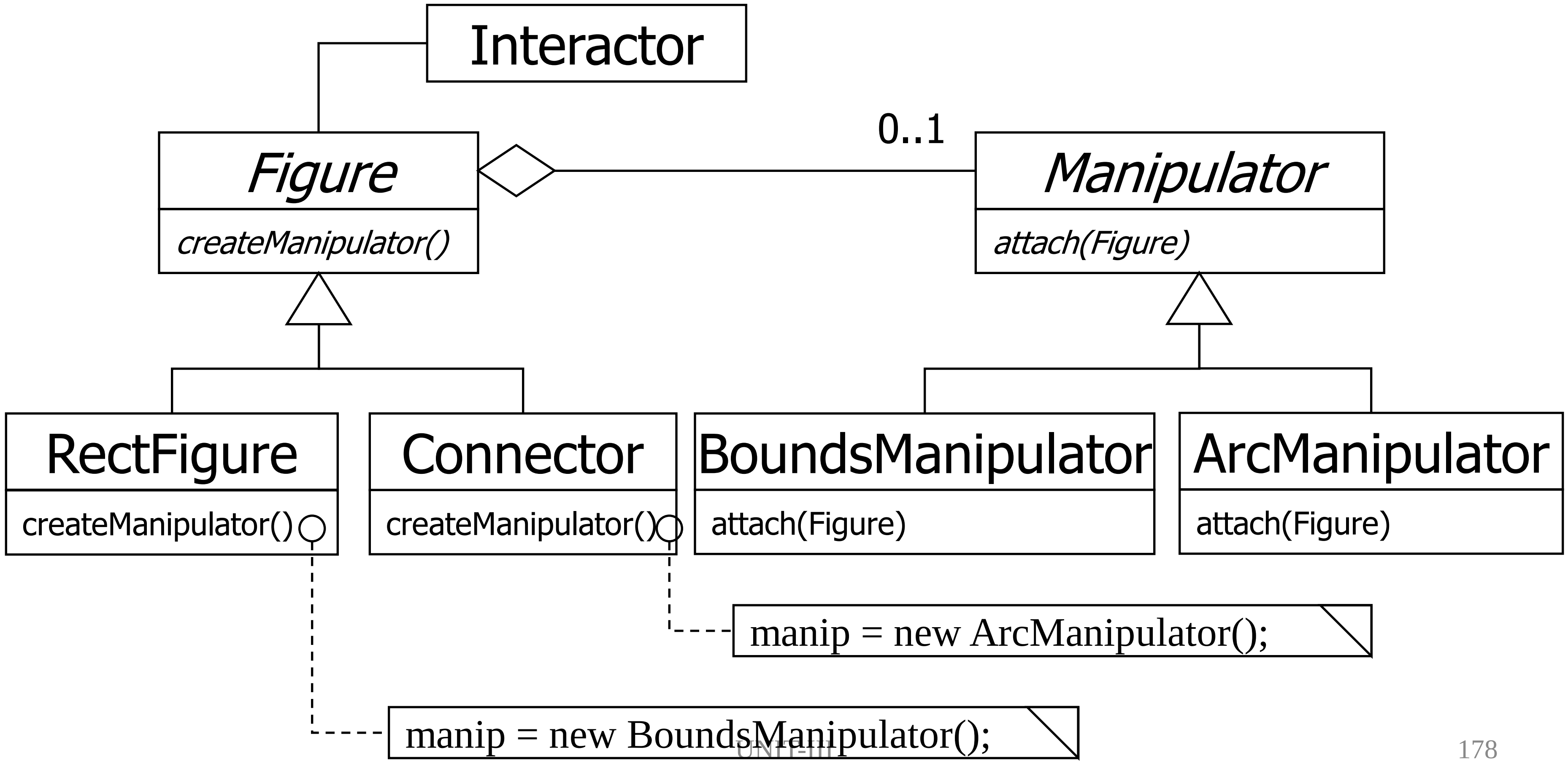
# Factory Method

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- A related pattern is AbstractFactory



# Factory Method (2)

- Example: creating manipulators on connectors



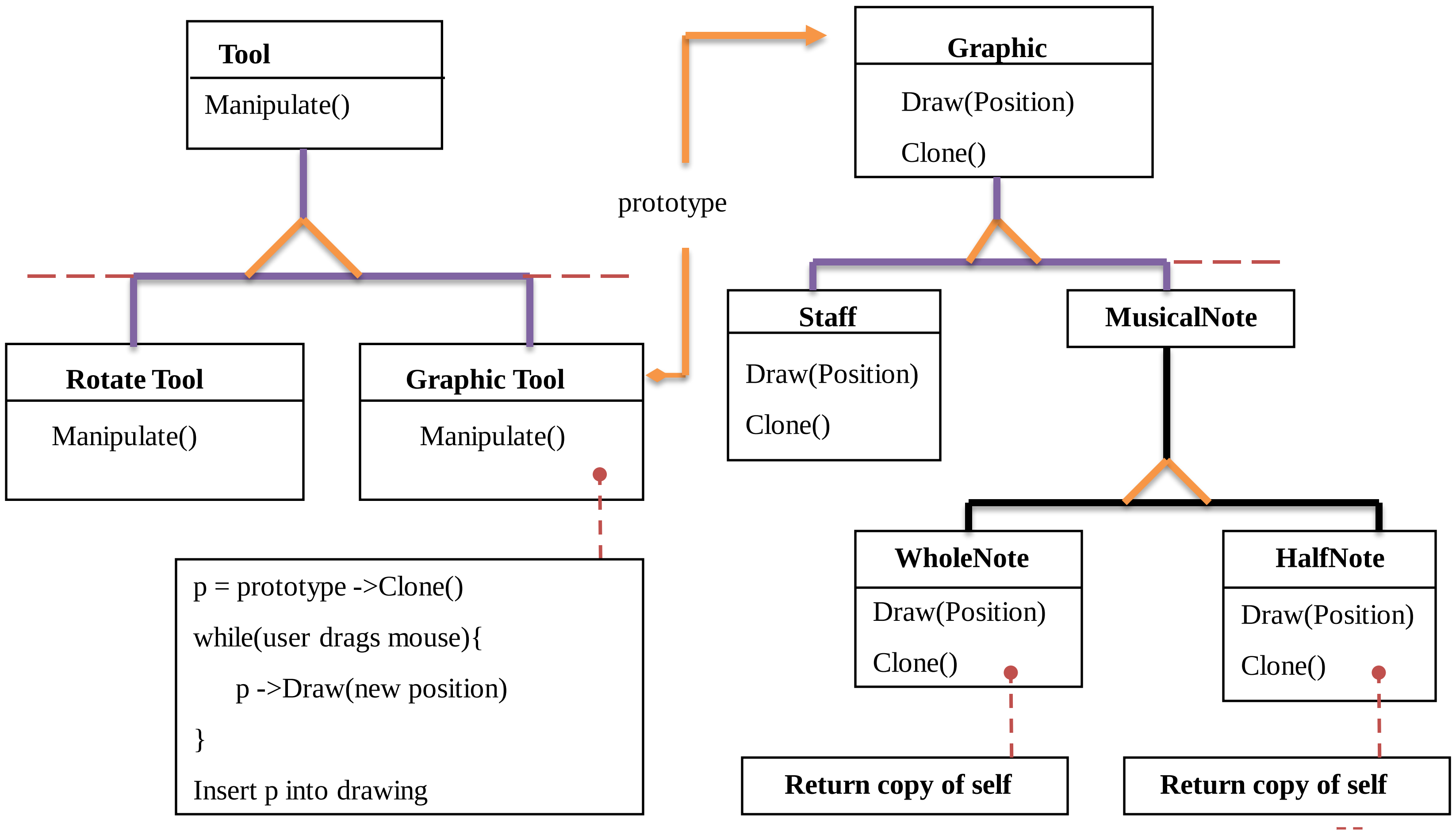
# PROTOTYPE (Object Creational)

- Intent:
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Motivation:
  - Framework implements Graphic class for graphical components and GraphicTool class for tools manipulating/creating those components

## Motivation

– Actual graphical components are application-specific
– How to parameterize instances of Graphic Tool class with type of objects to create?
– Solution: create new objects in Graphic Tool by cloning a **prototype** object instance

# PROTOTYPE Motivation



```
p = prototype ->Clone()
while(user drags mouse){
    p ->Draw(new position)
}
Insert p into drawing
```

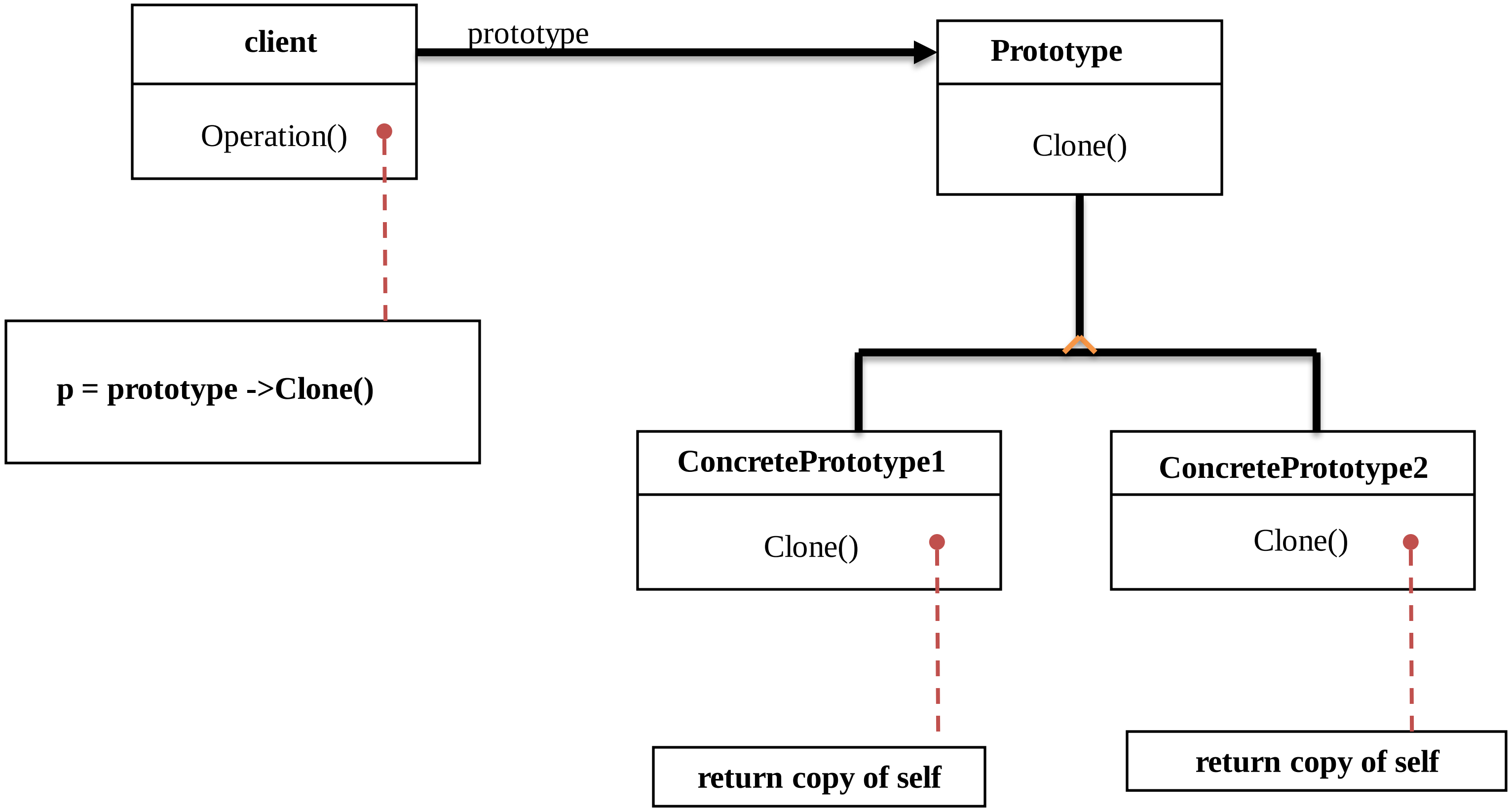
## Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

## Applicability

– when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

# PROTOTYPE
## Structure



## Participants:

- Prototype (Graphic)
  - Declares an interface for cloning itself
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a prototype to clone itself

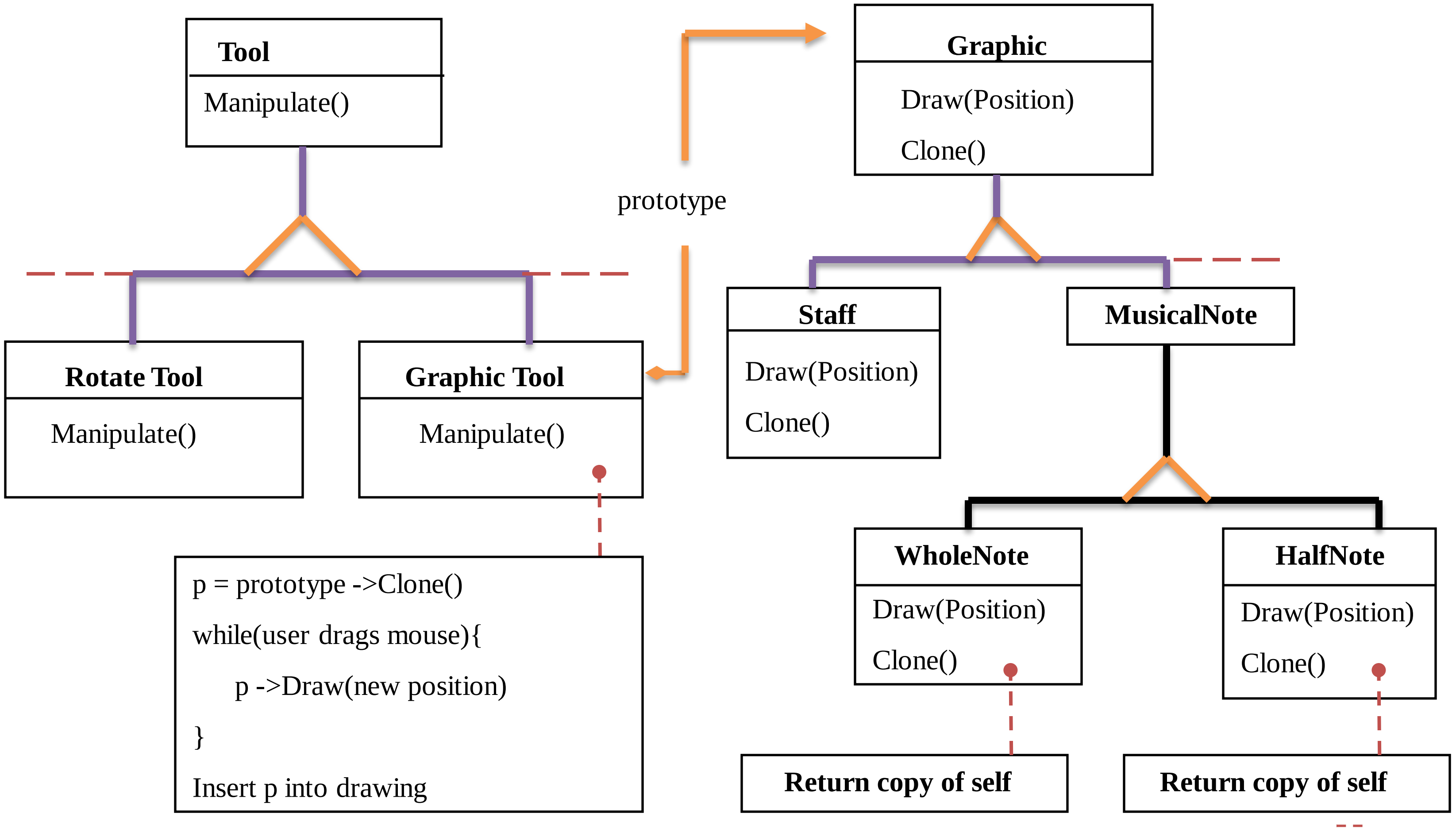
## Collaborations:

- A client asks a prototype to clone Itself.

## Motivation

- Actual graphical components are application-specific
- How to parameterize instances of Graphic Tool class with type of objects to create?
- Solution: create new objects in Graphic Tool by cloning a **prototype** object instance

# PROTOTYPE Motivation



```
p = prototype ->Clone()
while(user drags mouse){
    p ->Draw(new position)
}
Insert p into drawing
```

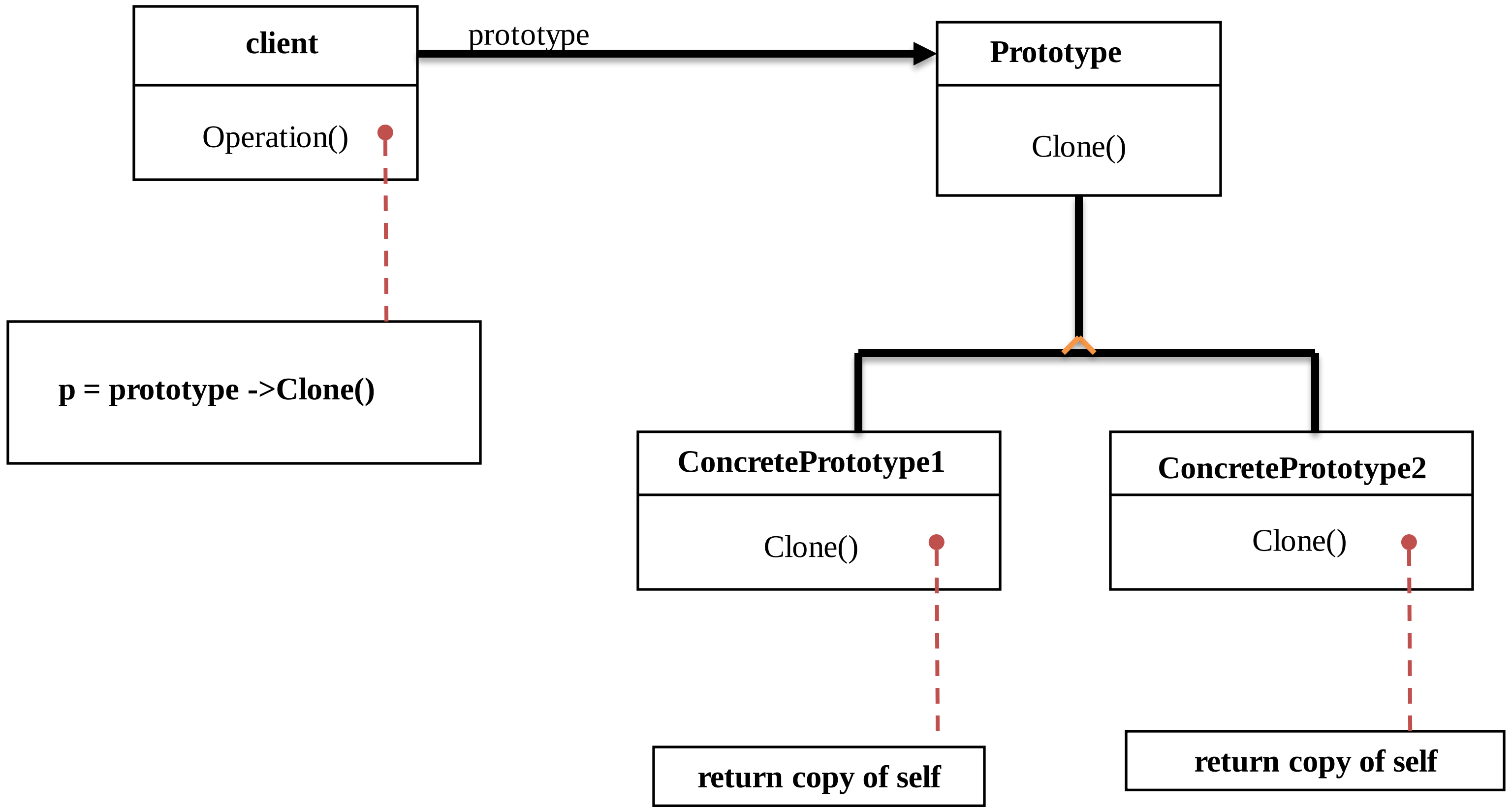
## Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

## Applicability

– when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

## PROTOTYPE Structure



## Participants:

- Prototype (Graphic)
  - Declares an interface for cloning itself
- Concrete Prototype (Staff, Whole Note, Half Note)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a prototype to clone itself

## Collaborations:

- A client asks a prototype to clone Itself.

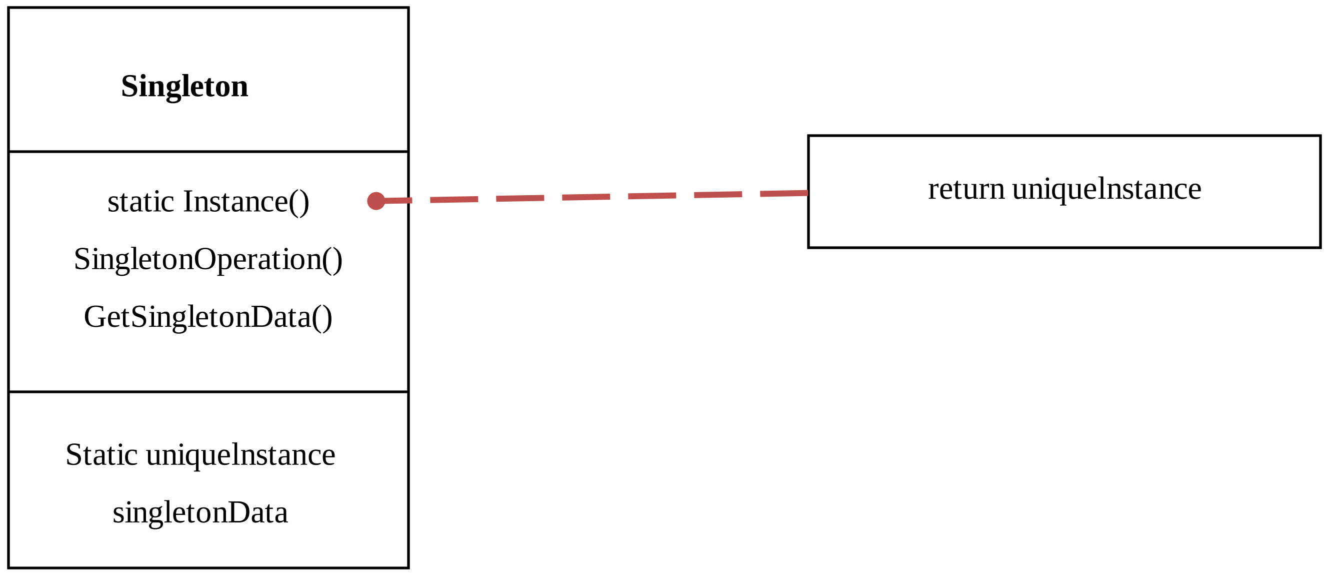
## SINGELTON

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation:
  - Some classes should have exactly one instance (one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but doesn't prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface

## Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a 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.

# SINGLETON
## Structure



## Participants and Collaborations

- Singleton: Defines an instance operation that lets clients access its unique interface
- Instance is a class operation (static in Java)
- May be responsible for creating its own unique instance
- Collaborations: Clients access a Singleton instance solely through Singleton's Instance operation.

## Singleton

- Ensures a class has only one instance
- Provides a single point of reference

## Singleton – Use When

- There must be exactly one instance of a class.
- May provide synchronous access to avoid deadlocks.
- Very common in GUI toolkits, to specify the connection to the OS/Windowing system

## Singleton - Benefits

- Controls access to a scarce or unique resource
- Helps avoid a central application class with various global object references
- Subclasses can have different implementations as required. Static or global references don’t allow this
- Multiple or single instances can be allowed

## Singleton – Example 1

- An Application class, where instantiating it makes a connection to the base operating system and sets up the rest of the toolkit’s framework for the user interface.

- In the Qt toolkit:

```
QApplication* app = new QApplication(argc, argv)
```

## Singleton – Example 2

- A status bar is required for the application, and various application pieces need to be able to update the text to display information to the user. However, there is only one status bar, and the interface to it should be limited. It could be implemented as a Singleton object, allowing only one instance and a focal point for updates. This would allow updates to be queued, and prevent messages from being overwritten too quickly for the user to read them.

# Singleton Code [1]

```
class Singleton {                    // Only one instance can ever be created.

   public:

              static Singleton* Instance();

    protected:                         // Creation hidden inside Instance().

               Singleton();


    private:

               Static Singleton* _instance

    }                                     // Cannot access directly.
```

## Singleton Code [2]

```
Singleton* Singleton::_instance=0;


Singleton* Singleton:: Instance(){

    if (_instance ==0) {
            _instance=new Singleton;
    }
Return _instance;

}
                              // Clients access the singleton
                              // exclusively via the Instance member
                              // function.
```

## Implementation Points

- Generally, a single instance is held by the object, and controlled by a single interface.

- Sub classing the Singleton may provide both default and overridden functionality.

# UNIT-III

# Structural patterns part-1

## Structural Pattern part-I introduction

| S. No | TOPIC | PPT Slides | |
|---|---|---|---|
| 1 | Adaptor | L1 | 3 – 18 |
| 2 | Bridge | L2 | 19 – 31 |
| 3 | Composite | L3 | 32 – 37 |
| 4 | Repeated key points for Structural Patterns | L4 | 38 – 40 |
| 5 | (Intent, Motivation, Also Known As…) | L5 | 41 – 42 |
| 6 | Code Examples | L6 | 43 – 44 |
| 7 | Reference | L7 | 45 – 45 |

# Agenda

- Intent & Motivation
- Structure
- Applicability
- Consequences
- Known Uses
- Related Patterns
- References

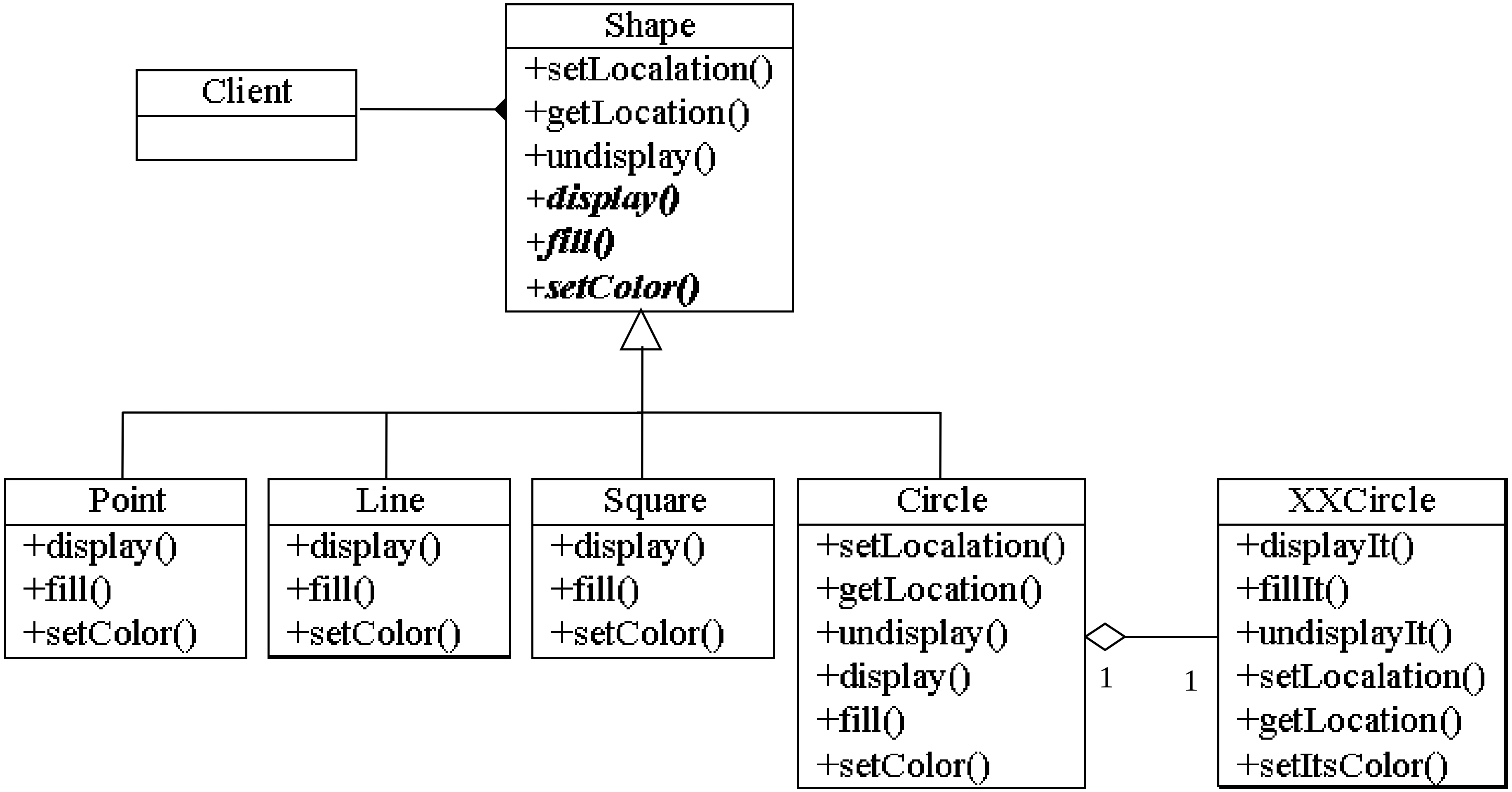
## What is Adapter?

- Intent:

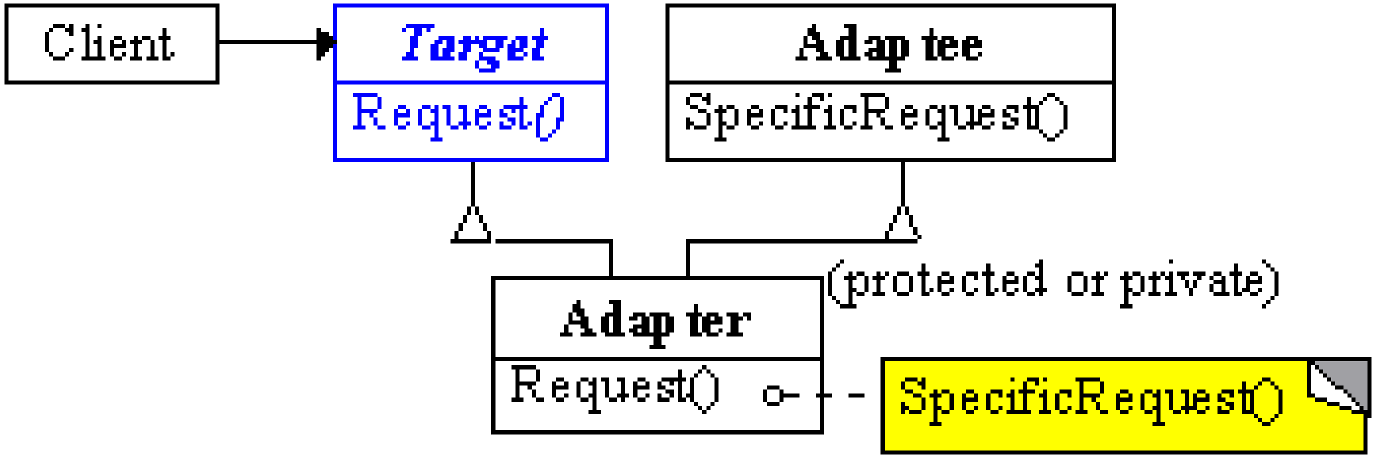
  Change the interface of a class into another interface which is expected by the client.
- Also Know As:

  Wrapper

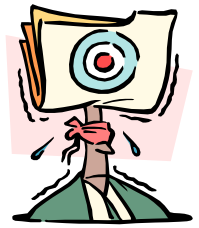
## Motivation

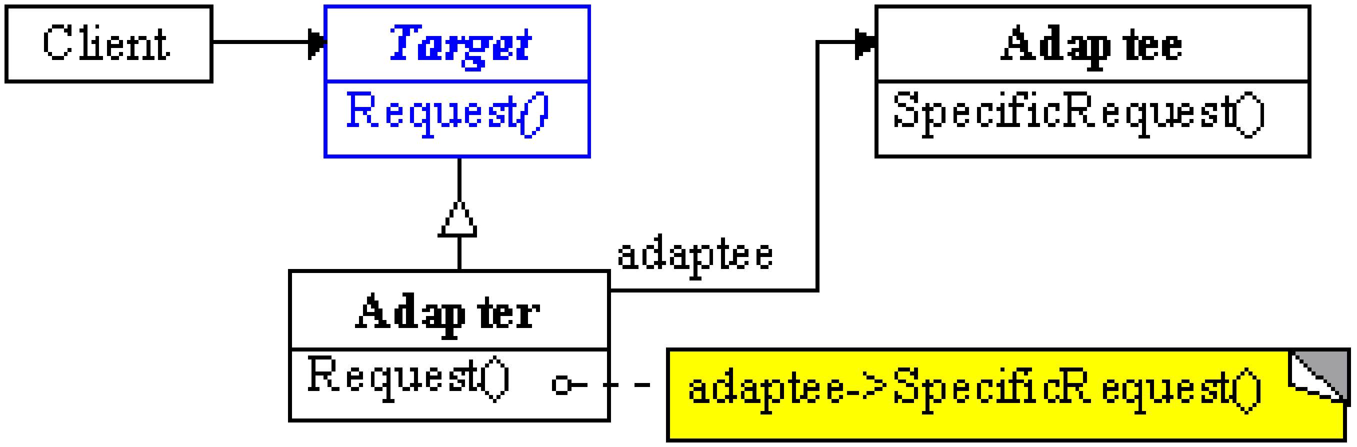

Client
Shape
+setLocalation()
+getLocation()
+undisplay()
+display()
+fill()
+setColor()
Point
+display()
+fill()
+setColor()
Line
+display()
+fill()
+setColor()
Square
+display()
+fill()
+setColor()
Circle
+setLocalation()
+getLocation()
+undisplay()
+display()
+fill()
+setColor()
1
1
XXCircle
+displayIt()
+fillIt()
+undisplayIt()
+setLocalation()
+getLocation()
+setItsColor()

## Structure (Class)



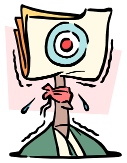
# Structure (Object)





## Applicability

- Use an existing class whose interface does not match the requirement
- Create a reusable class though the interfaces are not necessary compatible with callers

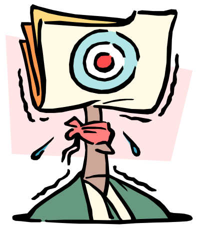


- Want to use several existing subclasses, but it is impractical to subclass everyone. (Object Adapter Only)

## Class Adapter Pattern

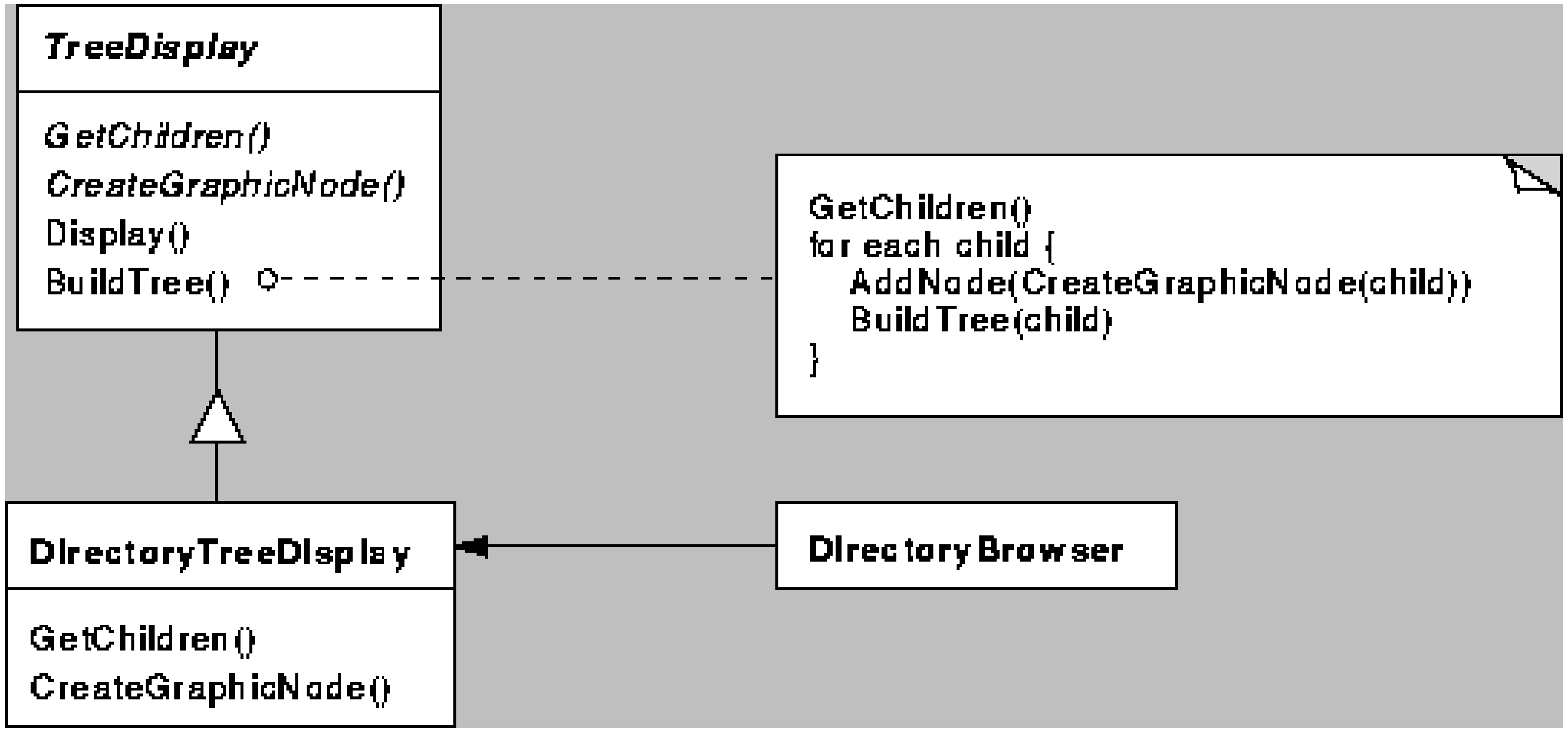
- Pros
  - Only 1 new object, no additional indirection
  - Less code required than the object Adapter
  - Can override Adaptee's behaviour as required
- Cons
  - Requires sub-classing (tough for single inheritance)
  - Less flexible than object Adapter

## Object Adapter Pattern



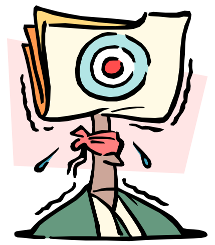
- Pros
  - More flexible than class Adapter
  - Doesn't require sub-classing to work
  - Adapter works with Adaptee and all of its subclasses
- Cons
  - Harder to override Adaptee behavior
  - Requires more code to implement properly

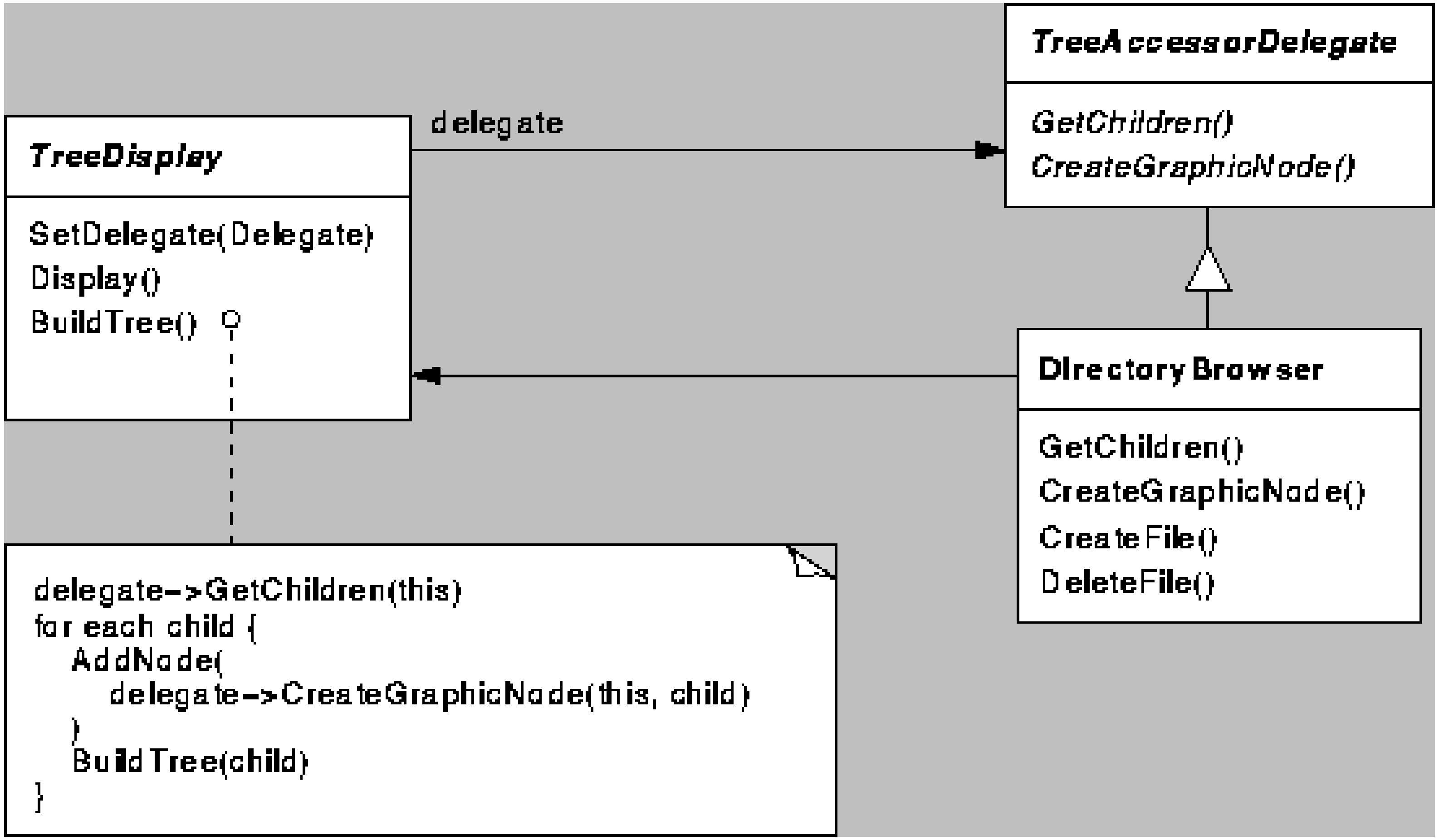
# Pluggable Adapters



- implemented with abstract operations

# Pluggable Adapters





```
delegate->GetChildren(this)
for each child {
    AddNode(
        delegate->CreateGraphicNode(this, child)
    )
    BuildTree(child)
}
```

- implemented with delegate objects

# Two-way Adapters

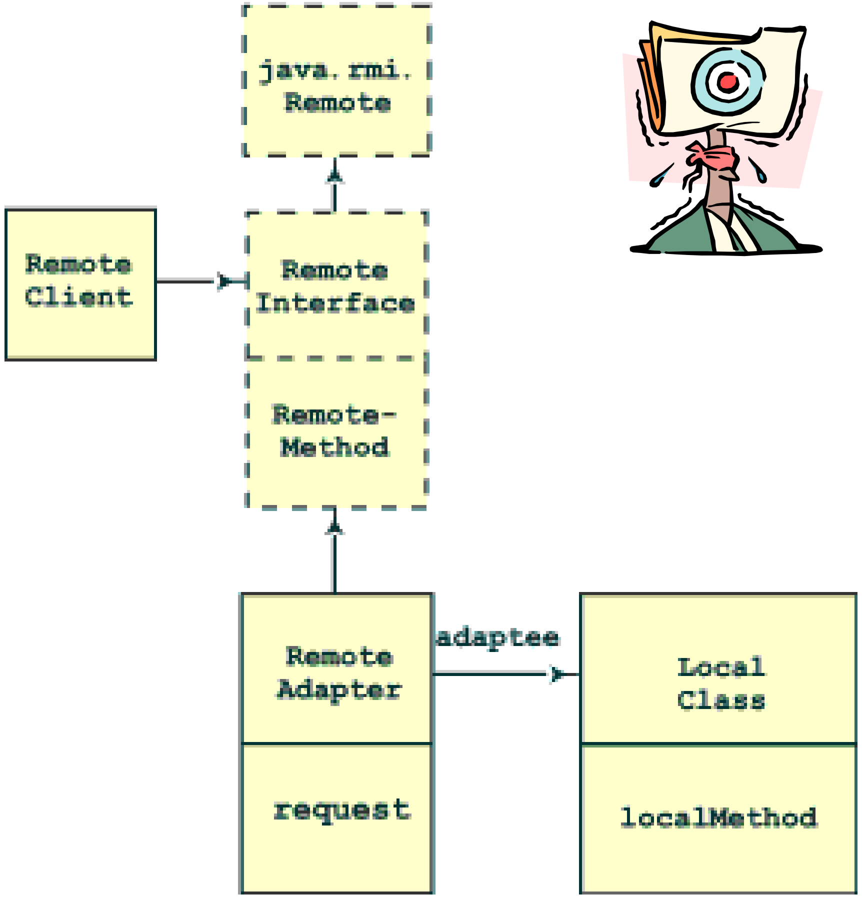
```
class SquarePeg {
   public:
      void virtual squarePegOperation() {
      blah }
}


class RoundPeg {

   public:

      void virtual roundPegOperation() { blah
      }

}
```

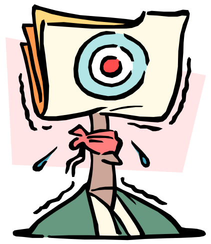
```
class PegAdapter: public SquarePeg,
     RoundPeg {

   public:

      void virtual roundPegOperation() {

         add some corners;

         squarePegOperation();

      }

      void virtual squarePegOperation() {

         add some corners;

         roundPegOperation();

      }
}
```

# Adapting Local Classes to RMI

Comparison:

- Increases reusability of local class
- Improves performance of local class
- Doesn't use Java single parent by subclassing (uses composition)





## Related Patterns

- Adapter can be similar to the remote form of Proxy. However, Proxy doesn't change interfaces.
- Decorator enhances another object without changing its interface.
- Bridge similar structure to Adapter, but different intent.  Separates interface from implementation.

## Conclusions

- Allows collaboration between classes with incompatible interfaces
- Implemented in either class-based (inheritance) or object-based (composition & delegation) manner
- Useful pattern which promotes reuse and allows integration of diverse software components

## Adapter

- You have
  - legacy code
  - current client
- Adapter changes interface of legacy code so client can use it
- Adapter fills the gap b/w two interfaces
- No changes needed for either
  - legacy code, or
  - client

## Adapter (cont.)

```
class NewTime
{
public:
int GetTime() {
        return m_oldtime.get_time() * 1000 + 8;
    }
private:
    OldTime m_oldtime;
};
```

# The Bridge Pattern

## Overview

Intent

Also Known As

Motivation

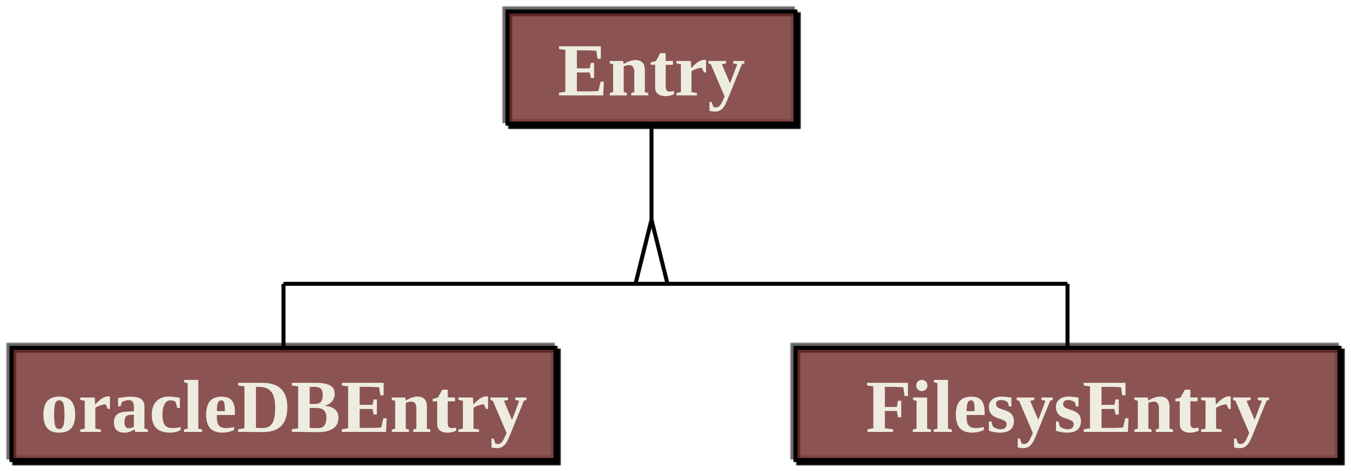
Participants

- Structure
- Applicability
- Benefits
- Drawbacks
- Related Pattern I

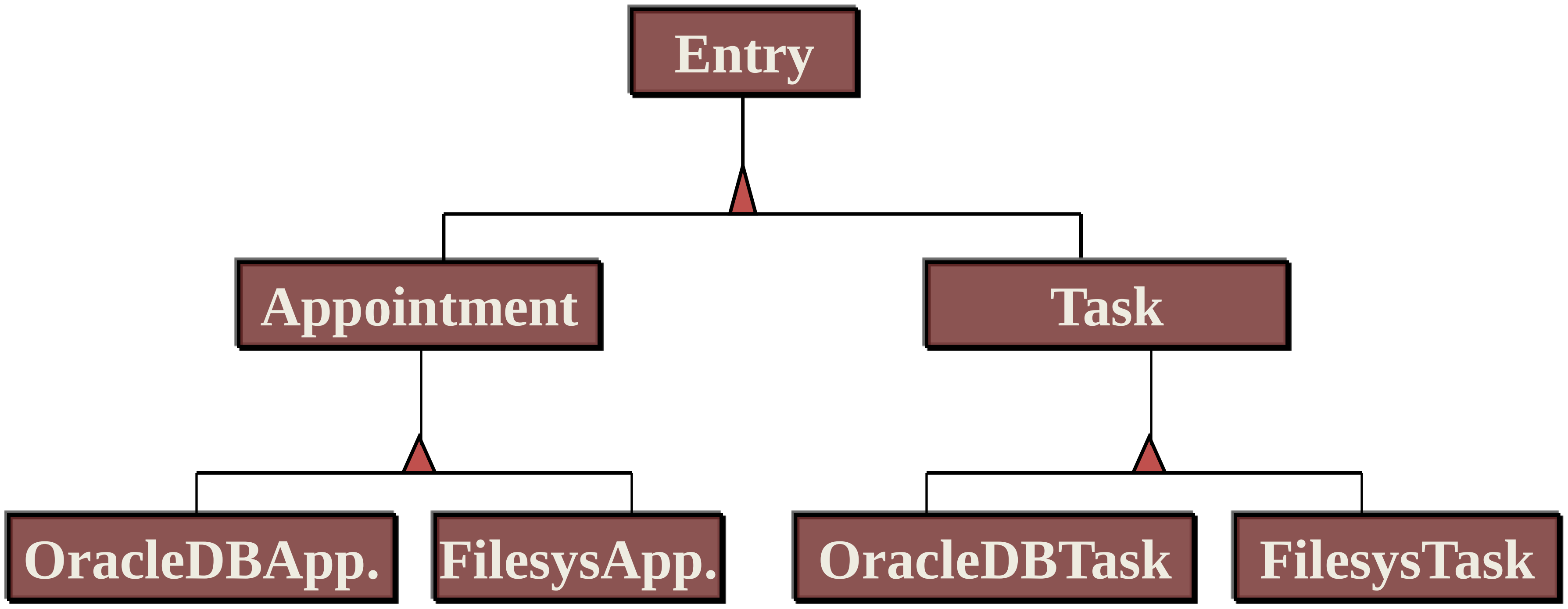
## BRIDGE (Object Structural)

- Intent: Decouple as abstraction from its implementation so that the two can vary independently.
- Also Known As: Handle/Body

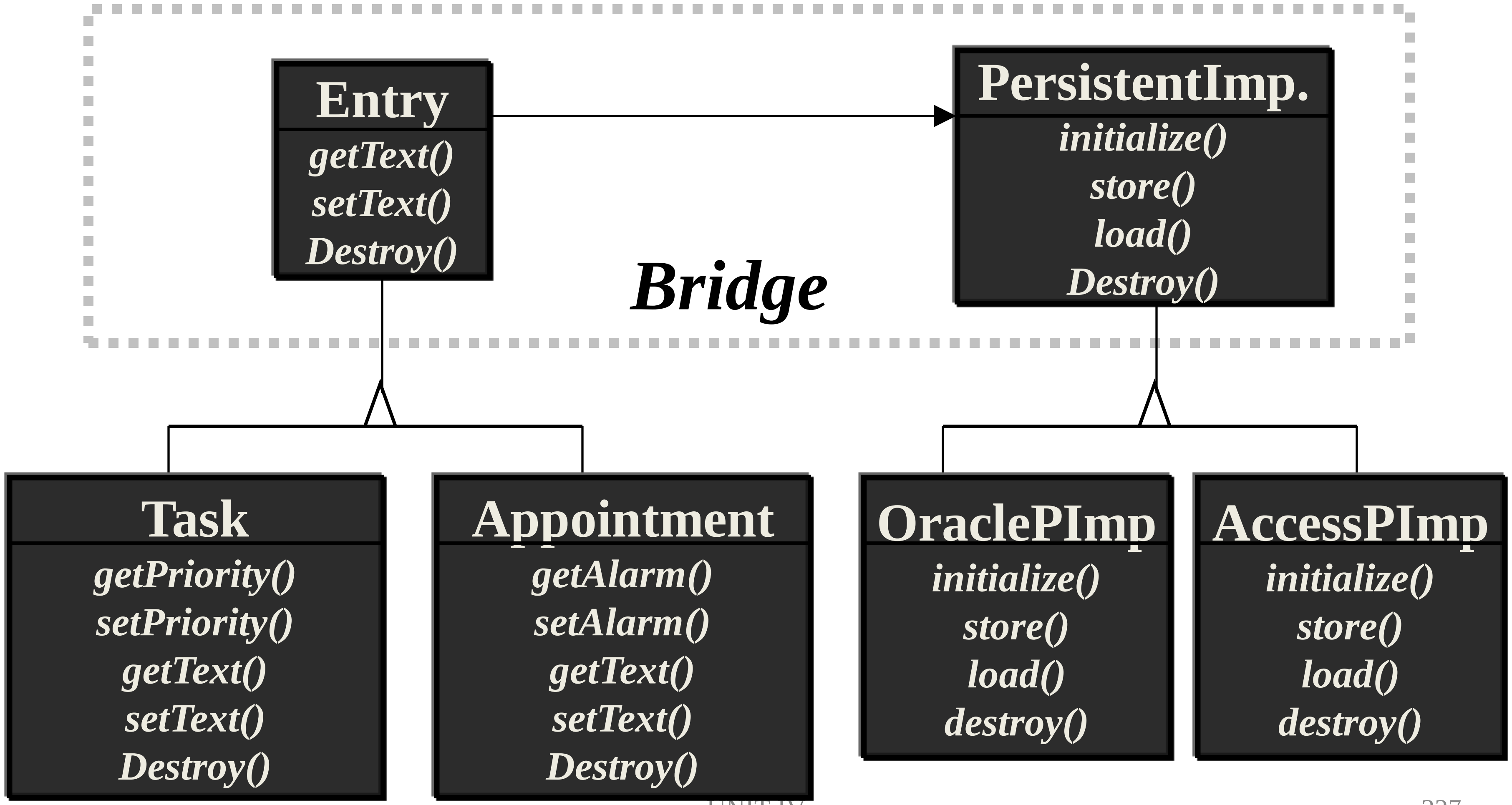
## Motivation



## Motivation


Entry
Appointment
Task
OracleDBApp.
FilesysApp.
OracleDBTask
FilesysTask

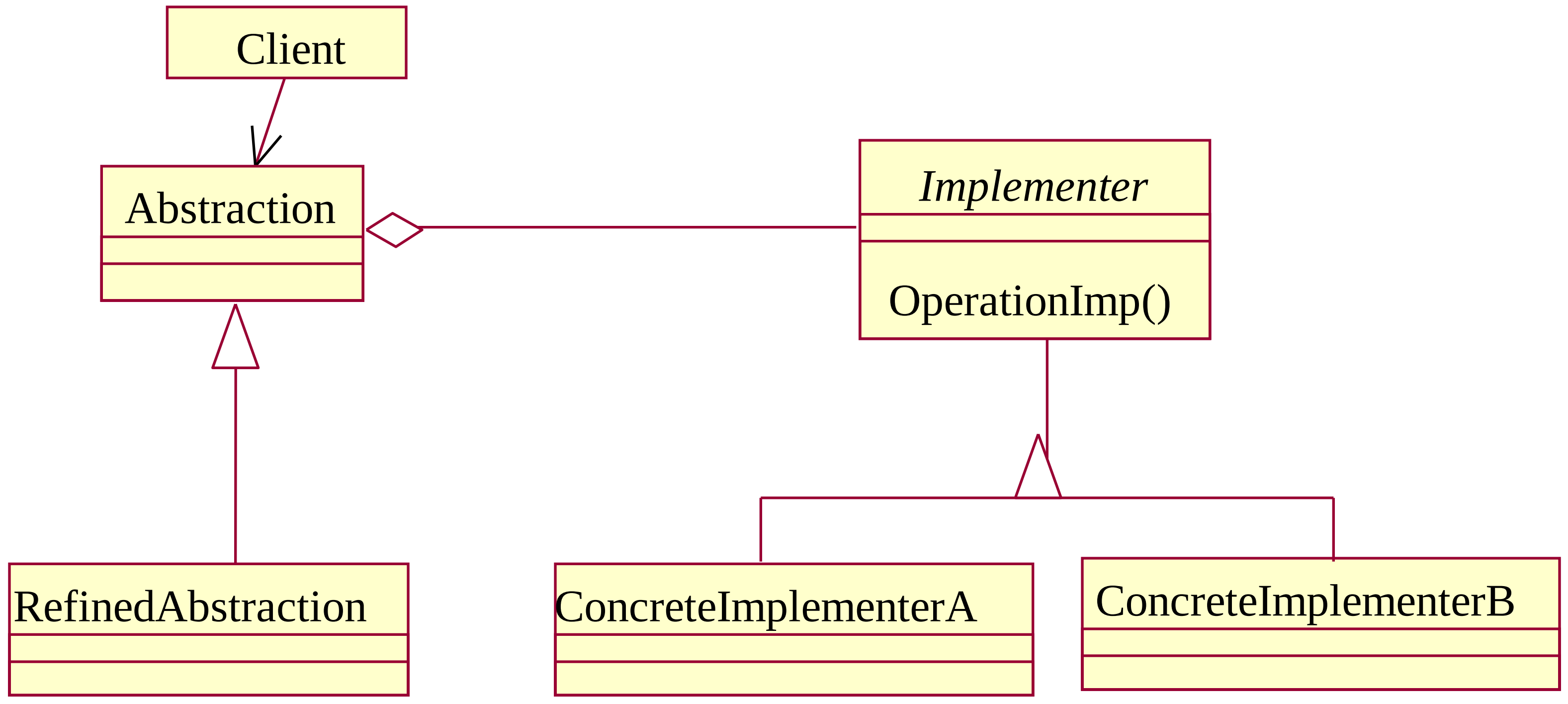
## Motivation


Entry
getText()
setText()
Destroy()
PersistentImp.
initialize()
store()
load()
Destroy()
Bridge
Task
getPriority()
setPriority()
getText()
setText()
Destroy()
Appointment
getAlarm()
setAlarm()
getText()
setText()
Destroy()
OraclePImp
initialize()
store()
load()
destroy()
AccessPImp
initialize()
store()
load()
destroy()

## Participants

## Participants (continue)

## Structure


Client
Abstraction
*Implementer*
OperationImp()
RefinedAbstraction
ConcreteImplementerA
ConcreteImplementerB

## Applicability

**Want to**

## Applicability (continue)

When the implementation should be completely hidden from the client. (C++)

When you have a proliferation of classes.

When, unknown to the client, implementations are shared among objects.

## Benefits

Avoid permanent binding between an abstraction and its implementation

Avoid nested generalizations

Ease adding new implementations

Reduce code repetition

Allow runtime switching of behavior

## Drawbacks

PersistentImp

PersistentImp

## Composite Pattern

- Intent :

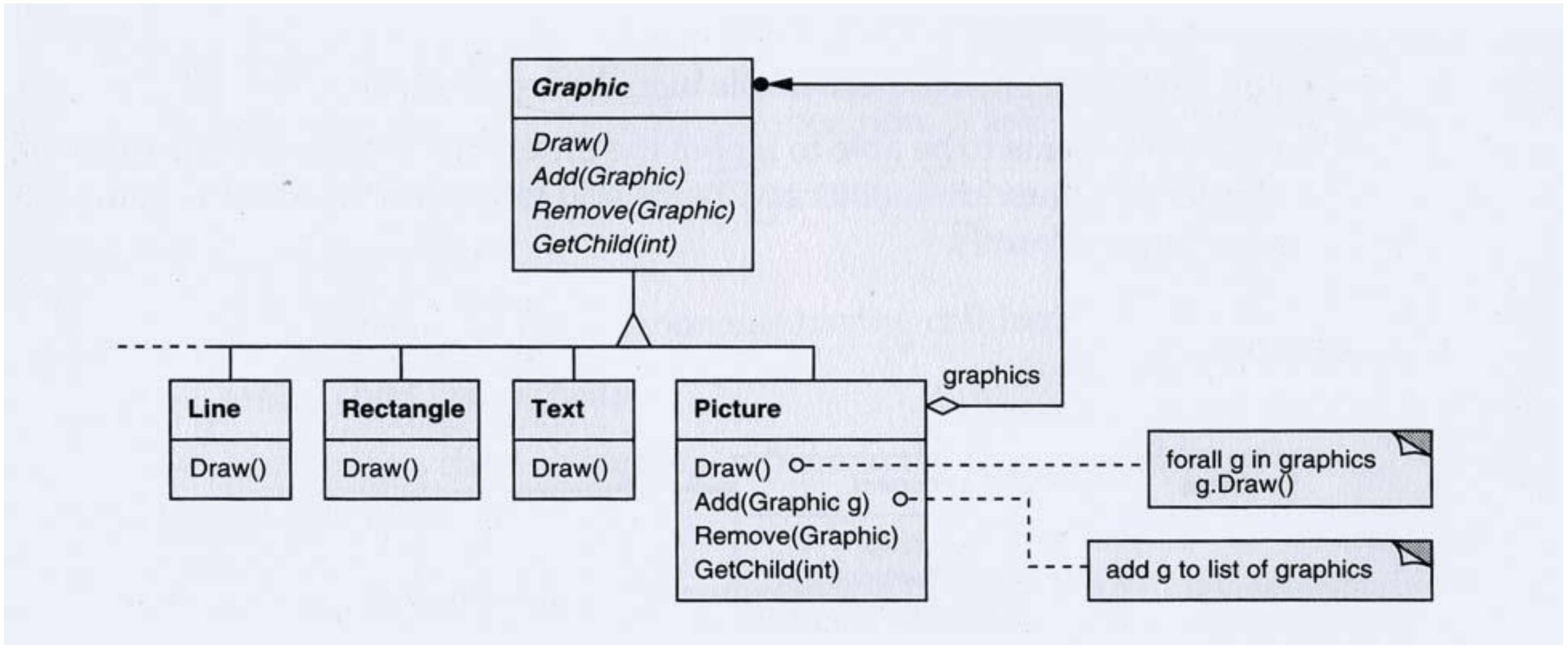
  Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

## Consequences

- **decoupling interface & implementation**
  - implementation of abstraction - can be configured at run-time
  - eliminate compile time dependencies on implementation
  - encourages layering
- **improved extensibility**
  - Abstraction & Implementer - can be extended independently
- **hiding implementation details from clients**

## Composite Pattern

- ❖Facilitates the composition of objects into tree structures that represent part-whole hierarchies.
- ❖These hierarchies consist of both primitive and composite objects.


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
graphics
forall g in graphics
g.Draw()
add g to list of graphics

## Observations

- The Component (Graphic) is an abstract class that declares the interface for the objects in the pattern. As the interface, it declares methods (such as Draw) that are specific to the graphical objects.
- Line, Rectangle, and Text are so-called Leafs, which are subclasses that implement Draw to draw lines, rectangles, and text, respectively.

## Observations (Continued)

- The Picture class represents a number of graphics objects. It can call Draw on its children and also uses children to compose pictures using primitive objects.

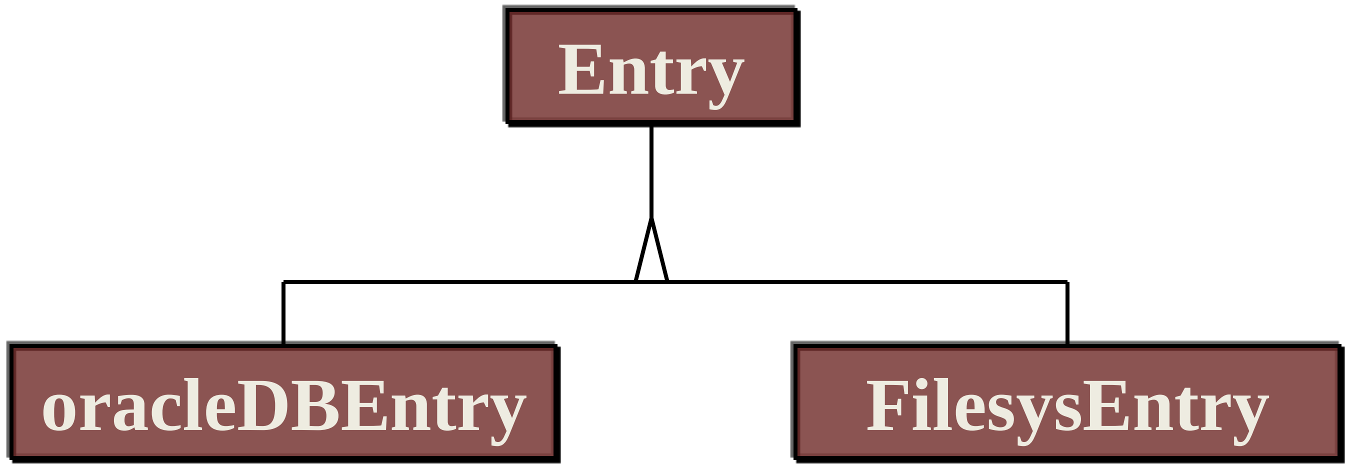
## Concluding Considerations

- The Composite Pattern is used to represent part-whole object hierarchies.
- Clients interact with objects through the component class.
- It enables clients to to ignore the specifics of which leaf or composite class they use.
- Can be used recursively, so that Display can show both flares and stars.
- New components can easily be added to a design.

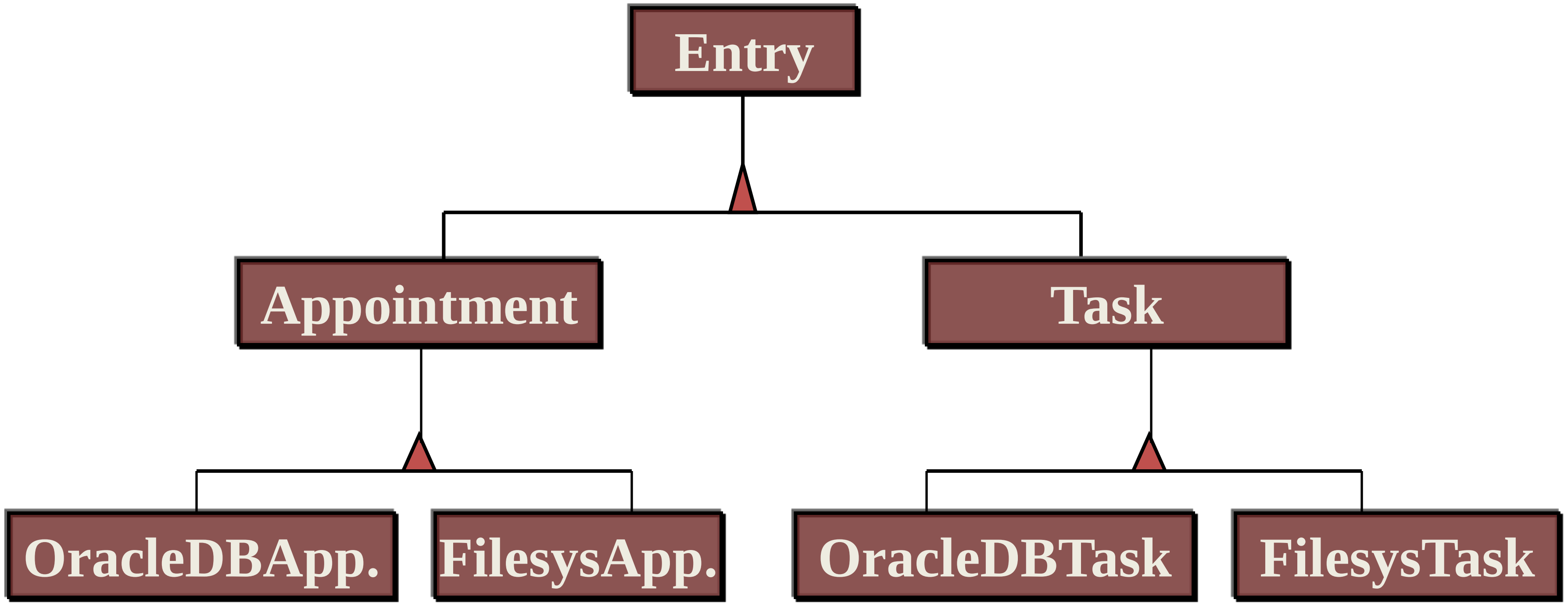
## BRIDGE (Object Structural)

- Intent: Decouple as abstraction from its implementation so that the two can vary independently.
- Also Known As: Handle/Body

## Motivation



## Motivation


Entry
Appointment
Task
OracleDBApp.
FilesysApp.
OracleDBTask
FilesysTask

## Two-way Adapters

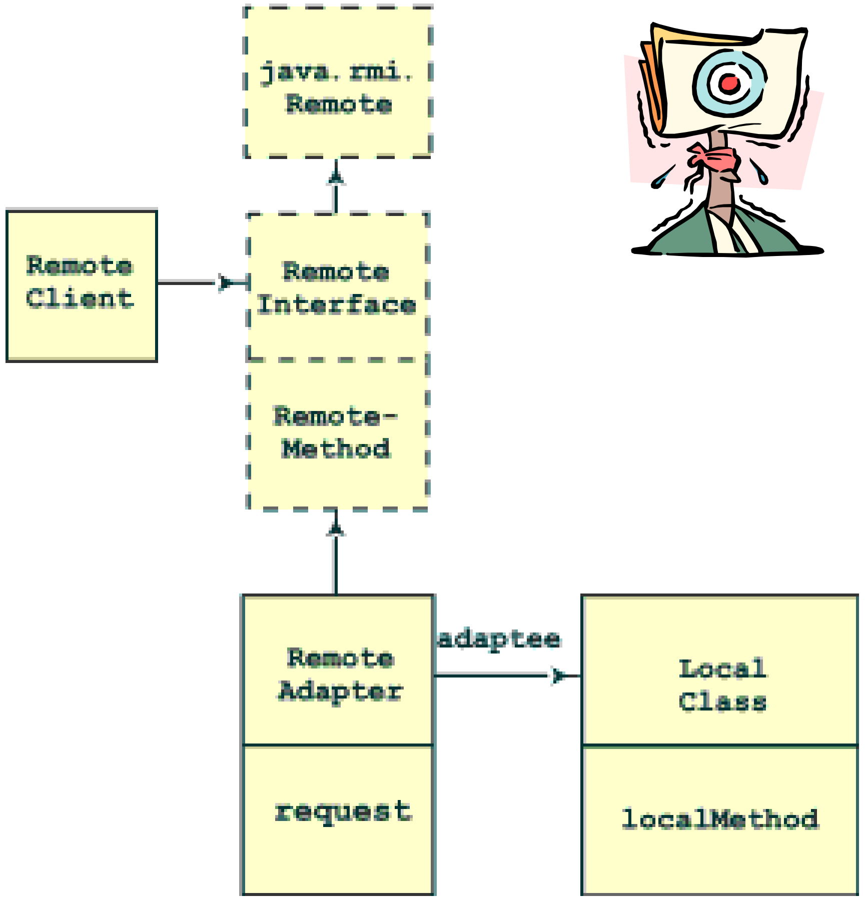
```
class SquarePeg {
   public:
      void virtual squarePegOperation() {
      blah }
}


class RoundPeg {

   public:

      void virtual roundPegOperation() { blah
      }

}
```

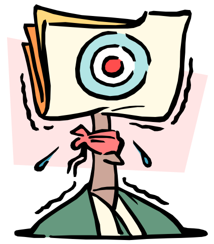
```
class PegAdapter: public SquarePeg,
     RoundPeg {

   public:

      void virtual roundPegOperation() {

         add some corners;

         squarePegOperation();

      }

      void virtual squarePegOperation() {

         add some corners;

         roundPegOperation();

      }
}
```

# Adapting Local Classes to RMI

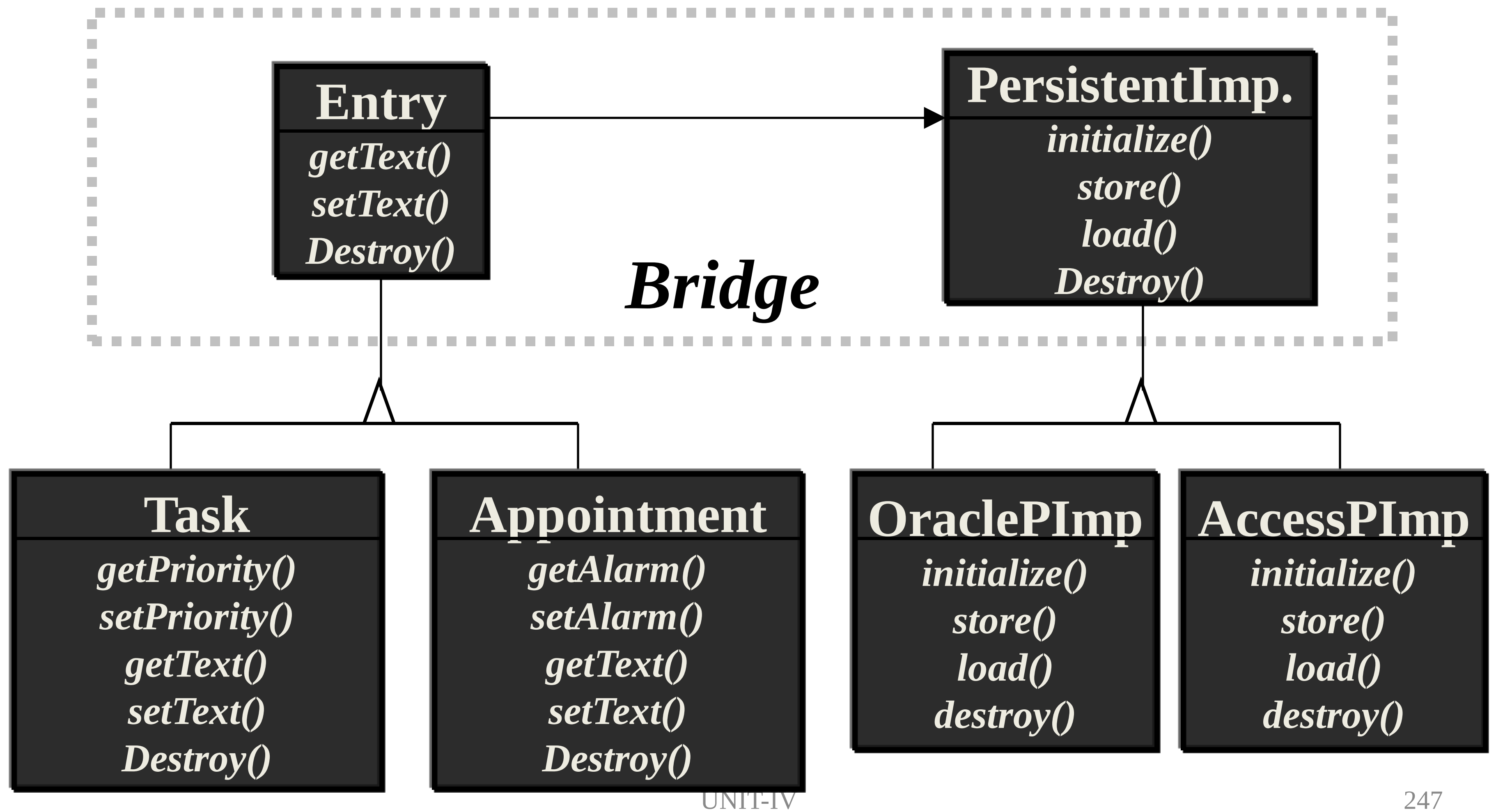
Comparison:

- Increases reusability of local class
- Improves performance of local class
- Doesn't use Java single parent by subclassing (uses composition)





# Motivation


Entry
getText()
setText()
Destroy()
PersistentImp.
initialize()
store()
load()
Destroy()
Bridge
Task
getPriority()
setPriority()
getText()
setText()
Destroy()
Appointment
getAlarm()
setAlarm()
getText()
setText()
Destroy()
OraclePImp
initialize()
store()
load()
destroy()
AccessPImp
initialize()
store()
load()
destroy()

## Participants

# References

- Becker, Dan. Design networked applications in RMI using the Adapter design pattern. JavaWorld Magazine, May 1999. http://www.javaworld.com/javaworld/jw-05-1999/jw-05-networked.html
- Buschmann et al. A System of Patterns: Pattern-Oriented Software Architecture. John Wiley and Sons. Chichester. 1996
- Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Boston. 1995
- Nguyen, D.X. Tutorial 10: Stacks and Queues: The Adapter Pattern. Rice University. 1999. http://www.owlnet.rice.edu/~comp212/99-fall/tutorials/10/tutorial10.html
- Whitney, Roger. CS 635 Advanced Object-Oriented Design & Programming. San Diego State University. 2001. http://www.eli.sdsu.edu/courses/spring01/cs635/notes/proxy/proxy.html#Heading10
- Shalloway, Alan., and Trott, James R., Design Patterns Explained: A New Perspective on Object-Oriented Design, Addison-Wesley, 2002.
- Rising, Linda., The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge university Press, 1998.

# Unit-3 part-2
# Structural Design Patterns-2

# Objectives

| S. No | Topic | | PPT Slides |
|---|---|---|---|
| | To introduce structural design patterns | | |
| 1. | Decorator | L1 | 3 – 20 |
| 2. | Façade | L2 | 21 – 27 |
| 3. | Proxy | L3 | 28 – 40 |
| 4. | Flyweight | L4 | 41 – 47 |
| 5. | Repeated key points for Structural Patterns | L5 | 48 – 52 |
| 6. | (Intent, Motivation, Also Known As ……………) | L6 | 53 – 57 |
| 7. | Review Unit-3 part-2 | L7 | 58 – 58 |

## Decorator Design Pattern

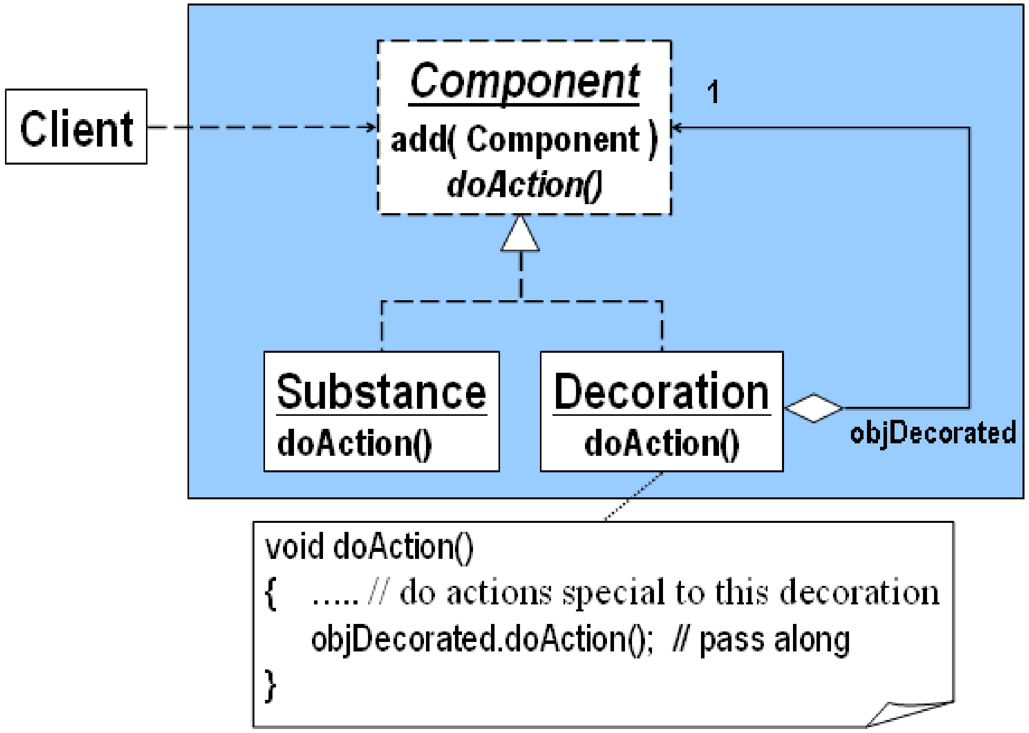
- Design Purpose

  Add responsibilities to an object at runtime.

- Design Pattern Summary

  – Provide for a linked list of objects, each encapsulating responsibility.

# Decorator: Class Model

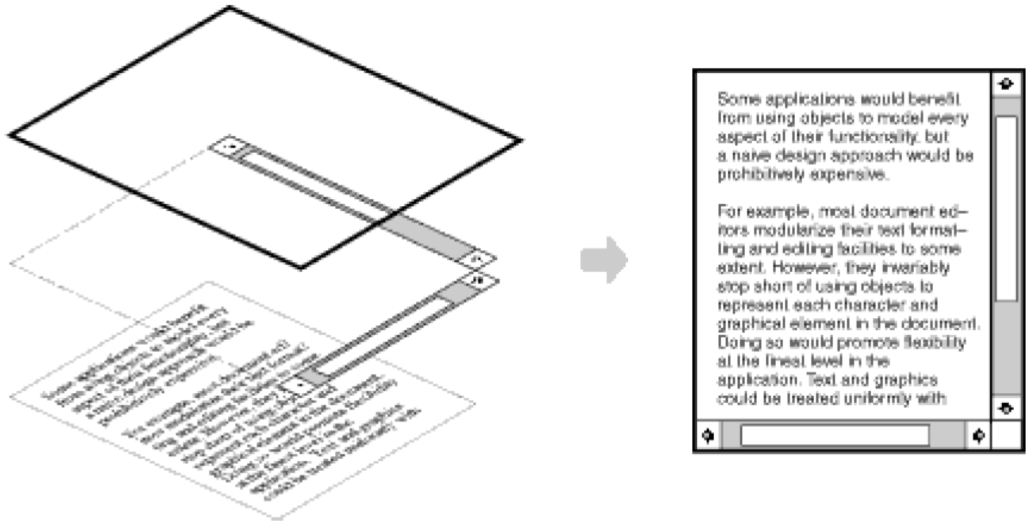


```
void doAction()
{    ..... // do actions special to this decoration
     objDecorated.doAction();  // pass along
}
```

Decorator creates an aggregated linked list of Decoration objects ending with the basic Substance object.

# Pattern: Decorator

objects that wrap around other objects to add useful features


Some applications would benefit
from using objects to model every
aspect of their functionality, but
a naive design approach would be
prohibitively expensive.

For example, most document ed–
itors modularize their text format–
ting and editing facilities to some
extent. However, they invariably
stop short of using objects to
represent each character and
graphical element in the document.
Doing so would promote flexibility
at the finest level in the
application. Text and graphics
could be treated uniformly with

## Decorator pattern

- **decorator**: an object that modifies behavior of, or adds features to, another object
  - decorator must maintain the common interface of the object it wraps up
- used so that we can add features to an existing simple object without needing to disrupt the interface that client code expects when using the simple object

- examples in Java:
  - multilayered input streams adding useful I/O methods
  - adding designs, scroll bars and borders to GUI controls

# Decorator example: I/O

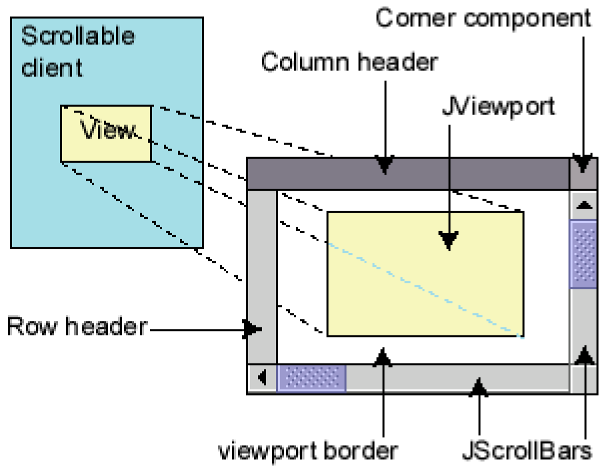
- normal InputStream class has only public int read() method to read one letter at a time
- decorators such as BufferedReader or Scanner add additional functionality to read the stream more easily

```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// because of decorator streams, I can read an
// entire line from the file in one call
// (InputStream only provides public int read() )

String wholeLine = br.readLine();
```

# Decorator example: GUI

- normal GUI components don't have scroll bars
- JScrollPane is a container with scroll bars to which you can add any component to make it scrollable

```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
  new JScrollPane(area);
contentPane.add(scrollPane);
```



## An example: Decorator

Intent:

Allow to attach responsibilities to objects, extend their functionality, dynamically

Motivation:

Consider a WYSIWYG editor

The basic textWindow needs various decorations: borders, tool bars, scroll bars, …

## An “obvious” solution: sub-classing

- A sub-class for every option

Disadvantages:

- Sub-classing is static – compile-time
- # of sub-classes = # of combinations – exponential

## The preferred solution: Use decorator objects

- Each decorator adds one decoration – property, behavior ➔ Linear
- Can add more than one, to obtain combinations of properties
- And, do it dynamically

## Structure, participants, collaborations:

- Component – an interface, describes the operations of a (GUI) component

- ConcreteComponent (extends Component)

  class of objects to which decorations can be added (e.g. the original TextWindow)

- Decorator (extends Component)– a class of decorator objects (there are several such classes)

## Implementation:

- Each decorator has a (private) reference to a `Component` object (`CC` or `D`)
- For each operation, the decorator performs its decoration job, & delegates `Component` operations to the `Component` object it contains
- `Decorator` extends `Component` (can accept `Component` requests)

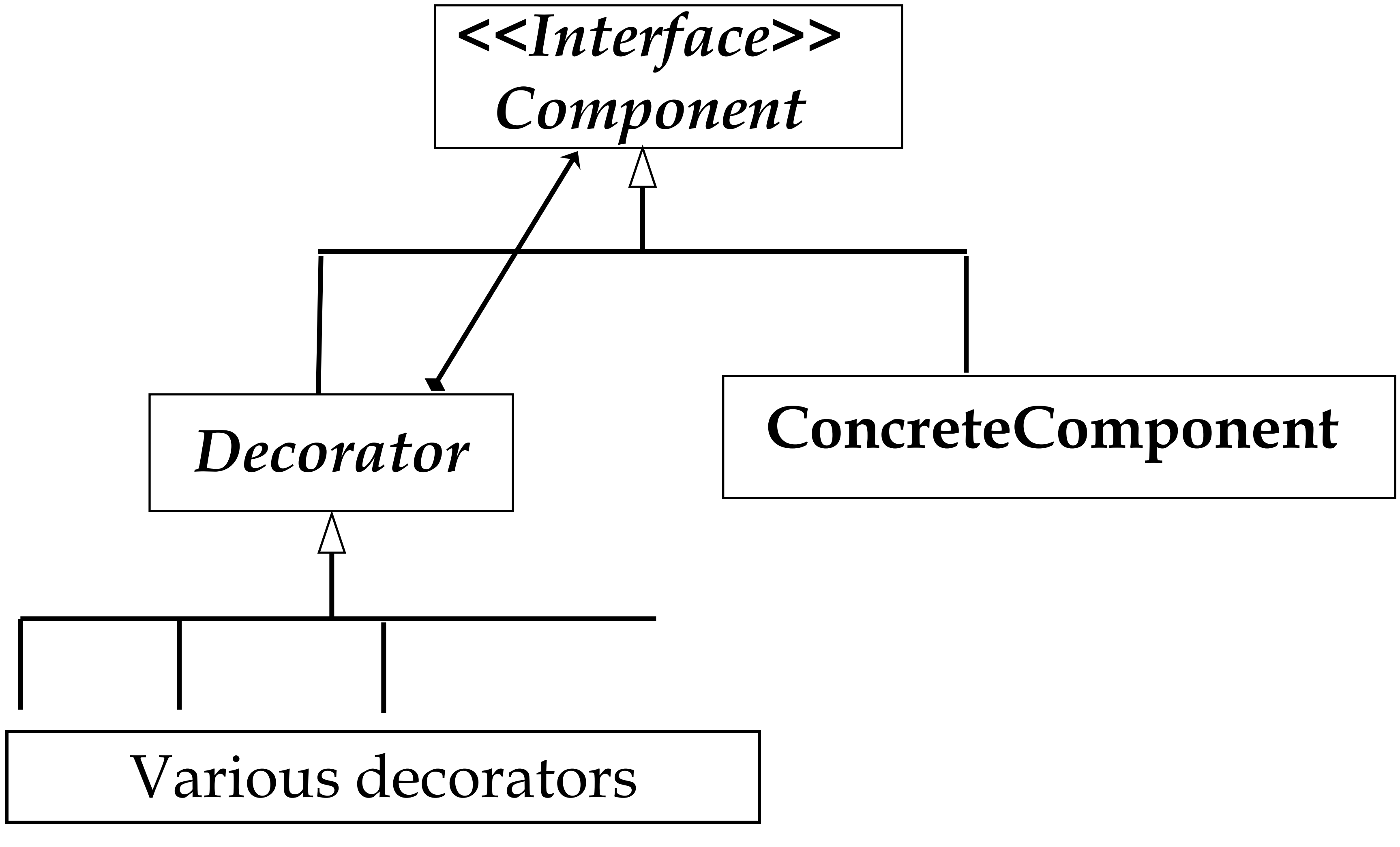
- ➔Decorators can perform all the operations of `Component`, often more

- ➔Decorators can be combined :

$$\begin{tikzcd} \boxed{D3} \arrow[r] & \boxed{D2} \arrow[r] & \boxed{D1} \arrow[r] & \boxed{CC} \end{tikzcd}$$

(but some combinations may hide some functionalities)

## The decorator pattern



## Consequences :

- Decoration can be done dynamically, at run-time
- When it makes sense, a decoration can be added several times
- The # of decorators is smaller than the # of sub-classes in original solution;

  avoids lots of feature-heavy classes in the inheritance hierarchy

## Issues to remember :

- A decorator and the Component object it contains are not identical
- Many small objects; one has to learn their roles, the legal connections, etc.

(but, still much better than sub-classing)

## Decorators in a java library:

### Stream decorators

A byte stream – r or w bytes

Decorators add:

- Buffering (does not add services)
- r or w primitive types, objects,… (adds services)

## Lessons from Decorator

More dynamic, flexible, less expensive

Composition is (often) superior to sub-classing

Program to interfaces,
**not** to concrete classes

## Facade Design Pattern

- Design Purpose

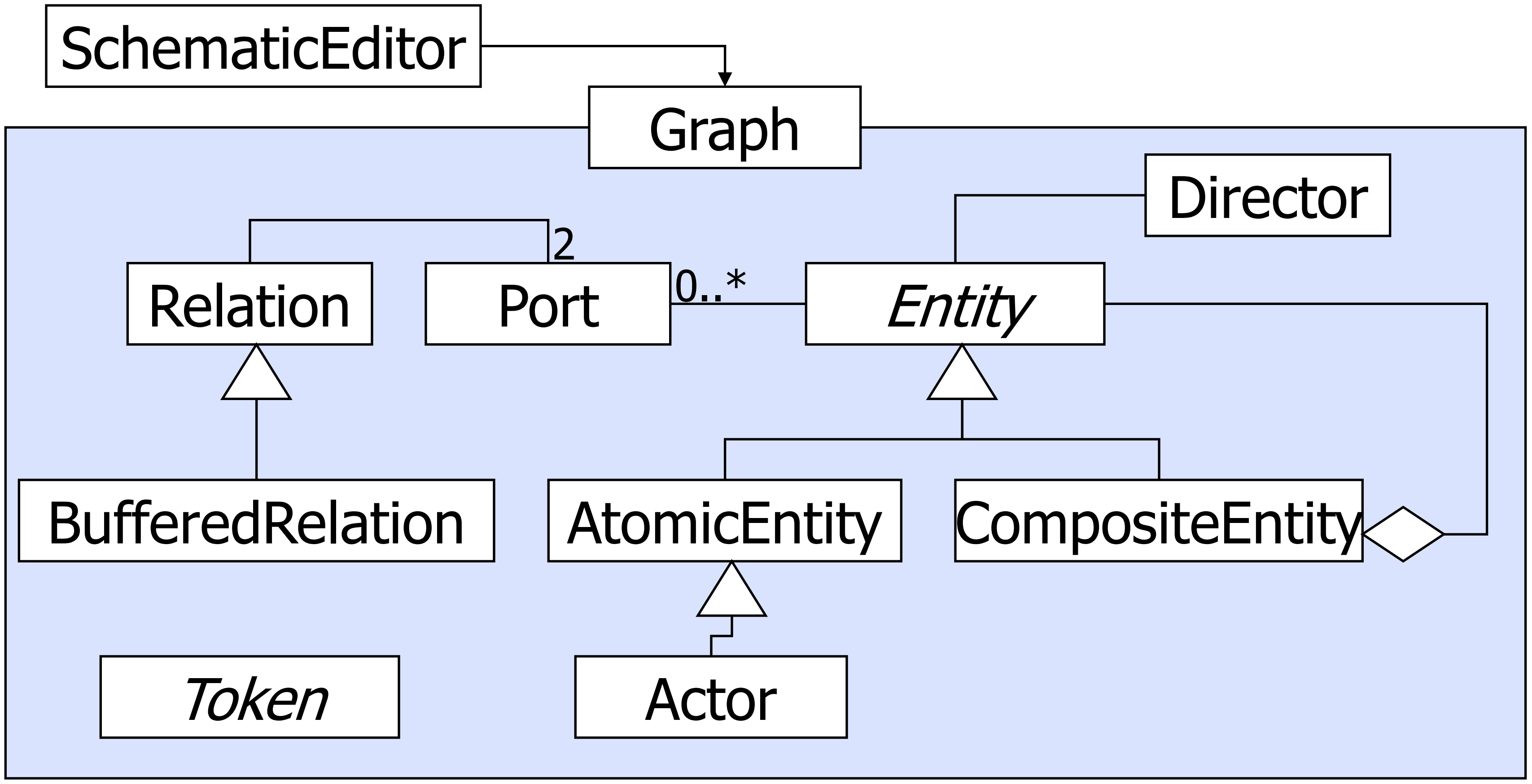
  Provide an interface to a package of classes.

- Design Pattern Summary

  – Define a singleton which is the sole means for obtaining functionality from the package.

# Façade (2)

- Example: graph interface to a simulation engine



# Facade: Encapsulating Subsystems

Name: Facade design pattern

Problem description:

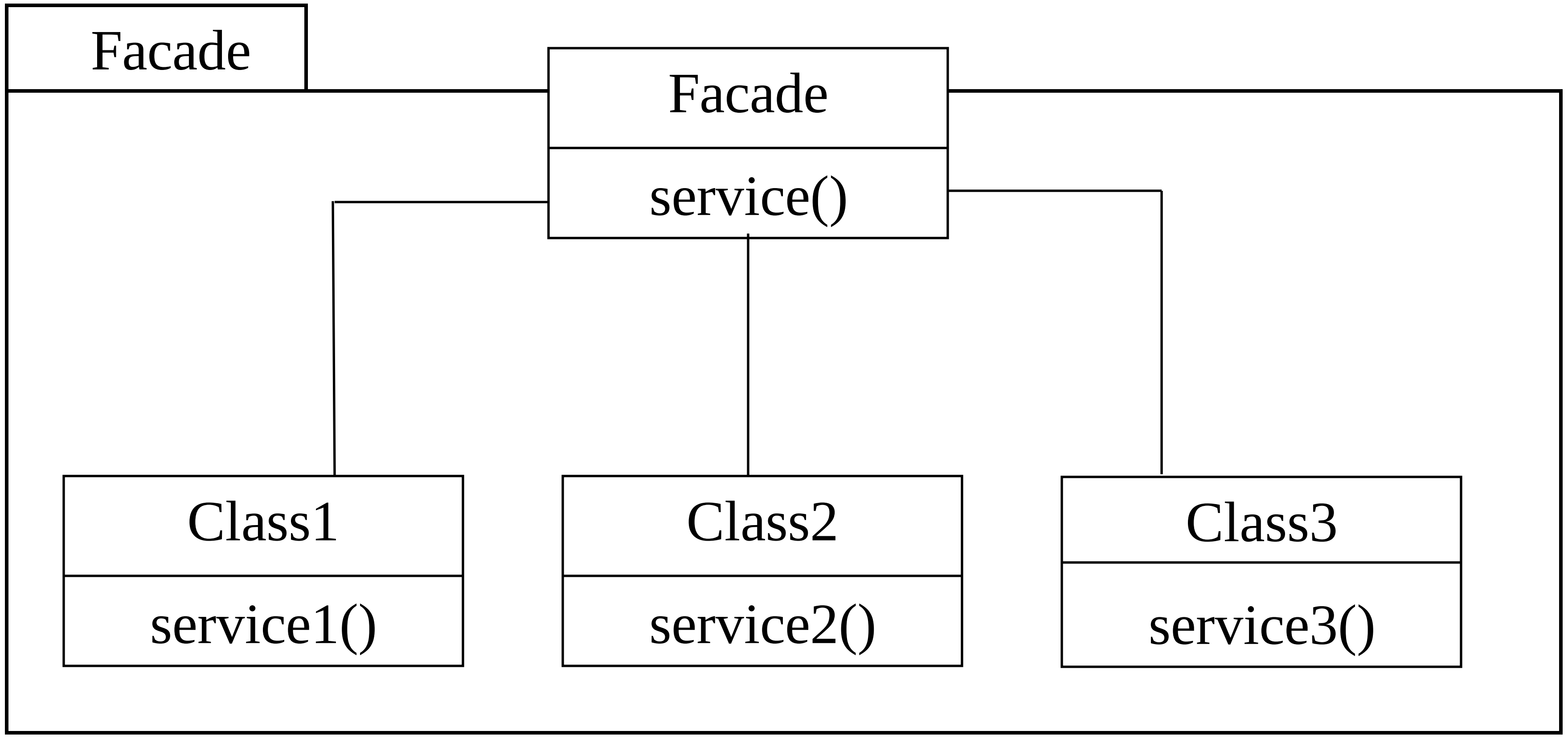
Reduce coupling between a set of related classes and the rest of the system.

Solution:

A single Facade class implements a high-level interface for a subsystem by invoking the methods of the lower-level classes.

Example. A Compiler is composed of several classes: LexicalAnalyzer, Parser, CodeGenerator, etc. A caller, invokes only the Compiler (Facade) class, which invokes the contained classes.

# Facade:
# Class Diagram



# Facade: Consequences

**Consequences:**

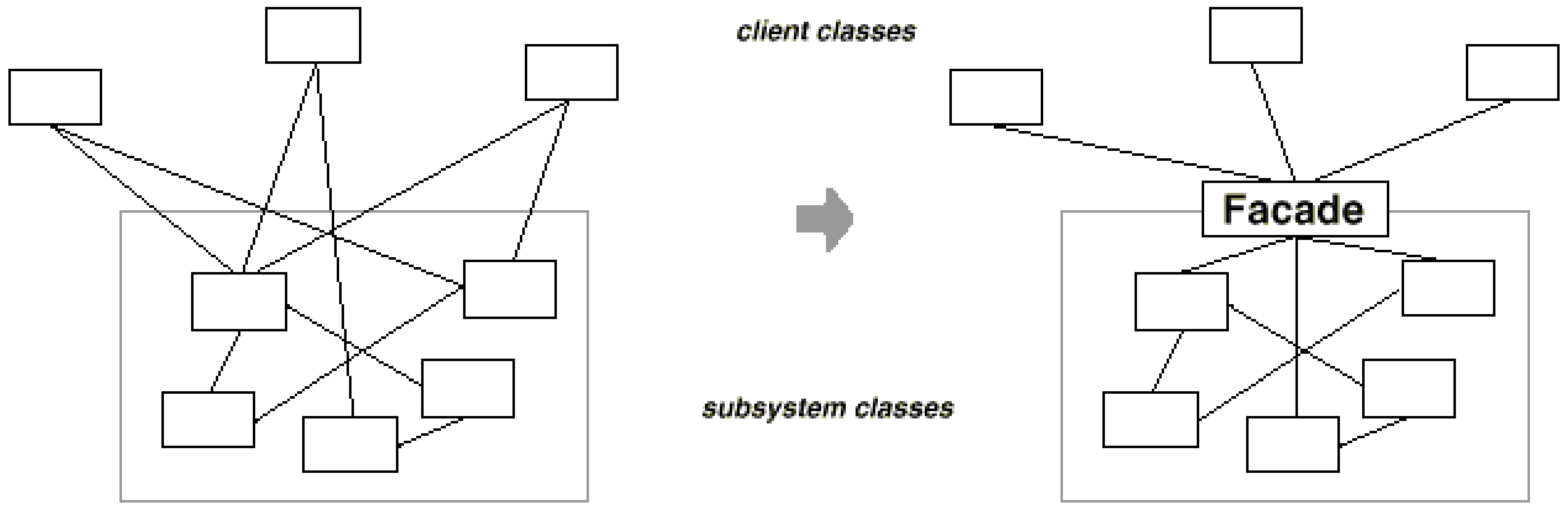
Shields a client from the low-level classes of a subsystem.

Simplifies the use of a subsystem by providing higher-level methods.

Enables lower-level classes to be restructured without changes to clients.

Note. The repeated use of Facade patterns yields a layered system.

## Facade: Motivation



- Clients communicate with the package (subsystem ) by sending requests to Facade, which forwards them to the appropriate package object(s).

## Facade: Applicability

- To provide simple interface to a complex package, which is useful for most clients.

- To reduce the dependencies between the client and the package, or dependencies between various packages.

## Facade: Consequences

- It shields clients from package components, thereby reducing the number of objects that clients deal with and making the package easier to use.
- It promotes weak coupling between the package and its clients and other packages, thereby promoting package independence and portability.

## Proxy

- You want to
  - delay expensive computations,
  - use memory only when needed, or
  - check access before loading an object into memory
- *Proxy*
  - has same interface as Real object
  - stores subset of attributes
  - does lazy evaluation

## Proxy Design Pattern

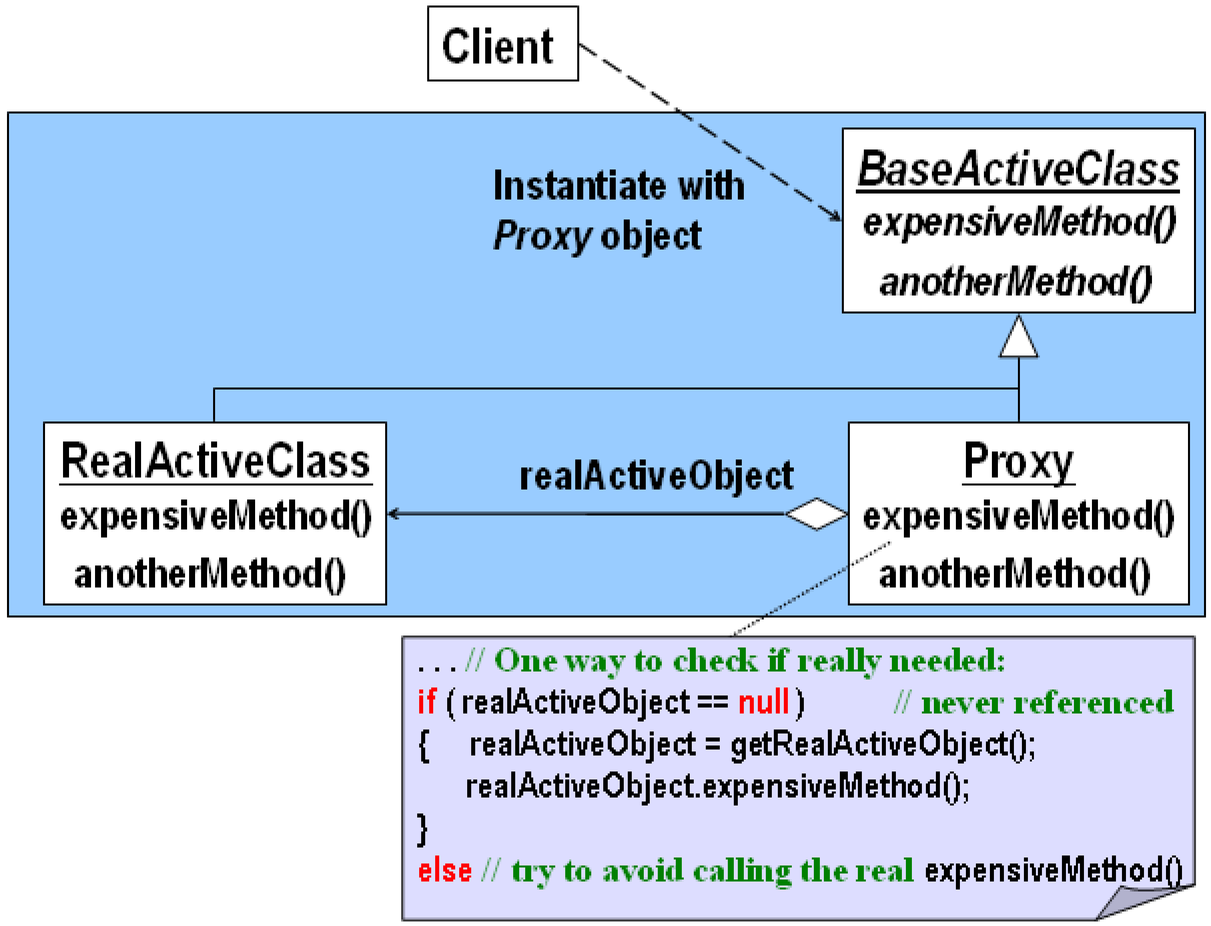
- Design Purpose

  Avoid the unnecessary execution of expensive functionality in a manner transparent to clients.

- Design Pattern Summary

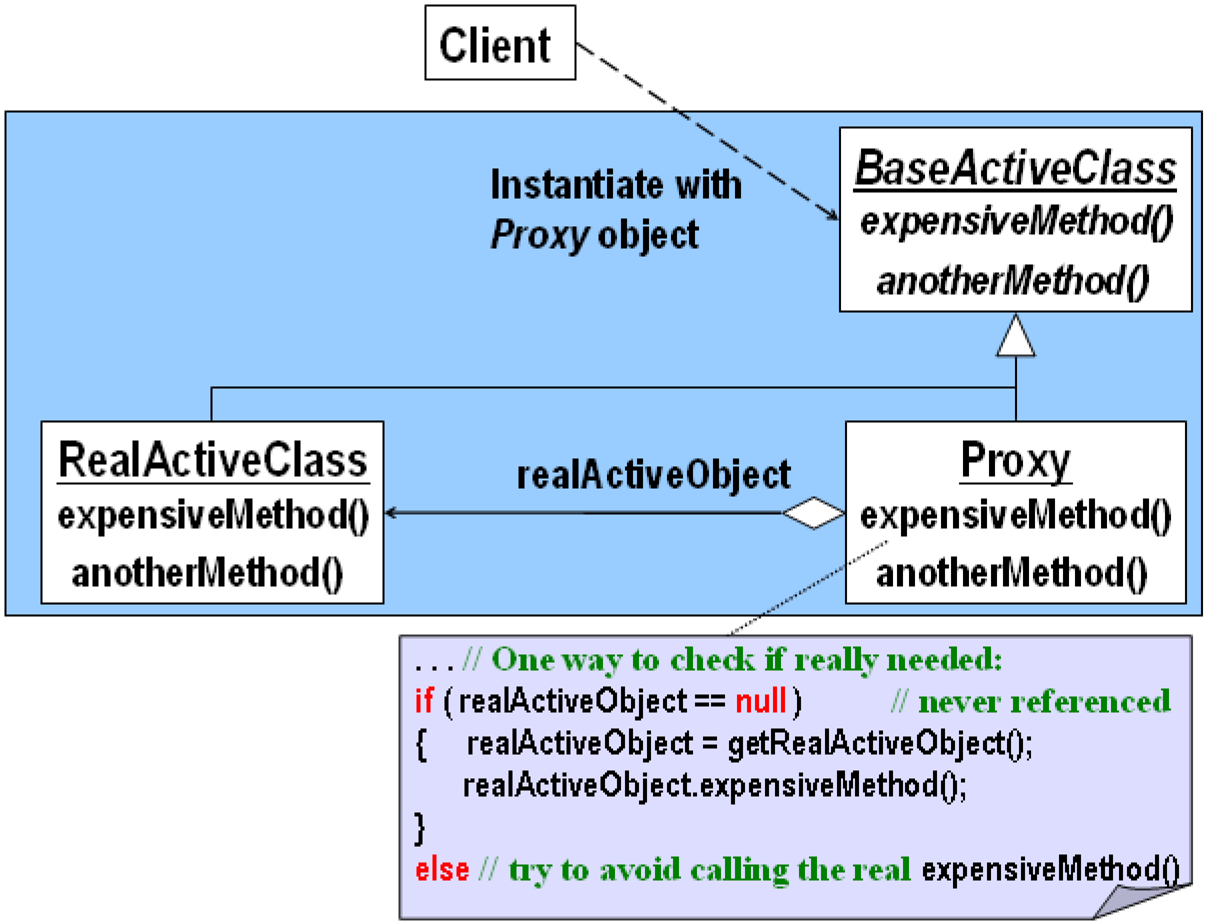
  – Interpose a substitute class which accesses the expensive functionality only when required.

# Proxy: Class Model



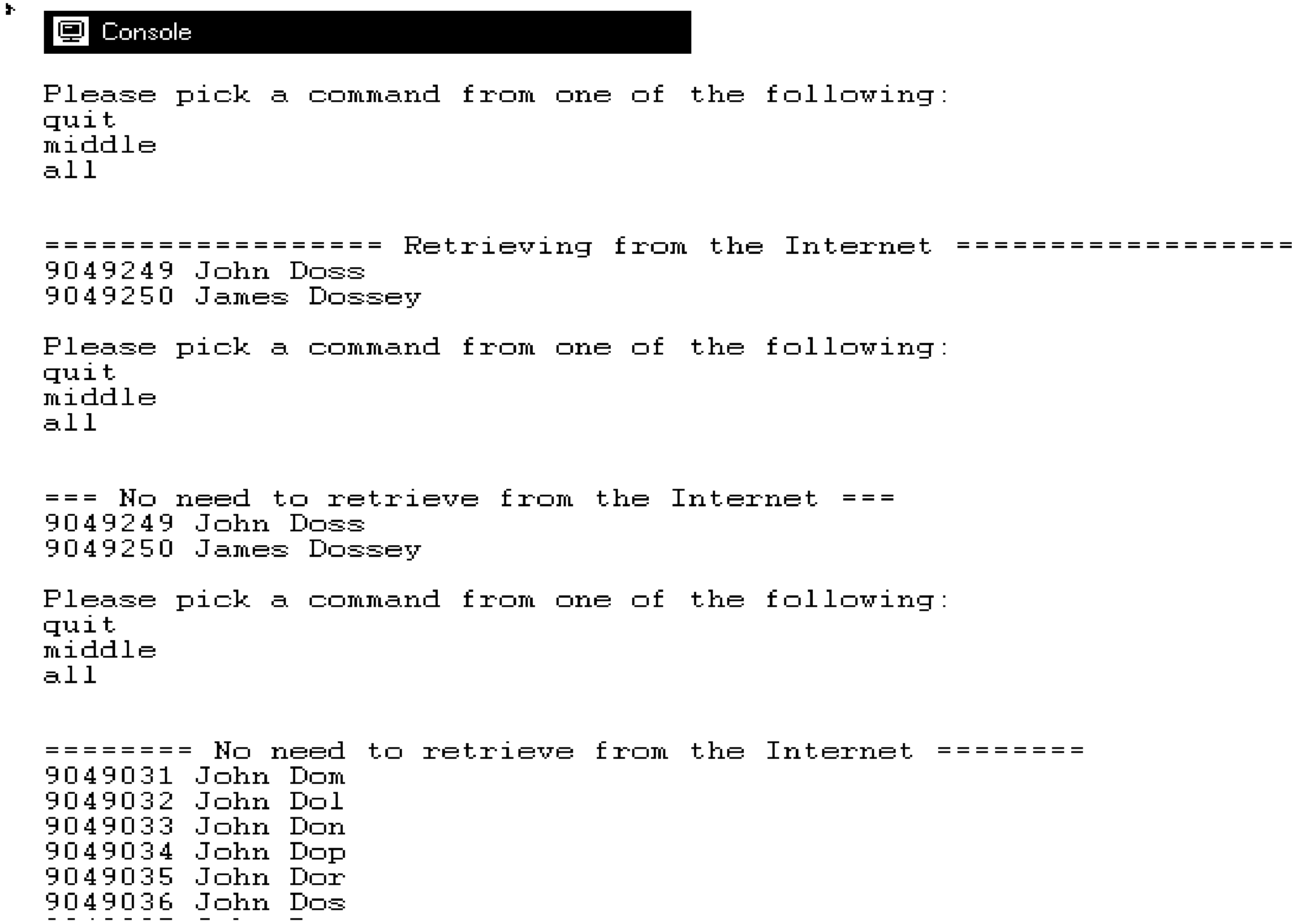
```
. . . // One way to check if really needed:
if ( realActiveObject == null )        // never referenced
{    realActiveObject = getRealActiveObject();
     realActiveObject.expensiveMethod();
}
else // try to avoid calling the real expensiveMethod()
```

# Proxy: Class Model



```
. . . // One way to check if really needed:
if ( realActiveObject == null )          // never referenced
{    realActiveObject = getRealActiveObject();
     realActiveObject.expensiveMethod();
}
else // try to avoid calling the real expensiveMethod()
```

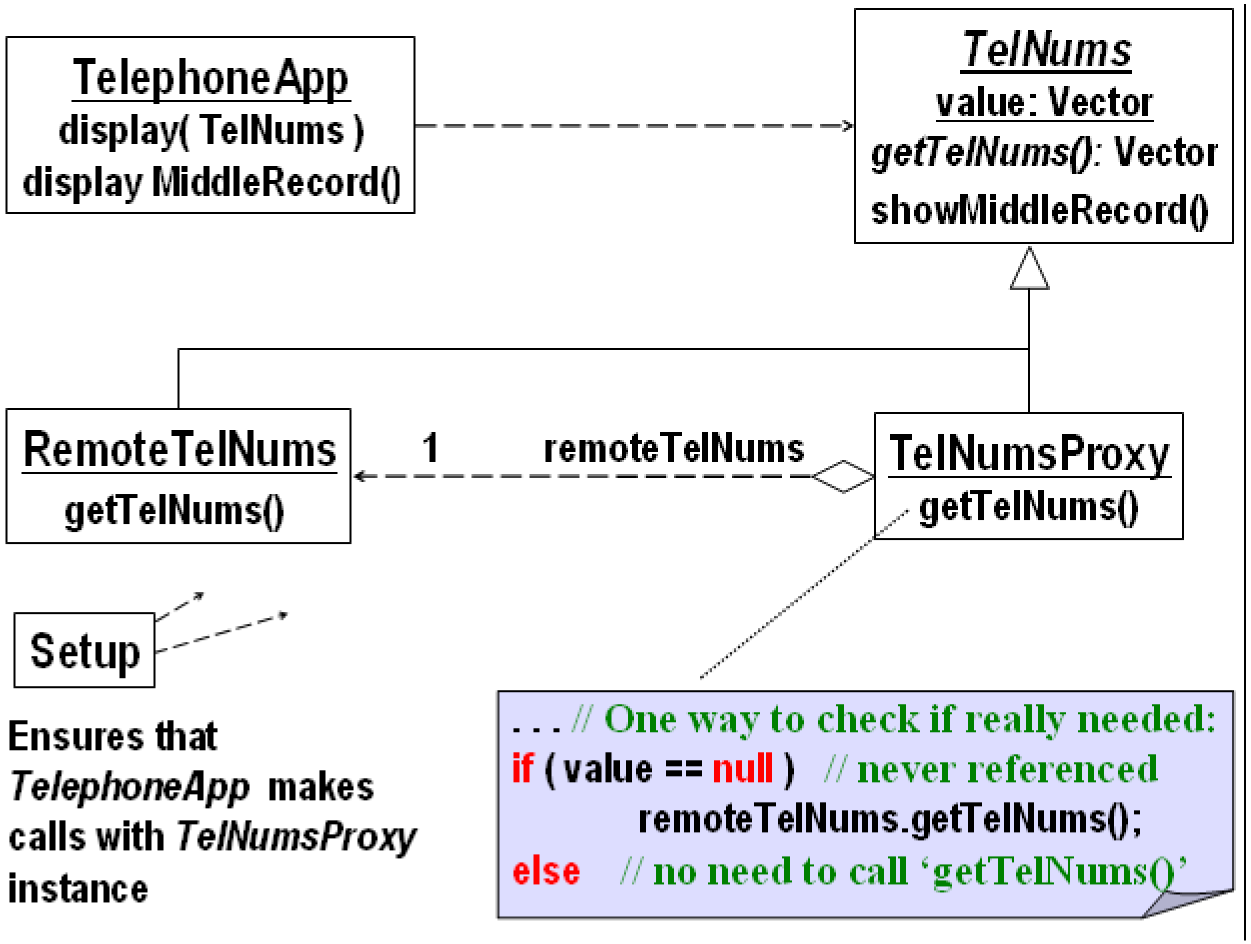
## Telephone Record Example



Console

```
Please pick a command from one of the following:
quit
middle
all


================== Retrieving from the Internet ==================
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all


=== No need to retrieve from the Internet ===
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all


======== No need to retrieve from the Internet ========
9049031 John Dom
9049032 John Dol
9049033 John Don
9049034 John Dop
9049035 John Dor
9049036 John Dos
```

# Telephone Record Example *(Cont'd)*



```
. . . // One way to check if really needed:
if ( value == null )   // never referenced
        remoteTelNums.getTelNums();
else   // no need to call 'getTelNums()'
```

## Proxy: Consequences

- Proxy promotes:
  - Efficiency: avoids time-consuming operations when necessary.
  - Correctness: separates design and code that are independent of retrieval/efficiency from parts concerned with this issue.
  - Reusability: design and code that are independent of retrieval efficiency are most likely to be reusable.
  - Flexibility: we can replace one module concerned with retrieval with another.
  - Robustness: isolates parts that check for the validity of retrieved data.

- The penalties we pay can sometimes be too high:
  - If the proxy forces us to keep very large amount of data in the memory and its use is infrequent.

## Proxy: Encapsulating Expensive Objects

**Name:** Proxy design pattern

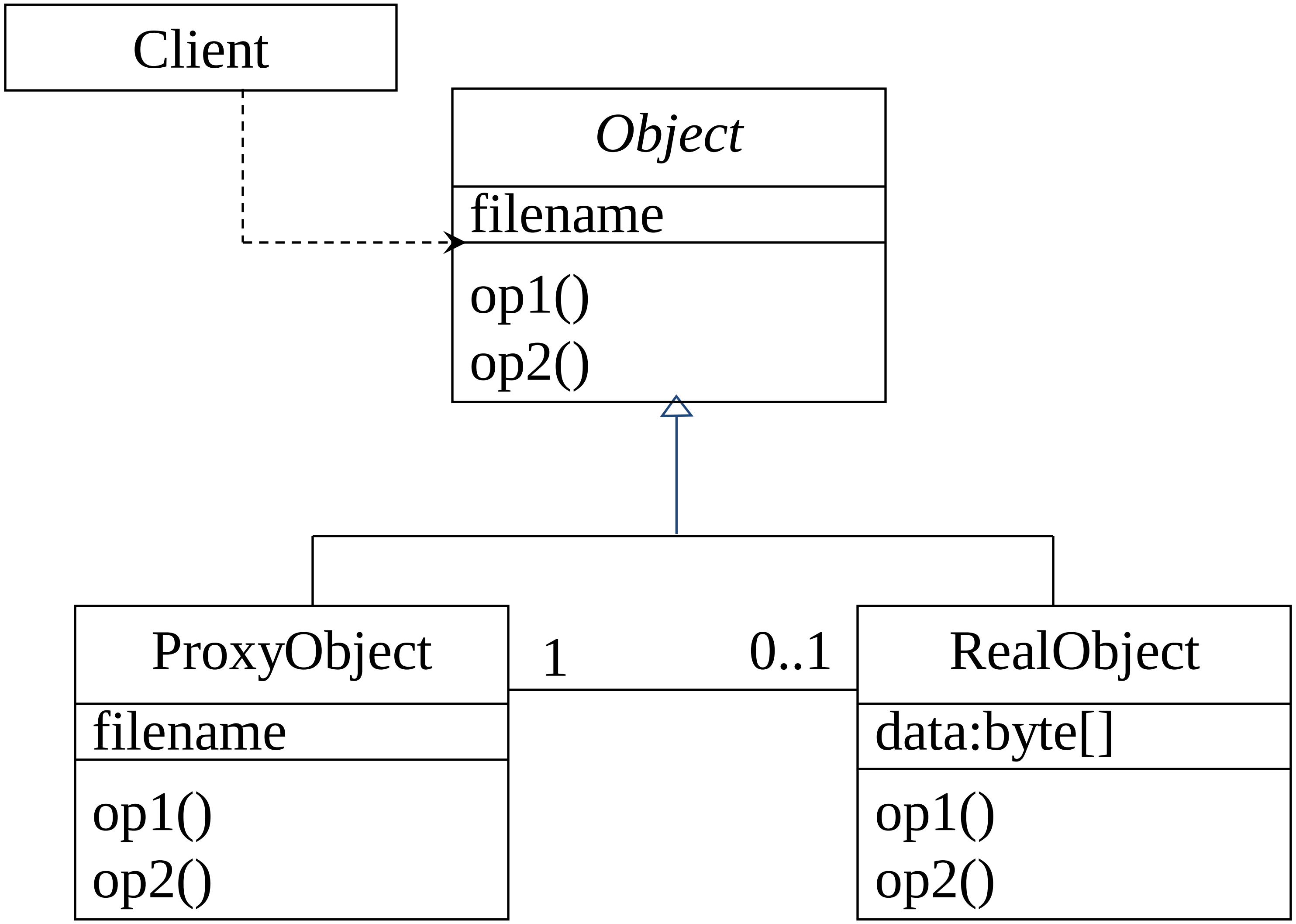
**Problem description:**

Improve performance or security of a system by delaying expensive computations, using memory only when needed, or checking access before loading an object into memory.

**Solution:**

The ProxyObject class acts on behalf of a RealObject class. Both implement the same interface. ProxyObject stores a subset of the attributes of RealObject. ProxyObject handles certain requests, whereas others are delegated to RealObject. After delegation, the RealObject is created and loaded into memory.

# Proxy: Class Diagram

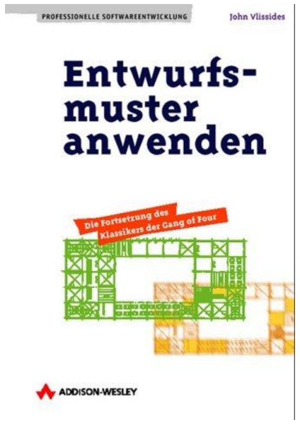


## Proxy: Consequences

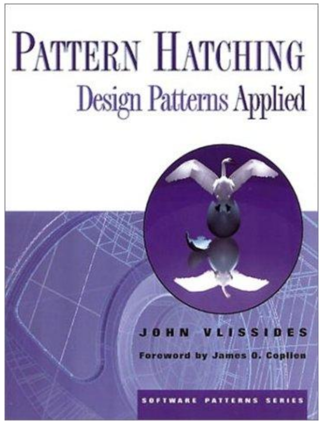
**Consequences:**

Adds a level of indirection between Client and RealObject.

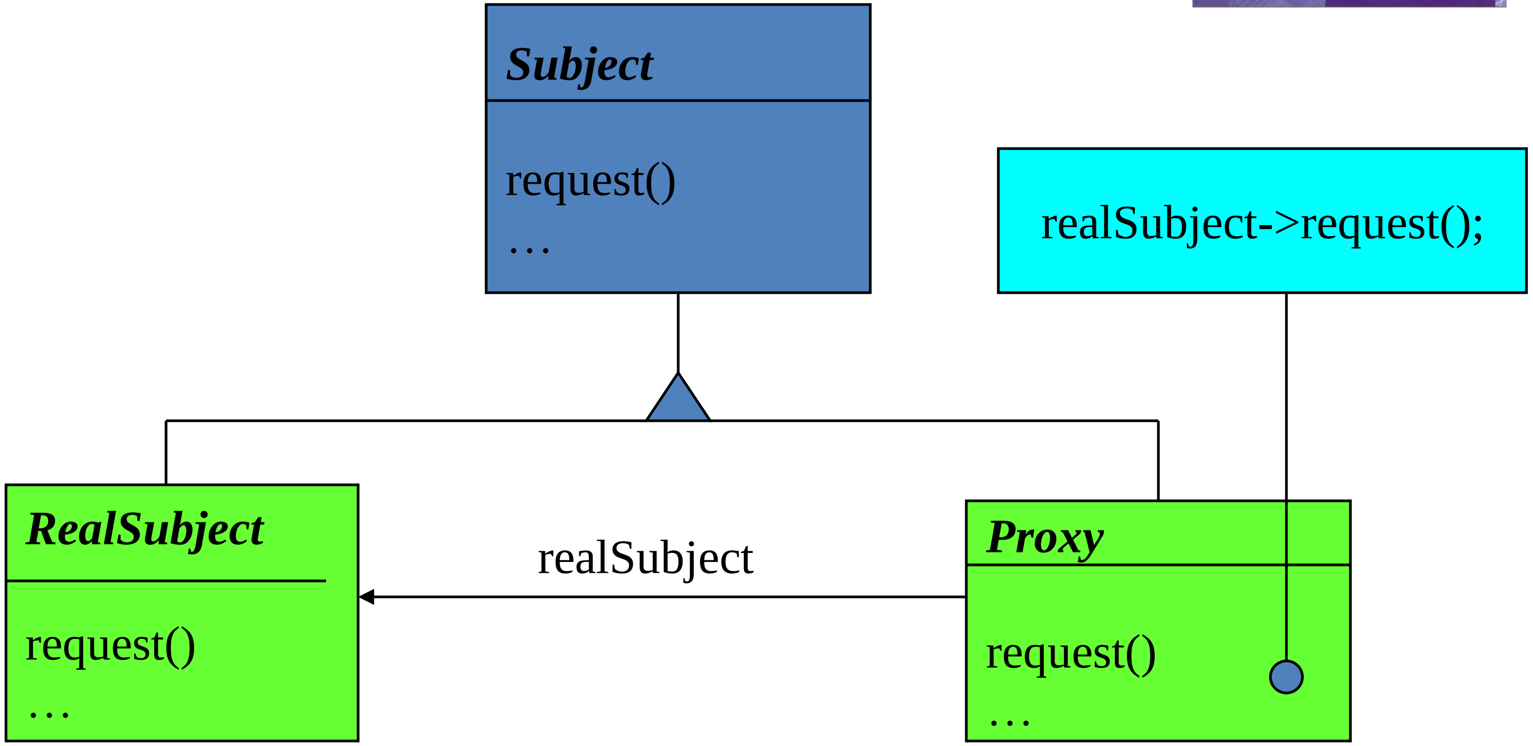
The Client is shielded from any optimization for creating RealObjects.

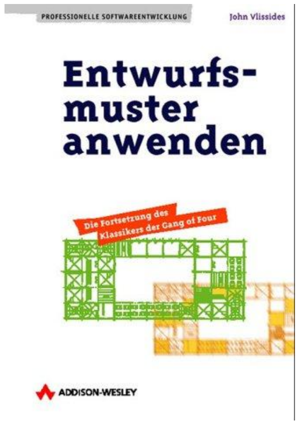

PROFESSIONELLE SOFTWAREENTWICKLUNG
John Vlissides
Entwurfs-muster anwenden
Die Fortsetzung des Klassikers der Gang of Four
ADDISON-WESLEY

# Pattern Hatching

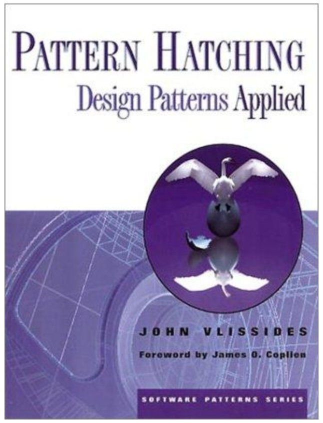


## Proxy Pattern




PROFESSIONELLE SOFTWAREENTWICKLUNG
John Vlissides
Entwurfs-muster anwenden
Die Fortsetzung des Klassikers der Gang of Four
ADDISON-WESLEY

# Pattern Hatching

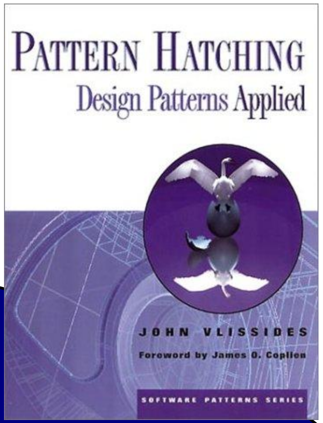


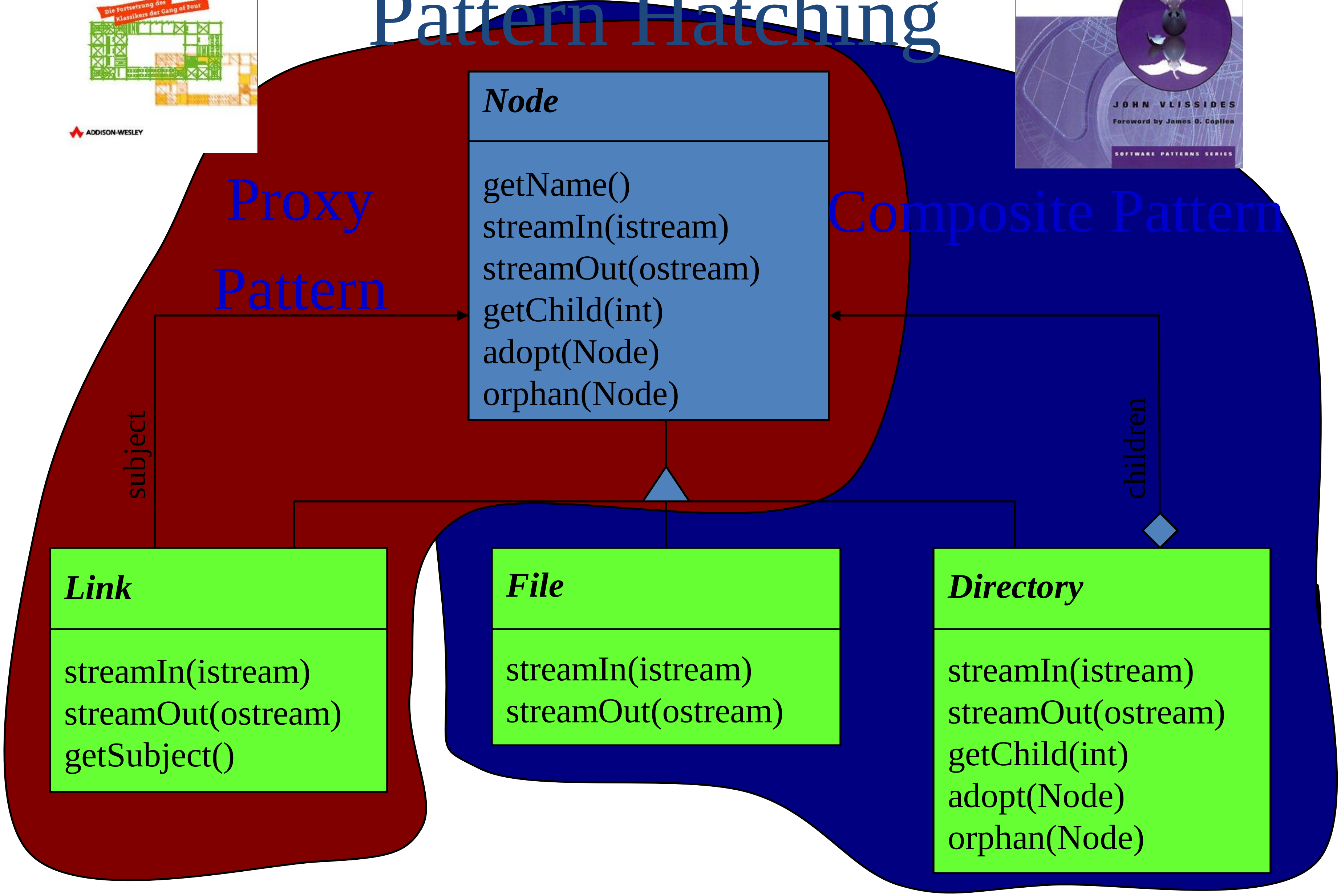
## Proxy Pattern

- We need to find a common structure for the proxy pattern with our composite pattern
- As we recognize, our common interface that we still want to use for the file-system is Node
- And because the Composite structure uses a common interface already, we can combine the Proxy “Subject” Interface into our Node Interface

# Pattern Hatching

PROFESSIONELLE SOFTWAREENTWICKLUNG
John Vlissides
Entwurfsmuster anwenden
Die Fortsetzung des Klassikers der Gang of Four
ADDISON-WESLEY





# Flyweight Design Pattern

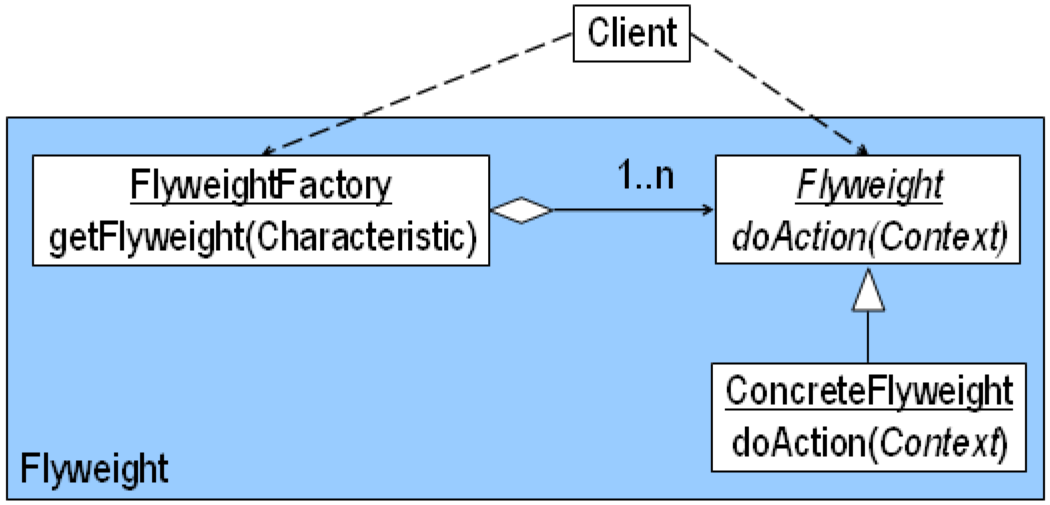
- Design Purpose

  Manage a large number of objects without constructing them all.

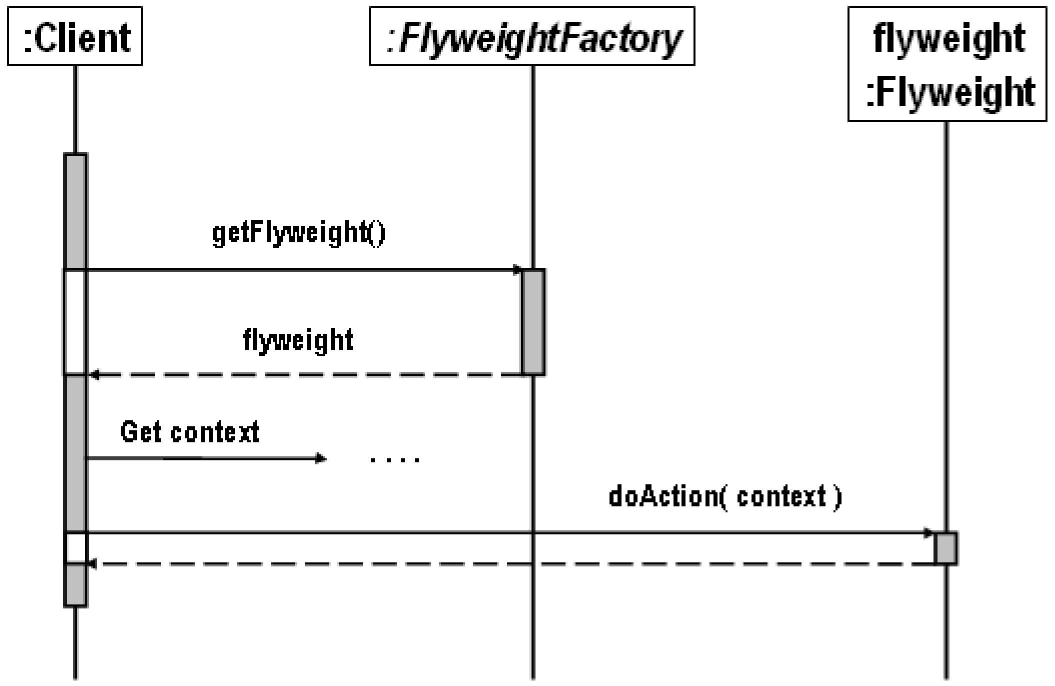
- Design Pattern Summary

  – Share representatives for the objects; use context to obtain the effect of multiple instances.

# Flyweight: Class Model



# Flyweight: Sequence Diagram



## Text Magnifier Example

Input

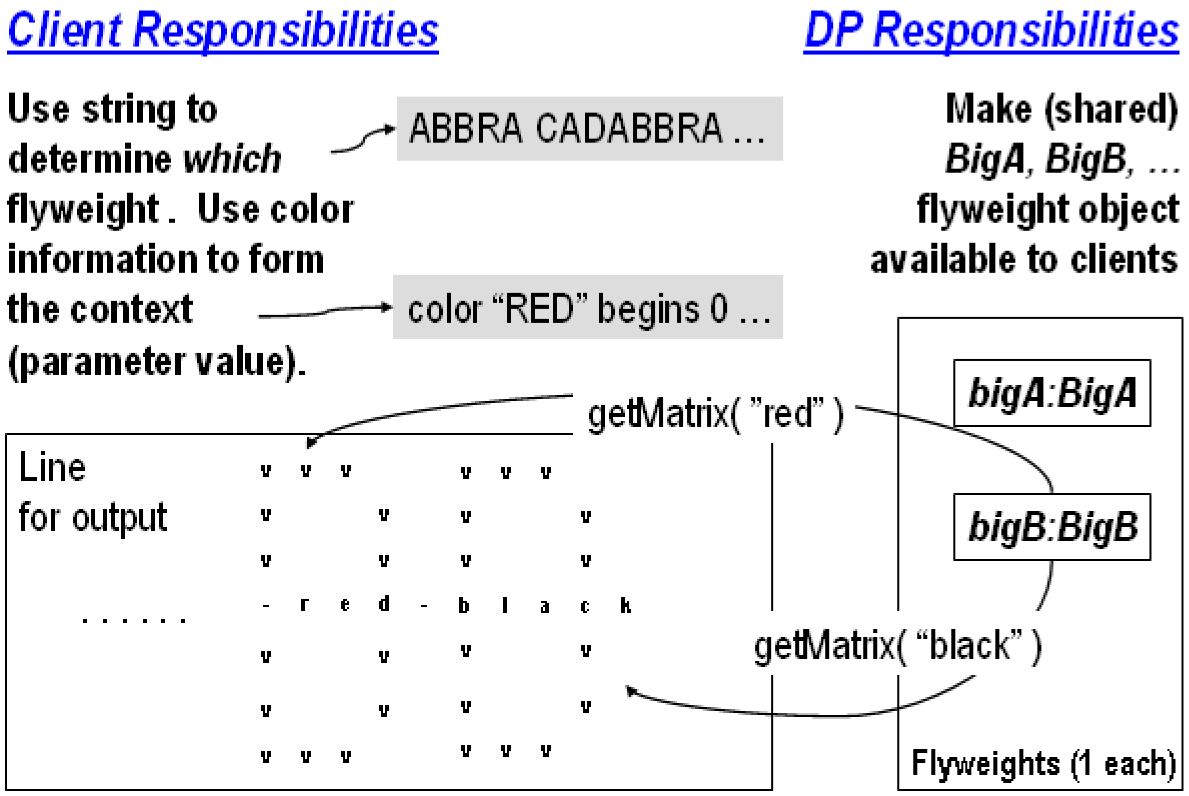
1 ABBRA CADABBRAA ARE THE FIRST TWO OF MANY WORDS IN THIS FILE …

2 Input color: RED ….. Starting character: 2 … Ending character: 3

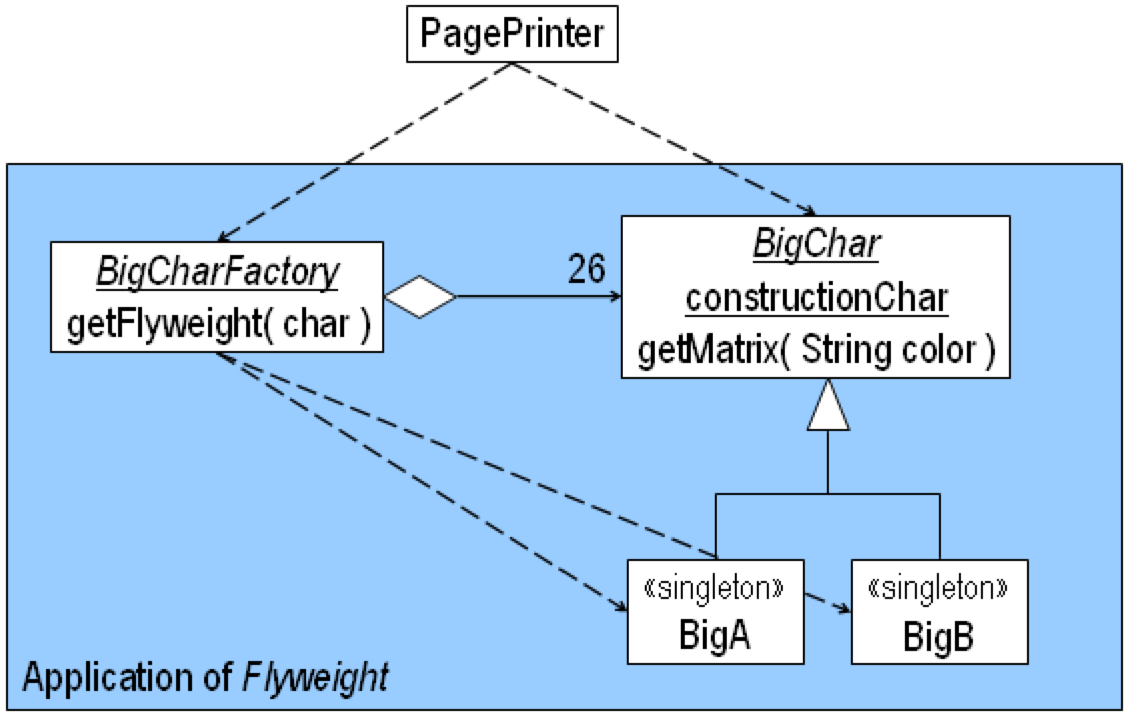
Output

```
    o       v v v       v v v
  o   o     v     v     v     v
  o   o     v     v     v     v
o       o   - R E D -   - R E D -       . . . . . .
o o o o o   v     v     v     v
o       o   v     v     v     v
o       o   v v v       v v v
```

# Text Magnifier Example *(Cont'd)*



# Text Magnifier Example *(Cont'd)*



## Flyweight: Consequences

- Space savings increase as more flyweights are shared.

## An example: Decorator

Intent:

Allow to attach responsibilities to objects, extend their functionality, dynamically

Motivation:

Consider a WYSIWYG editor

The basic textWindow needs various decorations: borders, tool bars, scroll bars, …

## An “obvious” solution: sub-classing

- A sub-class for every option

Disadvantages:

- Sub-classing is static – compile-time
- # of sub-classes = # of combinations – exponential

## The preferred solution: Use decorator objects

- Each decorator adds one decoration – property, behavior ➔ Linear
- Can add more than one, to obtain combinations of properties
- And, do it dynamically

## Structure, participants, collaborations:

- `Component` – an interface, describes the operations of a (GUI) component
- `ConcreteComponent` (extends `Component`)

  class of objects to which decorations can be added (e.g. the original TextWindow)
- `Decorator` (extends `Component`)– a class of decorator objects (there are several such classes)

## Implementation:

- Each decorator has a (private) reference to a Component object (CC or D)
- For each operation, the decorator performs its decoration job, & delegates Component operations to the Component object it contains
- Decorator extends Component (can accept Component requests)

# Facade Design Pattern

- Design Purpose

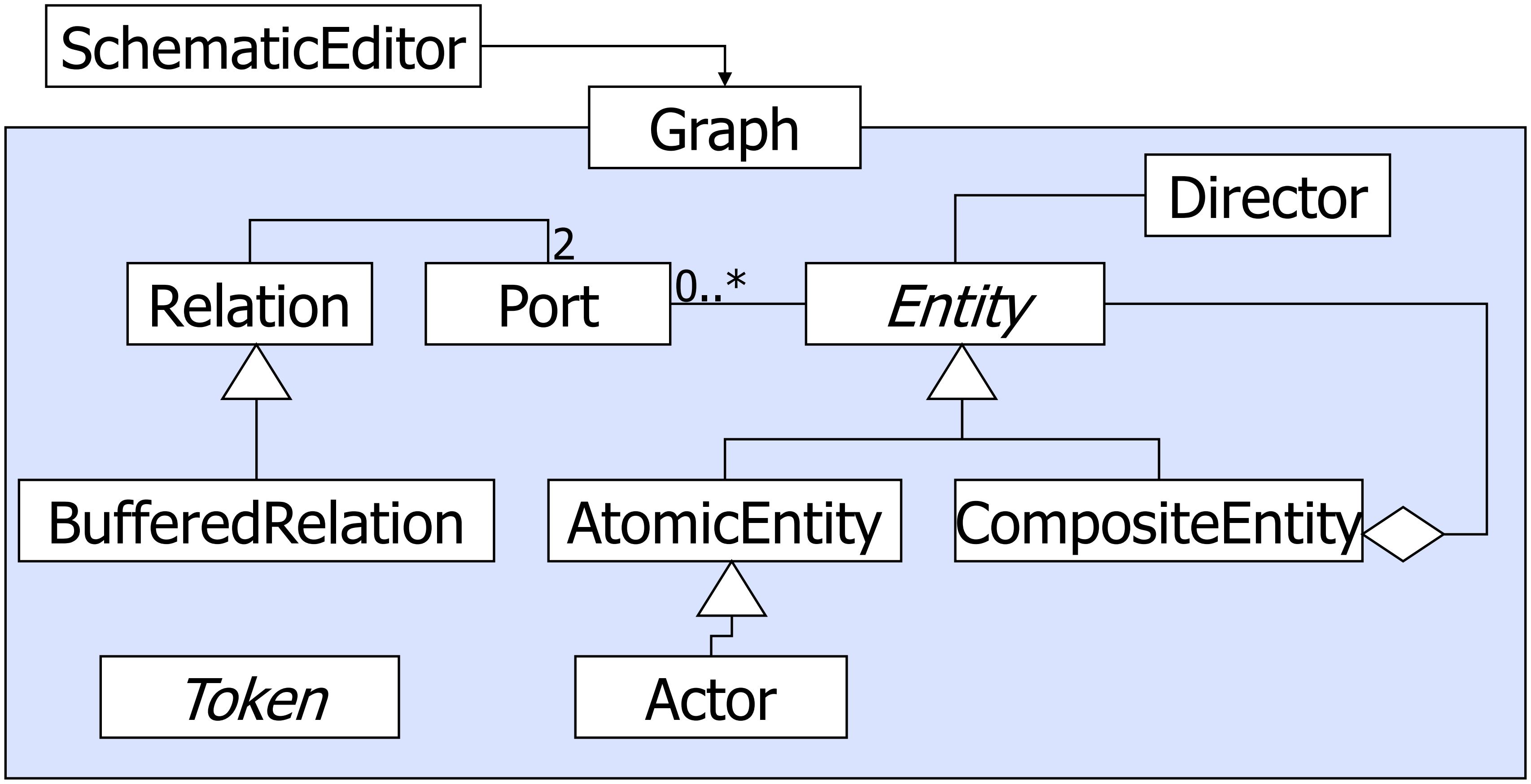
  Provide an interface to a package of classes.

- Design Pattern Summary

  – Define a singleton which is the sole means for obtaining functionality from the package.

# Façade (2)

- Example: graph interface to a simulation engine



## Facade: Encapsulating Subsystems

**Name:** Facade design pattern

**Problem description:**

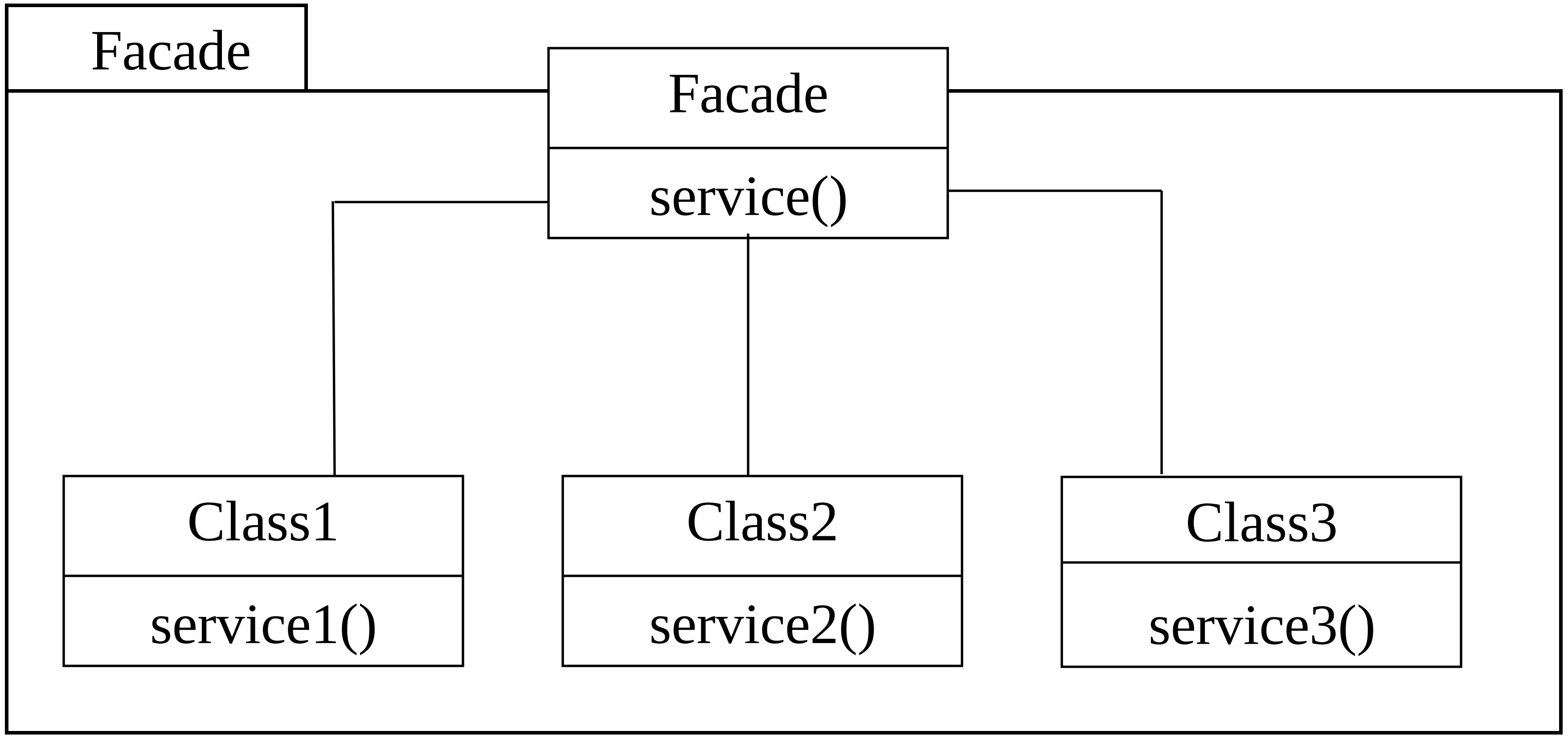
Reduce coupling between a set of related classes and the rest of the system.

**Solution:**

A single Facade class implements a high-level interface for a subsystem by invoking the methods of the lower-level classes.

Example. A Compiler is composed of several classes: LexicalAnalyzer, Parser, CodeGenerator, etc. A caller, invokes only the Compiler (Facade) class, which invokes the contained classes.

# Facade: Class Diagram



# Facade: Consequences

**Consequences:**

Shields a client from the low-level classes of a subsystem.

Simplifies the use of a subsystem by providing higher-level methods.

Enables lower-level classes to be restructured without changes to clients.

Note. The repeated use of Facade patterns yields a layered system.

## Summary of Structural Design Patterns

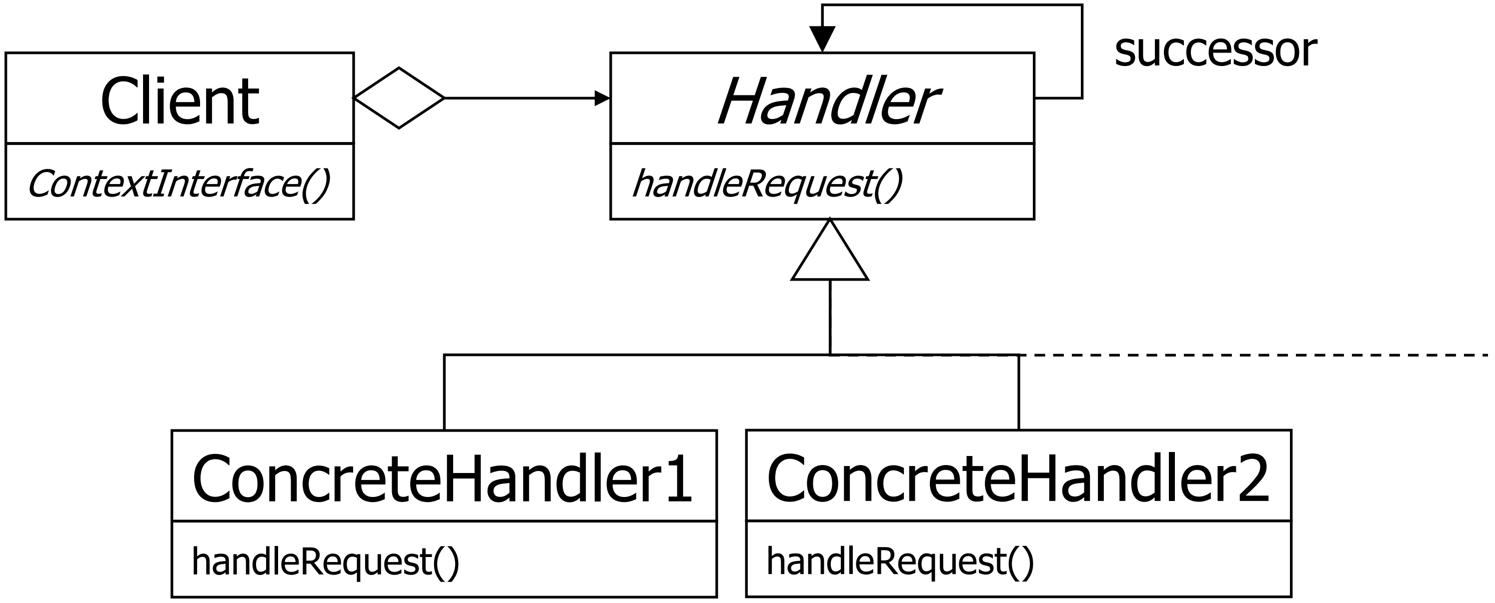
- *Structural Design Patterns* relate objects (as trees, lists etc.)
  - *Facade* provides an interface to collections of objects
  - *Decorator* adds to objects at runtime
  - *Composite* represents trees of objects
  - *Adapter* simplifies the use of external functionality
  - *Flyweight* gains the advantages of using multiple instances while minimizing space penalties
  - *Proxy* avoids calling expensive operations unnecessarily

# Unit-4 part-1
# behavioural patterns part-1

| S. No | TOPIC | PPT Slides | |
|---|---|---|---|
| | Behavioral Patterns Part-I introduction UNIT-4 | | |
| 1 | Chain of Responsibility | L1 | 2 – 3 |
| 2 | Command | L2 | 4 – 9 |
| 3 | interpreter | L3 | 10 – 12 |
| 4 | Iterator | L4 | 13 – 17 |
| 5 | Reusable points in Behavioral Patterns | L5 | 18 – 21 |
| 6 | (Intent, Motivation, Also Known As ……………) | L6 | 22 – 24 |
| 7 | Review Unit-VI | L7 | 25 – 25 |

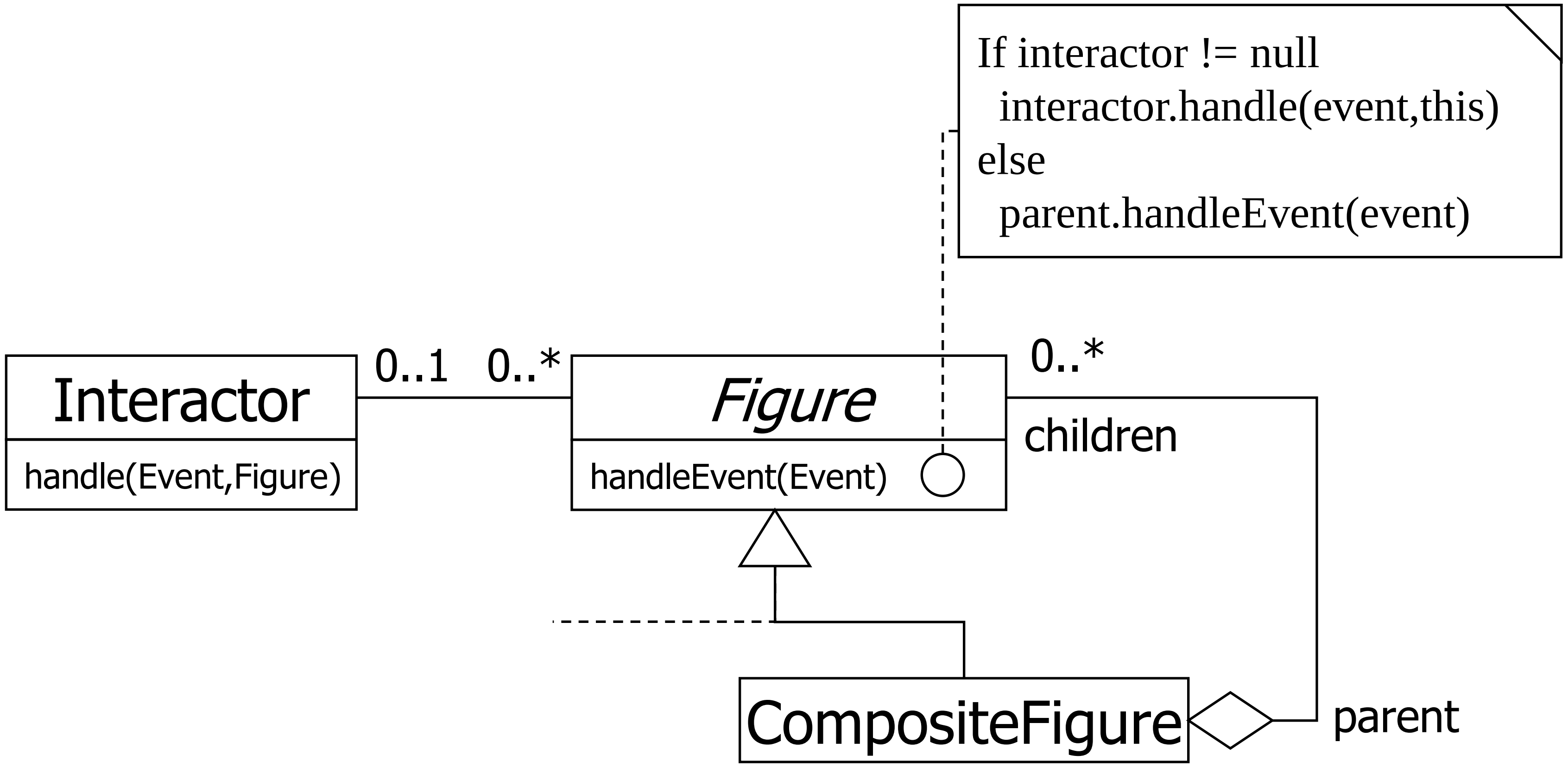
# Chain of Responsibility

- Decouple sender of a request from receiver
- Give more than one object a chance to handle
- Flexibility in assigning responsibility
- Often applied with Composite



# Chain of Responsibility (2)

- Example: handling events in a graphical hierarchy



# Command: Encapsulating Control Flow

**Name:** Command design pattern

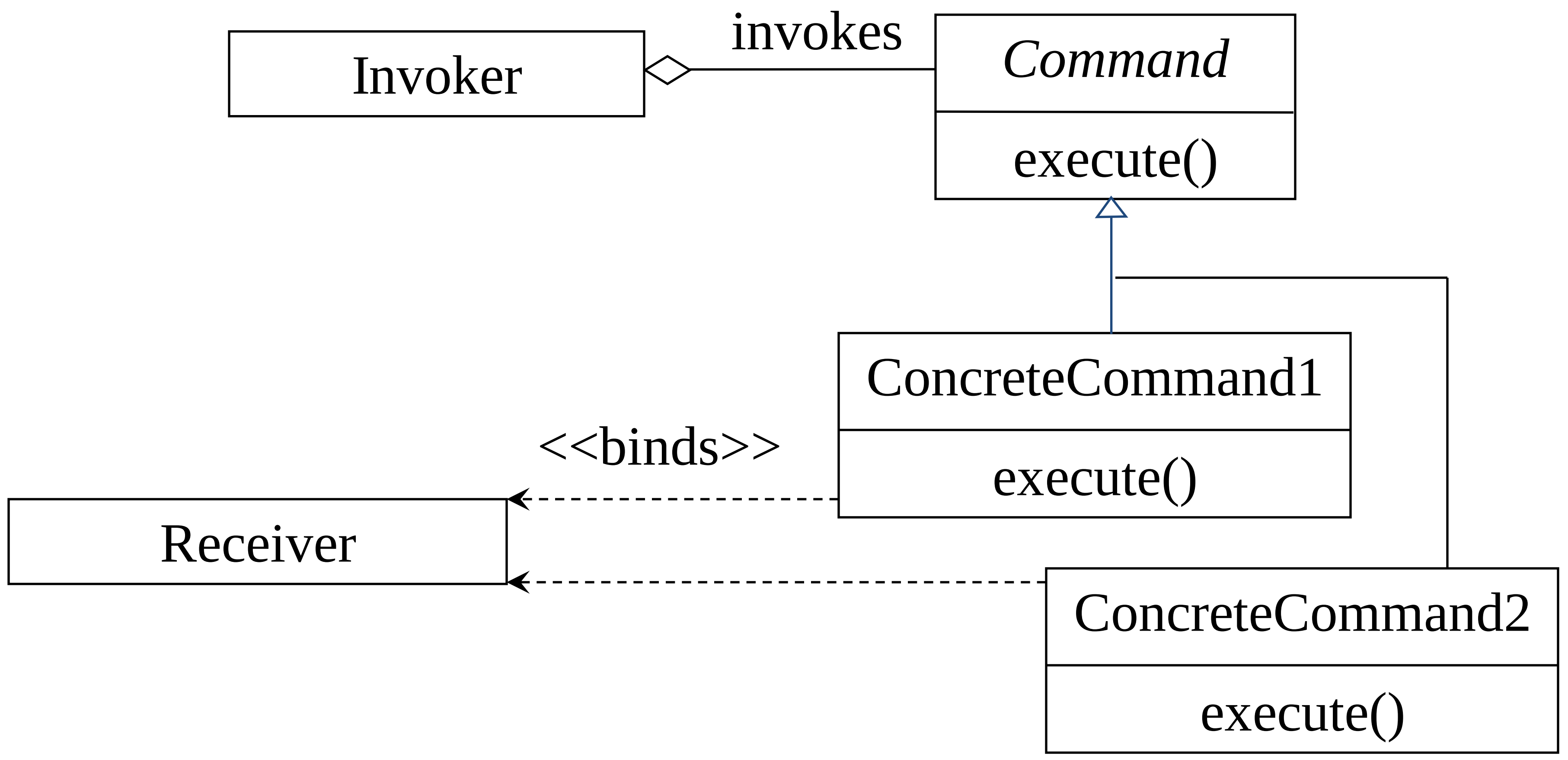
**Problem description:**

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

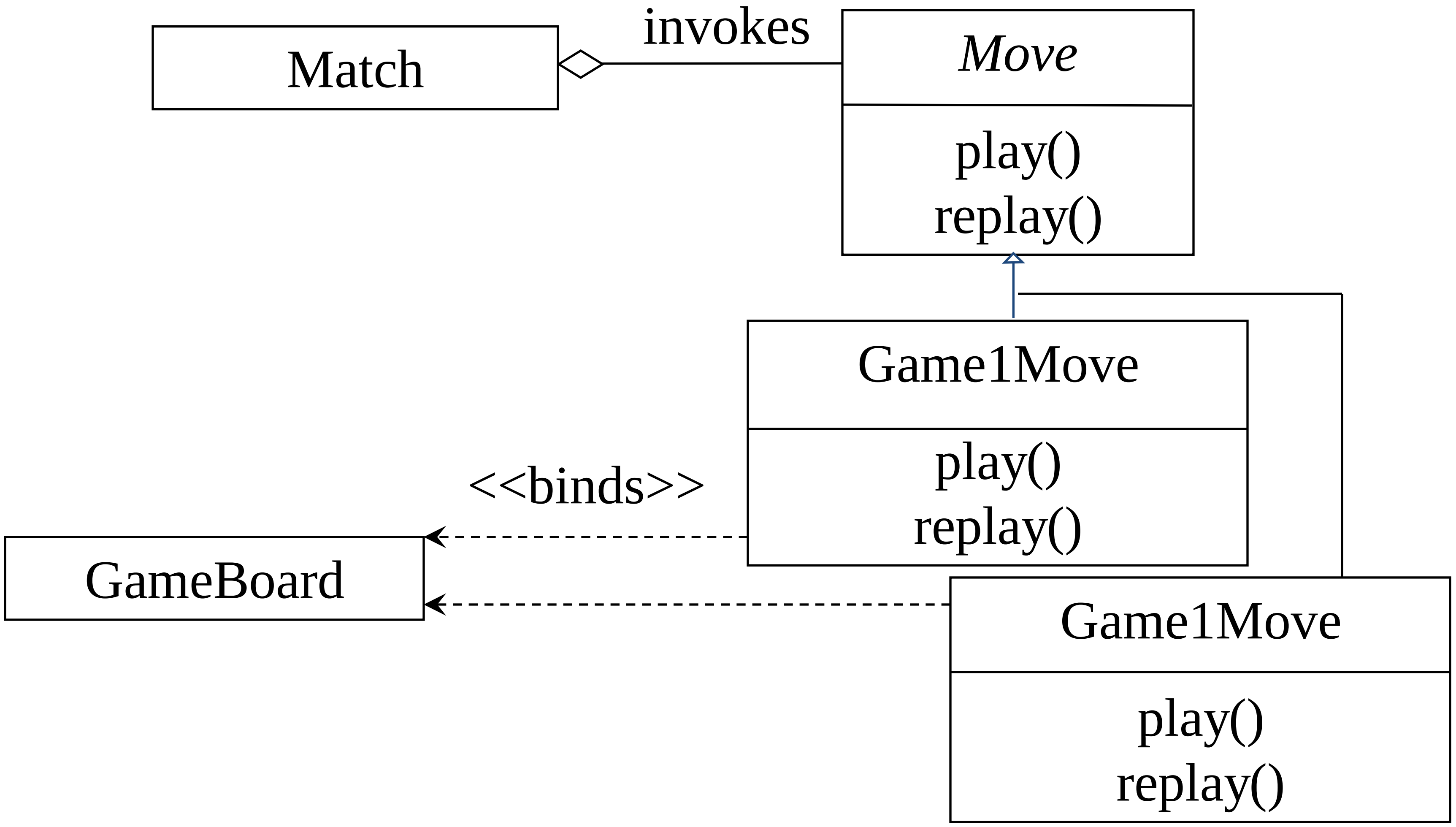
**Solution:**

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

# Command: Class Diagram



# Command: Class Diagram for Match



# Command: Consequences

**Consequences:**

The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

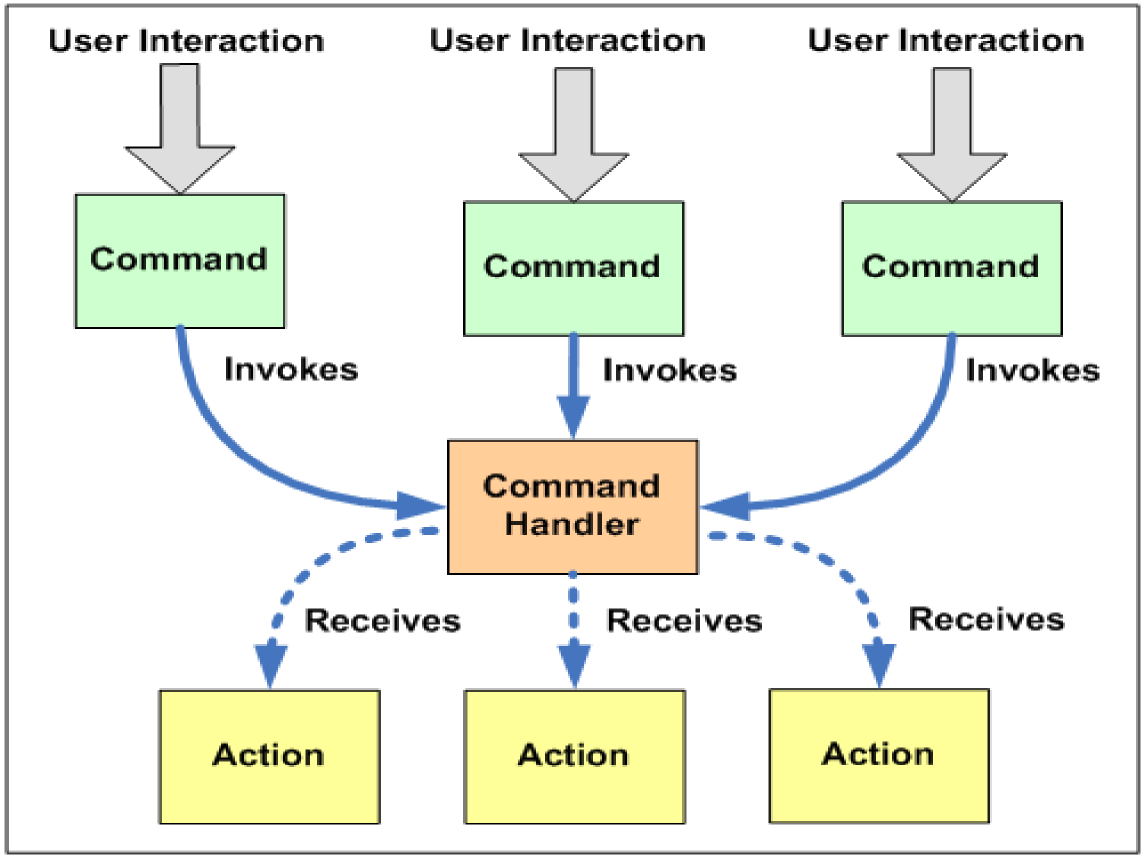
ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

## Command

- You have commands that need to be
  - executed,
  - undone, or
  - queued
- *Command* design pattern separates
  - Receiver from Invoker from Commands
- All commands derive from *Command* and implement do(), undo(), and redo()

# Command Design Pattern



- Separates command invoker and receiver

## Pattern: Interpreter

- **Intent**: Given a language, interpret sentences
- **Participants**: Expressions, Context, Client
- **Implementation**: A class for each expression type
  An Interpret method on each class
  A class and object to store the global state (context)
- No support for the parsing process
  (Assumes strings have been parsed into exp trees)

# Pattern: Interpreter with Macros

- **Example**: Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF
  Command(move(D)) -> “go”, Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation
- Builds tree structure only as needed
  (Or, can automatically build complete trees)
- May or may not use expression classes

# Method Combination

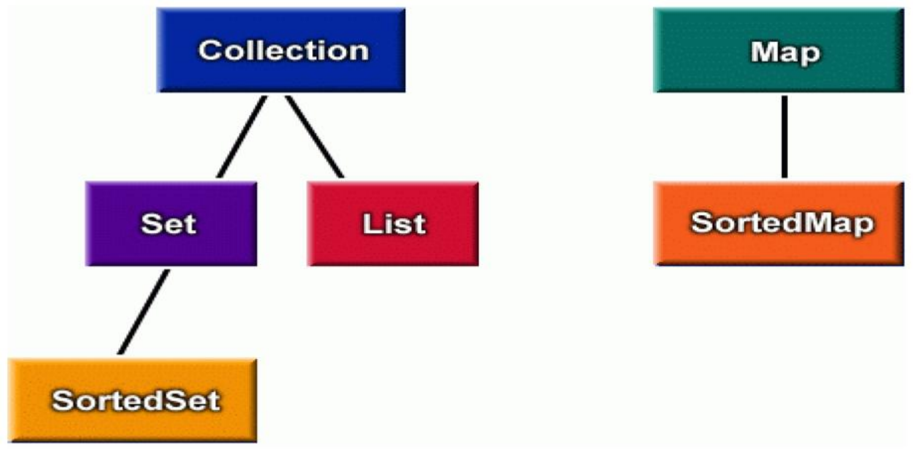
- Build a method from components in different classes
- **Primary** methods: the “normal” methods; choose the most specific one
- **Before/After** methods: guaranteed to run;
  No possibility of forgetting to call super
  Can be used to implement *Active Value* pattern
- **Around** methods: wrap around everything;
  Used to add tracing information, etc.
- Is added complexity worth it?
  Common Lisp: Yes;   Most languages: No

## Iterator pattern

- **iterator**: an object that provides a standard way to examine all elements of any collection
- uniform interface for traversing many different data structures without exposing their implementations
- supports concurrent iteration and element removal
- removes need to know about internal structure of collection or different methods differentcollections

## Pattern: Iterator

*objects that traverse collections*



## Iterator interfaces in Java

```
public interface java.util.Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
}

public interface java.util.Collection {
  ...   // List, Set extend Collection
  public Iterator iterator();
}

public interface java.util.Map {
  ...
  public Set keySet();          // keys,values are Collections
  public Collection values();   // (can call iterator() on them)
}
```

## Iterators in Java

- all Java collections have a method iterator that returns an iterator for the elements of the collection
- can be used to look through the elements of any kind of collection (an alternative to for loop)

```
List list = new ArrayList();
... add some elements ...
```

```
set.iterator()
map.keySet().iterator()
map.values().iterator()
```

```
for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

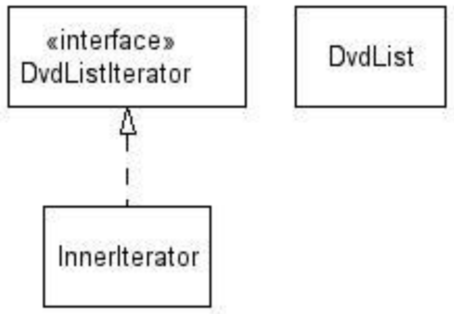
# Adding your own Iterators

- when implementing your own collections, it can be very convenient to use Iterators
  - discouraged (has nonstandard interface):

```
public class PlayerList {
  public int getNumPlayers() { ... }
  public boolean empty() { ... }
  public Player getPlayer(int n) { ... }
}
```

  - preferred:

```
public class PlayerList {
  public Iterator iterator() { ... }
  public int size() { ... }
  public boolean isEmpty() { ... }
}
```



# Command: Encapsulating Control Flow

**Name:** Command design pattern

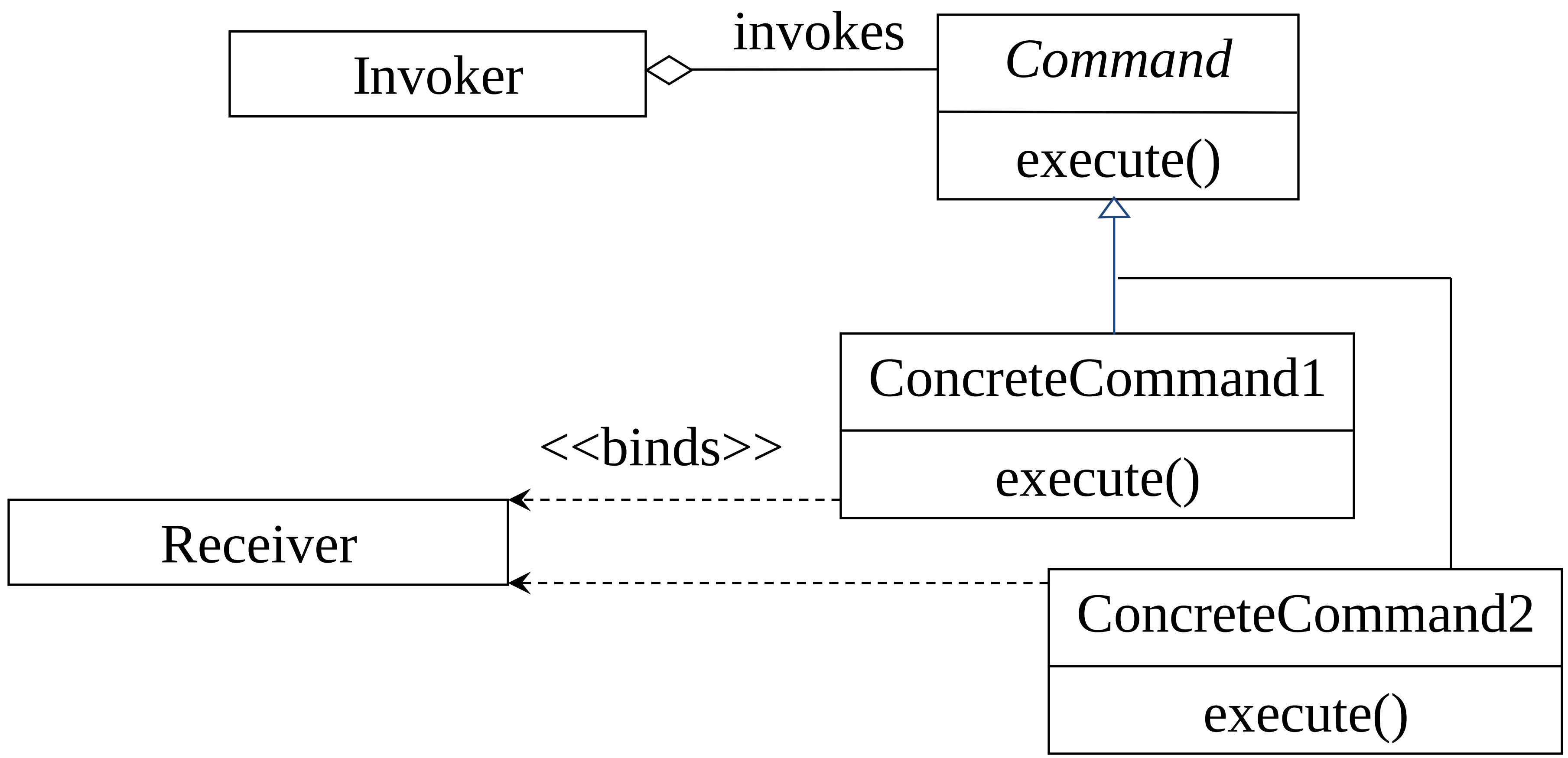
**Problem description:**

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

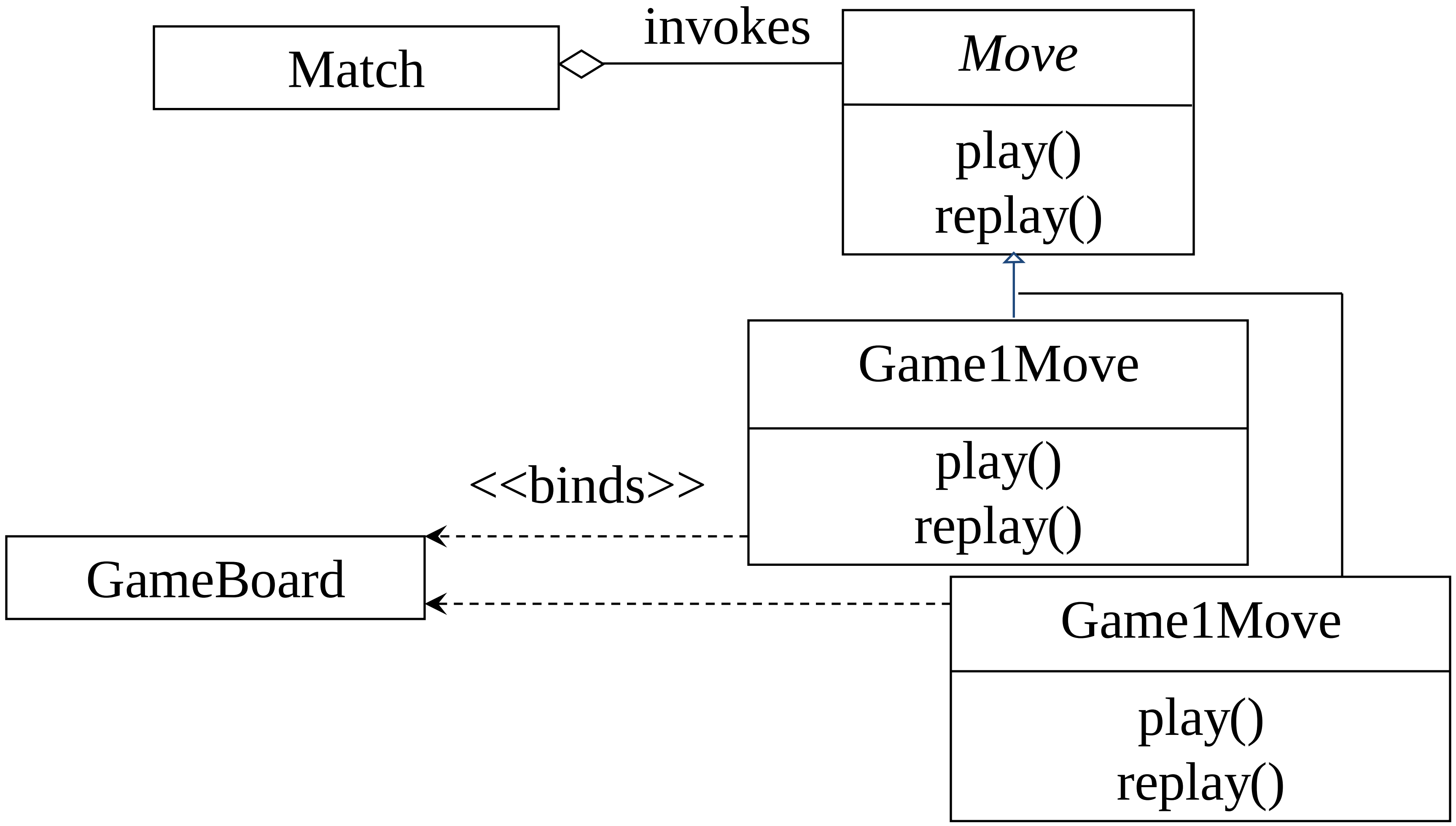
**Solution:**

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

# Command: Class Diagram



# Command:
# Class Diagram for Match



# Command: Consequences

**Consequences:**

The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

## Pattern: Interpreter

- **Intent**: Given a language, interpret sentences
- **Participants**: Expressions, Context, Client
- **Implementation**: A class for each expression type
  An Interpret method on each class
  A class and object to store the global state (context)
- No support for the parsing process
  (Assumes strings have been parsed into exp trees)

## Pattern: Interpreter with Macros

- **Example**: Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF
  `Command(move(D)) -> “go”, Direction(D).`
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation
- Builds tree structure only as needed
  (Or, can automatically build complete trees)
- May or may not use expression classes

## Method Combination

- Build a method from components in different classes
- **Primary** methods: the "normal" methods; choose the most specific one
- **Before/After** methods: guaranteed to run;
  No possibility of forgetting to call super
  Can be used to implement *Active Value* pattern
- **Around** methods: wrap around everything;
  Used to add tracing information, etc.
- Is added complexity worth it?
  Common Lisp: Yes;   Most languages: No

# References

- Information about Design Patterns:
  - http://msdn.microsoft.com/library/en-us/dnpag/html/intpatt.asp
  - http://www.patternshare.org/
  - http://msdn.microsoft.com/architecture/
  - http://msdn.microsoft.com/practices/
  - http://www.dofactory.com/Patterns/Patterns.aspx
  - http://hillside.net/
- Contact: alex@stonebroom.com
- Slides & code: http://www.daveandal.net/download/
- Article: http://www.daveandal.net/articles/

## Unit-4 part-2
## behavioural patterns part-2

| S. No | TOPIC | PPT Slides | |
|---|---|---|---|
| 1 | Behavioral patterns part-II introduction | L1 | 2 – 4 |
| 2 | Mediator | L2 | 5 – 12 |
| 3 | Memento | L3 | 13 – 32 |
| 4 | Observer | L4 | 33 – 54 |

## Behavioral Patterns (1)

• Deal with the way objects interact and distribute responsibility
• Chain of Responsibility: Avoid coupling the sender of arequest to its receiver by giving more than one object achance to handle the request. Chain the receiving objects an dpass the request along the chain until anobject handles it.
• Command: Encapsulate a request as an object, therebyletting you paramaterize clients with different requests,queue or log requests, and support undoable operations.
• Interpreter: Given a language, define a representationfor its grammar along with an interpreter that uses therepresentation to interpret sentences in the language.23

## Behavioral Patterns (2)

• Iterator: Provide a way to access the elements of anaggregate object sequentially without exposing itsunderlying representation.

• Mediator: Define an object that encapsulates how a setof objects interact. Mediator promotes loose coupling bykeeping objects from referring to each other explicitly,and lets you vary their interaction independently.

• Memento: Without violating encapsulation, capture andexternalize an object’s internal state so that the object
can be restored to this state later.

• Observer: Define a one-to-many dependency betweenobjects so that when one object changes state, all itsdependents are notified and updated automatically.

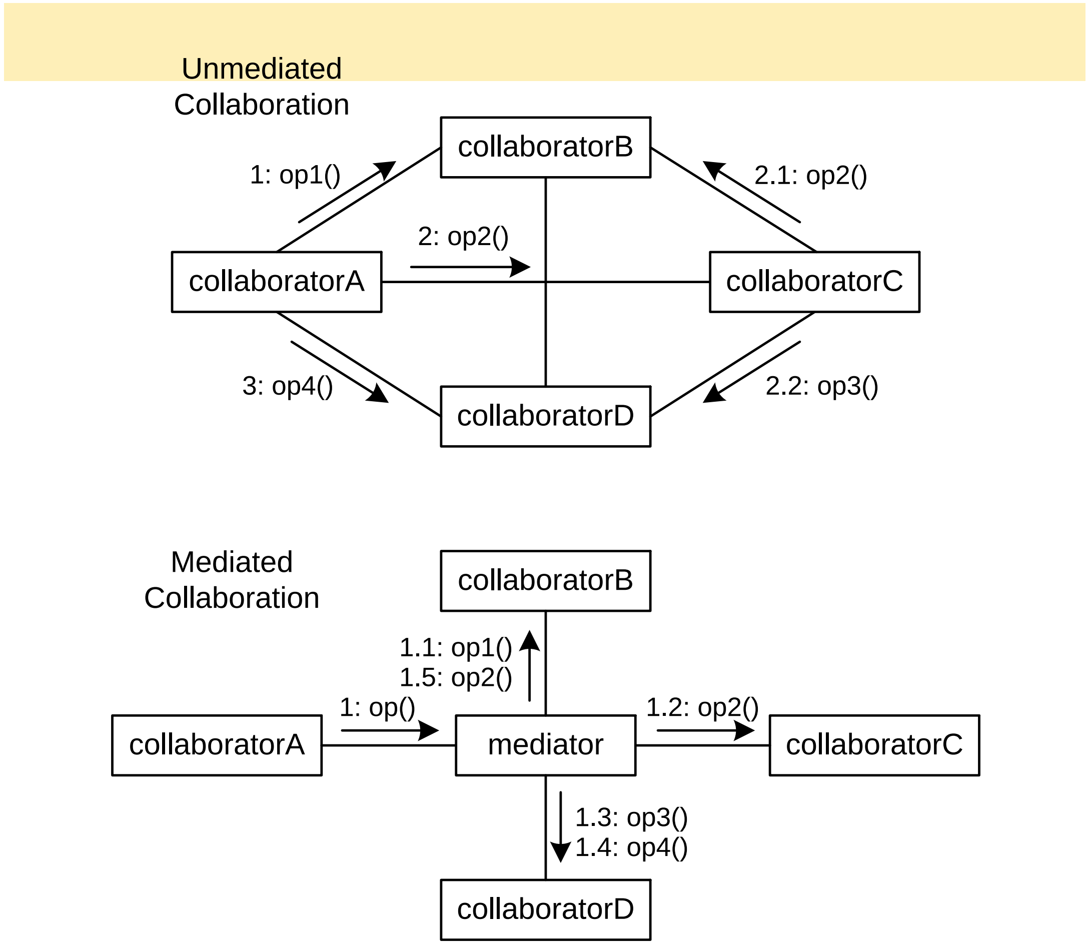
## Behavioral Patterns (3)

- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasseses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor: 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.

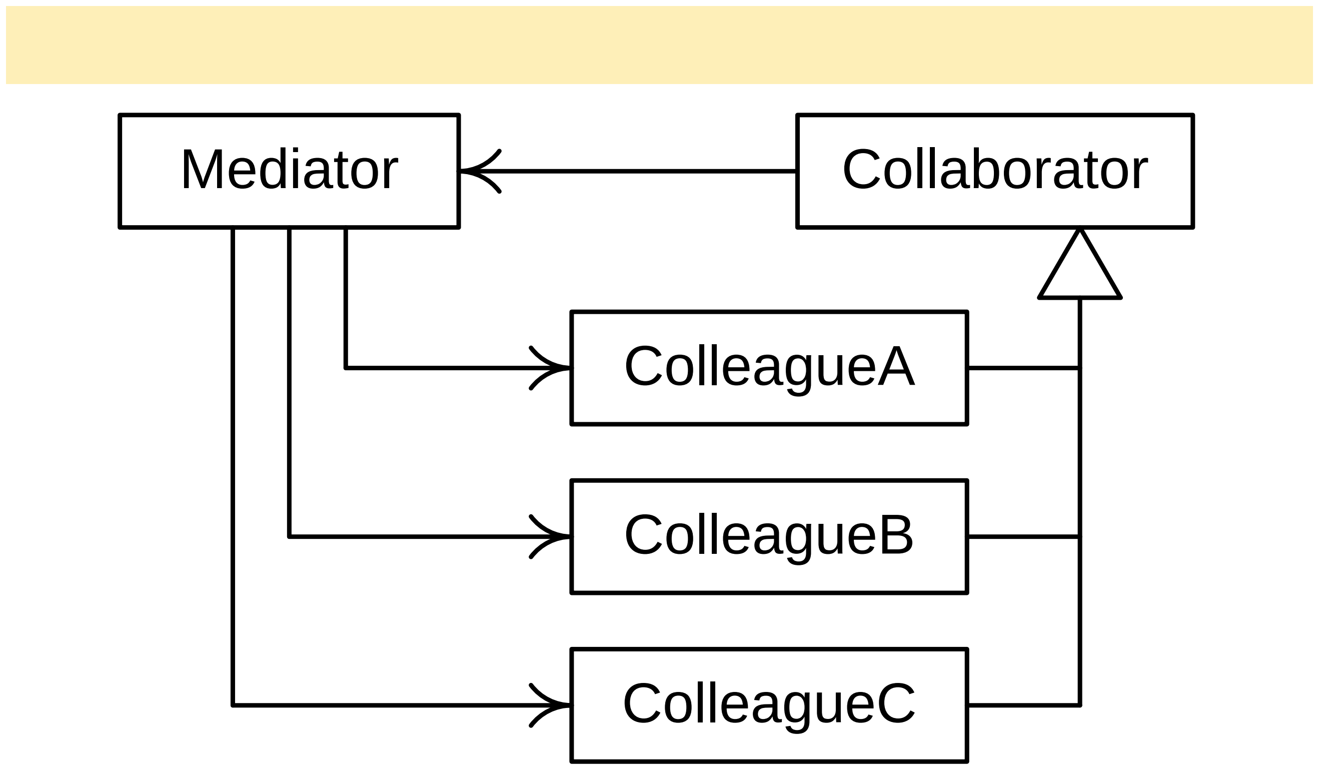
# The Mediator Pattern

- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.
- Classes interact only with a mediator class rather than with each other.
- Classes are coupled only to the mediator where interaction control code resides.
- Mediator is like a multi-way Façade pattern.
- Analogy: a meeting scheduler

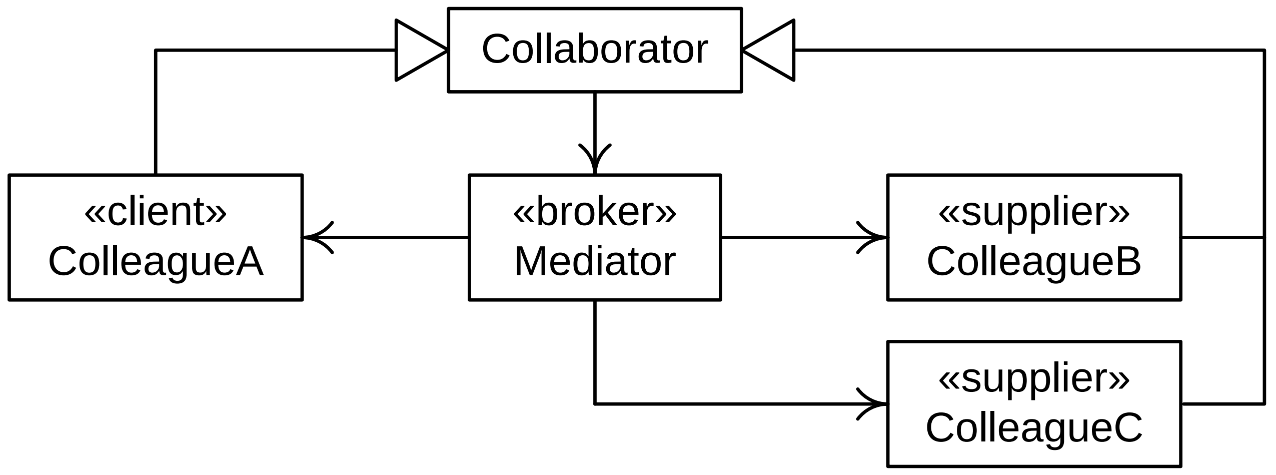
# Using a Mediator



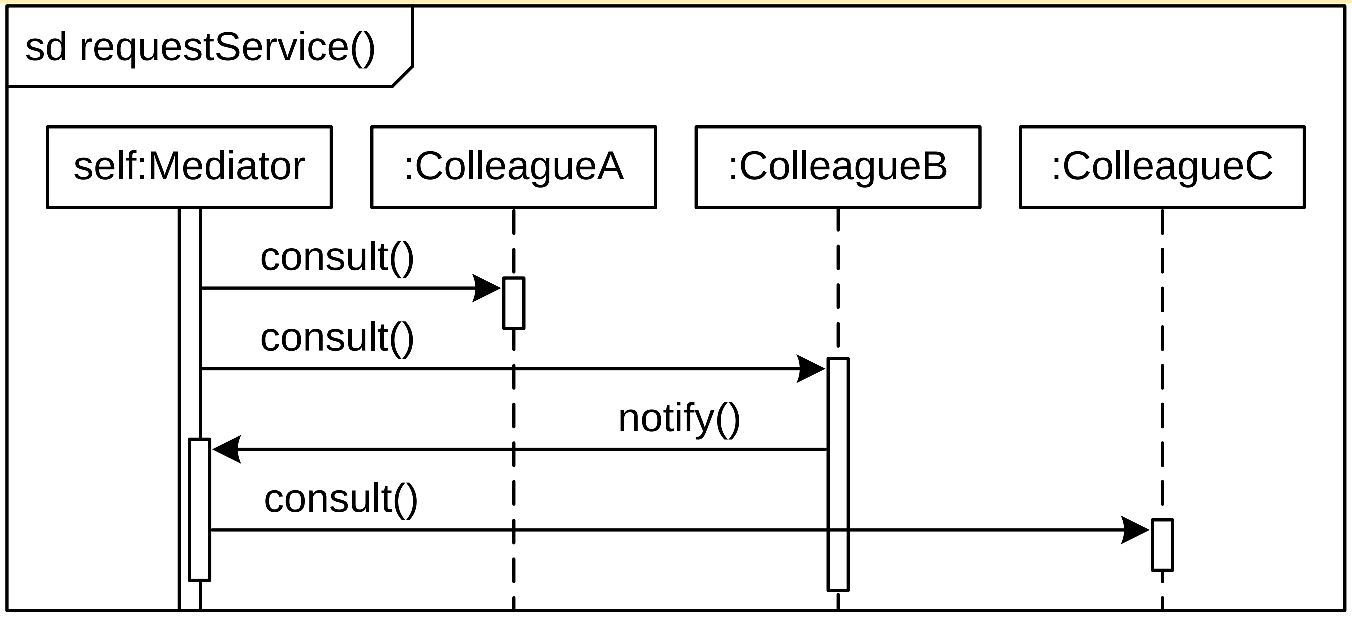
# Mediator Pattern Structure



# Mediator as a Broker



# Mediator Behavior



## When to Use a Mediator

- Use the Mediator pattern when a complex interaction between collaborators must be encapsulated to
  - Decouple collaborators,
  - Centralize control of an interaction, and
  - Simplify the collaborators.
- Using a mediator may compromise performance.

## Mediators, Façades, and Control Styles

- The Façade and Mediator patterns provide means to make control more centralized.
- The Façade and Mediator patterns should be used to move from a dispersed to a delegated style, but not from a delegated to a centralized style.

## Summary

- Broker patterns use a Broker class to facilitate the interaction between a Client and a Supplier.
- The Façade pattern uses a broker (the façade) to provide a simplified interface to a complex sub-system.
- The Mediator pattern uses a broker to encapsulate and control a complex interaction among several suppliers.

## Memento Pattern

## References

- doFactory.com
  - http://www.dofactory.com/Patterns/PatternMemento.aspx
- Marc Clifton’s Blog
  - http://www.marcclifton.com/tabid/99/Default.aspx
- Software Architecture, ETH, Zurich Switzerland
  - http://se.ethz.ch/teaching/ss2005/0050/slides/60_softarch_patterns_6up.pdf

## Intent

- Capture and externalize an object’s state without violating encapsulation.
- Restore the object’s state at some later time.
  - Useful when implementing checkpoints and <u>undo mechanisms</u> that let users back out of tentative operations or recover from errors.
  - Entrusts other objects with the information it needs to revert to a previous state <u>without exposing its internal structure</u> and representations.

## Forces

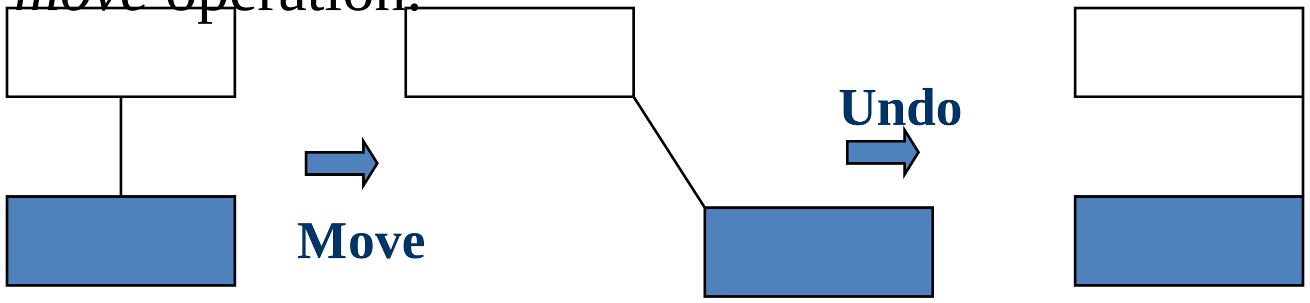
- Application needs to capture states at certain times or at user discretion. May be used for:
  - Undue / redo
  - Log errors or events
  - Backtracking
- Need to preserve encapsulation
  - Don’t share knowledge of state with other objects
- Object owning state may not know when to take state snapshot.

## Motivation

- Many technical processes involve the exploration of some complex data structure.
- Often we need to backtrack when a particular path proves unproductive.
  - Examples are graph algorithms, searching knowledge bases, and text navigation.

## Motivation

- Memento stores a snapshot of another object's internal state, exposure of which would violate encapsulation and compromise the application's reliability and extensibility.
- A graphical editor may encapsulate the connectivity relationships between objects in a class, whose public interface might be insufficient to allow precise reversal of a *move* operation.



## Motivation

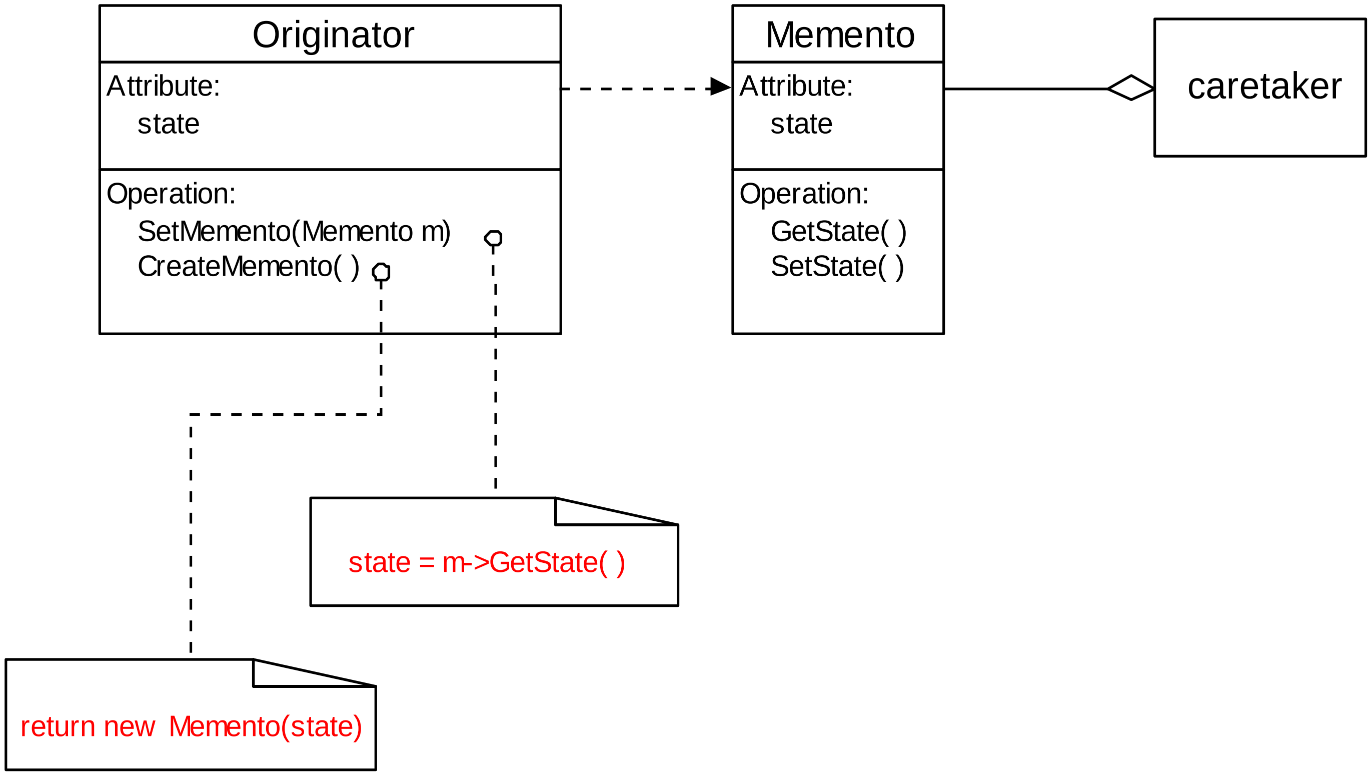
Memento pattern solves this problem as follows:

- The editor requests a memento from the object before executing *move* operation.
- Originator creates and returns a memento.
- During *undo* operation, the editor gives the memento back to the originator.

## Applicability

- Use the Memento pattern when:
  - A snapshot of an object's state must be saved so that it can be restored later, and
  - direct access to the state would expose implementation details and break encapsulation.

# Structure


Originator
Attribute:
state
Operation:
SetMemento(Memento m)
CreateMemento( )
Memento
Attribute:
state
Operation:
GetState( )
SetState( )
caretaker
state = m->GetState( )
return new Memento(state)

## Participants

- Memento
  - – Stores internal state of the Originator object. Originator decides how much.
  - – Protects against access by objects other than the originator.
  - – Mementos have two interfaces:
    - Caretaker sees a narrow interface.
    - Originator sees a wide interface.

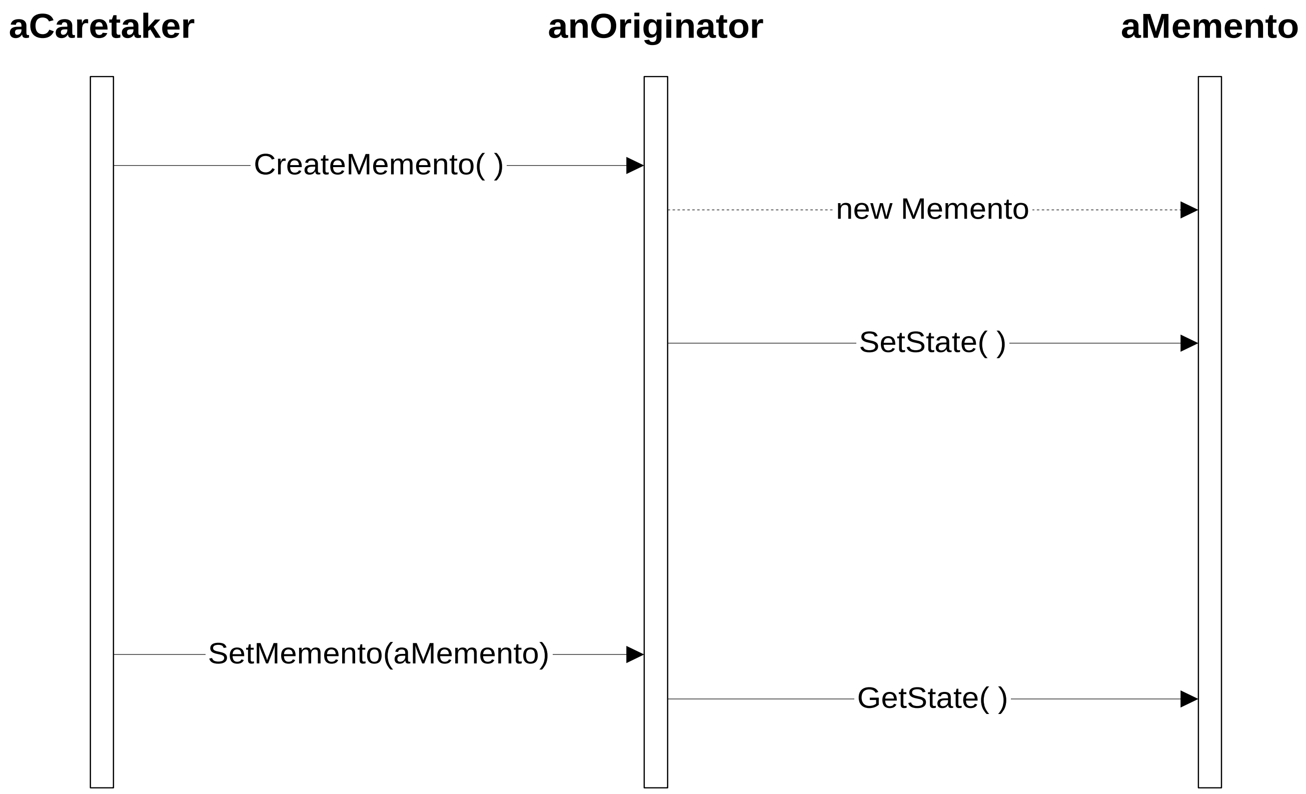
## Participants (continued)

- Originator
  - Creates a memento containing a snapshot of its current internal state.
  - Uses the memento to restore its internal state.

## Caretaker

- Is responsible for the memento’s safekeeping.
- Never operates on or examines the contents of a memento.

# Event Trace


aCaretaker
anOriginator
aMemento
CreateMemento( )
new Memento
SetState( )
SetMemento(aMemento)
GetState( )

## Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

## Consequences

- Memento has several consequences:
  - Memento avoids exposing information that only an originator should manage, but for simplicity should be stored outside the originator.
  - Having clients manage the state they ask for simplifies the originator.

## Consequences (continued)

- Using mementos may be expensive, due to copying of large amounts of state or frequent creation of mementos.
- A caretaker is responsible for deleting the mementos it cares for.
- A caretaker may incur large storage costs when it stores mementos.

## Implementation

- When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just incremental changes to originator's state.

## Known Uses

- ***Memento*** is a 2000 film about Leonard Shelby and his quest to revenge the brutal murder of his wife. Though Leonard is hampered with short-term memory loss, he uses notes and tatoos to compile .

"A new classic among thrillers!"
Peter Travers, Rolling Stone
GUY PEARCE CARRIE-ANNE MOSS JOE PANTOLIANO
MEMENTO
"A masterpiece!"
Mike D' Angelo, Time Out
DVD VIDEO

## Known Use of Pattern

- Dylan language uses memento to provide iterators for its collection facility.
  - Dylan is a dynamic object oriented language using the functional style.
  - Development started by Apple, but subsequently moved to open source.

## Related Patterns

- Command

  Commands can use mementos to maintain state for undo mechanisms.

- Iterator

  Mementos can be used for iteration.

## Observer Pattern

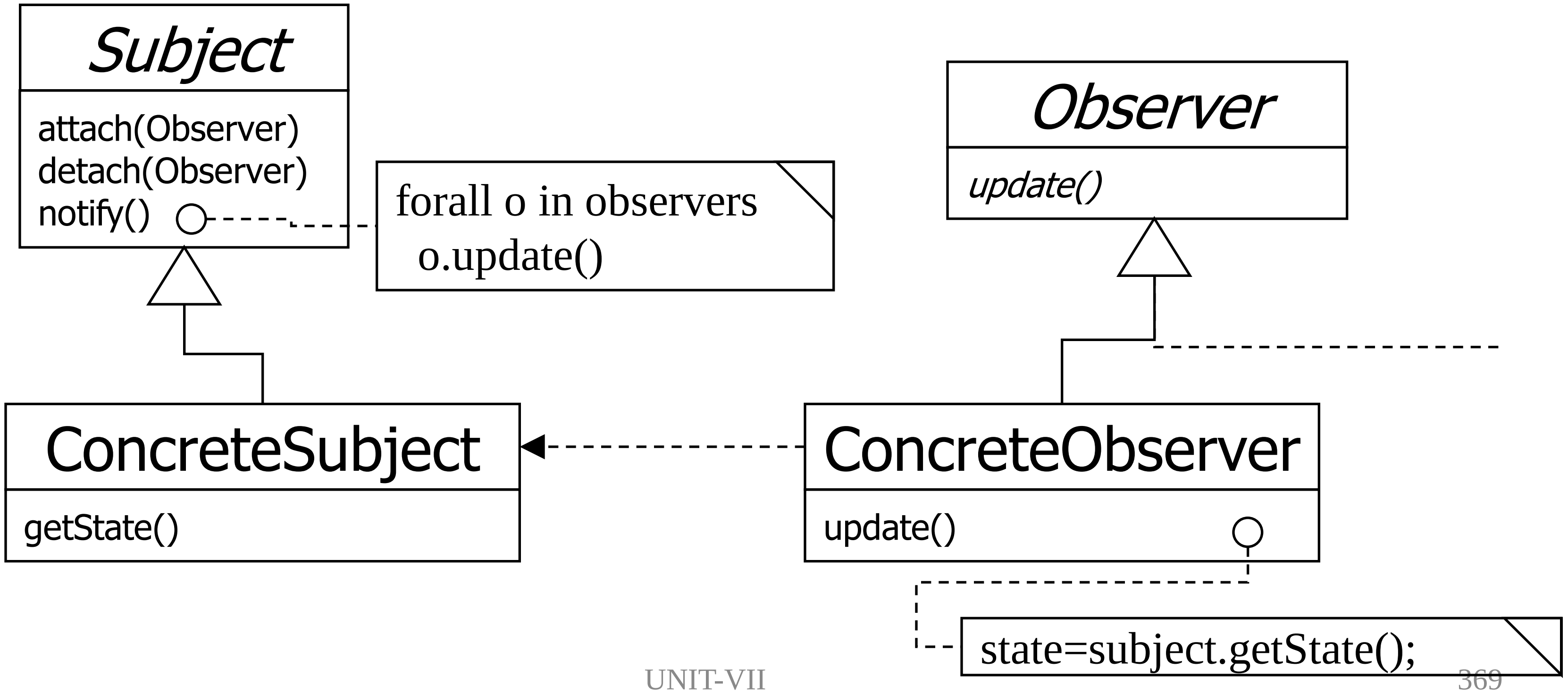
- Define a one-to-many dependency, all the dependents are notified and updated automatically
- The interaction is known as **publish-subscribe** or **subscribe-notify**
- Avoiding observer-specific update protocol: **pull model** vs. **push model**
- Other consequences and open issues

## Observer Pattern

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Key forces:
  - There may be many observers
  - Each observer may react differently to the same notification
  - The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

# Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same “data”
- aka “Publish and subscribe” mechanism
- Choice of “push” or “pull” notification styles



# Observer: Encapsulating Control Flow

**Name:** Observer design pattern

**Problem description:**

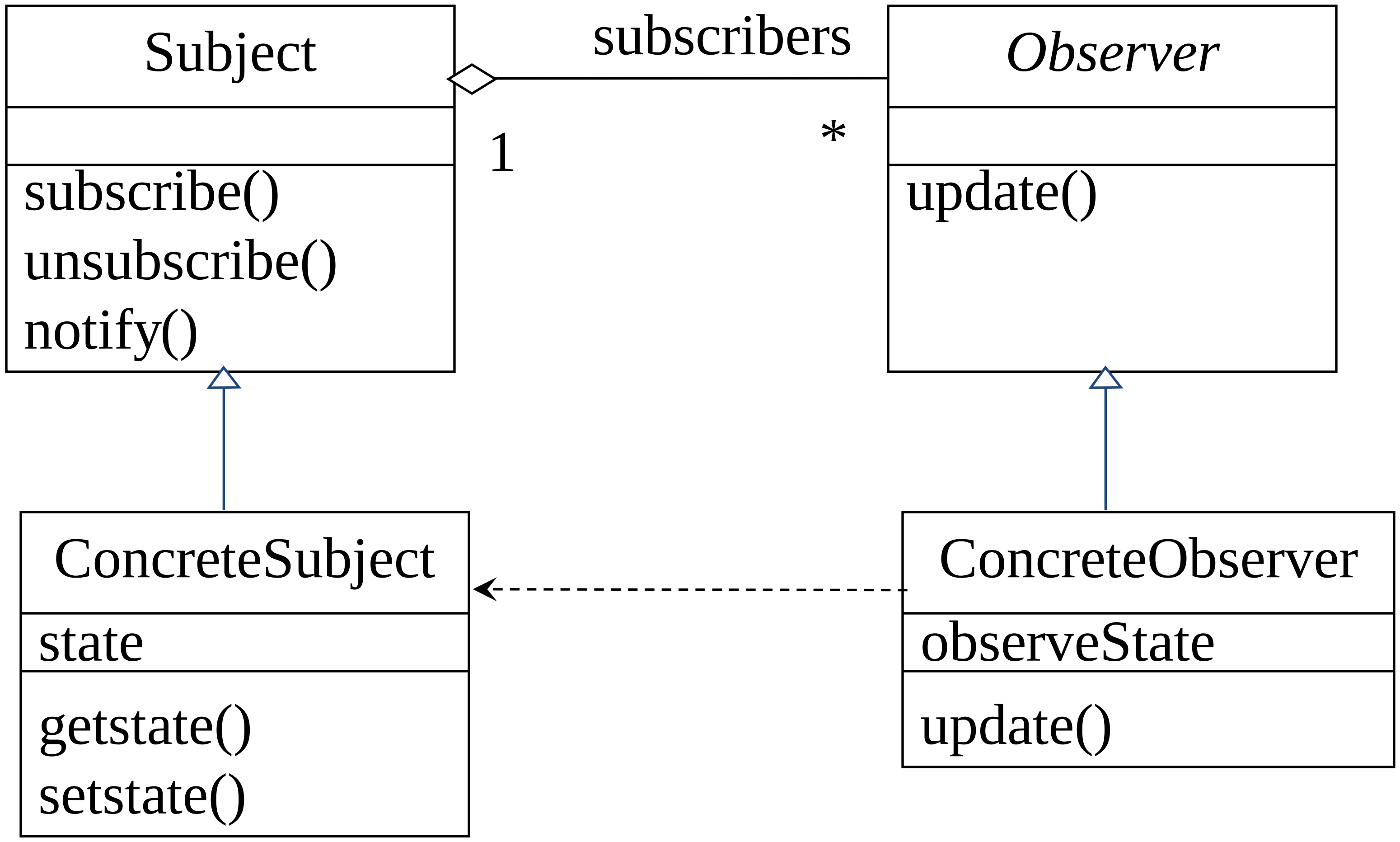
Maintains consistency across state of one Subject and many Observers.

**Solution:**

A Subject has a primary function to maintain some state (e.g., a data structure). One or more Observers use this state, which introduces redundancy between the states of Subject and Observer.

Observer invokes the subscribe() method to synchronize the state. Whenever the state changes, Subject invokes its notify() method to iteratively invoke each Observer.update() method.

# Observer: Class Diagram



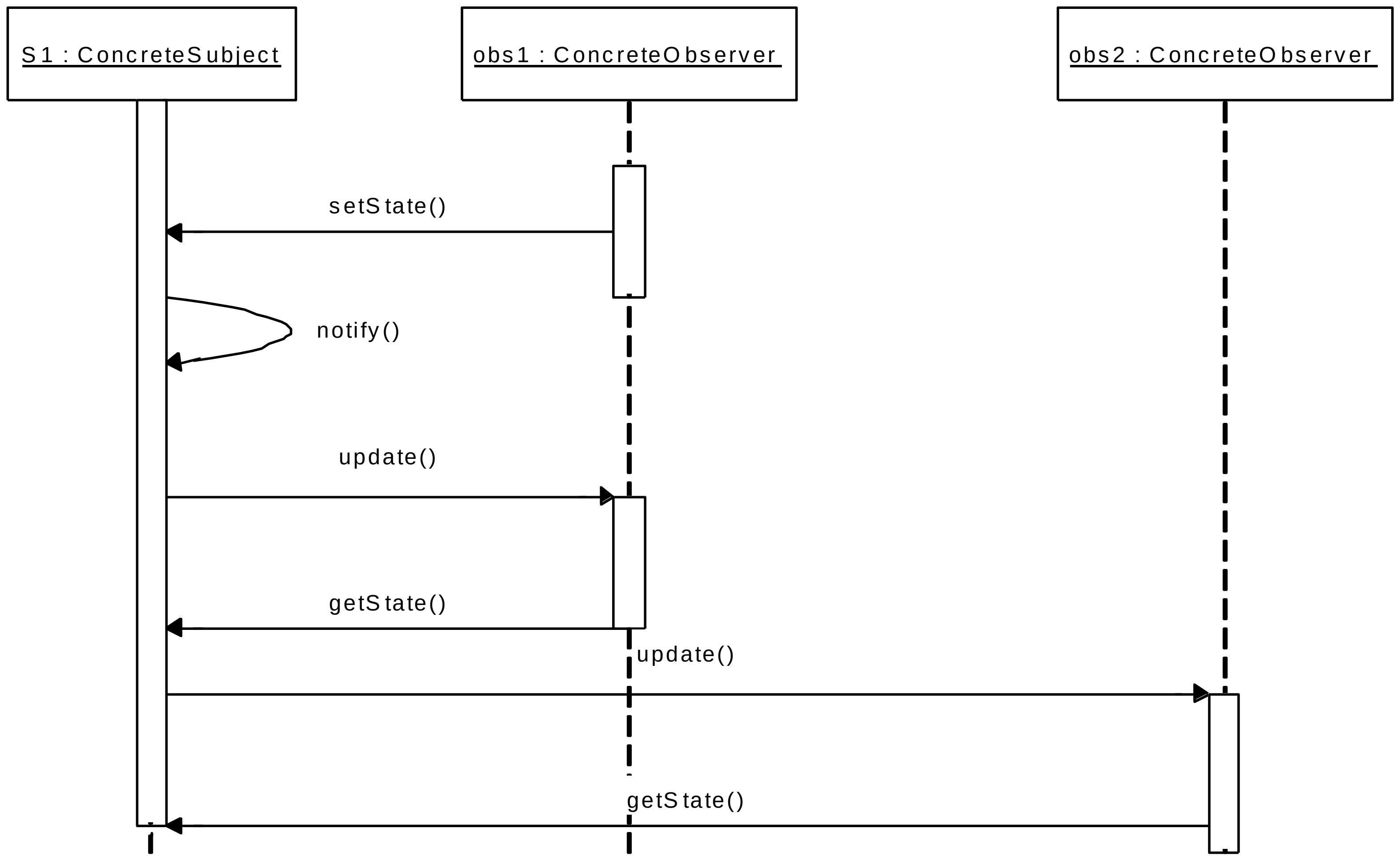
## Observer: Consequences

**Consequences:**

Decouples Subject, which maintains state, from Observers, who make use of the state.

Can result in many spurious broadcasts when the state of Subject changes.

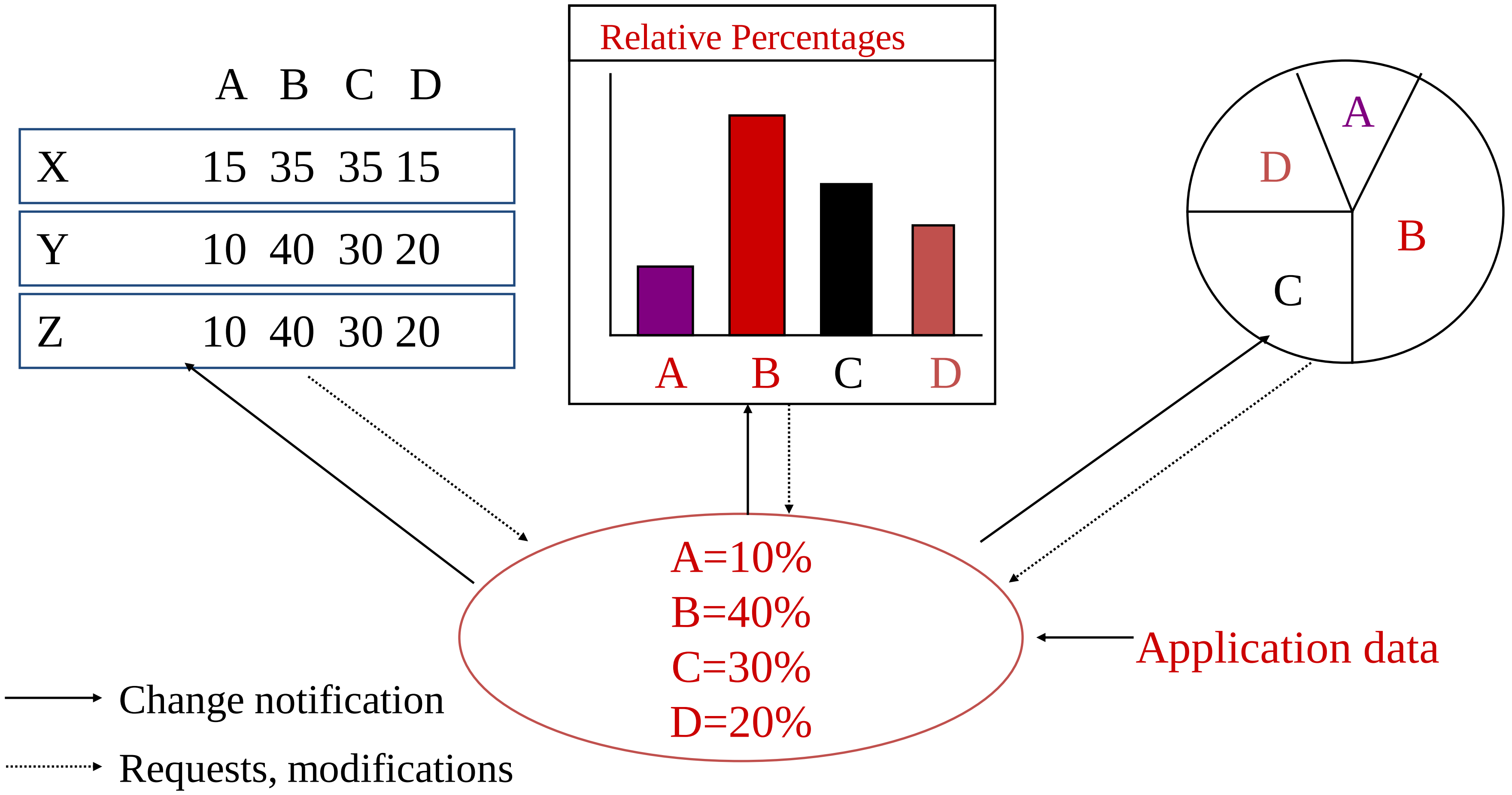
# Collaborations in Observer Pattern



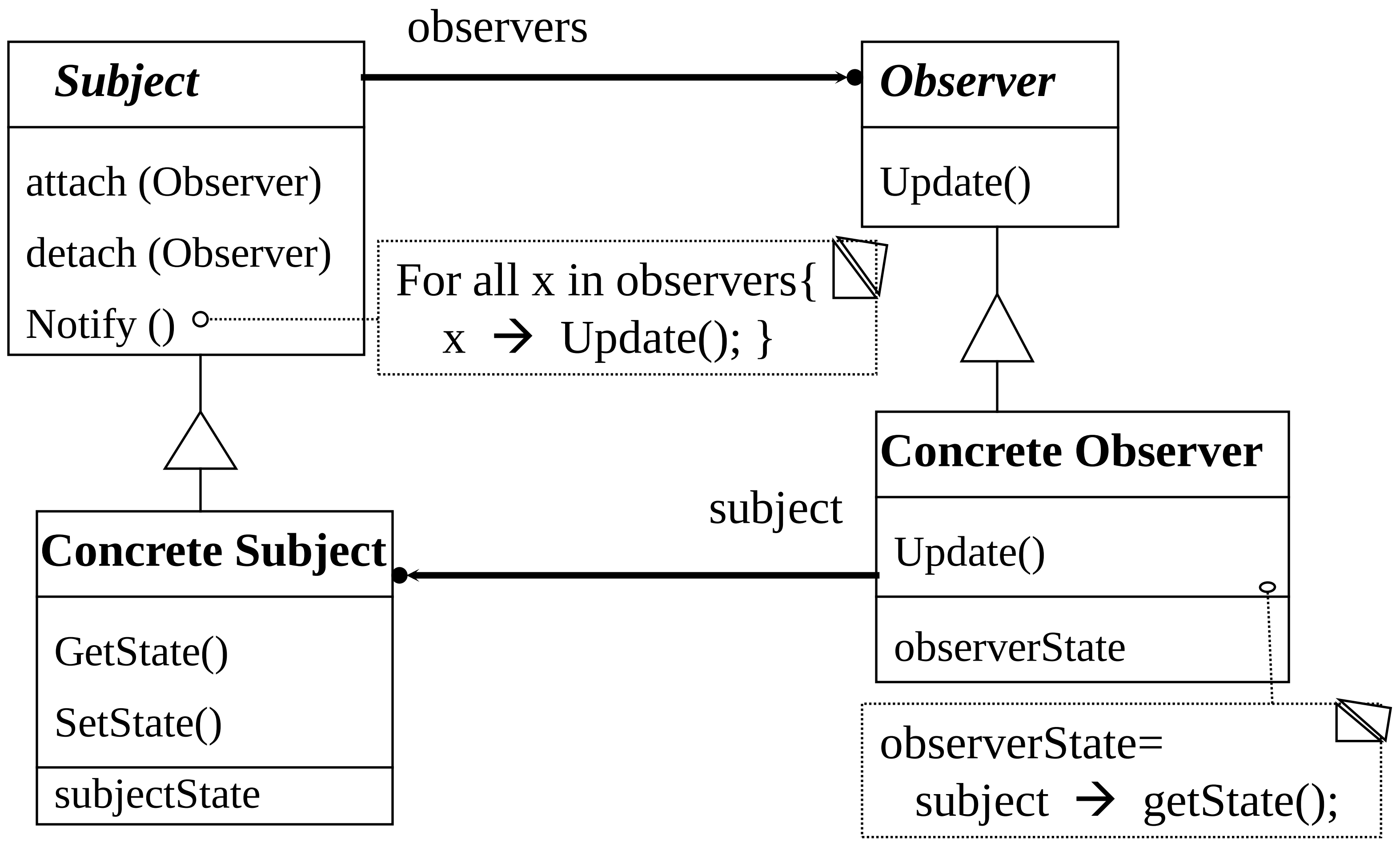
## Observer Pattern [1]

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.

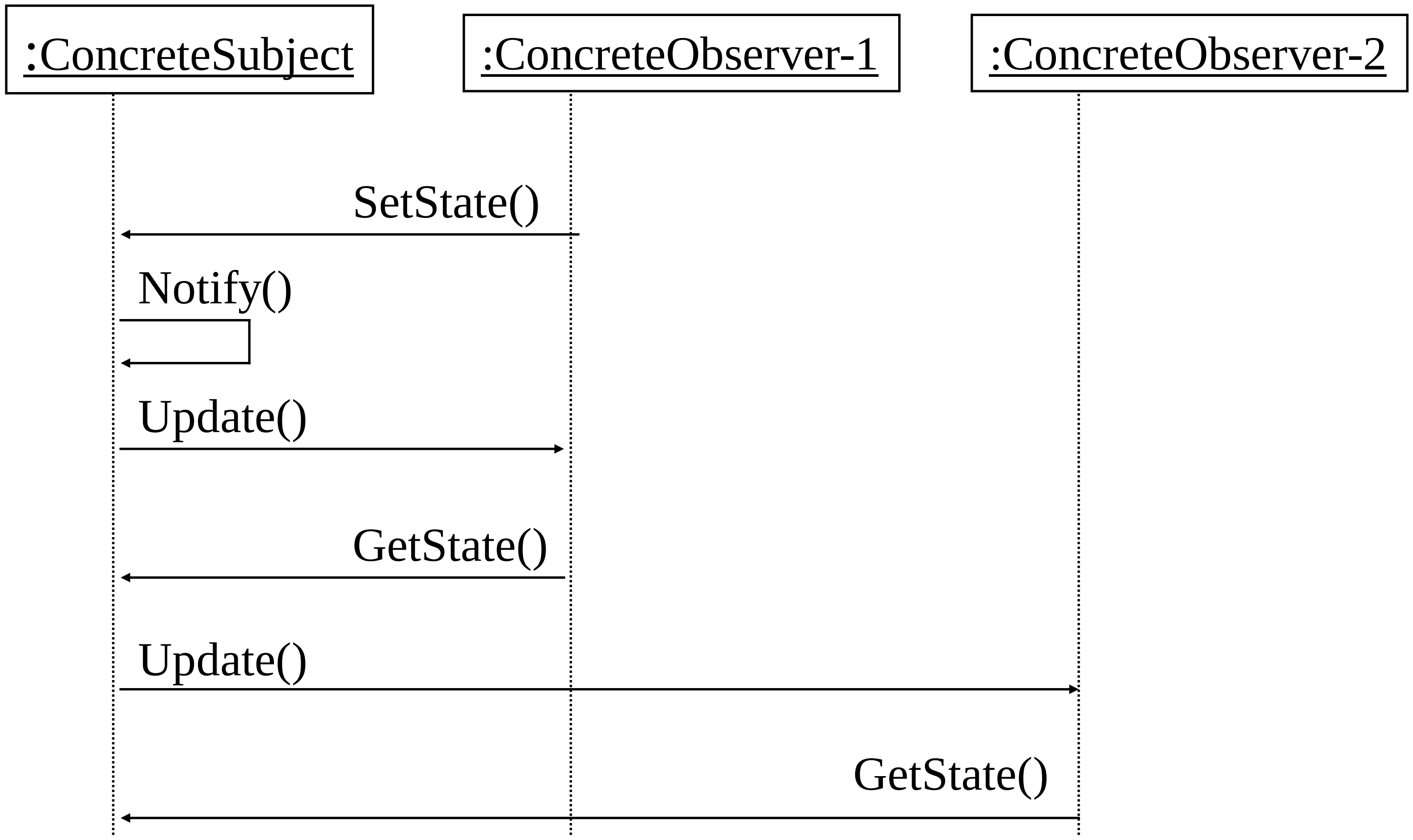
# Observer Pattern [2]



# Observer Pattern [3]



# Class collaboration in Observer



## Observer Pattern: Observer code

```
class Subject;


class observer {
public:
            virtual ~observer;

            virtual void Update (Subject* theChangedSubject)=0;

protected:

             observer ();

};
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

## Observer Pattern: Subject Code [1]

```
class Subject {

public:
        virtual ~Subject;
        virtual   void Attach (observer*);
        virtual   void Detach (observer*) ;
        virtual   void Notify();
protected:
        Subject ();
private:
        List <Observer*> *_observers;
};
```

Abstract class defining the Subject interface.

## Observer Pattern: Subject Code [2]

```
void Subject :: Attach (Observer* o){
        _observers -> Append(o);
}
void Subject :: Detach (Observer* o){
        _observers -> Remove(o);

}
void Subject :: Notify (){
        ListIterator<Observer*> iter(_observers);
        for ( iter.First(); !iter.IsDone(); iter.Next()) {
                        iter.CurrentItem() -> Update(this);

         }
}
```

## Observer Pattern: A Concrete Subject [1]

```
class ClockTimer : public Subject {
public:
            ClockTimer();
            virtual int GetHour();
            virtual int GetMinutes();
            virtual int GetSecond();
            void Tick ();
}
```

## Observer Pattern: A Concrete Subject [2]

```
ClockTimer :: Tick {

    // Update internal time keeping state.
    // gets called on regular intervals by an internal timer.


            Notify();

 }
```

## Observer Pattern: A Concrete Observer [1]

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void  Update(Subject*);
    virtual void  Draw();
private:
    ClockTimer* _subject;
    }
```

**Override Observer operation.**

**Override Widget operation.**

## Observer Pattern: A Concrete Observer [2]

```
DigitalClock ::DigitalClock (ClockTimer* s) {
        _subject = s;
        _subject→Attach(this);
}

DigitalClock ::~DigitalClock() {
        _subject->Detach(this);

}
```

## Observer Pattern: A Concrete Observer [3]

```
void DigitalClock ::Update (subject* theChangedSubject ) {
        If (theChangedSubject == _subject) {
                Draw();
        }
}
```

**Check if this is the clock’s subject.**

```
void DigitalClock ::Draw () {
  int hour = _subject->GetHour();
  int minute = _subject->GeMinute(); // etc.
  // Code for drawing the digital clock.
}
```

## Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;

DigitalClock* digitalClock = new DigitalClock (timer);
```

## Observer Pattern: Consequences

- *Abstract coupling* between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- *Support for broadcast communication*. A subject need not specify the receivers; all interested objects receive the notification.
- *Unexpected updates*: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

## When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.

- *When* a change to one object requires changing others and the number of objects to be changed is not known.

- *When* an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

# Unit-5 part-1
# behavioural patterns part-2(contd)

# STATE Pattern

By :

Raghavendar Japala

## General Description

- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

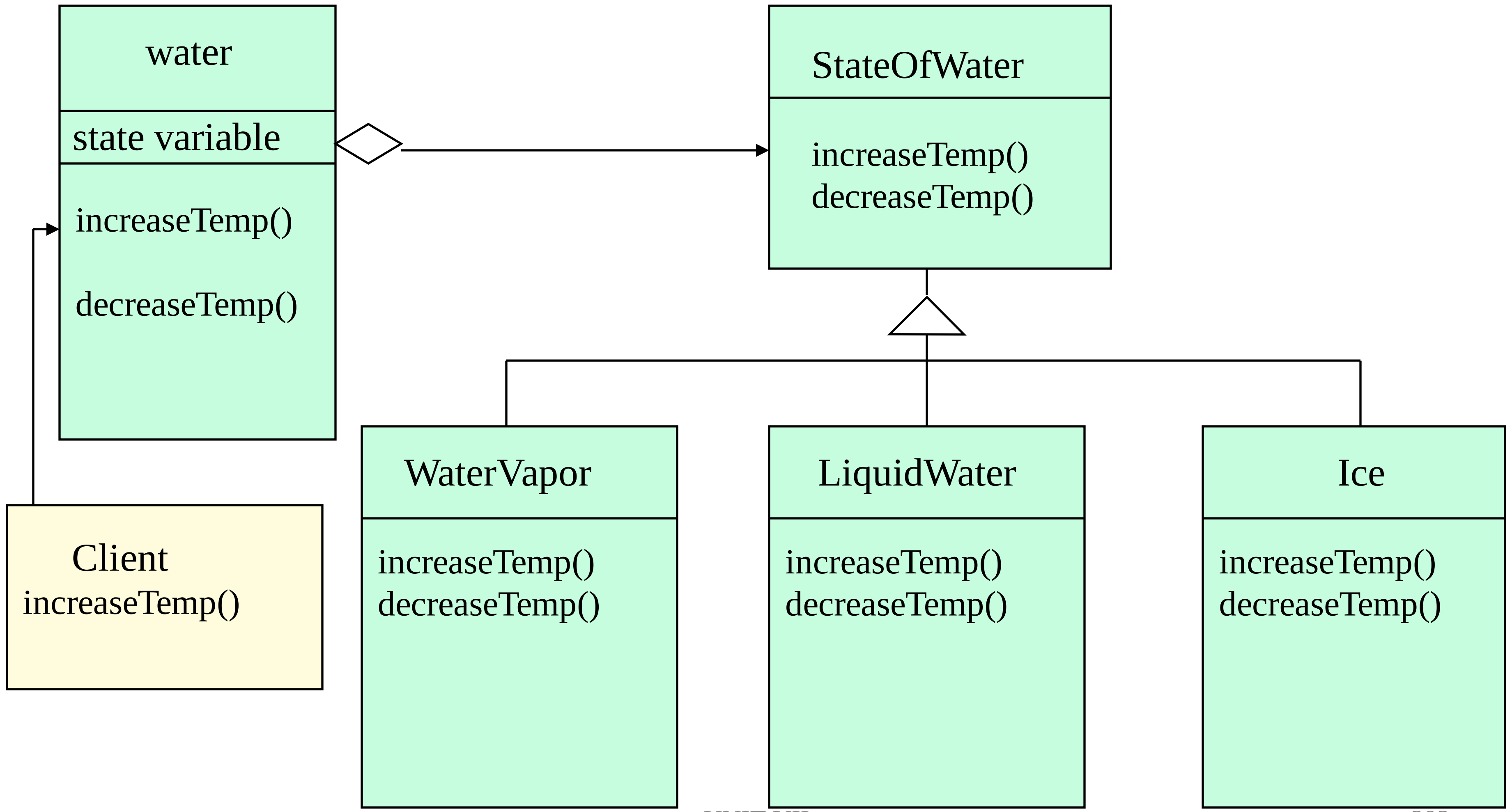
## When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.

- To simplify operations that have large conditional statements that depend on the object’s state.

```
if (myself = happy) then
{
     eatIceCream();
     ….
     }
else if (myself = sad) then
{
     goToPub();
     ….
   }
else if (myself = ecstatic) then
{
     ….
```

# Example I

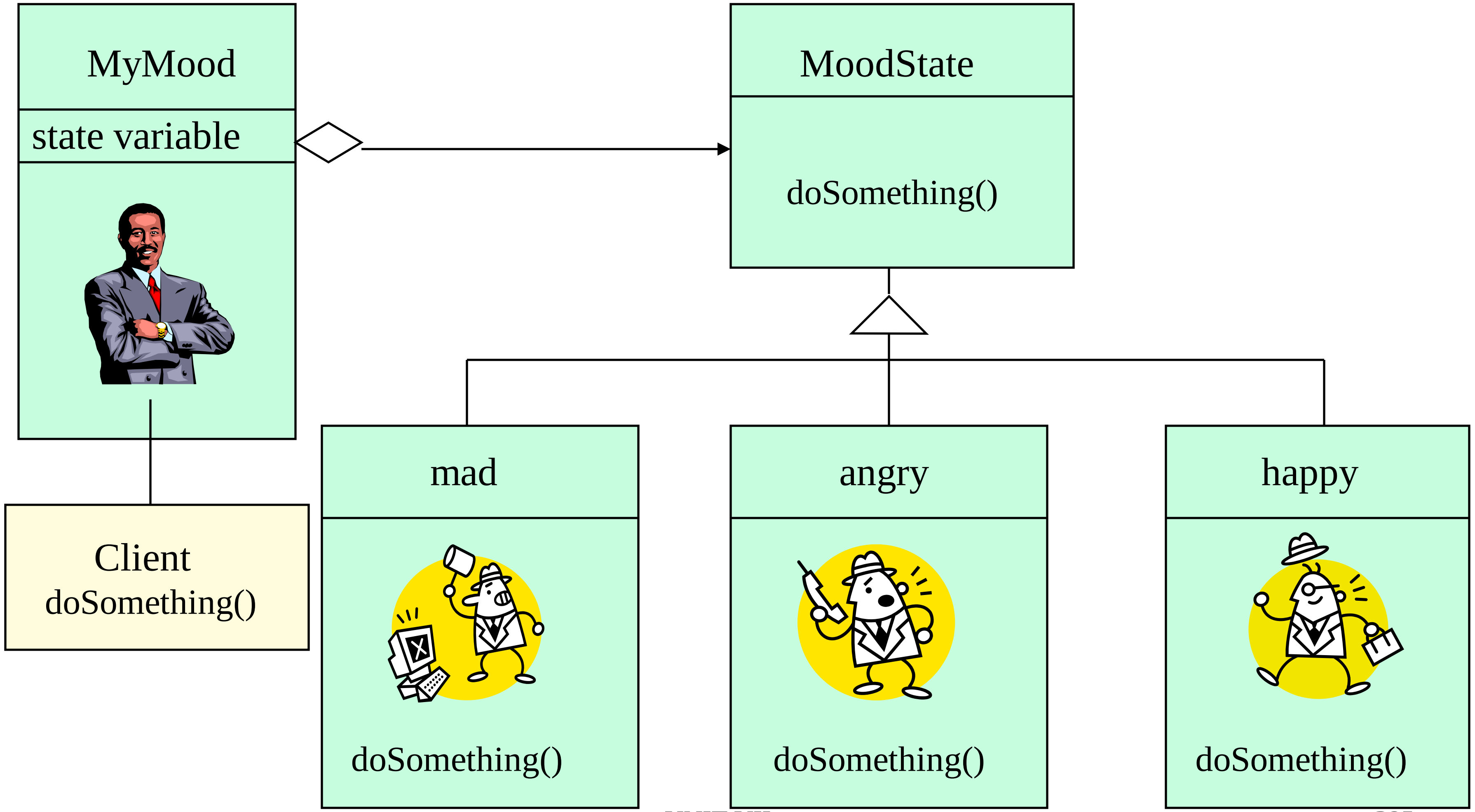

water
state variable
increaseTemp()
decreaseTemp()
StateOfWater
increaseTemp()
decreaseTemp()
Client
increaseTemp()
WaterVapor
increaseTemp()
decreaseTemp()
LiquidWater
increaseTemp()
decreaseTemp()
Ice
increaseTemp()
decreaseTemp()

## How is STATE pattern implemented ?

•"Context" class:
Represents the interface to the outside world.
•"State" abstract class:
Base class which defines the different states of the "state machine".
•"Derived" classes from the State class:
Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

# Example II


MyMood
state variable
MoodState
doSomething()
Client
doSomething()
mad
doSomething()
angry
doSomething()
happy
doSomething()

## Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - ➢ New state and transitions can be added easily by defining new subclasses.
  - ➢ Simplifies maintenance.
- **It makes state transitions explicit.**
  - ➢ Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
- **State objects can be shared.**
  - ➢ Context can share State objects if there are no instance variables.

## Food for thought…

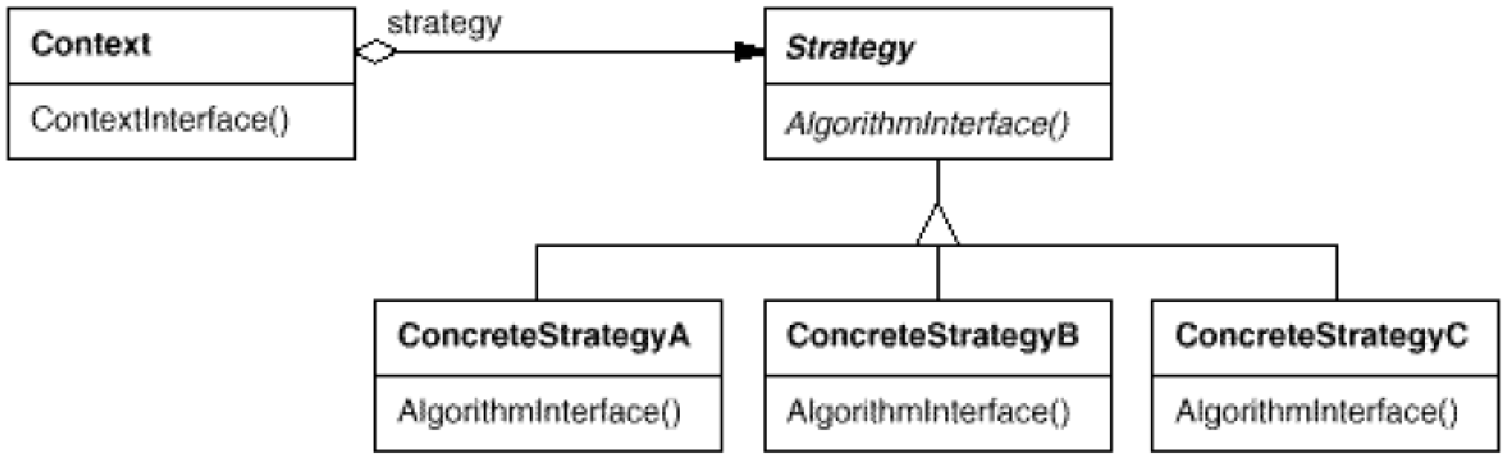
- **To have a monolithic single class or many subclasses ?**
  - ➢ Increases the number of classes and is less compact.
  - ➢ Avoids large conditional statements.

- **Where to define the state transitions ?**
  - ➢ If criteria is fixed, transition can be defined in the context.
  - ➢ More flexible if transition is specified in the State subclass.
  - ➢ Introduces dependencies between subclasses.

- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
  - ➢ First is desirable if the context changes state infrequently.
  - ➢ Later is desirable if the context changes state frequently.

## Pattern: Strategy

*objects that hold alternate algorithms to solve a problem*



## Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?
- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself

- **examples**:
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

## Strategy example: Card player

```
// Strategy hierarchy parent
// (an interface or abstract class)
public interface Strategy {
  public Card getMove();
}

// setting a strategy
player1.setStrategy(new SmartStrategy());

// using a strategy
Card p1move = player1.move();  // uses strategy
```

## Strategy: Encapsulating Algorithms

**Name:** Strategy design pattern

**Problem description:**

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

**Example:**

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

# Strategy: Encapsulating Algorithms

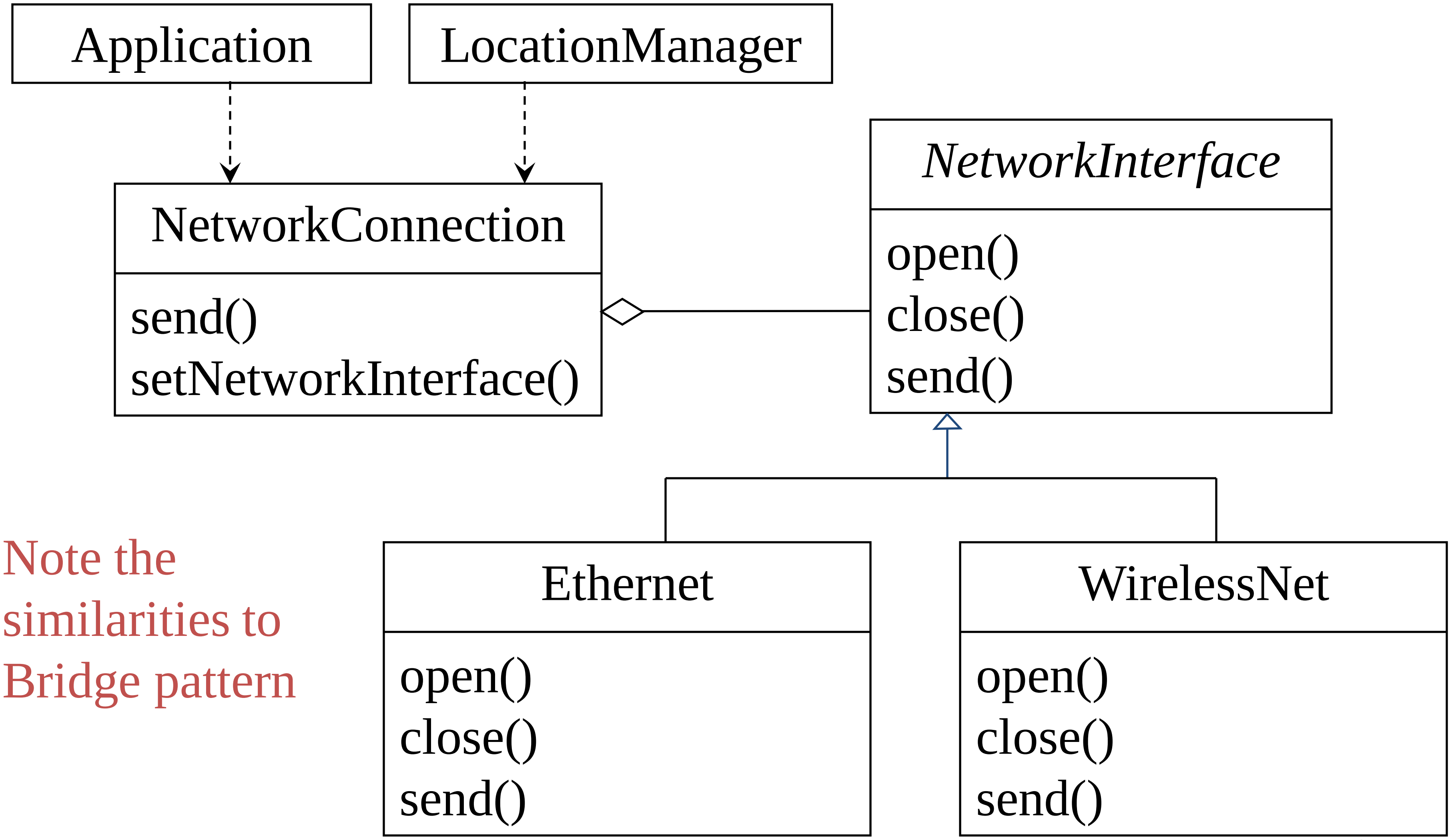
**Solution:**

A Client accesses services provided by a Context.

The Context services are realized using one of several mechanisms, as decided by a Policy object.

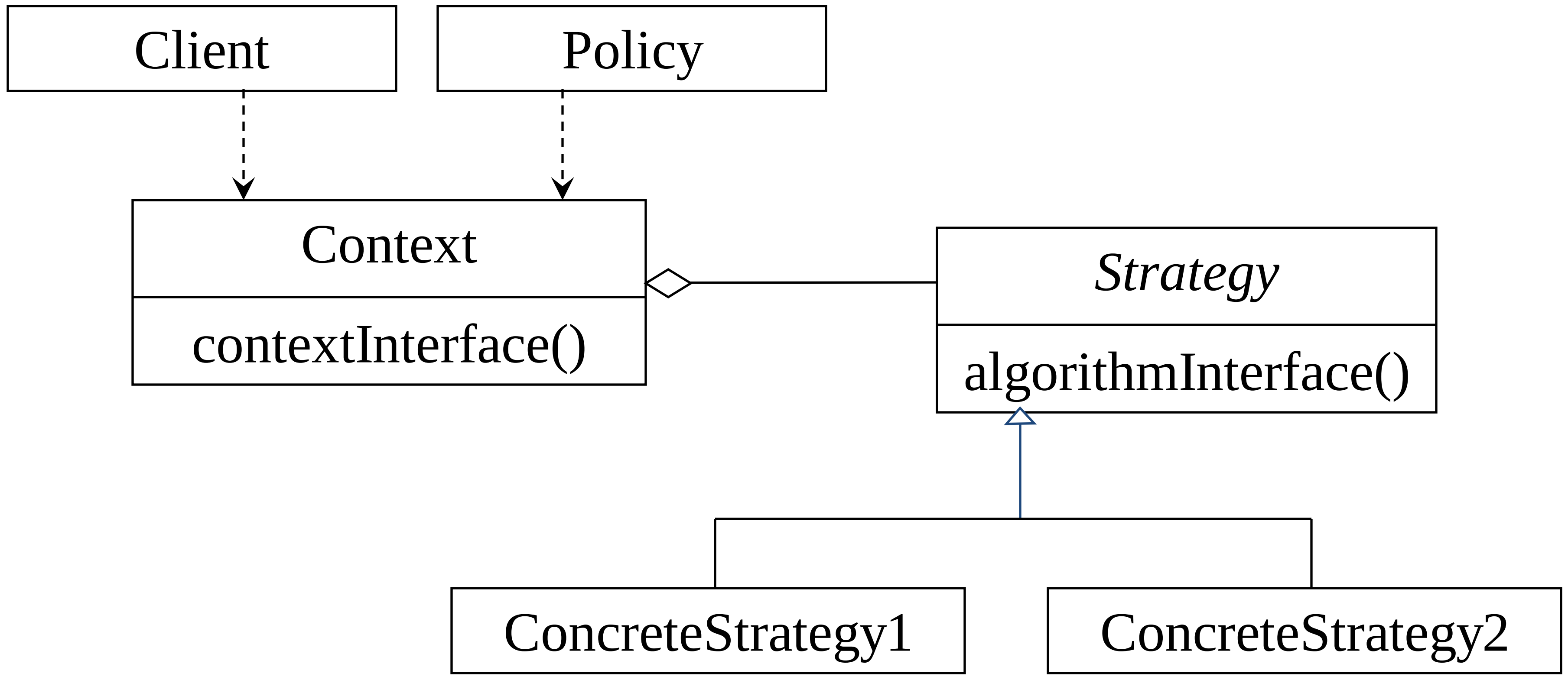
The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

# Strategy Example: Class Diagram for Mobile Computer



Note the similarities to Bridge pattern

# Strategy: Class Diagram



# Strategy: Consequences

**Consequences:**

ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

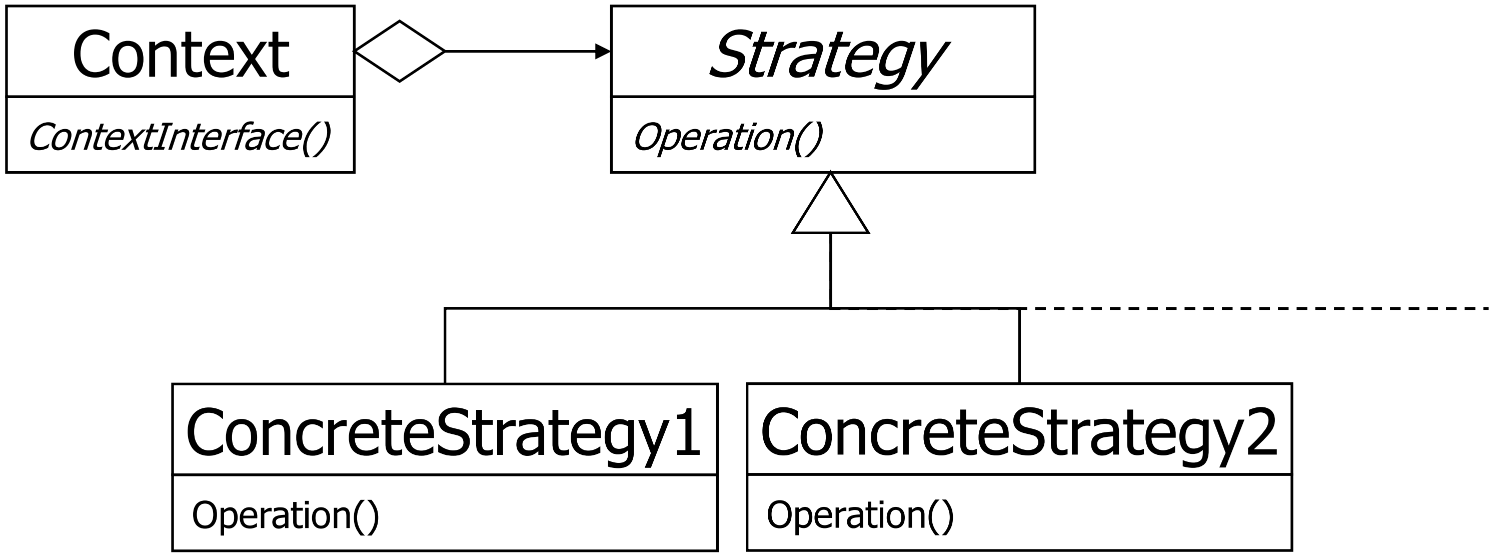
New policy algorithms can be added without modifying Context or Client.

## Strategy

- You want to
  - use different algorithms depending upon the context
  - avoid having to change the context or client
- *Strategy*
  - decouples interface from implementation
  - shields client from implementations
  - Context is not aware which strategy is being used; Client configures the Context
  - strategies can be substituted at runtime
  - example:  interface to wired and wireless networks

## Strategy

- Make algorithms interchangeable---”changing the guts”
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



# Design Patterns & Frameworks

## Chapter 6 – Template Method

Conducted By Raghavendar Japala

## Topics – Template Method

- Introduction to Template Method Design Pattern
- Structure of Template Method
- Generic Class and Concrete Class
- Plotter class and Plotter Function Class

## Introduction

The DBAnimationApplet illustrates the use of an **abstract class** that serves as a template for classes with shared functionality.

An abstract class contains behavior that is common to all its subclasses. This behavior is encapsulated in nonabstract methods, which may even be declared ***final*** to prevent any modification. This action ensures that all subclasses will inherit the same common behavior and its implementation.

The abstract methods in such templates ensure the interface of the subclasses and require that context specific behavior be implemented for each concrete subclass.

## Hook Method and Template Method

The abstract method paintFrame() acts as a placeholder for the behavior that is implemented differently for each specific context.

We call such methods, *hook* methods, upon which context specific behavior may be hung, or implemented.

The paintFrame() hook is placed within the method update(), which is common to all concrete animation applets. Methods containing hooks are called *template* methods.

## Hook Method and Template Method (Con’t)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

## Frozen Spots and Hot Spots

A template method uses hook methods to define a common behavior.

Template method describes the fixed behaviors of a generic class, which are sometimes called **frozen spots**.

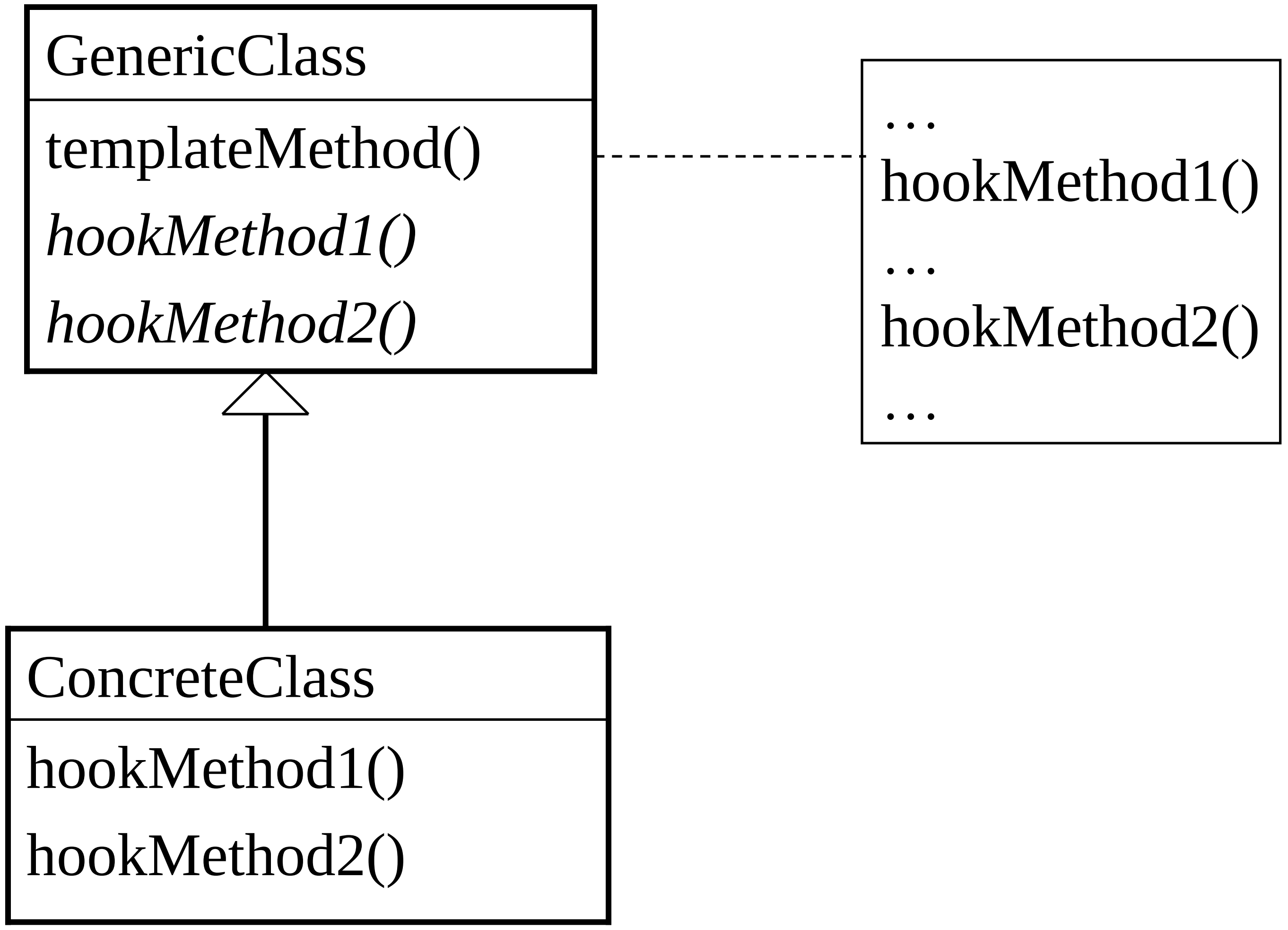
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

## Hook Method and Template Method (Con’t)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

## Structure of the Template Method Design Pattern



## Structure of the Template Method Design Pattern (Con’t)

**GenericClass** (e.g., DBAnimationApplet), which defines abstract hook methods (e.g., paintFrame()) that concrete subclasses (e.g., Bouncing-Ball2) override to implement steps of an algorithm and implements a template method (e.g., update()) that defines the skeleton of an algorithm by calling the hook methods;

**ConcreteClass** (e.g., Bouncing-Ball2) which implements the hook methods (e.g., paintFrame()) to carry out subclass specific steps of the algorithm defined in the template method.

## Structure of the Template Method Design Pattern (Con’t)

In the Template Method design pattern, *hook methods* **do not** have to be abstract.

The generic class may provide default implementations for the hook methods.

Thus the subclasses have the option of overriding the hook methods or using the default implementation.

The initAnimator() method in DBAnimationApplet is a nonabstract hook method with a default implementation.

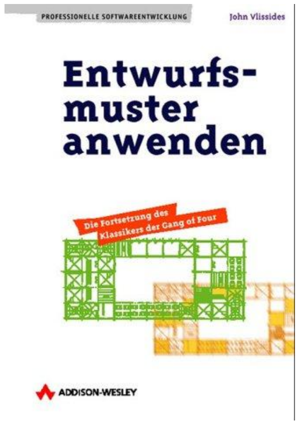
The init() method is another template method.

## A Generic Function Plotter

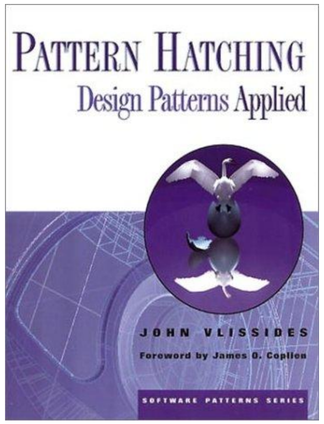
The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

A concrete plotter PlotSine will be implemented to plot the function

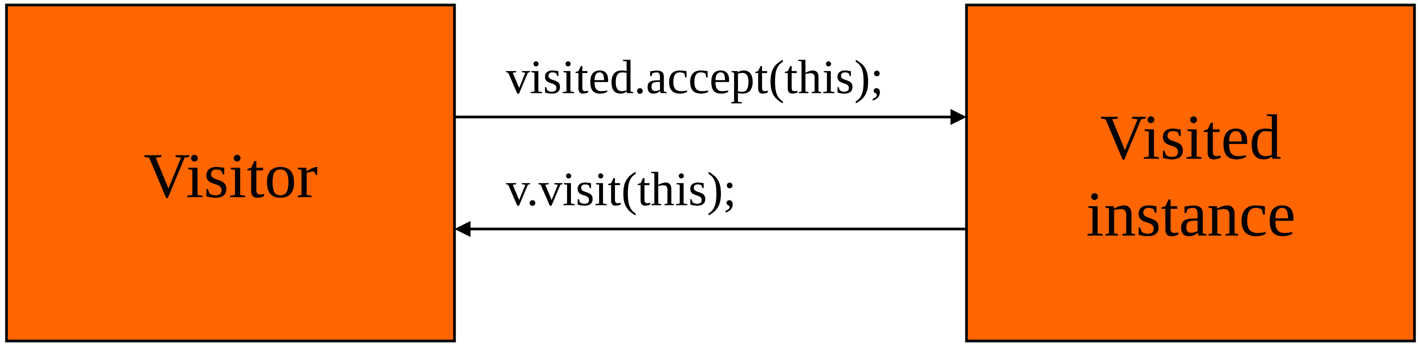
$$y = \sin x$$

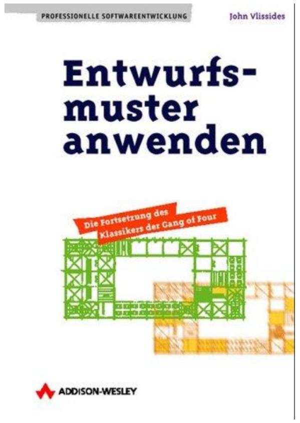


# Pattern Hatching



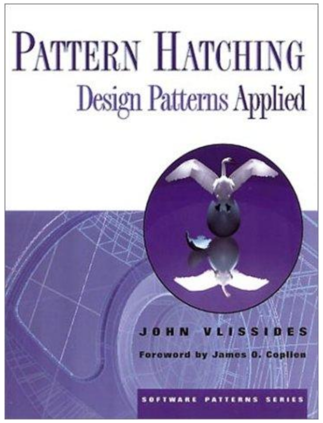
## Visitor pattern




PROFESSIONELLE SOFTWAREENTWICKLUNG
John Vlissides
Entwurfs-muster anwenden
Die Fortsetzung des Klassikers der Gang of Four
ADDISON-WESLEY

# Pattern Hatching

## Visitor Pattern



```
Class Visitor {
public:
        Visitor();
        void visit(File*);
        void visit(Directory*);
        void visit (Link*);
};
```

```
void Visitor::visit (File* f)
        {f->streamOut(cout);}

void Visitor::visit (Directory* d)
        {cerr << “no printout for a directory”;}

void Visitor::visit (Link* l)
        {l->getSubject()->accept(*this);}
```

```
Visitor cat;
node->accept(cat);
```

```
void File::accept (Visitor& v)           {v.visit(this);}
void Directory::accept (Visitor& v)      {v.visit(this);}
void Link::accept (Visitor& v)           {v.visit(this);}
```

## References

- Java API pages
  - http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html
  - http://java.sun.com/j2se/1.4.2/docs/api/java/util/Iterator.html
  - http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Container.html
  - http://java.sun.com/j2se/1.4.2/docs/api/java/awt/LayoutManager.html
  - http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JScrollPane.html

- Cunningham & Cunningham OO Consultancy, Inc.
  - http://c2.com/cgi/wiki?IteratorPattern
  - http://c2.com/cgi/wiki?DecoratorPattern
  - http://c2.com/cgi/wiki?CompositePattern

- Design Patterns Java Companion
  - http://www.patterndepot.com/put/8/JavaPatterns.htm

# Unit-5 part-1
# behavioural patterns part-2(contd)

| S. No | TOPIC | PPT Slides |
|---|---|---|
| 1 | What to Expect from Design Patterns | |
| 2 | A Brief History | |
| 3 | The Pattern Community An Invention | |
| 4 | A Parting Thought. | |
| 5 | Design Patterns advantageous and disadvantageous | |
| 6 | Scope of the design patterns in application programming. | |
| 7 | Review Unit-V | |

## What to Expect from Design Patterns

- A Common Design Vocabulary

- A Documentation and Learning Aid

- An Adjunct to Existing Methods

- A Target for Refactoring

## A common design vocabulary

1. Studies of expert programmers for conventional languages have shown that knowledge and experience isn’t organized simply around syntax but in larger conceptual structures such as algorithms, data structures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particular goal [SE84].

2. Designers probably don’t think about the notation they are using for recording the designing as much as they try to match the current design situation against plans, data structures, and idioms they have learned in the past.

3. Computer scientists name and catalog algorithms and data structures, but we don’t often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.

## A document and learning aid

1. Knowing the design patterns makes it easier to understand existing systems.
2. Most large object-oriented systems use this design patterns people learning object-oriented programming often complain that the systems they are working with use inheritance in convoluted ways and that it is difficult to follow the flow of control.
3. In large part this  is because they do not understand the design patterns in the system learning these design patterns will help you understand existing  object-oriented system.

# An adjacent to existing methods

1. Object-oriented design methods are supposed to promote good design, to teach new designers how to design well, and standardize the way designs are developed.

2. A design method typically defines a set of notations (usually graphical) for modeling various aspects of design along with a set of rules that govern how and when to use each notation.

3. Design methods usually describe problems that occur in a design, how to resolve them and how to evaluate design. But then have not been able to capture the experience of expert designers.

4. A full fledged design method requires more kinds of patterns than just design patterns there can also be analysis patterns, user interface design patterns, or performance tuning patterns but the design patterns are an essential part, one that’s been missing until now.

## A target for refactoring

1. One of the problems in developing reusable software is that it often has to be recognized or refactored [OJ90].

2. Design patterns help you determine how to recognize a design and they can reduce a amount of refactoring need to later.

3. The life cycle of object-oriented software has several faces. Brain Foote identifies these phases as the prototyping expansionary, and consolidating phases [Foo92]

## Design Patterns Applied

Example: An Hierarchical File System

Tree Structure → Composite

Patterns Overview

Symbolic Links → Proxy

Extending Functionality → Visitor

Single User Protection → Template Method

Multi User Protection → Singleton

User and Groups → Mediator

## A Brief History of Design Patterns

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns

## What’s In a Design Pattern--1994

- The GOF book describes a pattern using the following four attributes:
  - The ***name*** to describes the pattern, its solutions and consequences in a word or two
  - The ***problem*** describes when to apply the pattern
  - The ***solution*** describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The ***consequences*** are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk

## What's In a Design Pattern - 2002

- Grand's book is the latest offering in the field and is very Java centric.  He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution
  - Solution—general purpose solution to the problem
  - Implementation—important considerations when using a solution
  - Consequences—implications, good or bad, of using a solution
  - Java API usage—examples from the core Java API
  - Code example—self explanatory
  - Related patterns—self explanatory

## Grand’s Classifications of Design Pattern

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

The Pattern Community An Invention

❑Christopher Alexander is the architect who first
studied
Patterns in buildings and communities and developed
A PATTERN LANGUAGE for generating them.

❑His work has inspired time and again. So it’s fitting worth
while To compare our work to his.

❑Then we’ll look at other’s work in software-related patterns.

## What's In a Design Pattern--1994

- The GOF book describes a pattern using the following four attributes:
  - The ***name*** to describes the pattern, its solutions and consequences in a word or two
  - The ***problem*** describes when to apply the pattern
  - The ***solution*** describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The ***consequences*** are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk

## What's In a Design Pattern - 2002

- Grand's book is the latest offering in the field and is very Java centric. He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution
  - Solution—general purpose solution to the problem
  - Implementation—important considerations when using a solution
  - Consequences—implications, good or bad, of using a solution
  - Java API usage—examples from the core Java API
  - Code example—self explanatory
  - Related patterns—self explanatory

Alexander’s Pattern Languages
There are many ways  in which our work is like Alexander's

Both are based on observing existing systems and looking for patterns in them.

Both have templates for describing patterns although
our templates are quite different)..

But there are just as many ways in which our work different.

➢People have been making buildings for thousands of years, and there are many

classic examples to draw upon. We have been making Software systems for a

Relatively short time, and few are considered classics.

➢Alexander gives an order in which his patterns should be used; we have not.

- Alexander’s patterns emphasize the problems they adderss ,
- where as design patterns describes the solutions in more detail.

- Alexander claims his patterns will generate complete buildings.

We do not claim that our patterns will  generate complete programs.

When Alexander claims you can design a house simply applying his patterns one after

Another ,he has goals  similar to those of object-oriented design methodologies who

Gives step-by-step rules for design,

In fact ,we think it’s unlikely that there will ever be a compete pattern language for soft
-ware.

But certainly possible to make one that is more complte.

A Parting Thought.

The best designs will use many design patterns that dovetail

And intertwine to produce a greater whole.
As Alexander says:
It is possible to make buildings by stringing together pattern’s,
In a rather loose way,
A building made like this , is an assembly of patterns. it is not

Dense.
It is not profound. but it is also possible to put pattern’s together

In such a way that many patterns overlap in the same physical

Space: the building is very dense; it has many meaning captured
In a small space; and through this density, it becomes profound.