

11. Class Time Table**12. Individual Time Table**

13. Lecture schedule with methodology being used/adopted along with tutorial, assignment and NPTEL class schedule

Micro Plan with dates and closure report

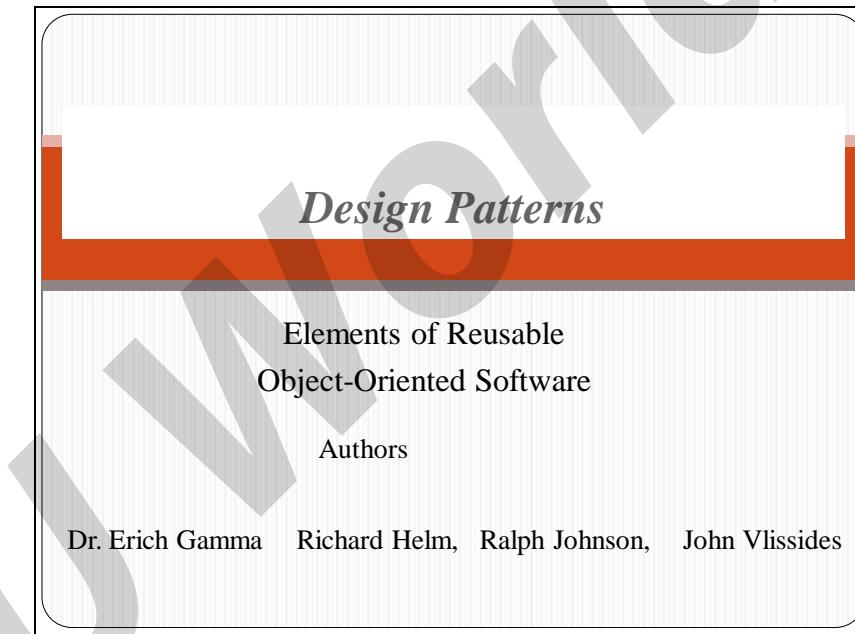
SNO	Unit No	Date	Topic Covered	No of Periods	Teaching aids used LCD/OHP/BB
1.	UNIT-1		What is a Design Pattern?	1	BB
2.			Describing Design Patterns,	1	BB
3.			Organizing The Design Catalog	1	BB
4.			How Design Patterns Solve Design Problems	1	BB
5.			How To Select A Design Pattern,	1	BB
6.			How To Use A Design Pattern	1	BB
7.			Design Patterns in Smalltalk MVC	1	BB

8.			The Catalog Of Design Patterns	1	BB/OHP
9.			Tutorial class /(NPTEL CLASS)	1	BB
10.			Assignment test	1	
11.	UNIT-2		Designing a Document Editor	1	BB
12.			Design problems, Document Structure	1	BB
13.			Formatting , Embellishing the User Interface	1	BB
14.			Supporting , Multiple Look and Feel Standards	1	BB/OHP
15.			Supporting Multiple Window Systems	1	BB
16.			User Operations Spell Checking And Hyphenation , Summary	1	BB
17.			Tutorial class/(NPTEL CLASS)	1	BB
18.			Assignment test	1	
19.	UNIT-3		Creational patterns	1	BB
20.			Abstract Factory	1	BB/OHP
21.			Builder, Factory Method	1	BB
22.			Prototype	1	BB
23.			Singleton	1	BB
24.			Discussion Of Creational Patterns	1	BB
25.			Tutorial class/(NPTEL CLASS)	1	BB
26.	UNIT-5		Structural pattern part-I	1	BB
27.			Adapter	1	BB
28.			Bridge	1	BB
29.			Composite	1	BB/OHP
30.			Tutorial class/(NPTEL CLASS)	1	BB
31.			Assignment test	1	

32.	UNIT-5	Structural pattern part-II	1	BB
33.		Decorator	1	BB
34.		Acade	1	BB
35.		Flyweight	1	BB
36.		Proxy.	1	BB/OHP
37.		Tutorial class/(NPTEL CLASS)	1	BB
38.		Assignment test	1	
39.	UNIT-6	Behavioral patterns part-I	1	BB
40.		Chain Of Responsibility	1	BB
41.		Command,	1	BB
42.		Interpreter	1	BB
43.		Iterator	1	BB/OHP
44.		Tutorial class/(NPTEL CLASS)	1	BB
45.		Assignment test	1	
46.	UNIT-7	Behavioral patterns part-II	1	BB
47.		Mediator	1	BB
48.		Memento	1	BB
49.		State , Strategy	1	BB
50.		Template, Method	1	BB
51.		Visitor,	1	BB
52.		Discussion of Behavioral patterns	1	BB
53.		Tutorial class/(NPTEL CLASS)	1	BB
54.		Assignment test	1	
55.	UNIT-8	What To Expect From Design Patterns	1	BB
56.		A Brief History	1	BB
57.		The Pattern Community	1	BB

58.			An Invitation	1	BB
59.			A parting Thought	1	BB/OHP
60.			Tutorial class/(NPTEL CLASS)	1	BB
61.			Assignment test	1	BB
62.			Solve University questions	1	BB
			Total number of Classes required	62	

15.Detailed Notes



S
l
i
d
e
2

UNIT-----1

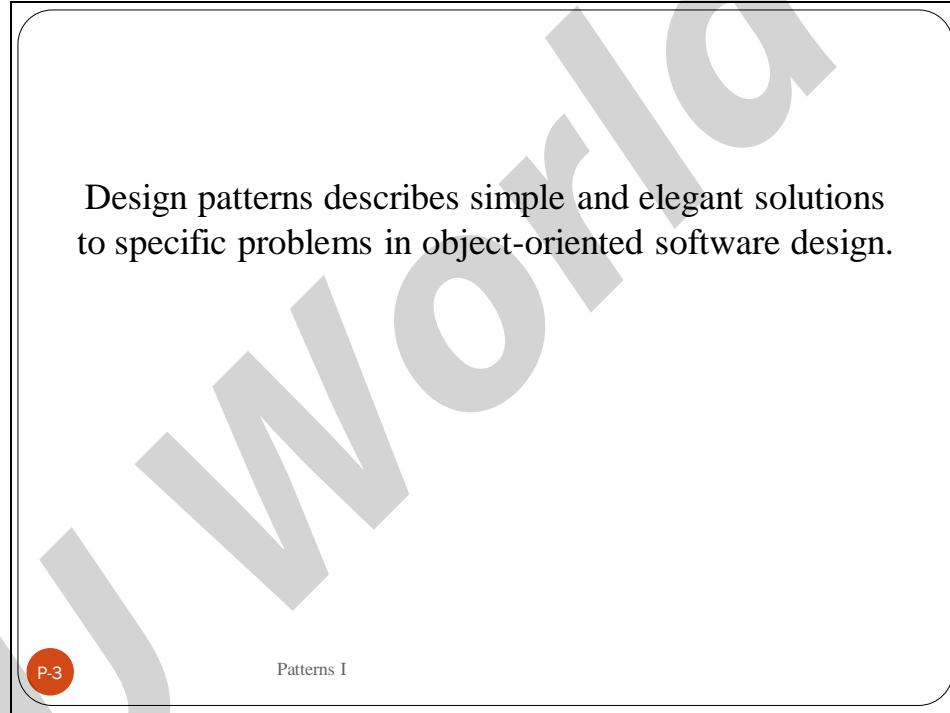
INTRODUCTION

P-2

Patterns I

S
l
i
d
e

3



S
l
i
d
e
4

Authors definition of **Design Patterns**

“The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context”.

OR

- A Design Pattern is essentially a description of a commonly occurring object-oriented design problem and how to solve it

P-4

Patterns I

S

I

I

d

e

5

Four essential elements of a pattern

- **Pattern Name**

- It is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- It gives higher level abstraction.
- Finding good names has been one of the hardest parts of developing out catalog.

- **Problem**

- It describes when to apply the pattern.
- It explains the problem and context.
- It describes how to represent algorithms as objects

S
l
i
d
e
6

- **Solution**

- It describes the elements that make up the design, the relationships, responsibilities and collaborations.
- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations
- Pattern provides a general arrangement of elements to solve it

- **Consequences**

- These are the results and trade –offs of applying the pattern
- When describing design decisions, these are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

P-6

Patterns I

S
I
D
E
7

A design pattern represents a widely accepted solution to a recurring design problem in OOP

Each design pattern focuses on a particular object-oriented design problem

P-7

Patterns I

S
I
D
E
8

Design Patterns in Smalltalk MVC

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

MVC consists of three kinds of objects. The **Model** is the application object, the **View** is its screen presentation, and the **Controller** defines the way the user interface reacts to user input

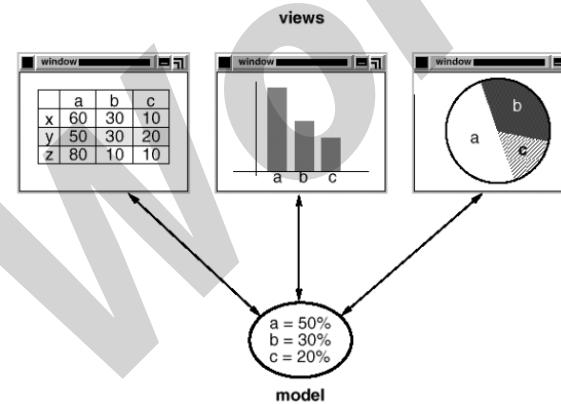
P.8

Patterns I

S
I
D
E

9

The example reflects a design that decouples views from models



P-9

S
l
i
d
e1
0

Describing Design patterns

Each pattern is divided into sections according to the following template

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly.

A good name is vital because it becomes the design vocabulary

Intent

A short statement that answers the following questions

- What does the design pattern do?
- What is its rationale and intent?
- What particular design issue or problem does it address?

S
l
i
d
e
1
1

Also known as
Other names for the pattern, if any

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

Applicability
What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

P-11 Patterns I

S
l
i
d
e1
2

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT).

We use interaction diagrams to illustrate sequences of requests and collaborations between objects.

Participants

The classes and/or objects participating in the design pattern and their responsibilities

P-12

Patterns I

S
l
i
d
e
1
3

Collaborations

How the participants collaborate to carry out their responsibilities

Consequences

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

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there any language –specific issues?

S
l
i
d
e
1
4

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk

Known Uses

Examples of the pattern found in real systems. At least 2 examples from different domains

Related patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

S

I

I

d

e

1

5

Patterns

- This book defined 23 patterns, classified into three categories.
 - *Creational patterns*, which deal with the process of object creation.
 - *Structural patterns*, which deal primarily with the static composition and structure of classes and objects.
 - *Behavioral patterns*, which deal primarily with dynamic interaction among classes and objects.

P-15

Patterns I

S
l
i
d
e

1
6

Catalog of Design Patterns

- *Creational Patterns*
 - Abstract Factory
 - Builder
 - Factory Method
 - Prototype
 - Singleton
- *Structural Patterns*
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Façade
 - Flyweight
 - Proxy
- *Behavioral Patterns*
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - Observer
 - State
 - Strategy
 - Template Method
 - Visitor

Patterns I

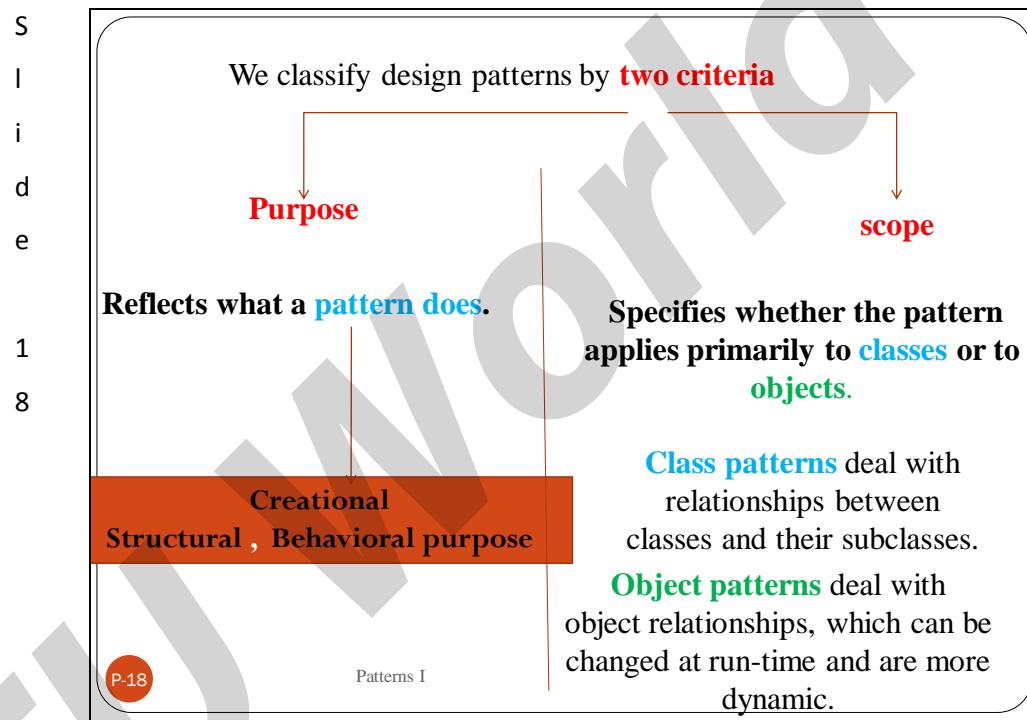
P-16

S
l
i
d
e

1
7

Scope	Purpose		
		Creational	Structural
Class	Factory Method	Adapter(Class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (Object) Bridge Composite Decorator Façade Flyweight proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy visitor

P-17



S
l
i
d
e

1
9

How design patterns solve design problems

• 1) Finding Appropriate Objects

- An object packages both data and the procedures(methods or operations) that operate on the data
- An object performs an operation when it receives a request from a client
- These requests causes an object to execute an operation, operations are the only way to change an object's internal data, then object internal state is said to be **encapsulated** (i.e. object's representation is invisible from the outside).

S

I

I

D

E

2

0

- How to Create an object
write problem statement find out nouns and verbs and create corresponding classes and operations .
- Design patterns help you identify less-obvious abstractions and the objects that can capture them.

The **State (305) pattern** represents each state of an entity as an object.

S
l
i
d
e2
1

P-21

• 2) Determining object Granularity

- How do we decide what should be an object?
- Subsystems can be represented as objects (Facade)
- Objects of finest granularity (Flyweight)
- Objects responsibilities can create other objects (Abstract Factory, Builder)

• 3) Specifying object interfaces

- An operation can be specified as operation name, parameters and its return value is known as *signature*
- The set of all signatures defined by an object's operations is called the *interface* to the object.

Patterns I

S
l
i
d
e

2
2

- A *type* is a name used to denote a particular interface
Eg. Window obj1;
- A *type* is a *subtype* of another if its interface contains the interface of its *supertype*.
- The runtime association of a request to an object and one of its operations is known as *dynamic binding*.
- Dynamic binding lets us to substitute objects that have identical interfaces for each other at runtime is called as *Polymorphism*.

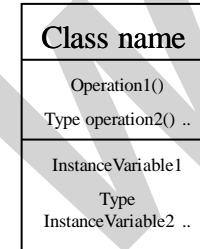
Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.

A design pattern might also tell you what *not to put in the interface*.

S
l
i
d
e2
3

4) Specifying Object Implementations

- An object's implementation is defined by its *class*. The class specifies the object's internal data and representation and defines the operations the object can perform
- Class can be rendered as

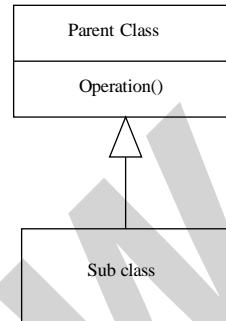


Patterns I

P-23

S
l
i
d
e
2
4

- Class inheritance



Patterns I

P-24

S
l
i
d
e2
5

P-25

- An *abstract class* is one whose main purpose is to define a common interface for its subclass(it defers implementation to operations defined in subclass).
 - It cannot be instantiated
- The operations that an abstract class declares but doesn't implement are called *abstract operations*.
(the names in italic indicates a class or operation as abstract)
- Classes that are not abstract are called *concrete classes*.
- A class may *override* an operation defined by its parent class
- A *Mixin class* is a class that's intended to provide an optional interface or functionality to other classes(it is an abstract class that can be instantiated).

Patterns I

S
l
i
d
e
2
6

- Class Vs Interface inheritance
 - The difference between object's class and type is class defines the object's internal state and the implementation of its operations, in contrast type refers to its interface.
 - Class inheritance defines an object's implementation in terms of another object's implementation(mechanism for code and representation sharing), in contrast interface inheritance describes when an object can be used in place of another.
 - Programming to an interface, not an Implementation
 - It is the principle of reusable OOD.

P-26

Patterns I

S
l
i
d
e

2
7

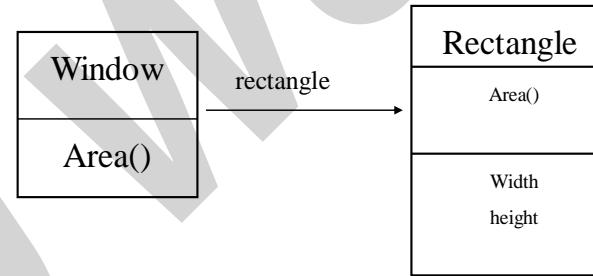
5) Putting reuse mechanisms to work

- It is easy to know class, interface, object etc., but applying them to build flexible, reusable s/w and Design Patterns can show how
- Inheritance versus Composition
 - White-box reuse – with inheritance, the internals of parent classes are often visible to subclasses (visibility).
 - Black-box reuse – object composing, i.e., objects that composed have well defined interfaces
- Disadvantages with class inheritance
 - Change the implementations inherited from parent classes at run-time is not possible.
 - Parent classes often define at least part of their subclasses physical representation.

S
l
i
d
e2
8

P-28

- Delegation
- It is a way of making composition as powerful for reuse as inheritance
- The following diagram depicts the Window class delegating its Area operation to a Rectangle instance



Patterns I

S
l
i
d
e

2
9

- Advantage of delegation – it makes it easy to compose behaviors at run-time and to change the way they are composed
- Disadvantage – it shares with other techniques that make software more flexible through object composition (I.e., understanding s/w, runtime inefficiencies)
- Several design patterns use delegation.
- The **State (338)**, **Strategy (349)**, and **Visitor (366)** patterns depend on it.

P-29

Patterns I

6) Relating run-time and compile time structures

- compile time structure is static and runtime structure is dynamic.
 - **Acquaintance or Association**
 - **Acquaintance implies that an object merely knows of another object.**
 - **Aggregation**
 - Aggregation implies that one object owns or is responsible for another object
 - :

P-30

S
I
I
d
e

In our diagrams, a plain arrowhead line denotes **acquaintance**. An arrowhead line with a diamond at its base denotes **aggregation**



Many design patterns capture the distinction between compile-time and run-time structures explicitly.

Composite(183) and Decorator (196) are especially useful for building complex run-time structures.

Observer (326) involves run-time structures that are often hard to understand unless you know the pattern

S
I
D
E
3
2

7) Designing for change

- A design that doesn't take change into account risks major redesign in the future.
- We must consider how the system might need to change over its lifetime
- These changes leads to class redefinition and reimplementations, client modification, and retesting
- Some common causes of redesign
 - Creating an object by specifying a class explicitly. (Abstract Factory, Factory Method, Prototype)
 - Dependence on hardware and software platform. (Chain of responsibility, Command)
 - Dependence on object representations or implementations. (Abstract Factory, Bridge)
 - Algorithmic dependencies
 - Tight coupling. etc.,

S
l
i
d
e
3
3

Role played by Design patterns

- Application Programs
 - When building application program such as a document editor or spreadsheet then internal reuse, maintainability, and extensions are high priorities.
 - design patterns makes it easy to have all these without any side effects.
- Toolkits
 - A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality.
- Frameworks
 - A framework is a set of cooperating classes that makeup a reusable design for a specific class of software.

S
l
i
d
e

3
4

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.

P-34

Patterns I

S
l
i
d
e

3
5

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 application- specific names for operations in the pattern.
- Implement the operations to carry out the responsibilities and collaborations in the pattern

UNIT-2 DESIGNING A DOCUMENT EDITOR

Slide 1



Slide 2

- This chapter presents a case study in the design of a "What-You-See-Is-What-You-Get" (or "WYSIWYG") document editor called **Lexi**.
- We'll see how design patterns capture solutions to **design problems** in Lexi and applications like it.

Figure ➔ depicts Lexi's user interface.

P-2

Slide 3



Slide 4

Design Problems

- Document structure
 - The choice of internal representation for the document affects nearly every aspect of Lexi's design.
 - 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?
- Embellishing the user interface
 - Lexi's user interface includes scroll bars, borders, and drop shadows that embellish (Decorating) the WYSIWYG document interface.

Slide 5

- Supporting multiple look and feel standards
 - Lexi should adapt easily to different look and feel standards such as Motif and Presentation Manager without major modification.
- Supporting multiple window systems
 - Different look and feel standards are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible.
- User operations
 - User control Lexi through various user interfaces, including buttons and pull-down menus.

Slide 6

- Spelling checking and hyphenation
 - How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points?

P-6

Slide 7

1) Document structure

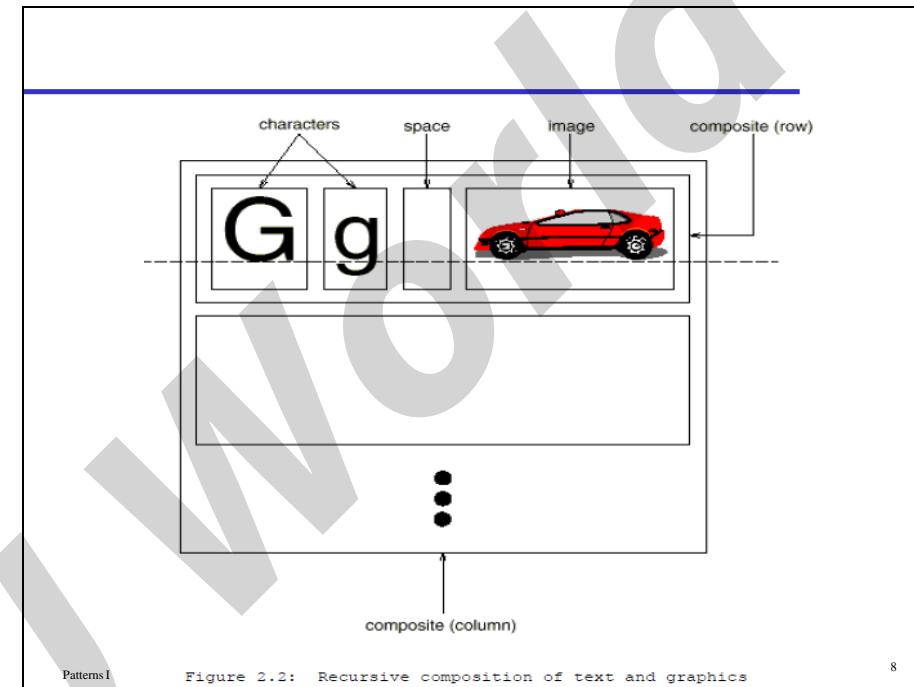
- **Recursive composition**

A common way to represent hierarchically structured information is through a technique called **recursive composition**

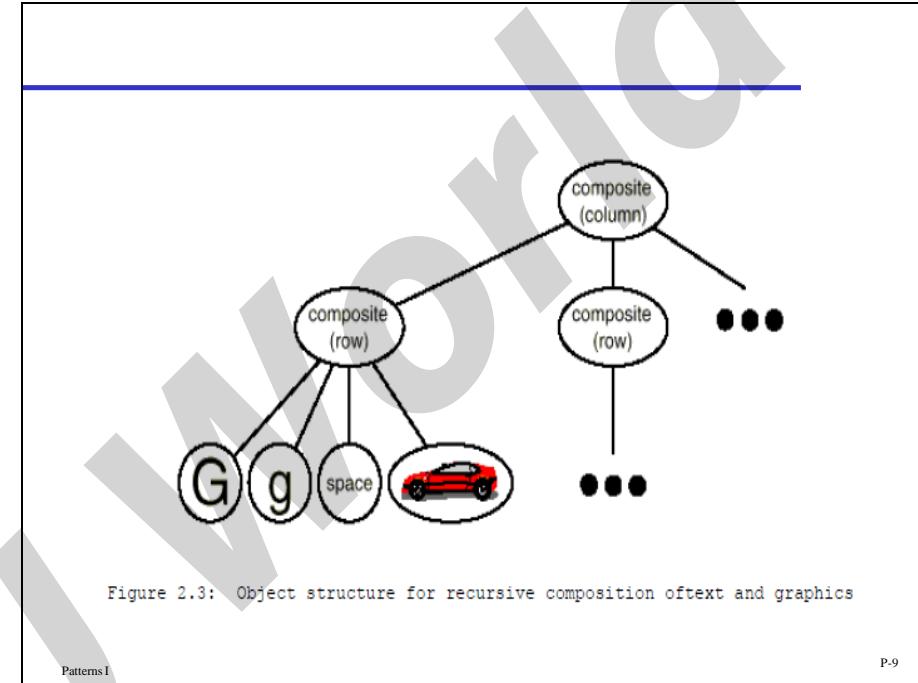
As a first step, we can tile a set of characters and graphics from left to right to form a line in the document.

Then multiple lines can be arranged to form a column , multiple columns can form a page, and so on.

Slide 8



Slide 9

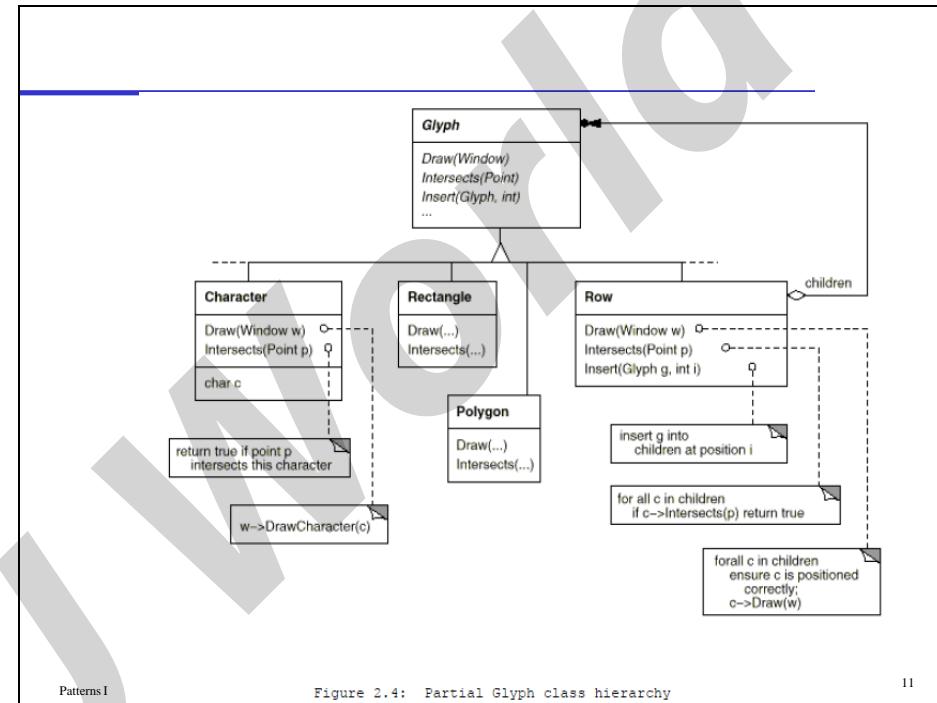


Slide 10

- **Glyphs**

- A glyph is an abstract class for all objects that can appear in a document structure.
- Its subclasses define both primitive graphical elements (char and images) and structural elements (rows and columns)
- Glyph have 3 basic responsibilities
 - How to draw themselves
 - What space they occupy and
 - Their children and parent

Slide 11



Slide 12

- **Composite pattern**

- We can use **recursive composition** to represent any potentially **complex hierarchical structure**.
- The **composite pattern** captures the essence of **recursive composition** in **object-oriented terms**

Slide 13

2) Formatting

- Formatting or Line Breaking → Lexi must break text into lines, lines into columns, and columns into pages, taking into account the users higher level desires.
 - ❖ In this case we are considering formatting as breaking a collection of glyphs into lines.

Encapsulating the Formatting Algorithm

Important trade-off

balance between
formatting quality and formatting speed.

balances formatting speed and storage
requirements

Slide 14

Define a separate class hierarchy for objects that encapsulate formatting algorithms

The root of the hierarchy will define an interface that supports a wide range of formatting algorithms, and each subclass will implement the interface to carryout a particular algorithm

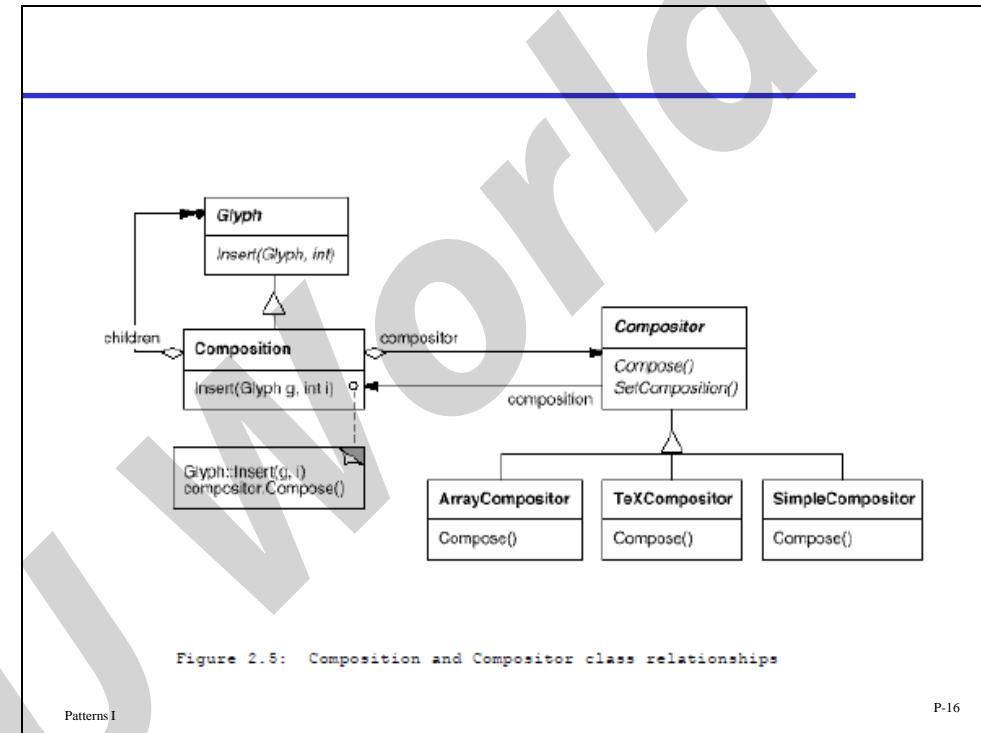
Responsibility	Operations
what to format	void SetComposition(Composition*)
when to format	virtual void Compose()

Table 2.2 Basic compositor interface

Slide 15

- **Compositor and composition**
 - We'll define **Compositor** class for objects that can encapsulate a formatting algorithm.
 - The glyphs it formats are the children of a special Glyph subclass called **Composition**
 - When the **composition** needs formatting, it calls its **compositor's Compose** operation
- The **compositor** in turn iterates through the **composition's** children and inserts new Row and Column glyphs according to its line breaking algorithm.

Slide 16



Slide 17

Glyphs that the compositor created and inserted
In to the object structure appear with gray backgrounds in the figure.

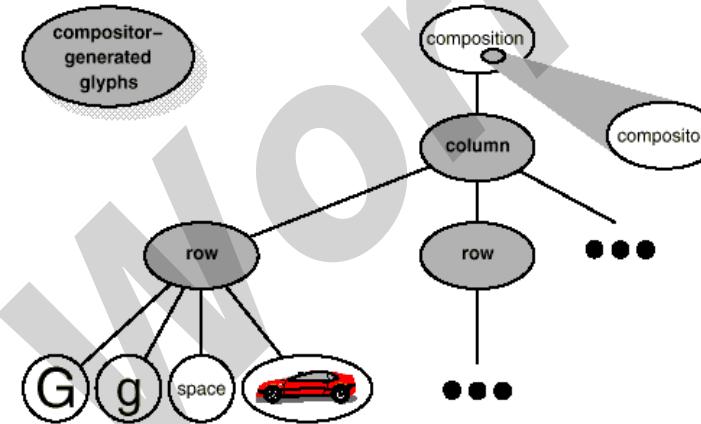


Figure 2.6: Object structure reflecting compositor-directed linebreaking

Patterns I

P-17

Slide 18

The **Composite-Composition** class split ensures a strong separation between code that supports the document's physical structure and the code for different formatting algorithms. We can add new **Composer** subclasses without touching the **glyph** classes, and vice versa.

Strategy pattern

- Encapsulating an algorithm in an object is the intent of the strategy pattern.
- The key to applying the Strategy pattern is designing interfaces for the strategy and its context that are general enough to support a range of algorithms.

Slide 19

3) Embellishing the User Interface

- We consider 2 types of embellishments
- 1. **Border** 2. **Scroll bars**
- The first adds a **border** around the text editing area to demarcate the page of text.
- The second adds **scrollbars** that let the user view different parts of the page.
- **Transparent Enclosure** → to experiment with different alternatives, and it keeps clients free of embellishment code.
- **Mono Glyph**
 - A subclass of glyph called **Mono Glyph** serve as an abstract class for embellishment glyphs.

P-19

Slide 20

Mono Glyph stores a reference to a component and forwards all requests to it

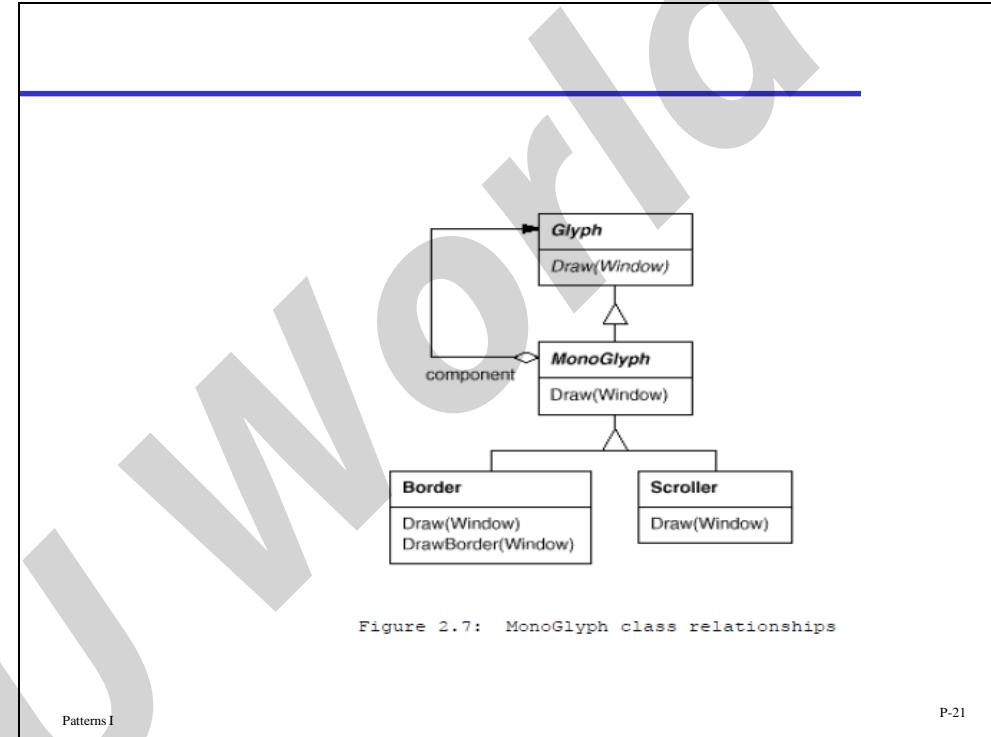
For example, MonoGlyph implements the Draw operation like this:

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

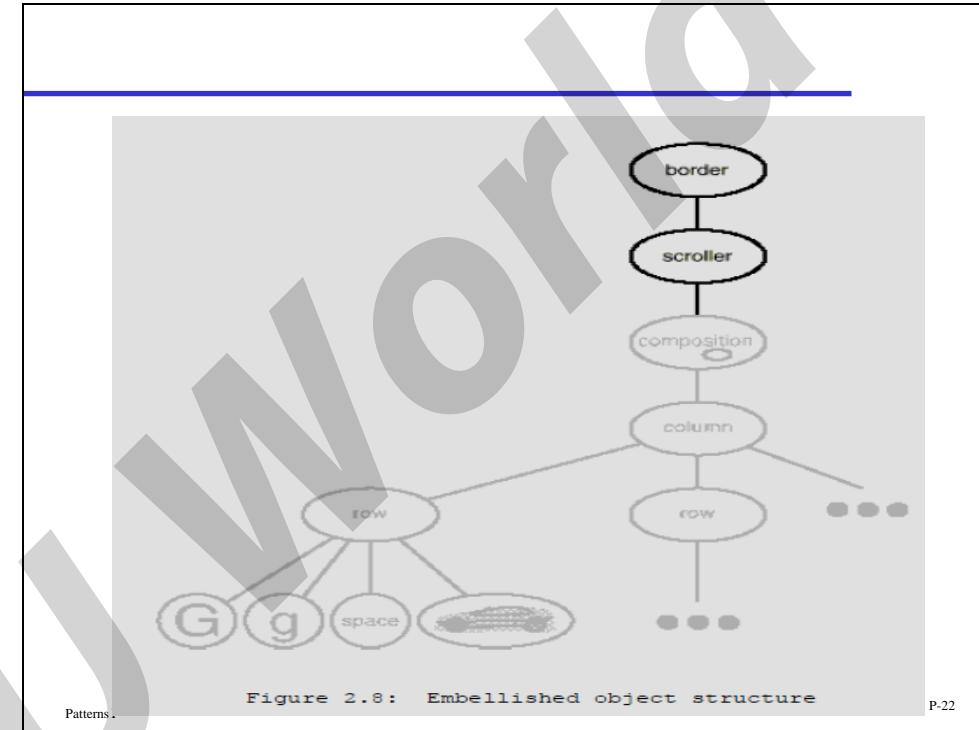
Decorator pattern

Decorator pattern captures class and object relationships that support embellishment by transparent enclosure.

Slide 21



Slide 22



Slide 23

4) Supporting multiple look-and feel standards

- Achieving portability is a major problem in system design
- One obstacle to portability is the diversity of look and feel standards
- Our design goals are to make lexi conform to multiple existing look and feel standards and to make it easy to add support for new standards as they emerge and to support the ultimate in flexibility (i.e., changing lexi's look and feel at runtime)

Slide 24

- **Abstracting Object Creation**
 - Lexi's user interface is a glyph composed in other, invisible glyphs like Row and Column.
- The **invisible** glyphs compose **visible** ones like Button and Character and lay them out properly.
 - **Widgets** is the term used for visible glyphs like buttons, scroll bars and menus that act as controlling elements in a user interface
- **Widgets** might use simpler glyphs such as characters, circles, rectangles, and polygons to present data.

P-24

Slide 25

There are two types of widget glyph class with which to implement multiple look and feel standards

- ◆ A set of abstract Glyph subclasses for each category of widget glyph.
- Button is an abstract class that adds button-oriented operations;
 - ◆ A set of concrete subclasses for each abstract subclass that implement different look and feel standards.
 - For example, ScrollBar might have MotifScrollBar and PMScrollBar subclasses that implement Motif and Presentation Manager-style scroll bar

P-25

Slide 26

Factories and Product Classes

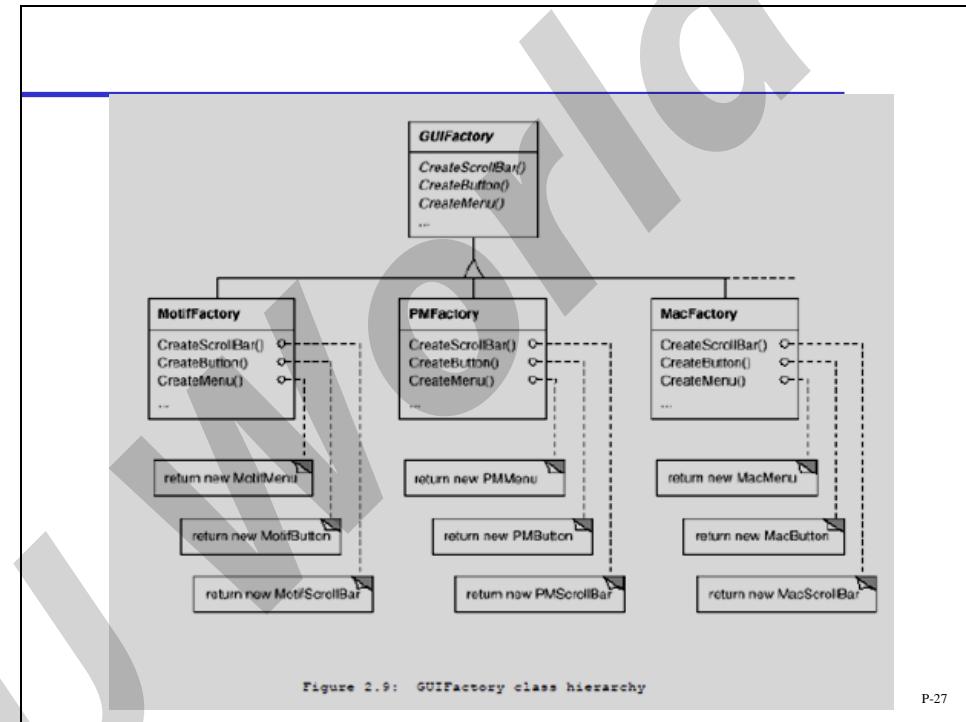
Factories create Product objects. The products that a factory produces are related to one another.

Abstract Factory pattern

Factories and products are the key participants in the Abstract Factory Pattern.

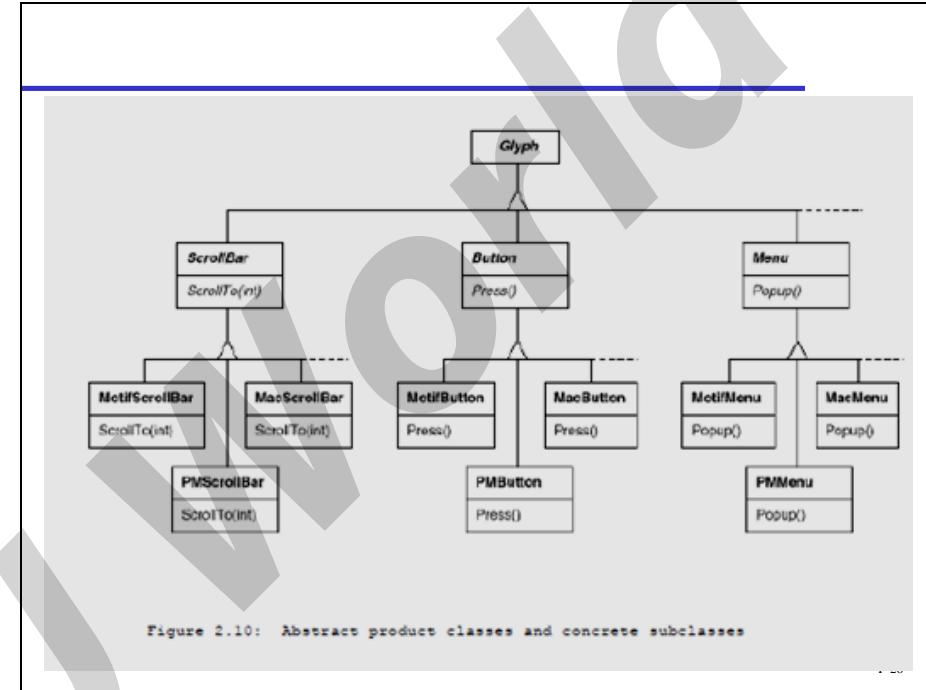
This pattern captures how to create families of related product objects without instantiating classes directly

Slide 27



P-27

Slide 28



Slide 29

5) Supporting Multiple Window systems

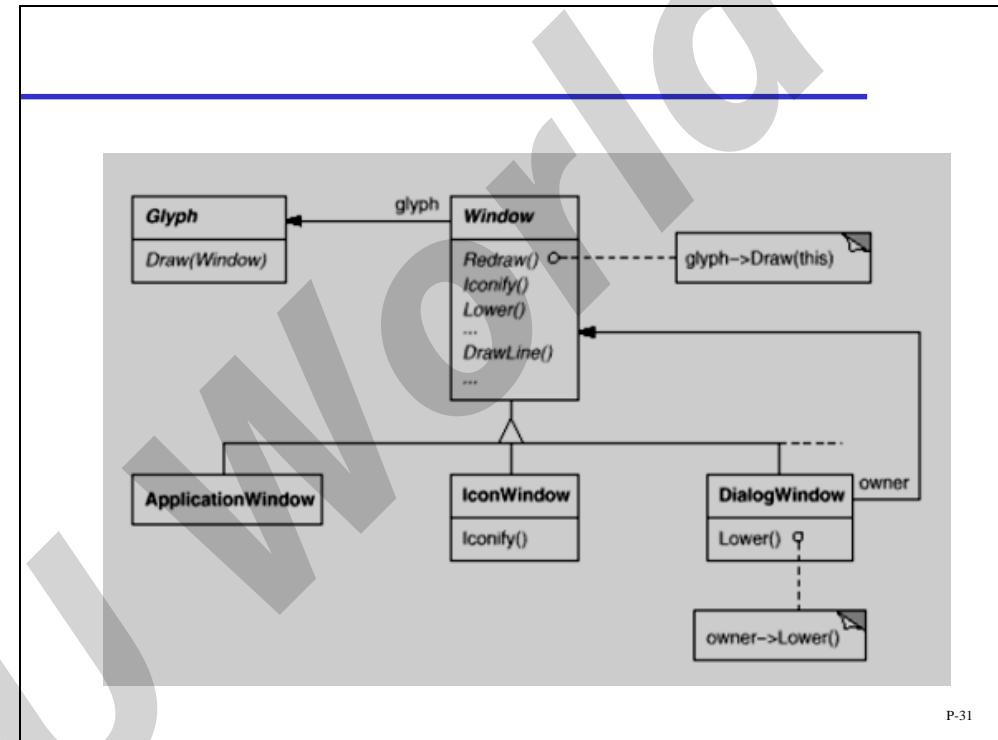
- **Can we use an Abstract Factory**
 - Two different windows of different vendors may not have same hierarchy. Hence we won't have a common abstract product class for each kind of widget and abstract factory pattern won't work
- **Encapsulating Implementation Dependencies**
 - Window class encapsulates the functionalities i.e.
 - ◆ They provide operations for drawing basic geometric shapes.
 - ◆ They can iconify and de-iconify themselves.
 - ◆ They can resize themselves
 - ◆ They can (re)draw their contents on demand,

P-29

Slide 30

- The Window class must span the functionality of windows from different window systems.
- Let's consider two extreme philosophies:
- **Intersection of functionality**
 - The window class interface provides only functionality that's common to all window systems.
- **union of functionality**
 - Create an interface that incorporates the capabilities of all existing systems.
- Extreme of any of the above is not a viable solution, so our design will fall somewhere between the two

Slide 31



Slide 32

✓ where does the **real platform-specific window** come in?

➤ One approach is to implement multiple **versions** of the Window class and its subclasses, one version for each windowing platform.

Both of these alternatives have another drawback:

➤ Neither gives us the **flexibility to change the window system** we use after we've **compiled** the program.

➤ We can configure window objects to the window system we want simply by passing them the right window system-encapsulating object.

✓ We can even configure the window at **run-time**

Slide 33

Window and WindowImp

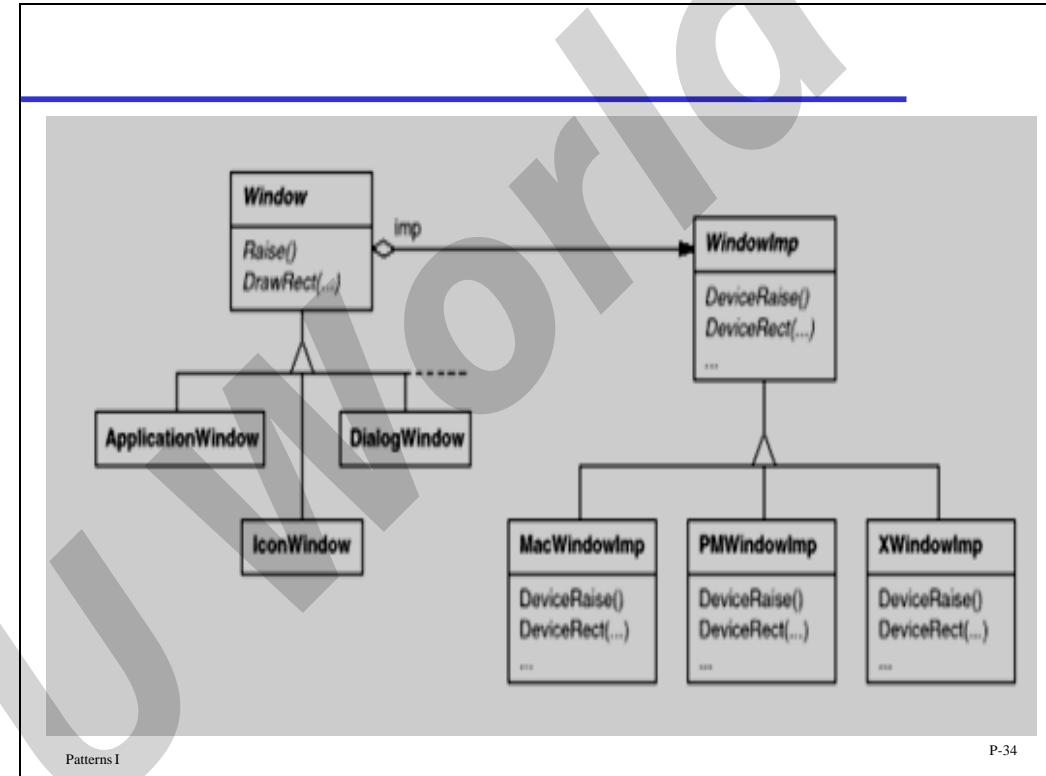
We'll define a separate **WindowImp class hierarchy** in which to **hide different window system implementations.**

WindowImp is an abstract class for objects that encapsulate window system-dependent code.

- ✓ We can easily **extend the implementation hierarchy** to support **new window systems.**

P-33

Slide 34



Slide 35

Bridge Pattern

The relationship between Window and WindowImp is an example of the **Bridge pattern**.

Bridge pattern allow separate class hierarchies to work together even as they evolve independently

Our design criteria led us to create two separate class hierarchies, one that supports the logical notion of windows,
And another for capturing different implementations of windows

Slide 36

6) User Operations

- Some user operations are
 - Creating a new document
 - Opening, saving, and printing an existing document
 - Cutting selected text out of the document and pasting it back in
 - Quitting the application etc.,
- Lexi provides different user interfaces for these operations.
- But problem is, you can turn the page using either a page button or a menu operation.

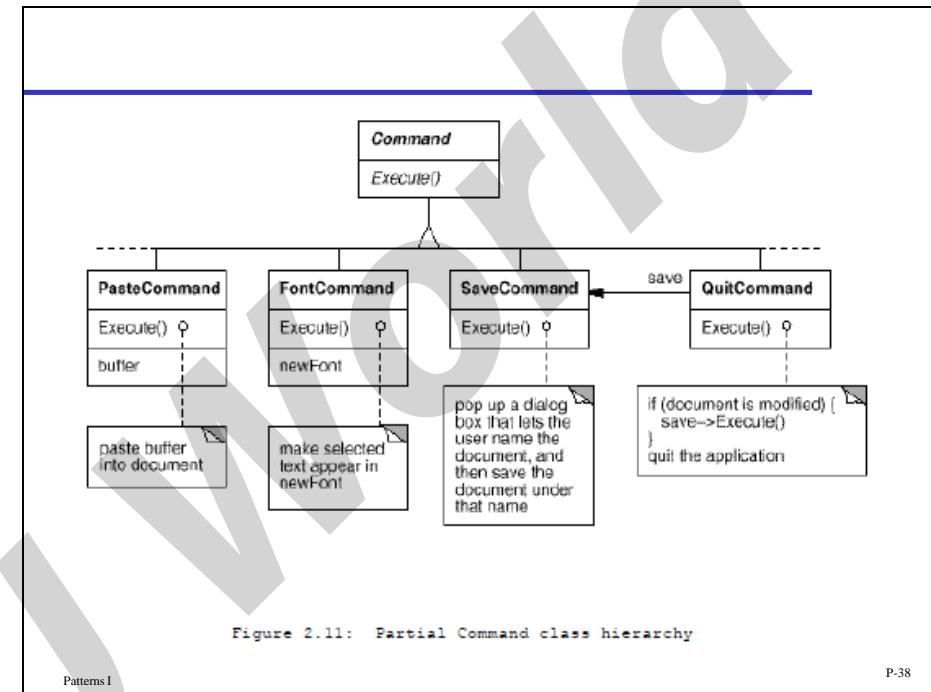
P-36

Slide 37

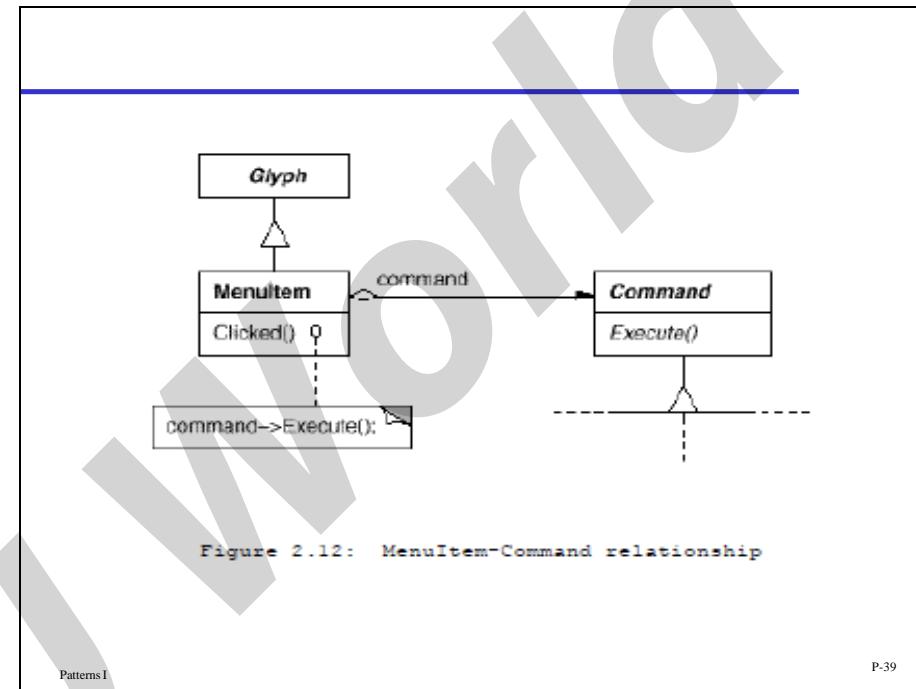
- **Encapsulating a request**
- A Glyph subclass called **MenuItem**
- Carrying out the request might involve an operation on one object, or many operations on many objects.
- But it cannot address
 - Undo / redo problem
 - Can't associate state with a function
 - Functions are hard to extend
- We will encapsulate each request in a **command object**
- **Command class and subclasses**
 - Command abstract class provides an interface for issuing a request.
 - Subclass of command implement this function (Execute()) in different ways to fulfill different requests.
 - Now MenuItem can store a command object that encapsulates a request.

P-37

Slide 38



Slide 39



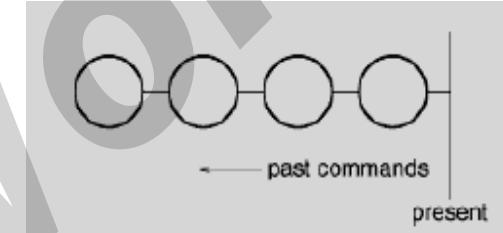
Slide 40

- **undo ability**

- **Unexecute()** operation which works reverse to **Execute()** to accommodate undo/redo functions

- **Command history or list of commands**

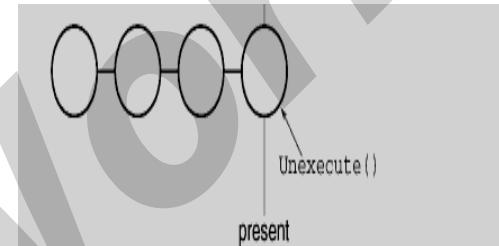
- To accommodate undo/redo command history should be maintained.



Each circle represents a Command object. In this case the user has issued four commands

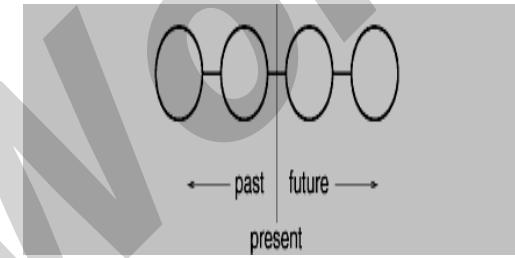
Slide 41

To undo the last command, we simply call **Unexecute** on the most recent command:



Slide 42

After **unexecuting** the command, we move the "present" line one command to the left. If the user chooses undo again, the next-most recently issued command will be undone in the same way, and we're left in the state depicted here:

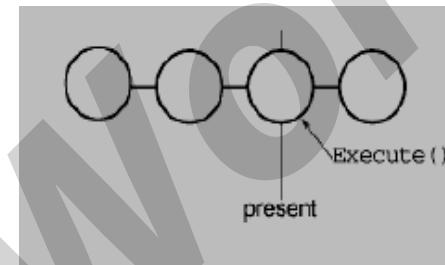


The number of levels is limited only by the length of the **command history**

P-42

Slide 43

To redo the last undone command, we call **Execute** on the command to the right of the present line:



Slide 44

Command pattern

Lexi's commands are an application of the **command pattern** which describes how to encapsulate a request.

The pattern also discusses undo and redo mechanisms built on the basic Command interface.

Slide 45

7) Spelling checking and Hyphenation

- This is a textual analysis, specifically **checking for misspellings** and introducing **hyphenation points** where needed for good formatting.
- **Here we could come across two problems**
 - Accessing the information to be analyzed, which we have scattered over the glyphs in the document structure and
 - Doing the analysis
- **Accessing Scattered Information**
 - The text we need to analyze is scattered throughout a hierarchical structure of glyph object
 - Some glyphs might store their children in linked lists, others might use arrays, and still others might use more complex data structures.

Slide 46

- ✓ So our access mechanism must accommodate differing data structures and we must support different kinds of traversals such as preorder, postorder, and inorder.

Encapsulating Access and Traversal

We can add the following abstract operations to Glyph's interface to support

Void First(Traversal kind)
Void Next()
Bool IsDone()
Glyph* GetCurrent()
Void Insert(Glyph*)

P-46

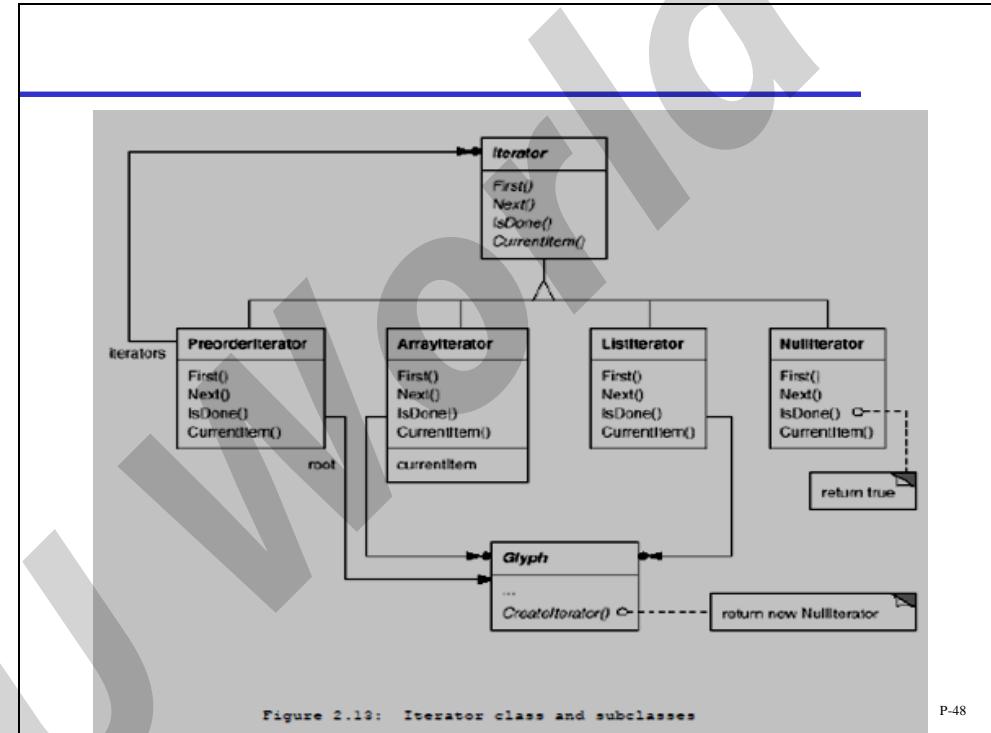
Slide 47

- A better solution is to encapsulate the concept that varies, in this case the access and traversal mechanisms is **Iterators**.
- **Iterator class and subclasses**
 - We'll use an abstract class called Iterator to define a general interface for access and traversal.
 - Concrete subclasses like ArrayIterator and ListIterator implement the interface to provide access to arrays and lists

Notice that we've added a CreateIterator abstract operation to the Glyph class interface to support iterators..

P-47

Slide 48



Slide 49

- **Iterator Pattern**
 - This pattern captures these techniques for supporting access and traversal over object structures
- **Traversal versus Traversal Actions**
 - Analysis and Traversal should be separate
- preorder traversal is common to many analyses, including spelling checking, hyphenation, forward search, and word count
 - Analysis must be able to distinguish different kinds of glyphs

Slide 50

Visitor class and subclasses

It applies a hyphenation algorithm to determine the potential hyphenation points in the word, if any.

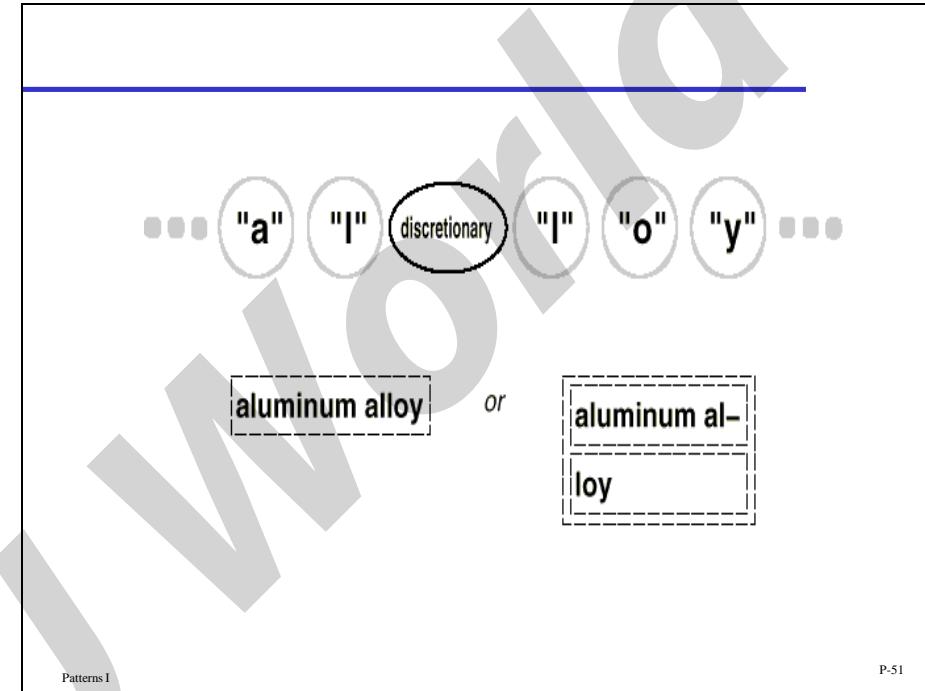
Then at each hyphenation point, it inserts a **discretionary glyph into the composition**.

Discretionary glyphs are subclass Of Glyph.

A discretionary glyph has one of two possible appearances depending on whether or not it is the last character on a line.

If it's the last character, then the discretionary looks like a hyphen;

Slide 51



UNIT-III

Unit-3

Creational Design Pattern

**Factory
Singleton
Abstract Factory
Prototype**

Factory Design Pattern

Factory

Design Purpose

Create individual objects in situations where the constructor alone is inadequate.

Design Pattern Summary

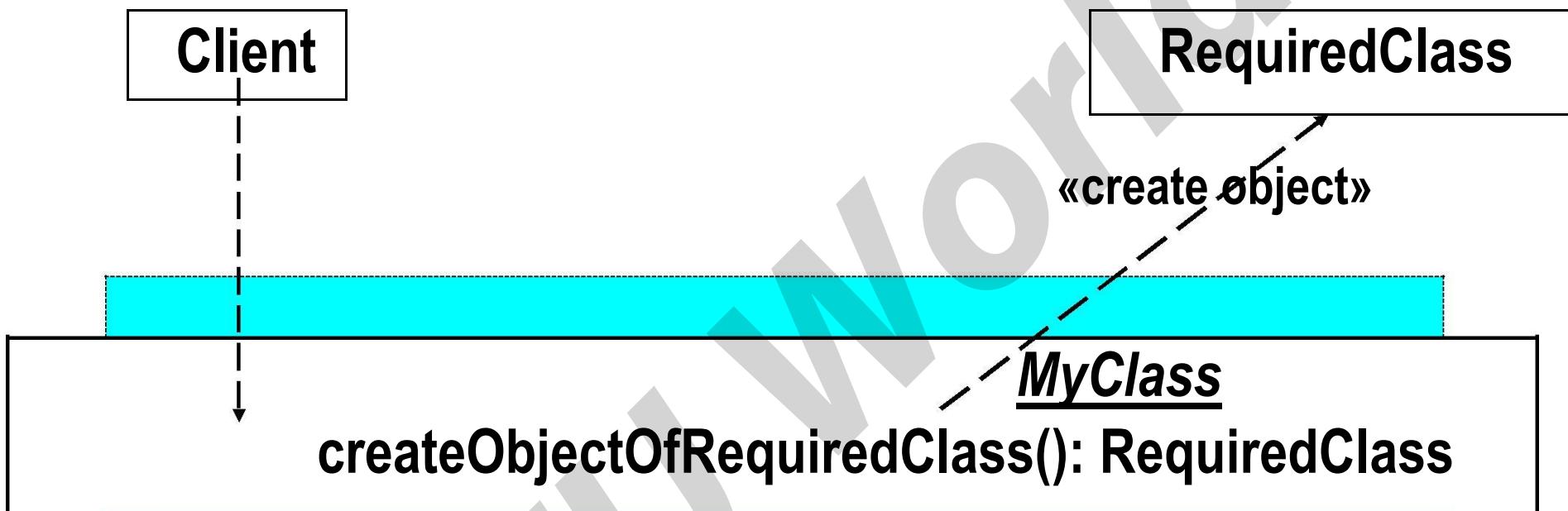
Use methods to return required objects.

Factory Interface for Clients

Demonstrates the use of a static method to create an instance of a class

```
public static void main(String[] args)
{
RequiredClass instanceOfRequiredClass = MyClass.getNewInstanceOfRequiredClass();
}
} // End main
```

Factory Class Model



Factory design pattern

Design Goal At Work: → Reusability and Correctness

We want to write code about automobiles in general: Code that applies to any make, exercised repeatedly (thus reliably).

Client

Factory Example

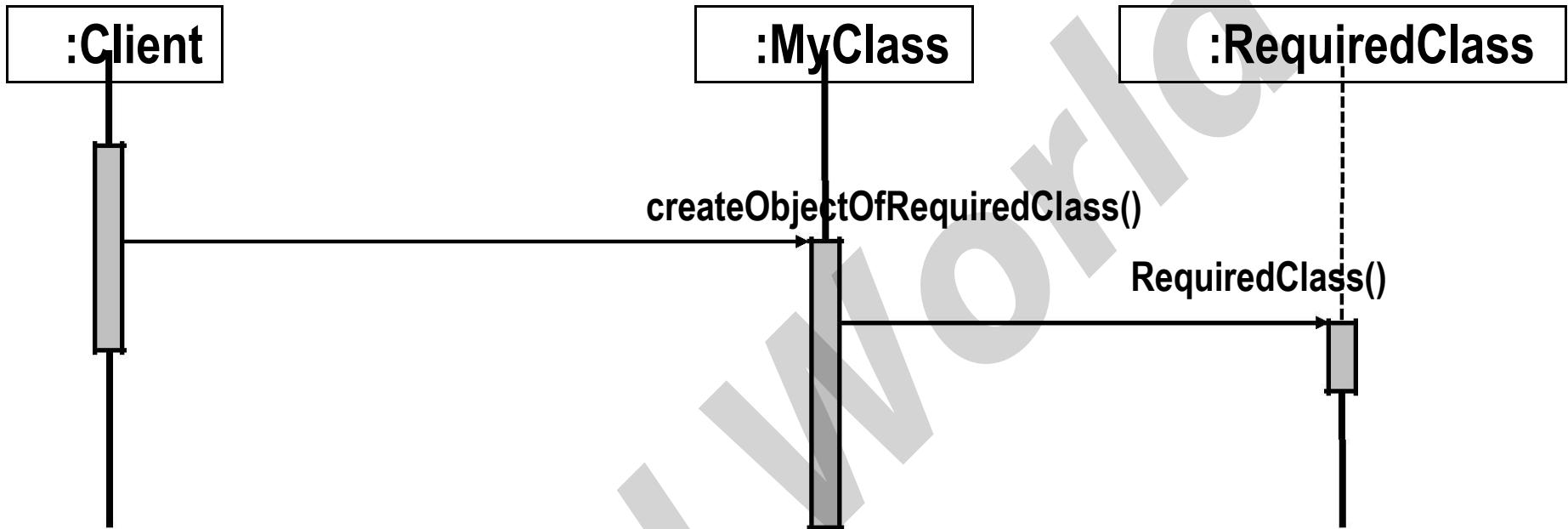
Application
of *Factory*
design
pattern

Automobile***createAutomobile(): Automobile*****Ford*****createAutomobile()*****Toyota*****createAutomobile()*****«create object»****«create object»**

Example Code

```
class Ford extends Automobile
{
    static Automobile createAutomobile()
    {
        return new Ford();
    } // End createAutomobile
} // End class
```

Sequence Diagram for Factory



Typical Output of E-Mail Generation Example

Please pick a type of customer from one of the following:
curious
returning
frequent
newbie
returning →
This message will be sent:

Word that was entered

Lots of material intended for all customers ...
... a (possibly long) message for returning customers ...

Design Goals At Work:

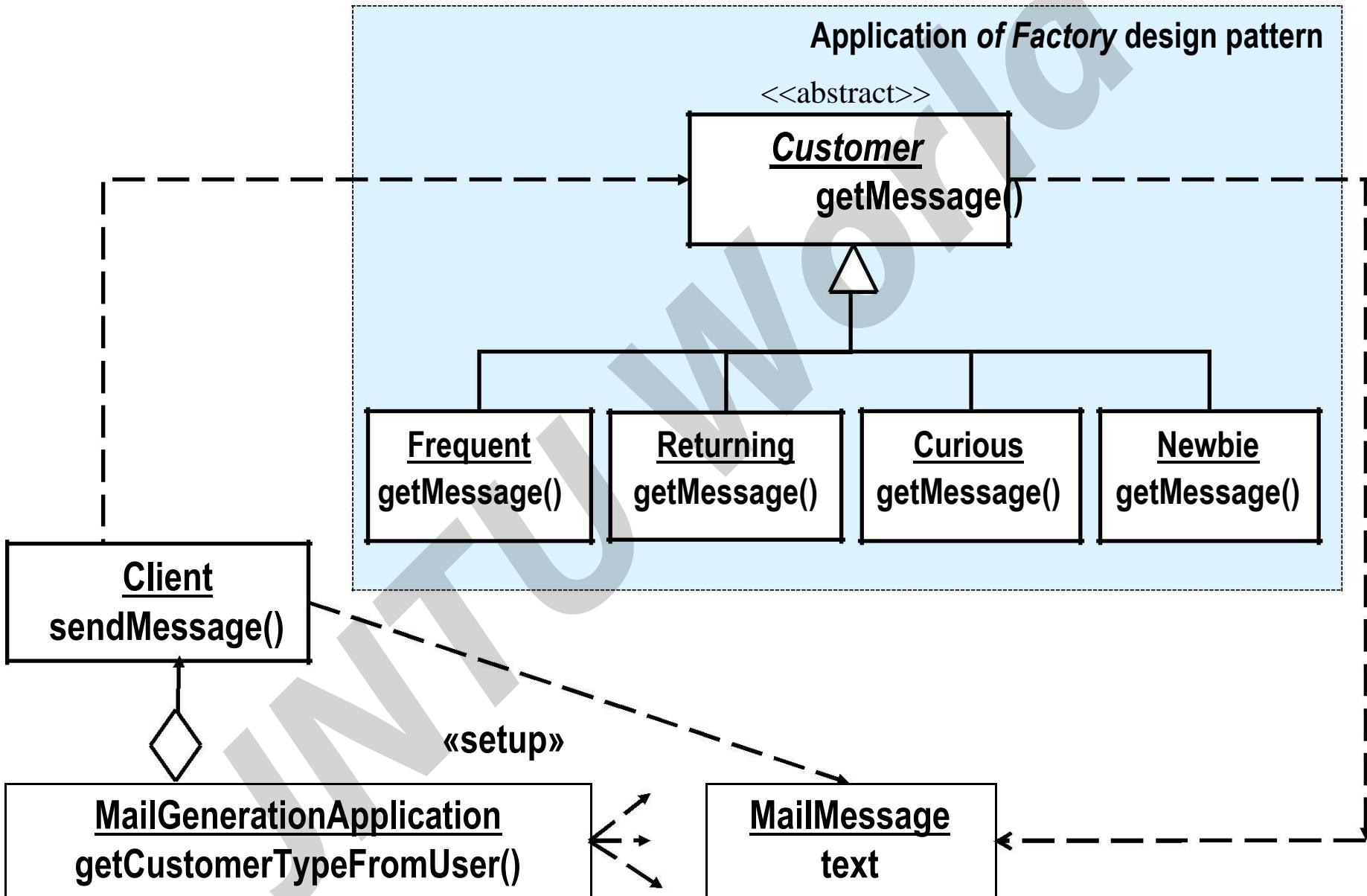


Correctness and Reusability

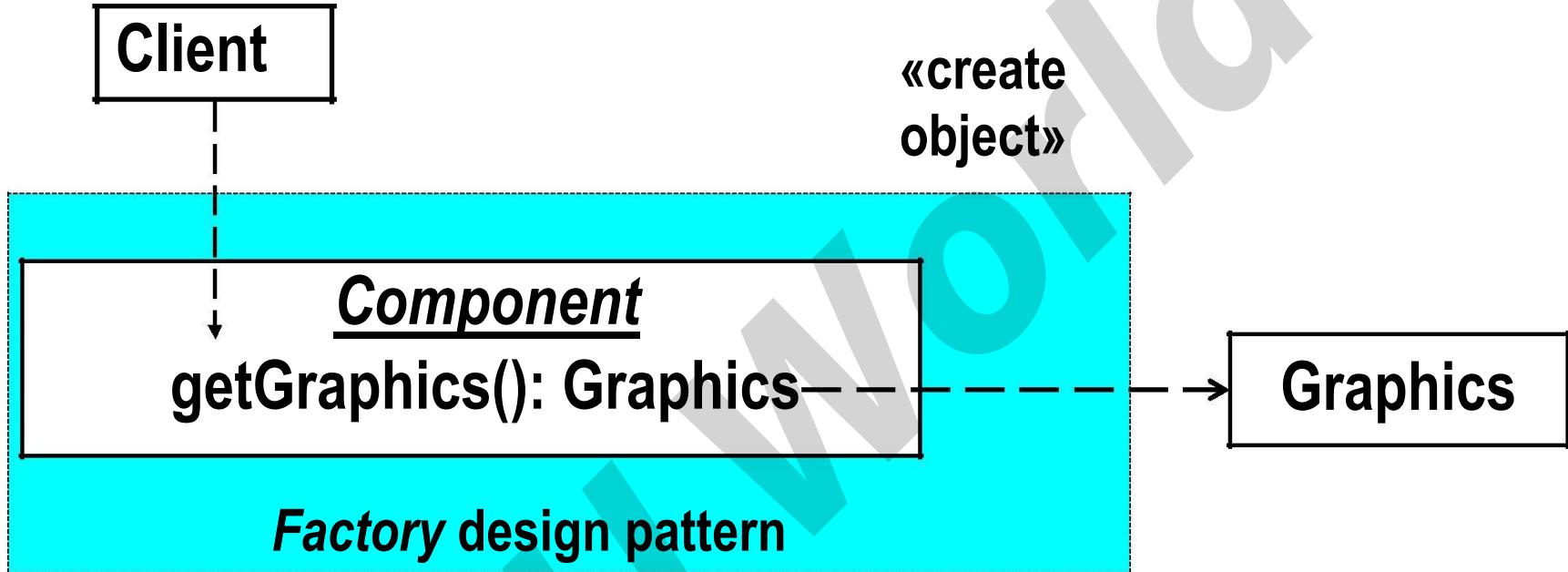


We want to separate the code common to all types of customers. We want to separate the specialized code that generates e-mail for each type of customer. This makes it easier to check for correctness and to reuse parts.

Factory: Email Generation Example



Factory Applied to `getGraphics()` in Java



```
public static Box createVerticalBox()
```

```
public static Box createHorizontalBox()
```

7.2 - Singleton Design Pattern

Key Concept: → Singleton Design Pattern

-- when a class has exactly one instance.

Singleton

Design Purpose

Ensure that there is exactly one instance of a class **S**. Be able to obtain the instance from anywhere in the application.

Design Pattern Summary

Make the constructor of **S** private; define a private static attribute for **S** of type **S**; define a public accessor for it.

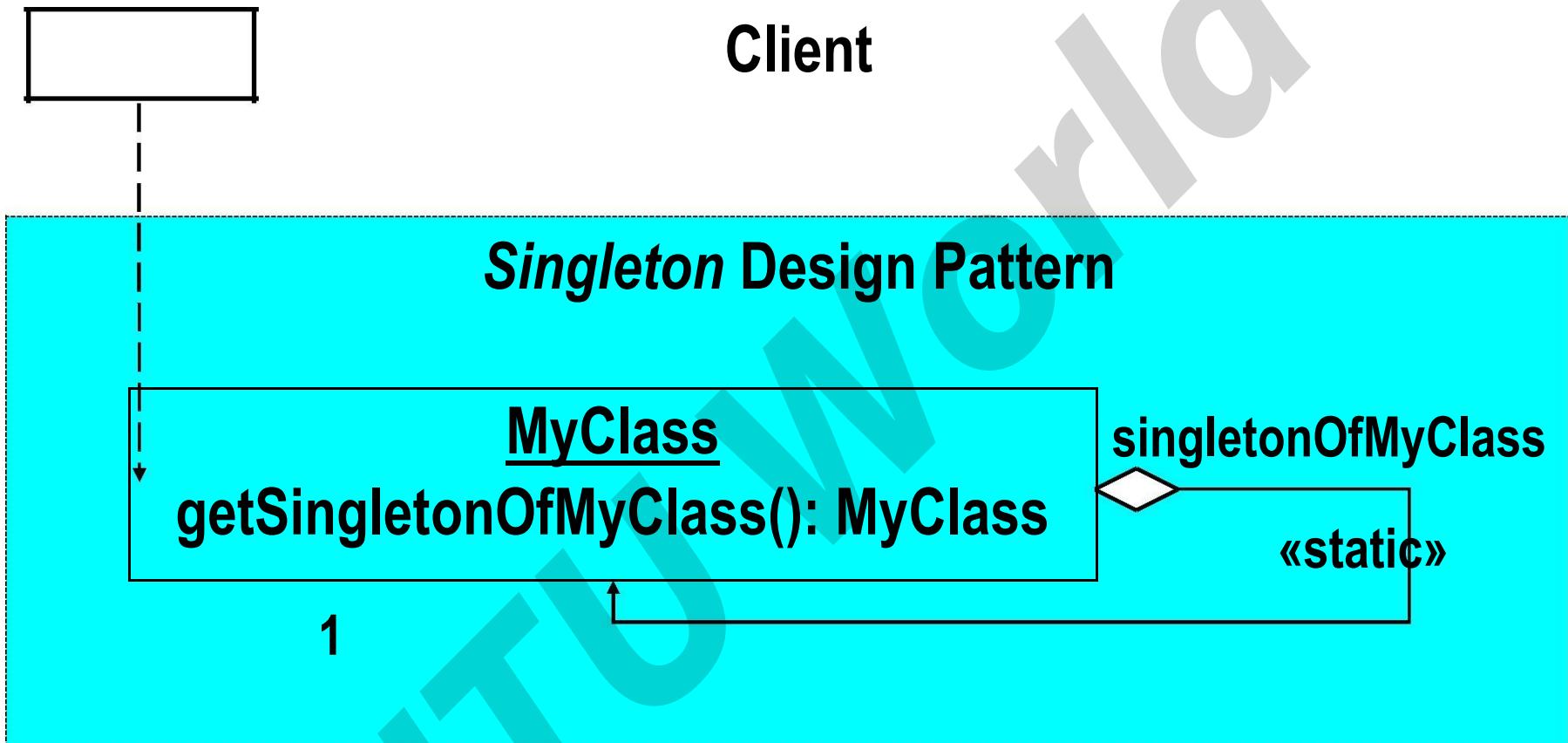
Design Goal At Work:  *Correctness* 

Singleton enforces the intention that only one User object exists, safeguarding the application from unanticipated User instance creation.

The Singleton Interface for Clients

```
User mainUser = User.getTheUser();
```

Singleton: Class Model



The Singleton Design Pattern -- applied to *MyClass*

1. Define a **private static member variable** of **MyClass** of type **MyClass**

```
private static MyClass singletonOfMyClass = new MyClass();
```

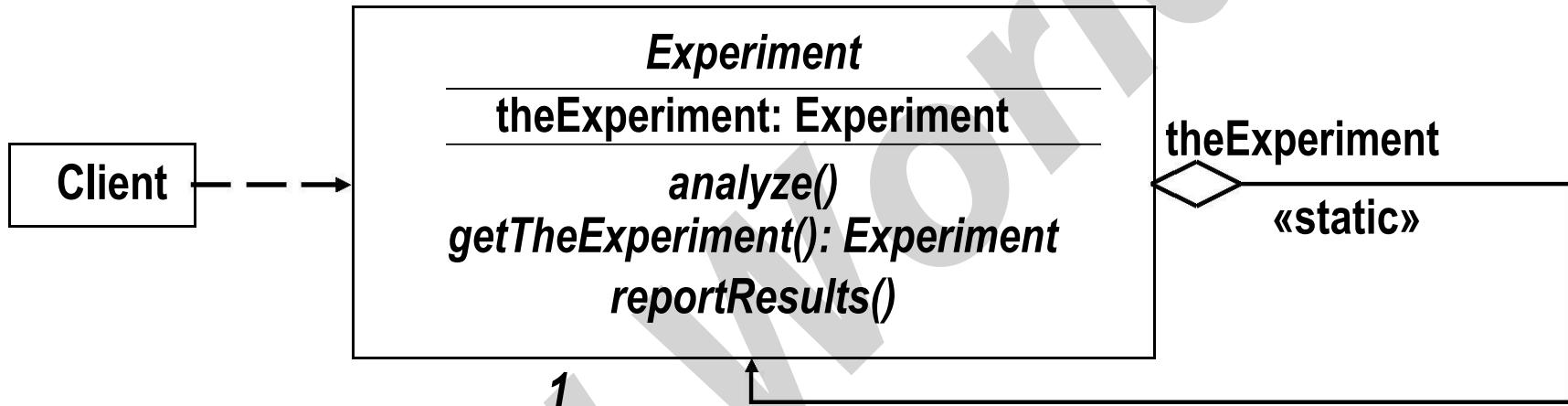
1. Make the constructor of **MyClass** **private**

```
private MyClass() { /* .... constructor code .... */ };
```

1. Define a **public static method** to access the member

```
public static MyClass getSingletonOfMyClass()
{
    return singletonOfMyClass;
}
```

Application of Singleton to Experiment Example



Key Concept: → Singleton Design Pattern

When a class must have exactly one instance, make the constructor private and the instance a private static variable with a public accessor.

Example Code

```
public class Runtime
{
    private static Runtime currentRuntime = new Runtime();

    // Returns the runtime object associated with the current
    // Java application.

    public static Runtime getRuntime()
    {
        return currentRuntime;
    }

    private Runtime() { }

}
```

7.3 - Abstract Factory Design Pattern

Abstract Factory

Design Purpose

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

Design Pattern

Capture family creation in a class containing a factory method for each class in the family.

* Gamma et al

Word Processor Interaction 1 of 2

---> Enter title:

My Life

---> Enter Heading or “-done”:

Birth

---> Enter text:

I was born in a small mountain hut

....

---> Enter Heading or “-done”:

Youth

---> Enter text:

I grew up playing in the woods ...

---> Enter Heading or “-done”:

Adulthood

....

---> Enter Heading or “-done”:

-done

(continued)

Word Processor Interaction 2 of 2: Output Options

....
>> Enter the style you want displayed:
big

Option 1

----- Title: MY LIFE -----

Section 1. --- BIRTH ---

I was born in a mountain hut

Section 2. --- YOUTH ---

I grew up sturdy ...

Section 3. --- ADULTHOOD ---

....

....
>> Enter the style you want displayed:
small

Option 2

My Life

Birth

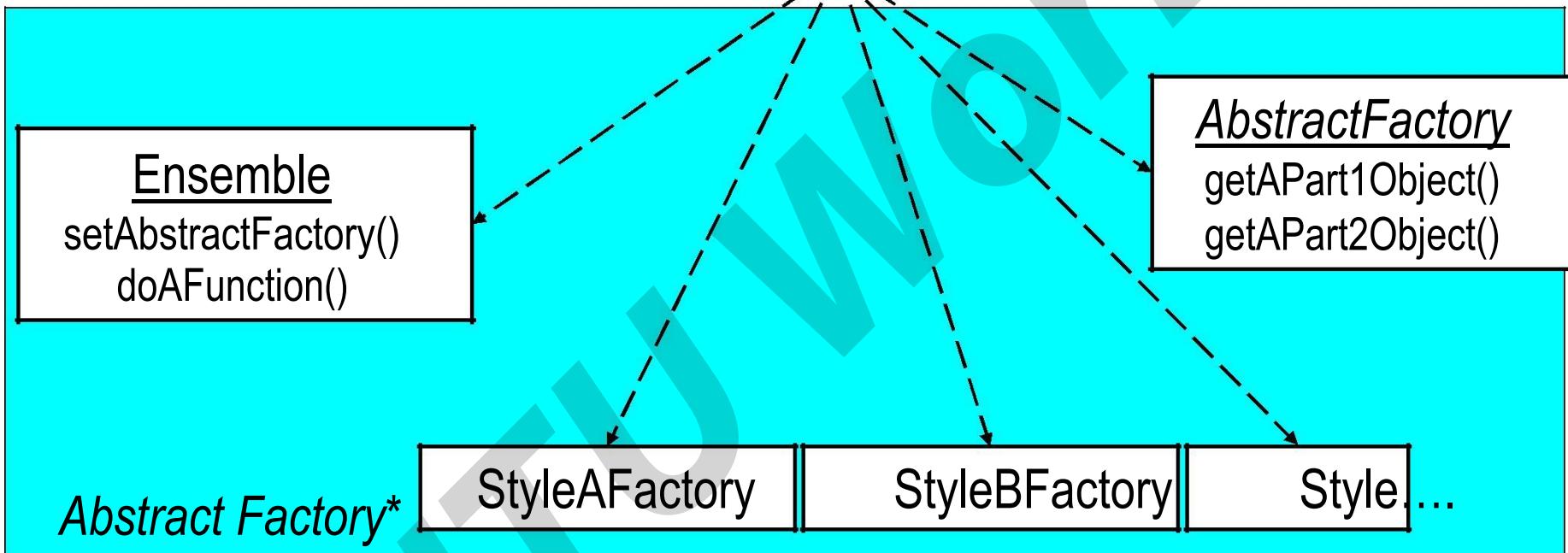
I was born in a mountain hut

Youth

I grew up sturdy ...

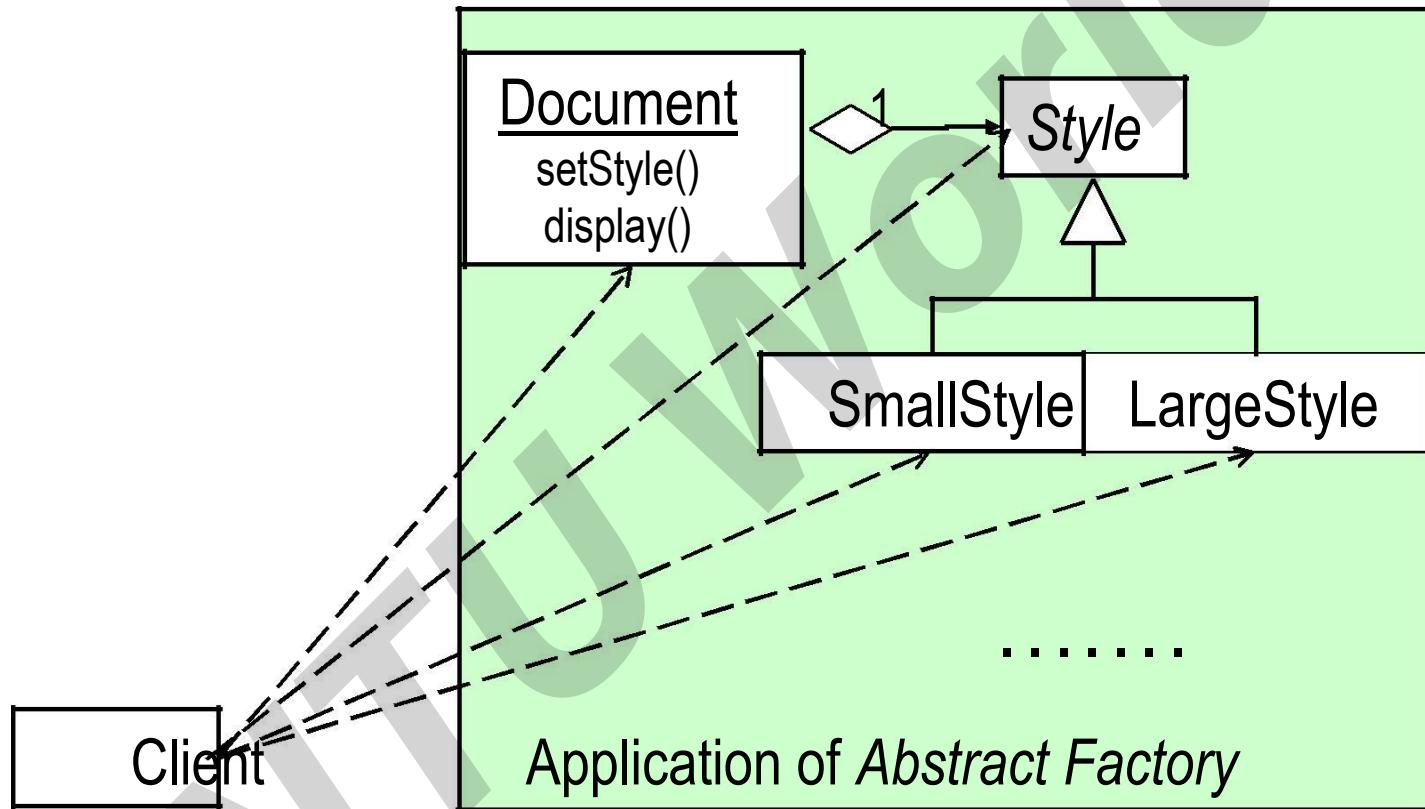
Adulthood

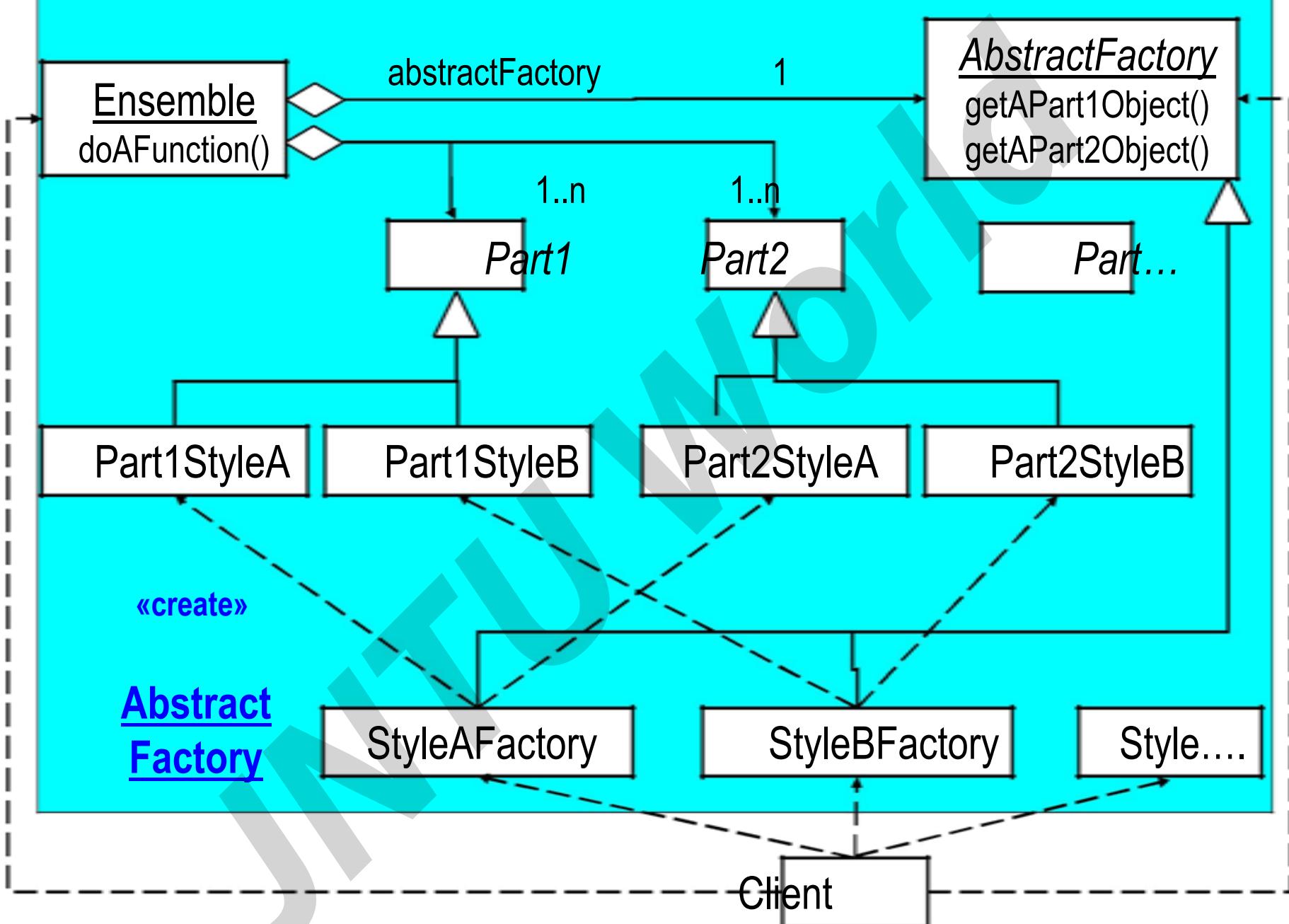
Abstract Factory Interface



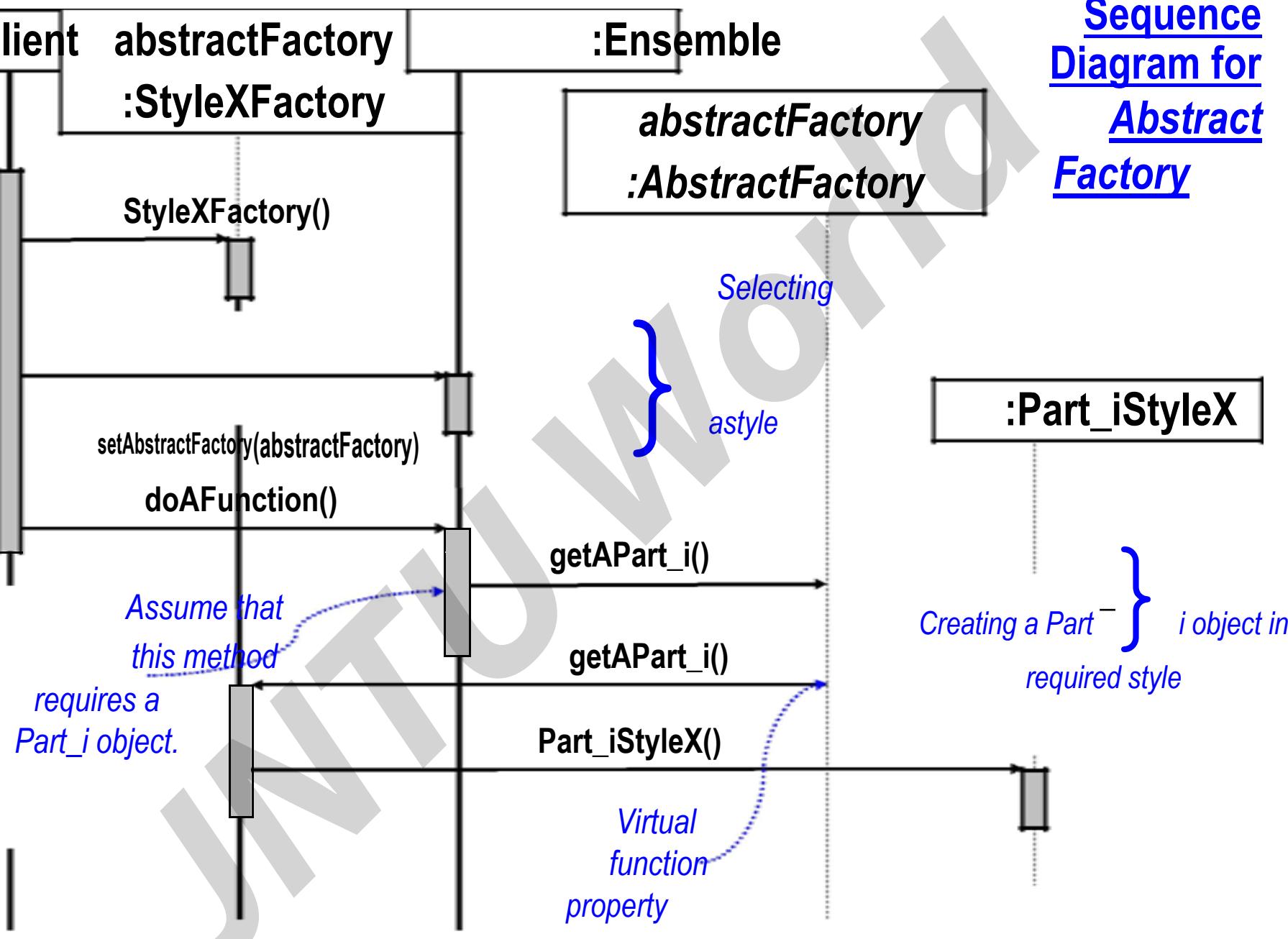
* relationships within pattern application not shown

Interface of Abstract Factory Applied to Word Processor



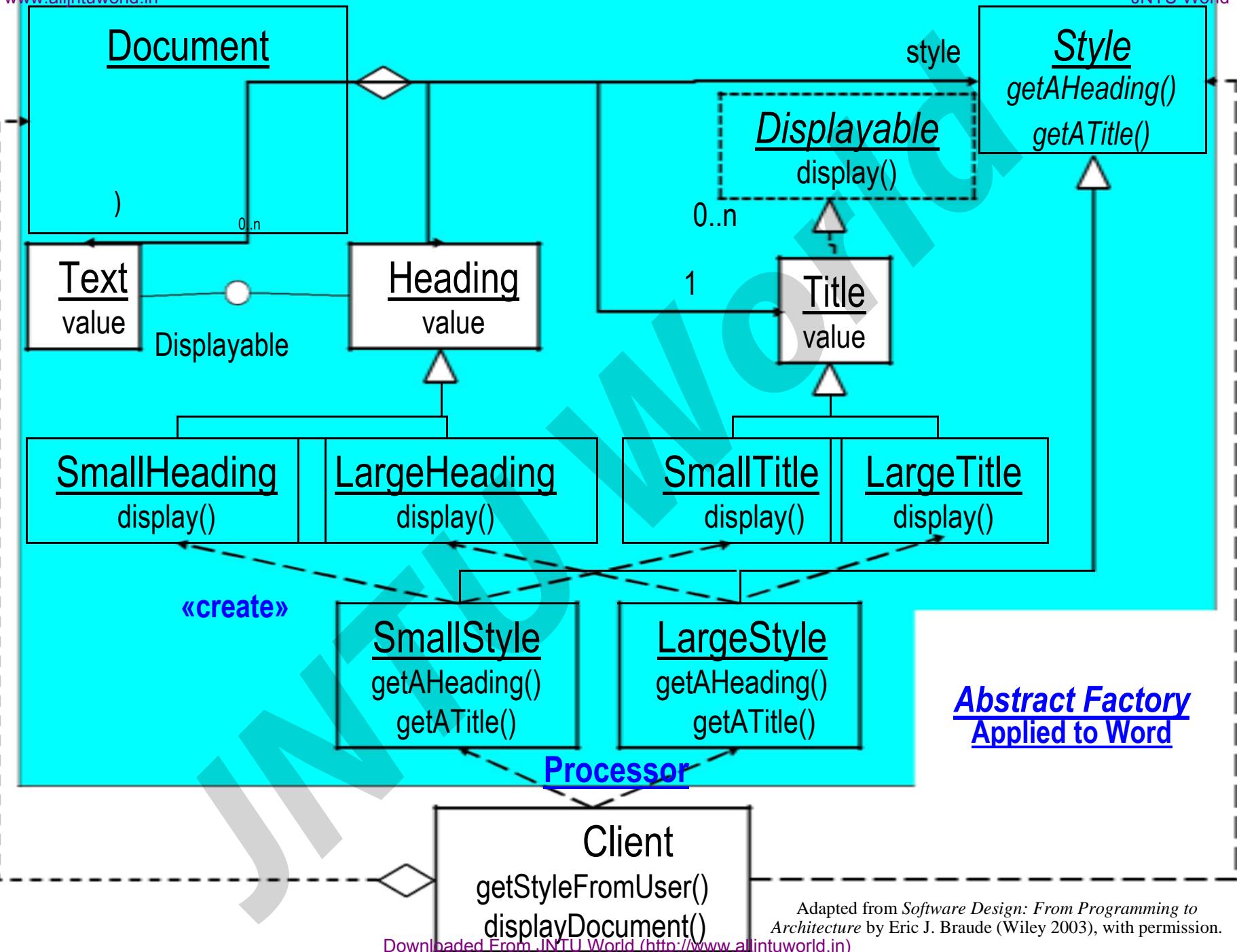


Sequence Diagram for Abstract Factory

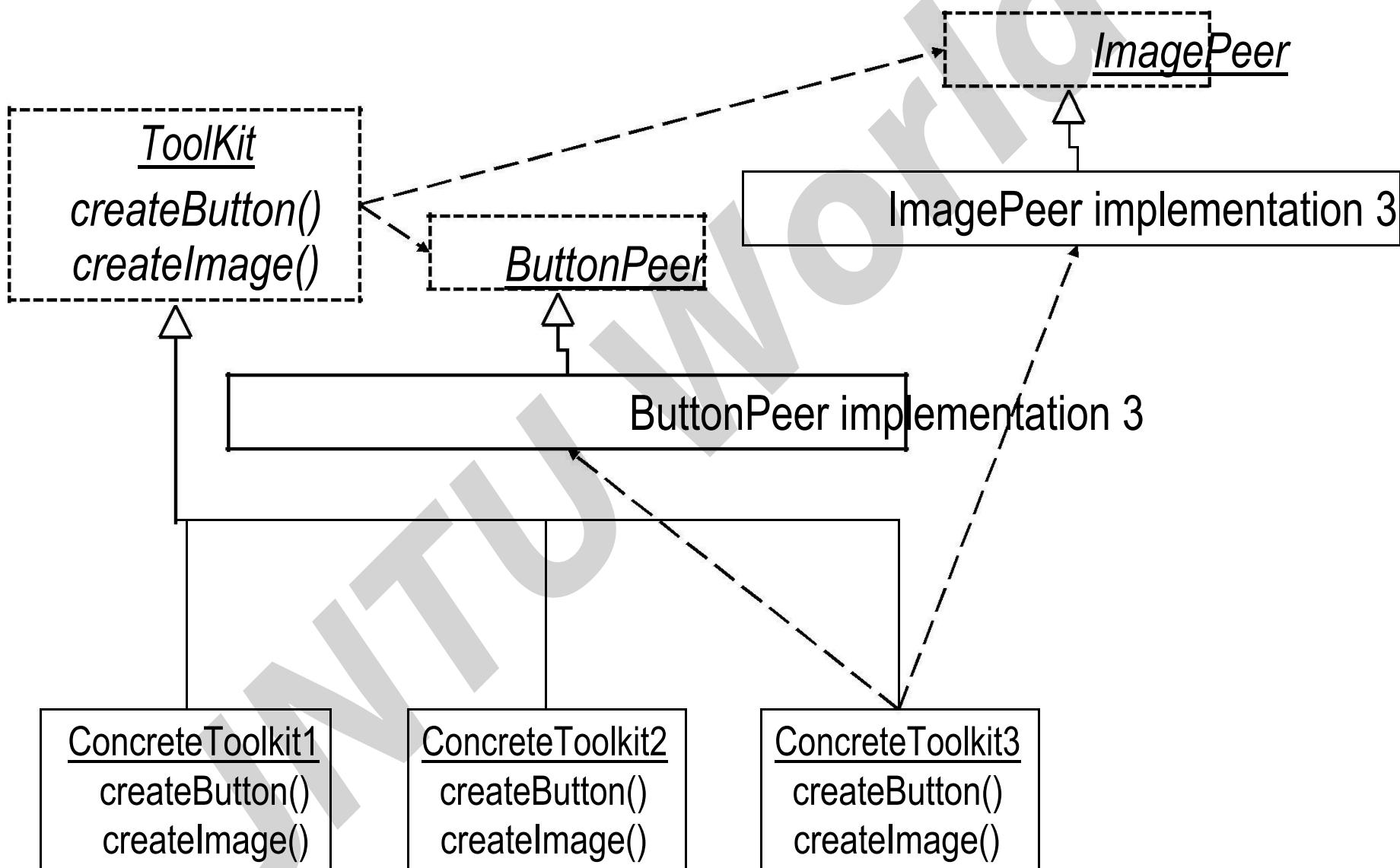


Design Goals At Work: → Correctness and Reusability ←

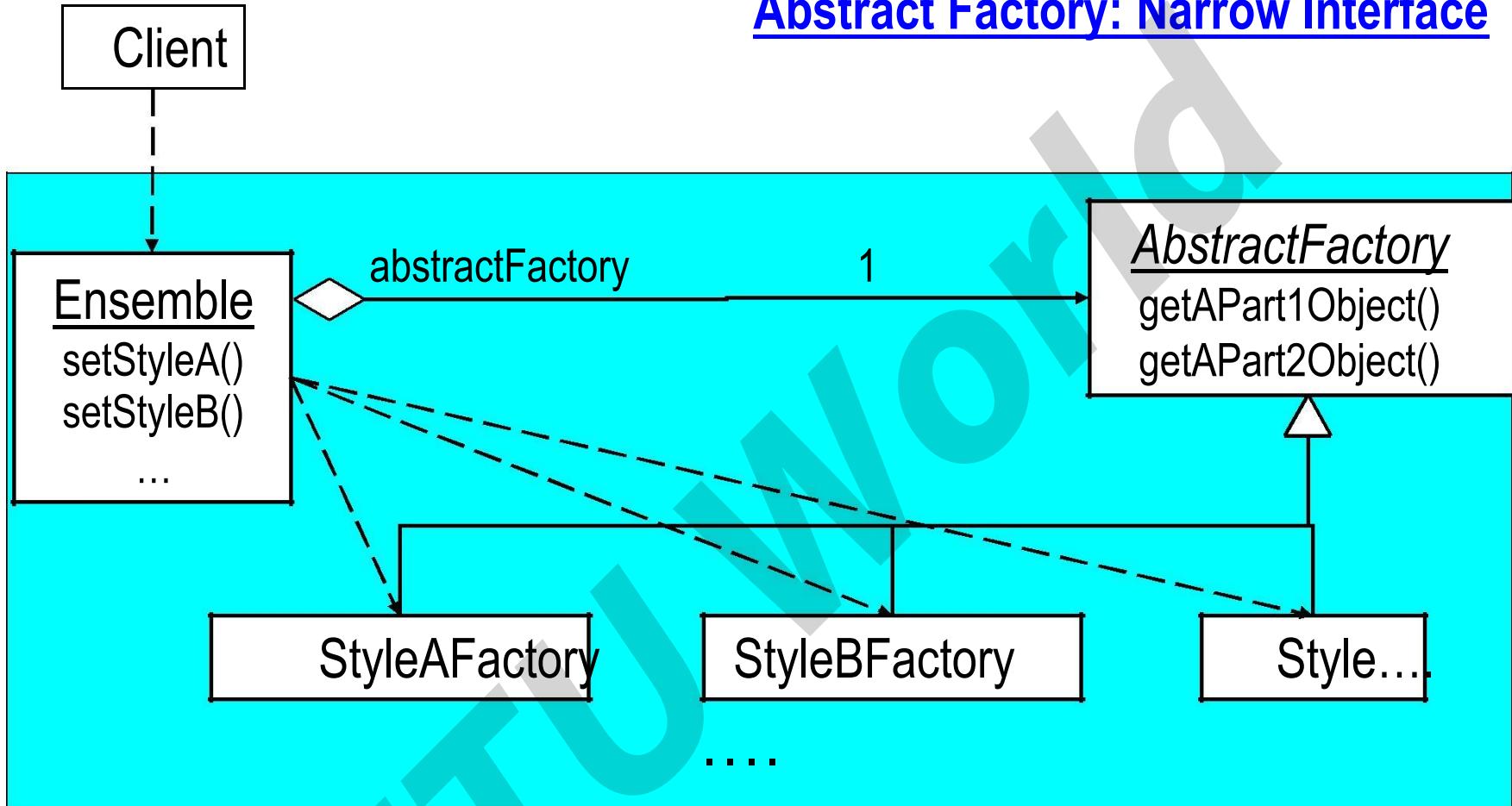
We want to separate the code parts that format the document in each style. We also want to separate the common document generation code. This facilitates reusing parts and checking for correctness.



An Abstract Factory Application: Java ToolKit



Abstract Factory: Narrow Interface



Key Concept: → Abstract Factory Design Pattern ←

To design an application in which there are several possible styles for a collection of objects, capture styles as classes with coordinated factory methods.

7.4 - Prototype Design Pattern

Prototype

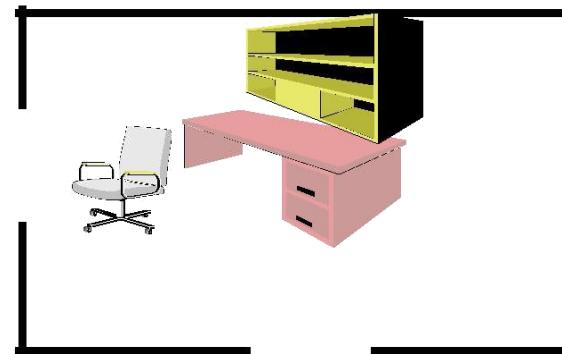
Design Purpose

Create a set of almost identical objects whose type is determined at runtime.

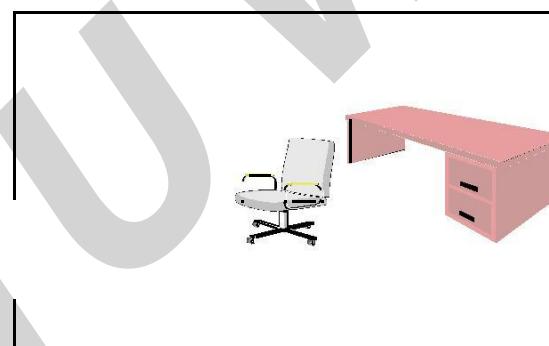
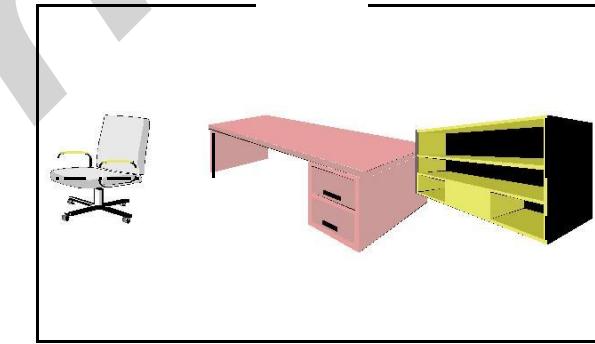
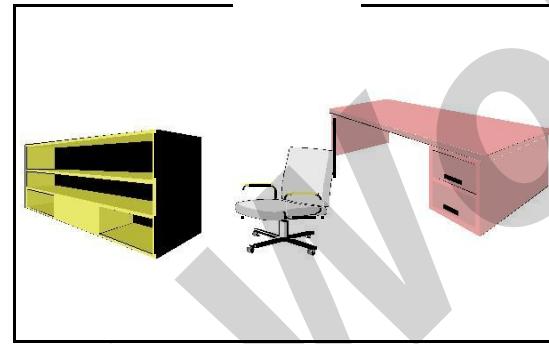
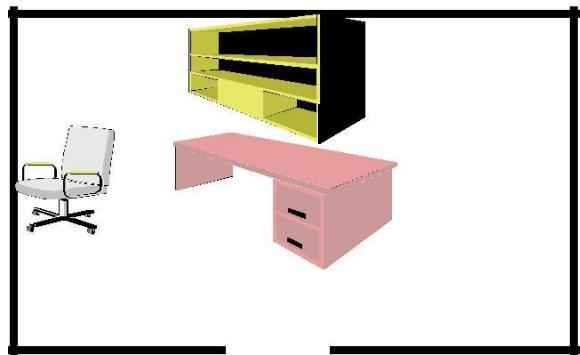
Design Pattern

Assume that a prototype instance is known; clone it whenever a new instance is needed.

Prototype Design Example: A Selection



Graphics courtesy COREL



Furniture
color



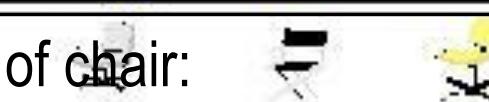
Click on choice of desk:



Click on choice of storage:



Click on choice of chair:



Furniture
hardware
type

colonial

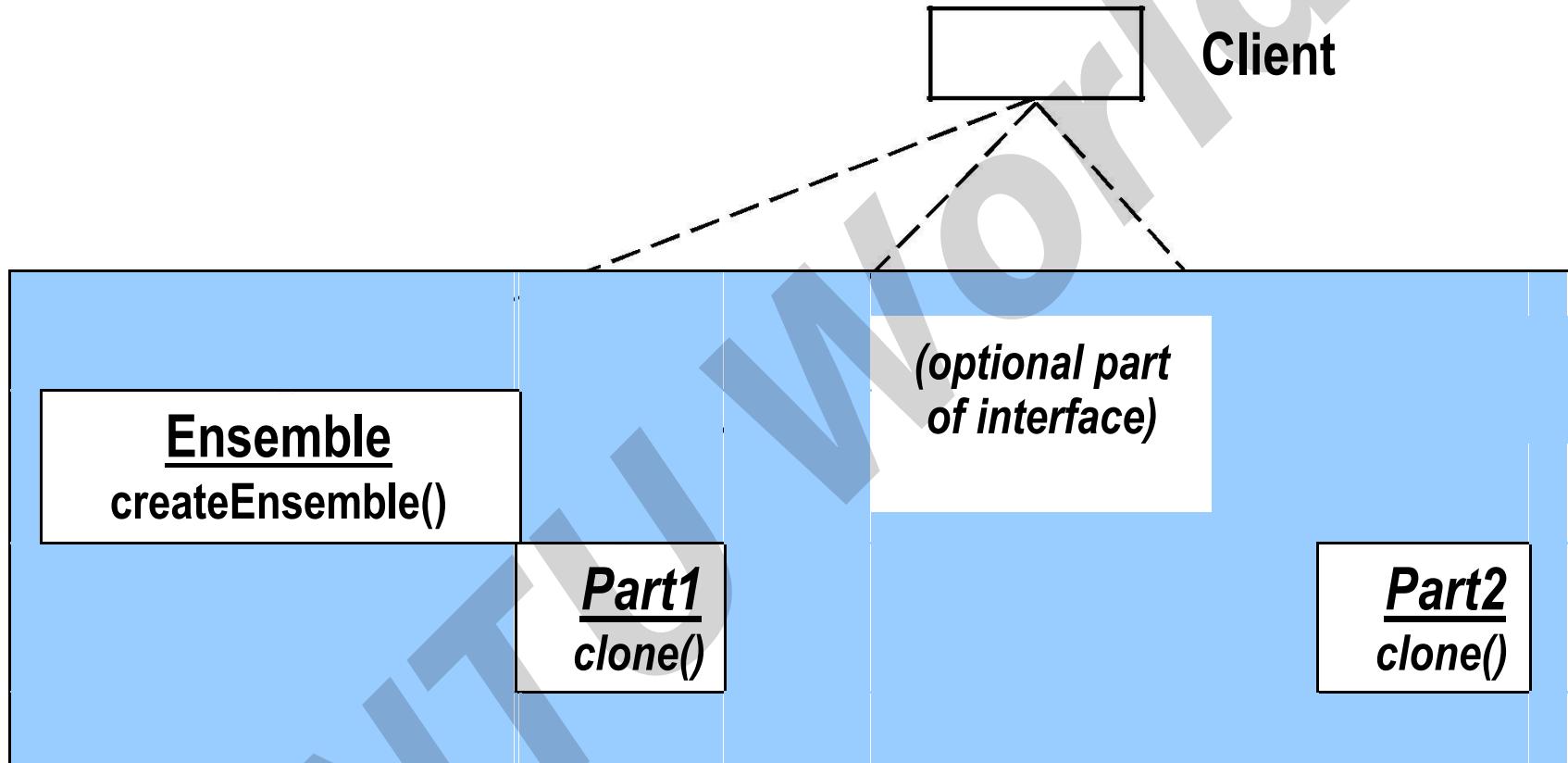
A Simplified Prototype Example

The screenshot illustrates a user interface for a furniture configuration tool. The top section displays a main workspace with four smaller windows labeled "Office" showing 3D models of a desk, chair, and storage unit. Below this is a row of three desks and two storage units. At the bottom, there are three rows of buttons for selecting furniture components:

- Click on choice of desk:** Three options: a red desk with drawers, a yellow L-shaped desk, and a green modular desk.
- Click on choice of storage:** Two options: a black filing cabinet and a brown chest.
- Click on choice of chair:** Three options: a white office chair, a black task chair, and a yellow swivel chair.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Downloaded From JNTU World (<http://www.alljntuworld.in>)

Prototype Interface With Clients

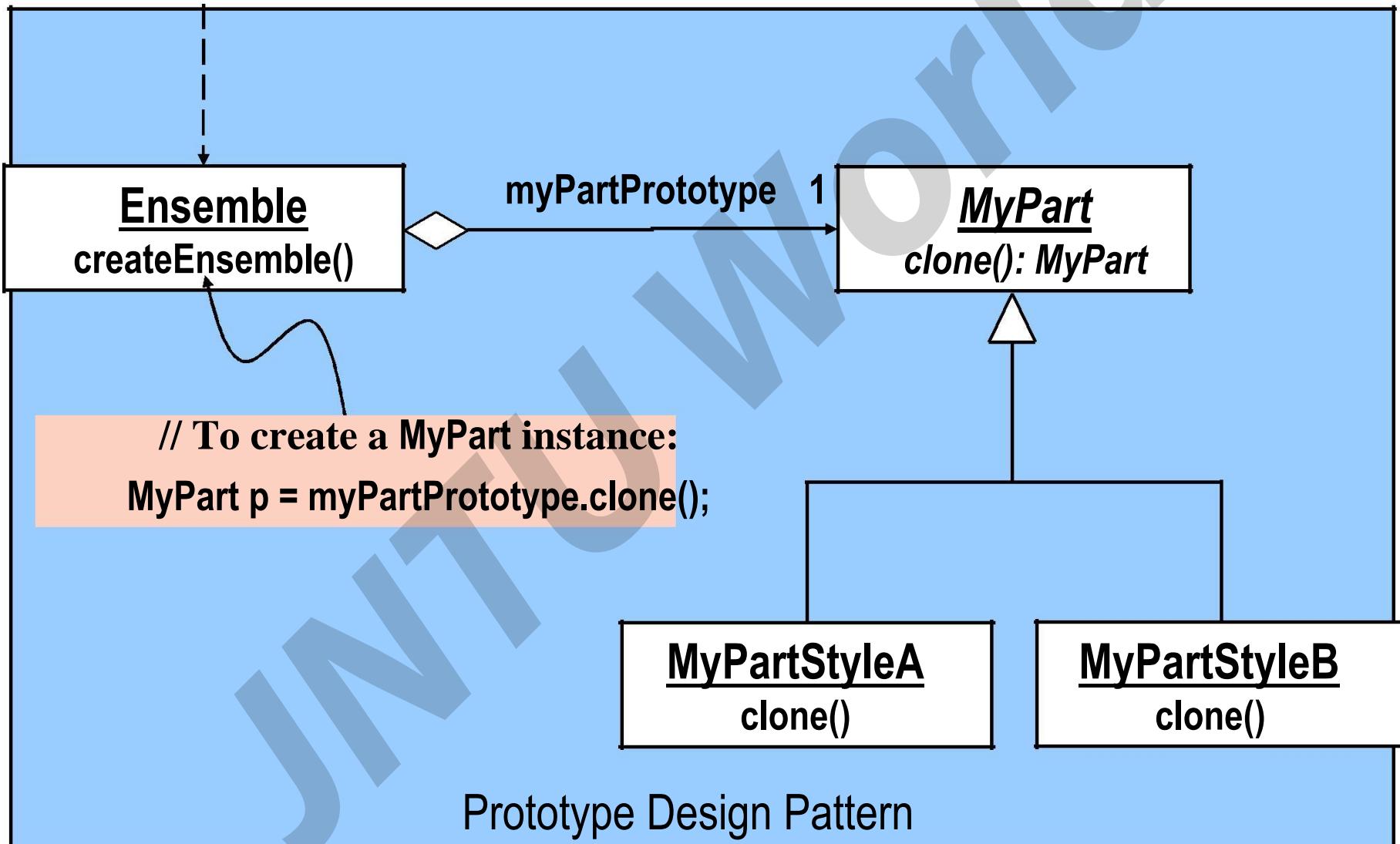


Code Example

```
OfficeSuite myOfficeSuite =  
OfficeSuite.createOfficeSuite( myDesk, myChair, myStorage );  
  
myGUI.add(myOfficeSuite);  
myOfficeSuite.setBackground("pink");
```

Client

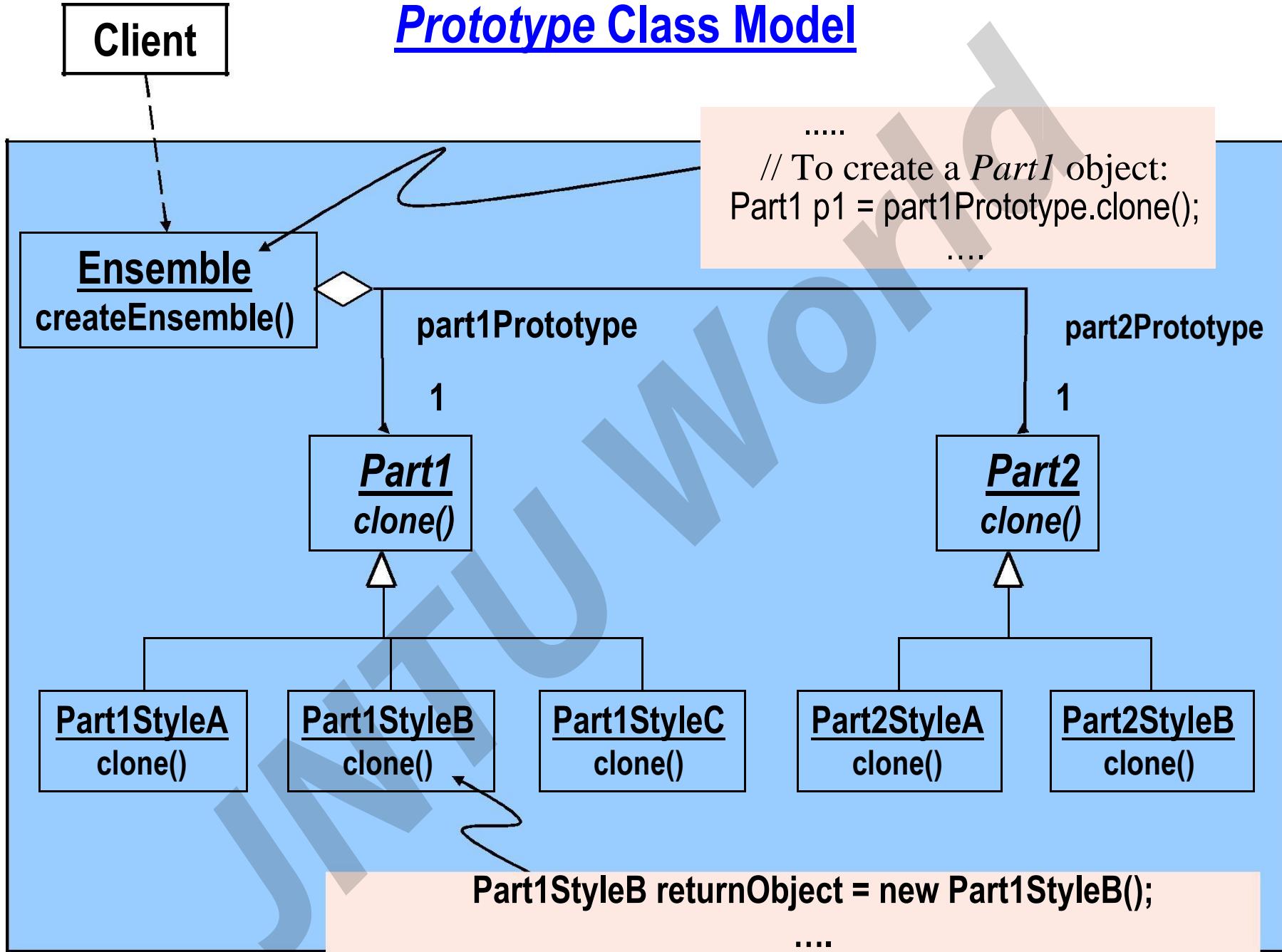
The Prototype Idea



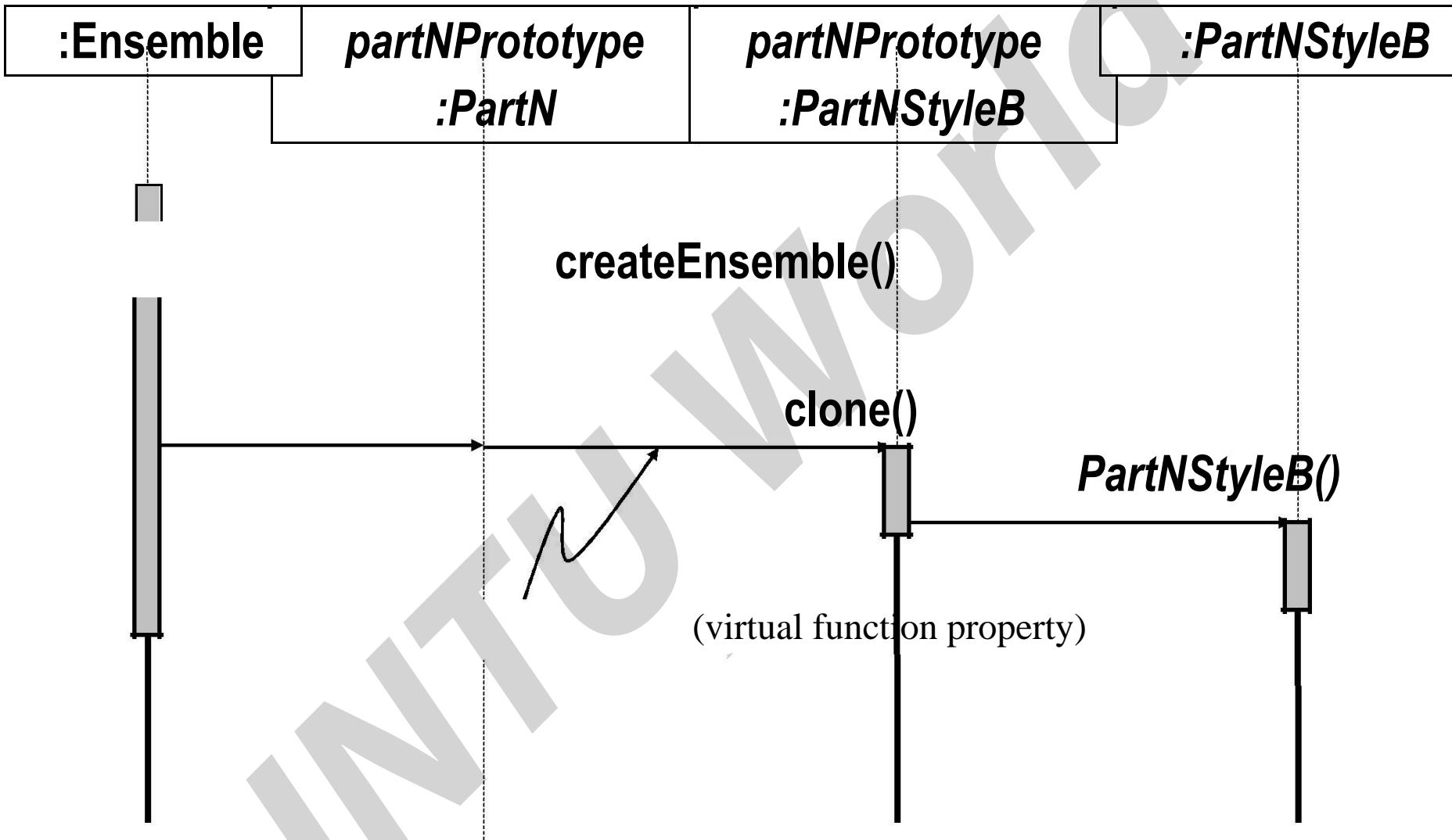
Code Example

```
Ensemble EnsembleA Ensemble.createEnsemble(.. . .);  
  
Ensemble EnsembleB Ensemble.createEnsemble();  
  
// This code is inside the Ensemble class MyPart  
    anotherMyPart = MyPartPrototype.clone();  
MyPart yetAnotherMyPart = MyPartPrototype.clone();
```

Prototype Class Model



Sequence Diagram for Prototype



Contrasting Abstract Factory and Prototype

Prototype allows the client to select any chair style, any desk style, and any cabinet style

This is all done separately rather than have to select an overall office style

Nevertheless, the client wants to keep a single style of chair and a single style of desk throughout the office suite

Design Goals At Work:

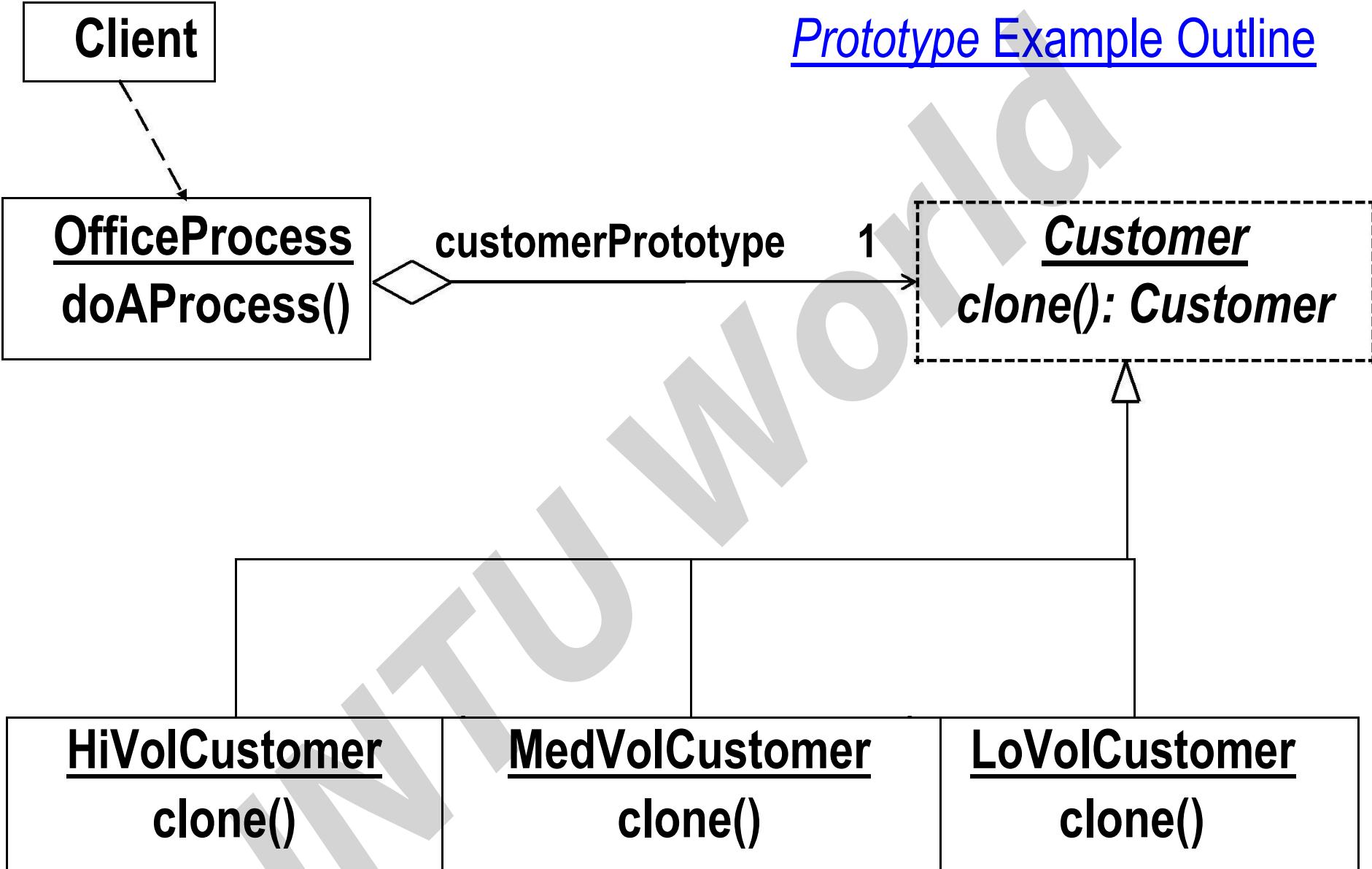


Correctness and Reusability



We want to isolate the parts pertaining to each type of customer. We also want to isolate the common customer code. This makes it easier to check the design and implementation for correctness, and to reuse the parts.

Prototype Example Outline



Given:

Requirement for (Deep) Cloning

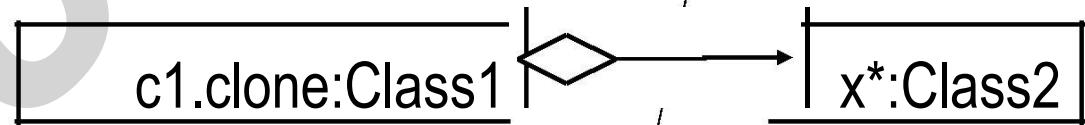
(1) Class model:



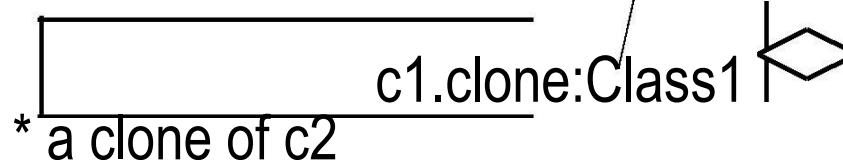
(2) $c1$ an instance of Class1 :



$c1.\text{clone}$ should be as follows (deep clone):



In shallow cloning, $c1.\text{clone}$ actually as follows!



Key Concept: → Prototype Pattern ←

-- when designing for multiple instances
which are the same in key respects,
create them by cloning a prototype.

UNIT-IV & V STRUCTURAL PATTERNS-I & II

Structural Design Patterns

Facade

Decorator

Composite

Adapter

Flyweight

Proxy

Façade Design Pattern

Facade

Design Purpose

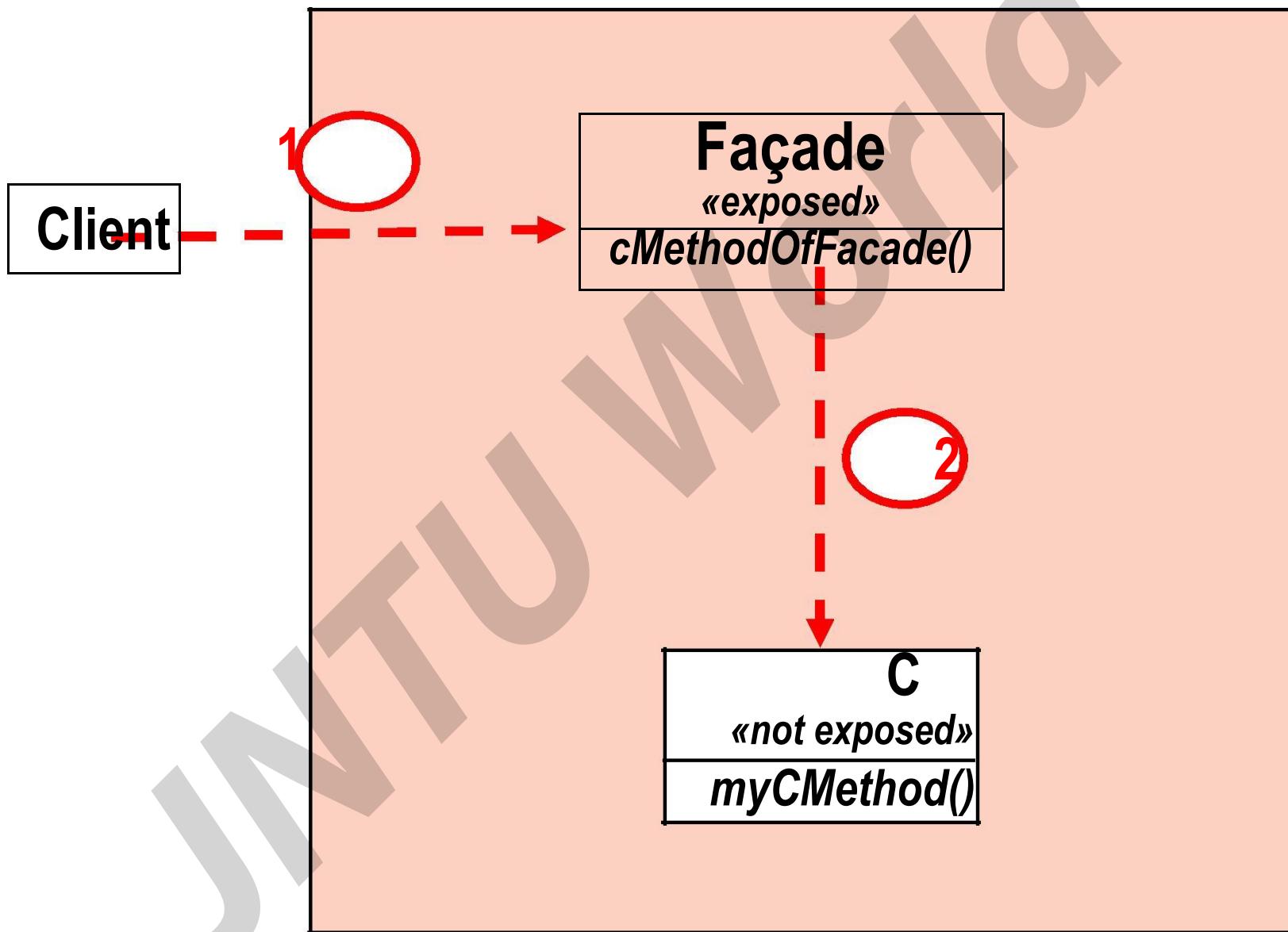
Provide an interface to a package of classes

Design Pattern Summary

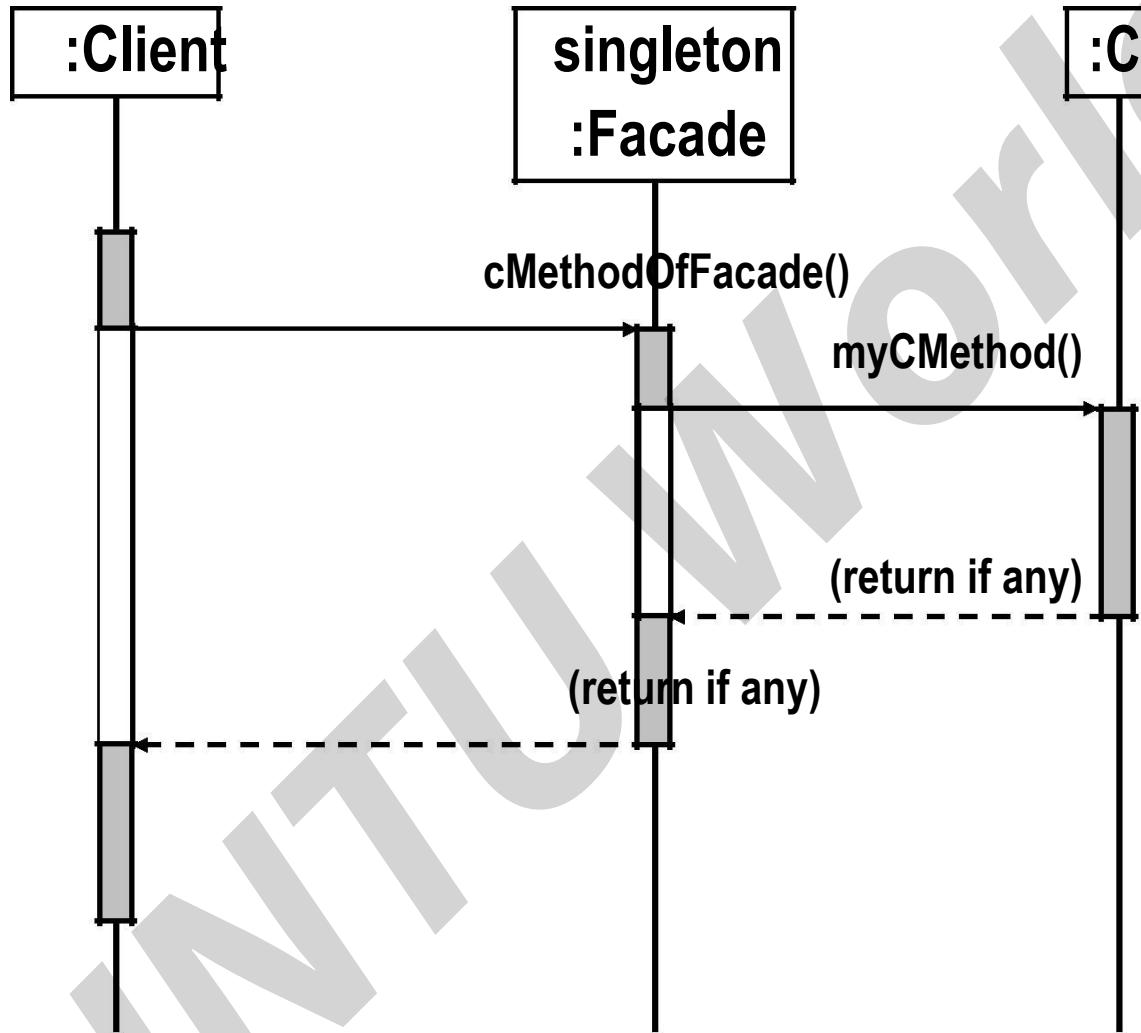
Define a class (typically a singleton) that is the sole means for obtaining functionality from the package.

Notes: the classes need not be organized as a package; more than one class may be used for the façade.

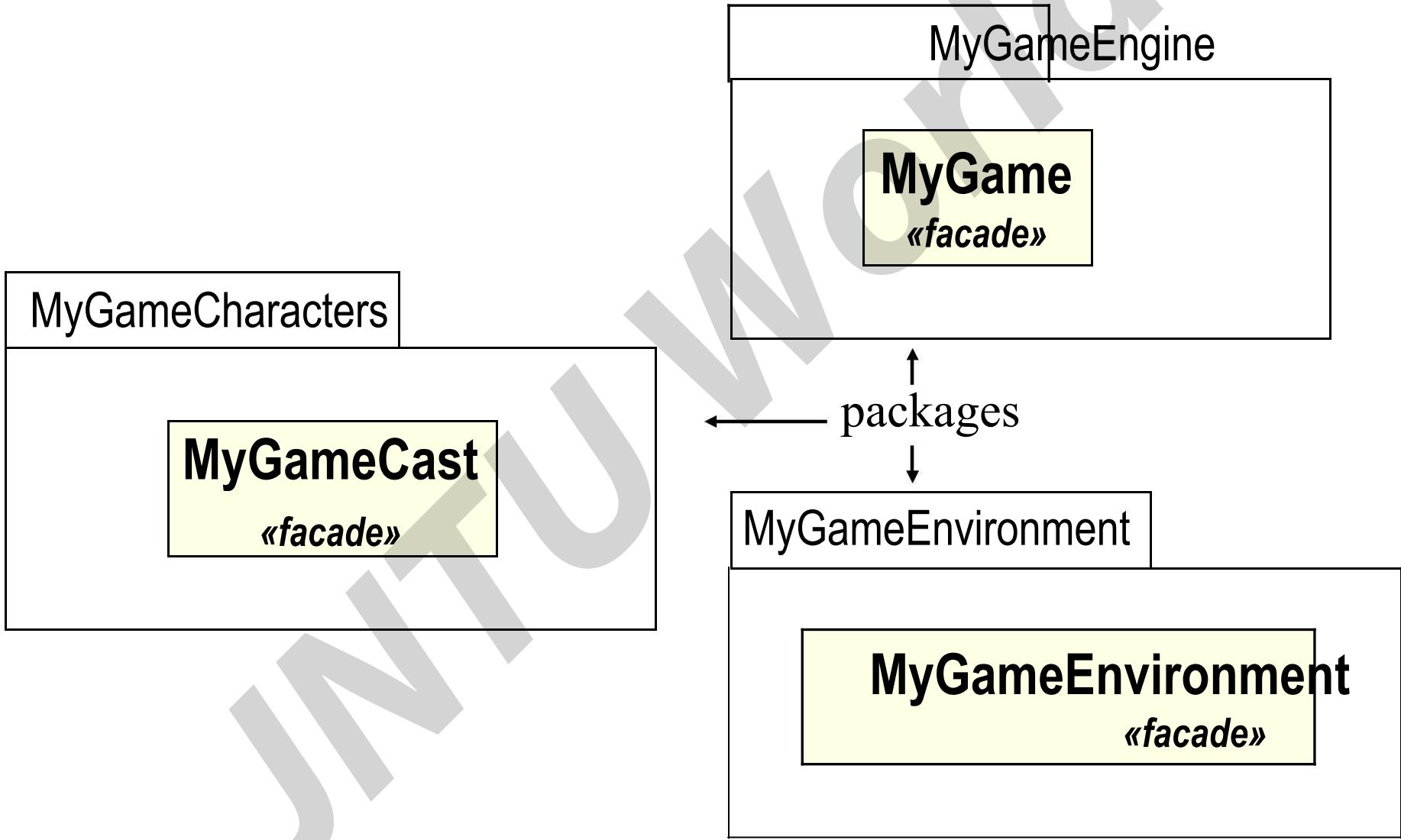
Facade Design Pattern Structure



Sequence Diagram for Façade

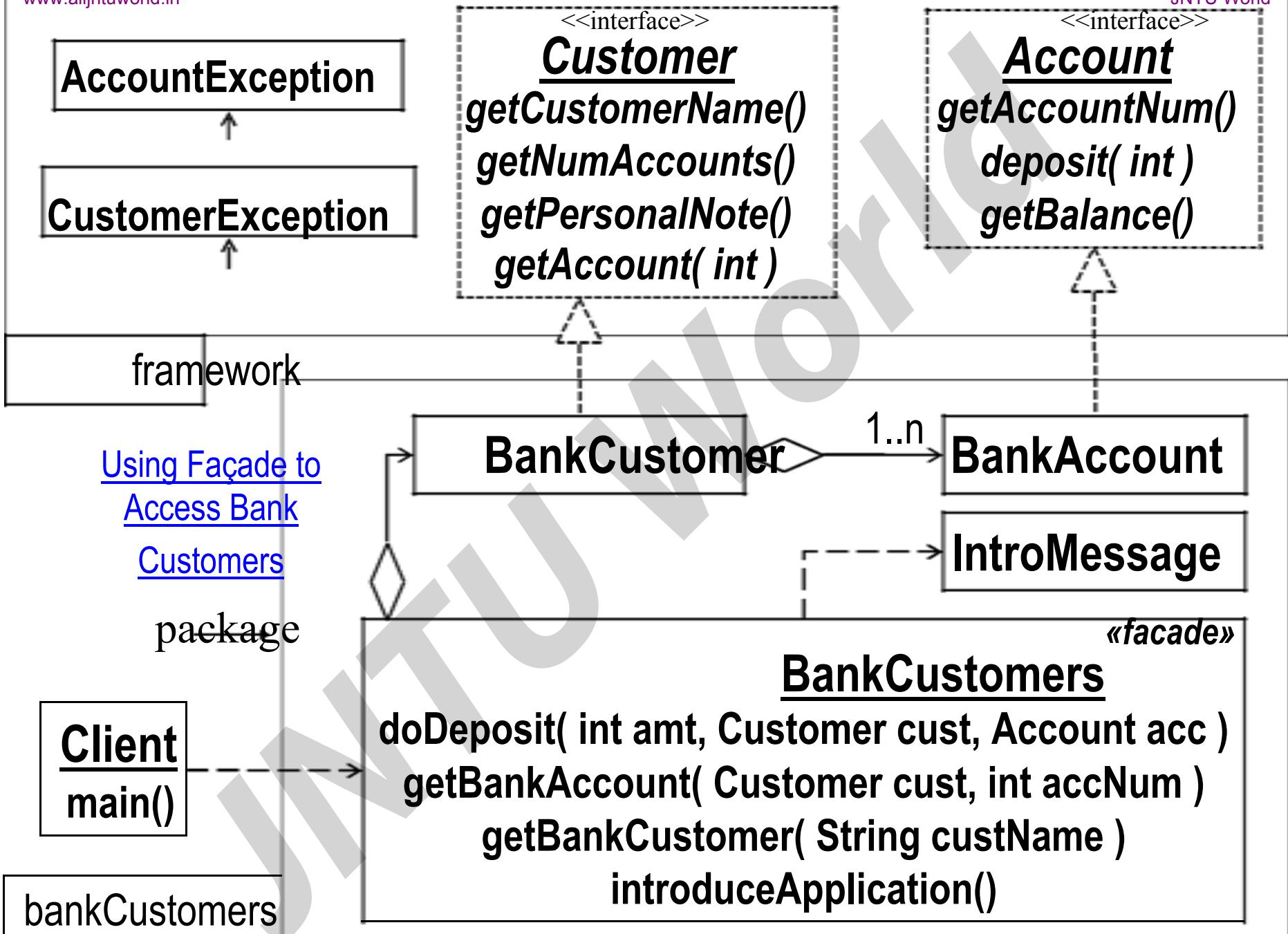


Using Façade for Architecture of a Video Game



Design Goals At Work: → Correctness and Reusability ←

Collecting customer-related classes in a package with a clear interface clarifies the design, allows independent verification, and makes this part reusable.



Key Concept: → Facade Design Pattern ←

-- modularizes designs by hiding complexity (similar to the web services provided by a web site)

Decorator Design Pattern

Decorator

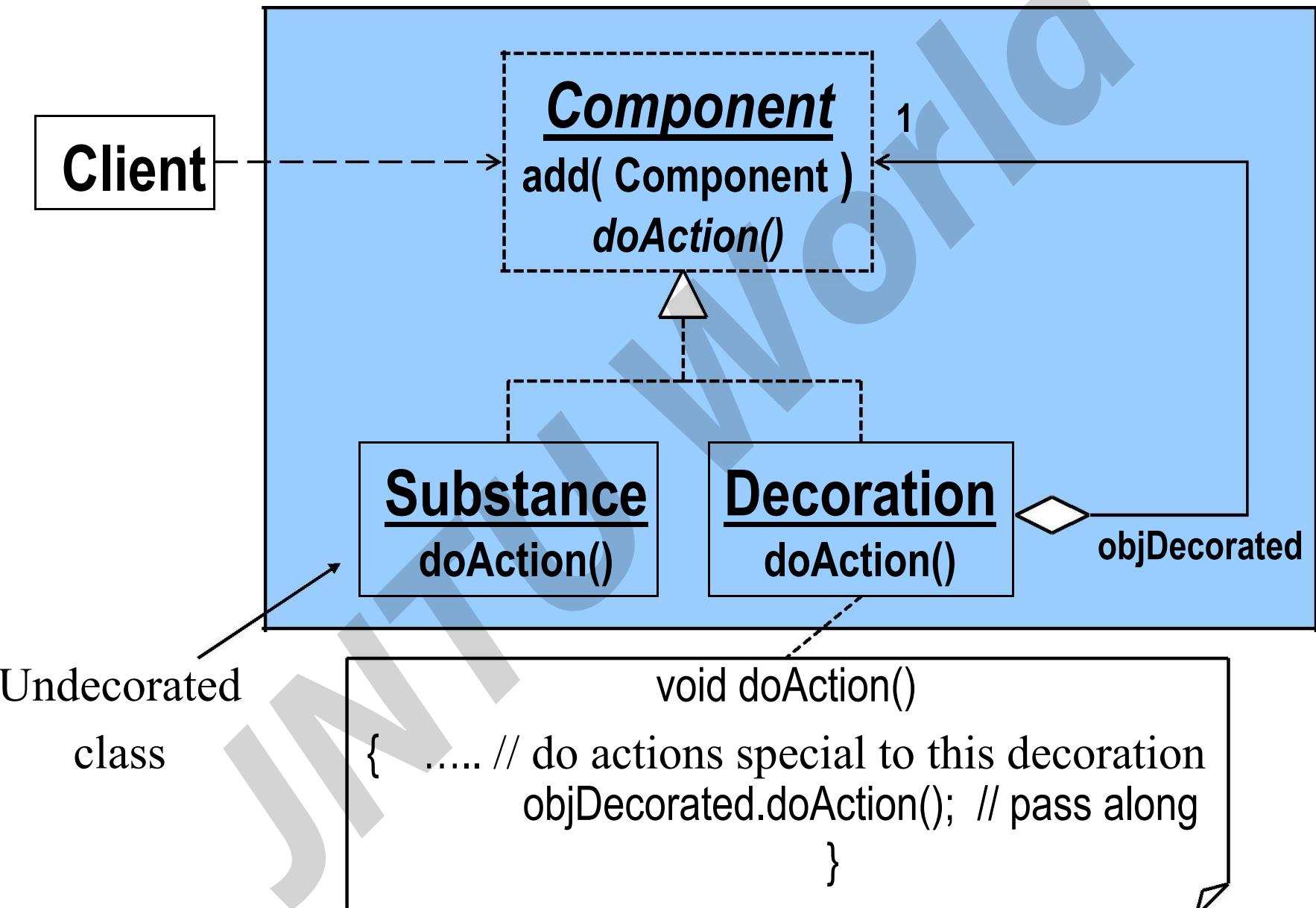
Design Purpose

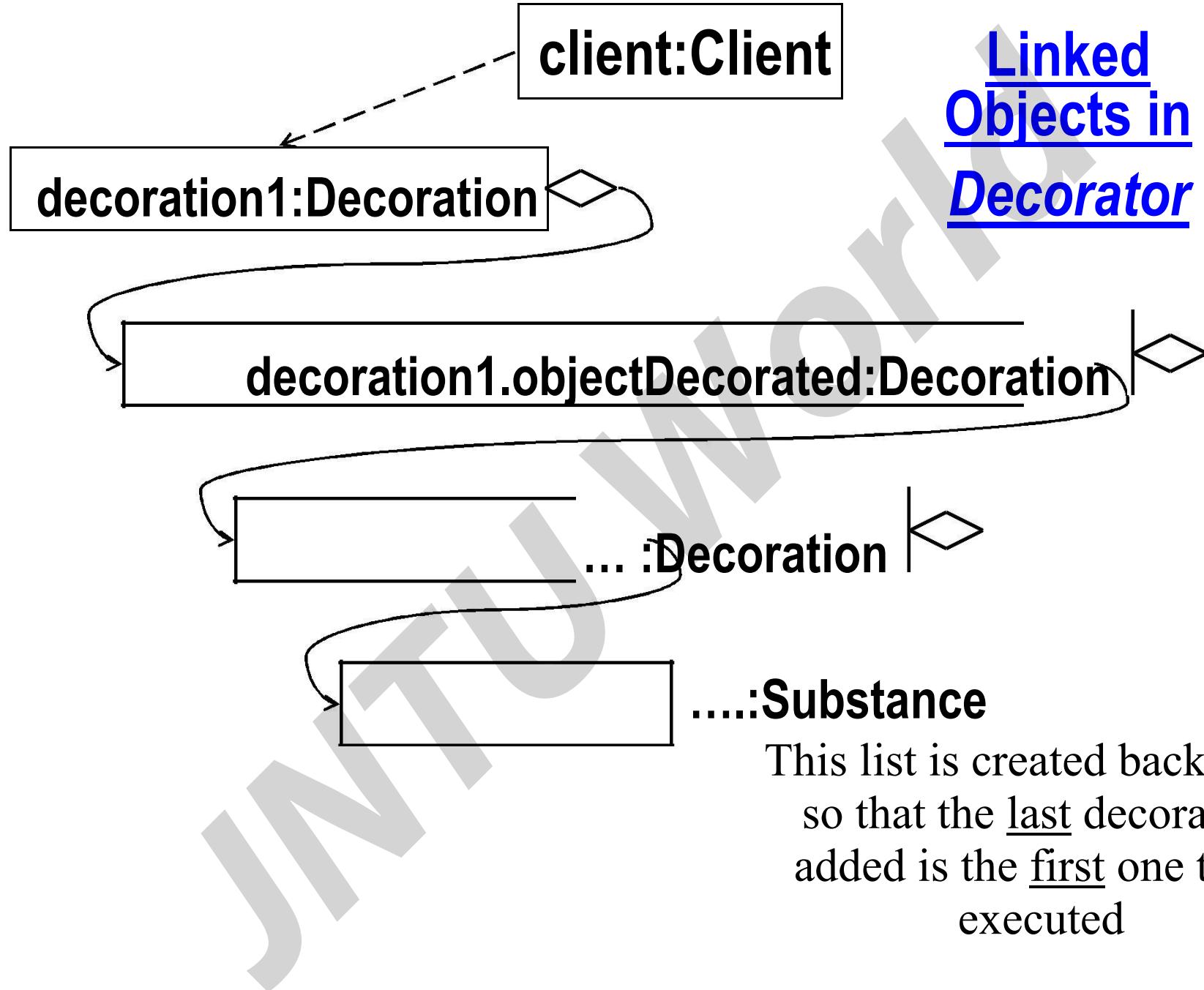
Add responsibilities to an object at runtime.

Design Pattern Summary

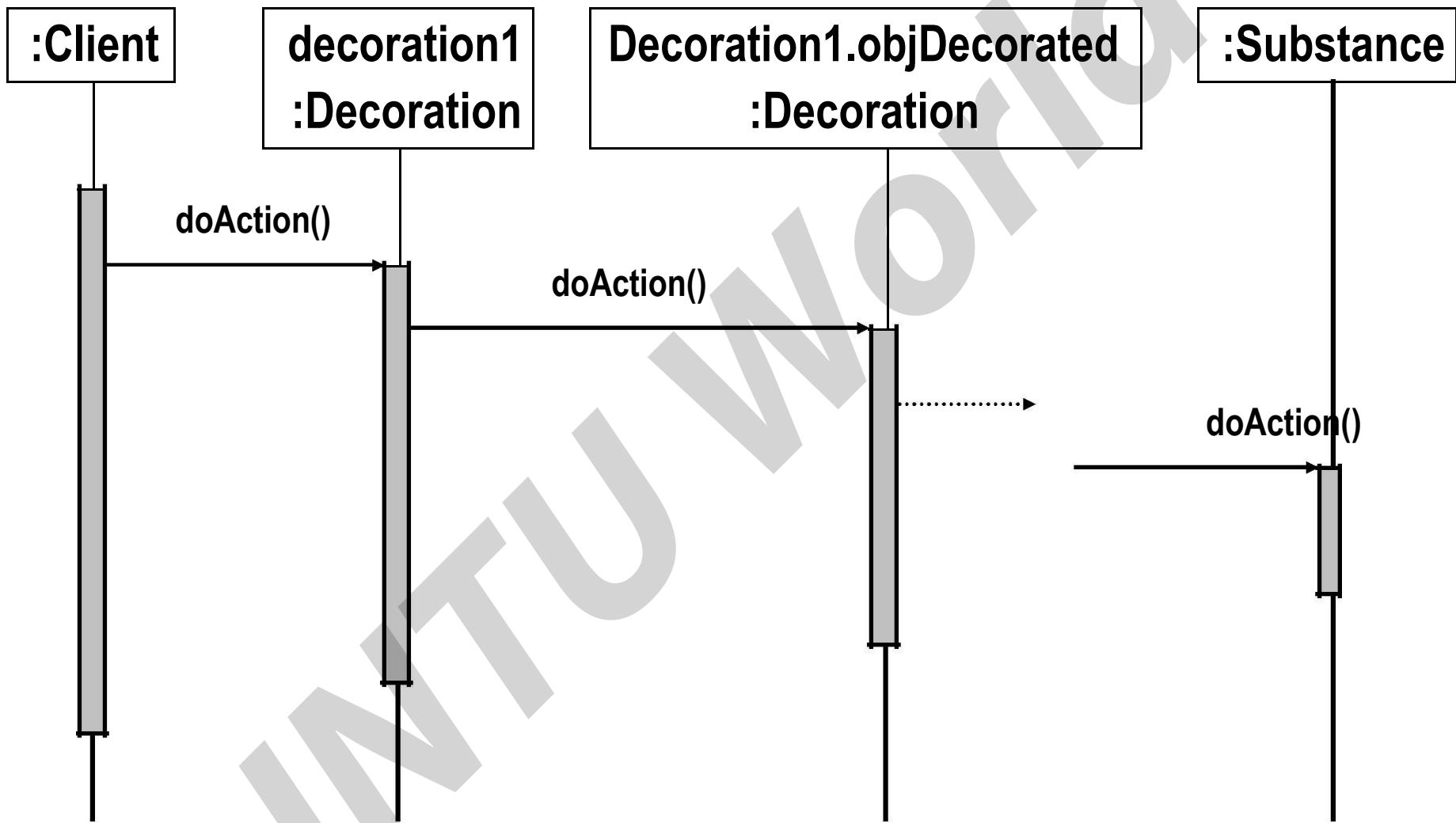
Provide for a linked list of objects,
each encapsulating responsibility.

Decorator Class Model



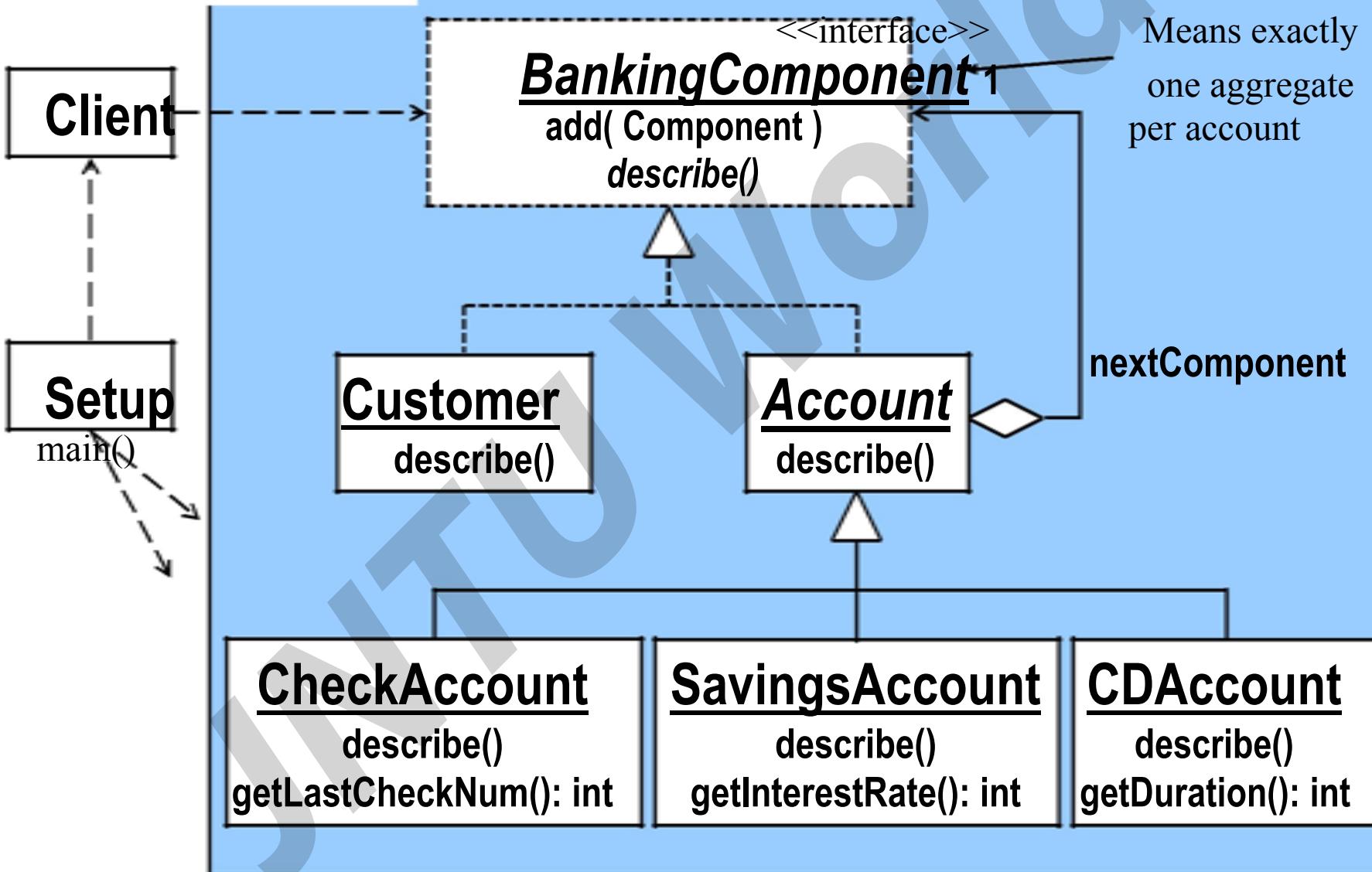


Sequence Diagram for Decorator



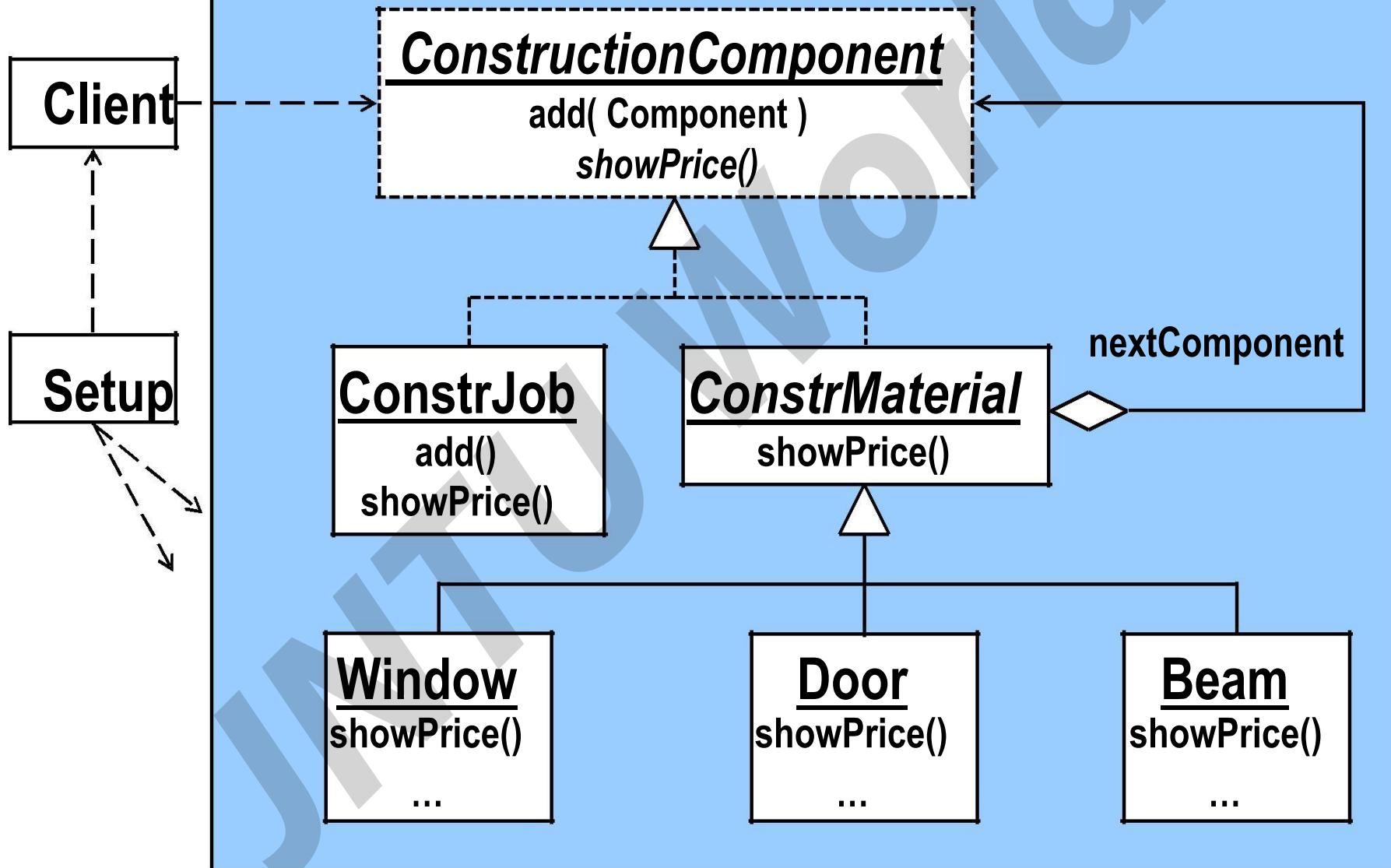
Decorator Applied to

Customer / Accounts Example

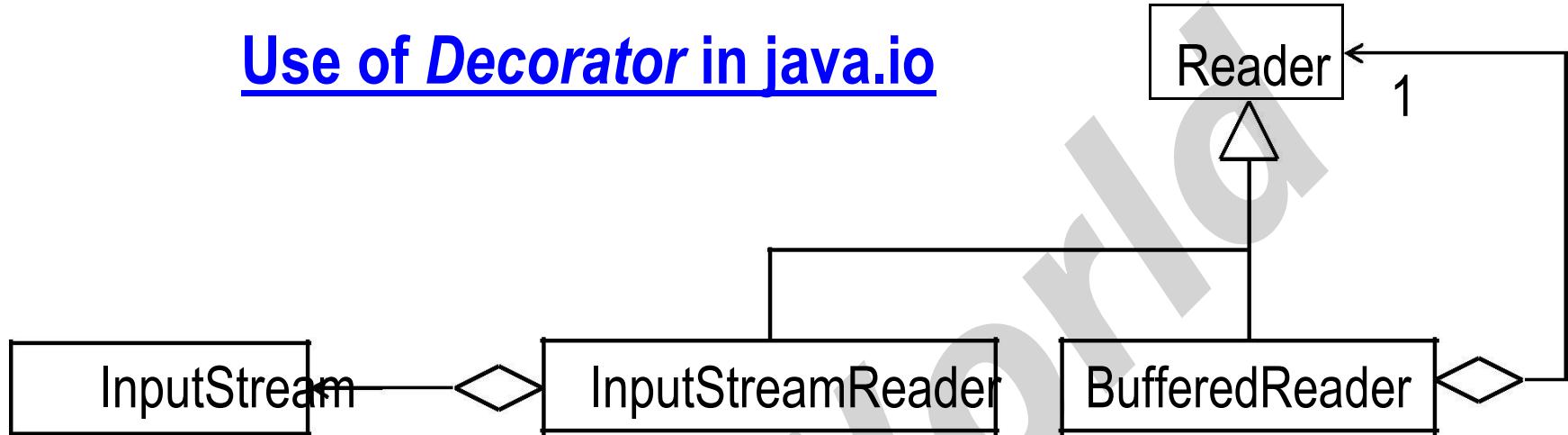


Decorator Applied to Construction Example

construction



Use of Decorator in java.io



```
BufferedReader bufReader = new BufferedReader  
    (new InputStreamReader(System.in) );
```

Key Concept: → Decorator Design Pattern ←

- \} allows addition to and removal from objects at runtime
- \} represents an object version of a linked list where a client does not need to know about the subclasses in the list
- \} emphasizes the structural properties of the substance in the list

Composite Design Pattern

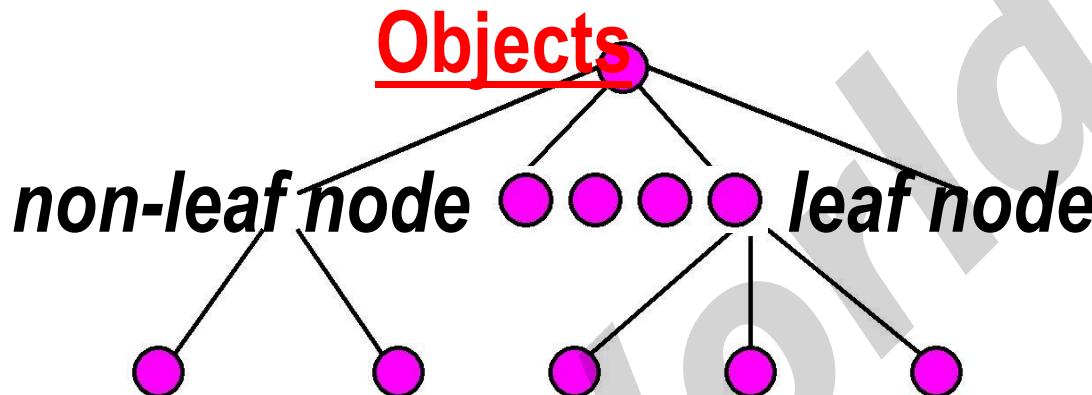
Composite Design Purpose

Represent a Tree of Objects

Design Pattern Summary

Use a recursive form in which the tree class aggregates and inherits from the base class for the objects.

Basis for Composite Class Model



Classes

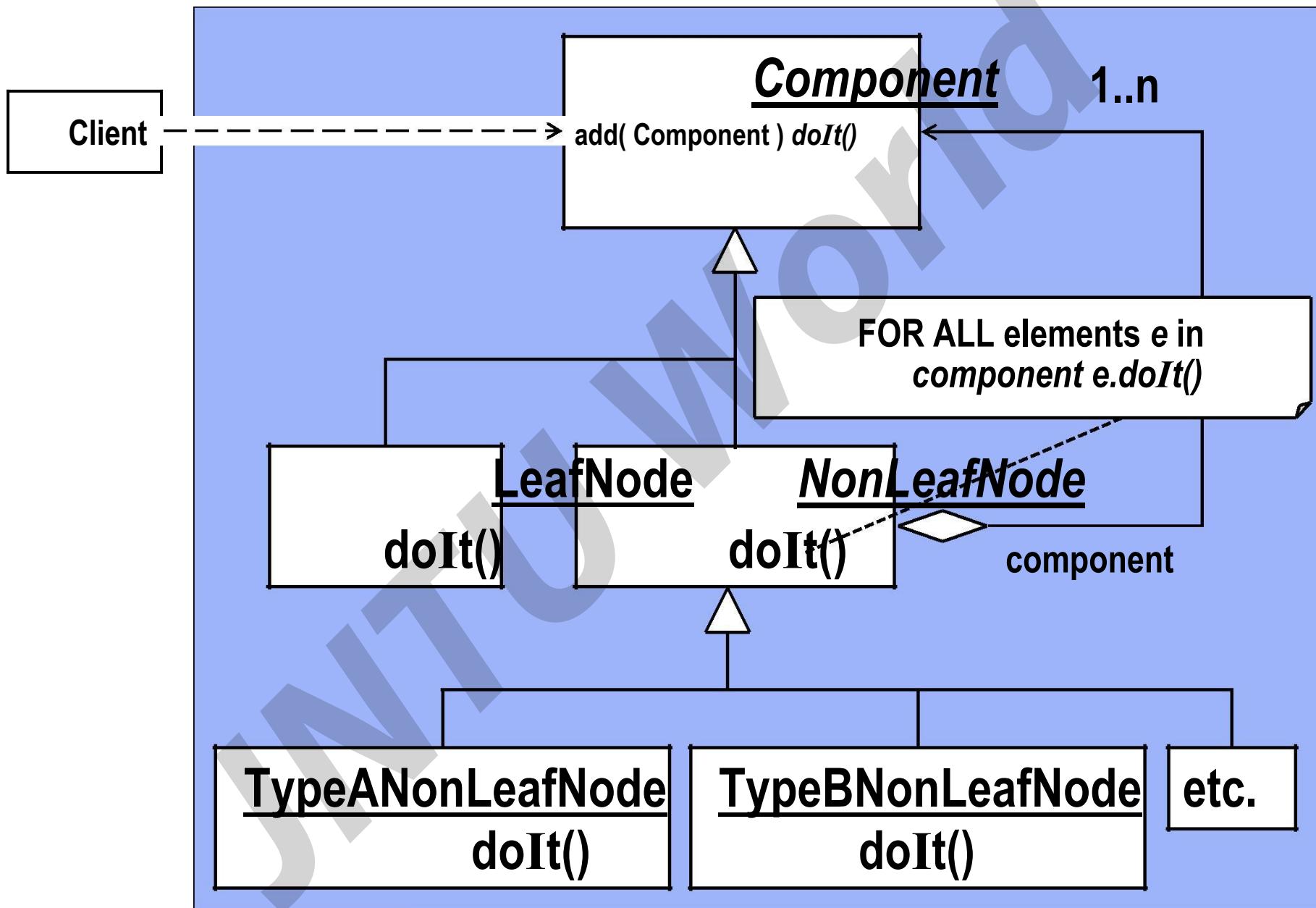
“every object involved
is a **Component** object”



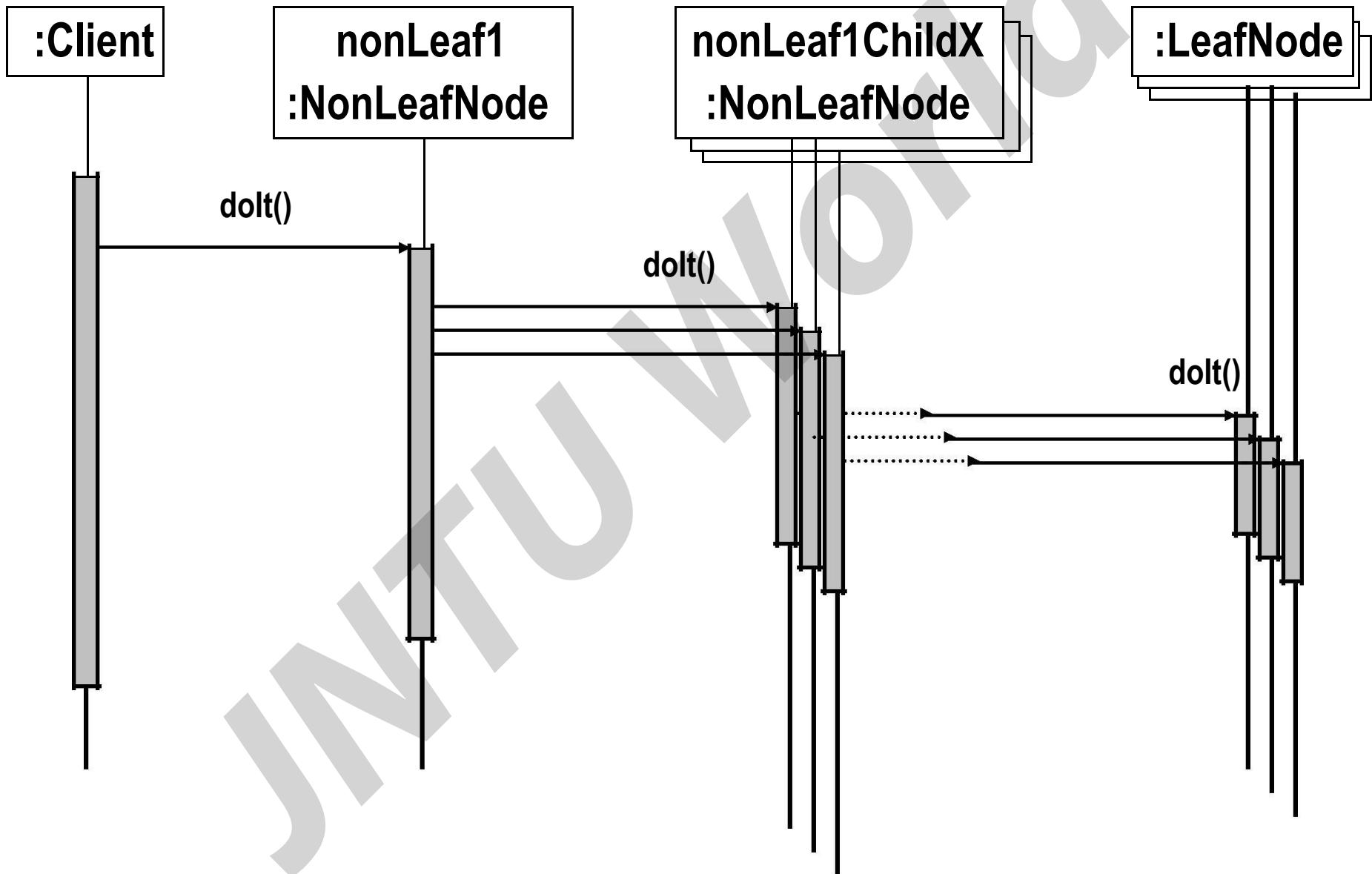
“non-leaf nodes
have one or more
components”



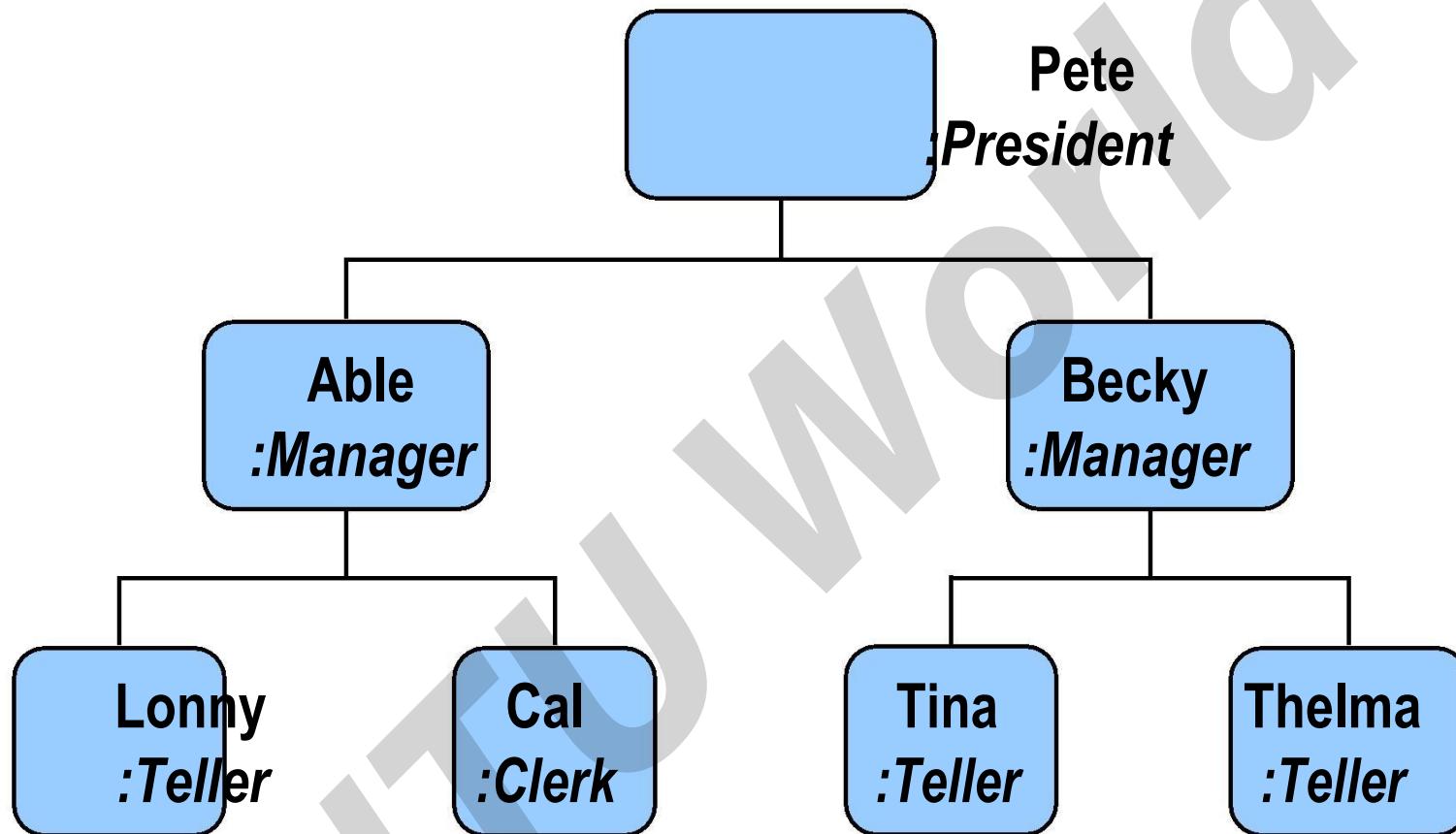
Composite Class Model



Sequence Diagram for Composite



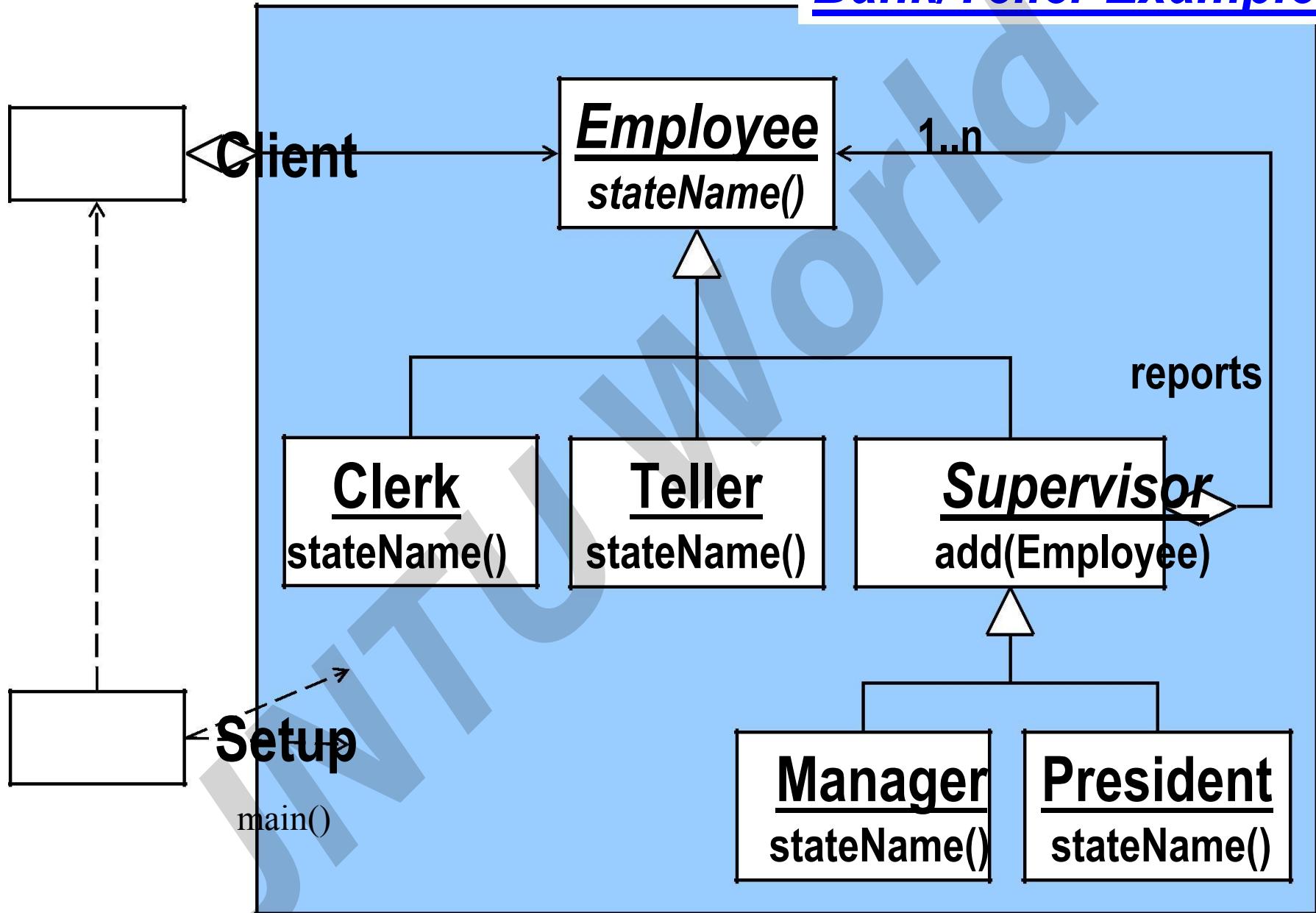
Employee Hierarchy



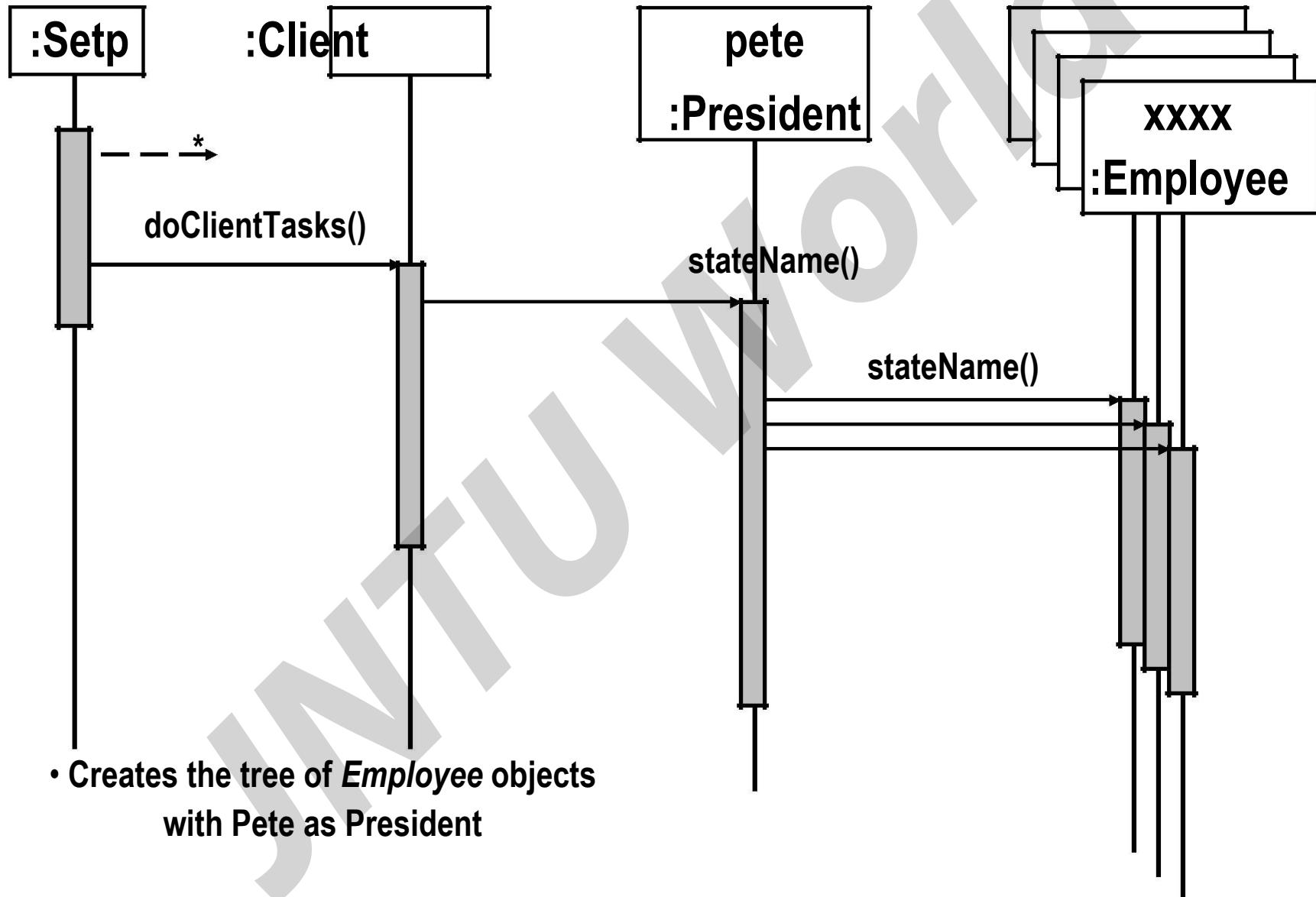
Design Goal At Work: → Flexibility, Correctness ←

We need to add and remove employees at runtime and execute operations on all of them.

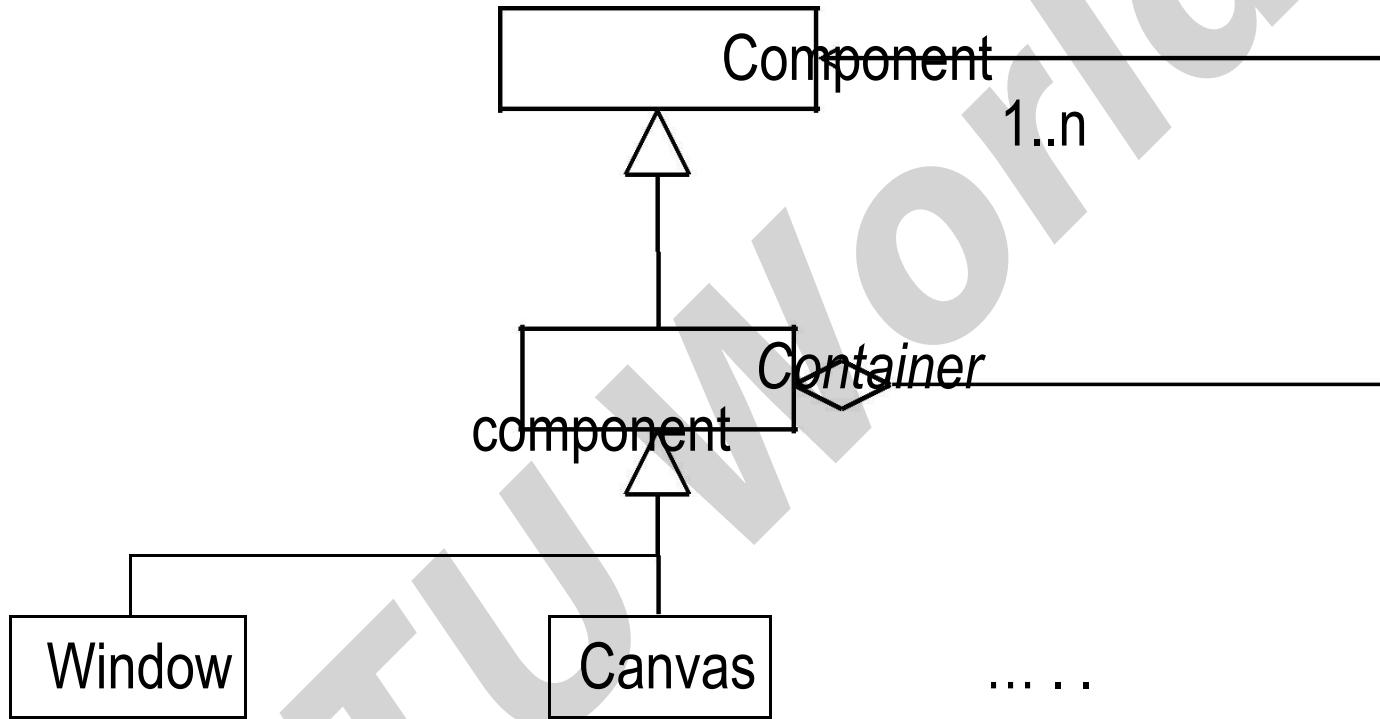
Composite Design Pattern

Bank/Teller Example

Sequence Diagram for Bank/Teller Example



Composite in java.awt



Key Concept: → Composite Design Pattern ←

-- used to represent trees of objects.

Adapter Design Pattern

Adapter

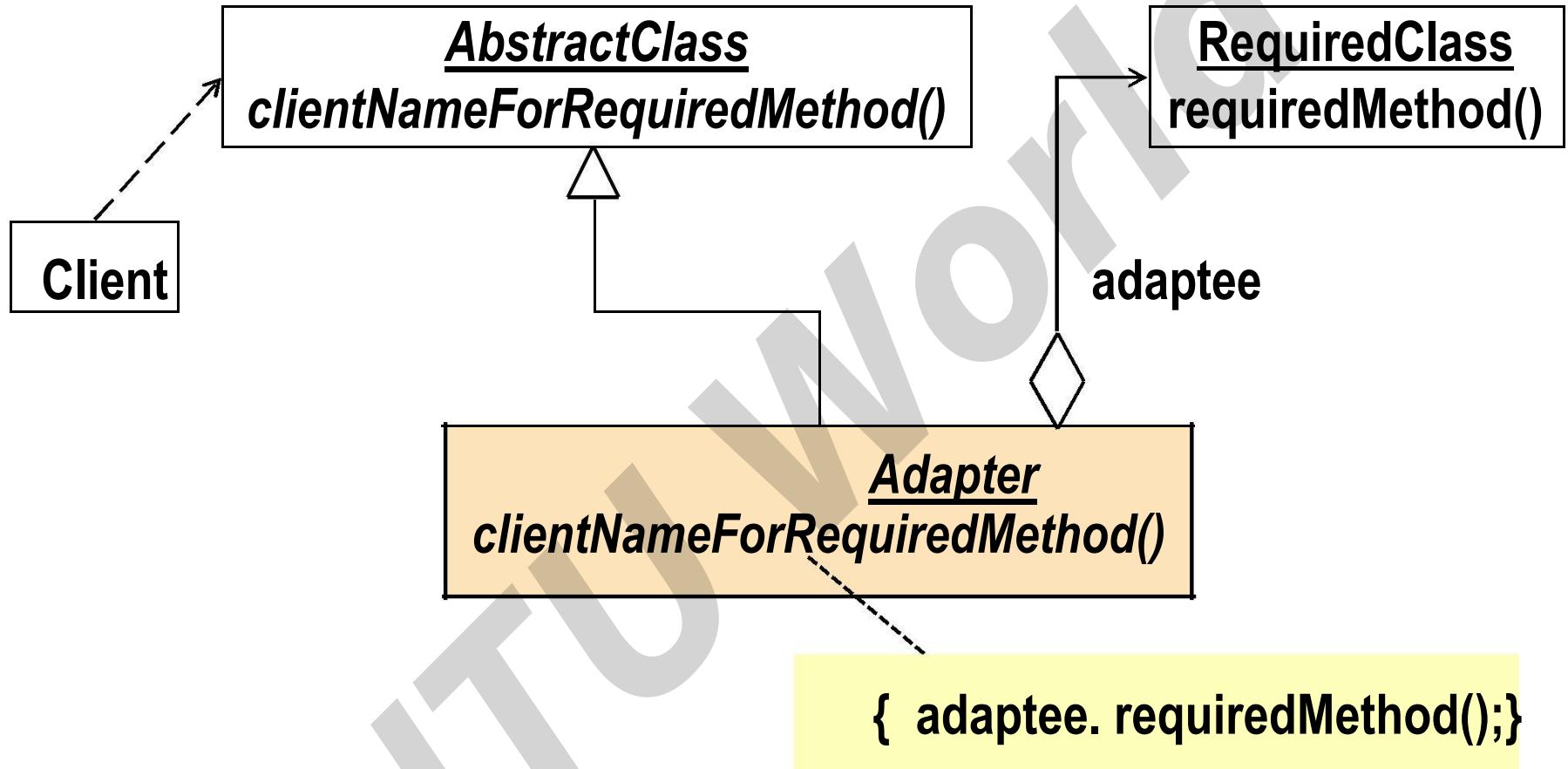
Design Purpose

Allow an application to use external functionality in a retargetable manner.

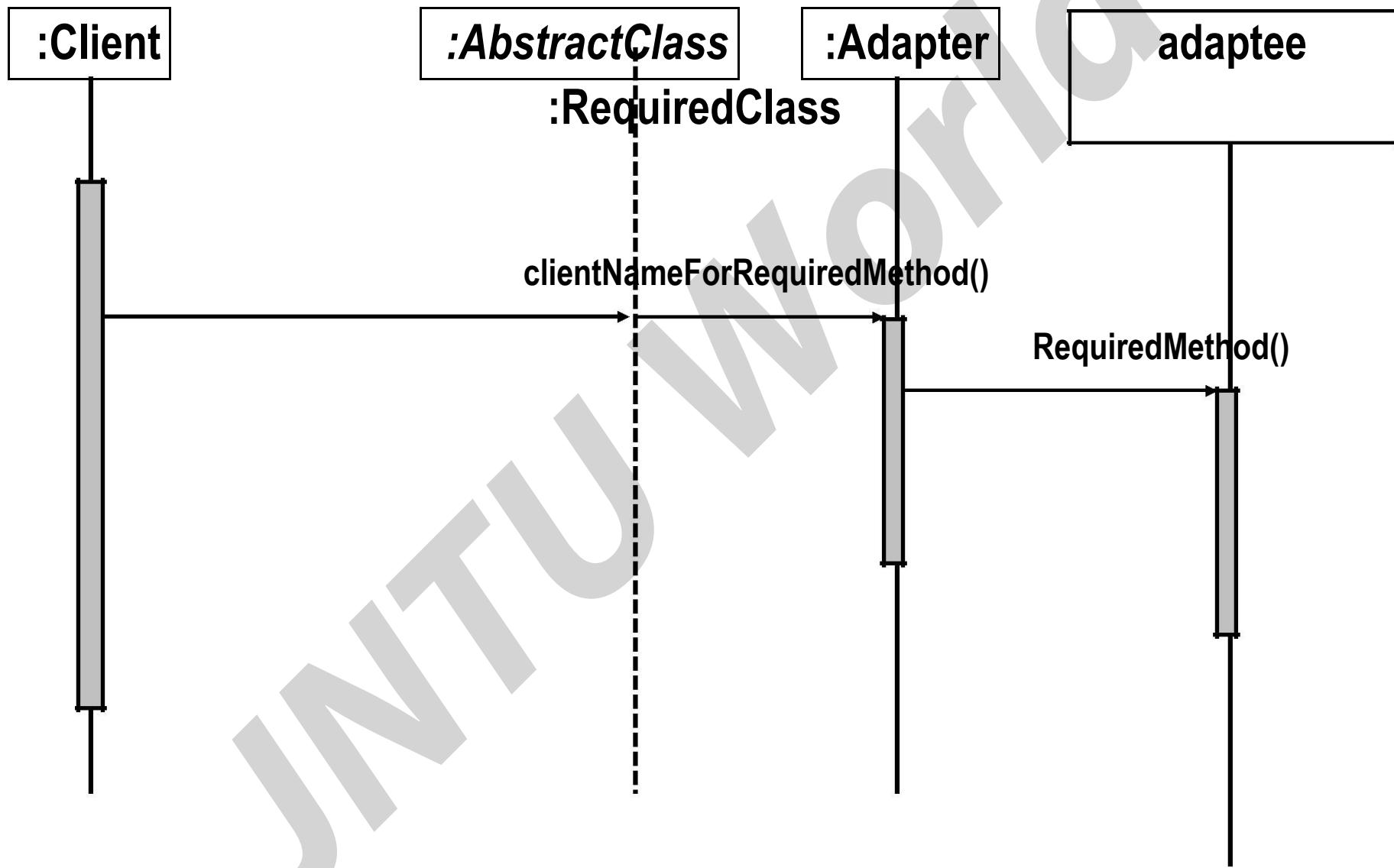
Design Pattern Summary

Write the application against an abstract version of the external class; introduce a subclass that aggregates the external class.

Adapter Example



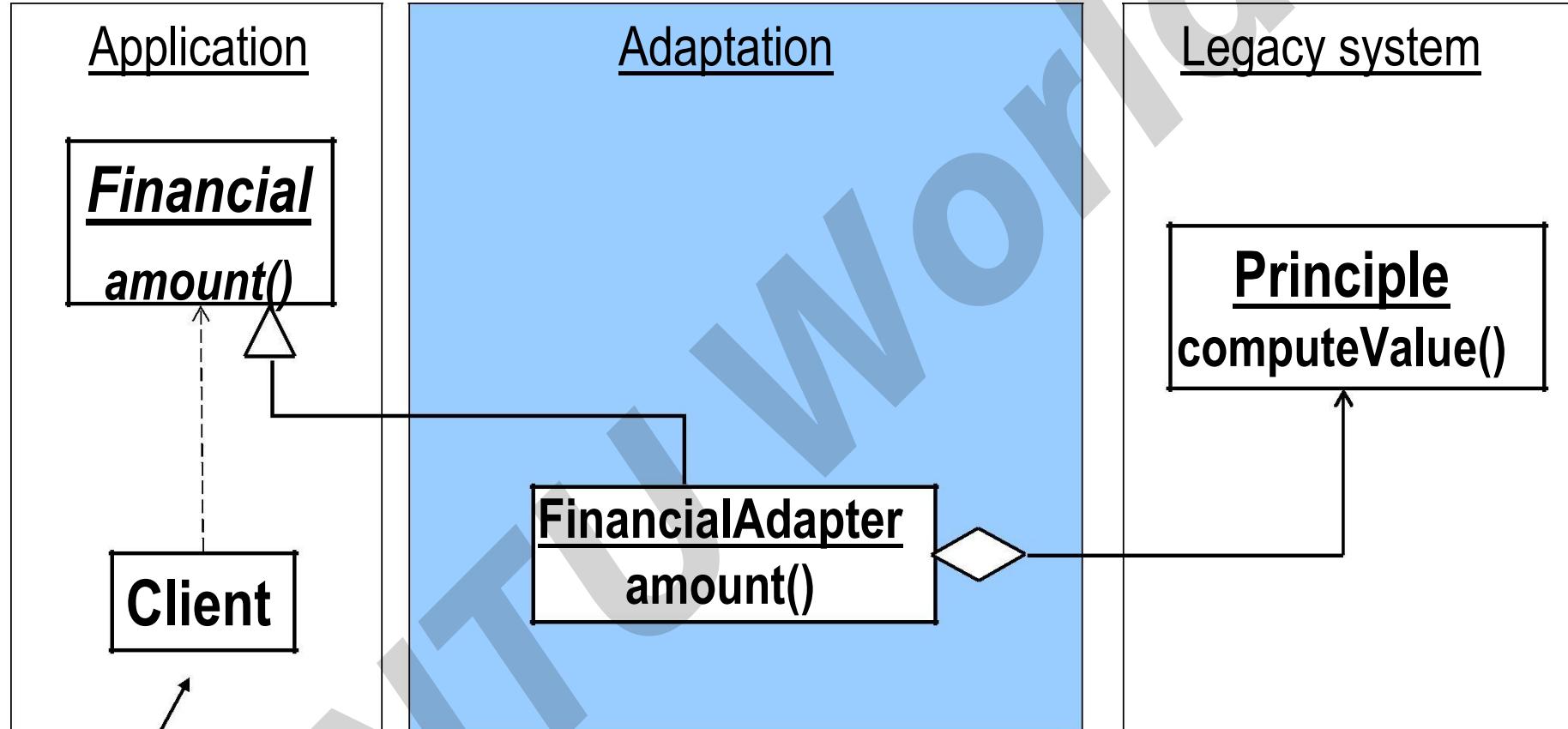
Sequence Diagram for Adapter



Design Goal At Work: → Flexibility and Robustness ←

We want to separate the application as a whole from financial calculations which will be performed externally.

Adapter Design Pattern



Setup code in the client instantiates the Financial object as a FinancialAdapter instance

Code Example

```
class FinancialAdapter extends Financial
{
    Principal legacyAdaptee = null;
    • Constructor goes here . . .
    • This method uses the legacy computeValue() method
float amount(float originalAmount, float numYears, float intRate)
{
    return legacyAdaptee.computeValue(orginalAmount, numYears, intRate);
}
executeFinanceApplication(new FinancialAdapter() );
```

Key Concept: →Adapter Design Pattern ←

-- to interface flexibly with external functionality.

Flyweight Design Pattern

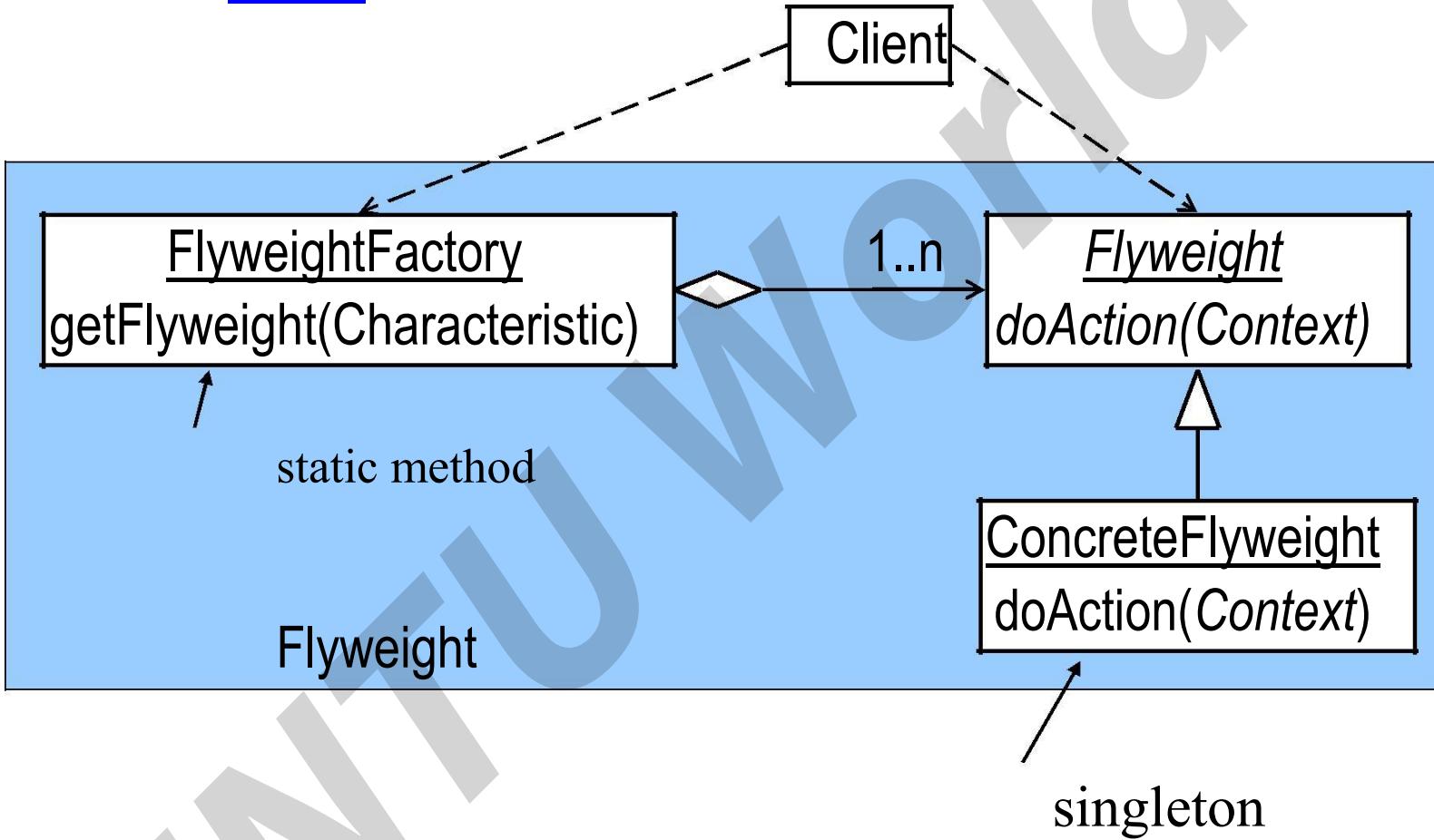
Flyweight Design Purpose

Manage a large number of almost indistinguishable objects without constructing them all at once.

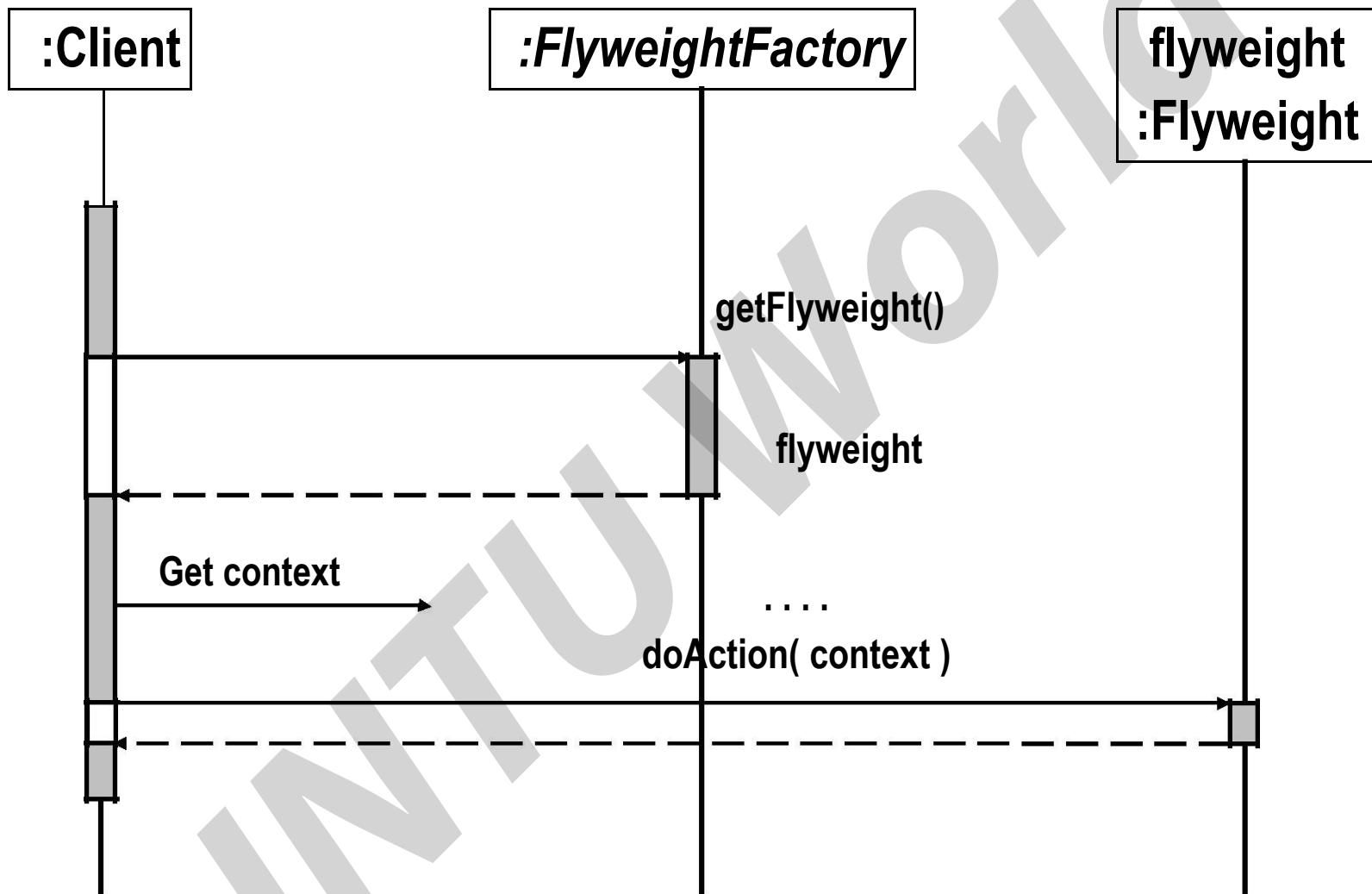
Design Pattern Summary

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

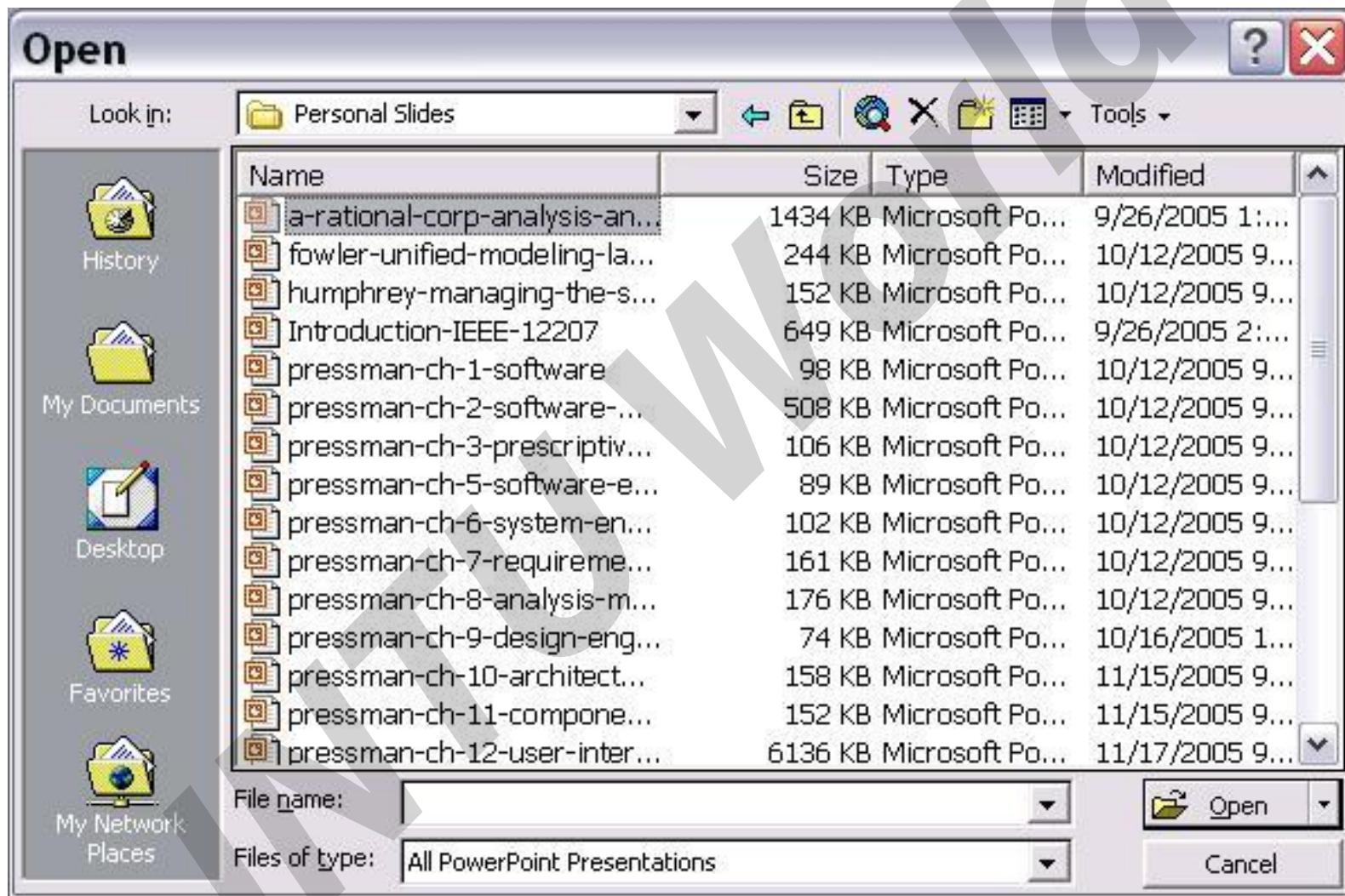
Flyweight Class Model



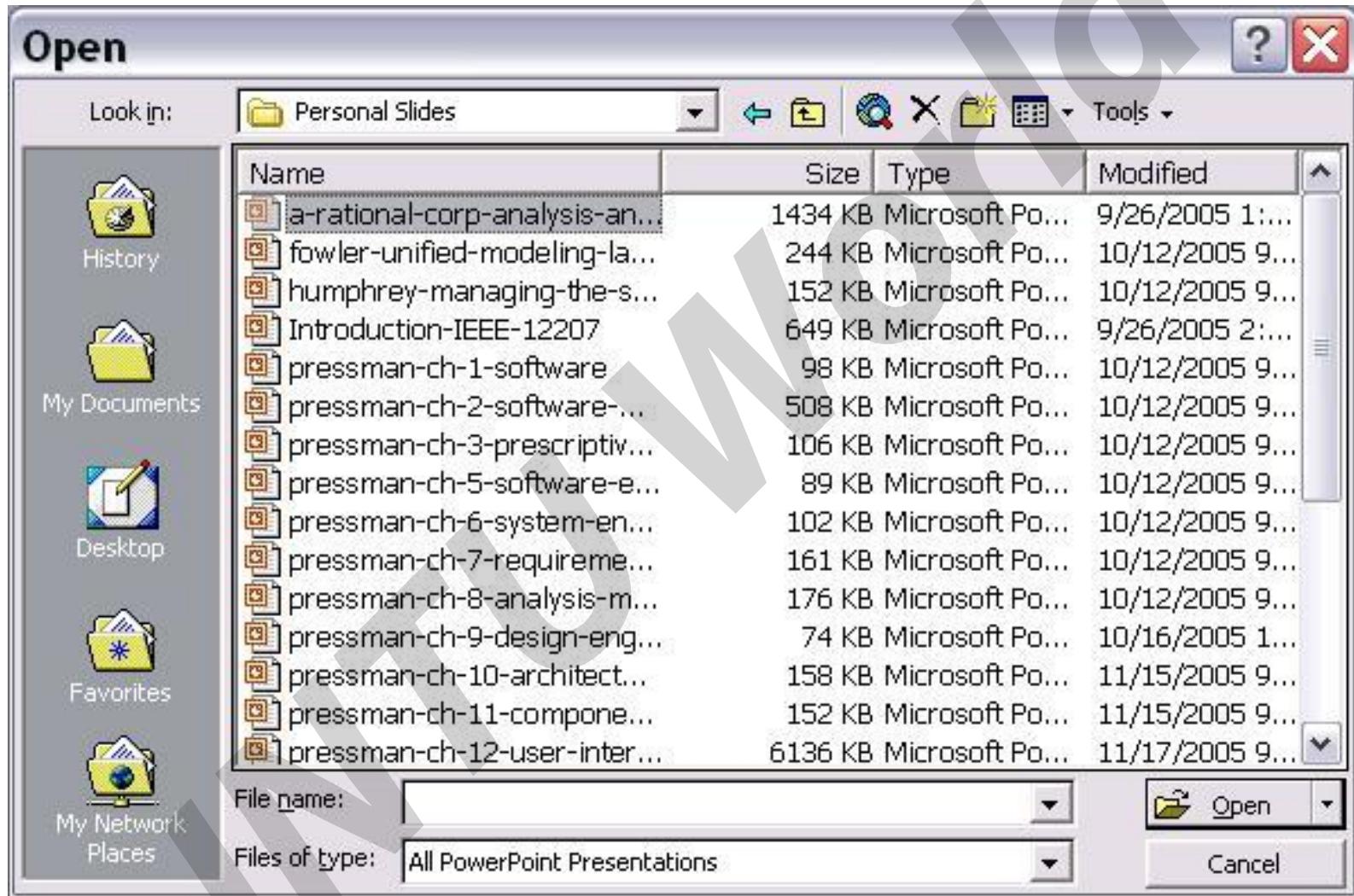
Sequence Diagram for Flyweight



Example A: Window size of 15; folder contains 20 items



Example B: Window size of 15; folder contains over 500 items



Each line item had a drawing window associated with it

Design Goal At Work: → Space Efficiency ←

We want to avoid proliferating an object for every item to be displayed.

Key Concept: →Flyweight Design Pattern ←

-- to obtain the benefits of a large set of individual objects without efficiency penalties.

Proxy Design Pattern

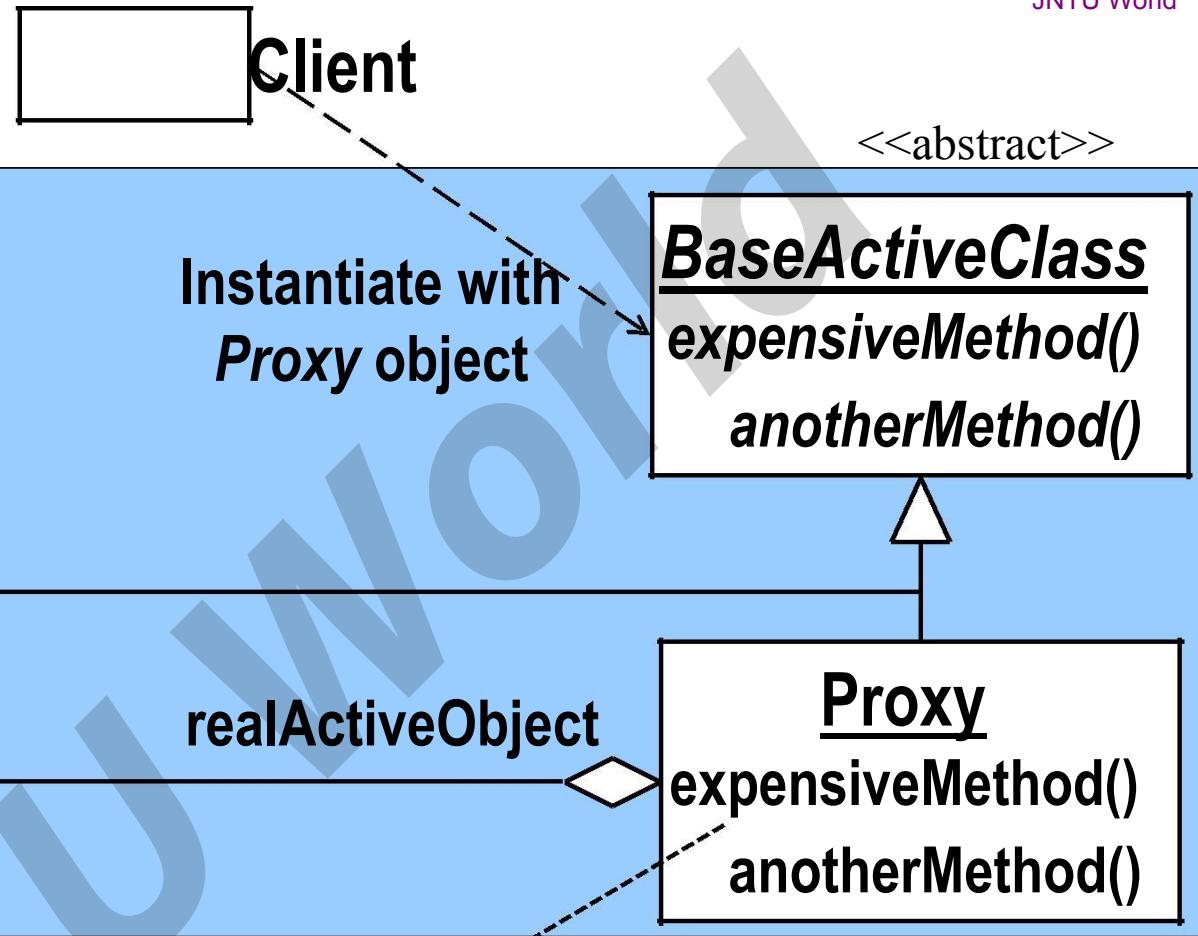
Proxy Design Purpose

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

Design Pattern Summary

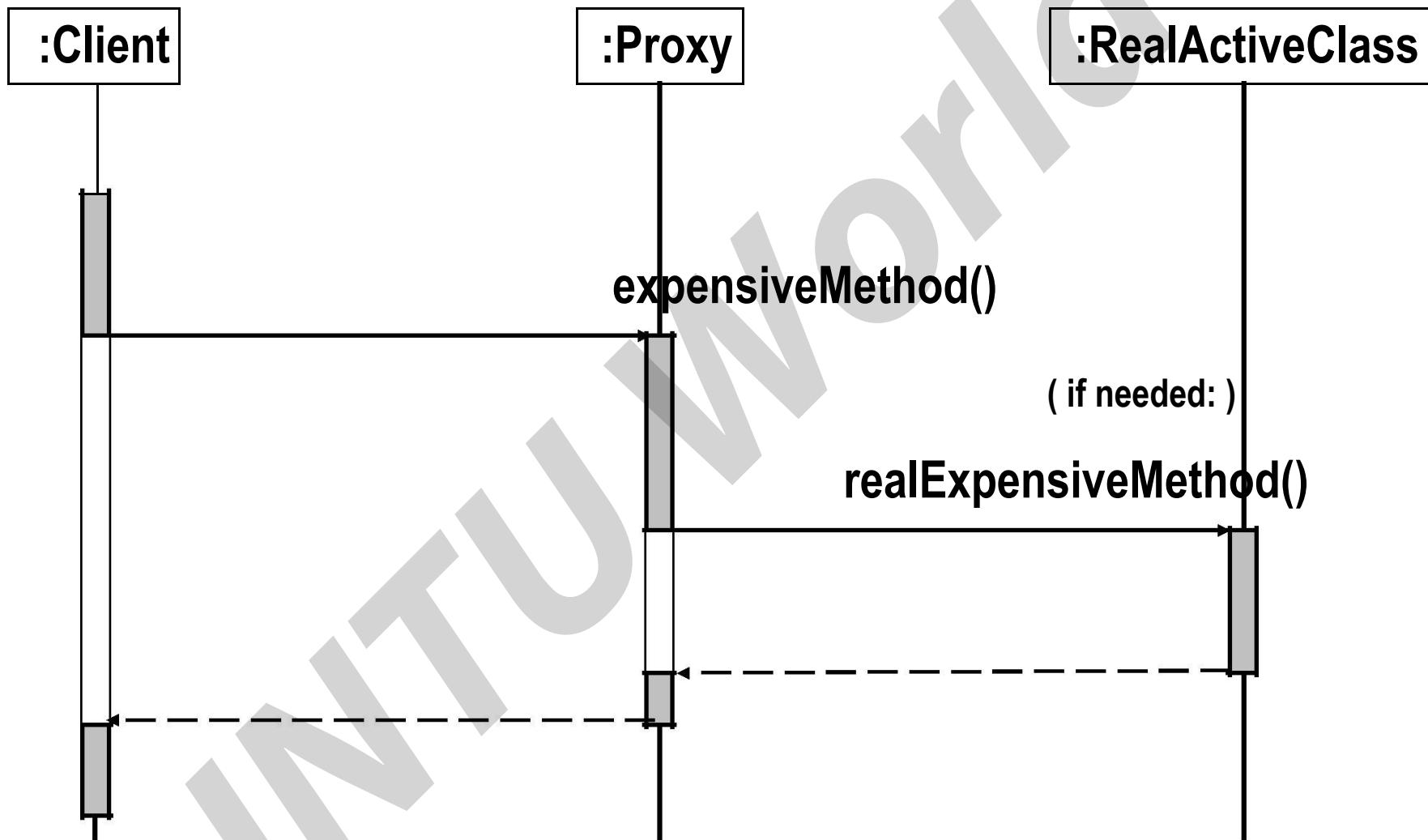
Interpose a substitute class which accesses the expensive functionality only when required.

Proxy Design Pattern



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for Proxy





Please pick a command from one of the following:

quit
middle
all

> all

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

Please pick a command from one of the following:

quit
middle
all

> middle

== 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

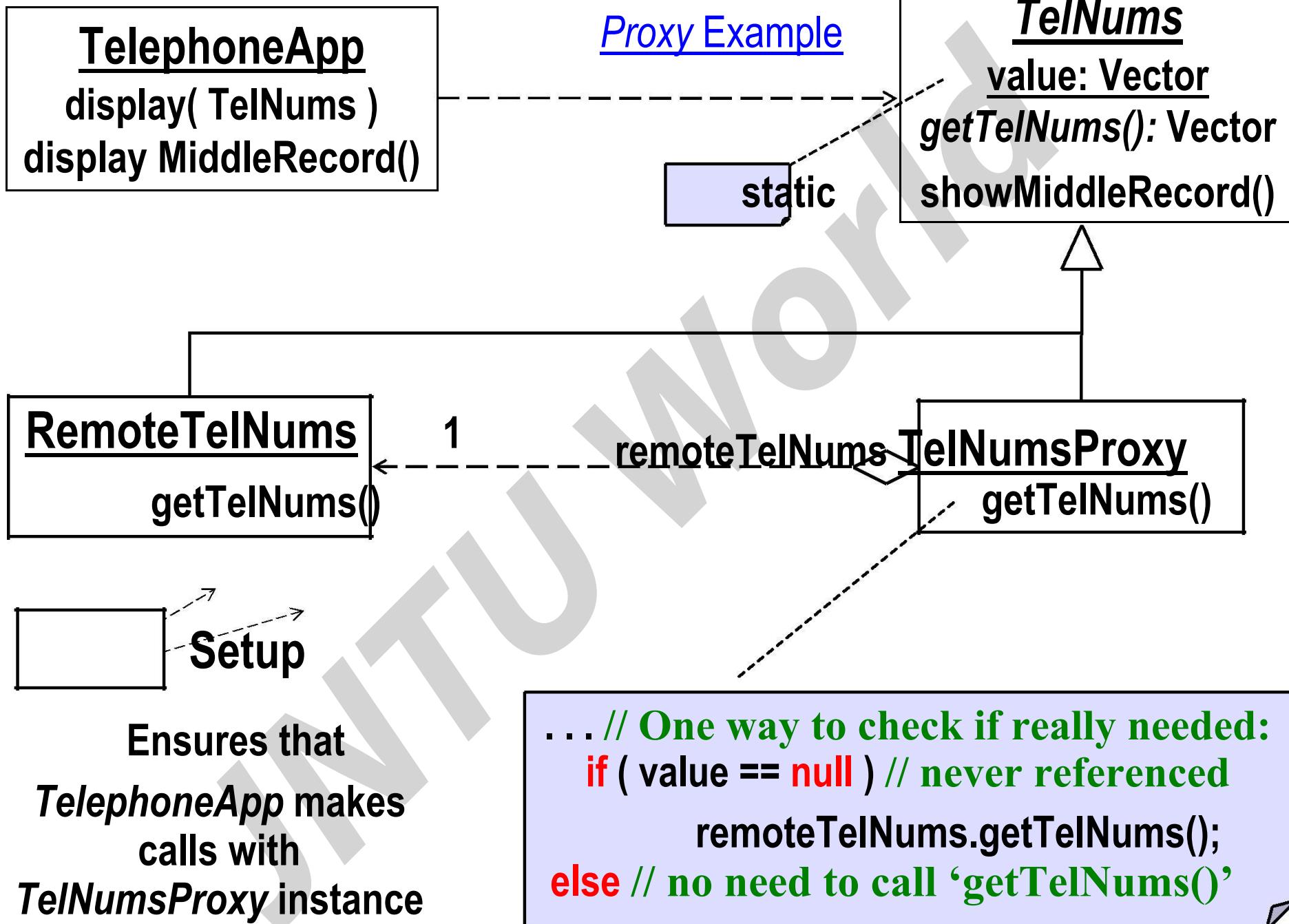
> all

===== 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

I/O of Telephone Record Proxy Example

Design Goal At Work: → Efficiency and Reuse ←

Avoid unnecessary data downloads.



Ensures that
TelephoneApp makes
calls with
TelNumsProxy instance

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

Key Concept: →Proxy Design Pattern ←

-- to call expensive or remote methods.

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 (in a list structure)
 - **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-VI & VII BEHAVIORAL
PATTERNS-I & II

Behavioral Patterns

Chain of Responsibility

Command

~~Interpreter~~

Iterator

\} Mediator

\} Memento

\} Observer

\} State

Strategy

• ~~Template Method~~

Visitor

(requests through a chain of candidates) (encapsulates a request)

(grammar as a class hierarchy) (abstracts traversal and access)
(indirection for

loose coupling)

(externalize and re-instantiate object state)

(defines and maintains dependencies)

(change behaviour according to changed state)

(encapsulates an algorithm in an object)

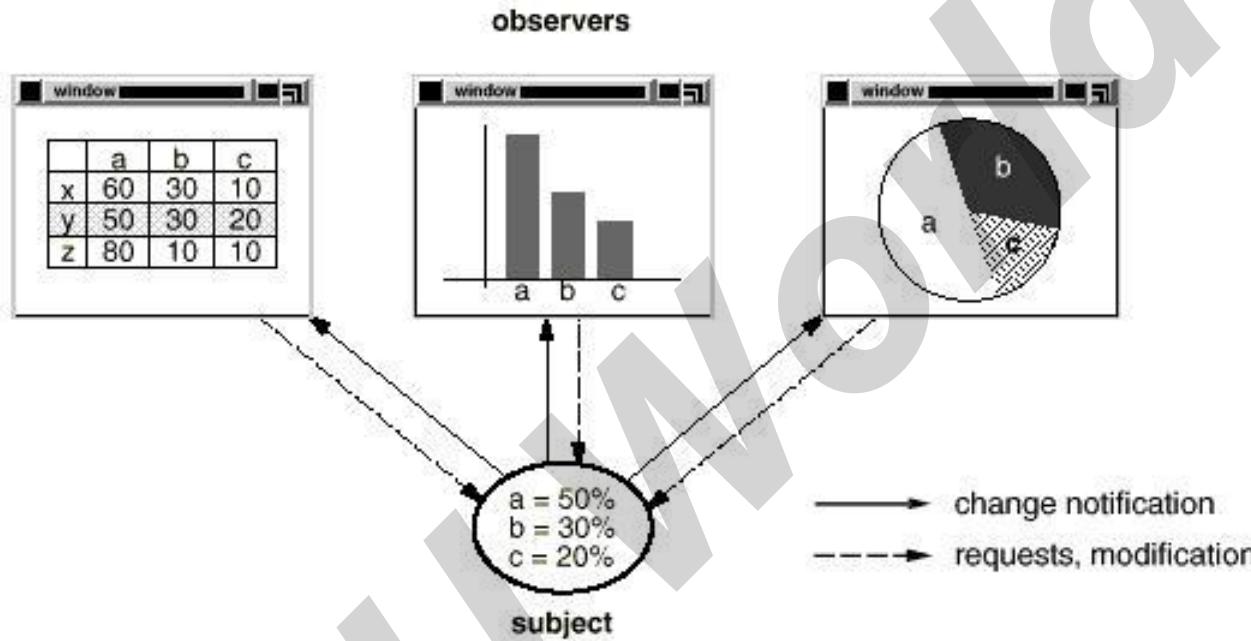
(step-by-step algorithm w/ inheritance)

(encapsulated distributed behaviour)

Observer

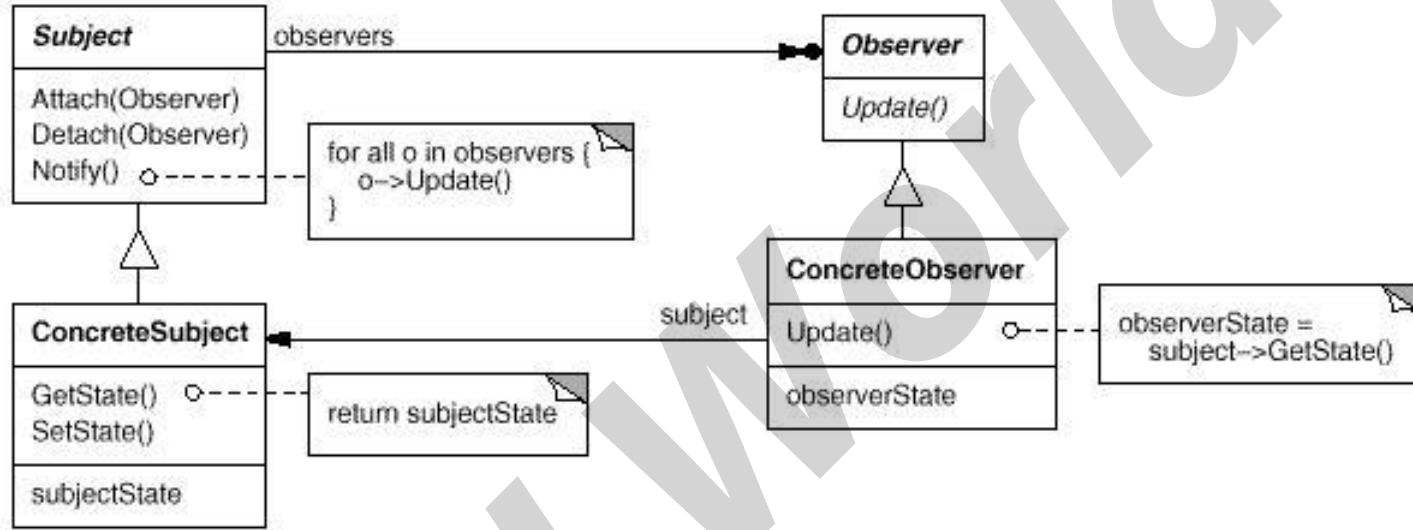
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
 - A common side-effect of partitioning a system into a collection of cooperating classes is
 - the need to maintain consistency between related objects
 - You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
 - a.k.a. Publish-Subscribe
 - Common related/special case use: MVC
 - Model-View-Controller pattern

Motivation



- Separate presentation aspects of the UI from the underlying application data.
 - e.g., spreadsheet view and bar chart view don't know about each other
 - they *act* as if they do: changing one changes the other.

Structure



Subject

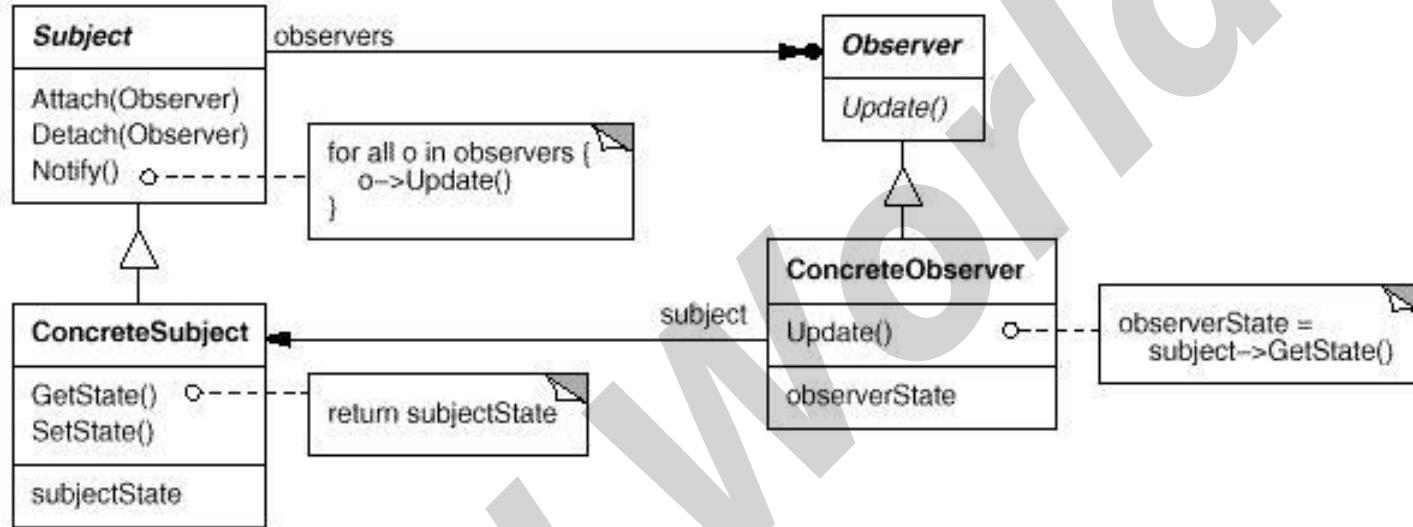
- knows its observers

- any number of Observers may observe one subject

Observer

- defines an updating interface for objects that should be notified of changes to the subject

Structure



- **Concrete Subject**

- stores the state of interest to **ConcreteObservers**

- send notification when its state changes

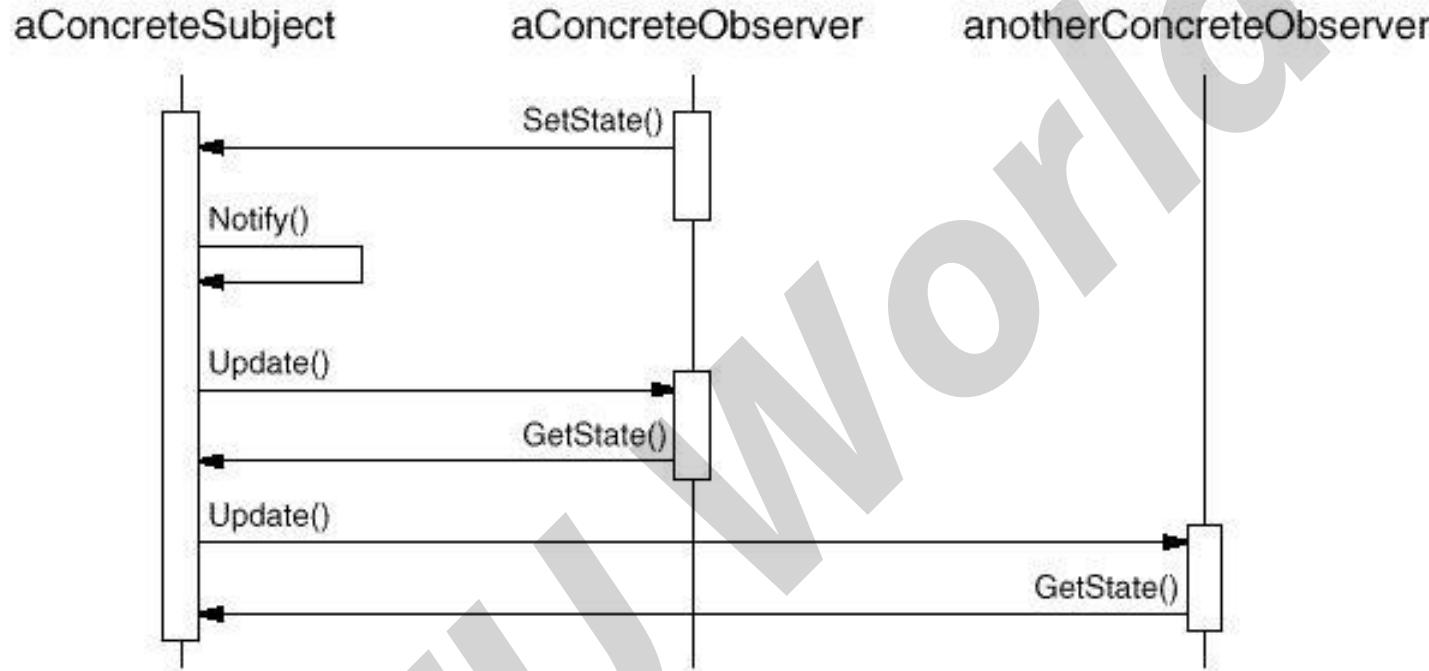
- **Concrete Observer**

- maintains a reference to the **ConcreteSubject** objects

- stores state that should remain consistent with subject's

- implements the **Observer** updating interface

Collaborations



- subject notifies its observers whenever a change occurs that would make its observers' state inconsistent with its own
- After being informed, observer may query subject for changed info.
 - uses query to adjust its state

Applicability

- When an abstraction has two aspects, one dependent upon the other
 - e.g., view and model

Encapsulating these aspects into separate objects lets you vary them independently.

- when a change to one object requires changing others, and you don't know ahead of time how many there are or their types
 - when an object should be able to notify others without making assumptions about who these objects are,
 - you don't want these objects tightly coupled

Consequences

- abstract coupling
 - no knowledge of the other class needed
- supports broadcast communications
 - subject doesn't care how many observers there are
 - spurious updates a problem
 - can be costly
 - unexpected interactions can be hard to track down
 - problem aggravated when simple protocol that does not say what was changed is used

Implementation

- Mapping subjects to observers
 - table-based or subject-oriented
- Observing more than one subject
 - interface must tell you which subject
 - data structure implications (e.g., linked list)
 - Who triggers the notify()
 - subject state changing methods
 - > 1 update for a complex change
 - clients
 - complicates API & error-prone
 - can group operations and send only one update
 - transaction-oriented API to client

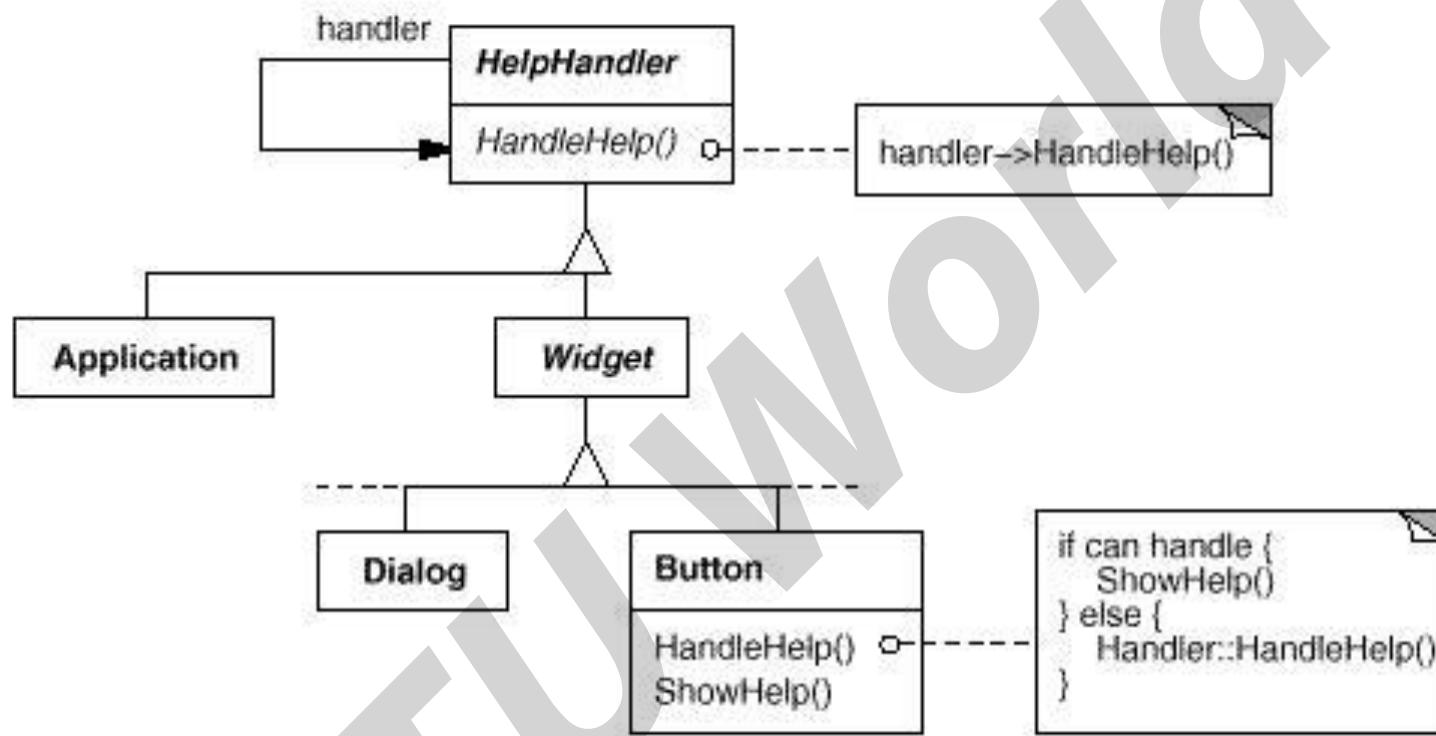
Implementation

- dangling references to deleted subjects
 - send 'delete message'
 - complex code
- must ensure subject state is self-consistent before sending update
 - push versus pull
 - push: subject sends info it thinks observer wants
 - pull: observer requests info when it needs it
 - registration: register for what you want
 - when observer signs up, states what interested in
 - ChangeManager
 - if observing more than one subject to avoid spurious updates
 - Can combine subject and observer

Chain Of Responsibility

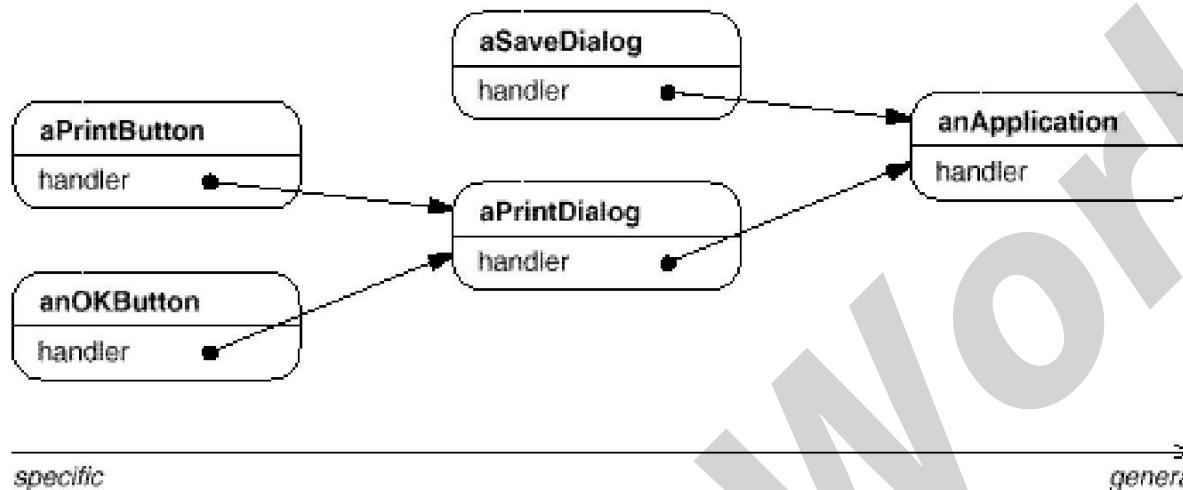
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
 - Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation



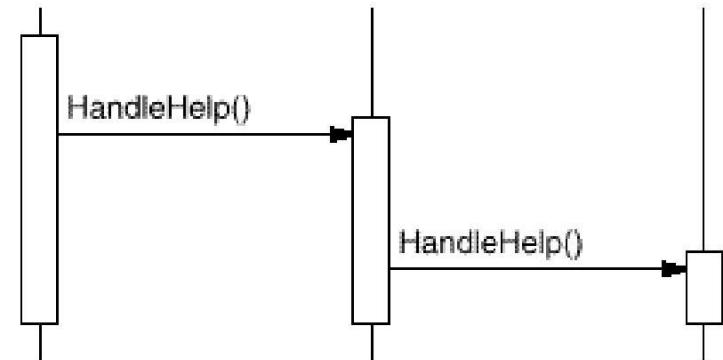
- Context-sensitive help
 - User can obtain information on any part of a UI by clicking on it.
 - If no help available (e.g., for a button), system should display a more general help message about the context (e.g., the dialog box containing the button).

Motivation



- Objects forward the request until there is one that can handle it.
 - The key is that the client does not know the object that will eventually handle the request.

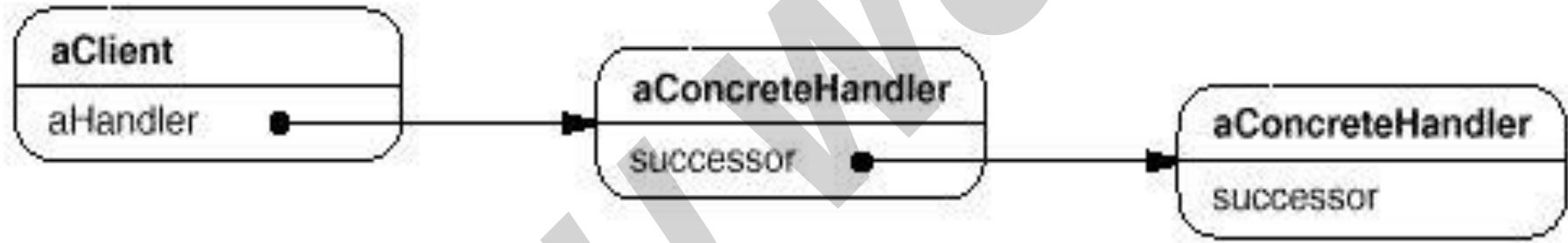
aPrintButton aPrintDialog anApplication



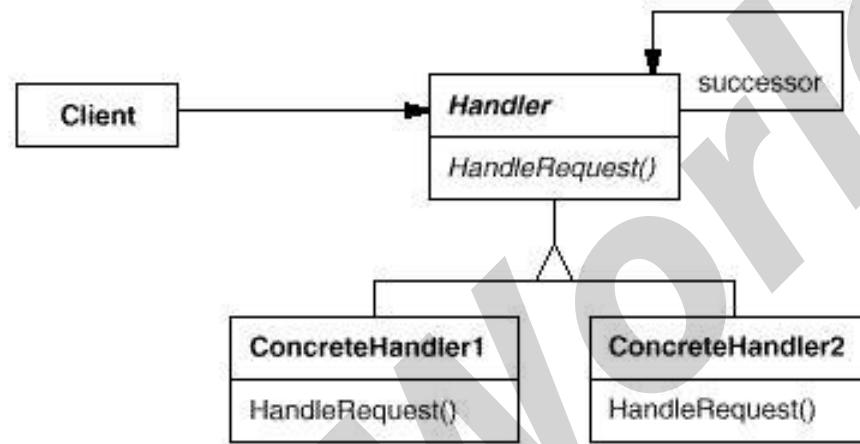
Applicability

- More than one object may handle a request, and the handler isn't known *a priori*.
 - The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Structure



Structure



- **Handler**
 - defines an interface for handling requests
 - implements the successor list (optional)
 - **ConcreteHandler**
 - handles requests for which it is responsible
 - can access its successor
 - forward to successor if it can't handle the request
- **Client**
 - initiates the request to the first **ConcreteHandler** in the chain.

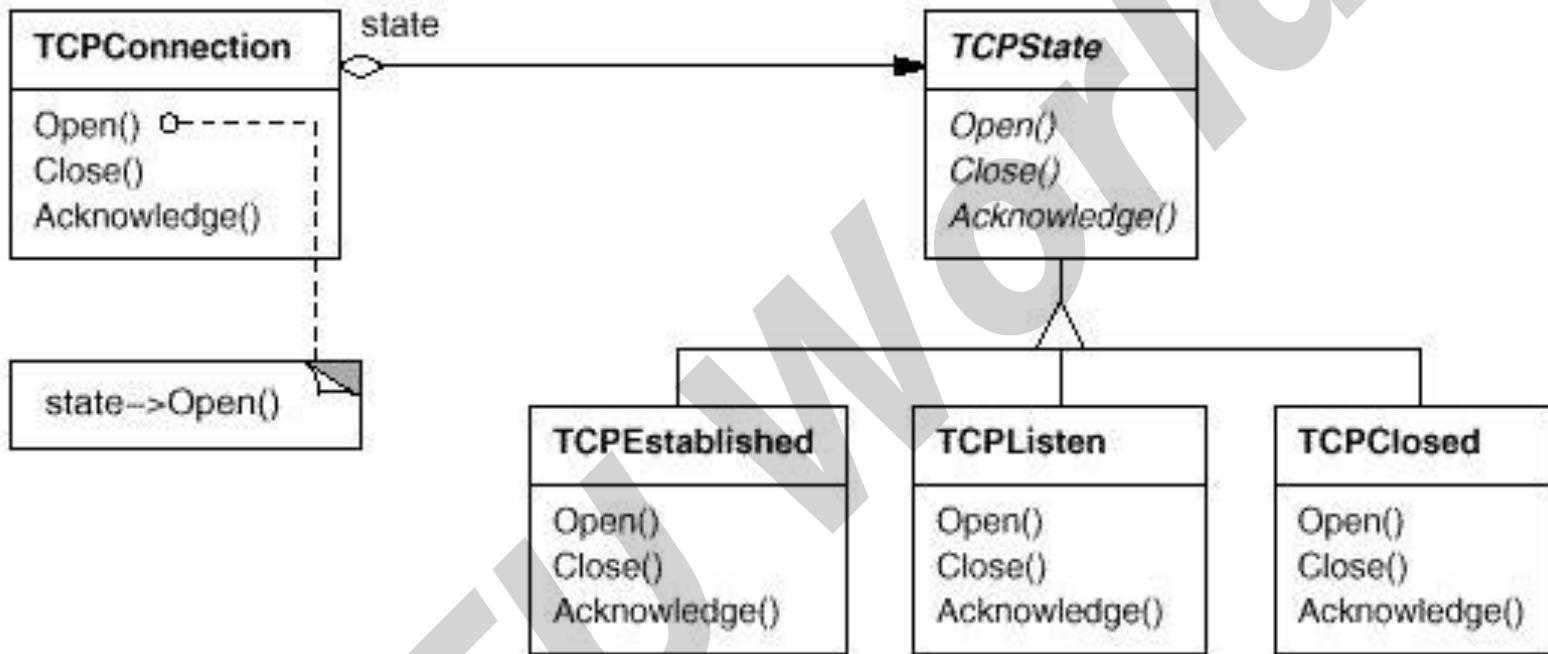
Consequences

- reduced coupling
- receiver and sender have no explicit knowledge of each other
 - can simplify object interactions
 - added flexibility
- can add or change responsibilities by changing the chain at run-time.
 - receipt is not guaranteed.
 - request may fall off the end of the chain

State

- Allow an object to alter its behavior when its internal state changes.
 - The object will appear to change its class.

Motivation

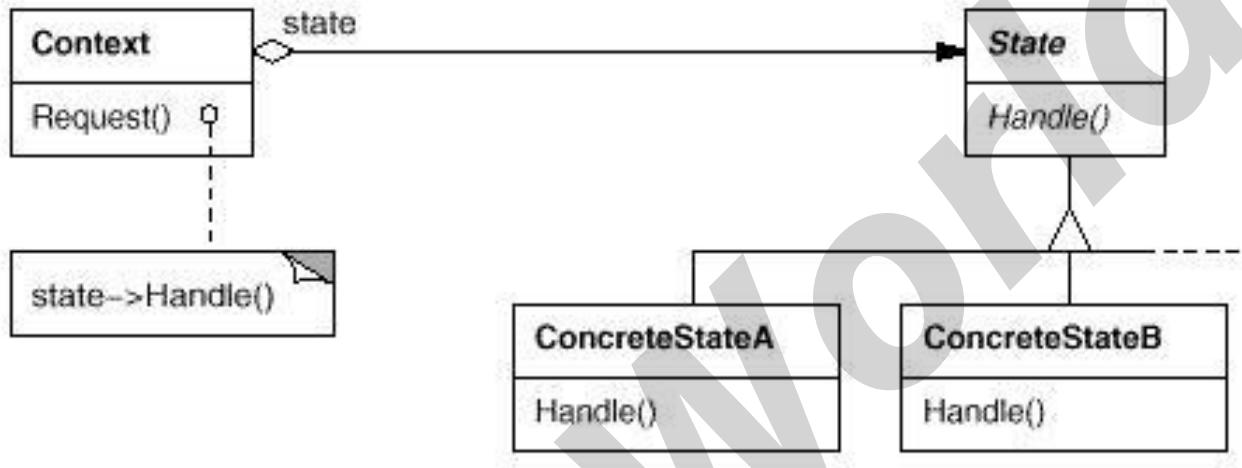


- A **TCPConnection** object that responds differently to requests given its current state.
 - All state-dependent actions are delegated.

Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
 - Operations have large, multipart conditional statements that depend on the object's state.
 - This state is usually represented by one or more enumerated constants.
 - Often, several operations will contain this same conditional structure.
 - The State pattern puts each branch of the conditional in a separate class.
 - This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure



- **Context**

- defines the interface of interest to clients.
- maintains an instance of a **ConcreteState** subclass that defines the current state.

- **State**

- defines an interface for encapsulating the behavior associated with a particular state of the **Context**.

- **ConcreteState subclasses**

- each subclass implements a behavior associated with a state of the **Context**.

Consequences

- It localizes state-specific behavior and partitions behavior for different states.
 - The State pattern puts all behavior associated with a particular state into one object.
 - Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.
 - It makes state transitions more explicit
 - State is represented by the object pointed to.
- It protects the object from state-related inconsistencies.
 - All implications of state changed wrapped in the atomic change of 1 pointer.
 - State object can be shared
 - if no data members they can be re-used across all instances of the Context

Mediator

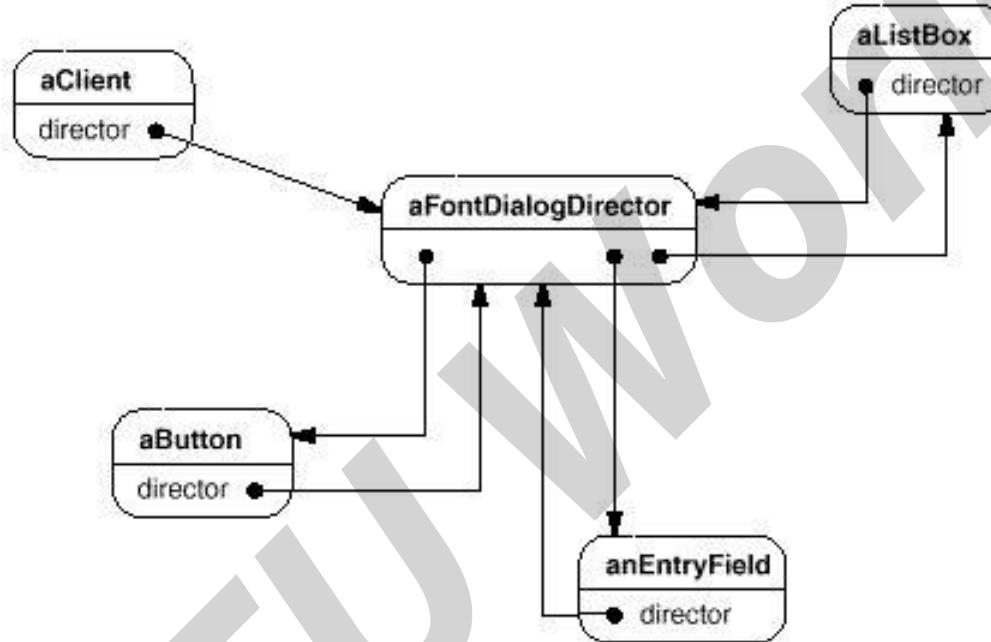
- Defines an object that encapsulates how a set of objects interact.
 - promotes loose coupling by keeping objects from referring to each other explicitly
 - lets you vary their interaction independently

Motivation



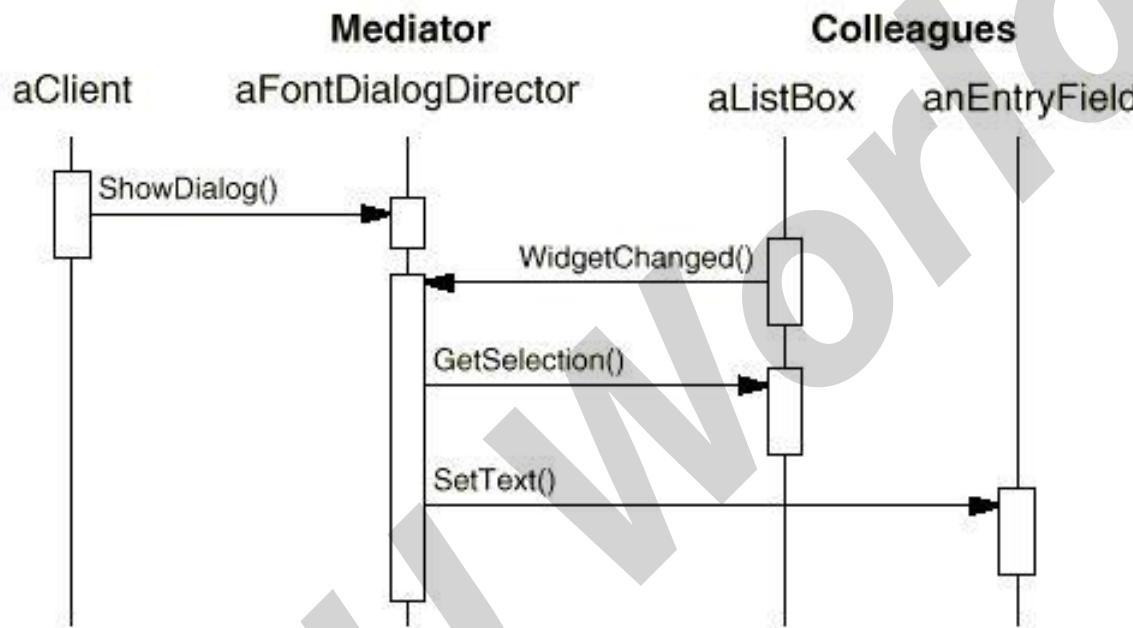
- A collection of widgets that interact with one another.
 - e.g., certain families may not have certain weights
 - disable 'demibold' choice

Motivation



- Create a mediator to control and coordinate the interactions of a group of objects.

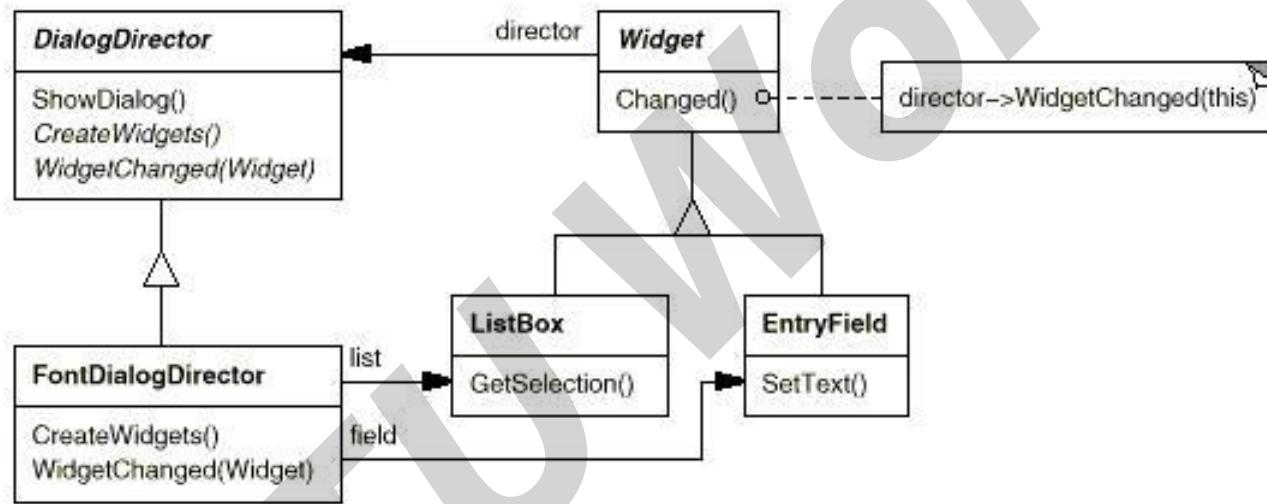
Motivation



• e.g.,

- list box selection moving to entry field
- entryField now calls WidgetChanged() and enables/disables
- entry field does not need to know about list box and *vice-versa*

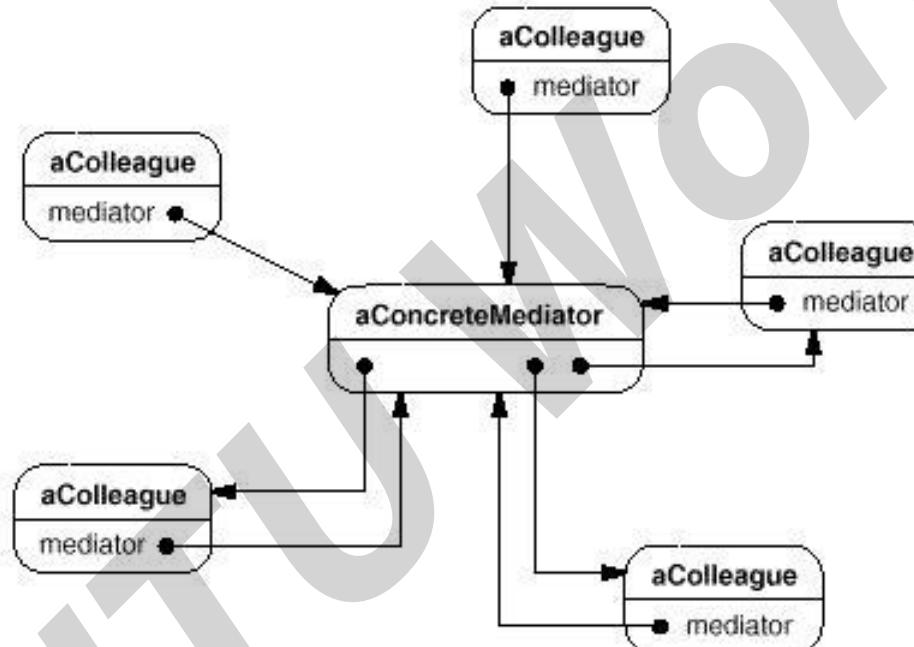
Motivation



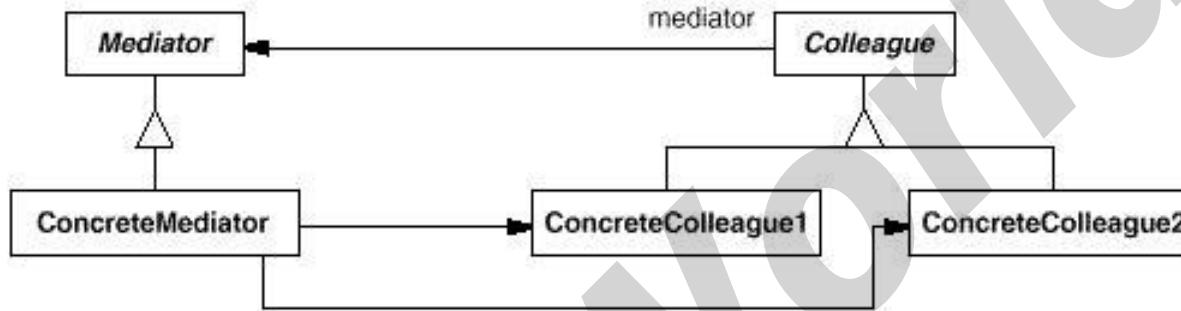
Applicability

- A set of objects communicate in a well-defined but complex manner
- reusing an object is difficult because it refers to and communicates with many other objects
- a behavior that's distributed between several classes should be customizable without a lot of subclassing

Structure



Structure



- Mediator
 - defines an interface for communicating with Colleague objects
 - **ConcreteMediator**
 - knows and maintains its colleagues
 - implements cooperative behavior by coordinating Colleagues
 - **Colleague** classes
 - each Colleague class knows its Mediator object
 - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Conseq uences

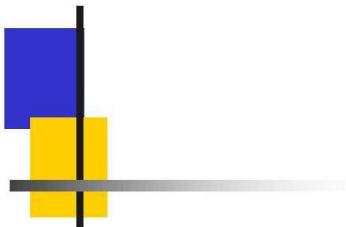
- limits subclassing
 - localizes behaviour that otherwise would need to be modified by subclassing the colleagues
- decouples colleagues
 - can vary and reuse colleague and mediator classes independently
- simplifies object protocols
 - replaces many-to-many interactions with one-to-many

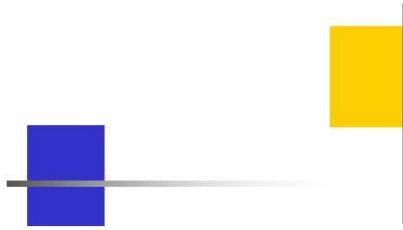
- one-to-many are easier to deal with
- abstracts how objects cooperate
- can focus on object interaction apart from an object's individual behaviour
- centralizes control
 - mediator can become a monster

UNIT-VIII

WHAT TO EXPECT FROM DESIGN PATTERNS

Adapter Pattern





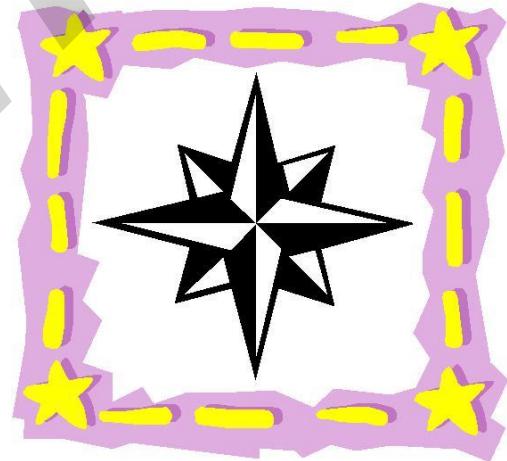
What is Adapter?

Intent

Change the interface of a class into another interface which is expected by the client.

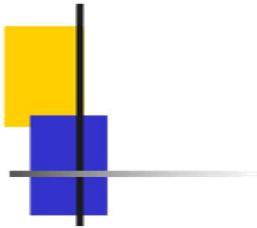
Also Known as

Wrapper



- Total 17 pages -

1



Example :- Drawing editor -lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.

A TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management.

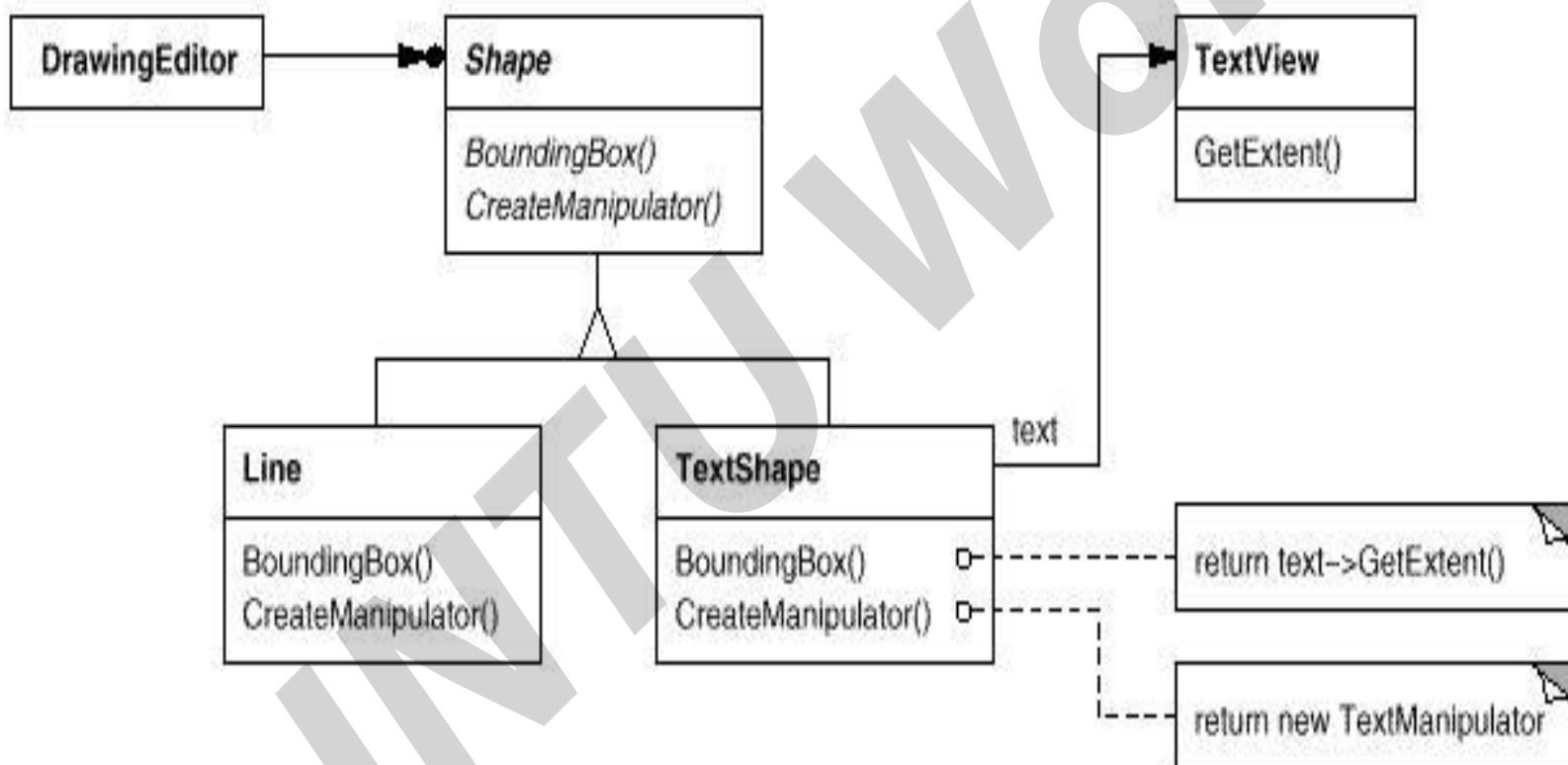
Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text. Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

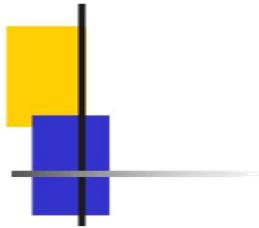
- Total 17 pages -

1



Motivation example





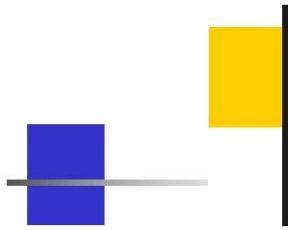
Solution

TextShape should adapt the **TextView** interface to **Shape**'s.

This can be achieved in two ways

- 1) By inheriting **Shape**'s interface and **TextView**'s implementation
- 2) By composing a **TextView** instance within a **TextShape** and implementing **TextShape** in terms of **TextView**'s interface

It shows how BoundingBox requests, declared in class **Shape**, are converted to GetExtent requests defined in **TextView**.

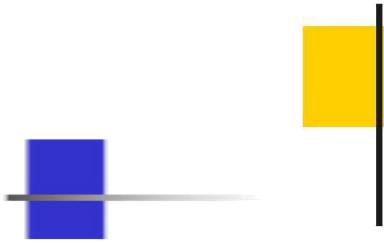


Applicability

2. Use an existing class whose interface does not match the requirement
3. Create a reusable class though the interfaces are not necessarily compatible with callers

- Total 17 pages -

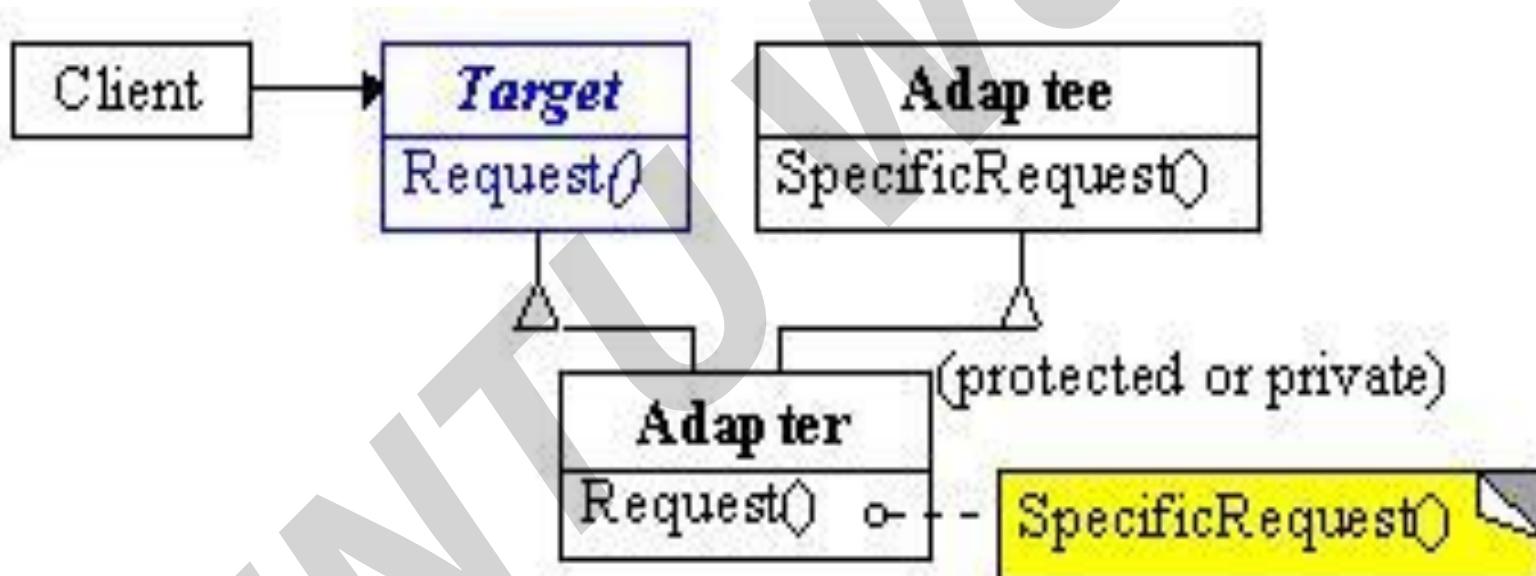
1

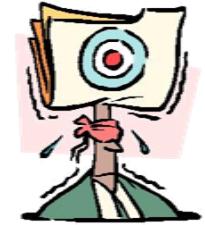


Structure (Class)

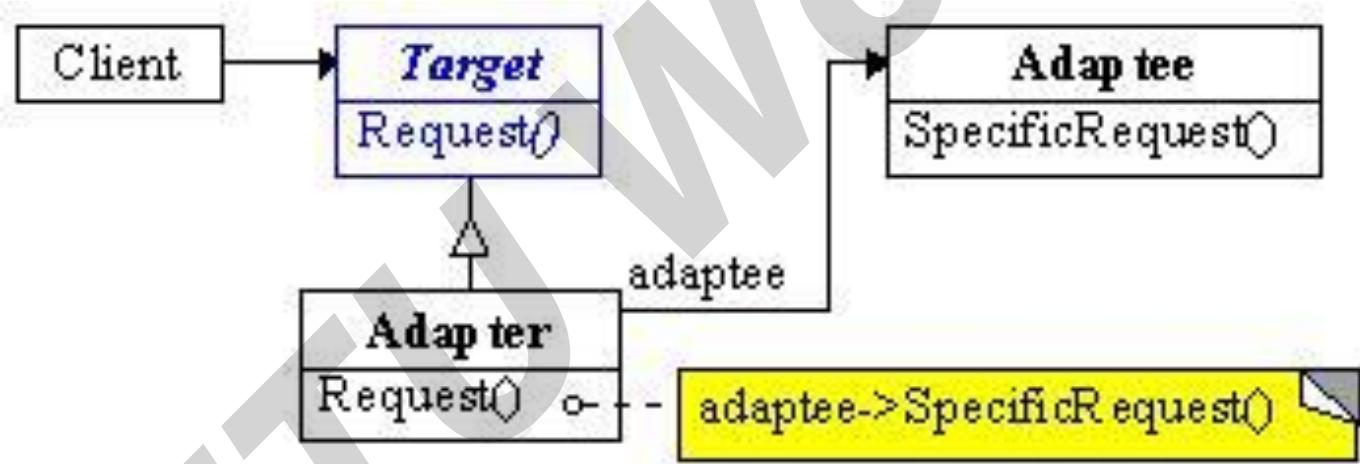


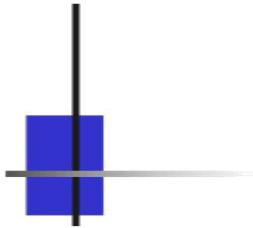
A class adapter uses multiple inheritance to adapt one interface to another:





Structure (Object)





Participants

// **Target**(Shape)

Defines the domain-specific interface that client uses

// **Client** (DrawingEditor)

Collaborates with objects conforming to the Target interface

// **Adaptee** (TextView)

Defines an existing interface that needs adapting

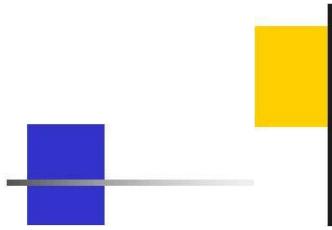
// **Adapter** (TextShape)

interface

Adapts the interface of Adaptee to the Target

- Total 17 pages -

1

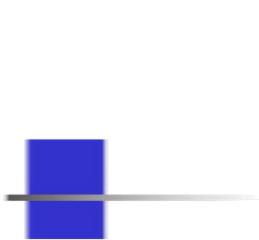


Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

- Total 17 pages -

1



Class Adapter Pattern

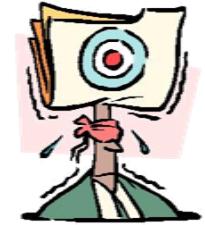
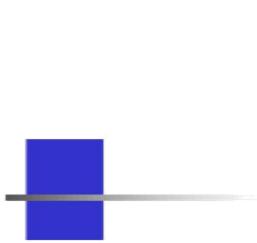
Consequences



- Adapts Adaptee to Target by committing to a concrete Adapter class
 - Lets adapter override some of Adaptee's behaviour
- Introduces only one object, and no additional pointer indirection is needed to get to the adaptee

- Total 17 pages -

1



Object Adapter Pattern

Consequences

- Lets a single Adapter work with many Adaptees (i.e., adaptee and all its subclass)
 - Makes it harder to override adaptee behaviour

- Total 17 pages -

1



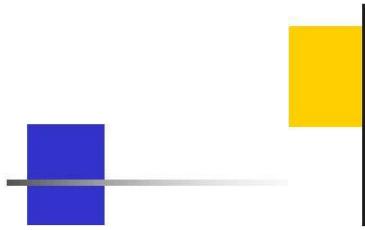
Consequences cont..

other issues to consider when using the Adapter pattern:

- How much adapting does Adapter do?
 - Pluggable adapters
- Using two way adapters to provide transparency

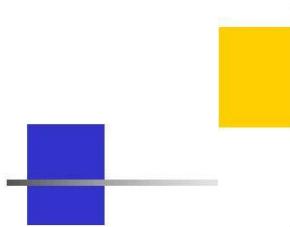
- Total 17 pages -

1



Implementation

- Implementing class adapters in C++
 - Pluggable adapters
 - \endash Using abstract operations
 - \endash Using delegate objects
 - \endash Parameterized adapters

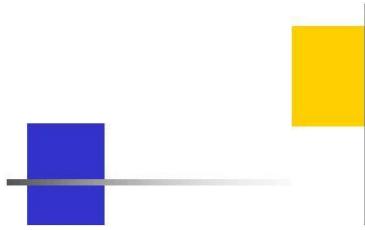


Known uses

- ET++Draw
- ObjectWorks / smalltalk

- Total 17 pages -

1

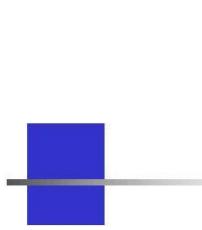


Related Patterns

- Decorator enhances another object without changing its interface.
- Bridge similar structure to Adapter, but different intent. Separates interface from implementation.

- Total 17 pages -

1

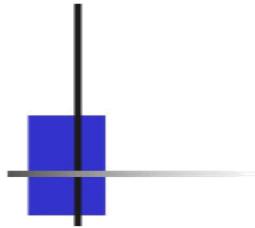


Conclusions

- 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

- Total 17 pages -

1



Motivation



- How can unrelated classes like TextView work in an application that expects classes with a different and incompatible interface

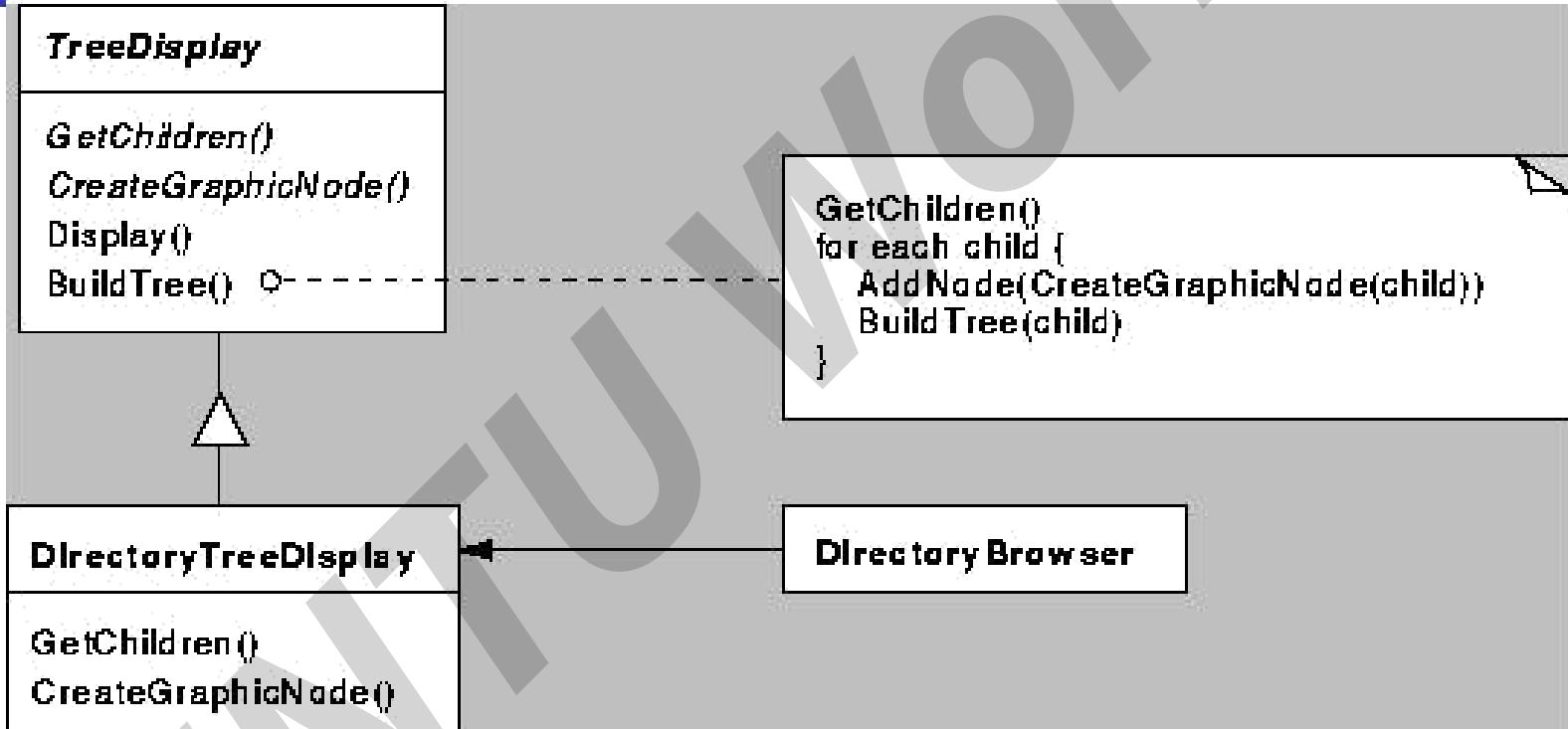


- Total 17 pages -

1



Pluggable Adapters



□ implemented with abstract operations



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();  
        }  
}
```

15.Additional Topics

The Bridge Pattern

Bridge Pattern

Intent

Decouple an abstraction from its implementation so that the two can vary independently

Also Known As

Handle / Body

Motivation

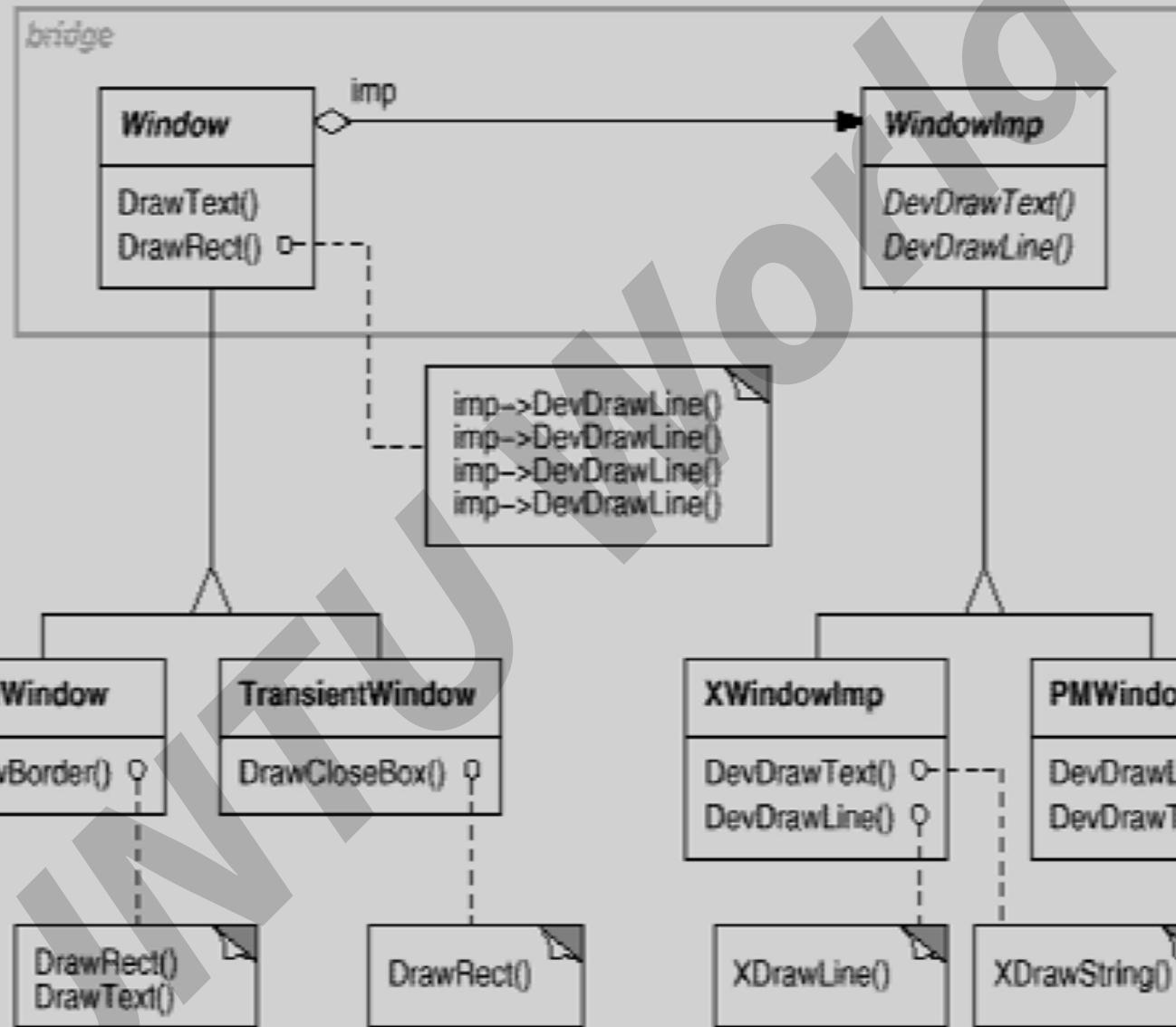
- 1. It's inconvenient to extend the window abstraction to cover different kinds of windows or new platforms**
- 2. It makes client code platform-dependent.**

SOLUTION

Clients should be able to create a window without committing to a concrete implementation.

The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies.

There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations, with WindowImp as its root



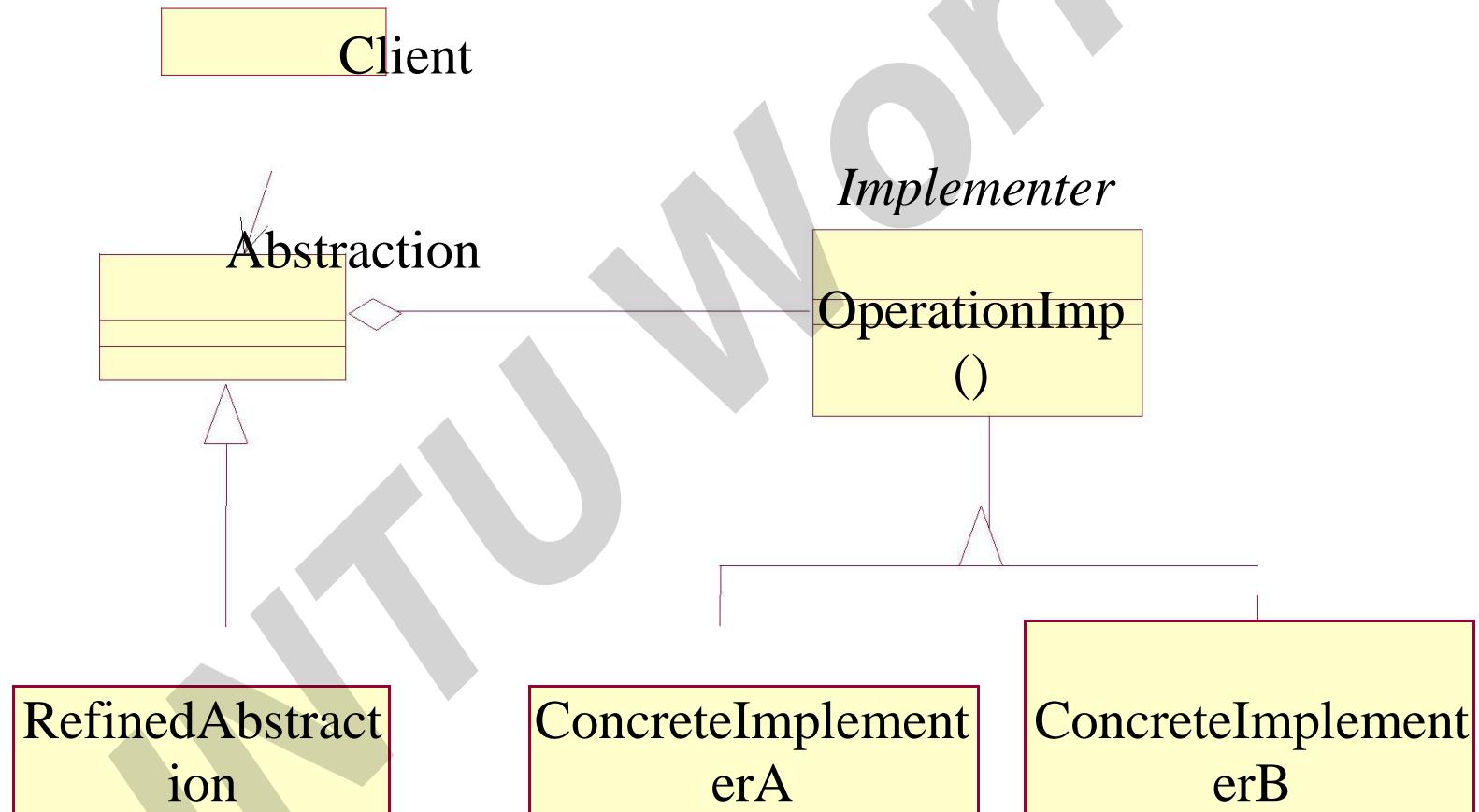
Applicability

You Want to avoid a permanent binding between an abstraction and implementation.

When abstractions and implementations should be extensible through subclassing.

3) you want to hide the implementation of an abstraction completely from clients

Structure





JNTUWorld

Participants

Abstraction (window):

- *define the abstraction's interface*
- *maintains a reference to an object of type Implementor*

Refined Abstraction (iconwindow):

- *extends the interface defined by Abstraction*

Implementor (WindowImp):

- *defines an interface for implementation classes.*

CONCREteImplementor (XWindowImp, PMWindowImp):

- *implements the Implementor's interface and defines its concrete implementation.*

Collaborations

- Abstraction forwards client requests to its Implementor object.

Consequences

- *Decoupling interface and implementation*
 \endash *Improved extensibility*

3. Hiding implementation details from clients

Implementation

- 1) *Only one Implementor*
2. *Creating the right Implementor*

17. Question Bank

UNIT-1

1. Discuss about implementation issues in builder design pattern.
2. Explain sample code of builder design pattern.

3. Explain the features of structural patterns in detail.
4. Explain the motivation of adapter design pattern.

5. State the differences between Traversal and Traversal actions.
6. Explain visitor class and subclasses in detail.

7. Explain with a neat diagram the Design Pattern relationships.

8. Explain the Known uses & related patterns of Visitor pattern.
9. What is the structure & participants of Momento pattern?

UNIT-II

10. Explain the class design structure of an editor for music scores with suitable design pattern.

11. Explain the motivation of Iterator pattern.
12. Explain the structure & participants of chain of Responsibility with one example.

13. Explain the motivation for known Facade method with relevant Patterns.
15. What is the intent uses & related pattern of Decorator Method?

15.What is a design pattern?

16.Explain how to select a Design Pattern.

17.Give the Catalog for DP

18.Organize the catalog on design patterns.

19.Explain the Abstract Factory design pattern in detail.

UNIT-III

20.What is the intent on Write a detailed code on Builder design pattern?

21.Explain Facade Design pattern in detail.

22.Give the intent and code on the Bridge design pattern.

23.Give the indent and code on the Chain of Responsibility design pattern.

25.Explain the structure & participants of Command pattern? Give an example
to Command pattern

25.What is an architectural pattern? Discuss any two types of architectural

26. Briefly discuss about the architectural structures with suitable example.

27. Discuss the factors for evaluating architecture.

28. Explain the roles and responsibilities involved in ATAM evaluation.

29. Briefly explain how stakeholders influence the architect.
30. Illustrate the process of evaluating the product line architecture.

UNIT-IV

31. Which pattern separates user interface from functional core?
32. Compare and contrast architectural patterns, design patterns
33. Briefly discuss a step-by-step approach how to use a design pattern
35. Illustrate the relationship between the different types of design patterns
35. A patient is suffering from Headache for some time and hence forth hospital where he stated the same to the receptionist.
36. Briefly discuss applicability of façade structural patterns.
37. Discuss the application of visitor class and visitor pattern
38. Compare and contrast Mediator, Strategy and Observer design patterns.
39. Discuss the case study of A-7E in utilizing architectural structures.
40. What is software architecture? Explain.

UNIT-V

41. Explain the Architectural patterns, Reference models,
42. Draw the process flow diagram for the Cost Benefit Analysis Model (CBAM)
43. discuss the case study of NASA ECS project.
45. Explain the product lines architecture and discuss the reasons that makes
45. Discuss the motivation, applicability, participants, collaborations,
46. Draw the structure and also discuss the motivation, applicability,
47. Discuss the motivation, applicability, participants, collaborations,
48. Draw the structure and also discuss the motivation, applicability,
49. Describe the three views of Celsius Tech architecture case study.

50.Explain the implementation issues of chain or responsibility pattern with sample code.

51. Illustrate Alexander's pattern language.

UNIT-VI

52.Explain about a Target Refactoring approach method.

53. Discuss about implementation issues in builder design pattern.

55. Explain sample code of builder design pattern.

55. What are the implementation considerations of Momento pattern?

56.Discuss the Applicability of Observer pattern.

57. Discuss about embellishing the user interface in detail.

58.Explain about supporting multiple window systems in designing a document Editor.

59.Differentiate between adapter and bridge design pattern.

60. What are the known uses of adapter structural pattern? Explain.

61.Explain the collaborations and consequences of Proxy pattern.

62.Explain the Motivation of Flyweight Pattern.

63.What are the problems of Object Oriented design? Explain.

UNIT-VII

65.What are the applications of Flyweight Pattern? Explain the structure & participants of Flyweight pattern with suitable example.

65.Explain how to access scattered information and to encapsulate access and Traversal.

66.Explain Transperent Enclosure with an example.

67. Write a detailed note on builder design pattern.

68. Explain the Motivation of command pattern.

69. Explain the structure & participants of command pattern with one example.

UNIT-VIII

70. Explain how to select a Design Pattern.

71. How can we relate RunTime and Compile Time structures? Explain.

72. Explain the implementation issues of Visitor pattern with sample code.

73. Explain the following:

(a) A target refactoring

(b) Patterns in software.

75. Mention the uses & related patterns of bridge design pattern.

75. Mention the participants of bridge pattern and explain the functions of each.

18. Assignment Questions

UNIT-1

1. a) What is an architectural pattern? Discuss any two types of architectural

b) Briefly discuss about the architectural structures with suitable example.

2. a) Discuss the factors for evaluating architecture.

b) Explain the roles and responsibilities involved in ATAM evaluation.

UNIT-11

3. a) Briefly explain how stakeholders influence the architect.

b) Illustrate the process of evaluating the product line architecture

5. a) Which pattern separates user interface from functional core?

b) Compare and contrast architectural patterns, design patterns

UNIT-III

6. a) Briefly discuss a step-by-step approach how to use a design pattern
- b) Illustrate the relationship between the different types of design patterns
7. a) A patient is suffering from Headache for some time and hence forth hospital where he stated the same to the receptionist.
b) Briefly discuss applicability of façade structural patterns.
8. a) Discuss the application of visitor class and visitor pattern
- b) Compare and contrast Mediator, Strategy and Observer design patterns.

UNIT-IV

9. Discuss the case study of A-7E in utilizing architectural structures.
- 10.a) Discuss about implementation issues in builder design pattern.
(b) Explain sample code of builder design pattern.
11. (a) Explain the features of structural patterns in detail.
(b) Explain the motivation of adapter design pattern

UNIT-V

- 12 (a) State the differences between Traversal and Traversed actions.
(b) Explain visitor class and subclasses in detail
13. Explain with a neat diagram the Design Pattern relationships.
- 14.(a) Explain the Known uses & related patterns of Visitor pattern.
(b) What is the structure & participants of Momento pattern?
- 15 Explain the class design structure of an editor for music scores with suitable design pattern

UNIT-VI

- 16(a) Explain the motivation of Iterator pattern.
- (b) Explain the structure & participants of chain of Responsibility with one example.

17(a) Explain the motivation for known Facade method with relevant Patterns.

(b) What is the intent uses & related pattern of Decorator Method?

UNIT-VII

18. what are the uses of abstract factory design pattern ? Explain

19. write a detailed note on prototype design pattern

20. Explain how to select a design pattern

21. How can we relate run time and compile time structures. Explain

UNIT-VIII

22. Write about (a)delegation (b) A common design vocabulary (c) The Object Community

23. Explain the collaboration of bridge pattern.

24. Write a short note on a implementation issues of composite pattern.

19. Unit-wise Quiz Questions And Long Answer Questions

QUIZ QUESTIONS

UNIT-1

1. Describe the basic approach used in functional decomposition.

Functional decomposition is the approach to analysis that breaks down (decomposes) a problem into its functional parts without too much concern for global requirements and future modifications. (p. 5)

2. What are three reasons that cause requirements to change?

The user's understanding of what they need and what is possible grows and changes as they discuss the problem with analysts. The developer's understanding of what is possible and what is needed evolves as they become familiar with the domain and with the software. The technical environment evolves, forcing changes in how to implement. (p. 6)

3. I advocate thinking about responsibilities rather than functions. What is meant by this? Give an example.

Rather than thinking first about how something is done (functions), the analyst should focus on what the routine is responsible for doing - how it does it does not matter. The control program is much simpler in this case. (p. 12).

5. Define "coupling" and "cohesion". What is "tight" coupling?

Cohesion is how strongly the internal operations of a routine are related to each other. Coupling is how strongly a routine is dependent upon other routines. (p. 8)

5. What is the purpose of an "interface" to an object?

It provides the methods whereby other objects can tell the object what to do. (p. 16)

6. Define instance of a class.

A specific, unique occurrence of a more abstract object. An object is an instance of a class. (p. 17)

7. A class is a complete definition of the behavior of an object. What three aspects of an object does it describe?

The three elements of a class are: the data elements, the methods, the interfaces (ways that data and methods can be accessed). (p. 17)

8. What does an abstract class do?

At the conceptual level, an abstract class is a placeholder for a set of classes. It gives a way to assign a name or label to a set of classes. At the specification level, an abstract class is a class that does not get instantiated. (p. 19)

9. What are the three main types of accessibility that objects can have?

Public, Protected, Private (p. 20)

10. Define encapsulation. Give one example of encapsulation of behavior.

Any kind of hiding. Both data and behavior may be encapsulated. (p. 21)

11. Define polymorphism. Give one example of polymorphism.

The ability to refer to different derivations of a class in the same way.

12. What are the three perspectives for looking at objects?

Conceptual: the high-level concepts in a system (concepts, not software). At the conceptual level, an object is a set of responsibilities.

Specification: the interfaces between things in the software (software, not code). At the specification level, an object is a set of methods.

Implementation: how an individual routine works (code). At the implementation level, an object is code and data. (p. 13, 15-16)

UNIT-1I Interpretations

13. Sometimes, programmers use "modules" to isolate portions of code. Is this an effective way to deal with changes in requirements? Why or why not?

Changes to one function or routine can have impacts on other routines. Usually, routines are not independent (p. 10).

15. It is too limited to define an abstract class as a class that does not get instantiated. Why is this definition too limited? What is a better (or at least alternative) way to think about abstract classes?

It is too limited because it only talks in terms of its implementation: what the abstract class does and how it is treated as software. It does not describe why I would want to use an abstract class: the motivation for it and how to think about it. It ignores the "conceptual perspective" of objects that analysts need to keep in mind as they work with users to understand problems. At the conceptual level, an abstract class is a placeholder for a set of classes. It gives a way to assign a name or label to a set of classes so that I can interact with them as a whole without getting trapped by the details. (p. 19)

15. How does encapsulation of behavior help to limit the impact of changes in requirements? How does it save programmers from unintended side effects?

It makes the control program much less complicated since it does not have to be responsible for as much. It limits the impact that changes to the internals of an object can have on the rest of the application. (p. 25)

16. How do interfaces help to protect objects from changes that are made to other objects?

Interfaces define the only ways that those external objects can communicate with the object. It protects me from side effects because I know what is coming into the system.

17. A classroom is used to describe objects in a system. Describe this classroom from the conceptual perspective.

The classroom contains students who are responsible for their own behaviors: how to move from here to there, how to go from class to class. It contains a teacher who tells students where to go.

Opinions and Applications

- 1. Changing requirements is one of the greatest challenges faced by systems developers.
Give one example from your own experience where this has been true.**
- 2. There is a fundamental weakness in functional decomposition when it comes to changes
in requirements. Do you agree? Why or why not?**

- 1. What is the difference between an "is-a" relationship and a "has-a" relationship? What
are the two types of "association" relationships?**

"is-a" indicates that one object is a "kind of" a class; for example, a "sail boat" is a kind of "boat" which is a kind of "type of transportation".

"has-a" indicates that one class "contains" another class; for example, a car has wheels.

There are two types of "associations": containment (has-a) and "uses" (p. 29)

- 2. In the Class diagram, a class is shown as a box, which can have up to three parts.
Describe these three parts.**

The top box is the name (label) of the class. This is required.

The middle box, if it is shown, shows the data members of the class.

The bottom box, if it is shown, shows the methods (functions) of the class. (p. 32)

3. Define cardinality.

Cardinality indicates the number of things that another object can have (p. 36)

5. What is the purpose of a Sequence diagram?

The Sequence diagram is one type of Interaction Diagram in the UML. It shows how objects interact with other objects. (p. 38)

Interpretations

- 1. Give an example of an "is-a" relationship and the two "association" relationships. Using
these examples,**

Draw them in a Class diagram

Show cardinality on this Class diagram

Is-a example: "Sailboat" is-a "boat"

Has-a example: Sailboat has-a sail (one to many)

Uses example: A marina contains one or more Sailboats (p. 32)

2. Figure 2-8 shows a Sequence diagram. How many steps are shown in the figure? How many objects are shown and what are they?

There are 13 steps in the diagram

There are 6 objects shown: Main, ShapeDB, Collection, shape1:Square, shape2:Circle, and Display. (p. 39)

3. When objects communicate with each other, why is it more appropriate to talk about "sending a message" than "invoking an operation"?

When objects "talk" to each other, it is called "sending a message." You are sending a request to another object to do something rather than telling the other object what to do. You allow the other object to be responsible enough to figure out what to do. Transferring responsibility is a fundamental principle of object-oriented programming. It is quite different from procedural programming where you retain control of what to do next, and thus might "call a method" or "invoke an operation" in another object.

Opinions and Applications

1. How many steps should be shown on a Sequence diagram?

As many as it takes to communicate clearly, and no more (p. 31)

[top](#)

UNIT-III A Problem That Cries Out for Flexible Code

Observations

1. What five features in sheet metal will this system have to address?

The features are Slot, Hole, Cutout, Special, and Irregular (p. 57)

2. What is the difference between the V1 system and the V2 system?

The V1 system has a collection of subroutine libraries that interacts with the CAD/CAM model. To get information about the CAD/CAM model, you have to make a series of calls (p. 53)

The V2 system is an object-oriented system. The geometry is stored in objects, each of which represents a feature. To get information about a feature, you interrogate the object for that feature. (p. 55)

1. What is the essential challenge of the CAD/CAM problem?

We have different types of CAD/CAM systems. A third system (the "expert system") has to extract information from whichever CAD/CAM system in order to work with the geometry. The two CAD/CAM systems are implemented in completely different ways and require completely different ways of interacting with them, even though they contain essentially the same information (p. 55)

2. Why is polymorphism needed at the geometry-extractor level but not at the feature level?

Polymorphism is required at the geometry extractor level because the "expert system" needs to know what type of features it is dealing with: slot, hole, etc. It is insufficient for the expert system simply to work on generic "features." Polymorphism does not buy me anything at the feature level. The expert system does not need to care about the particular method that is used to extract that feature. While we could hard-code the extraction method into the expert system, that would be bad if we ended up getting a new CAD/CAM system that uses yet another method of working with geometry. Polymorphism frees us from having to worry about the particular extraction method: the expert system can simply use a generic "geometry extractor" that worries about extractions. (p. 52)

Opinions and Applications

1. I spend time defining terms related to the CAD/CAM problem.

Why did I do this?

Did you find this useful or a distraction?

Is it important to understand the user's terminology?

What is the most effective method you have found for recording user terminology?

[top](#)

A Standard Object-Oriented Solution

Observations

1. Identify each of the elements of the UML diagram in Figure 5-3.

Abstract class

Cardinality

Derivation

Composition

Public methods

Abstract class: Feature (in italics)

Cardinality: A Model can have no Features, 1 Feature, or many Features.

Derivation: SlotFeature, a HoleFeature, a CutoutFeature, an IrregularFeature, or a SpecialFeature all derive from Feature. They are all "kinds of" Features.

Composition: A Model is composed of Features

Public method: GetOperations is a public method of the CutoutFeature.

2. What is the essential ability required by the CAD/CAM application?

They need the ability to plug-and-play different CAD/CAM systems without changing the expert system (p. 63)

3. The first solution exhibits four problems. What are they?

There is redundancy amongst the methods It is messy

It has tight coupling: features are related to each other

It has low cohesion: core functions are scattered amongst many classes. (p. 63)

Interpretations

1. Describe the first approach to solving the CAD/CAM problem. Was it a reasonable first approach?

The first object-oriented approach to a solution is to specialize a feature for each case: a Slot class for V1 and a Slot class for V2. Each V1 type case communicates with the V1 libraries and V2 type case communicates with V2 libraries. It is a reasonable approach to begin with (p. 59). It gives insights into the problem. But it should not be implemented!

Opinions and Applications

1. "Delay as long as possible before committing to the details." Do you agree? Why or why not?

2. One solution was rejected because "intuition told me it was not a good solution." Is it appropriate for analysts / programmers to be guided by their instincts?

[top](#)

UNIT-1VAn Introduction to Design Patterns

Observations

1. Who is credited with the idea for design patterns?

The architect, Christopher Alexander developed design patterns in the late 1970s. The "Gang of Four" took this idea in the 1990s and applied them to software design. I point out that one school of anthropology used patterns to study cultures in the 1950s. (p. 72). Also, the ESPRIT consortium used patterns for understanding human thought patterns in ways that could be implemented in computer programs in the 1980s (p. 77)

2. Alexander discovered that by looking at structures that solve similar problems, he could discern what? Designs / solutions that are high quality. And that this was objectively measurable (p. 73)

3. Define pattern. A pattern is a solution to a problem that occurs in a given context. (p. 75)

5. What are the key elements in the description of a design pattern?

To be complete, a pattern description must have the following eight elements:

- Name: a label that identifies it
- Intent: a description of the purpose of the pattern
- Problem: a description of the problem being solved
- Solution: what the solution is in the given context
- Participants / Collaborators: the entities involved in the solution
- Consequences: what happens as a result of using the pattern. What forces are at work.
 - Implementation: how to implement the pattern in one or more concrete ways.

- GoF Reference: where to look in the Gang of Four book for more information. (p. 79)

5. What are three reasons for studying design patterns?

Patterns make it possible to reuse solutions

Patterns help with communication between analysts, giving a shorthand terminology.

Patterns give you perspective on the problem, freeing you from committing to a solution too early. (p. 80)

6. The Gang of Four suggests a few strategies for creating good object-oriented designs. What are they?

Design to interfaces

Favor aggregation over inheritance Find what varies and encapsulate it (p. 85)

Interpretations

1. "Familiarity sometimes keeps us from seeing the obvious." In what ways can patterns help avoid this? We can gain insights from previous solutions, have our attention drawn to features of the problem that I might not otherwise think of (until too late) (p. 80)

2. The Gang of Four cataloged 23 patterns. Where did these patterns come from? It came from their insights into solutions that had already been developed within the software community. (p. 78)

3. What is the relationship between "consequence" and "forces" in a pattern? Consequences are the cause-and-effect of using the pattern (p. 79) Forces are the factors at play in a particular problem that constrain and shape the possible solutions. (p. 79)

5. What do you think "find what varies and encapsulate it" means? Look for what is changing and make a more generic version of it so that you can see what is truly going on in your system and not get caught up in the details. (p. 78)

5. Why is it desirable to avoid large inheritance hierarchies? They are very complex to understand and to maintain. (p. 86)

Opinions and Applications

1. Think of a building or structure that felt particularly "dead". What does it not have in common with similar structures that seem to be more "alive"?

2. "The real power of patterns is the ability to raise your level of thinking." Have you had an experience in which this was true? Give an example.

[top](#)

The Façade Pattern

Observations

1. Define Façade.

A Façade is "The face of a building, especially the principal face" - dictionary.com. It is the front that separates the street from the inside.

2. What is the intent of the Façade pattern?

Provide a unified interface to a set of interfaces in a sub-system (p. 87)

3. What are the consequences of the Façade pattern? Give an example.

The Façade simplifies the use of the required subsystem. However, since the Façade is not complete, certain functionality may be unavailable to the client. (p. 90). Example is a reporting application that needs a routine way to access on certain portions of a database system: The Façade would provide an interface to those portions and not the entire API of the database.

5. In the Façade pattern, how do clients work with subsystems?

Clients work with sub-systems through the Façade's interfaces. They do not interact with the underlying methods directly (p. 91)

5. Does the Façade pattern usually give you access to the entire system?

Not usually. In general, Façade give access to a portion of the system, one that is customized to our needs. (p. 89)

Interpretations

1. The Gang of Four says that the intent of the Façade pattern is to "provide a unified interface to a set of interfaces in a sub-system. Façade defines a higher-level interface that makes the subsystem easier to use." What does this mean? Give an example.

The Façade gives a simpler way to access an existing system by giving an interface that is customized to the needs you have. (p. 90)

Example is a class that insulates a client program from a database system (p. 92)

2. Here is an example of a Façade that comes from outside of software. Pumps at gasoline stations in the US can be very complex. There are many options on them: how to pay, the type of gas to use, watch an advertisement. One way to get a unified interface to the gas pump is to use a human gas attendant. Some states even require this.

- What is another example from real life that illustrates a Façade?
- Another example could be a stockbroker who serves as the interface to a complex system of stock trades.

Opinions and Applications

- 1. If you need to add functionality beyond what the system provides, can you still use the Façade pattern?**
- 2. What is a reason for encapsulating an entire system using the Façade pattern?**
- 3. Is there a case for writing a new system rather than encapsulating the old system with Façade? What is it?**
- 5. Why do you think the Gang of Four call this pattern "Façade"? Is it an appropriate name for what it is doing? Why or why not?**

[top](#)

UNIT-V The Adapter Pattern

Observations

1. Define Adapter.

"Adapter" is something that allows one thing to modify itself to conform to the needs of another thing.

2. What is the intent of the Adapter pattern?

The intent of the Adapter is to match an existing object that is beyond your control to a particular interface. (p. 102)

3. What are the consequences of the Adapter pattern? Give an example.

A consequence is that the pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces (p. 102). The example in the book is the drawing program that wants to use an existing Circle object but the existing object doesn't provide exactly

the same methods as the rest of the system. The Adapter provides a translation to these methods. (p. 101)

5. Which object-oriented concept is being used to define the relationship between Shape and Points, Lines, and Squares?

Polymorphism (p. 101)

5. What is the most common use for the Adapter pattern?

To allow for continued use of polymorphism. It is often used in conjunction with other design patterns. (p. 101)

6. What does the Adapter pattern free you from worrying about?

Adapter frees me from worrying about the interfaces of existing classes when doing a design. If the class doesn't do what I need, I can create an Adapter to give it the correct interface. (p. 103)

7. What are the two variations of the Adapter pattern?

Object Adapter: relies on one object to contain the other object.

Class Adapter: uses multiple inheritance to provide the interface. (p. 103)

Interpretations

1. The Gang of Four says that the intent of the Adapter pattern is to "convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces."

- What does this mean?
- Give an example.

It means that I have a class that needs to interact with another class through a certain set of method calls. If the interface of that other class does not provide these method calls, the Adapter sets up a new interface to do the translation. (p. 103) An example would be a reporting application that needs pulls data from two different database systems. My application wants to use a "GetDate" method to pull information from the database, but the database systems don't provide that through their API. I write an Adapter that provides the GetDate method in its interface and is responsible for pulling the data appropriately.

2. "The Circle object wraps the XXCircle object." What does this mean?

Circle completely insulates XXCircle from the system. Circle manifests the entire behavior of XXCircle to the system, although with a different interface / way of accessing XXCircle. (p. 100)

3. The Façade pattern and the Adapter pattern may seem similar. What is the essential difference between the two?

In both cases, there is a preexisting class or classes that have functionality I need. In both cases, I create an intermediary object with interfaces that my system wants to use and that has responsibility for mapping that to the preexisting class. Both Façade and Adapter are wrappers.

The Adapter is used when the client already has predefined interfaces that it expects to use and when I need to use polymorphism.

The Façade is used when I need a simpler interface to the existing object. (p. 105)

5. Here is an example of an Adapter that comes from outside of software. A translator at the UN lets diplomats from different countries reason about and argue for the positions of their own countries in their own languages. The translator makes "dynamically equivalent" representations from one language to the other so that the concepts are communicated in the way that the recipient expects and needs to hear it.

What is another example from real life that illustrates an Adapter?

Another example could be a travel agent, seen as the common interface between a passenger making arrangements and an airline with its own systems. Each has competing systems, speaking different languages

Opinions and Applications

1. When is it more appropriate to use the Façade pattern rather than the Adapter pattern? How about the Adapter pattern instead of Façade pattern?

2. Why do you think the Gang of Four call this pattern Adapter? Is it an appropriate name for what it is doing? Why or why not?

[top](#)

Expanding Our Horizons

Observations

1. What do I say is the right way to think about encapsulation?

Encapsulation is best thought of as "any kind of hiding." This can mean hiding data, or behavior, or implementations, or derived classes, or any other thing. (p. 113)

2. What are the three perspectives for looking at a problem? (You may need to review Chapter 1, "The Object-Oriented Paradigm").

The three perspectives are the Conceptual perspective, the Specification perspective, and the Implementation perspective.

Interpretations

1. There are two mention different ways to understand objects: "data with methods" and "things with responsibilities."

- In what ways is the second approach superior to the first?
 - What additional insights does it provide?

The second approach takes looks at what an object is supposed to do, what its essential concepts are, without worrying about how to do them. It fights against the tendency of programmers to want to jump to coding too soon. (p. 110-111).

By focusing on what an object is supposed to do rather than how it does it, I can be more flexible in design. It helps to think about the public interfaces that will be required and what those interfaces need to do. (p. 111).

2. Can an object contain another object? Is this different than one object containing a data member?

In object-oriented systems, everything is an object. An object can contain another object, data, or anything. In fact, data types are also objects, so there is no difference. (p. 118)

3. What is meant by the phrase find what varies and encapsulate it? Give an example

Variation represents special cases that complicate understanding. At the conceptual level, find a common label to a set of these variations. Variation can be in data, in behavior (p. 116)

5. Explain the relationship between commonality/variability analysis and the three perspectives of looking at a problem.

By looking at what objects must do (the Conceptual perspective), we determine how to call them (the Specification perspective). (p. 119). Commonality / Variability analysis reveals the interfaces I need to handle all of the cases of the concept. (p. 121)

Specifications become abstract classes at the implementation level (p. 120). Given a specification, the Implementation perspective shows how each of its variations must handled. (p. 121).

5. An abstract class maps to the "central binding concept." What does this mean?

The core concept is what defines what is common across a set of things that vary. An abstract class represents this core concept. The name you give to that core concept is the name for the abstract class. (p. 120)

6. "Variability analysis reveals how family members vary. Variability only makes sense within a given commonality."

- What does this mean?
- What types of objects are used to represent the common concepts?
 - What types of objects are used to represent the variations?

Variability analysis looks for all of the variants of a concept: all of the concrete instances of an abstract class. The "commonality" labels the essential concept that ties the variations together. The goal is to find the best unifying name for the set of variations so that you can have a handle to work with them as a set: to work with the forest instead of the trees.

Abstract classes are used to represent the common concept. Concrete instances are used to represent the variations. (p. 139)

Opinions and Applications

1. Why is it better to start out focusing on motivations rather than on implementation?
Give an example where this has helped you.

2. Preconceived notions limit one's ability to understand concepts. This was shown to be the case with encapsulation. Can you think of a situation in which your preconceived notions got in the way of understanding requirements? What happened and how did you overcome it?

3. The term inheritance is used both when a class derives from a nonabstract class to make a specialized version of it and when an abstract class is used as a starting point for different implementations. Would it be better if we had two different terms for these concepts instead of using the same term?

5. How might you use commonality/variability analysis to help you think about ways to modify a system?

5. It is important to explore for variations early and often.

- Do you believe this? Why or why not?
- How does it help to avoid pitfalls?

6. Commonality/variability analysis is an important primary tool for identifying objects, better than "looking for the nouns." Do you agree? Why or why not?

7. This chapter tried to present a new perspective on objects? Did it succeed? Why or why not?

[top](#)

UNIT-VI The Strategy Pattern

Observations

1. What are some alternatives for handling new requirements?

Cut and paste

Switches or ifs on a variable specifying the case we have Using function pointers or delegates (a different one representing each case) Inheritance (make a derived class that does it the new way)
Design patterns

2. What are the three fundamental principles proposed by the Gang of Four that guide how to anticipate change?

"Program to an interface, not an implementation." 1

"Favor object aggregation over class inheritance." 2 "Consider what should be variable in your design."

3. What is the intent of the Strategy pattern?

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

5. What are the consequences of the Strategy pattern?

The Strategy pattern defines a family of algorithms.

Switches and/or conditionals can be eliminated. You must invoke all algorithms in the same way (they must all have the same interface).

Interpretations

1. The Gang of Four suggests "considering what should be variable in your design." How is this different from focusing on the cause of redesign?

The focus is on seeing where change might occur and then encapsulating it so that your system will not be affected by change when it occurs. It assumes you will not be able to anticipate what will change.

2. What is wrong with copy-and-paste?

duplications of code result in higher maintenance costs

3. What is "switch creep"?

The flow of the switches themselves becomes confusing. Hard to read. Hard to decipher. When a new case comes in, the programmer must find every place it can be involved (often finding all but one of them). I like to call this "switch creep".

5. What are the advantages of the design patterns approach to handing variation?

Improves cohesion

Aids flexibility Makes it easier to shift responsibility Aids understandability

5. Why is the object-aggregation approach to inheritance superior to direct class inheritance for handling variation?

But this simplifies the bigger, more complicated program. Second, by doing this, I have made inheritance better. When I need to use inheritance, there is now only one piece of functionality changing within any one class. The bottomline is, the approach espoused by patterns will scale while the original use of inheritance will not.

Opinions and Applications

- 1. Have you ever been in a situation where you did not feel you could afford to anticipate change? What drove you that way? What was the result?**
- 2. Should you ever use switch statements? Why or why not?**

[top](#)

UNIT-VII The Bridge Pattern

Observations

- 1. Define decouple and abstraction.**

Decouple means to separate or detach one thing from another. In our context, it means to have one thing behave independently from another (or at least to state explicitly what that relationship is)

Abstraction means to generalize or conceptualize: to step back from the more concrete to the more conceptual or abstract. (p. 125)

2. How is implementation defined in the context of the Bridge pattern?

Implementation refers to the objects that the abstract class and its derivations use to put themselves into operation or into service. (p. 125)

3. What are the basic elements of a sequence diagram?

The basic elements are:

- Boxes. These are shown at the top and represent the objects that are interacting.
 - Name in the form objectname:classname. The object name is optional.
- Dashed vertical lines, also known as swim lanes, one for each object, to indicate time.
- Arrows, may be horizontal or vertical, showing the interaction between objects. Each arrow is labeled to describe the interaction
 - Notes. This is optional. (p. 131)

5. What is Alexander's view of how to use patterns? Does he advocate starting with the solution first or the problem to be solved first?

Alexander says that a pattern describes a problem which occurs over and over again in the environment and then describes the core of the solution to that problem. This means that it is most important to understand the problem first and then tackle the solution. It is a mistake to try finding the solution first. (p. 137)

5. What does commonality analysis seek to identify? What does variability analysis seek to identify?

Commonality analysis focuses on finding structures that will not change over time while variability analysis looks for structures that are likely to change. (p. 139)

6. What is the basic problem being solved by the Bridge pattern?

The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes (p. 151).

7. Define the "one rule, one place" strategy.

"one rule, one place" says you should implement a rule in only one place. (p. 155). Note that this results in a greater number of smaller methods.

8. What are the consequences of the Bridge pattern?

Decoupling of the implementations from the objects that use them increases extensibility. Client objects are freed from being aware of implementation issues. (p. 151)

Interpretations

- 1. The Gang of Four says that the intent of the Bridge pattern is to "decouple an abstraction from its implementation so that the two can vary independently." What does this mean? Give an example.**

What it means is that you can have an abstraction that is independent of its implementations. (p. 150) An example is a shape object that is responsible for knowing shapes and a Drawing class that is responsible implementing drawing routines. Individual shapes don't have to know how to do drawings (p. 156)

2. Why can tight coupling lead to an explosion in the number of classes?

Tight coupling means that as you get more variations in implementation, each class has to be responsible for its own implementation.

Opinions and Applications

- 1. "Look at objects in terms of their responsibilities rather than their behaviors." How does this affect your view of the use of inheritance in an object-oriented system?**
- 2. Why do you think the Gang of Four call this pattern "Bridge"? Is it an appropriate name for what it is doing? Why or why not?**

[top](#)

The Abstract Factory Pattern

Observations

- 1. While using "switches" can be a reasonable solution to a problem that requires choosing among alternatives, it caused problems for the driver problem discussed in this chapter. What were these problems? What might a switch indicate the need for? The rules for determining which driver to use are intermixed with the actual use of the drivers. This creates both tight coupling and strong cohesion. (p. 165)**

Switches may indicate a need for abstraction (p. 166)

- 2. Why is this pattern called "Abstract Factory"?**

At first glance, you might be tempted to conclude it is because the factory is implemented as an abstract class with a derivation for each case. But that is not the case. This pattern is called the "Abstract Factory" because the things it is intended to build are themselves defined by abstractions. How you choose to implement the factory variations is not specific to the pattern.

3. What are the three key strategies in the Abstract Factory?

Find what varies and encapsulate it

Favor aggregation over inheritance

Design to interfaces, not to implementations (p. 171)

5. In this pattern, there are two kinds of factories. What does the "Abstract Factory" class do? What do the "concrete factory" classes do?

The "Abstract Factory" class specifies which objects can be instantiated by defining a method for each type of object.

The "concrete factory" classes specify which objects are to be instantiated. (p. 175)

5. What are the consequences of the Abstract Factory pattern?

The Abstract Factory isolates the rules about which objects to use from the logic about how to use these objects. (p. 176)

Interpretations

1. The Gang of Four says that the intent of the Abstract Factory pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes." What does this mean? Give an example.

It means that I need to coordinate the instantiation of several objects, a family of objects. However, I want to insulate my system from having to know specifics of the particular concrete object being instantiated. That is, the selection of which particular concrete instance to use might depend upon another factor. An example would be a system that wants to manage records in a database but be insulated from the specifics of which DBMS is being used. (p. 163)

Opinions and Applications

1. Why do you think the Gang of Four call this pattern "Abstract Factory"? Is it an appropriate name for what it is doing? Why or why not?

2. How do you know when to use the Abstract Factory pattern?

[top](#)

How Do Experts Design?

Observations

- 1. Alexander uses the term, "alive" to characterize good designs. What terms do I suggest using when it comes to software?**

"When you read 'alive', think 'robust' and 'flexible' systems. (p. 189)

- 2. Good design requires keeping what in mind?**

Keeping the big picture in mind. Being able to consider the forest first and then the trees. (p. 189)

- 3. Alexander suggests that the best approach to design involves "complexification." What does this mean?**

Complexification is the approach to design to starts by looking at the problem in its simplest terms and then adds additional features (distinctions), making the design more complex as we go because we are adding information (p. 190)

- 5. To Alexander, what relationships does a pattern define?**

A pattern defines relationships between the entities in his problem domain (p. 191, 192) This is why define a pattern as a solution to a problem in a context. The entities describe the context in which the pattern exists.

- 5. What are Alexander's five steps to design?**

Identify patterns that are present in your problem.

Start with context patterns (those that create context for other patterns)

Work inward from the context

Refine the design

Implement (p. 193)

Interpretations

- 1. I quote Alexander, "But it is impossible to form anything which has the character of nature by adding preformed parts." What does Alexander mean by this?**

Alexander believes that designs that have the "character of nature" are those that humans would judge has being superior in design. They are "alive" and feel right. He believes that buildings (or

in our case, software) that is built simply by assembling stock parts will not feel "alive". They will have all the charm of 60s style block houses: functional but dead. In software terms, it works the same way: cobbling together objects does not create solutions that are easily maintained: robust and flexible. (p. 188, 189)

Opinions and Applications

- 1. Sometimes, the case that is made for object-oriented programming is that it gives you small, reusable components that you can assemble to create a program. Does this align with Alexander or contradict him? Or is Alexander speaking at a different level? Why?**
- 2. Have you ever seen a courtyard or entryway in a house or building that has felt particularly "dead" or uninviting? As you look at Alexander's description of the Courtyard pattern, what entities did your courtyard fail to resolve or involve?**
- 3. Think of one software project in which you think Alexander's approach would apply or an approach in which it would not apply. What are the issues? Keep this case in mind as you read the rest of the book.**

[top](#)

Solving The CAD/CAM Problem with Patterns

Observations

- 1. What are the three steps to software design with patterns that I use?**

Find the patterns in the problem domain

For the set of patterns to be analyzed, pick the pattern that provides the most context and apply it to the conceptual design. Identify additional patterns that are now suggested. Repeat.

Add detail to the conceptual design. Expand the method and class definitions. (p. 199).

2. Define "context."

One definition is "the interrelated conditions in which something exists or occurs. An environment or a setting." (p. 201)

3. What do I mean by "seniormost" pattern?

The seniormost pattern is the pattern that creates the context for the other patterns. When it comes to applying patterns to a design, we want start with the "seniormost" patterns first and then work down (p. 203)

5. When comparing two patterns, I suggest two rules for discerning which pattern might be seniormost. What are these rules? Does one pattern define how the other pattern behaves?

Are the two patterns interrelated? Mutually dependent? (p. 205)

5. Define "canonical form" of a pattern. When is it used?

The canonical form of a pattern is its standardized, simplified representation. This is generally what is shown in the Gang of Four book and is shown in each of the pattern descriptions in Design Patterns Explained. I suggest starting with the canonical form and then mapping classes and elements of the problem into it. (p. 208)

Interpretations

1. Do I believe that your entire problem can always be defined in terms of patterns? If not, what else is needed?

The answer is "not always." Generally, patterns give you a way to get started with understanding the problem. However, analysis remains a human activity! (which is good because we still have jobs!). It is usually the case that the analyst ends up having to identify relationships amongst concepts in the problem domain. One good approach to this is Commonality / Variability analysis, which has been discussed before. (p. 199)

2. In the CAD/CAM problem, I reject the Abstract Factory as the "seniormost" pattern. What reasons do they give?

The Abstract Factory requires knowing what classes will be defined. These are defined by other patterns. Therefore, Abstract Factory depends upon other patterns; they create the context for the Abstract Factory. Therefore, it is not seniormost. (p. 203)

3. In the CAD/CAM problem, what reason(s) do I give for labeling Bridge as senior to Adapter?

Clearly, there is a relationship between Bridge and Adapter. But Adapter's interfaces cannot be determined without Bridge. Without the Bridge, Adapter's interfaces simply don't exist. Since Adapter depends upon Bridge and not vice-versa, Bridge is more senior (p. 205)

Opinions and Applications

1. Once all of the patterns are applied, there are still likely to be more details. I assert that Alexander's general rules (design by starting with the context) still apply. Does this ever stop? Is there ever a time when you should go ahead and dive into the details? Isn't that what "rapid prototyping" suggests? How can you avoid this temptation that all programmers have? Should you?

- 2. Compare the first solution to the CAD/CAM problem (Figure 13-12) with the new version (Figure 13-11). What do you like better about the new design?**

[top](#)

The Principles and Strategies of Design Patterns

Observations

- 1. When it comes to choosing how to implement a design, what question do I suggest asking?**

Rather than ask, "Which implementation is better?" ask, for each alternative, "Under what circumstances would this alternative be better than the other alternative" and then "Which of these circumstances do I have in my problem domain?" (p. 221)

- 2. What are the five errors of using design patterns?**

Superficiality, Bias, Selection, Misdiagnosis, Fit

Interpretations

- 1. The "open-closed" principle says, "modules, methods, and classes should be open for extension while closed for modification." What does this mean?**

Bertrand Meyer puts this forward as a way to minimize risk when changing systems. Basically, it means that we want to be able to extend the capabilities of our systems without substantially changing it. Design in such a way that the software can absorb new variations without having to introduce new fundamental structure (p. 218).

- 2. In what way does the Bridge pattern illustrate the open-closed principle?**

Bridge allows us to add new implementations without changing any existing classes (p. 218).

Opinions and Applications

- 1. I suggest that even though a design pattern might give you insights into what could happen, you do not have to build your code to handle those possibilities. How do you decide which possibilities to handle now and which to be ready for in the future?**

- 2. Give a concrete example of the danger of misapplying a design pattern, based upon your current work.**

[top](#)

Commonality and Variability Analysis (CVA)

Observations

1. I suggest two approaches to identifying commonalities and variabilities. What are they?

Pick any two items in the problem domain and ask, "is one of these a variation of the other" and "are both of these a variation of something else".

Look at the problem and identify the major concepts.

Interpretations

1. CVA says you should have only one issue per commonality. Why is this important?

Having two issues per commonality leads to confusion in the relationship amongst the concepts.

When the connection is clear, then there is clear and thus strong cohesion amongst concepts.

2. How do CVA and design patterns complement each other?

CVA helps to identify what the essential concepts are. Design patterns do not necessarily lead to that.

Design patterns tell you what to do with those concepts, how to relate them based upon good designs from the past. CVA does not speak about best-practices, leaving that to the designer's imagination.

Opinions and Applications

1. I state that experienced developers - even more than inexperienced ones - often focus on entity relationships too early, before they are clear what the right entities are. Is that your experience? Give an example to confirm or refute this statement.

2. Relate the approach to design - starting with CVA - with Alexander's approach.

[top](#)

The Analysis Matrix

Observations

1. What goes in the far left column of the Analysis Matrix?

The essential concept represented by a function. (p. 293)

2. What do the rows of the Analysis matrix represent?

Each row represents specific, concrete implementations of the generalized concept described in the row. (p. 296)

3. What do the columns of the Analysis matrix represent?

Each column represents the specific implementations for each case. (p. 297)

5. Which patterns described in this book might be present in an Analysis Matrix?

In general, any pattern that uses polymorphism could be present in an Analysis Matrix. In this book, that involves Bridge, Decorator, Template, and Observer. (p. 299).

Interpretations

1. At what level of perspective does the Analysis Matrix operate?

The Analysis Matrix is focused on variations in concepts. It is used at the Conceptual Level (p. 293).

2. In what way is the Analysis Matrix similar to Commonality/Variability Analysis?

The Analysis Matrix is focused on variations in concepts. It starts by understanding the concept that a function represents and putting a label onto it. The Analysis Matrix works with these labels as abstractions for the function. CVA also works by abstracting variations and labeling them. (p. 293).

Opinions and Applications

1. Can patterns help handle variation more efficiently?

2. Do you agree with I' observations about users (p. 296)? Can you give examples from your own experience?

3. Do you believe that the Analysis Matrix is generally useful in most problem domains?

[top](#)

UNIT-VIII The Decorator Pattern

Observations

1. What does each Decorator object wrap?

Decorators wrap their trailing objects. Each Decorator object wraps its new function around its trailing object.

2. What are two classic examples of decorators?

Heading and footers

Stream I/O

Interpretations

1. How does the Decorator pattern help to decompose the problem?

The Decorator pattern helps to decompose the problem into two parts: How to implement the objects that give the new functionality; and how to organize the objects for each special case

2. In discussing the essence of the Decorator, I say that "the structure is not the pattern." What does this mean? Why is this important?

The Decorator pattern comes into play when there are a variety of optional functions that can precede or follow another function that is always executed.

Implementing the pattern by rote can lead to bad design. Instead, you need to think about the forces at work in the pattern and then think about ways to implement the intent of the pattern.
Patterns are not recipes.

Opinions and Applications

1. Why do you think the Gang of Four call this pattern "Decorator"? Is it an appropriate name for what it is doing? Why or why not?

2. Sometimes, people think of patterns as recipes. What is wrong with this?

[top](#)

The Observer Pattern

Observations

1. According to the Gang of Four, what are structural patterns responsible for?

Structural patterns are used for tying together existing functionality.

2. What are the three classifications of patterns, according to the Gang of Four? What is the fourth classification that I suggest?

The GoF specified three types: Structural, Behavioral, and Creational. I suggest "decoupling" as a fourth type.

3. What is the one true thing about requirements?

Requirements always change! Plan for it.

5. What is the intent of the Observer pattern?

The Gang of Four says that the intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Interpretations

1. Why are the Bridge and Decorator patterns more correctly classified as structural rather than behavioral patterns?

They both are tying together functionality, which is what structural patterns do. In the Bridge pattern, we typically start with abstractions and implementations and then bind them together with the bridge. In the Decorator pattern, we have an original functional class, and want to decorate it with additional functions.

2. One example of the Observer pattern from outside of software is a radio station: It broadcasts its signal; anyone who is interested can tune in and listen when they want to. What is another example from "real-life"?

Newspaper publishing could be another example

3. Under what conditions should an Observer pattern not be used? When the dependencies are fixed (or virtually so), adding an Observer pattern probably just adds complexity.

Opinions and Applications

1. I put forward the idea of a "fourth category" of patterns, that somewhat includes patterns from other categories. Is this a good idea? Why or why not?

[top](#)

The Template Method Pattern

Observations

1. The Template Method pattern makes the method call in a special way. What is that?

The method itself is general. It makes the method call via a reference pointing to one of the derived classes to handle the special details.

Interpretations

- 1. According to the Gang of Four, the intent of the Template Method pattern is to "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm's structure" What does this mean?**

It helps us to generalize a common process - at an abstract level - from a set of different procedures. It helps to identify the common ground between the set of different procedures while encapsulating the differences in derived classes

- 2. The Gang of Four calls this a "Template Method". Why do they do this?**

Because it provides a boilerplate (or a "template") that specifies the generic actions and derived class implements the specific steps required for the actions to take

- 3. What is the difference between the Strategy pattern (chapter 9) and the Template Method pattern?**

The Template Method pattern is applicable when there are different, but conceptually similar processes.

The Strategy pattern controls a family of algorithms. They do not have to be conceptually similar. You choose the algorithm to employ just in time.

[top](#)

Lessons from Design Patterns: Factories

Observations

- 1. How do I define a factory?**

A factory is a method, an object, or anything else that is used to instantiate other objects.

- 2. Name one factory pattern that was shown in a previous chapter. Name the factory patterns mentioned in this chapter**

The Abstract Factory was shown in chapter 11. In this chapter, the factories mentioned are Builder, Factory Method, Prototype, and Singleton

- 3. When it comes to managing object creation, what is a good, universal rule to use?**

An object should either make and/or manage other objects, or it should use other objects but it should never do both

Interpretations

1. I state that developers who are new to object-oriented programming often lump the management of object creation in with object instantiation. What is wrong with this?

It can lead to decreased cohesion because an object depends on or more other objects to ensure work is done before it can successfully continue.

2. I suggest that factories increase cohesion. What is their rationale for saying so?

Factories help to keep together both the functionality and the rules that determine which objects should be built and/or managed under different circumstances.

3. I suggest that factories also help in testing. In what ways is this true?

The "using objects" should behave in exactly the same way with any set of derivatives present. I should not need to test every possible combination, because I can test each piece individually. No matter how I combine them, the system will work in the same manner.

Opinions and Applications

1. I suggest that factories are useful for more than simply deciding which object to create or use. They also help with encapsulating design by solving a problem created by patterns? Evaluate this argument.

[top](#)

The Singleton Pattern and the Double-Checked Locking Pattern

Observations

1. What type of pattern is the Singleton? What general category of pattern does it belong to?

It is a type of Factory pattern

2. What is the intent of the Singleton pattern?

Ensure a class only has one instance, and provide a global point of access to it

3. How many objects is the Singleton responsible for creating?

one

5. The Singleton uses a special method to instantiate objects. What is special about this method?

When this method is called, it checks to see if the object has already been instantiated. If it has, the method simply returns a reference to the object. If not, the method instantiates it and returns a reference to the new instance.

To ensure that this is the only way to instantiate an object of this type, I define the constructor of this class to be protected or private.

5. What do I say is the difference in when to use the Singleton and Double-Checked Locking patterns?

The distinction between the patterns is that the Singleton pattern is used in single-threaded applications while the Double-Checked Locking pattern is used in multithreaded applications. Double-Checked must focus on synchronization in creations in case two objects try to create an object at exactly the same moment. This avoids unnecessary locking. It does this by wrapping the call to new with another conditional test. Singleton does not have to worry about this.

Interpretations

1. I state that they would rather have the objects be responsible for handling their own single instantiation than to do it globally for the objects. Why is this better?

It is encapsulation of behavior: it helps objects be less dependent on some other object to do something that will directly impact what that object can do.

It also frees other objects from worrying whether the object already exists. They can assume it does (or will) and that there is only one of that object to reference. They don't have to worry about getting the right one. As systems grow in size and complexity, trying to manage all of this quickly can get out of hand. But you have to be careful not to create global variables.

Opinions and Applications

2. Why do you think the Gang of Four call this pattern "Singleton"? Is it an appropriate name for what it is doing? Why or why not?

3. The authors state, "When it was discovered that the Double Checked Locking pattern as initially described did not work in Java, many people saw it as evidence that patterns were over-hyped. I drew exactly the opposite conclusion." Do you agree with their logic? Why or why not?

[top](#)

The Object Pool Pattern

Observations

1. What three general strategies do I suggest you follow?

Look for ways to insulate yourself from the impacts of changes to your system.

Focus on the hard things first

Trust your instincts.

2. What two patterns does the Object Pool pattern incorporate?

The Singleton pattern ensures that only one

The Factory pattern manages the creation and logic

3. What is the intent of the Object Pool pattern? Manages the reuse of objects when a type of object is expensive to create or only a limited number of objects can be created

Interpretations

1. What do the XP community mean by YAGNI?

You Aint Gonna Need It

It reflects the idea that you should build what you need now while ignoring the rest. You should work on the most important things early, when solving them can make the greatest impact. It also means you avoid working on things are at a minimum distracting, and typically never used (and therefore building is a waste of resources).

Opinions and Applications

1. Reading widely is an important discipline. You never know when you will find something you can use. One example is the example they found from Steve Maguire's book, Writing Solid Code. Give at least one example from your own experienced where this has been true for you.

[top](#)

The Factory Method Pattern Observations

1. What are factories responsible for?

Factories are responsible for creating objects and ensuring objects are available to be used.

2. What is the essential reason to use a Factory Method?

You want to defer the decision for instantiating a derivation of another class to a derived class

3. The Factory Method pattern has been implemented in all of the major object-oriented languages. How has it been used in Java, C#, and C++?

In Java, the iterator method on collections is a Factory Method.

In C#, the GetEnumerator is a Factory method on the different C# collections where it is present.

In C++, the methods used include begin() and end().

Interpretations

1. Why is this pattern called a "factory method?"

It uses a method to handle the factory

2. How does the Factory Method pattern fit in with other factories?

The Factory Method allows these other patterns to defer instantiation to subclasses. One uses the Factory Method to defer responsibility to subclass objects. The Abstract Factory can use a family of Factory Methods, one for each different family of objects involved. The Template Method can use a method to handle the instantiation; giving responsibility to that method is the factory.

3. The Gang of Four says that the intent of the Factory Method is to "define an interface for creating an object, but let subclasses decide which class to instantiate." Why is this important?

It is not always desirable for a class to have to know how to instantiate derived classes

Opinions and Applications

5. How should you go about deciding whether a method should be public, private, or protected?

5. This is a small chapter but this is not a small pattern. Think of one example where this pattern could be used.

[top](#)

Design Patterns Reviewed From Our New Perspective of Object-Oriented Principles

Observations

1. Several of the patterns have the characteristic of shielding implementations from what? What is this called? Give examples.

They shield implementation details from the Client object. This is one type of encapsulation. Bridge is one such pattern: It hides from the Client how the Abstraction is implemented. (p. 306)

2. What is one example of a pattern helping to think about decomposing responsibilities?

The Decorator pattern gives a way of decomposing responsibilities into the main set that are always used (ConcreteComponent) and variations that are options (decorators). (p. 309)

3. As you learn patterns, what five forces and concepts do I urge you to look for?

The five forces to look for are:

What implementations does this pattern hide?

What commonalities are present in this pattern?

What are the responsibilities of the objects in this pattern?

What are the relationships between these objects?

How may the pattern itself be a microcosmic example of designing by context? (p. 309)

Interpretations

1. What is the value of hiding implementations?

The patterns allow for adding new implementations by hiding details of current implementations.

This reflects the open-closed principle, making systems easier to endure over time

20.Tutorial Topics

Creational patterns

Name	Description
Abstract factory	Provide an interface for creating <i>families</i> of related or dependent objects without specifying their concrete classes.

Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Factory method	Define an interface for creating a <i>single</i> object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection^[15]).
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Resource acquisition is initialization	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

Structural patterns

Name	Description
Adapter or Wrapper or Translator.	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Twin ^[17]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.
----------------------	---

Behavioral Patterns

Name	Description
Blackboard	Generalized observer, which allows multiple readers and writers. Communicates information system-wide.
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes
Specification	Recombinable business logic in a Boolean fashion
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 subclasses 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.

Concurrency Patterns

Name	Description
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Balking	Only execute an action on an object when the object is in a particular state.
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way. ^[19]
Double-checked locking	<p>Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.</p> <p>Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.</p>
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multithreaded programs. ^[20]
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
Join	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the

	use of threads and locks, this is a high level programming model.
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it. ^[21]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations.
Scheduler	Explicitly control when threads may execute single-threaded code.
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
Thread-specific storage	Static or "global" memory local to a thread.

21.Known gaps ,if any and inclusion of the same in lecture schedule

No gaps since compared JNTU syllabus with NIIT Warangal syllabus both are same and didn't find any gaps

22Discussion topics

- 1: The Object-Oriented Paradigm
- 2: The UML - The Unified Modeling Language
- 3: A Problem That Cries Out for Flexible Code
- 5: A Standard Object-Oriented Solution
- 5: An Introduction to Design Patterns
- 6: The Façade Pattern
- 7: The Adapter Pattern
- 8: Expanding Our Horizons
- 9: The Strategy Pattern
- 10: The Bridge Pattern
- 11: The Abstract Factory Pattern
- 12: How Do Experts Design?
- 13: Solving The CAD/CAM Problem with Patterns
- 15: The Principles and Strategies of Design Patterns
- 15: Commonality and Variability Analysis (CVA)
- 16: The Analysis Matrix
- 17: The Decorator Pattern
- 18: The Observer Pattern
- 19: The Template Method Pattern
- 20: Lessons from Design Patterns: Factories