ISB 26504 Software Design and Integration

Chapter 07:Pattern in the Real World

INTRODUCTION TO DESIGN PATTERN

PRESENTATION BY:
MADAM ROBIAH HAMZAH

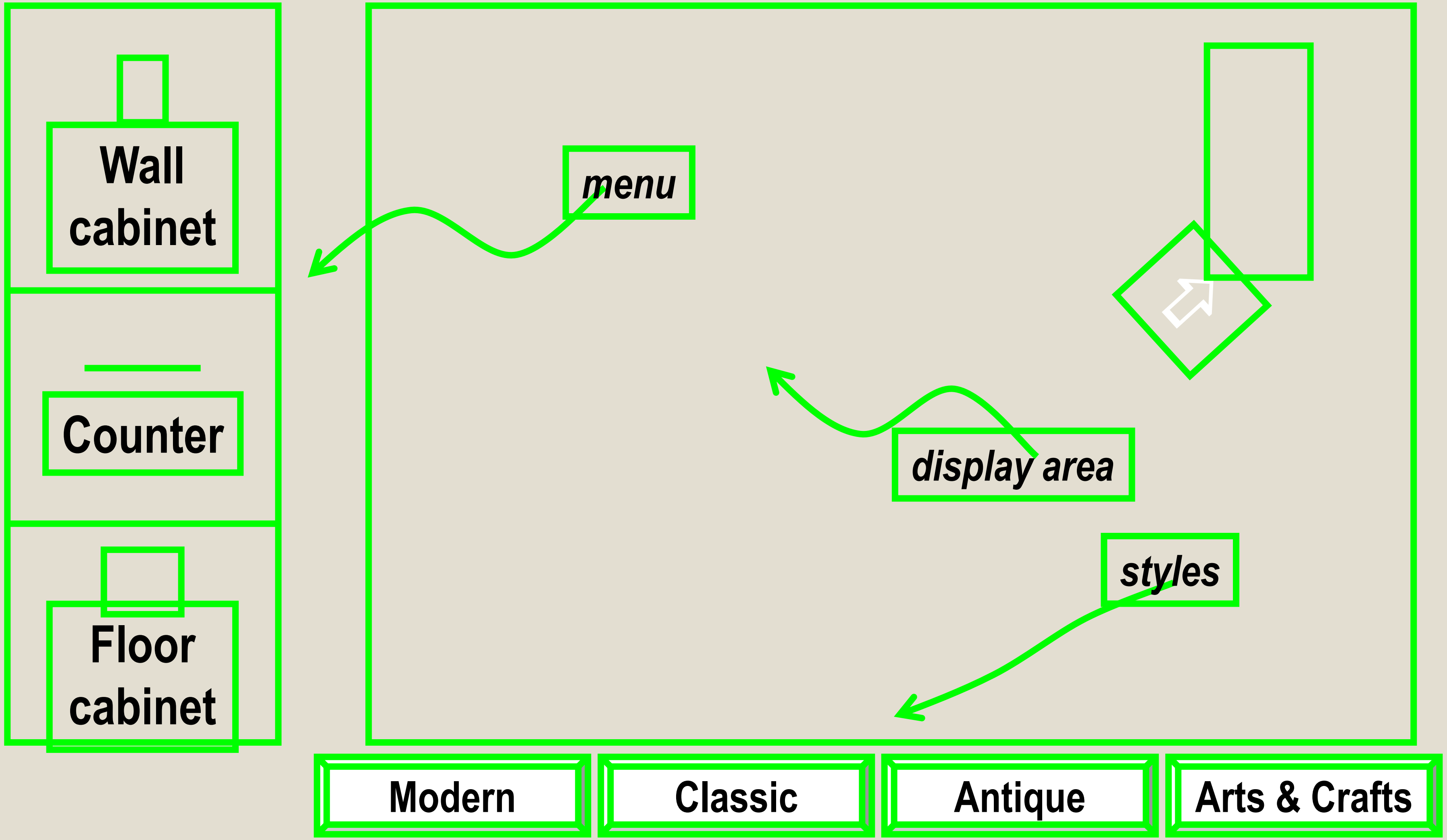
Topics covered

- What are design patterns
- Types of design patterns:
  - Creational, structural, and behavioral
- Characteristics of design patterns
  - Viewpoints, roles, and levels
- Design pattern forms
  - Delegation and recursion

Sample design goals and ways to accomplish them

- Reusability, Flexibility, and Maintainability
  - Reuse flexible designs
  - Keep code at a general level
  - Minimize dependency on other classes
- Robustness
  - Reuse reliable designs
  - Reuse robust parts
- Sufficiency / Correctness
  - Modularize design
  - Reuse trusted parts

KitchenViewer interface



Wall cabinet

Counter

Floor cabinet

menu

display area

styles

Modern

Classic

Antique

Arts & Crafts

Design pattern

Key Concept — Design pattern

Class combination and algorithm fulfilling a common design purpose.

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Best Practices

- Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development.
- Learning these patterns helps new developers to learn software design in an easy and faster way.

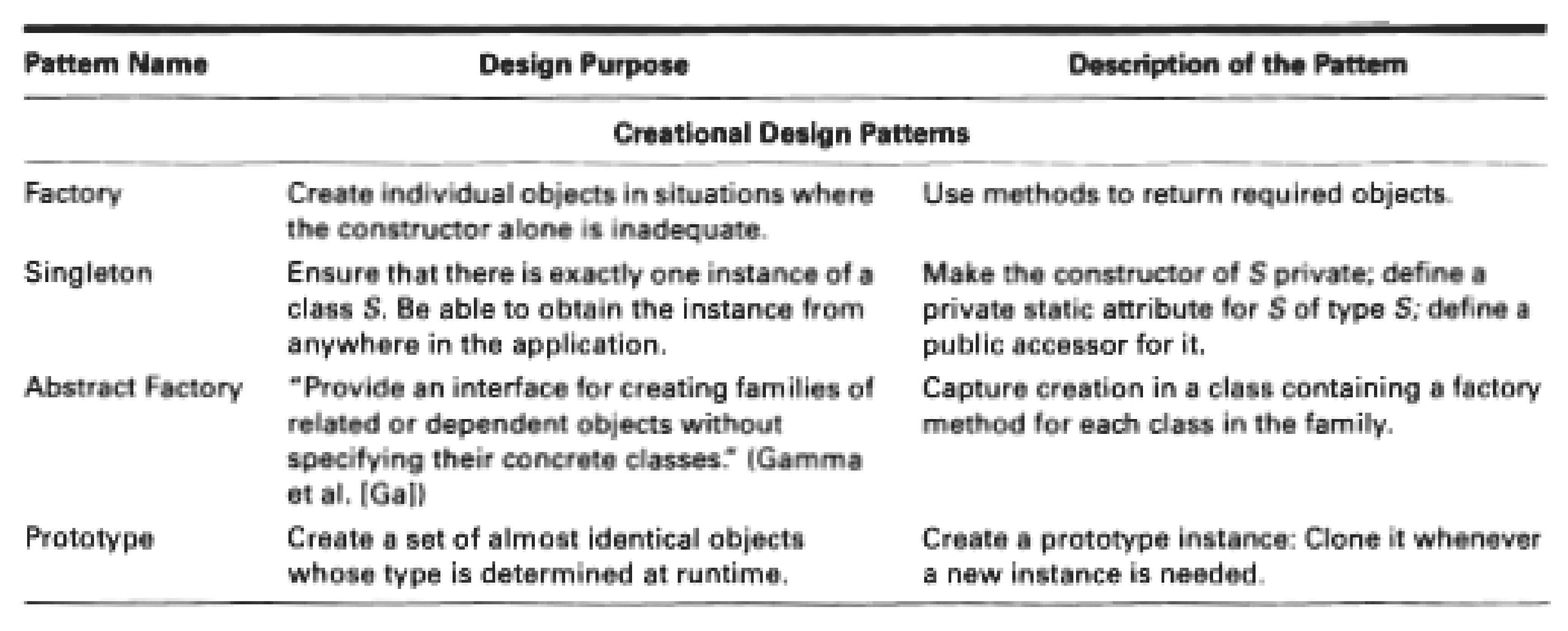
Summary of design patterns by type

- Creational patterns: creating a collection of objects in flexible ways.
  - Allow to create many versions of the collection at runtime.
  - Once object create, only instance of its class.
- Structural patterns: representing a collection of related objects.
  - To arrange collections of objects in form such as linked lists or tree.
- Behavioral patterns: capturing behavior among a collection of objects.
  - We want to be able to use classes separately in other application

Creational design patterns

Key Concept — Creational Design Patterns

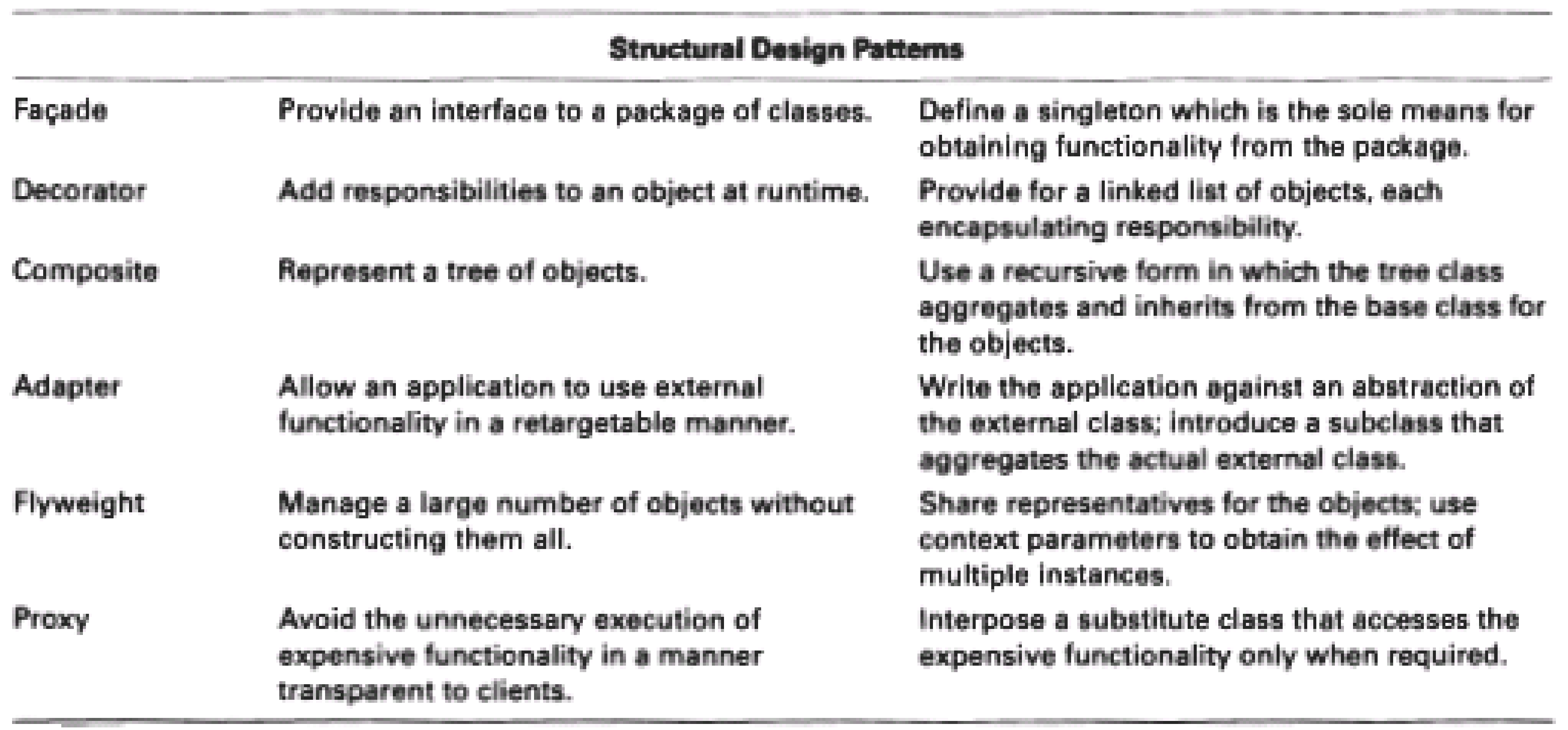
Used to create objects in flexible or constrained ways.



Pattern Name	Design Purpose	Description of the Pattern
	Creational Design Patterns	
Factory	Create individual objects in situations where the constructor alone is inadequate.	Use methods to return required objects.
Singleton	Ensure that there is exactly one instance of a class *S*. Be able to obtain the instance from anywhere in the application.	Make the constructor of *S* private; define a private static attribute for *S* of type *S*; define a public accessor for it.
Abstract Factory	"Provide an interface for creating families of related or dependent objects without specifying their concrete classes." (Gamma et al. [Ga])	Capture creation in a class containing a factory method for each class in the family.
Prototype	Create a set of almost identical objects whose type is determined at runtime.	Create a prototype instance: Clone it whenever a new instance is needed.

Key Concept — Structural Design Patterns

Used to represent data structures such as trees, with uniform processing interfaces.

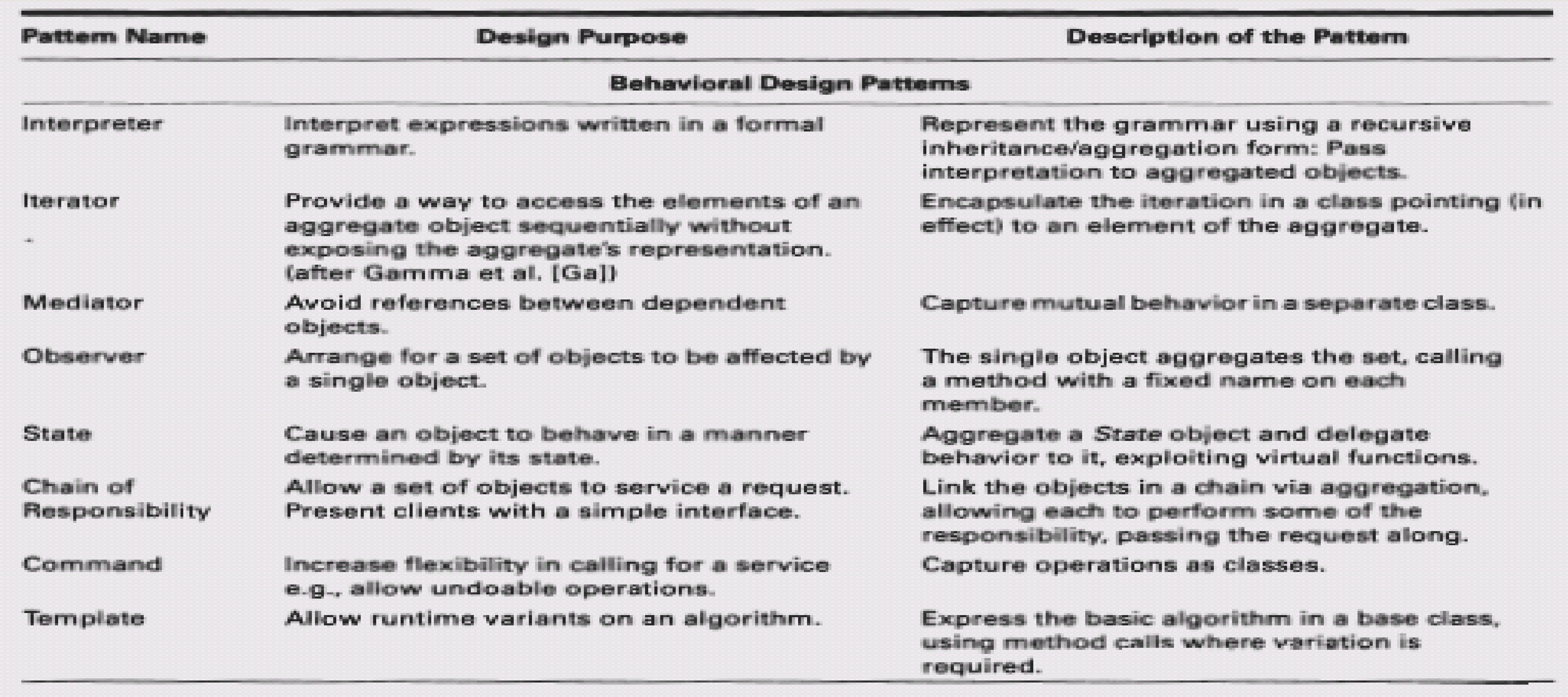


Structural Design Patterns

Façade	Provide an interface to a package of classes.	Define a singleton which is the sole means for obtaining functionality from the package.
Decorator	Add responsibilities to an object at runtime.	Provide for a linked list of objects, each encapsulating responsibility.
Composite	Represent a tree of objects.	Use a recursive form in which the tree class aggregates and inherits from the base class for the objects.
Adapter	Allow an application to use external functionality in a retargetable manner.	Write the application against an abstraction of the external class; introduce a subclass that aggregates the actual external class.
Flyweight	Manage a large number of objects without constructing them all.	Share representatives for the objects; use context parameters to obtain the effect of multiple instances.
Proxy	Avoid the unnecessary execution of expensive functionality in a manner transparent to clients.	Interpose a substitute class that accesses the expensive functionality only when required.

Key Concept — Behavioral Design Patterns

To capture behavior among objects.

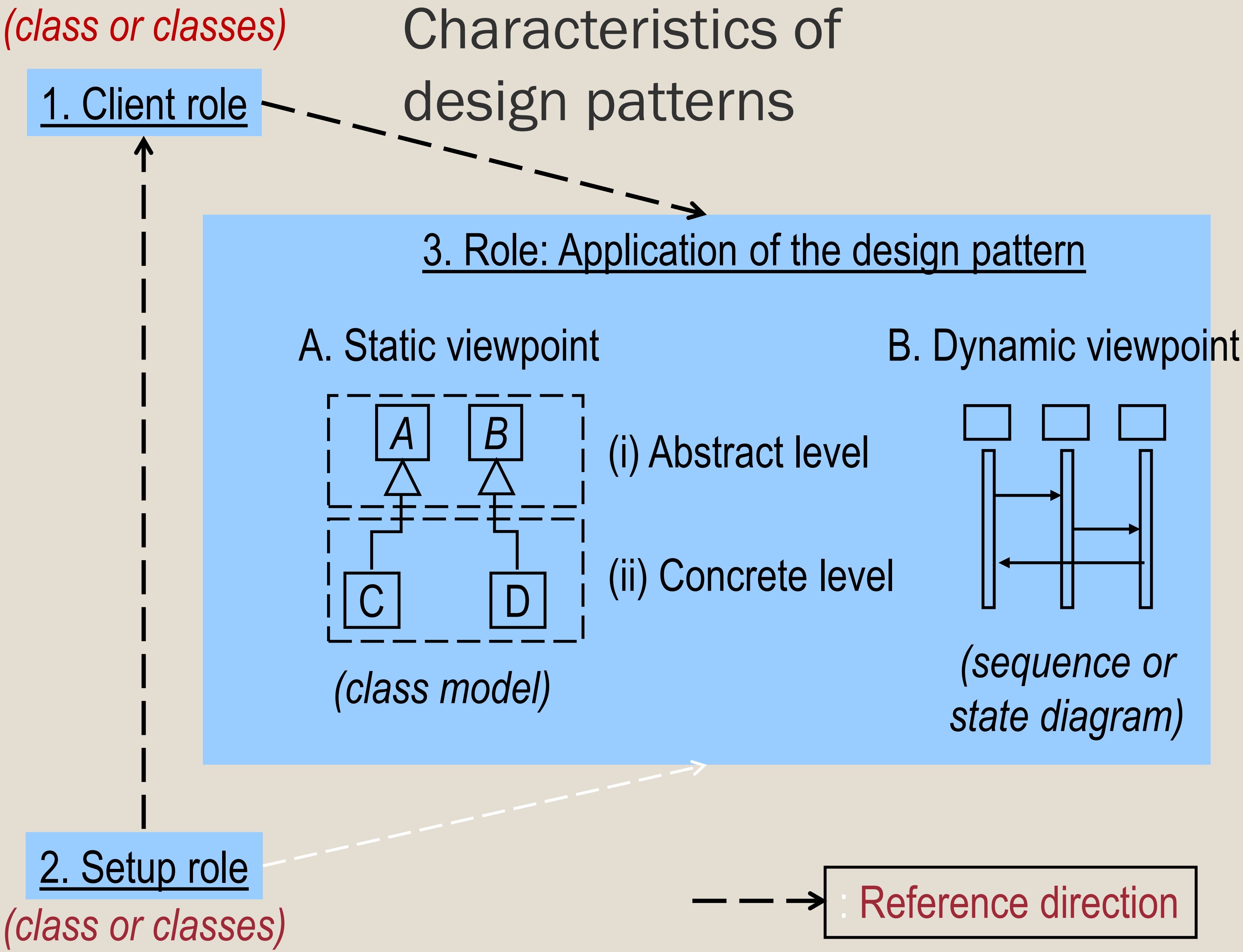


Pattern Name	Design Purpose	Description of the Pattern
Behavioral Design Patterns		
Interpreter	Interpret expressions written in a formal grammar.	Represent the grammar using a recursive inheritance/aggregation form: Pass interpretation to aggregated objects.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing the aggregate's representation. (after Gamma et al. [Ga])	Encapsulate the iteration in a class pointing (in effect) to an element of the aggregate.
Mediator	Avoid references between dependent objects.	Capture mutual behavior in a separate class.
Observer	Arrange for a set of objects to be affected by a single object.	The single object aggregates the set, calling a method with a fixed name on each member.
State	Cause an object to behave in a manner determined by its state.	Aggregate a *State* object and delegate behavior to it, exploiting virtual functions.
Chain of Responsibility	Allow a set of objects to service a request. Present clients with a simple interface.	Link the objects in a chain via aggregation, allowing each to perform some of the responsibility, passing the request along.
Command	Increase flexibility in calling for a service e.g., allow undoable operations.	Capture operations as classes.
Template	Allow runtime variants on an algorithm.	Express the basic algorithm in a base class, using method calls where variation is required.

Characteristics of design patterns

- Viewpoints – ways to describe patterns
  - *Static*: class model (building blocks)
  - *Dynamic*: sequence or state diagram (operation)
- Levels – decomposition of patterns
  - *Abstract* level describes the core of the pattern
  - *Concrete* (= non abstract) level describes the particulars of this case
- Roles – the "players" in pattern usage
  - *Application* of the design pattern itself
  - *Clients* of the design pattern application
  - *Setup* code initializes and controls

Characteristics of design patterns



(class or classes)

1. Client role

3. Role: Application of the design pattern

A. Static viewpoint

(i) Abstract level

(ii) Concrete level

(class model)

B. Dynamic viewpoint

(sequence or state diagram)

2. Setup role

(class or classes)

: Reference direction

Two viewpoints

Key Concept — Two Viewpoints

We consider design patterns from the static viewpoint (what they are made from) and the dynamic viewpoint (how they function).

- **Design Goal at Work**
  - Provide an interface to a design pattern
  - Functionality is clear and separate.
  - It good practice –code be more general classes
  - Make program more versatile

Correctness and two levels

Key Concept — Two Levels

Design patterns usually have an abstract level and a non-abstract (“concrete”) level.

Design Goal At Work — Correctness

We want to provide an interface to a design pattern so that its functionality is clear and separate.

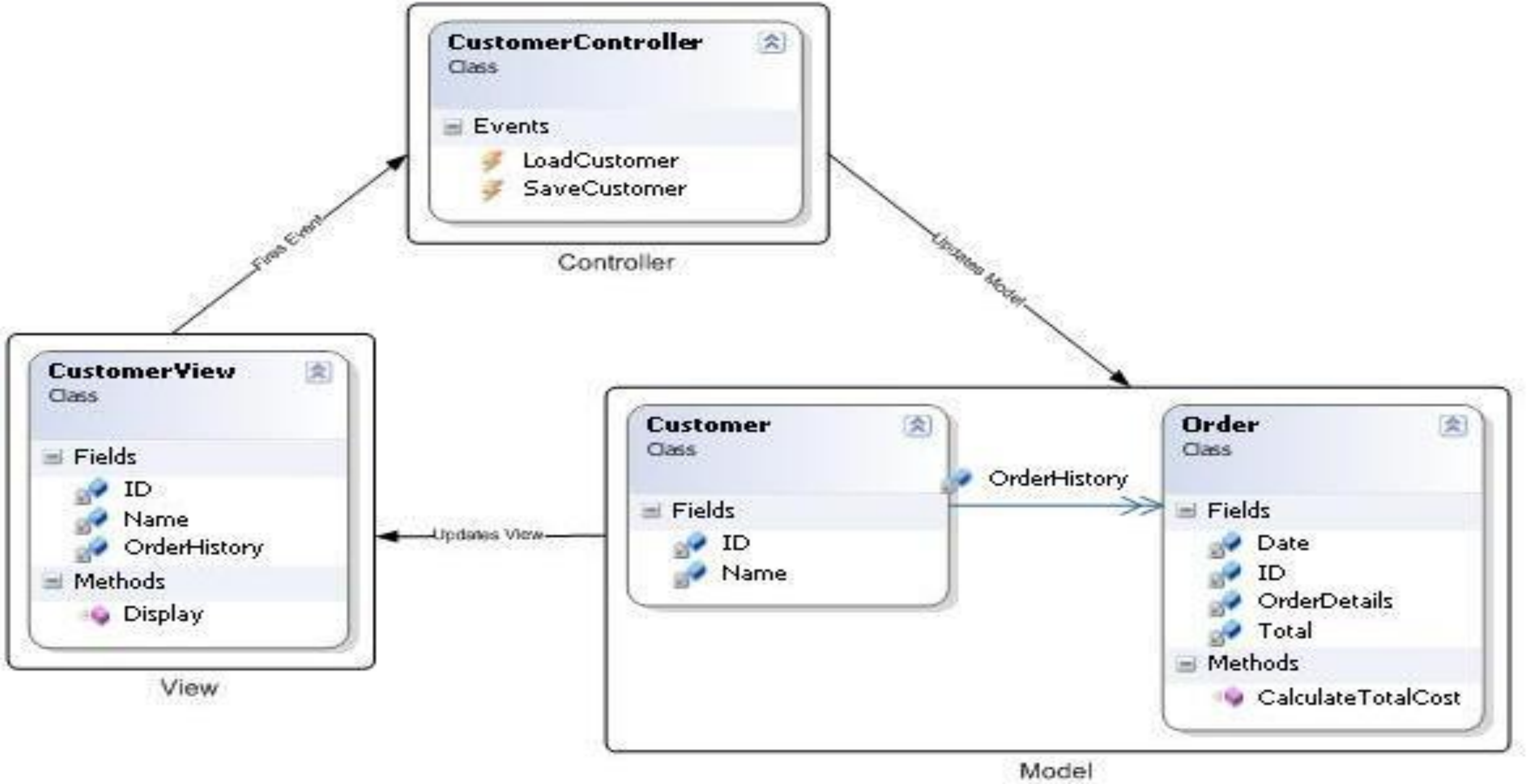
Three roles

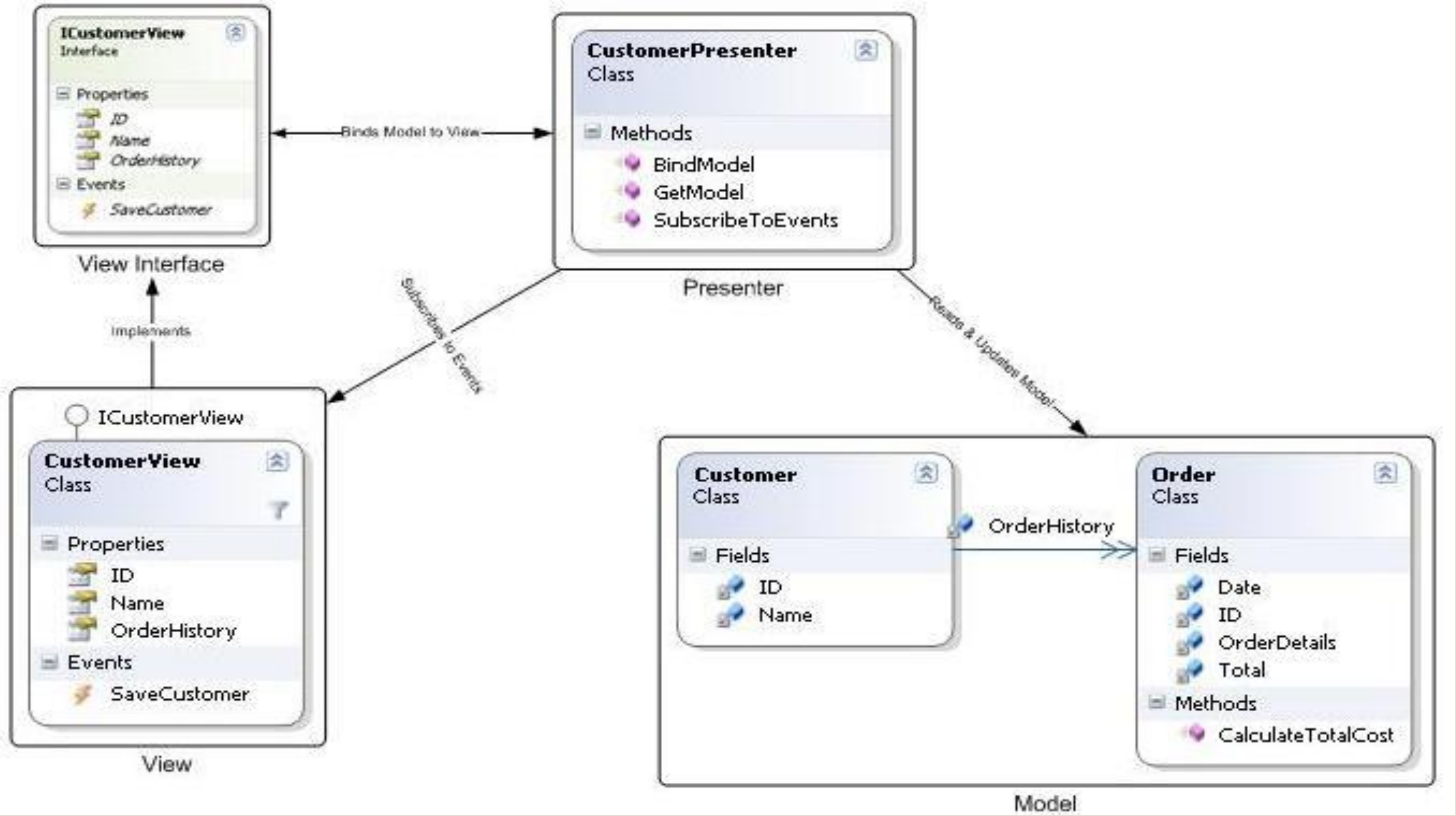
Design Goal At Work — Correctness

To use design patterns effectively, we distinguish the roles involved.

- Three roles involved in the usage of a design pattern with a design:
  - The application of the design pattern itself
  - The code that utilizes this application (the client role)
  - The code required, that utilizes or changes the design pattern application (the setup role)

Three roles -example




ICustomerView
Interface
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View Interface
Binds Model to View
CustomerPresenter
Class
Methods
BindModel
GetModel
SubscribeToEvents
Presenter
Implements
Subscribes to Events
Reads & Updates Model
ICustomerView
CustomerView
Class
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

Two forms

Key Concept — Two Forms

A design pattern’s form is usually a delegation of responsibility or a class that relates to itself (recursive).

Delegation Example

- Delegation is like **assigning a task to someone else** without doing it yourself.
- **Scenario:**
- You are the manager of a company, and you assign tasks to employees.
- **Explanation:**
- You delegate the task to another employee.
- Once they finish, they report back to you.
- You are not directly doing the task; instead, you are **passing it on**.

Example Code in Java

```
class Manager {
    void assignTask() {
        Employee employee = new Employee();
        employee.doTask(); // Delegating the task to Employee
    }
}

class Employee {
    void doTask() {
        System.out.println("Employee is completing the task.");
    }
}

public class DelegationExample {
    public static void main(String[] args) {
        Manager manager = new Manager();
        manager.assignTask(); // Manager delegates the task
    }
}
```

Output

- Employee is completing the task.

Delegation Design Pattern Form

- To achieve flexibility, the kitchen viewer design replaces direct code such as :

```
new AntiqueWallCabinet();//     applies only to antique style
```

- Version delegates construction to an intermediary method.

```
mySTyle.getWallCabinet();//   applies to    whatever style
is chosen at runtime
```

- Delegation is implemented using the **virtual function** property.

Recursion Design Pattern Form

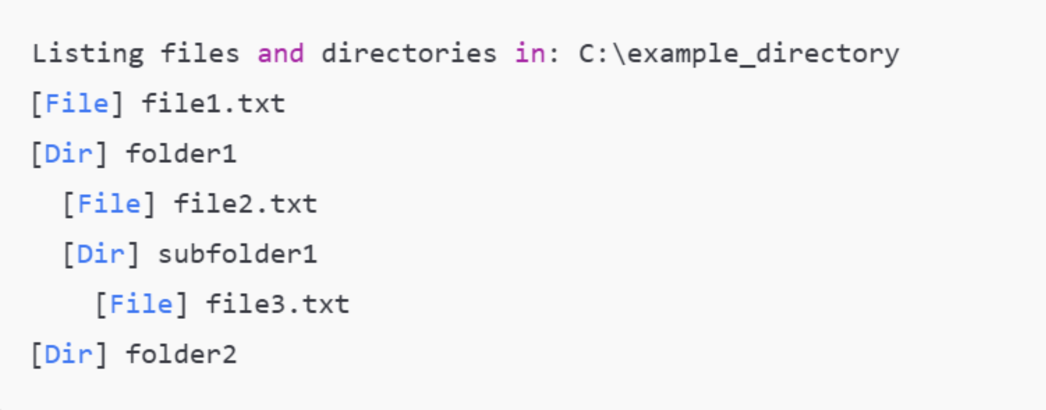
- Pattern essentially uses itself.
- Useful representing a linked list of objects in which each object of a class aggregates another object of the same class.
- Example : GUI allow for windows within windows within windows…
  - Windows object aggregates itself.

Recursive Example

- Recursion is like **doing the same task repeatedly but breaking it into smaller parts**.
- **Scenario:**
- You have a stack of dishes, and you need to wash all of them. You pick up one dish, wash it, and repeat until the stack is empty.
- **Explanation:**
- You are solving the problem (washing dishes) **one step at a time**.
- You call yourself (recursively) to handle the rest of the problem (remaining dishes).

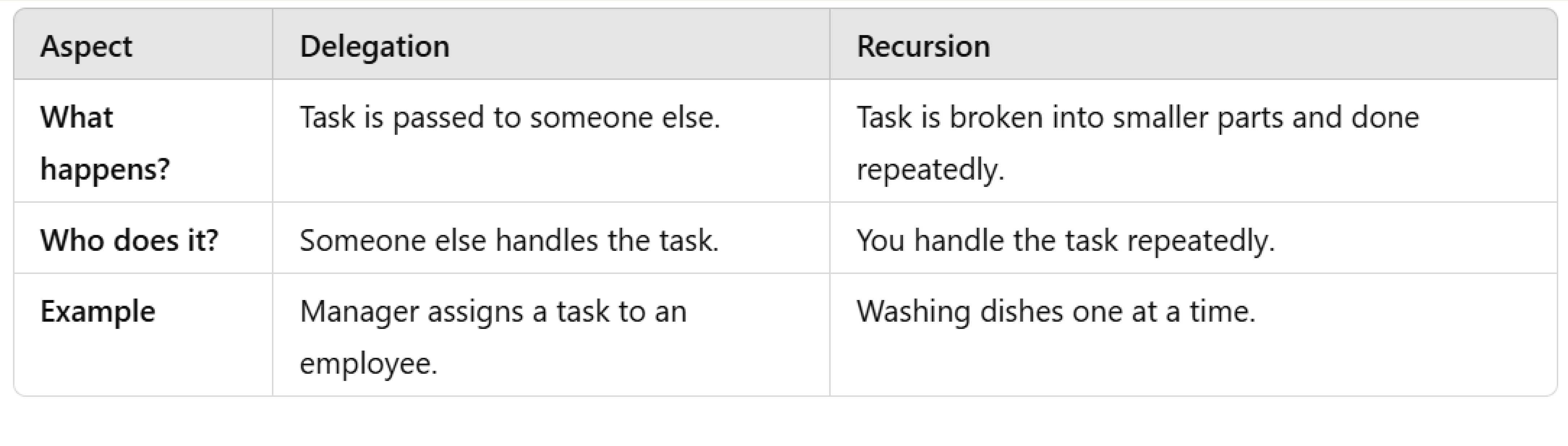
```java
import java.io.File;

public class RecursiveFileExplorer {

    public static void main(String[] args) {
        // Specify the starting directory (can be user input as well)
        String startDirectory = "C:\\example_directory"; // Replace with your directory path

        System.out.println("Listing files and directories in: " + startDirectory);
        listFilesAndDirectories(startDirectory, 0); // Start recursion
    }

    // Recursive method to list files and directories
    public static void listFilesAndDirectories(String path, int level) {
        File directory = new File(path);

        // Check if the path is a directory
        if (directory.isDirectory()) {
            // Get all files and subdirectories in the directory
            File[] files = directory.listFiles();

            if (files != null) { // Check to avoid NullPointerException
                for (File file : files) {
                    // Print with indentation based on the depth (level) of recursion
                    System.out.println("  ".repeat(level) + (file.isDirectory() ? "[Dir] " : "[File] ") + file.getName());

                    // If the file is a directory, recursively list its contents
                    if (file.isDirectory()) {
                        listFilesAndDirectories(file.getAbsolutePath(), level + 1);
                    }
                }
            }
        } else {
            System.out.println("The path provided is not a directory.");
        }
    } }
```

- Output



```
Listing files and directories in: C:\example_directory
[File] file1.txt
[Dir] folder1
  [File] file2.txt
  [Dir] subfolder1
    [File] file3.txt
[Dir] folder2
```

Key difference



Aspect	Delegation	Recursion
What happens?	Task is passed to someone else.	Task is broken into smaller parts and done repeatedly.
Who does it?	Someone else handles the task.	You handle the task repeatedly.
Example	Manager assigns a task to an employee.	Washing dishes one at a time.

Summary

- Design Patterns are recurring designs satisfying recurring design purposes
- Described by Static and Dynamic Viewpoints
  - Typically class models and sequence diagrams respectively
- Use of a MVC pattern application is a Client Role
  - Client interface carefully controlled
  - “Setup,” typically initialization, a separate role
- Design patterns Forms usually Delegation or Recursion
- Classified as Creational, Structural, or Behavioral

Thank you

Please send all questions to:
robiah@unikl.edu.my