IOE 321 Software Design Patterns

Chapter I

Introduction to Design Patterns

Presented by

Dr. Koppala Guravaiah

Assistant Professor

IIIT Kottayam

Syllabus

Design Pattern

- Introduction
- Architectural Design Patterns will be discussed in next chapter
- Describing Design Patterns
- The Catalog of Design patterns
- How Design patterns solve Design problems
- How to select a Design Pattern
- How to use a Design Pattern

Text Book:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns: Elements of Reusable Object oriented Software Addison-Wesley

Introduction

- Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder
- Objects → Classes → interfaces & inheritance → relationships
- Design should be specific to the problem at hand but also general enough to address future problems and requirements
- Avoid redesign, or at least minimize
- Object-oriented designers follow patterns like "represent states with objects" and "decorate objects so you can easily add/remove features."

Introduction … Contd.

- The purpose of subject is to record experience in designing object-oriented software as design patterns.
- Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems

Design Pattern

Christopher Alexander says,

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such away that you can use this solution a million times over, without ever doing it the same way twice“

The pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.

Design Pattern … Contd.

Advantages of Design Patterns

- Design patterns make it easier to reuse successful designs and architectures.
- Expressing proven techniques as design patterns makes them more accessible to developers of new systems.
- Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.

Design Pattern … Contd.

Advantages of Design Patterns

- Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.
- Put simply, design patterns help a designer get a design "right" faster.

Design Patterns … Elements

Four Essential Elements:

- Pattern Name
- Problem: When to apply a pattern
- Solution: Elements that make up the design, their relationship, responsibilities, and collaborations
- Consequences: are the results and trade-offs of applying the pattern

Design Patterns … Contd.

- The design patterns are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*
- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.
- The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

Describing Design Patterns

- How do we describe design patterns?
- Describe design patterns using a consistent format, template lends a uniform structure to the information, making design patterns easier to learn, compare, and use
- **Pattern Name and Classification**
- **Intent**
- **Also Known As**
- **Motivation**
- **Applicability**
- **Structure**
- **Participants**
- **Collaborations**
- **Consequences**
- **Implementation**
- **Sample Code**
- **Known Uses**
- **Related Patterns**

Catalog of Design Patterns

23 Design patterns

- Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Adapter: Convert the interface of a class into another interface clients expect.
- Bridge: Decouple an abstraction from its implementation so that the two can vary independently.
- Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Catalog of Design Patterns …Contd.

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

Catalog of Design Patterns …Contd.

- Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
- Facade: Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a cl ass defer instantiation to subclasses.

Catalog of Design Patterns …Contd.

- Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.
- 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, promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Catalog of Design Patterns …Contd.

- Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object ca n be restored to this state later.
- Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Proxy: Provide a surrogate or placeholder for another object to control access to it.

Catalog of Design Patterns …Contd.

- Singleton: Ensure a class only has one instance, and provide a global point of access to it.
- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Catalog of Design Patterns …Contd.

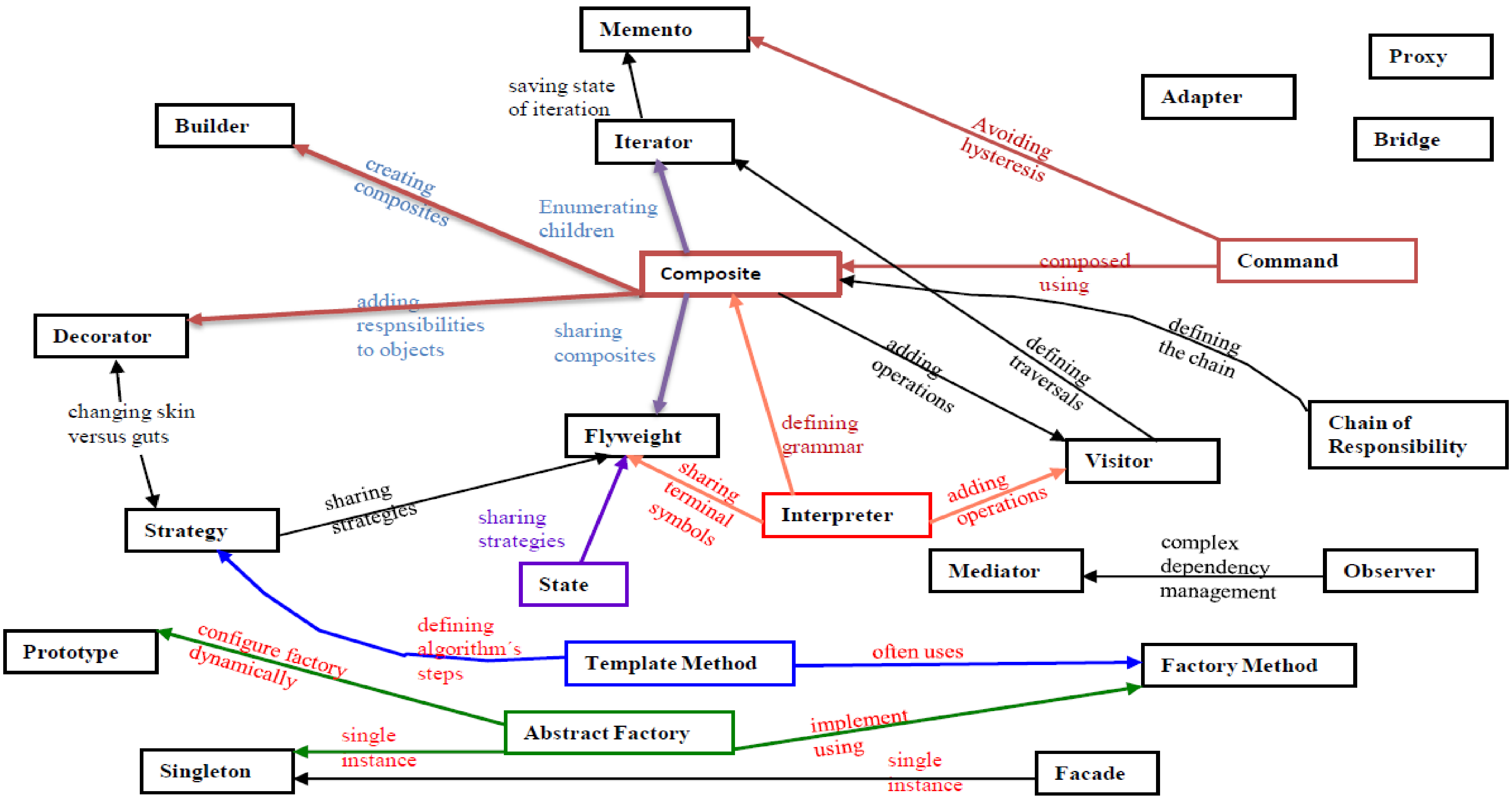
- Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Organizing the Catalog

Indian Institute of Information Technology Kottayam

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

Relationship between Patterns



How Design Patterns Solve design Problems

- Finding Appropriate Objects
- Determining Object Granularity
- Specifying Object Interface
- Specifying Object Implementations
  - Class versus Interface Inheritance
  - Programming to an Interface, not an Implementation
- Putting Reuse Mechanisms to Work
  - Inheritance versus Composition
  - Delegation
  - Inheritance versus Parameterized Types

How Design Patterns Solve design Problems … Contd.

- Relating Run-Time and Compile-Time Structures
- Designing for Change
  - Application Programs
  - Toolkits
  - Frameworks

Inheritance versus Composition

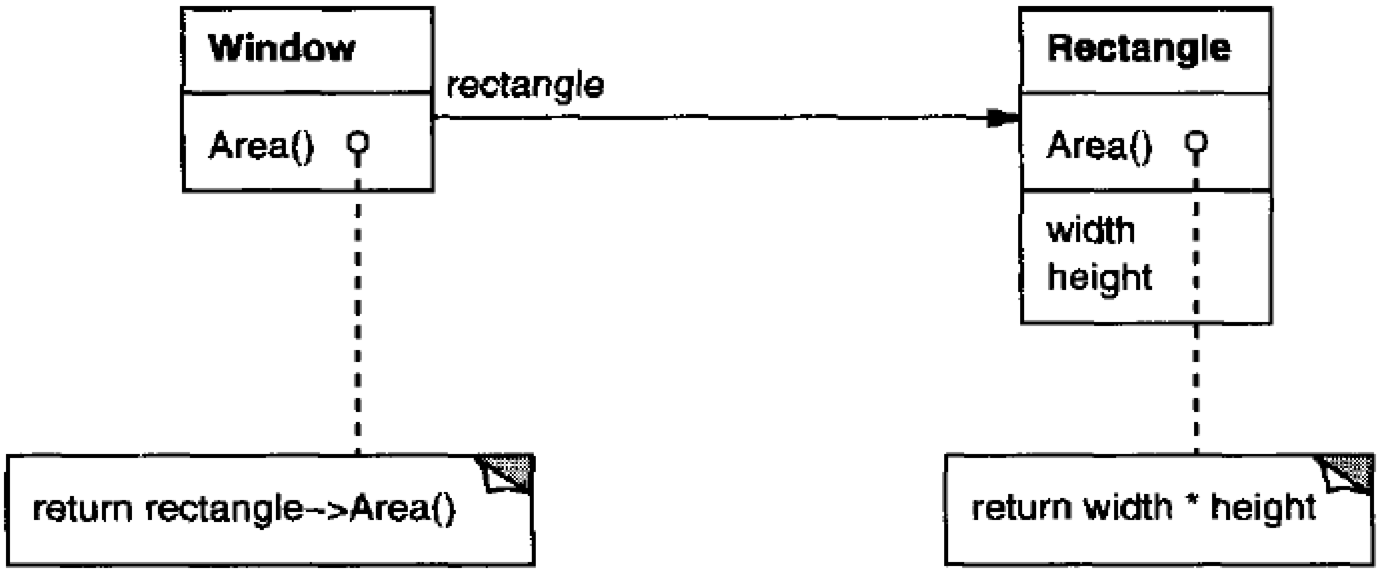
- Two most common techniques for reuse
  - class inheritance - white-box reuse
  - object composition - black-box reuse
- Class inheritance - Advantages
  - Static, straight forward to use
  - Make the implementations being reuse more easily
- Disadvantages
  - The implementations inherited can‘t be changed at run time
  - Parent classes often define at least part of their subclasses‘ physical representation
    - breaks encapsulation
  - Implementation dependencies can cause problems when you‘re trying to reuse a subclass

Inheritance versus Composition … Contd.

- Object composition
  - Dynamic at run time
  - composition requires objects to respect each others' interfaces
    - but does not break encapsulation
  - any object can be replaced at run time
  - Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task
  - class and class hierarchies will remain small
  - but will have more objects
- ***Favor object composition over class inheritance***
- Inheritance and object composition should work together

Delegation

- Two objects are involved in handling a request: a receiving object delegates operations to its ***delegate***
- Makes it easy to compose behaviors at run-time and to change the way they're composed



Delegation … Contd.

- Disadvantage: dynamic, highly parameterized software is harder to understand than more static software
- Delegation is a good design choice only when it simplifies more than it complicates
- Delegation is an extreme example of object composition

Inheritance versus Parameterized Types

- The unspecified types are supplied as parameters at the point of use
  - Parameterized types, generics, or templates
- Parameterized types give us a third way to compose behavior in object-oriented systems
- Three ways to compose
  - ***object composition*** lets you change the behavior being composed at run-time, but it requires indirection and can be less efficient
  - ***Inheritance*** lets you provide default implementations for operations and lets subclasses override them
  - ***parameterized types*** let you change the types that a class can use

Relating Run-Time and Compile-Time Structures

- An object-oriented program‘s ***run-time structure*** often bears little resemblance to its ***code structure***
- The code structure is frozen at compile-time
- A program‘s run-time structure consists of rapidly changing networks of communicating objects
- ***Aggregation*** versus ***acquaintance*** (***association***)
  - *Part-of* versus *knows of*
- The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms
- The system‘s run-time structure must be imposed more by the designer than the language

Designing for Change

- Here are some common causes of redesign along with the design pattern(s) that address them:
  - *Creating an object by specifying a class explicitly.*
  - *Dependence on specific operations.*
  - *Dependence on hardware and software platform.*
  - *Dependence on object representations or implementations*
  - *Algorithmic dependencies.*
  - *Tight coupling*
  - *Extending functionality by subclassing*
  - *Inability to alter classes conveniently*
  - Application Programs
  - Toolkits
  - Frameworks

Frameworks

- A framework is a set of cooperating classes that makeup a reusable design for a specific class of software
- Because patterns and frameworks have some similarities, people often wonder how or even if they differ.
- They are different in three major ways:
  - *Design patterns are more abstract than frameworks.*
  - *Design patterns are smaller architectural elements than frameworks.*
  - *Design patterns are less specialized than frameworks.*

How to Select a Design Pattern

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

Design aspects that design patterns let you vary

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

Design aspects that design patterns let you vary … Contd.

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

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

Thank you