Module 3 (Part I)

Design Patterns

Index

❖ Basic concepts of Design patterns

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design
- A design pattern isn’t a finished design
- It is a description or template for how to solve a problem that can be used in many different situations
- Patterns can be viewed as helping designers to make certain important design decisions.
- At a basic level, patterns can also be viewed as well-documented and well thought-out building blocks for software design.

- **“Use a pattern million times over, without ever doing it the same way twice” by Christopher Alexander**
- Each design pattern systematically names, explains and evaluates an important and recurring design in object-oriented systems
- Our goal is to capture design experience in a form that people can use effectively
- To this end GoF Design Pattern documented some of the most important design patterns and present them as a catalog

- **GoF Design Patterns are divided into three categories:**
- **Creational:** The design patterns that deal with the creation of an object.
- **Structural:** The design patterns in this category deals with the class structure such as Inheritance and Composition.
- **Behavioral:** This type of design patterns provide solution for the better interaction between objects, how to provide loose coupling, and flexibility to extend easily in future.

- In general, a pattern has four essential elements:
  1) **Pattern name**
  2) **Problem**
  3) **Solution**
  4) **Consequences**

- **Pattern name**
  - Use to describe a design problem, its solutions, and consequences in a word or two
  - Naming a pattern immediately increases our design vocabulary
  - It makes it easier to think about designs and communicate to others
  - Finding good names has been one of the hardest part

- **Problem**
  - Describes when to apply the pattern
  - It explains the problem and its context
- **Solution**
  - 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
  - Describes the elements that make up the design, their relationships, responsibilities, and collaborations
- **Consequences**
  - They are the results and trade-offs of applying the pattern

How to select a design pattern

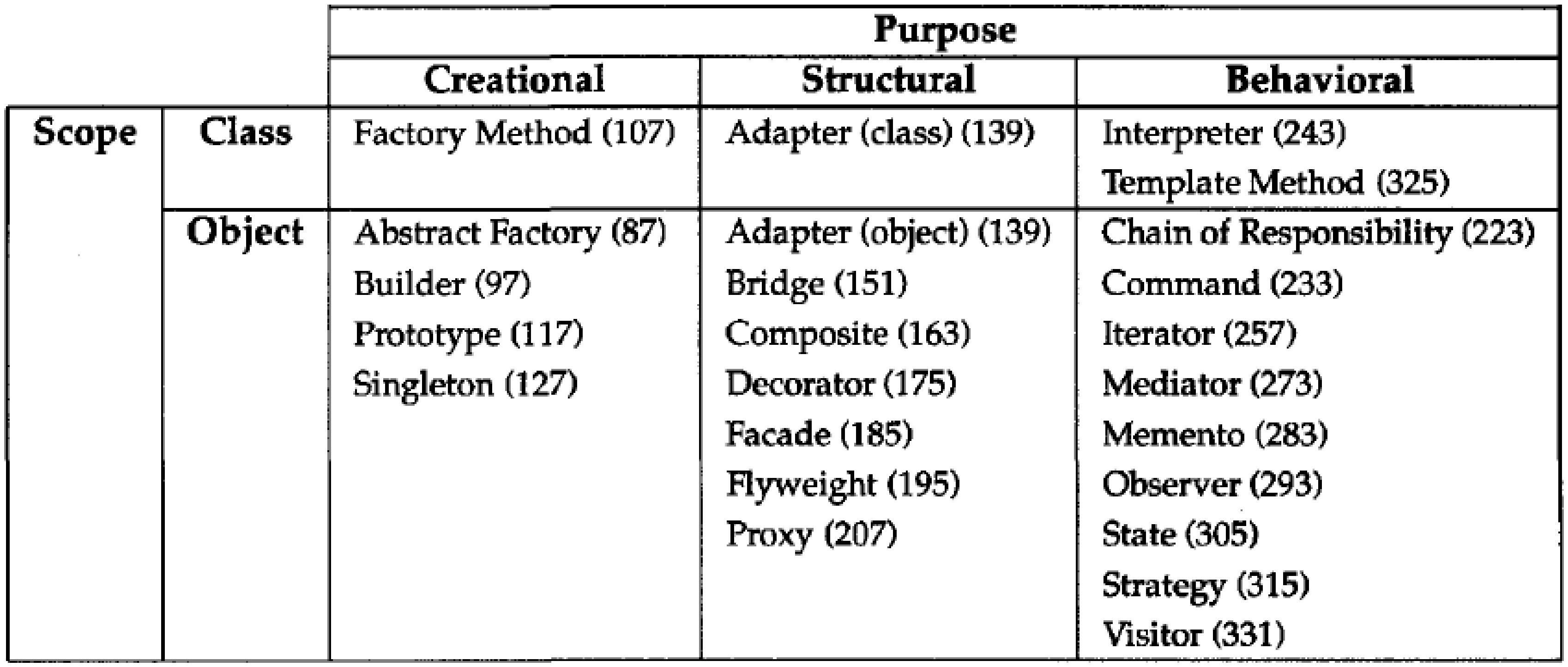
- With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you
- Here are several different approaches to finding the design pattern that's right for your problem:

- **Consider how design patterns solve design problems**
  - Find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems
- **Scan Intent sections**
  - Read through each pattern's intent to find one or more that sound relevant to your problem
- **Study how patterns inter-relate**
  - Studying these relationships between design patterns graphically can help direct you to the right pattern or group of patterns

- **Study patterns of like purpose**
  - Study the similarities and differences between creational patterns, structural patterns and behavioral patterns
- **Examine a cause of redesign**
  - Look at the patterns that help you avoid the causes of redesign
  - Eg : **Algorithmic dependencies**
    - Algorithms are often extended, optimized, and replaced during development and reuse
    - Objects that depend on an algorithm will have to change when the algorithm changes
    - Therefore algorithms that are likely to change should be isolated
    - Design patterns: Builder, Iterator, Strategy, Template, Method, Visitor

❖ Organizing the Catalog

- There are many design patterns, we need a way to organize them



		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Table 1.1: Design pattern space

Creational patterns are ones that create objects, rather than having to instantiate objects directly. This gives the program more flexibility in deciding which objects need to be created for a given case.

This pattern can be further divided into class-creation patterns and object-creational patterns

While class-creation patterns use inheritance effectively in the instantiation process

Object-creation patterns use delegation effectively to get the job done

- Abstract factory groups object factories that have a common theme.
- Builder constructs complex objects by separating construction and representation.
- Factory method creates objects without specifying the exact class to create.
- Prototype creates objects by cloning an existing object.
- Singleton restricts object creation for a class to only one instance.

❖ Factory Method

- **Class Creational**

Intent

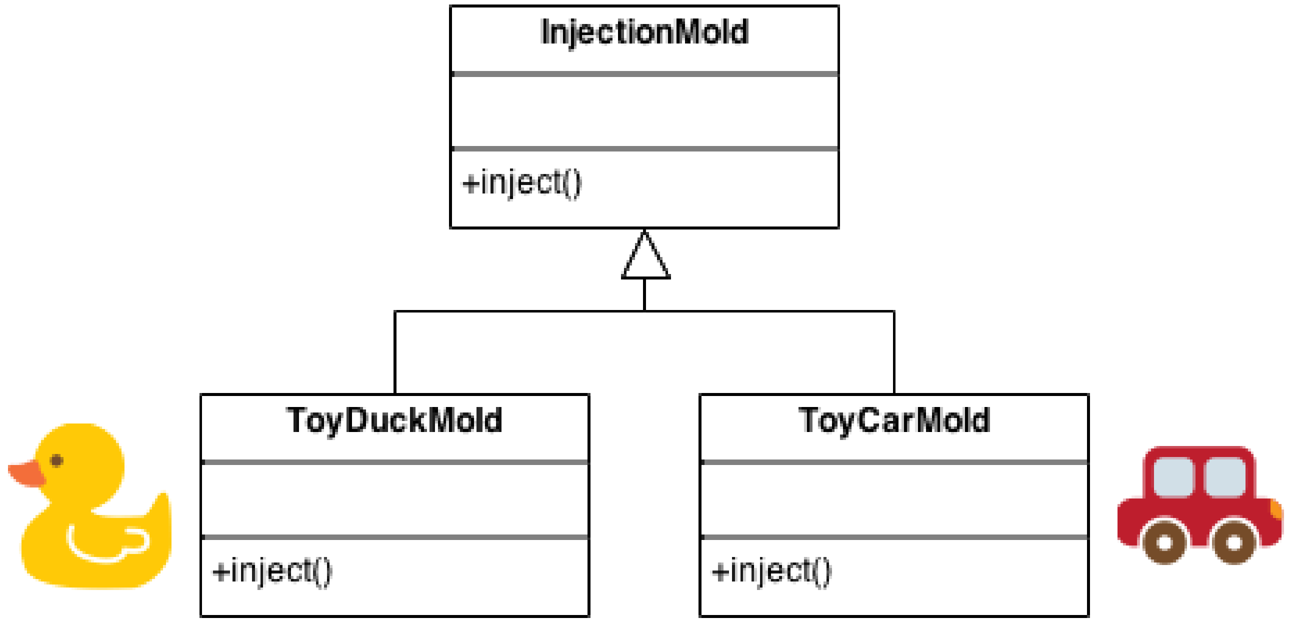
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As

Virtual Constructor

Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

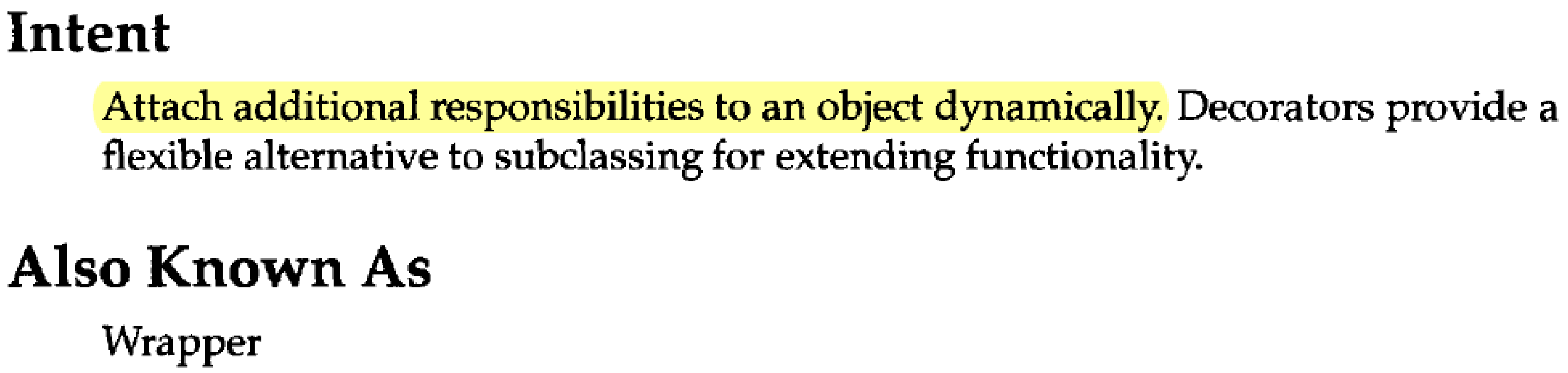


Structural patterns concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

- Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- Bridge decouples an abstraction from its implementation so that the two can vary independently.
- Composite composes zero-or-more similar objects so that they can be manipulated as one object.
- Decorator dynamically adds/overrides behavior in an existing method of an object.
- Facade provides a simplified interface to a large body of code.
- Flyweight reduces the cost of creating and manipulating a large number of similar objects.
- Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

❖ Decorator

- **Object Structural**

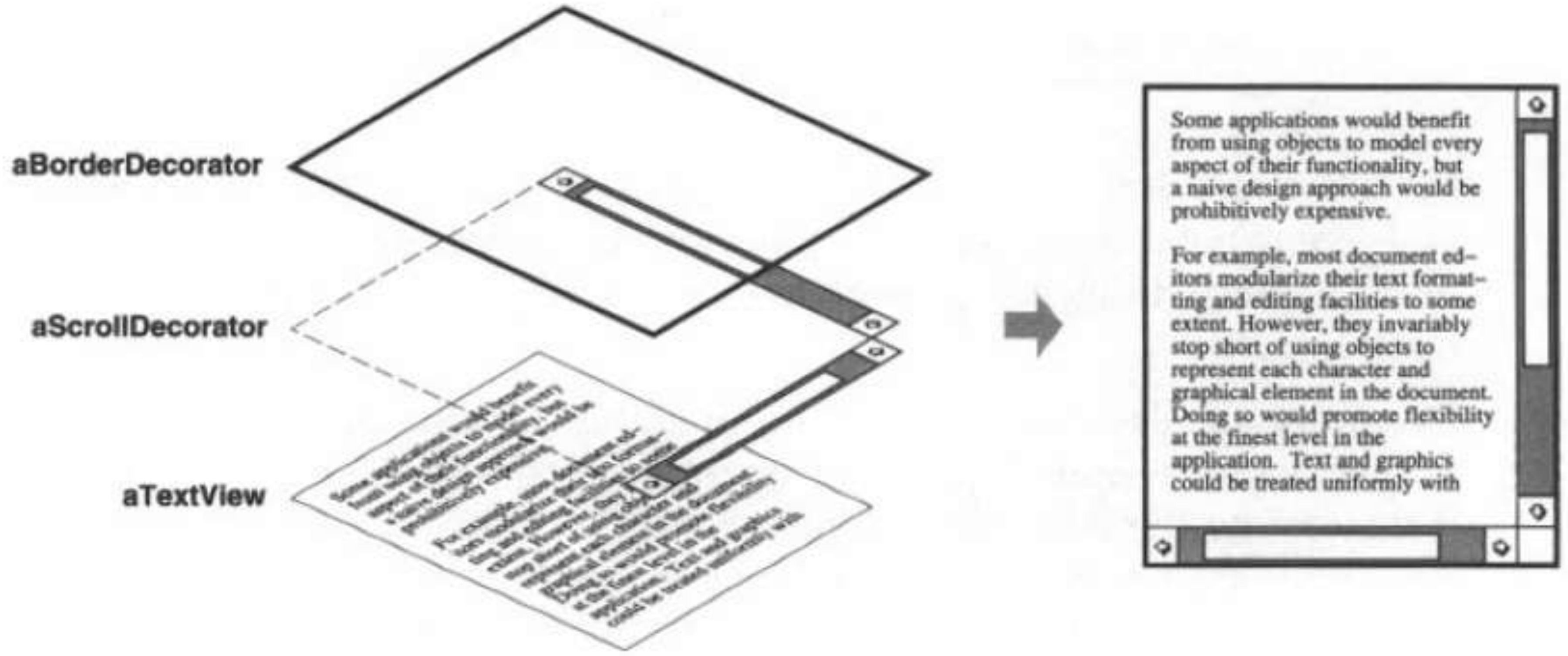


Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

Wrapper

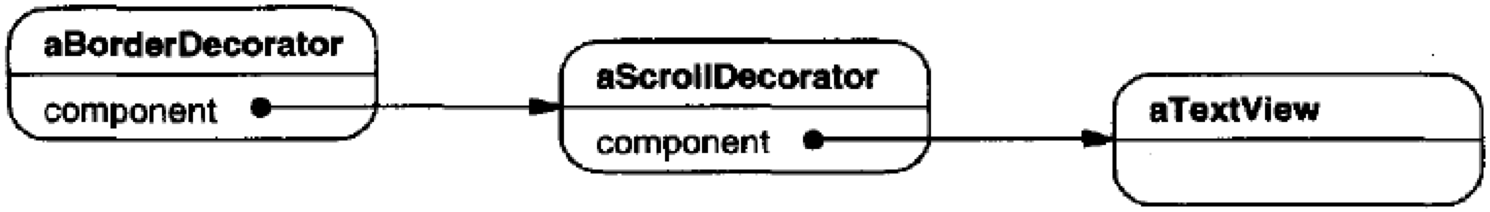


Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

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

For example, suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well. We simply compose the decorators with the TextView to produce the desired result.

The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Behavioral Most behavioral design patterns are specifically concerned with communication between objects.

- Chain of responsibility delegates commands to a chain of processing objects.
- Command creates objects that encapsulate actions and parameters.
- Interpreter implements a specialized language.
- Iterator accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento provides the ability to restore an object to its previous state (undo).
- Observer is a publish/subscribe pattern, which allows a number of observer objects to see an event.
- State allows an object to alter its behavior when its internal state changes.
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

❖ Chain of Responsibility

- **Object Behavioral**

Intent

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.

Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.

100
50
20
5
ATM

- The following are the main pros strengths of design patterns:
  - Design patterns provide a common vocabulary that helps to improve communication among the developers.
  - Design patterns help to capture and disseminate expert knowledge.
  - Use of design patterns help designers to produce designs that are flexible, efficient, and easily maintainable.
  - Design patterns guide developers to arrive at correct design decisions and help them to improve the quality of their designs.
  - Design patterns reduce the number of design iterations, and help improve the designer productivity.

- Important cons shortcomings of design patterns are the following:
  - ➢Design patterns do not directly lead to code reuse. Since a design pattern is tailored for a specific circumstance of reuse, and therefore it is difficult to associate a fixed code segment with a pattern.
  - ➢Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem
  - ➢At present no methodology is available that can be used to select the right design pattern at the right point during a design exercise.

❖ Antipattern

- If a pattern represents a best practice, then an antipattern represents lessons learned from a bad design.
- The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context
- The following are two types of antipatterns that are popular:
  - Those that describe bad solutions to problems, thereby leading to bad situations.
  - Those that describe how to avoid bad solutions to problems.

AntiPattern Examples

❖ Antipattern

- We mention here only a few interesting antipatterns without discussing them in detail.
- **Input kludge:** This concerns failing to specify and implement a mechanism for handling invalid inputs.
- **Magic pushbutton:** This concerns coding implementation logic directly within the code of the user interface, rather than performing them in separate classes.
- **Race hazard:** This concerns failing to see the consequences of all the different ordering of events that might take place in practice.
- A key goal of development AntiPatterns is to describe useful forms of **software refactoring**. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance