Chapter 6
Introduction to Design Patterns

3 Types:
Creational
 Structural
 Behavioral
Process Phase Discussed in This Chapter

Key: ⨉ = main emphasis │ = secondary emphasis

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
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
1. User clicks on the “wall cabinet” icon
2. Application displays a wall cabinet in center of the work area
3. User resizes the wall cabinet
4. User drags the wall cabinet in the upper half of the work area
5. User releases cursor
6. Application places the wall cabinet in a position in the upper half of the work area
7. User clicks on the “floor cabinet” icon
8. Application displays a floor cabinet in the center of the work area.

............
**KitchenViewer Example**

Wall cabinets

Floor cabinets

Countertop

Modern

Classic

Antique

Arts & Crafts

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

Modern  Classic  Antique  Arts & Crafts
KitchenViewer Without Design Patterns

Client
renderKitchen()

Kitchen

WallCabinet

ModernWallCabinet
AntiqueWallCabinet

FloorCabinet

ModernFloorCabinet
AntiqueFloorCabinet

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

• Design patterns are class combinations and accompanying algorithms that fulfill common design purposes.

• A design pattern expresses an idea rather than a fixed class combination.
Design Goal At Work: ➔ Flexibility ➚

Our design should be flexible enough to produce any of several kitchen styles.
The Abstract Factory Idea

KitchenStyle
getWallCabinet()
getFloorCabinet()

WallCabinet

AntiqueWallCabinet

FloorCabinet

ModernKStyle
getWallCabinet()
getFloorCabinet()

AntiqueKStyle
getWallCabinet()
getFloorCabinet()

FloorCabinet getFloorCabinet()
{ return new ModernFloorCabinet(); }

FloorCabinet getFloorCabinet()
{ return new AntiqueFloorCabinet(); }
Abstract Factory Design Pattern Applied to KitchenViewer

```
Client
renderKitchen(KitchenStyle)

Kitchen
getWallCabinet()
getFloorCabinet()

WallCabinet
FloorCabinet

ModernWallCabinet
AntiqueWallCabinet

ModernFloorCabinet
AntiqueFloorCabinet

ModernKStyle
getWallCabinet()
getFloorCabinet()

AntiqueKStyle
getWallCabinet()
getFloorCabinet()
```

“reflecting polymorphism”
Abstract Factory Design Pattern

Client
```
doOperation( Style myStyle )
```
Abstract Factory Design Pattern Alternative

Style
getComponentA()
GetComponentB()

Collection
getComponentA()
GetComponentB()

Client
doOperation()

doComponentA()
doComponentB()

does not reference style directly
doOperation() takes no parameters
Collection has methods for getting the various components

Style1
getComponentA()
GetComponentB()

Style1 ComponentA

Style2
getComponentA()
GetComponentB()

Style2 ComponentA

Style1 ComponentB

Style2 ComponentB

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

-- class combination and algorithm fulfilling a common design purpose.

*Key Concept:* $\rightarrow$ Creational Design Patterns $\leftarrow$
-- used to create objects in flexible or constrained ways.

*Key Concept:* $\rightarrow$ Structural Design Patterns $\leftarrow$
-- used to represent data structures such as trees, with uniform processing interfaces.

*Key Concept:* $\rightarrow$ Behavioral Design Patterns $\leftarrow$
-- to capture behavior among objects.
Creational Design Patterns

• Creational design patterns help us to design applications involving collections of objects:
  – They allow the creation of several possible creations from a single block of code such as:
    • Creating many versions of the collection at runtime
    • Constraining the objects created: for example, ensuring that there is only one instance of its class

• Creational design patterns discussed in next lecture:
  – Factory
  – Abstract Factory
  – Prototype
  – Singleton
Structural Design Patterns

- Structural design patterns help us to arrange collections of objects in forms such as linked lists or trees.
- Structural design patterns discussed in lecture 8:
  - Composite
  - Decorator
  - Adapter
  - Facade
  - Flyweight
  - Proxy
Example of Behavioral Design

Goal: *Port Traffic*

obstacles

to drydock →

A systematic simulation of the transportation process
Design Goal At Work: \( \rightarrow \) Reusability \( \leftarrow \)

We want to be able to use classes *Ship* and *Tugboat* separately in other applications.
Avoiding Dependencies

Harbor application

Ship 1 ✗ 1..n Tugboat

Customs application: reuse Ship alone

Ship Longshoreman
Mediator Pattern: Captures the interaction between objects without having them reference each other (thus permitting reuse). Captures each interaction in a separate class that aggregates the objects involved.

The full Mediator Pattern allows for the fact that the mediated classes may need to communicate (e.g., to respond to events on either of them). To accomplish this, the mediated classes can be made to subclass a base class that aggregates a generic mediator class (*PortMission*).
Applying the Mediator Design Pattern to The Harbor Problem

Mediator base class

PortMission
estimateTime()

EnteringPort
estimateTime()

LeavingPort
estimateTime()

BeingMaintained
estimateTime()

Ship

Tugboat

Vessel

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

• **Goal:** A collection of objects to exhibit functionality.
  – At design time we want to be able to easily add or delete objects that handle all or part of the responsibility.

• **Example:** Design a GUI for a web application to view automobiles with requested color etc.
  – The display is dynamic, depending on the model, color etc. chosen.
  – Reuse the GUI parts among the displays.

• **Solution:** Link objects to each other via aggregation. Each performs some of the work, and then calls on the next object in the chain to continue the work.
Choose an automobile to view.

<table>
<thead>
<tr>
<th>Model</th>
<th>Body Style</th>
<th>Body color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Basic</td>
<td>Black</td>
</tr>
<tr>
<td>Midsize</td>
<td>Extra</td>
<td>Brown</td>
</tr>
<tr>
<td>Luxury</td>
<td>Luxury</td>
<td>Red</td>
</tr>
<tr>
<td>SUV</td>
<td></td>
<td>White</td>
</tr>
</tbody>
</table>

You have requested to view a luxury model with luxury body style, and black body color.
Characteristics of Design Patterns

- **Viewpoints** – ways to describe patterns
  1. *Static*: class model (building blocks)
  2. *Dynamic*: sequence or state diagram (operation)

- **Levels** – decomposition of patterns
  1. *Abstract* level describes the core of the pattern
  2. *Concrete* (= non abstract) level describes the particulars of this case

- **Roles** – the “players” in pattern usage
  1. *Application* of the design pattern itself
  2. *Clients* of the design pattern application
  3. *Setup* code initializes and controls
Characteristics of Design Patterns 2

1. Client role

2. Setup role

3. Role: Application of the design pattern

A. Static viewpoint
   (class model)

B. Dynamic viewpoint
   (sequence or state diagram)

(i) Abstract level
(ii) Concrete level

: Reference direction
Abstract Factory Application Sequence Diagram

Client

myStyle: KitchenStyle

-- IF myStyle BELONGS TO ModernKStyle --

myStyle: ModernKStyle

wallCabinet1: ModernWallCabinet

-- IF myStyle BELONGS TO AntiqueKStyle --

myStyle: AntiqueKStyle

wallCabinet1: AntiqueWallCabinet

getWallCabinet()
Key Concept: Two Viewpoints

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

Key Concept: Two Levels

Design patterns usually have an abstract level and a non-abstract ("concrete") level.
Concrete and Abstract Layers

Abstract level

Concrete level

ModernKStyle
AntiqueKStyle
ModernWallCabinet
AntiqueWallCabinet
ModernFloorCabinet
AntiqueFloorCabinet

Client
WallCabinet
FloorCabinet
KitchenStyle
Kitchen

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Design Goal At Work: \( \rightarrow \) Correctness \( \leftarrow \)

We want to provide an interface to a design pattern so that its functionality is clear and separate.
Key Concept: Three Roles

A part of a design either 1) applies a design pattern, 2) is a client of the pattern application, or 3) reinitializes these.
The Three Roles Involved in Design Pattern Usage

2. Client Role
   Interacts with the design pattern only through its interface

   DPClient

1. Design Pattern Application
   Interface to the pattern (frequently abstract; possibly several classes)

   DPInterface

3. Setup Role
   No limitations

   Rest of the Application

   Rest of the design pattern application
Metapatterns –
design patterns based on delegation or recursion

**Key Concept: → Two Forms ←**

A design pattern’s form is usually a delegation of responsibility or a class that relates to itself (recursive).
The Abstract Factory design pattern replaces several direct method calls (constructor calls actually) with delegated calls to separate methods, which call the desired methods in turn.
Basic Design Pattern Form #1: Delegation

```
Client

... interfaceMethod(...) 
{ ... doerObject.dolt() ... }

DPInterface
interfaceMethod()

doerObject

DoerBase
dolt()

ConcreteDoer1
dolt()

ConcreteDoer2
dolt()

... 
```
Basic Design Pattern Form #2: Recursion

Client

RecursionBase

NonrecursiveClass

RecursiveClass

void doOperation( ... )
{ ... aggregate ... }
The recursion form applied to an organization chart

The `printOrganization()` method in `Supervisor` programmed to print the supervisor’s name, then call the `printOrganization()` method in each of the aggregated employees. For employee objects in `IndividualContributor`, only the name of the employee is printed.
Summary of This Chapter

- **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 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**