Chapter 6

Introduction to Design Patterns
Sample Design Goals and Ways to Accomplish Them

- **Reusability, Flexibility, and Efficiency**
  - 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

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

A design pattern is a combination of classes and accompanying algorithms that fulfill a common design purpose.

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
6.1 - Categories of Design Patterns
Key Concept: Creational Design Patterns

-- used to create objects in flexible or constrained ways.

- Factory
- Abstract Factory
- Prototype
- Singleton

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

-- used to represent data structures such as linked lists or trees, with uniform processing interfaces.

☐ Composite
☐ Decorator
☐ Adapter
☐ Facade
☐ Flyweight
☐ Proxy

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

-- used to capture behavior among objects.

- Chain of Responsibility
- Command
- Interpreter
- Mediator
- Observer
- State
- Template
- Iterator

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
6.2 - Example Use of a Creational Design Pattern

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

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

Our design should be flexible enough to produce any of several kitchen styles.

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

**The Abstract Factory Idea**

```
KitchenStyle
getWallCabinet()
getFloorCabinet()

WallCabinet

FloorCabinet

... AntiqueWallCabinet ...

AntiqueKStyle
getWallCabinet()
getFloorCabinet()

ModernKStyle
getWallCabinet()
getFloorCabinet()

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

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

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Abstract Factory Design Pattern Applied to KitchenViewer

Client
renderKitchen(KitchenStyle)

KitchenStyle
getWallCabinet()
getFloorCabinet()

Kitchen
getWallCabinet()
getFloorCabinet()

WallCabinet

FloorCabinet

WallCabinet

FloorCabinet

ModernWallCabinet

AntiqueWallCabinet

ModernFloorCabinet

AntiqueFloorCabinet

ModernKStyle
getWallCabinet()
getFloorCabinet()

AntiqueKStyle
getWallCabinet()
getFloorCabinet()

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Example of Code inside `renderKitchen(KitchenStyle myStyle)`

```java
// Create the wall cabinets; type is determined by the class of myStyle
WallCabinet wallCabinet1 = myStyle.getWallCabinet();
WallCabinet wallCabinet2 = myStyle.getWallCabinet();

// Create the floor cabinets; type determined by the class of myStyle
FloorCabinet floorCabinet1 = myStyle.getFloorCabinet();
FloorCabinet floorCabinet2 = myStyle.getFloorCabinet();

// Create the kitchen object (in the style required)
Kitchen kitchen = new Kitchen();
kitchen.add(wallCabinet1, . . .);
kitchen.add(wallCabinet2, . . .);

. . .
kitchen.add(floorCabinet1, . . .);
kitchen.add(floorCabinet2, . . .);
. . .
```
Abstract Factory Design Pattern

Client

```
doOperation( Style myStyle )
```

Style

```
getComponentA()
getComponentB()
```

Collection

ComponentA

ComponentB

Style1

```
getComponentA()
getComponentB()
```

Style2

```
getComponentA()
getComponentB()
```

Style1ComponentA

Style2ComponentA

Style1ComponentB

Style2ComponentB

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

Alternative

Client
doOperation()

getComponentA()
getComponentB()

Collection
getComponentA()
getComponentB()

Style
getComponentA()
getComponentB()

ComponentA

ComponentB

Style1
getComponentA()
getComponentB()

Style2
getComponentA()
getComponentB()

Style1ComponentA

Style2ComponentA

Style1ComponentB

Style2ComponentB

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
6.3 - Example Use of a Behavioral Design Pattern

Mediator
Example of Behavioral Design Goal: Port Traffic

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

A customs application: reuses Ship alone

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Mediator Concept Applied to The Harbor Problem

LeavingPort
estimateTime()

Ship

Tugboat

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Applying the *Mediator* Design Pattern to The Harbor Problem

`PortMission`
- `estimateTime()`

Mediator base class

`Vessel`

`Ship`

`Tugboat`

`EnteringPort`
- `estimateTime()`

`LeavingPort`
- `estimateTime()`

`BeingMaintained`
- `estimateTime()`

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

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

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

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Characteristics of Design Patterns 2

1. Client role

2. Setup role

3. Role: Application of the design pattern

A. Static viewpoint
   - (i) Abstract level
   - (ii) Concrete level

B. Dynamic viewpoint
   - (sequence or state diagram)

(class or classes)

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Key Concept:  ➔ Two Levels  ←

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

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

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

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

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

Each part of a design pattern fills one of three roles:
1) Applies (i.e., instantiates) the design pattern, or
2) is a client of the pattern application, or
3) reinitializes or sets up the two previous roles

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

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

*Interacts with the design pattern only through its interface*

3. Setup Role

*No limitations*

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
6.5 - Forms of Design Patterns
Design Pattern Forms

-- Forms represent patterns to the design patterns (i.e., metapatterns)
-- They consist of delegation forms and recursion forms

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

Client

```
clientFunction()
{
  intermediaryFunction( ... ) ...
}
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Basic Design Pattern Form #1: Delegation

DoerBase

doit()

DPInterface

interfaceMethod()

ConcreteDoer1
doit()

ConcreteDoer2
doit()

... interfaceMethod( ... )
{ ... doerObject.doit() ... }

Client

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Basic Design Pattern Form #2: Recursion

Client

\[ \text{RecursionBase} \]
\[ \text{doOperation()} \]

\[ \text{NonrecursiveClass} \]
\[ \text{doOperation()} \]

\[ \text{RecursiveClass} \]
\[ \text{doOperation()} \]

void doOperation(\ldots)
{ ... aggregate ... }

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
The *Recursion* Form Applied to an Organization Chart

```
void printOrganization( … )
{
    … supervisees.printOrganization() …
}

Client

Employee

IndividualContributor

Supervisor

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

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

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

- Design Patterns are recurring designs satisfying recurring design purposes

- Classified as Creational, Structural, or Behavioral

  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 are Delegation or Recursion

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