Example: Duck Game

• A startup produces a duck-pond simulation game
• The game shows a large variety of duck species swimming and making quacking sounds
Initial Design

```java
class Duck {
    void quack() {
    }
    void swim() {
    }
    void display() {
        // Other duck-like method
    }
}

class MillardDuck {
    void display() {
        looks like a mallard
    }
}

class RedheadDuck {
    void display() {
        looks like a redhat
    }
}

// Other types of ducks
```
Design Change: add fly()

Duck

quack()
swim()
display() = 0
fly()
// Other duck-like method

MillardDuck

display() {
  looks like a mallard
}

RedheadDuck

display() {
  looks like a redhat
}

Other types of ducks
Problem

• Generalization may lead to unintended behaviors: a rubber duck is flying and quacks

<table>
<thead>
<tr>
<th>Duck</th>
</tr>
</thead>
<tbody>
<tr>
<td>quack()</td>
</tr>
<tr>
<td>swim()</td>
</tr>
<tr>
<td>display() = 0</td>
</tr>
<tr>
<td>// Other duck-like method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MillardDuck</th>
</tr>
</thead>
<tbody>
<tr>
<td>display() { looks like a mallard }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RubberDuck</th>
</tr>
</thead>
<tbody>
<tr>
<td>display() {\textit{looks like a rubber duck} }</td>
</tr>
<tr>
<td>quack() { // sqeak }</td>
</tr>
<tr>
<td>fly() { // cannot fly }</td>
</tr>
</tbody>
</table>
Anticipating Changes

• Identify the aspects of your application that may vary
  – What may change?

• Anticipate that
  – new types of ducks may appear and
  – behaviors (quack, swimming, and flying) may change, even change at run time (swirl flying, circular flying, ...)

Handling Varying Behaviors

- Solution: take what varies and “encapsulate” it
  - Since fly() and quack() vary across ducks, separate these behaviors from the Duck class and create a new set of classes to represent each behavior.
Design

• Each duck object has a fly behavior

```java
<<interface>>
FlyBehavior
fly()

FlyWithWings
fly() {
    //
}

FlyNoWay
fly() {
    // cannot fly
}
```
Programming to implementation vs. interface/supertype

• Programming to an implementation
  – Dog d = new Dog();
  – d.bark();

• Programming to an interface/supertype
  – Animal a = new Dog();
  – a.makeSound();
Implementation

Duck

quack()
swim()
display() = 0
// Other duck-like method

MallardDuck
display() {
  looks like a mallard
}

RedheadDuck
display() {
  looks like a redhat
}

<<interface>>
FlyBehavior

fly()

FlyWithWings

fly()

//

FlyNoWay

fly()

// cannot fly
Exercise

• Add rocket-powered flying?
The Strategy Pattern

• Defines a set of algorithms, encapsulates each one, and makes them interchangeable by defining a common interface
Exercise

```java
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```
Summary: Design Principles

• Identify the aspects of your application that vary and separate them from what stay the same

• Program to an interface not implementation

• Favor composition over inheritance
Example: KitchenViewer Interface

- Wall cabinet
- Counter
- Floor cabinet

- menu
- display area
- styles

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

Wall cabinets

Floor cabinets

Countertop

Modern  Classic  Antique  Arts & Crafts
Selecting *Antique* Style
Aspect of the system that may change/vary?
The Abstract Factory Idea

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

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

Client
renderKitchen(KitchenStyle)

KitchenStyle
getWallCabinet()
g-getFloorCabinet()

Kitchen
getWallCabinet()
g-getFloorCabinet()

WallCabinet

FloorCabinet

ModernKStyle
getWallCabinet()
g-getFloorCabinet()

ModernWallCabinet

AntiqueKStyle
getWallCabinet()
g-getFloorCabinet()

AntiqueWallCabinet

ModernFloorCabinet

AntiqueFloorCabinet
Abstract Factory Design Pattern

- **Style**
  - getComponentA()
  - getComponentB()

- **Style1**
  - getComponentA()
  - getComponentB()

- **Style2**
  - getComponentA()
  - getComponentB()

- **ComponentA**

- **ComponentB**

- **Style1ComponentA**

- **Style1ComponentB**

- **Style2ComponentA**

- **Style2ComponentB**

- **Style1ComponentB**

- **Style2ComponentB**

- **Collection**

- **Client**
  - doOperation( Style myStyle )
Concrete and Abstract Layers

KitchenStyle

Abstract level

Kitchen

Concrete level

ModernWallCabinet  AntiqueWallCabinet

ModernKStyle  AntiqueKStyle

ModernFloorCabinet  AntiqueFloorCabinet
Abstract Factory Application
Sequence Diagram

Client

myStyle: KitchenStyle

-- IF myStyle BELONGS TO ModernKStyle --

myStyle: ModernKStyle

getWallCabinet()

ModernWallCabinet()

-- IF myStyle BELONGS TO AntiqueKStyle --

myStyle: AntiqueKStyle

getWallCabinet()

AntiqueWallCabinet()
Potential use of this Design Pattern?

```plaintext
Client
  doOperation( Style myStyle )

Style
  getComponentA()
  getComponentB()

Style1
  getComponentA()
  getComponentB()

Style2
  getComponentA()
  getComponentB()

Collection

ComponentA

ComponentB

Style1ComponentA

Style1ComponentB

Style2ComponentA

Style2ComponentB

Style1
  getComponentA()
  getComponentB()

Style2
  getComponentA()
  getComponentB()
```

Collection

ComponentA

ComponentB

Style1ComponentA

Style1ComponentB

Style2ComponentA

Style2ComponentB
References

• Design Patterns

• Headfirst Design Patterns

• Software Design