Reusability, Flexibility, and Maintainability

• One thing constant in software development is CHANGE

• For software that is used over a period of years, the cost of keeping it current in the face of changing needs often exceeds the cost of originally developing it.

• A key need in software design is the ability for maintenance and modification to keep abreast of changes.
The Waterfall Software Process

Milestone(s)
- Requirements Analysis

Phases (activities)
- Design
- Implementation
- Testing
- Maintenance

Two phases may occur at the same time for a short period

Release product X
Why a Pure Waterfall Process is Usually Not Practical

- *Don’t know up front everything wanted and needed*
  - Usually hard to visualize every detail in advance

- To gain confidence in an estimate, we need to design and actually implement parts, especially the riskiest ones, this may probably lead to modify requirements as a result

- *We often need to execute intermediate builds*
  - Stakeholders need to gain confidence
  - Designers and developers need confirmation they're building what’s needed and wanted

- *Team members can't be idle while the requirements are being completed*
The Spiral Process

**Requirements analysis**
- Iteration #1
  - 1

**Design**
- Iteration #2
  - 2
- Iteration #3
  - 3

**Coding**
- Iteration #1
  - 1
- Iteration #2
  - 2
- Iteration #3
  - 3

**Testing**
- Iteration #1
  - 1
- Iteration #2
  - 2
- Iteration #3
  - 3

**Intermediate version** completed X
**Product released** X

*MILESTONES*

- Iteration #1
- Iteration #2
- Iteration #3

*typically a prototype*

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

OO systems exhibit recurring structures that promote

- Abstraction
- Modularity
- Flexibility
- Extensibility
- Elegance
Aspect of Reusability

• **Classes – in source code form**
  – Thus, we write *generic code* whenever possible

• **Assemblies of related classes**
  ▪ A *toolkit* is a library of reusable classes designed to provide useful, general-purpose functionality.
    • E.g., C++ standard library, Boost
  ▪ An *application framework* is a specific set of classes that cooperate closely with each other and together embody a reusable design for a category of problems.
    • E.g., Java APIs (Applet, Thread, etc), gtkmm

• Design pattern
Making a Class Re-usable

- Define a useful abstraction
  - attain broad applicability
- Reduce dependencies on other classes

...
Reducing Dependency Among Classes

Replace ...

Student

Course

with ...

Student

Enrollment

Course
Aspect of Flexibility

• Making small variation to existing functionality

• Adding new kinds of functionality

• Changing functionality
### Some Techniques to Achieve Flexibility

<table>
<thead>
<tr>
<th>Flexibility Aspect: ability to ...</th>
<th>Some techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>... create objects in variable configurations determined at runtime</td>
<td>“Creational” design patterns</td>
</tr>
<tr>
<td>... create variable trees of objects or other structures at runtime</td>
<td>“Structural” design patterns</td>
</tr>
<tr>
<td>... change, recombine, or otherwise capture the mutual behavior of a set of objects</td>
<td>“Behavioral” design patterns</td>
</tr>
<tr>
<td>... create and store a possibly complex object of a class.</td>
<td>Component</td>
</tr>
<tr>
<td>... configure objects of predefined complex classes – or sets of classes – so as to interact in many ways</td>
<td>Component</td>
</tr>
</tbody>
</table>
Roadmap

• We will focus on flexibility and reusability
  – It is important to remember that real systems also need to consider efficiency and robustness

• We will start with design patterns, and then look into the design of some OO libraries/toolkit/framework

• We will learn by examples:
  – *Example* is not another way to teach, it is the *only* way to teach. -- *Albert Einstein*
What is a Design Pattern

• Abstracts a recurring design structure
• Comprises class and/or object
  ▪ dependencies
  ▪ structures
  ▪ interactions
  ▪ conventions

• Distills design experience
• Names & specifies the design structure explicitly

• Language- & implementation-independent
  • A “micro-architecture”
UML/OMT Notation

```
AbstractClass
  abstractOperation()

ConcreteClass

ConcreteSubclass1
  operation()
  instanceVariable

ConcreteSubclass2

object reference
aggregation
creates

one
many

implementation pseudocode
```
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

**Duck**

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

**MillardDuck**

- display() {
  looks like a mallard
}

**RedheadDuck**

- 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

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

// MillardDuck
display() {
    looks like a mallard
}

// RubberDuck
display() {looks like a rubber duck}
quack() { // squeak }
fly() { // cannot fly }
```
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

![Diagram showing the separation of behaviors from the Duck class](image)
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

```
FlyWithWings
fly()

FlyNoWay
fly()
```

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

```
MallardDuck
display() {
  looks like a mallard
}
```

```
RedheadDuck
display() {
  looks like a redhat
}
```

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

```
Flock
```

```
Duck
```

```
FlyWithWings
fly()
```

```
FlyNoWay
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

Character
WeaponBehavior weapon;
fight();

Queen
fight() {...}

King
fight() {...}

Troll
fight() {...}

Knight
fight() {...}

KnifeBehavior
useWeapon() // implements cutting with a knife

WeponBehavior
useWeapon();

AxeBehavior
useWeapon() // implements chopping with an axe

BowAndArrowBehavior
useWeapon() // implements shooting with a bow

SwordBehavior
useWeapon() // implements swinging a sword

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

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

Wall cabinets

Countertop

Floor cabinets

Modern  Classic  Antique  Arts & Crafts
Selecting *Antique* Style

Modern

Classic

Antique

Arts & Crafts
KitchenViewer Using Standard Inheritance

Aspect of the system that may change/vary?
The Abstract Factory Idea

```
AntiqueKStyle
  getWallCabinet()
  getFloorCabinet()

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)

**KitchenStyle**

getWallCabinet()
getFloorCabinet()

**Kitchen**

getWallCabinet()
getFloorCabinet()

**WallCabinet**

**FloorCabinet**

**ModernKStyle**

getWallCabinet()
getFloorCabinet()

**AntiqueKStyle**

getWallCabinet()
getFloorCabinet()

**ModernWallCabinet**

**AntiqueWallCabinet**

**ModernFloorCabinet**

**AntiqueFloorCabinet**

ModernKStyle

AntiqueKStyle

ModernWallCabinet

AntiqueWallCabinet

ModernFloorCabinet

AntiqueFloorCabinet
Abstract Factory Design Pattern

- **Client**
  - `doOperation( Style myStyle )`

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

- **Collection**

- **ComponentA**
  - **Style1ComponentA**
  - **Style2ComponentA**

- **ComponentB**
  - **Style1ComponentB**
  - **Style2ComponentB**

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

- **Style2**
  - `getComponentA()`
  - `getComponentB()`
myStyle: KitchenStyle

Client

-- IF myStyle BELONGS TO ModernKStyle --

myStyle:
ModernKStyle

getWallCabinet()

renderKitchen ( myStyle )

getWallCabinet()

ModernWallCabinet()

-- IF myStyle BELONGS TO AntiqueKStyle --

myStyle:
AntiqueKStyle

getWallCabinet()

AntiqueWallCabinet()
Potential use of this Design Pattern?

Client
doOperation( Style myStyle )

Collection

Style
getComponentA()
getComponentB()

ComponentA

ComponentB

Style1
getComponentA()
getComponentB()

Style2
getComponentA()
getComponentB()

Style1ComponentA

Style2ComponentA

Style1ComponentB

Style2ComponentB