COMP2059 Developing Maintainable Software

LECTURE 04 – DESIGN PRINCIPLES AND PATTERNS

Boon Giin Lee (Bryan) and Heng Yu

From Code to Patterns

DESIGN PRINCIPLES AND PATTERNS EXTEND THE OBJECT-ORIENTED PHILOSOPHY

From Code to Patterns

- OO Concepts.
  - Object model (abstraction, encapsulation, modularity); inheritance; polymorphism; interfaces.

- OO Design Principles.
  - *Guidelines* that you should be aware and follow when designing and writing your code.

- OO Design Patterns.
  - Some pieces of *reusable code patterns* that has already been proved effective and maintainable, so that you can reuse them when similar scenarios are encountered.

Why Design Principles and Patterns

- Software development specifications keep changing constantly.
  - This is in particular true in the business world.
- Software needs to be maintainable enough to cope.
  - Easy to maintain (by applying changes with minimal effort).
  - Easy to extend (without changing existing code).
- Follow programming guidelines and systematic conventions, to allow one to be capable of maintaining another’s code.

Code Sense

- "A programmer without code-sense can look at a messy module and recognise the mess but will have no idea what to do about it."
- "A programmer with code-sense will look at a messy module and see options and variations. The code-sense will help the programmer choose the best variation and guide him or her to plot a sequence of behavior-preserving transformations to get from here to there." (Robert C. Martin a.k.a. Uncle Bob, "Clean Code: A handbook of Agile Software Craftsmanship", 2008.)
- Design principles and patterns help us to develop our code sense.
- As with all design principles and patterns, they do not involve code reuse – they employ "experience reuse".

University of Nottingham
UK | CHINA | MALAYSIA

Learning aims and outcomes

- *Aims*
  - Develop students' programming ability and experience, by employing software design principles and patterns to understand and develop real-world, large software projects.
- *Learning outcomes*
  - Understand the difficulties to maintain real-world large software. **[Why]**
  - Define various software design principles and design patterns. **[What]**
  - Examine and investigate appropriate locations to apply design principles and patterns. **[When & Where]**
  - Prepare students to code with design patterns in software development. **[How]**
  - Prepare students to effectively communicate and collaborate with real world professional software developers. **[Make it yours!]**

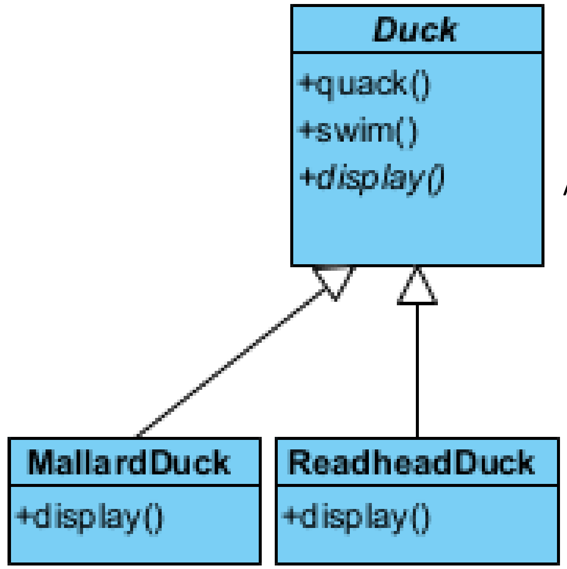
This week's outline

- SimuDuck: First impression on maintenance
  - How hard is to maintain your code
  - First several design principles
  - First design pattern: Strategy pattern
- More pattern stories:
  - StarBuzz: decorate your coffee
  - Arabina: needs a pizza factory
- More principles and patterns
  - SOLID Principle
  - Design patterns in general
- Summary

The Addison Wesley Signature
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
software development 15th annual productivity award
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts
O'REILLY
10th Anniversary Updated for Java 8
Head First Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

A story to start off…

- Joe works for a company that is famous for its duck pond simulating software, SimuDuck. The game can show a large variety of duck species swimming and making quacking sounds. This is the initial design.

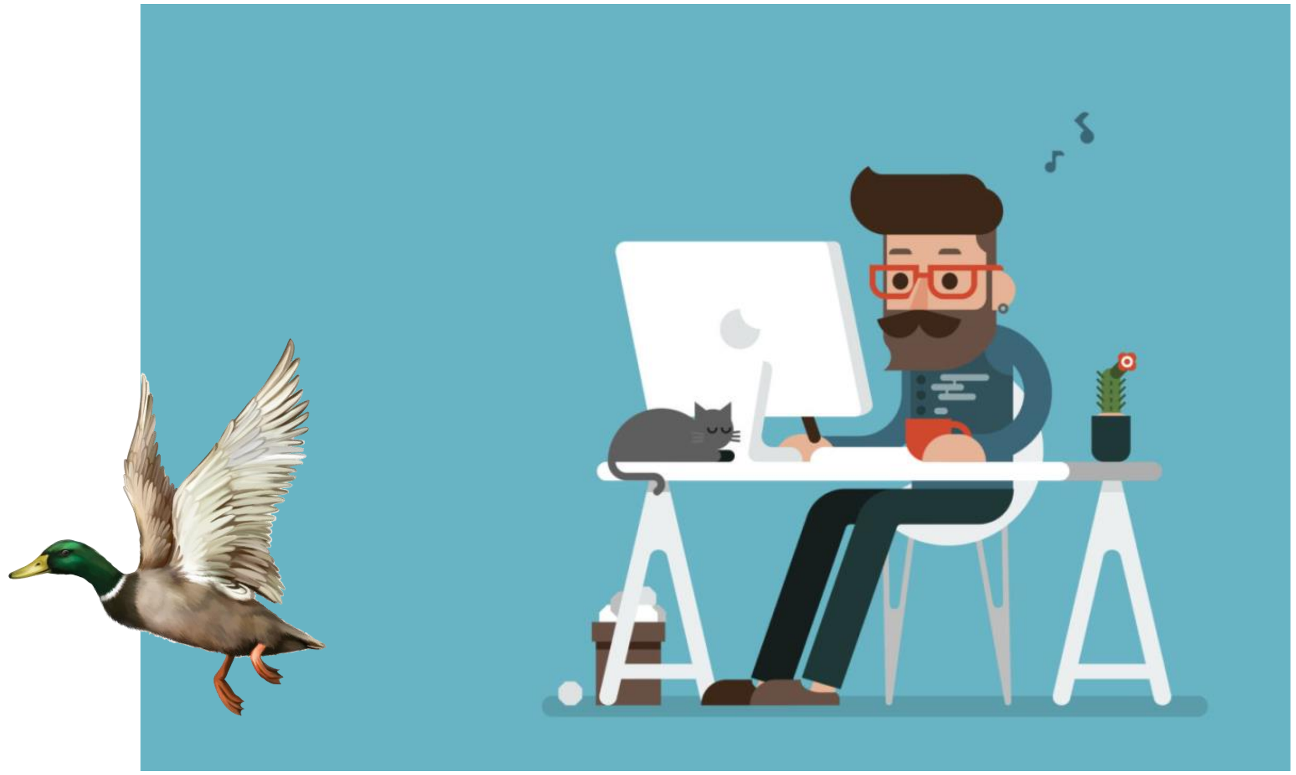


// Display the duck on screen

… Lots of other types of duck …

Now we need the ducks to FLY

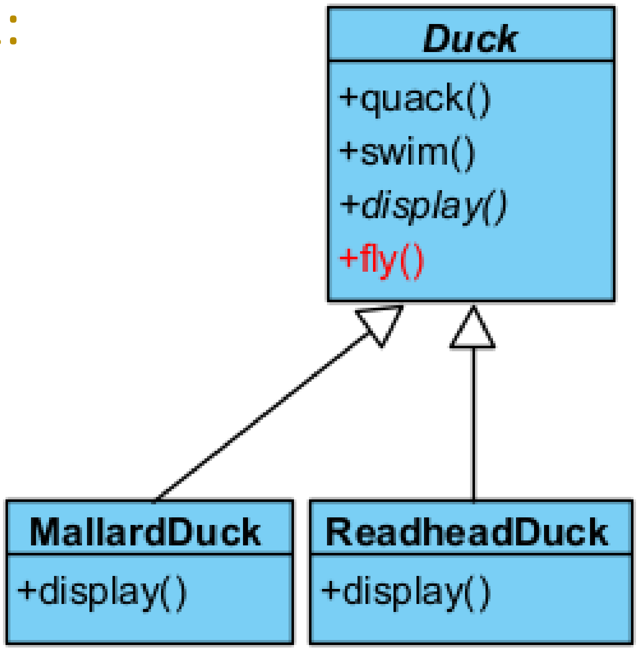
- Now, the CEO decides to simulate a flying duck to maintain the software's competitiveness.



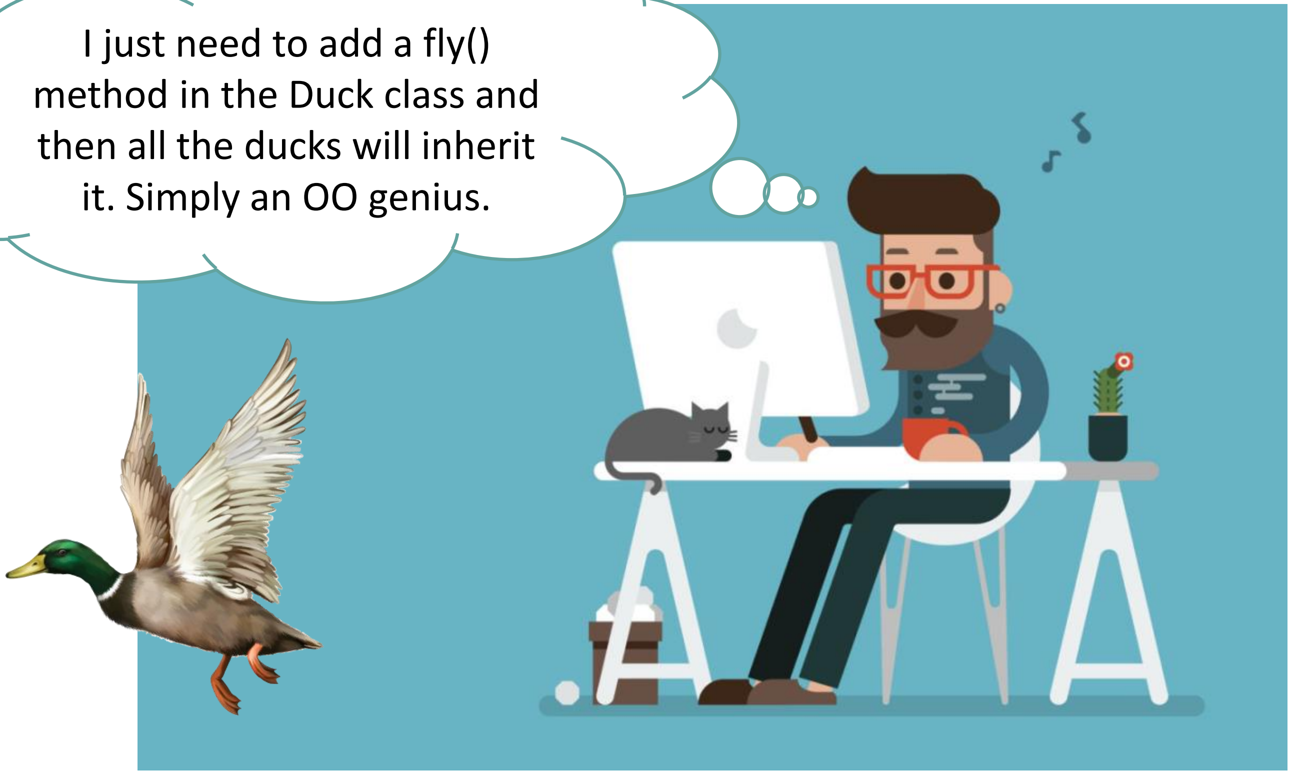
Now we need the ducks to FLY

- Now, the CEO decides to simulate a flying duck to maintain the software's competitiveness.

Design 1:

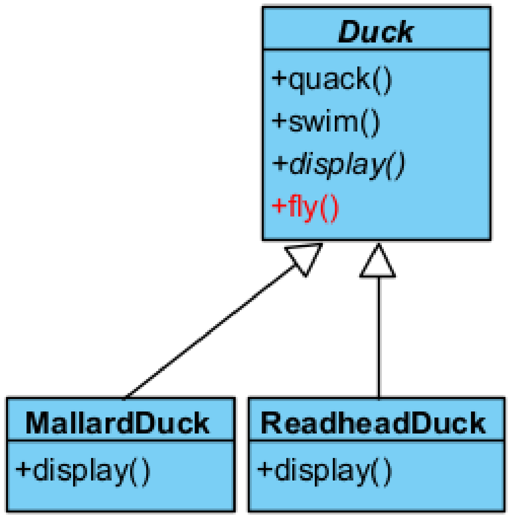


I just need to add a fly() method in the Duck class and then all the ducks will inherit it. Simply an OO genius.



Now we need the ducks to FLY

- What do you think of this design? Provide your opinion via the QR code.



But something went wrong…

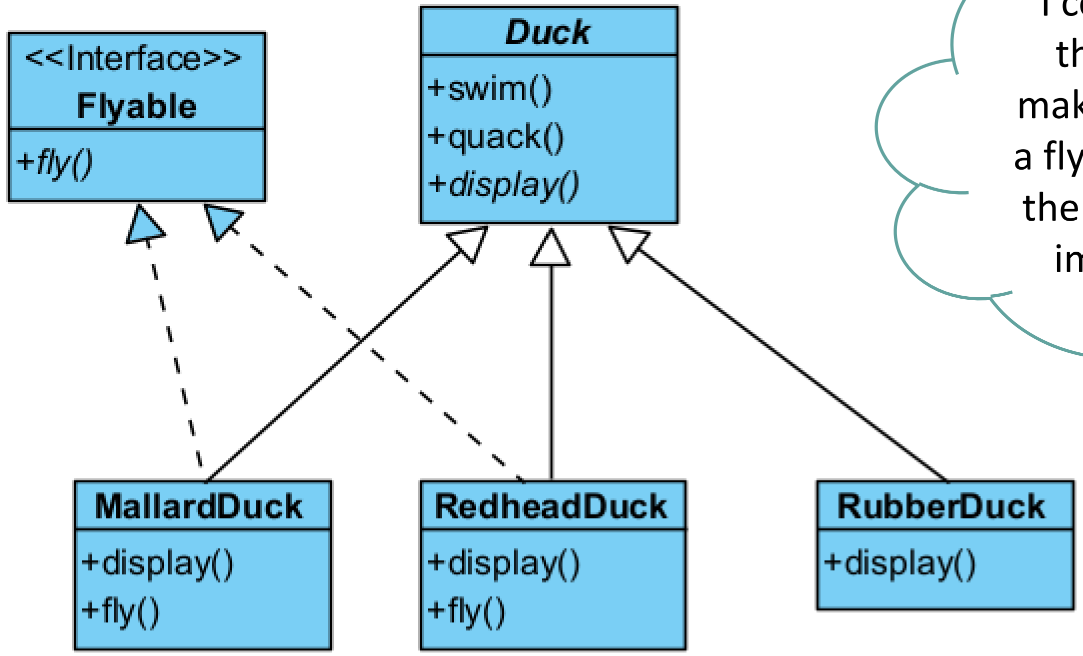
- CEO: Joe, I am at the stakeholder's meeting. I demo our App to them, and why I see rubber duckies flying around!! ☹

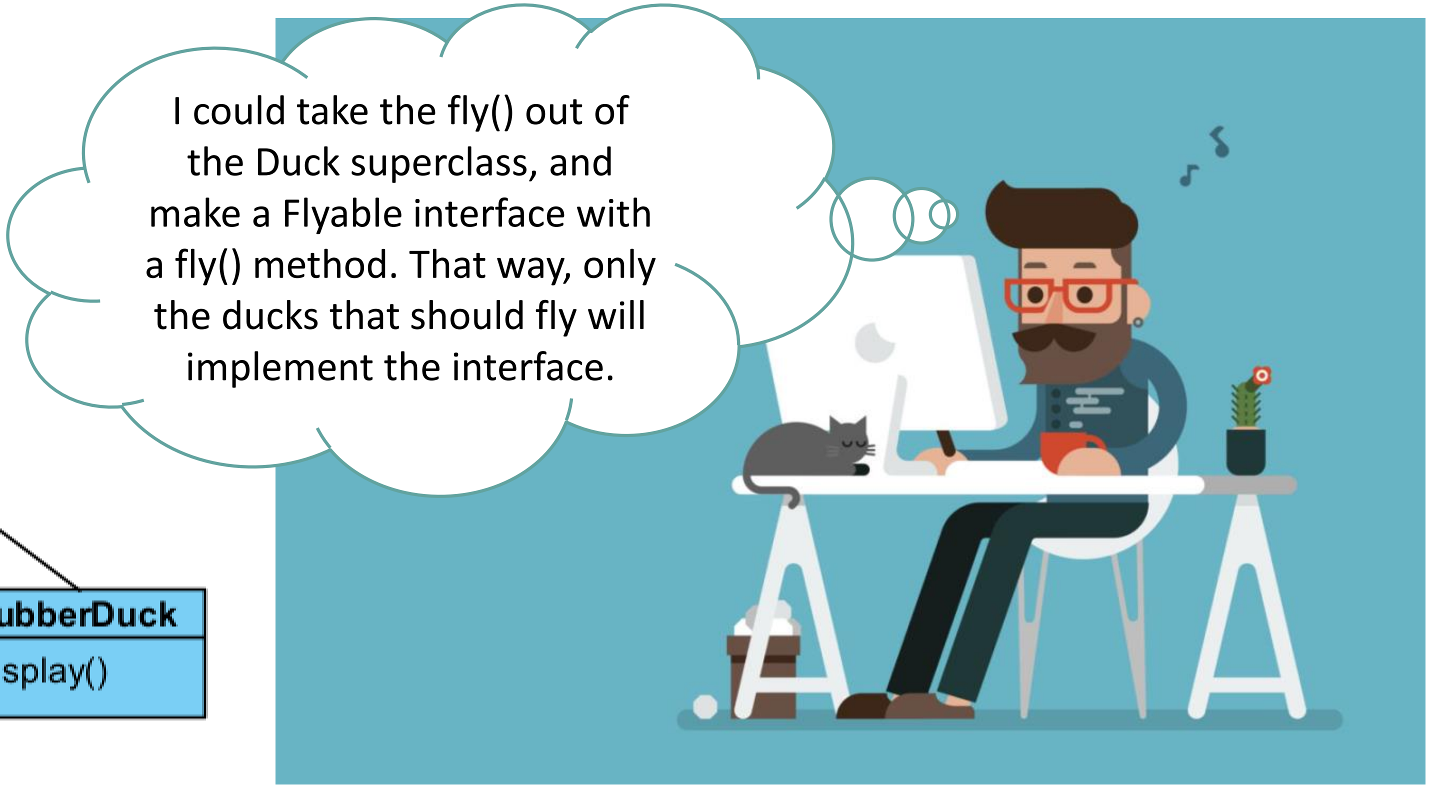
- Joe failed to notice that *not all* subclasses of Duck should *fly*.
  - When Joe added new behavior to the Duck superclass, he was also adding behavior that was not appropriate for some subclasses.
  - Inheritance may not be the best solution.

How about using Interface?

- Since not all subclasses implements fly, how about letting the subclasses who should fly implement the Flyable Interface?

Design 2:



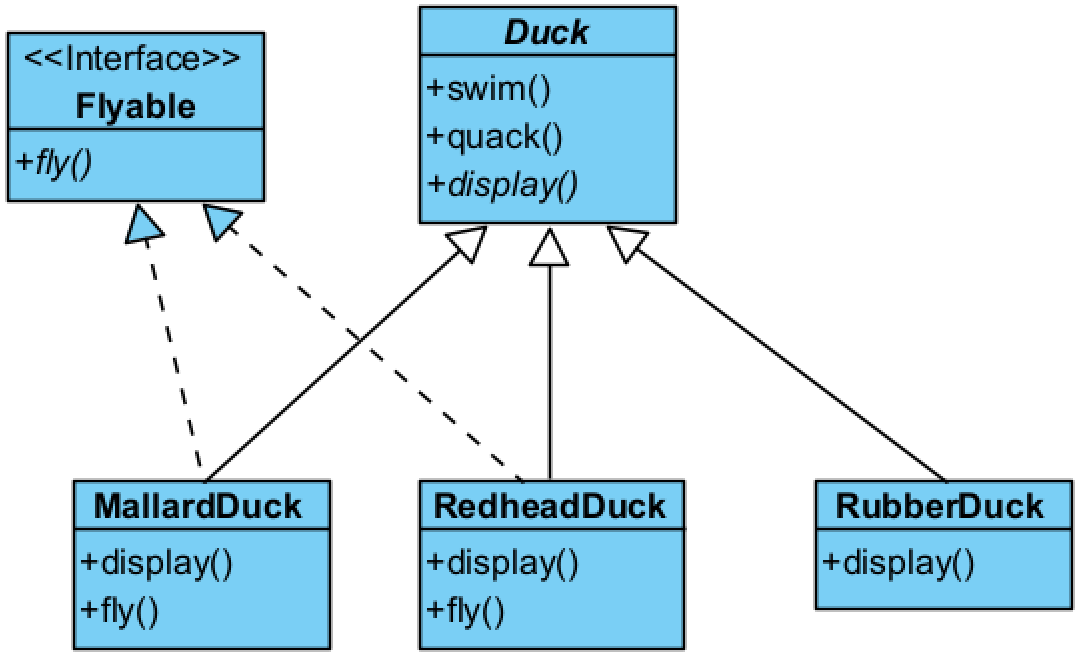


I could take the fly() out of the Duck superclass, and make a Flyable interface with a fly() method. That way, only the ducks that should fly will implement the interface.

But another maintenance issue…

- While this solved the inheritance problem, it completely destroys the code reuse of fly() method.
  - What if a common procedure of the fly() method later needs to be modified?
  - All subclasses should modify the fly() method.
  - A maintenance nightmare.

- How to solve this issue indeed?
  - Let’s resort to Design Principles.



Encapsulating what varies

Design Principle:
Identify the aspects of your application that vary, and separate them from what stays the same.

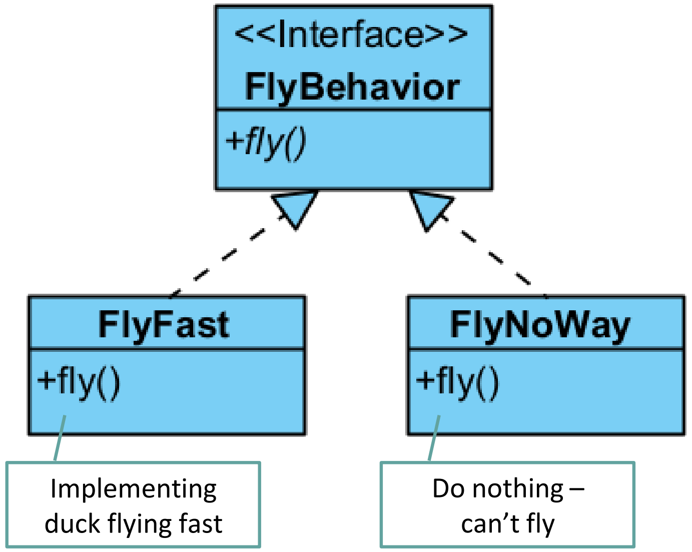
- If some aspects of your code is changing (with new requirement), then pull out what varies and "encapsulate" it so it won't affect the rest of your code.
- Advantage: fewer unintended consequences from code changes, and more flexibility to your system.
- Forms the basis of almost all design patterns. Namely, all design patterns provide a way to *let some part of a system vary independently of all other parts*.

The Duck example

- What varies in the Duck example?
  - Their flying behavior varies.
    - Some types of bird fly fast
    - Some fly slow
    - Some do not fly
    - …

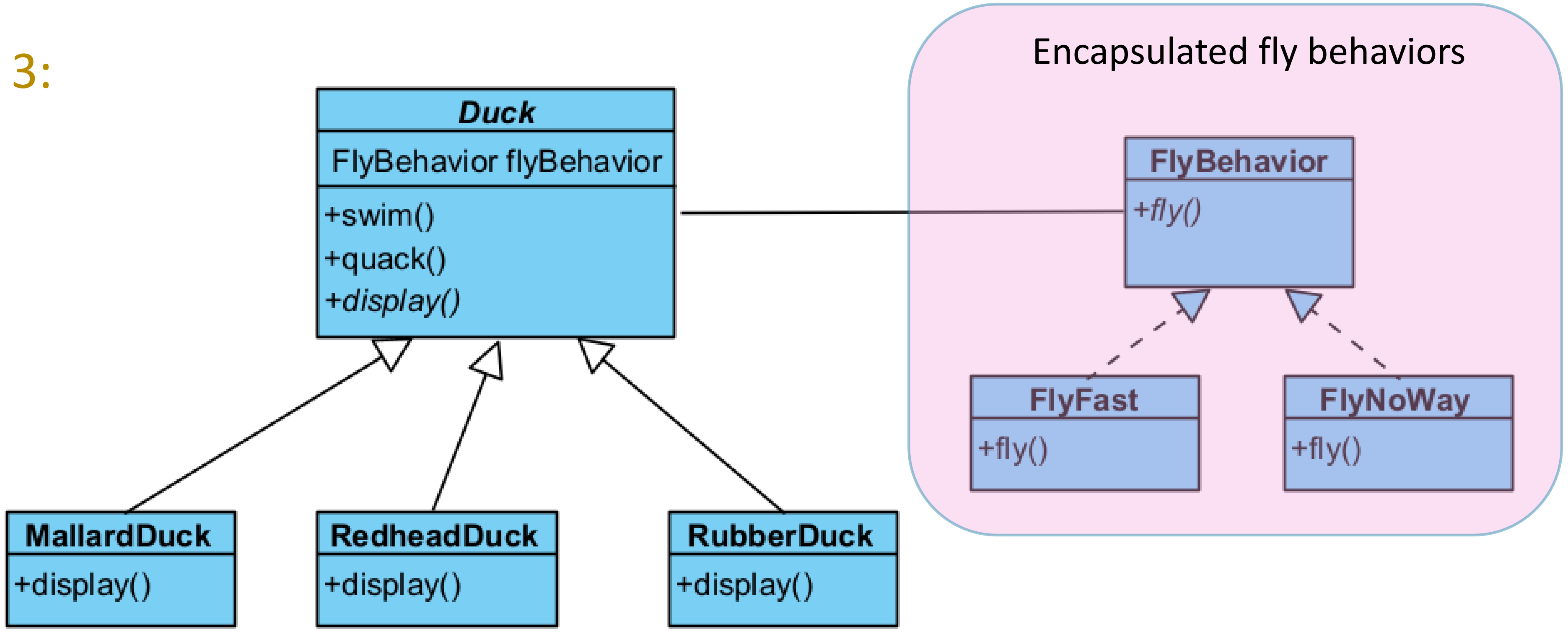
- What to do?
  - Pull out the flying behaviors into Interface
  - Each implementation of a flying behavior will implement the interface.

Design Principle:
Identify the aspects of your application that vary, and separate them from what stays the same.



The big picture

Design 3:



- Note how the Fly behaviors are encapsulated.
  - Any new behaviors added/changed later will be *separated* from the Duck class and subclasses.

Programming to an interface

Design Principle:
Program to an interface, not an implementation.

- Recall that:
  - Method 1: fly() defined in Duck class
  - Method 2: fly() defined in subclasses
    - In both cases, replying on specific Duck implementations – rigid when changing the fly behaviors.
  - Method 3: fly() *defined in a FlyBehavior interface*
    - The actual implementation is separated out from Duck class/subclasses.

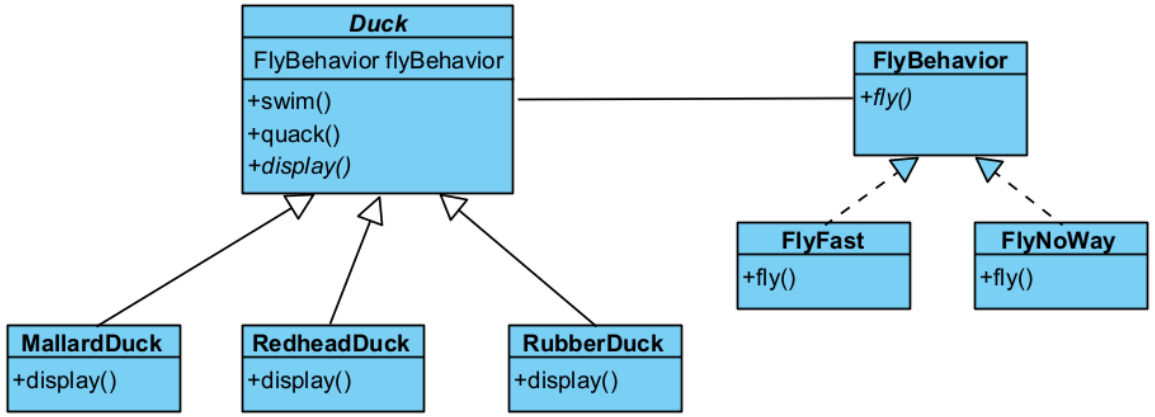
HAS-A can be better than IS-A

Design Principle:

Favor composition over inheritance.

- Method 3 exhibits a HAS-A relationship.
  - "A duck has a fly behavior."
  - Creating systems using composition provides lots more flexibility:

1) Separation of what vary and what don't vary.
2) *Change behavior at runtime!* [Polymorphism]

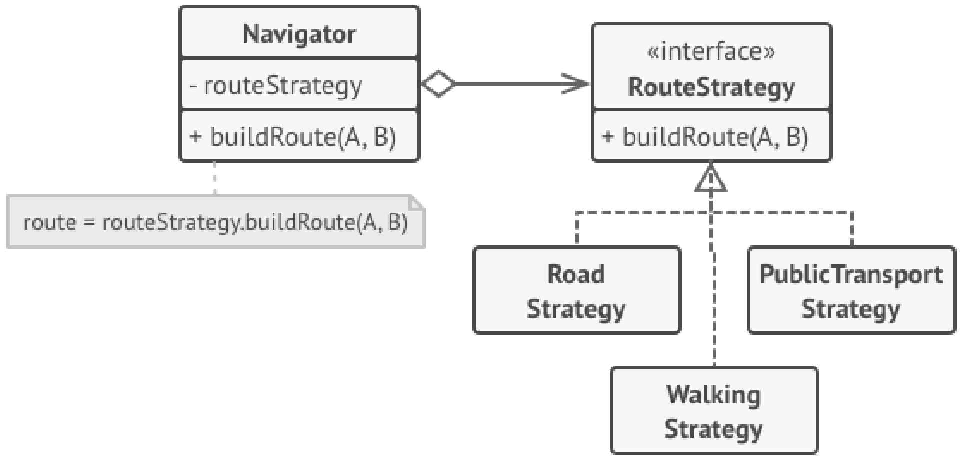


Your first design pattern!

Design Pattern:

The Strategy Pattern is a behavioral pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

- Congratulations! By going through this Duck's example, you have just applied your first design pattern: *Strategy pattern*.
  - An *algorithm* is a set of steps of completing a task. In the Duck example, an algorithm is actually a specific flying behavior (fast, slow, no, etc.)



A *Route planning* example found at refactoring guru

What you've learnt so far

- It's easy for your code to become hard to maintain.
  - Think about the Duck example: given new requirements, modifying your code needs to be very careful.
- Follow Design Principles <u>when</u> you code.
  - It means <u>when</u> you design your system from zero
  - It also means <u>when</u> you modify an existing system
  - It means … <u>whenever</u> you code.
- Think about existing Design Patterns that can help with your scenario.

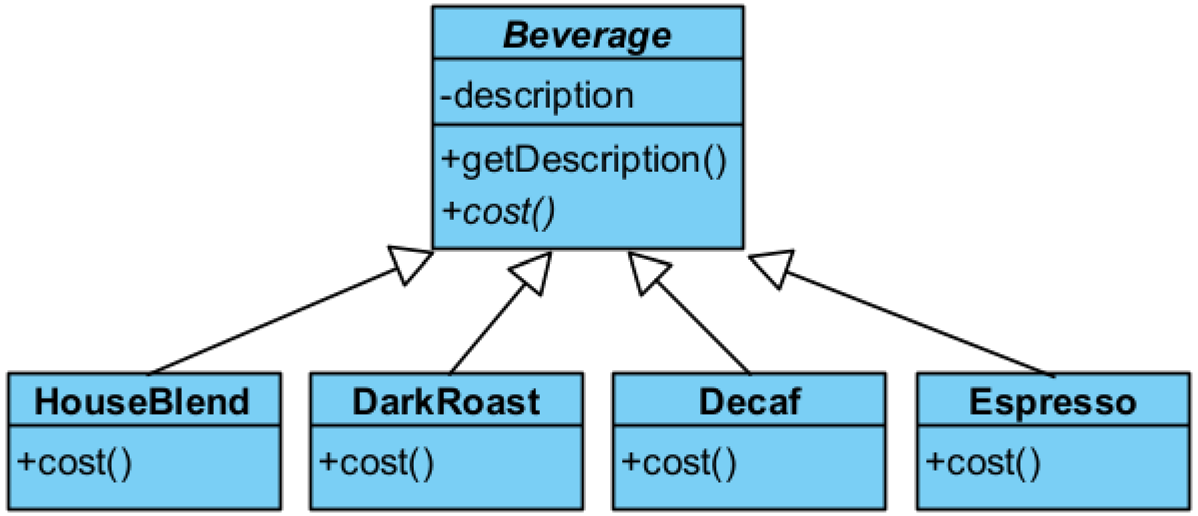
This week's outline

- SimuDuck: First impression on maintenance
  - How hard is to maintain your code
  - First several design principles
  - First design pattern: Strategy pattern
- More pattern stories:
  - StarBuzz: decorate your coffee
  - Arabina: needs a pizza factory
- More principles and patterns
  - SOLID Principle
  - Design patterns in general
- Summary

The Addison Wesley Signature
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts
O'REILLY
Head First Design Patterns
A Brain-Friendly Guide
10th Anniversary Updated for Java 8
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Welcome to StarBuzz

- As the cw deadline for DMS approaches, the business of StarBuzz coffee on campus grows so quickly, and they're scrambling to update their ordering system to match their beverage offerings.
- Below is their version zero system, with only coffees…



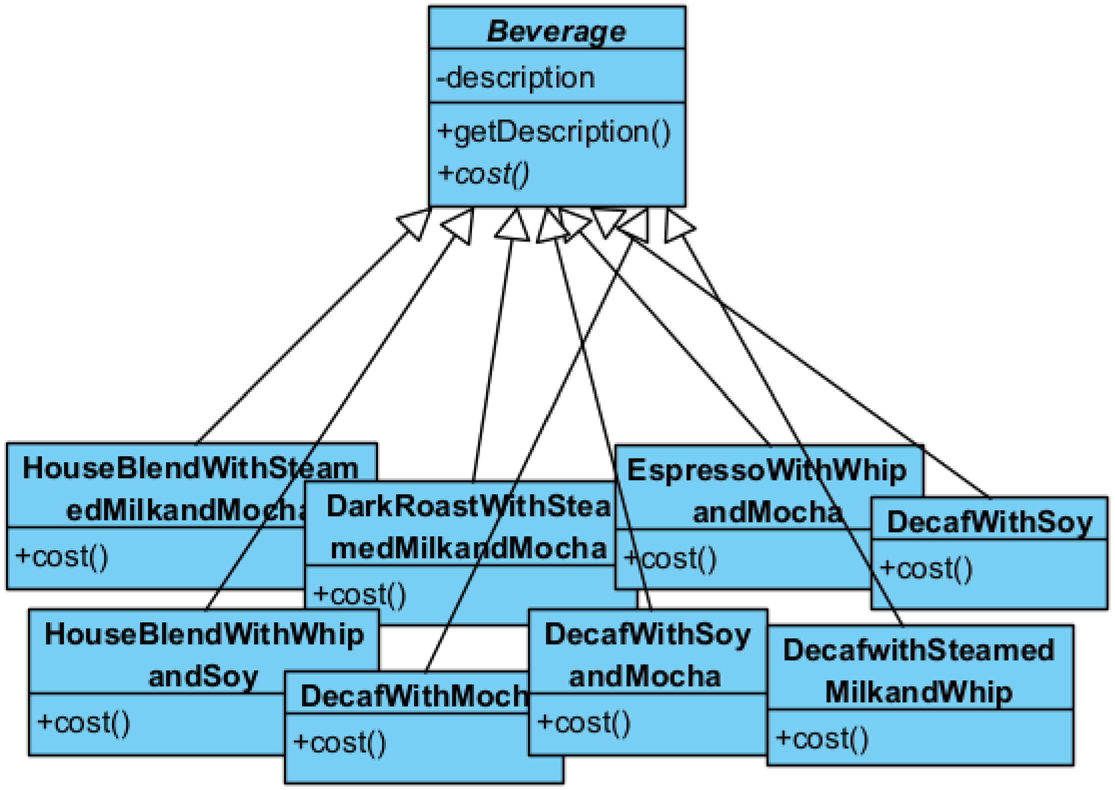
Adding condiments to coffee

- In addition to the coffee, you can also ask for condiments like *steamed milk*, *soy*, *mocha*, and have it all topped with the *whip*. Of course, each of these bears some cost too. So StarBuzz <u>need to build these into their systems</u>.

- What could be your design to obtain the cost? E.g., How to represent the price of a cup of *decaf with mocha and whipped*.

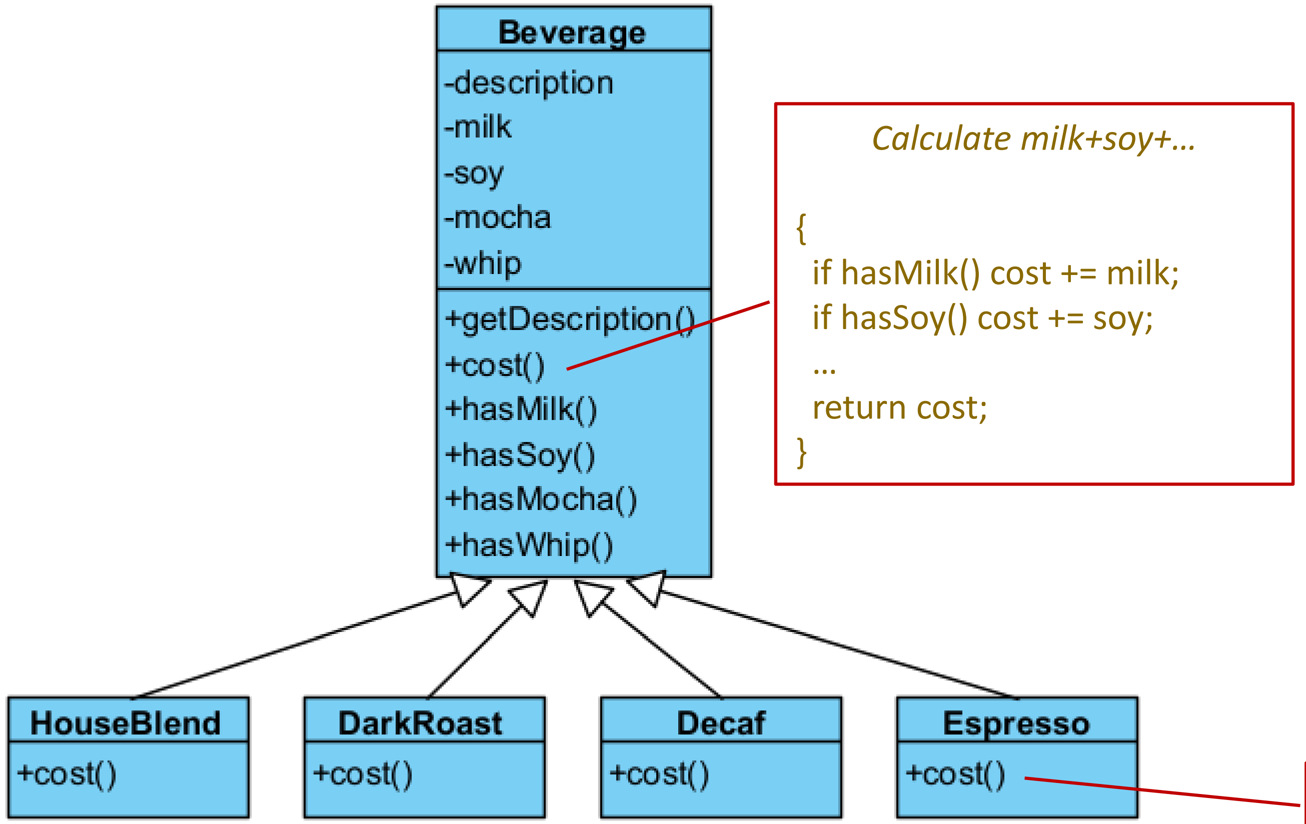
Image Credit: Rawpixel.com/Shutterstock.com

Initial design



- Any comments about the maintainability of this design?

Can’t just use variables + inheritance?



Calculate milk+soy+…

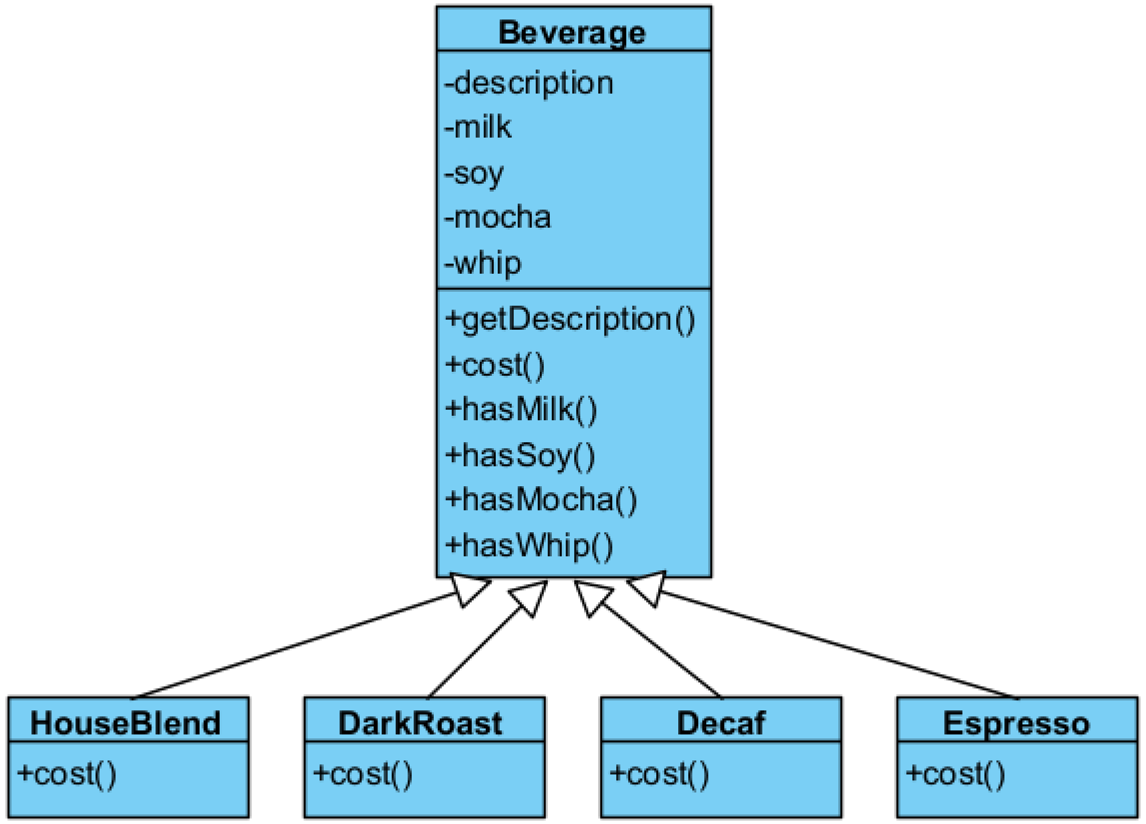
```
{
  if hasMilk() cost += milk;
  if hasSoy() cost += soy;
  …
  return cost;
}
```

Calculate super.cost()+this coffee’s cost

- Any comments about the maintainability of this design?

Can’t just use variables + inheritance?

- If condiment price changes, need to alter the existing code, namely base class Beverage’s cost(). *– you don’t want to change existing code here and there, and often.*
- Adding new condiments (e.g. caramel) will also force change on Beverage’s cost(). *– others may not know this linkage, they may just add a Caramel subclass and leave.*
- What if a customer wants a double mocha? *– quite a challenge, may bring a much complicated cost() method.*

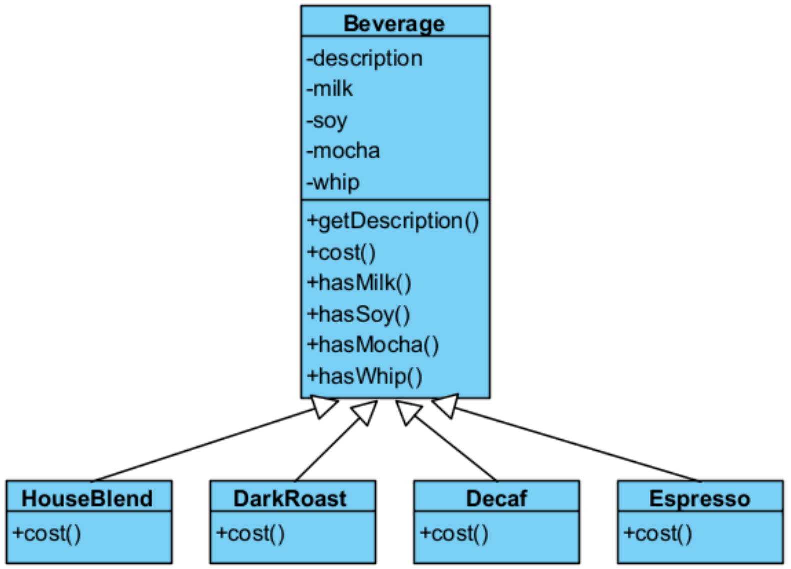


University of Nottingham
UK | CHINA | MALAYSIA

The Open-Closed Principle

Design Principle:
Classes should be open for extension, but closed for modification.

- The variable+inheritance design needs alteration of existing code (mainly Beverage's cost()) during maintenance, which may be prone to errors.
- Can we easily extend the classes to incorporate new behaviors *without modifying existing code?*
- If so, our design is resilient and flexible to take on new functionality to meet changing requirements.

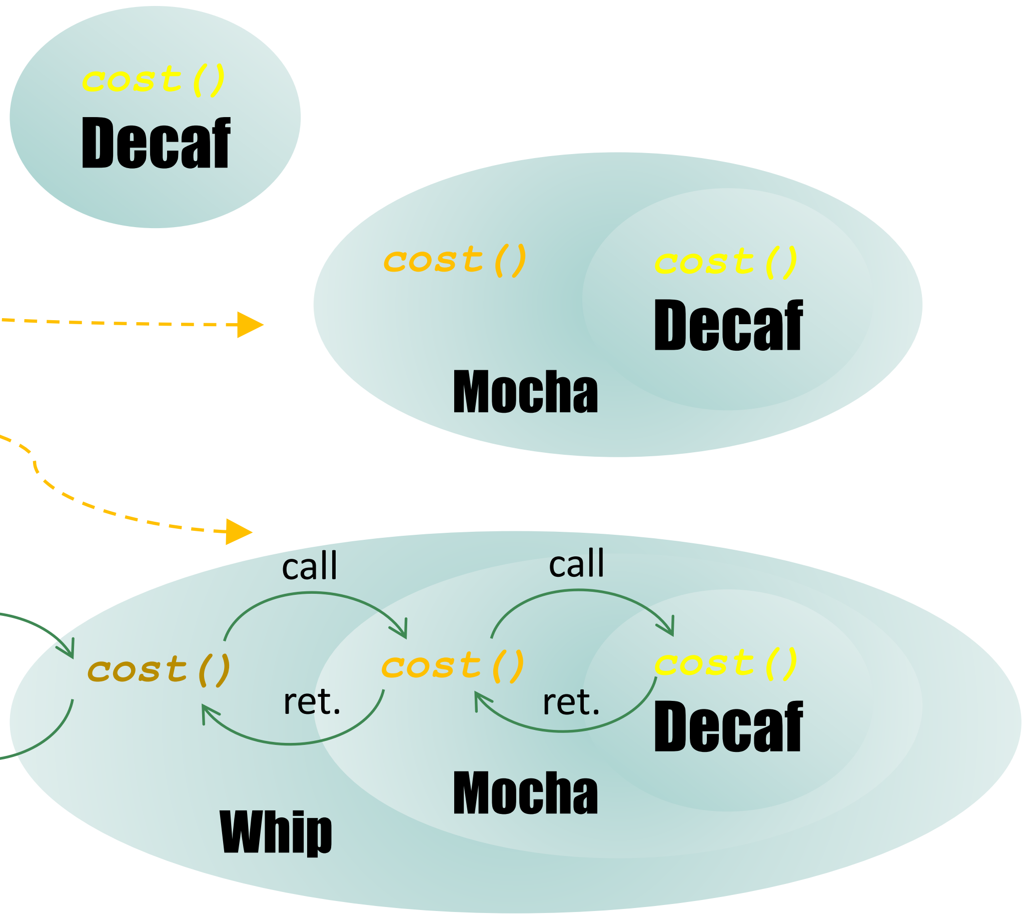


'Decorating' the coffee

- What is the price of a cup of *decaf with mocha and whipped*.
- Take a Decaf object
- **Decorate** it with a Mocha object
- **Decorate** it with a Whip object
- Call the *cost()* method and rely on **recursion** to add on the condiment costs.

First, call the cost() on the outermost decorator, Whip

Final cost will be returned by adding Decaf, Mocha, and Whip along the way

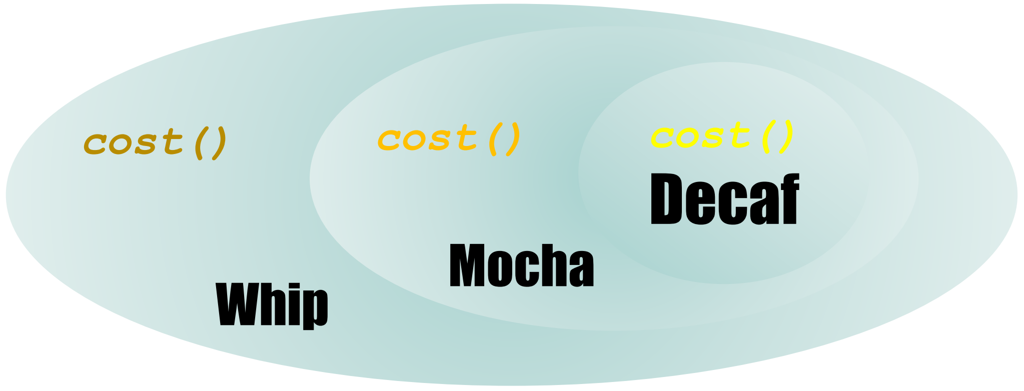


The Decorator pattern

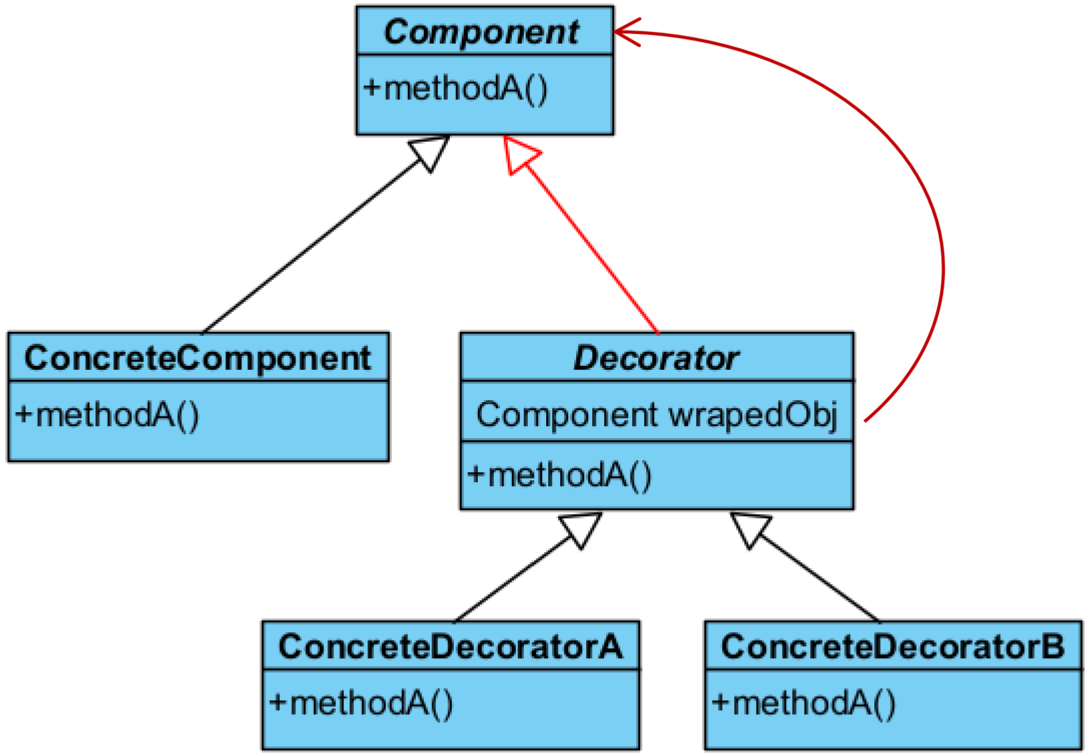
Design Pattern:

The Decorator Pattern attaches additional behaviors to an object dynamically, by placing the object inside special wrapper objects (decorators) that contain the behaviors.

- Applied under combinatorial explosion of subclasses.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.
  - E.g., *decaf with mocha and whipped and soy and another mocha...*
- Wrapped objects are *closed for modification, but open for functionality extension* by decorators.

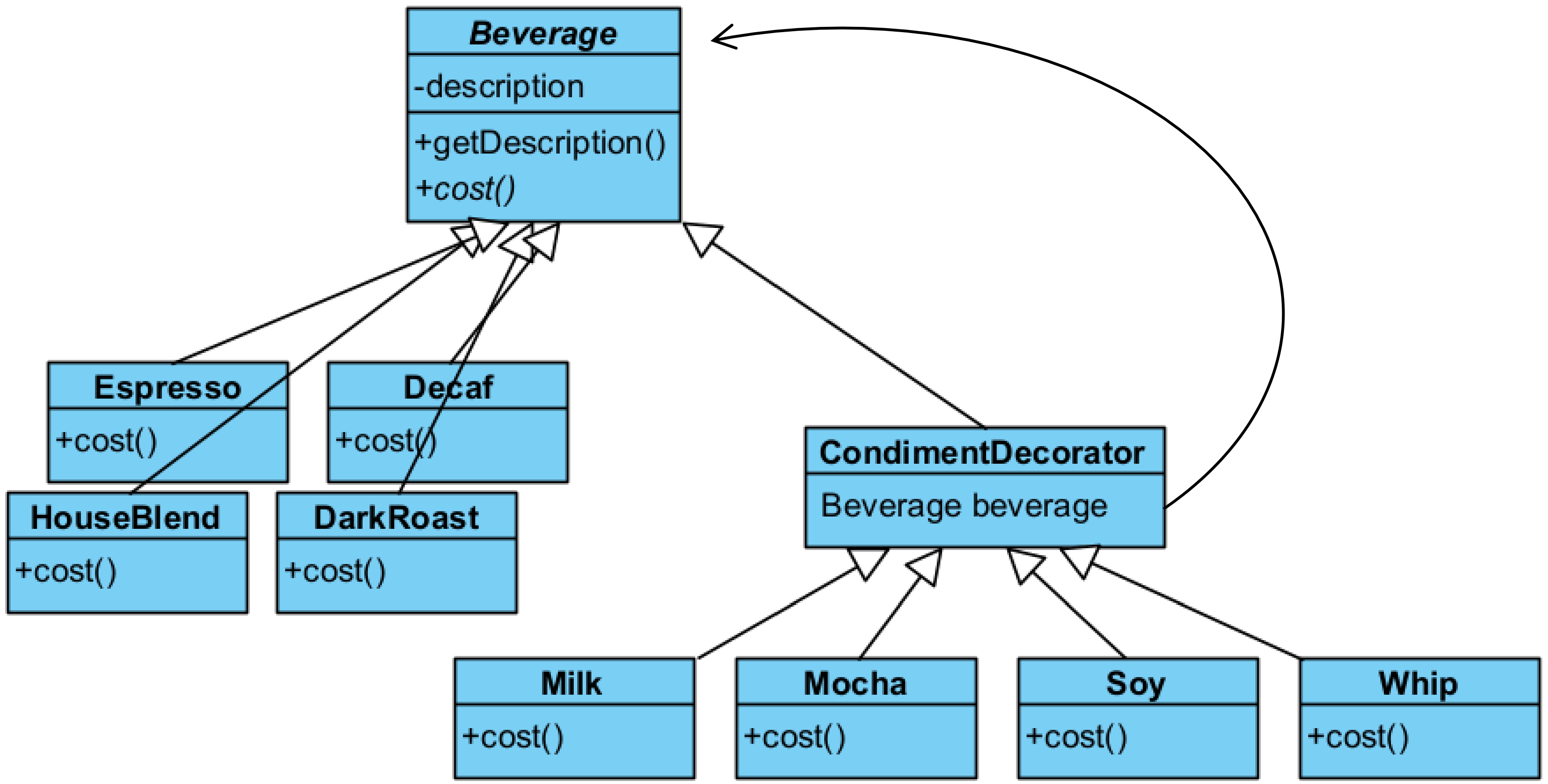


Realizing the Decorator pattern

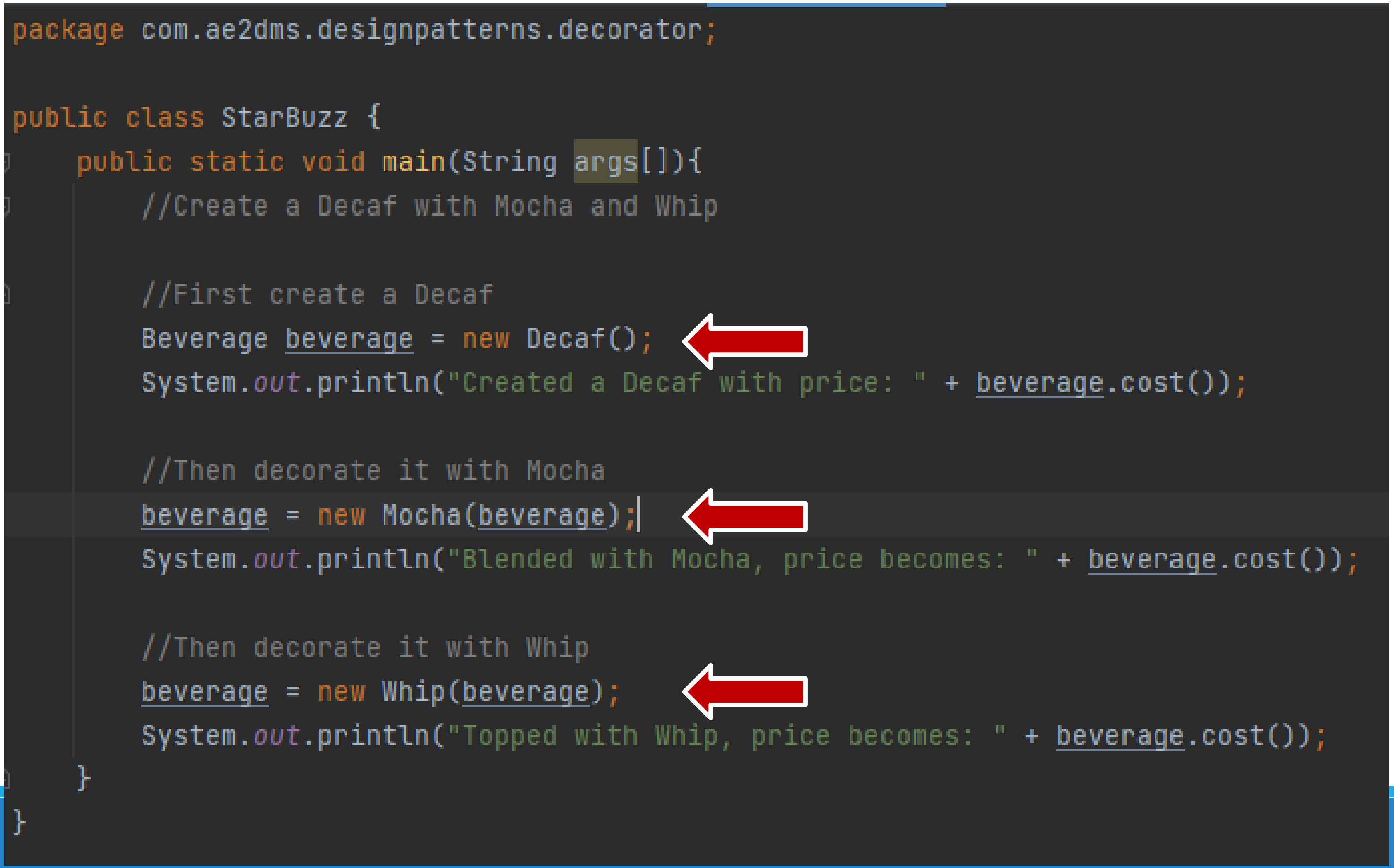


- Decorator should be a *subclass of the Component*, same as ConcreteComponent.
- Decorator *contains a Component object* to wrap (passed from the Constructor).
- Decorator can add new behavior, however the new behaviors typically happens before or after (namely around) the wrapped behavior
  - E.g., cost() in Mocha adds new behavior to cost() in Decaf.

Decorating our Beverages

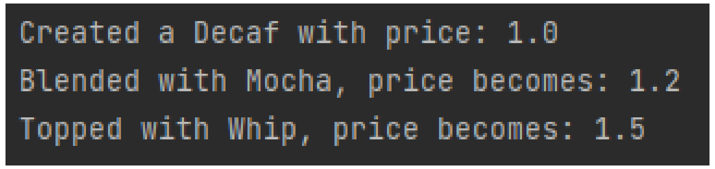


Make a Mocha Whipped Decaf at runtime



```java
package com.ae2dms.designpatterns.decorator;

public class StarBuzz {
    public static void main(String args[]){
        //Create a Decaf with Mocha and Whip

        //First create a Decaf
        Beverage beverage = new Decaf();
        System.out.println("Created a Decaf with price: " + beverage.cost());

        //Then decorate it with Mocha
        beverage = new Mocha(beverage);
        System.out.println("Blended with Mocha, price becomes: " + beverage.cost());

        //Then decorate it with Whip
        beverage = new Whip(beverage);
        System.out.println("Topped with Whip, price becomes: " + beverage.cost());
    }
}
```

Output:

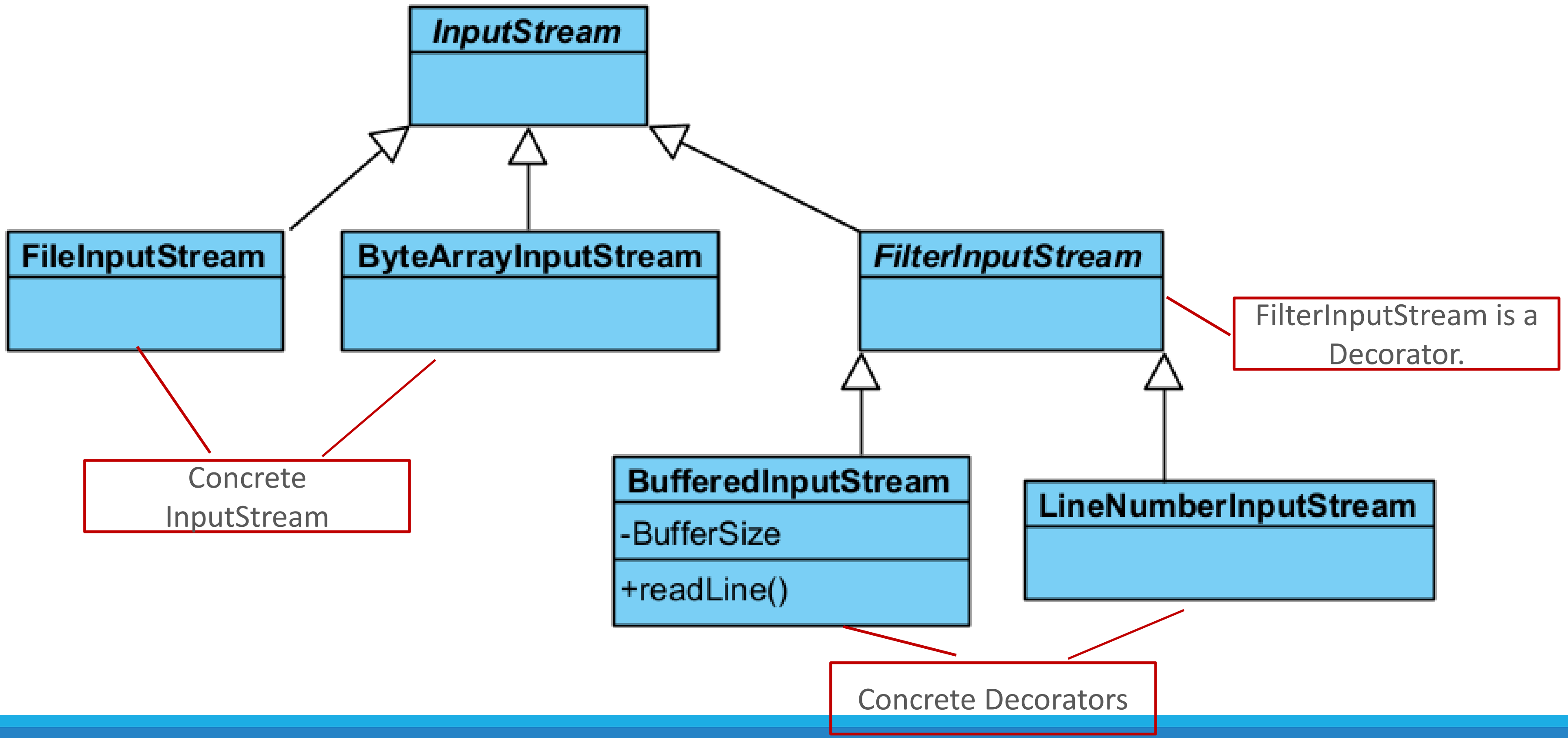


```
Created a Decaf with price: 1.0
Blended with Mocha, price becomes: 1.2
Topped with Whip, price becomes: 1.5
```

Note how the polymorphism is applied at runtime!

Real World Decorators: Java I/O

```
InputStream in = new BufferedInputStream(new FileInputStream("input.txt"));
```



New behaviors in **BufferedInputStream**:
(1) Buffers input to improve performance;
(2) read a line at a time

New behaviors in **LineNumberInputStream**:
(1) Can count the line numbers as it reads data.

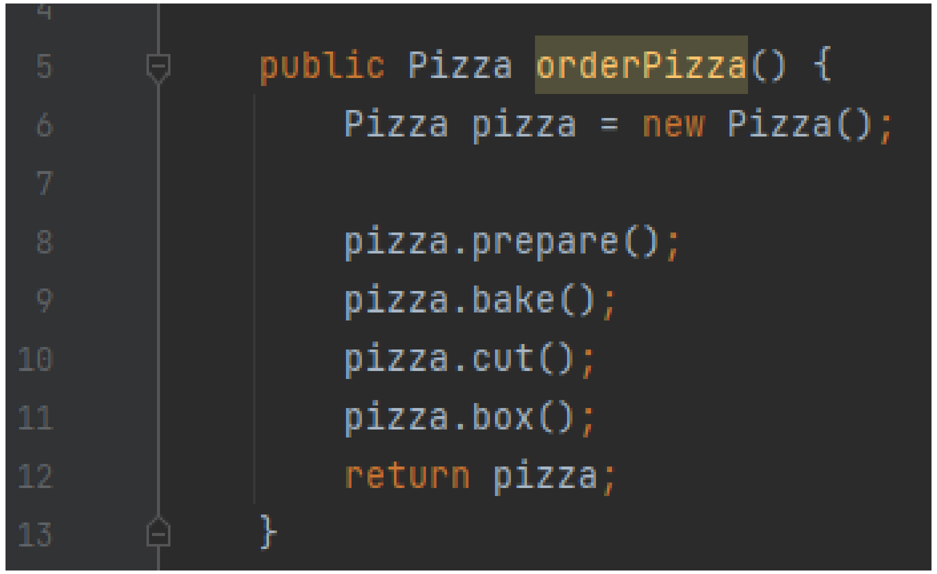
This week's outline

- SimuDuck: First impression on maintenance
  - How hard is to maintain your code
  - First several design principles
  - First design pattern: Strategy pattern
- More pattern stories:
  - StarBuzz: decorate your coffee
  - Arabina: needs a pizza factory
- More principles and patterns
  - SOLID Principle
  - Design patterns in general
- Summary

The Addison Wesley Signature
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
software development 15th annual productivity award
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts
O'REILLY
Head First Design Patterns
A Brain-Friendly Guide
10th Anniversary Updated for Java 8
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Arabina cooks decent pizza

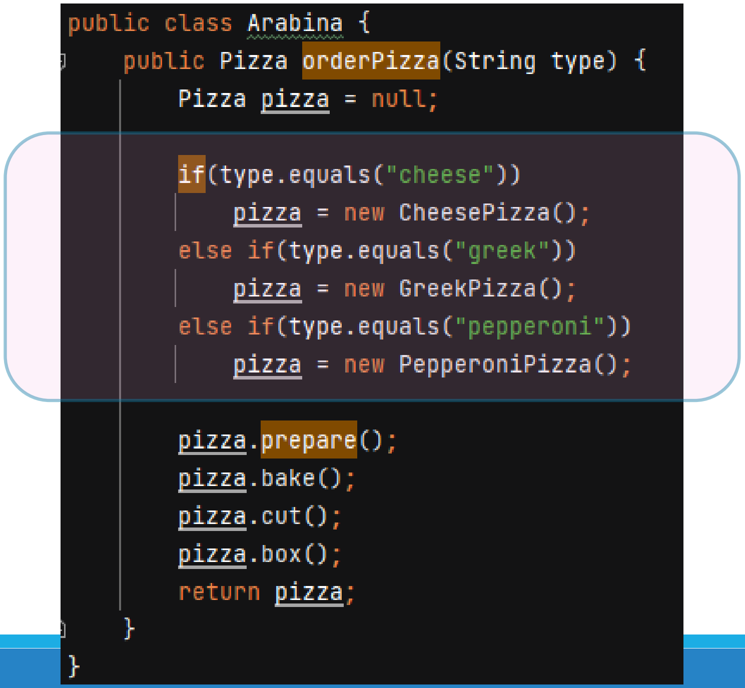
- Arabina makes quite decent pizza to students in UNNC, although may not be truly authentic…
- You are writing code for Arabina to realize the pizza ordering process, like below.



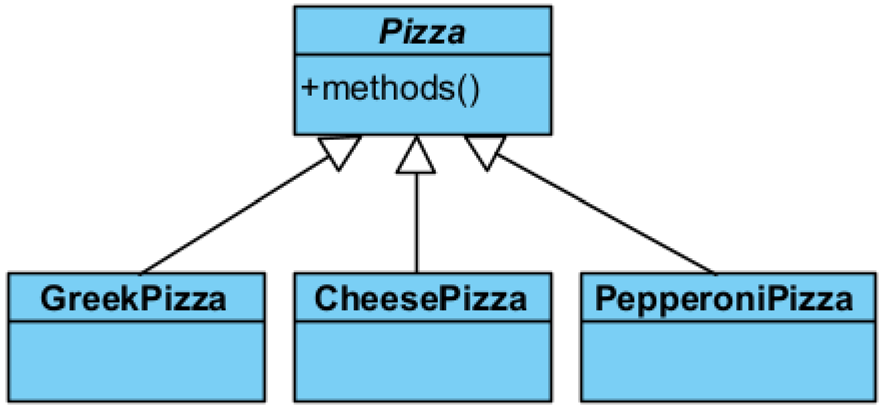
```java
    public Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
```

Arabina cooks decent pizza

- But Arabina offers more than one types of pizzas.
- So you need to extend the code to determine the appropriate type of pizza to be ordered.



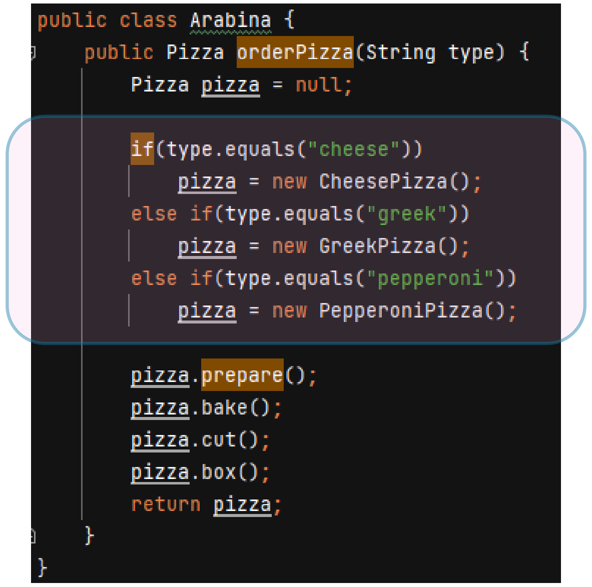
```java
public class Arabina {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        if(type.equals("cheese"))
            pizza = new CheesePizza();
        else if(type.equals("greek"))
            pizza = new GreekPizza();
        else if(type.equals("pepperoni"))
            pizza = new PepperoniPizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```



Single Responsibility Principle

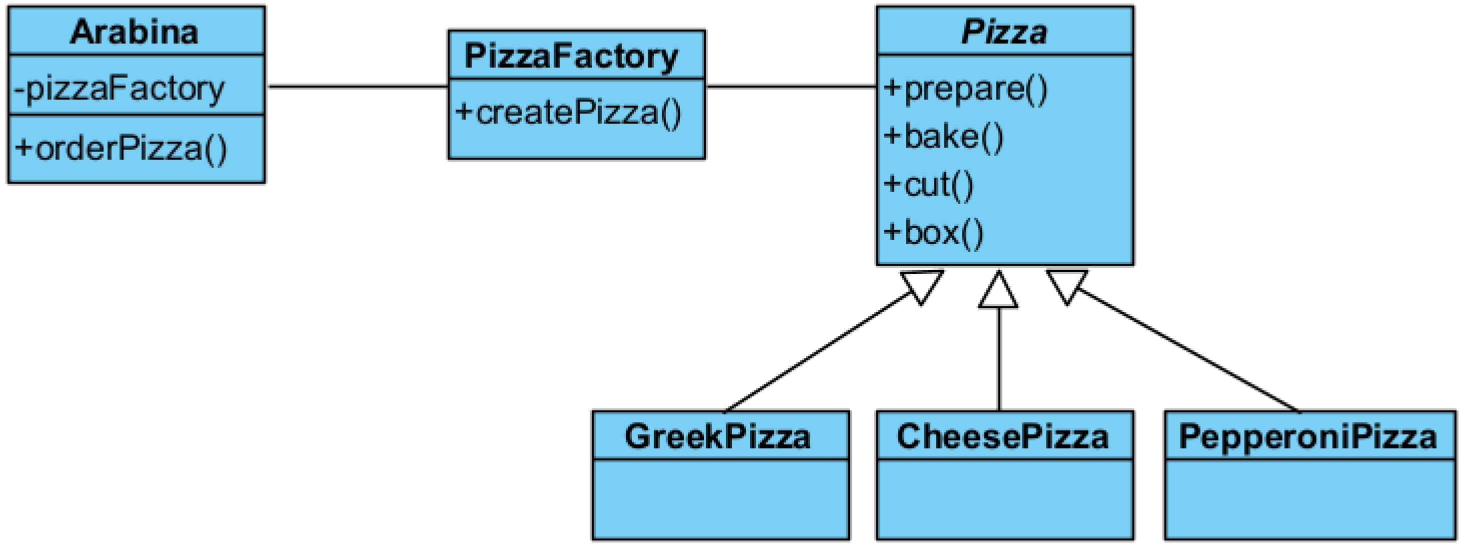
Design Principle:
A code module (a class, method, etc.)
should have one and only one responsibility.

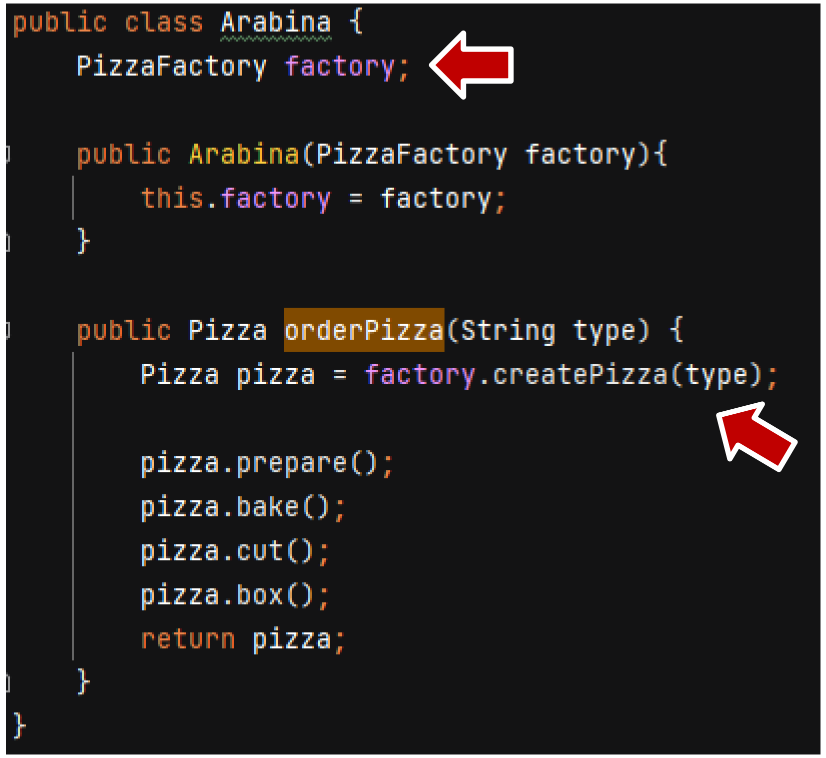
- The `orderPizza()` method should *focus* on the process of pizza ordering. Although the pizza creation logic is part of it, if it becomes cumbersome, it dilutes the focus of the ordering process, and is hard to maintain.
  - Better separate it out.
- What if later needs to add or remove a certain type of pizza? This part needs to be changed over and over later.



```java
public class Arabina {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        if(type.equals("cheese"))
            pizza = new CheesePizza();
        else if(type.equals("greek"))
            pizza = new GreekPizza();
        else if(type.equals("pepperoni"))
            pizza = new PepperoniPizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Defining a simple Pizza Factory





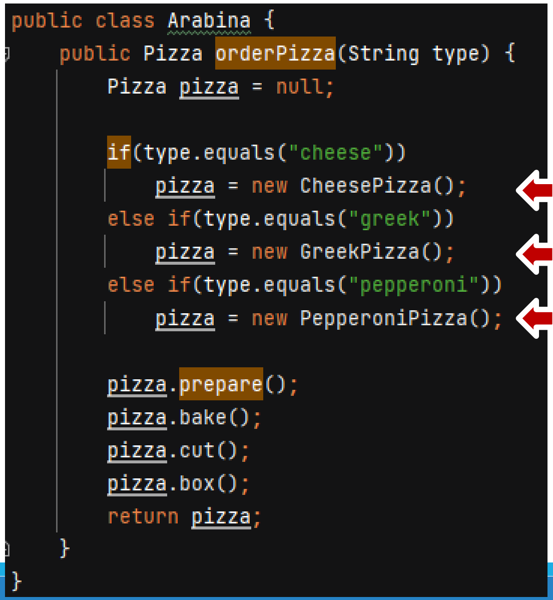
```
public class Arabina {
    PizzaFactory factory;

    public Arabina(PizzaFactory factory){
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

- The burden of 'if-else' checking and creating different types of Pizzas is now moved to the PizzaFactory class.
  - That is, the burden of maintenance (adding, removing specific pizza types) is separated to PizzaFactory class.
  - Remember our first principle: Identify the aspects of your application that vary, and separate them from what stays the same.

Dependency Inversion Principle

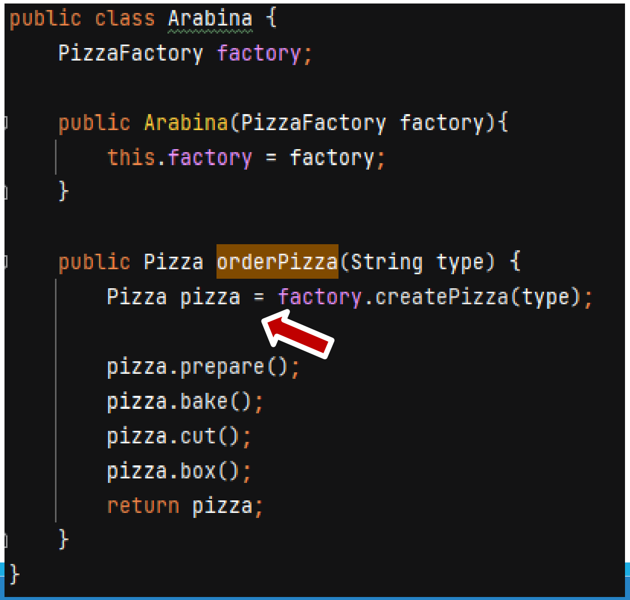
Design Principle:

Depend upon abstractions. Do not depend upon concrete classes.



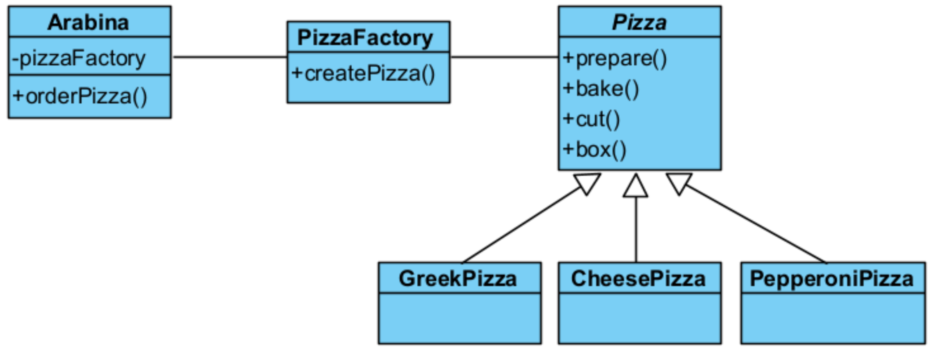
```
public class Arabina {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        if(type.equals("cheese"))
            pizza = new CheesePizza();
        else if(type.equals("greek"))
            pizza = new GreekPizza();
        else if(type.equals("pepperoni"))
            pizza = new PepperoniPizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

- Bad design: the Arabina class depends on *concrete* Pizza subclasses (CheesePizza, GreekPizza, PepperoniPizza)
- With the factory class defined, the Arabina class depends only on the *abstract* Pizza class.
- Check your code, whether one class is dependent on subclasses of another class…



```
public class Arabina {
    PizzaFactory factory;

    public Arabina(PizzaFactory factory){
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

The Factory Method Pattern?



Caution! This is **NOT YET** a Factory Method Pattern!

Let's complete the pizza story.

Arabina grows big, and franchises two concrete restaurants in Ningbo and Shanghai: **`NBArabina`** and **`SHArabina`**.

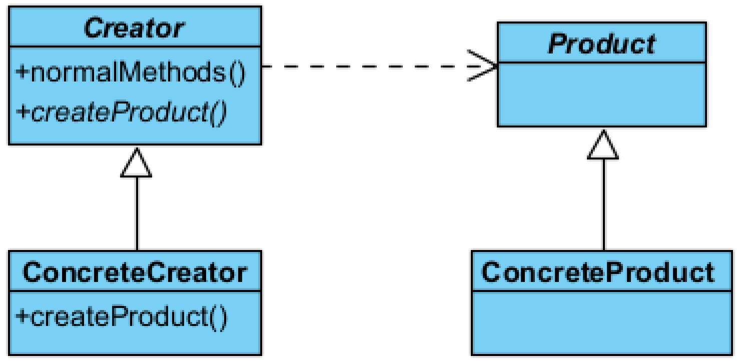
`NBArabina` only sells GreekPizza and CheesePizza, while **`SHArabina`** only sells PepperoniPizza.

The Factory Method Pattern

Design Pattern:

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

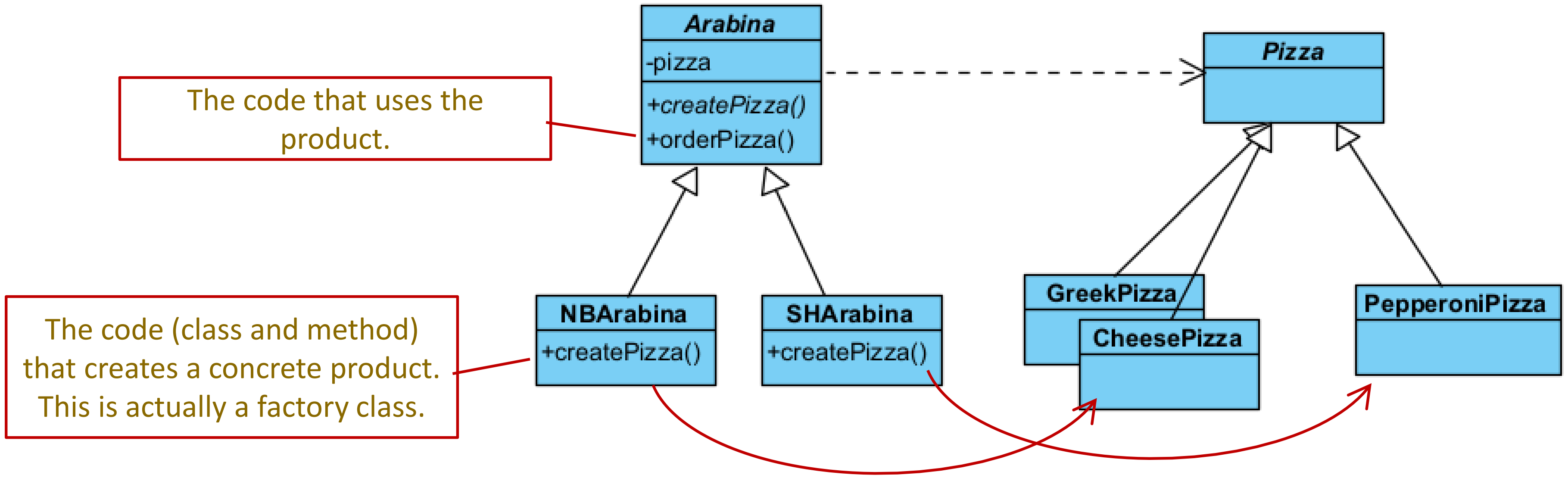
- The Factory Method *separates* product construction code from the code that actually uses the product.
- Therefore it is easier to extend the product construction code *independently* from the rest of the code.
- E.g., to add a new product type to the app, you only need to create/modify a creator subclass and override the factory method in it.



The Factory Method Pattern

Design Pattern:

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



This week's outline

- SimuDuck: First impression on maintenance
  - How hard is to maintain your code
  - First several design principles
  - First design pattern: Strategy pattern
- More pattern stories:
  - StarBuzz: decorate your coffee
  - Arabina: needs a pizza factory
- More principles and patterns
  - SOLID Principle
  - Design patterns in general
- Summary

O'REILLY
Head First Design Patterns
A Brain-Friendly Guide
10th Anniversary Updated for Java 8
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates
The Addison Wesley Signature
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

Design principles in this lecture

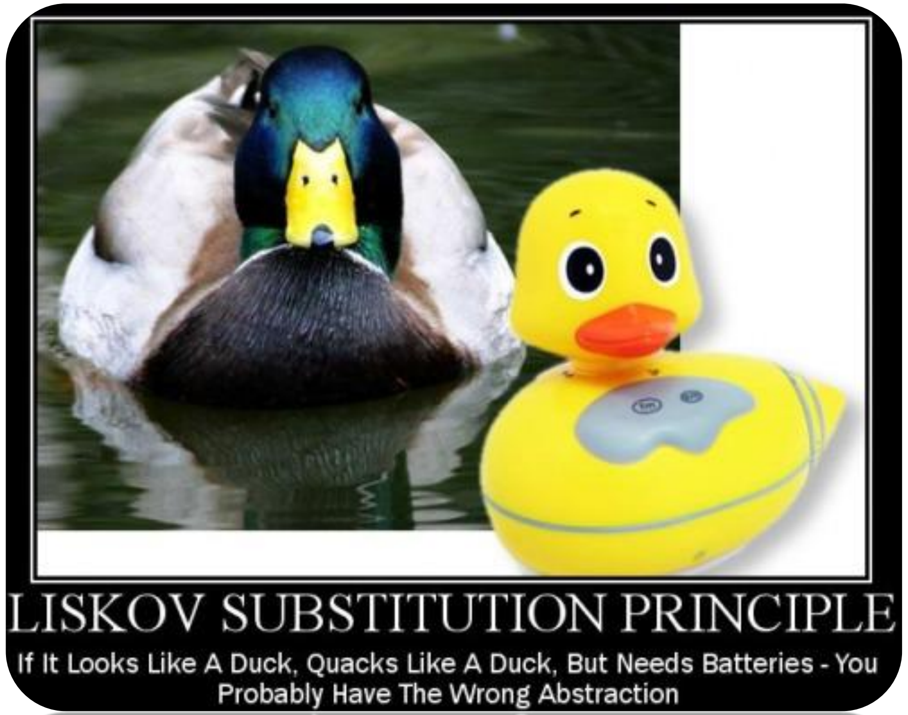
▪ Identify the aspects of your application that vary, and separate them from what stays the same.	← The fundamental one.
▪ Program to an interface, not an implementation.	
▪ Favor composition over inheritance.	
▪ A code module (a class, method, etc.) should have one and only one responsibility.	← Single Responsibility Principle
▪ Classes should be open for extension, but closed for modification.	← Open-Closed Principle
▪ Subtypes must be substitutable for their base types.	← Liskov's Substitute Principle
▪ Clients should not be forced to depend upon interfaces that they do not use.	← Interface Segregation Principle
▪ Depend upon abstractions. Do not depend upon concrete classes.	← Dependency Inversion Principle

Uncle Bob

- Robert C. Martin a.k.a. Uncle Bob.
- Agile and OOP guru.
- Derive principles to improve the state of the art of software craftsmanship.
  - "Writing clean code is what you must do to call yourself a professional. There is no reasonable excuse for doing anything less than your best."
  - Raised the well-known SOLID principles. [https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start]

Liskov's Substitution Principle

- Uncle Bob: "Subtypes must be substitutable for their base types."
  - Methods that use references to base classes must be able to use objects of derived classes without knowing it.
  - The "is-a" technique of determining inheritance relationships is simple and useful, but occasionally results in bad use of inheritance.


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

This is a way of ensuring that "inheritance" is used correctly.

Liskov's Substitution Principle

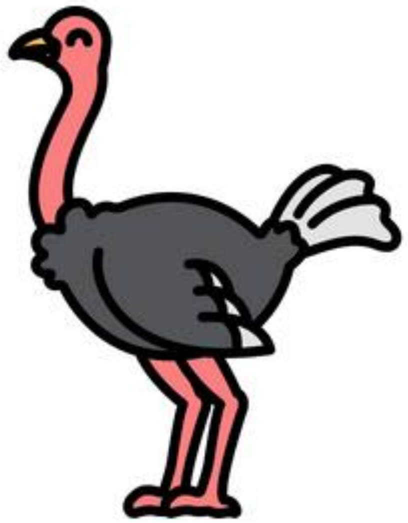
- A classic example from Uncle Bob by thinking about the functionality of classes.
  - There should be a separate class for birds that can't really fly, and Ostrich should inherit that.
  - The subclass needs to be able to properly use the methods of the superclass, else the abstraction is wrong.

Bad Design

```
public class Bird {
    public void fly() { }
}

class Parrot extends Bird { }

class Ostrich extends Bird { }
```

Better Design

```
public class Bird { }

class FlyingBirds extends Bird {
    public void fly() { }
}

class Parrot extends FlyingBirds { }

class Ostrich extends Bird { }
```

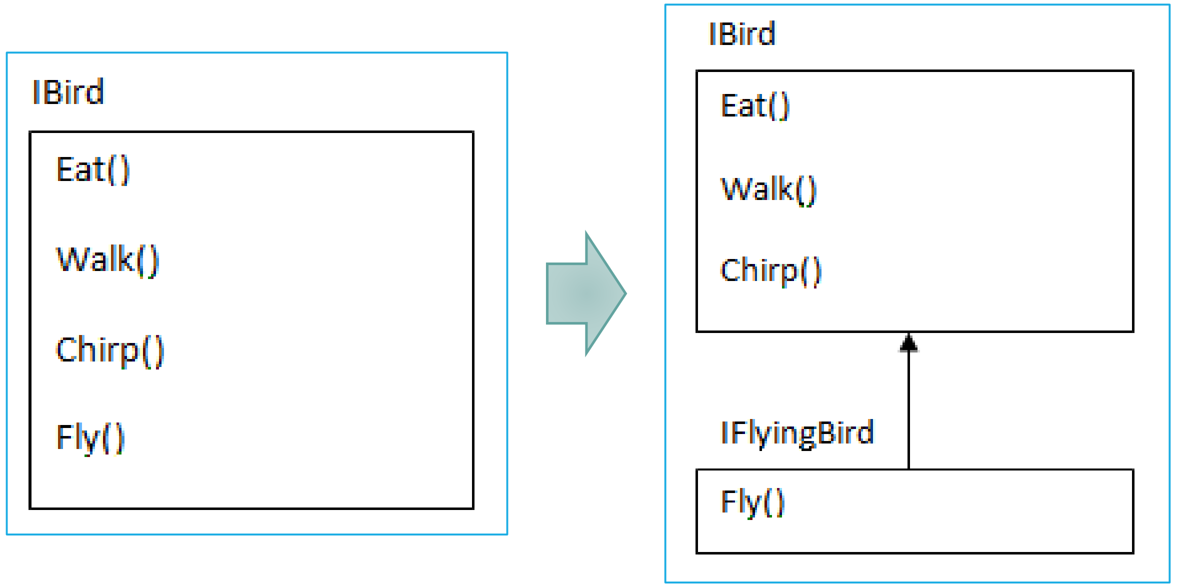


Interface Segregation Principle

- Uncle Bob: "Clients should not be forced to depend upon interfaces that they do not use."
  - If a class wants to implement the interface, it has to *implement all the methods*.
  - Fat interfaces should be broken down.
  - The principle ensures that interfaces are developed, so that each of them have their *own functionality* and thus, they are specific, easily understandable and reusable.
  - This is an extension of the single responsibility principle.
  - Nice example https://dzone.com/articles/solid-principles-interface-segregation-principle

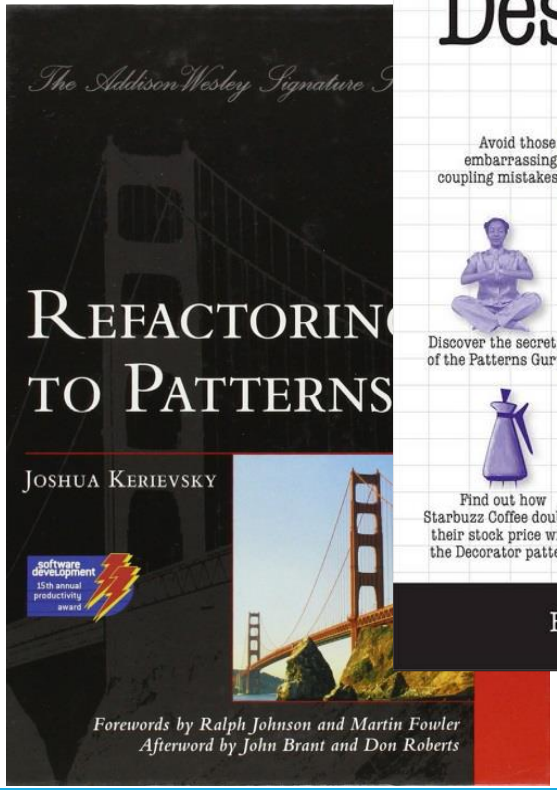
Interface Segregation Principle

- Example
  - A `SeaGull` would implement the `IFlyingBird` interface.
  - A `Ostrich` would implement the `IBird` interface.



This week's outline

- SimuDuck: First impression on maintenance
  - How hard is to maintain your code
  - First several design principles
  - First design pattern: Strategy pattern
- More pattern stories:
  - StarBuzz: decorate your coffee
  - Arabina: needs a pizza factory
- More principles and patterns
  - SOLID Principle
  - Design patterns in general
- Summary


The Addison Wesley Signature
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
software development 15th annual productivity award
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

O'REILLY
10th Anniversary Updated for Java 8
Head First Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Design Pattern: Someone has already solved your problems…

NOT EXACTLY CODE REUSE, WITH PATTERNS IT’S EXPERIENCE RESUE.

Design Patterns Formalize and Extend OO Principles

- What is a Design Pattern?
  - A pattern describes a problem which occurs repeatedly, and then describes the core of the solution to that problem in such a way that it can use the solution over and over again.
  - A pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.
  - A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.
- A pattern is a **code template** that you can re-use.
- But more than code re-use, it is **experience re-use**!

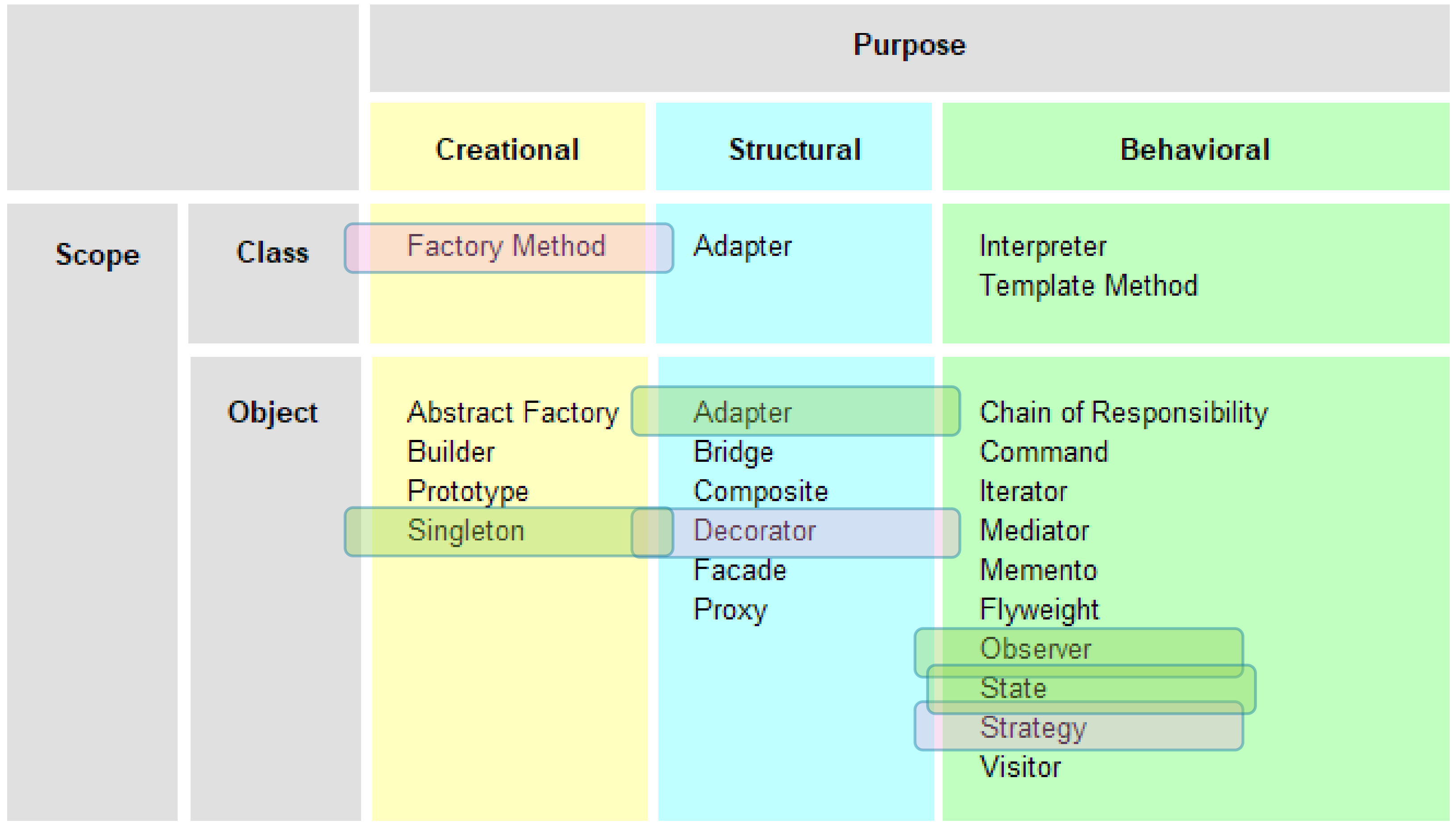
Design Patterns: 4 Essential Elements

1. **Name**: A handle that we can use to describe a design problem, its solution and the consequences in one or two words; having a vocabulary for patterns lets us talk about them with our colleagues and in our documentation.
2. **Problem**: Describes when to apply a pattern; sometimes the problem includes a list of conditions that must be met before it makes sense to apply a pattern.
3. **Solution**: Describes the elements that make up the design and their relationships.
4. **Consequences**: Describes the results and trade-offs of applying the pattern; critical for evaluating design alternatives.

Design Patterns: Organized in Two Ways

- Purpose: Reflects what the pattern does in 3 categories.
  - **Creational**: Concern the process of object creation.
  - **Structural**: Deal with the composition of classes and objects.
  - **Behavioral**: Characterize the way in which classes and objects interact and distribute responsibility.
- Scope: Specifies whether the pattern applies to classes or objects.
  - **Class**: These patterns deal with relationships between classes and sub-classes (which are fixed at compile time).
  - **Object**: Deal with object relationships (which can be changed at runtime).

Categorization



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

- Listed are 24 patterns which are either:
  - Creational
  - Structural
  - Behavioral
- This lecture will look at some of the most used patterns.

University of Nottingham
UK | CHINA | MALAYSIA

Creational Patterns

- **Abstract Factory**: Interface to create related objects without declaring the concrete class.
- **Builder**: The same construction process can create different representations.
- **Factory Method**: Defers instantiation to subclass.
- **Prototype**: Create new objects by copying the prototype.
- **Singleton**: Ensures that one and only one instance of a class is created.

University of Nottingham
UK | CHINA | MALAYSIA

Structural Patterns

Adapter: Provides compatible interfaces for classes that couldn’t work together otherwise.

Bridge: Decoupling of abstraction and implementation so that the two can be independent.

Composite: Put objects into a tree structure to represent the hierarchies.

Decorator: Lets individual instances have addition of dynamically adding new function.

Facade: Providing a unified interface to a set of interfaces.

Flyweight: Use sharing to support large numbers of complex objects.

Proxy: Provide a placeholder for another object to access.

University of Nottingham
UK | CHINA | MALAYSIA

Behavioural Patterns – 1

- **Chain of responsibility**: Gives more than one object the chance the handle a request.
- **Command**: Encapsulates a request as an object.
- **Interpreter**: Converts problems expressed in natural language into a representation.
- **Iterator**: Accesses to objects without exposing underlying representation.
- **Mediator**: Promotes loose coupling by preventing objects from referring to each other.

Behavioural Patterns – 2

- **Memento**: Captures and displays an object's internal state.
- **Observer**: Defines a one-to-many relationship so that when one object changes state, all its dependents are notified.
- **State**: Allows an object to alter its behaviour when its internal state changes.
- **Strategy**: Defines and encapsulates a family of algorithms; let the algorithms vary independently of who is using it.
- **Template Method**: Defines the skeleton of an algorithm, and let subclasses redefine certain steps without charging the structure of the algorithm.
- **Visitor**: Defines a new operation without changing the classes.

Just Enough Design Patterns to Get By

- How many design patterns do you really need to know?
  - The most useful should be known intimately;
  - The lesser used should be known well enough to be able to discuss them (even if implementation requires a quick refresher).
  - The remaining can be postponed until relevant (although not forgotten completely).
  - What experts say [https://www.drdobbs.com/architecture-and-design/just-enough-design-patterns-to-get-by/232200458]

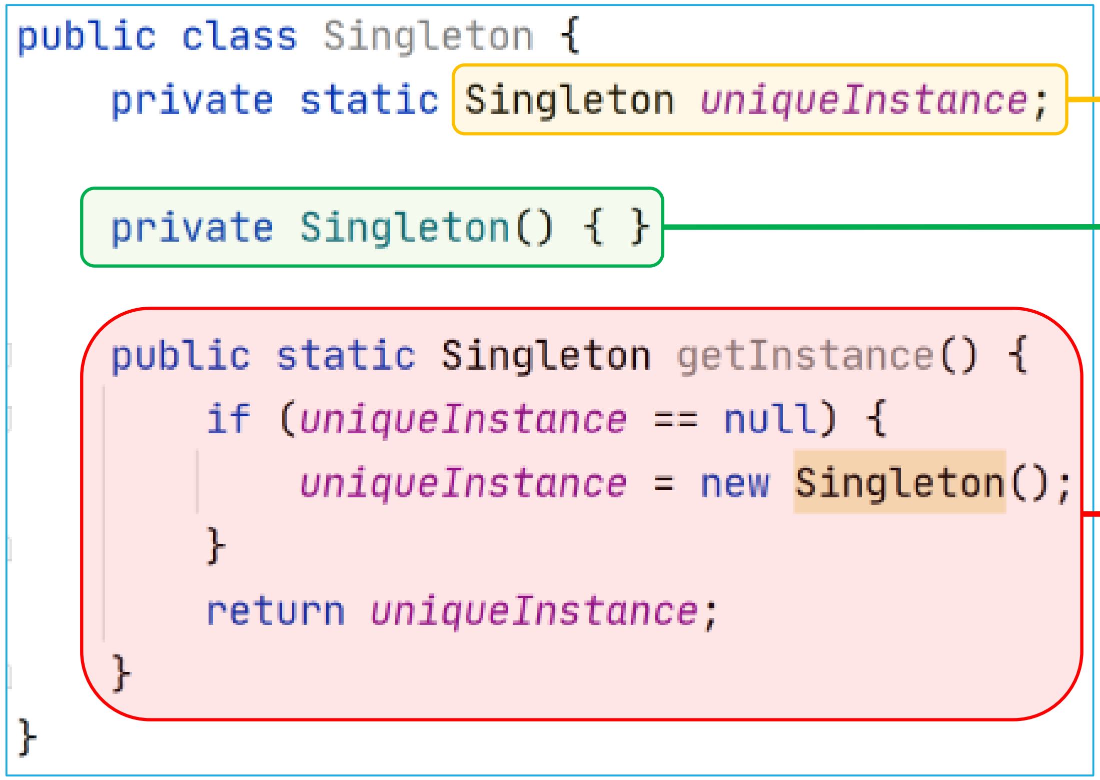
- Most useful design patterns.
  - Adapter; Command; Decorator; Façade; Factory method; Abstract Factory; Observer; Strategy;
  - State (works on agent-based simulation).
  - Factory method and Singleton (works on sensor data in cybersecurity).

University of Nottingham
UK | CHINA | MALAYSIA

Singleton Pattern

- Sometimes we want just a single instance of a class to exist in the system.
  - A car has only one driver.
  - An university is registered in only one address.
  - We want just one UI window manager or print spooler.
- We need to have that one instance easily accessible, and we want to ensure that additional instances of the class cannot be created.
- Has a single class in its class diagram.
  - BUT how do we ensure that there is **ONLY** one instance??
  - By using static methods and private classes.

Singleton Pattern



```java
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

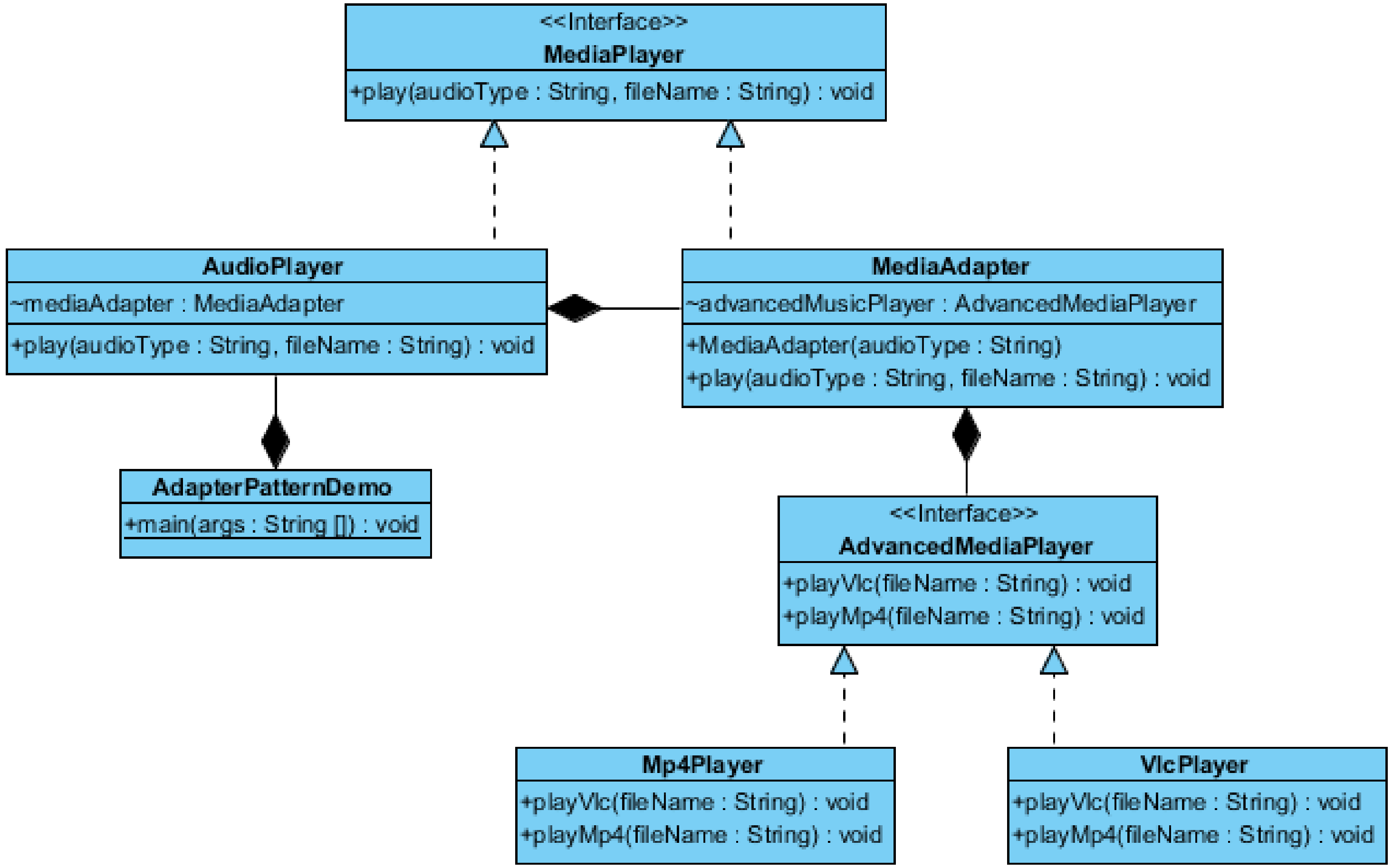
Static variable to hold single instance.

Private constructor.

Instantiate class and return a single instance of it.

Adapter Pattern

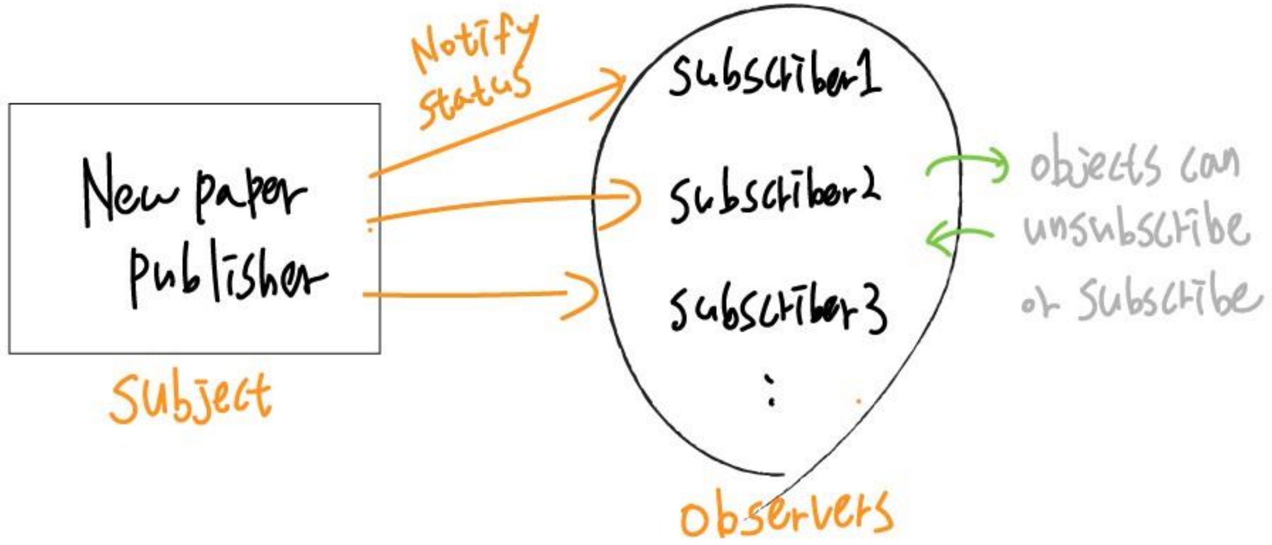
- This pattern involves a single class which is responsible to join functionalities of multiple independent or incompatible interfaces.
- An audio player device can play mp3 files only. But you want it to play video files such as vlc and mp4 format. The audio player then **adapt** a MediaAdapter which is a superclass of MP4 or Vlc players, for this purpose.



<<Interface>>
MediaPlayer
+play(audioType : String, fileName : String) : void
AudioPlayer
~mediaAdapter : MediaAdapter
+play(audioType : String, fileName : String) : void
MediaAdapter
~advancedMusicPlayer : AdvancedMediaPlayer
+MediaAdapter(audioType : String)
+play(audioType : String, fileName : String) : void
AdapterPatternDemo
+main(args : String []) : void
<<Interface>>
AdvancedMediaPlayer
+playVlc(fileName : String) : void
+playMp4(fileName : String) : void
Mp4Player
+playVlc(fileName : String) : void
+playMp4(fileName : String) : void
VlcPlayer
+playVlc(fileName : String) : void
+playMp4(fileName : String) : void

Observer Pattern

- Used when there is one-to-many relationship between objects such that if the subject is modified, its dependent objects are to be notified automatically.

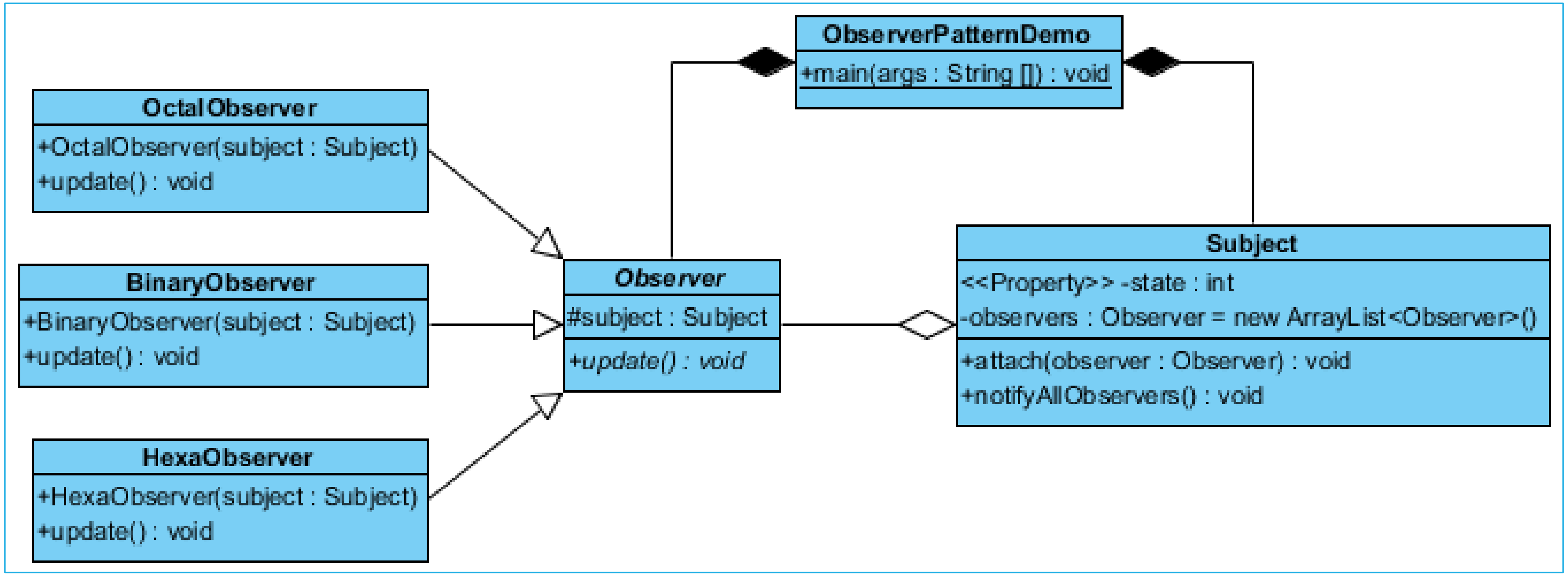


Loose Coupling

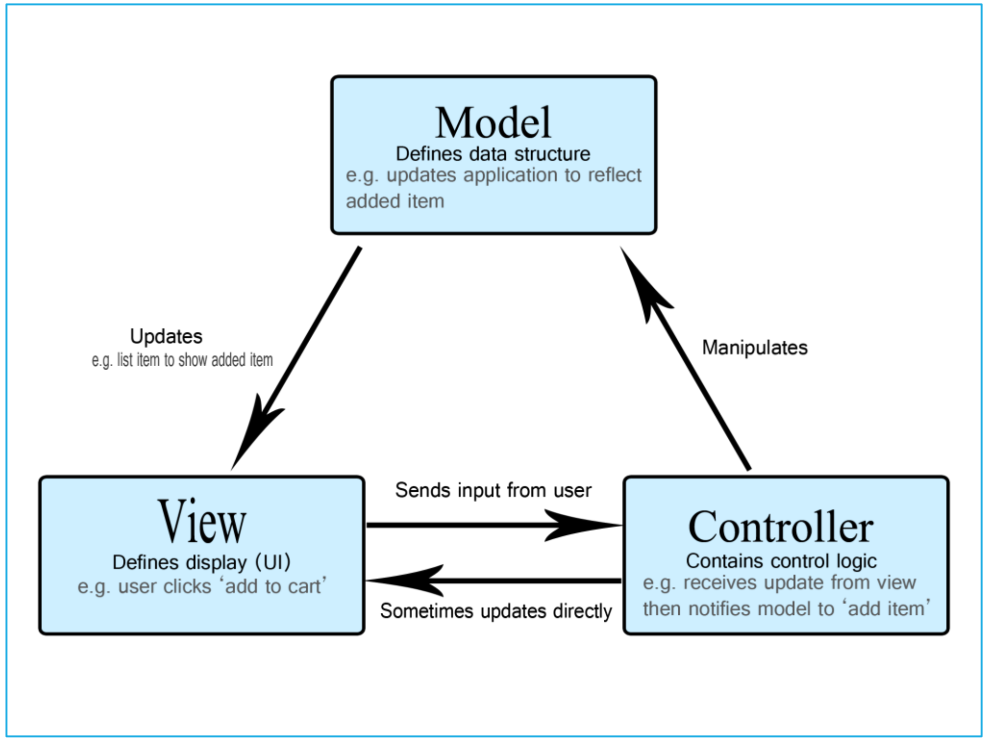
- Two objects are loosely coupled if they interact but have very little knowledge of each other (plastic friendship?!).
- Subject does not need to know the concrete class of the observer.
  - It only notifies to the observer interface.
- We can add new observers at any time irrespective of the state of the subject or object and without modifying the subject or object.
  - Subject register the observer or de-registers the observer.
- Changes to either subject or object do not affect the others.
- BTW, Java loves observer pattern.
  - Built-in into `java.util` and in JDK, in JavaBeans and Swing.
  - UI elements such as `Jbutton` (Typically called ‘XXListener’)
  - **Model-View-Controller** user interface framework.

Observer Pattern Example

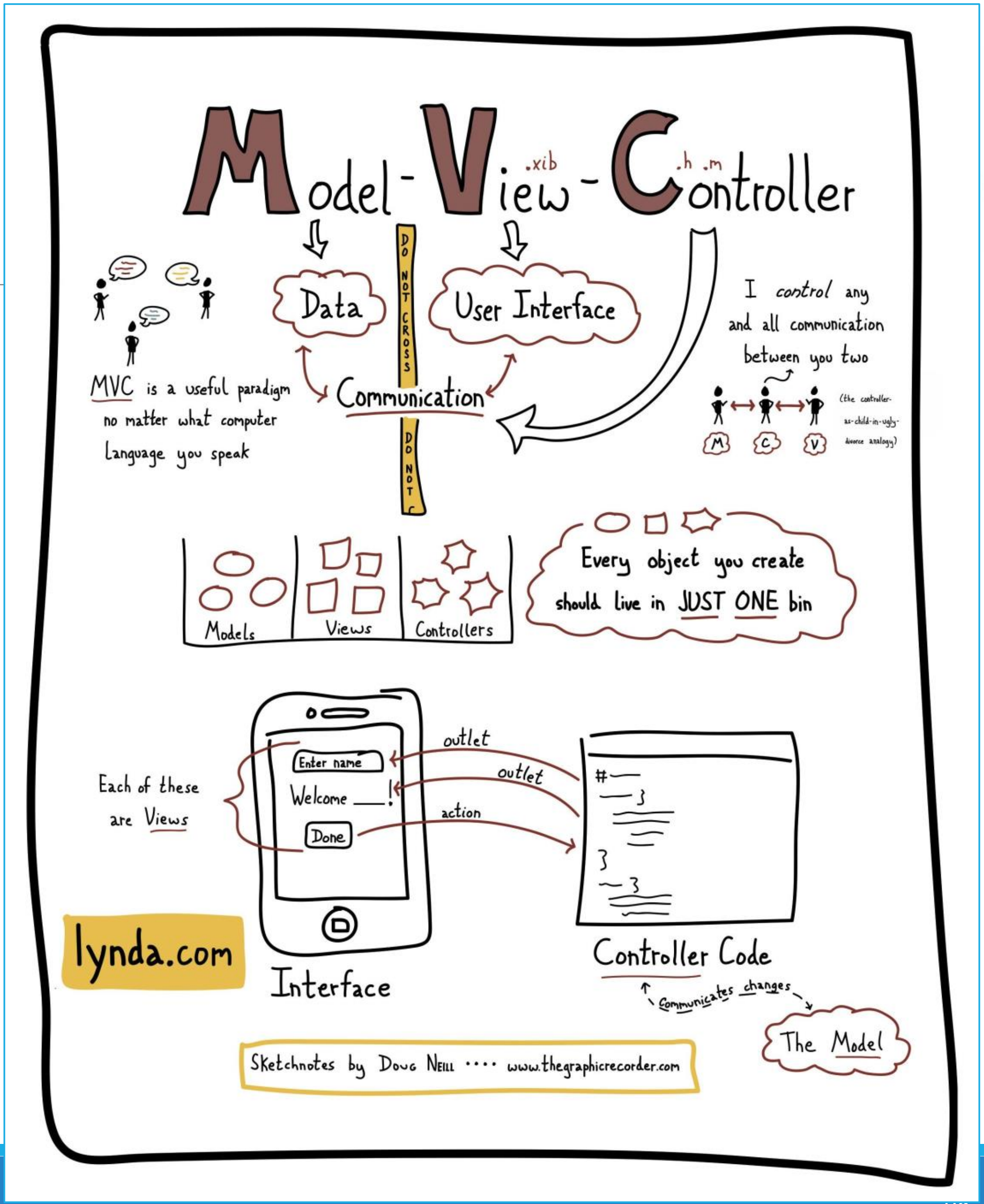
- Number Converter for Programmers.


ObserverPatternDemo
+main(args : String []) : void
OctalObserver
+OctalObserver(subject : Subject)
+update() : void
BinaryObserver
+BinaryObserver(subject : Subject)
+update() : void
HexaObserver
+HexaObserver(subject : Subject)
+update() : void
Observer
#subject : Subject
+update() : void
Subject
<<Property>> -state : int
-observers : Observer = new ArrayList<Observer>()
+attach(observer : Observer) : void
+notifyAllObservers() : void

MVC???



Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'



Model-View-Controller
.xib
.h .m
Data
User Interface
DO NOT CROSS
Communication
MVC is a useful paradigm no matter what computer language you speak
I control any and all communication between you two
(the controller-as-child-in-ugly-divorce analogy)
M
C
V
Models
Views
Controllers
Every object you create should live in JUST ONE bin
Each of these are Views
Enter name
Welcome ___!
Done
outlet
outlet
action
lynda.com
Interface
Controller Code
Communicates changes
The Model
Sketchnotes by Doug Neill ···· www.thegraphicrecorder.com

Summary

- SimuDuck, StarBuzz, and Arabina examples to reveal the importance of proper software maintenance methodologies.
- Design principles (including SOLID)
- Major Design Patterns
- **Note!** We used *Open-Closed* principle + *Decorator* pattern in the StarBuzz example. But don’t be misled that the open-closed principle can only be applied with the Decorator pattern, or vice-versa.
  - We should say that the Decorator is a powerful pattern to reflect the open-closed principle, whereas there could be other patterns doing so, and even, your coding (whether to employ any design patterns or not) should respect the open-closed principle whenever possible.
  - The same *apply to all* other principles and patterns.
- This week’s lab: Look at practical implementation of design patterns, including the **State Pattern**.

References

- SOLID Design Principles: https://thedavidmasters.com/2018/10/27/solid-design-principles/
- SOLID Principles: A Simple and Easy Explanation (PHP): https://hackernoon.com/solid-principles-simple-and-easy-explanation-f57d86c47a7f
- SOLID Principles – Simplified with Illustrations: https://levelup.gitconnected.com/solid-principles-simplified-with-illustrations-fe5265f68ec6
- Factory Pattern: http://csc.columbusstate.edu/woolbright/java/factory.html
- Printable Design Patterns Quick Reference Cards: http://blog.markturansky.com/archives/32
- Design Patterns in Java Tutorial: https://www.tutorialspoint.com/design_pattern/index.htm
- Source Making – Design Patterns: https://sourcemaking.com/

University of Nottingham
UK | CHINA | MALAYSIA

Put Your Mind in Maintenance Mode