Tasarım Şablonları 1. Hafta

24.02.2024

What is a Design Pattern?

- Design patterns are:
  - *Descriptions of communicating objects and classes*
  - *Objects and classes are customized to solve a general design problem in a particular context*

- GANG OF FOUR - GoF (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch) the writers of the book:

  Design Patterns: Elements of Reusable Object-Oriented Software

- Design patterns offer solutions to common application design problems.

Why Design Patterns?

- ❑ Simplifies object identification
- ❑ Simplifies system decomposition
- ❑ Proven & tested technique for problem solving
- ❑ Improves speed & quality of design / implementation
- ❑ Can be adapted / refined for specific system under construction

Design Patterns Classification

Two Categories of Design Patterns

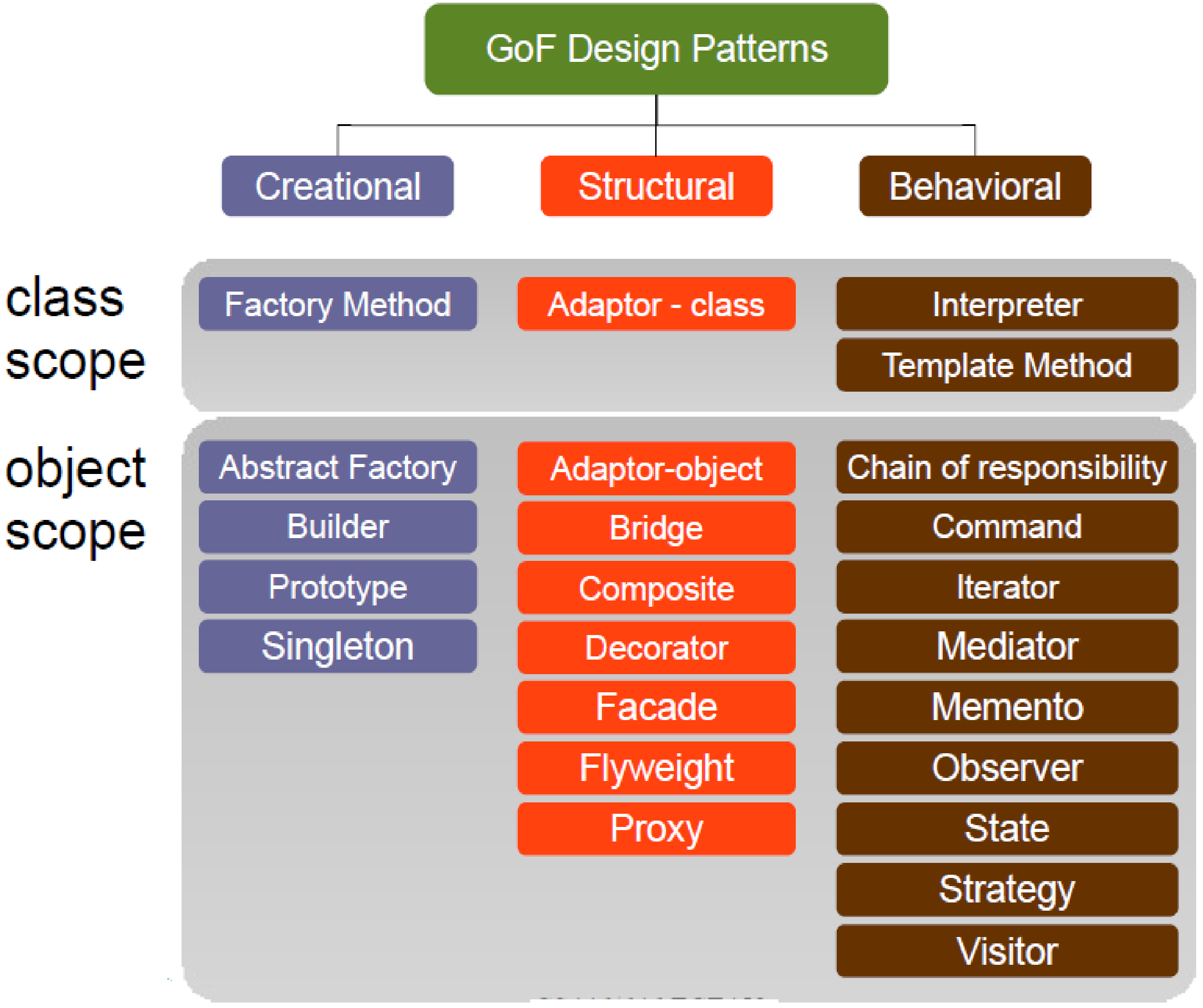
1)Class Scope

Relationship between classes & subclasses statically defined at run-time

2) Object Scope:

Object relationships (***what type?)***

Can be manipulated at runtime (***so what?)***



Design Patterns Classification

Creational class

- defers object creation to sub-classes (*factory method*)

Structural class

- inheritance to compose classes (*adapter*)

Behavioral class

- uses inheritance to describe flow of control, algorithms (*template*)

Creational object

- defers object creation to other objects (*abstract factory*)

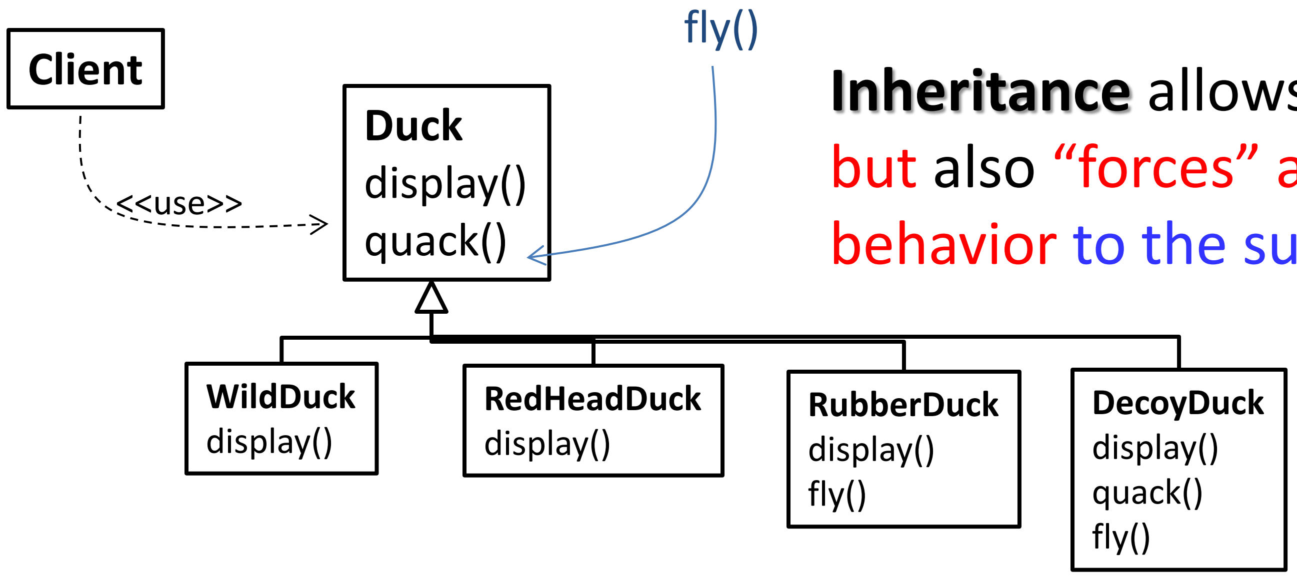
Structural object

- deals with object assembly (*adapter*)

Behavioral object

- group of objects working together to carry out a task (*iterator*)

Fundamental Object Oriented Properties
Example: Design of “simDuck”

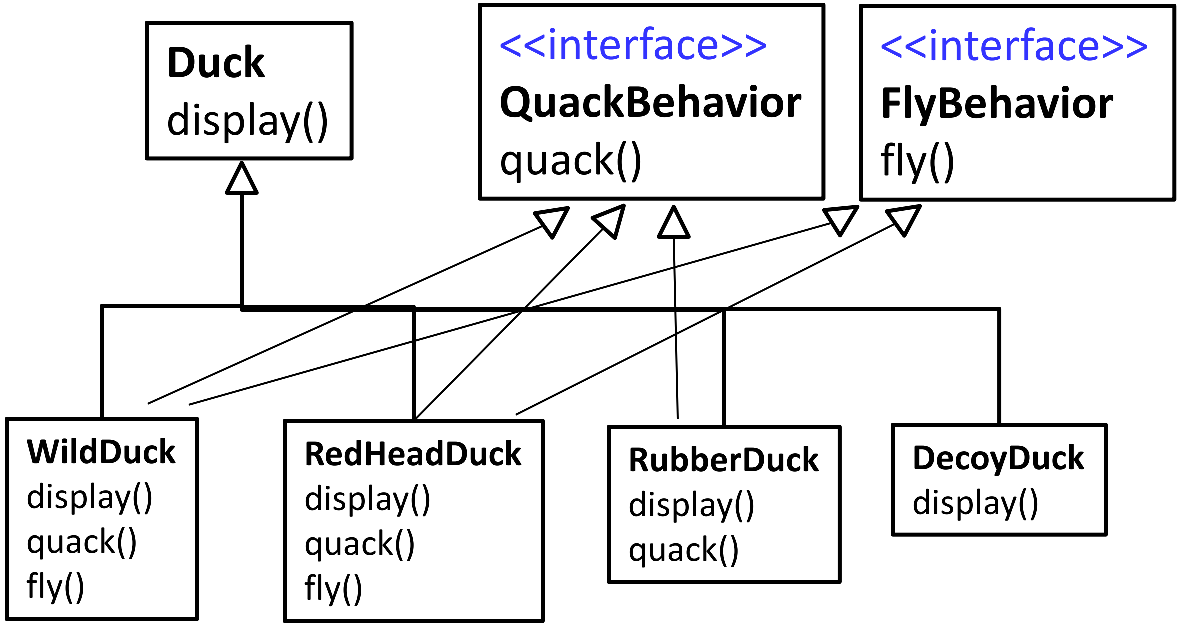


Inheritance allows you to reuse code, but also “forces” attributes and behavior to the subclasses.

In this example, rubber ***doesn’t fly****;*
decoy ***doesn’t quack and doesn’t fly.***

- ❑ Adding fly() method to Duck triggers need to modify elsewhere, in subclass RubberDuck,
  - ❖ Since RubberDuck doesn’t fly, you need it to override fly to do nothing
- ❑ We can do this, but if we have a lot of this kind of changes, we will have a Maintenance problem!

Fundamental of Object Orientes Properties
How about Factoring out to Multiple Interfaces?



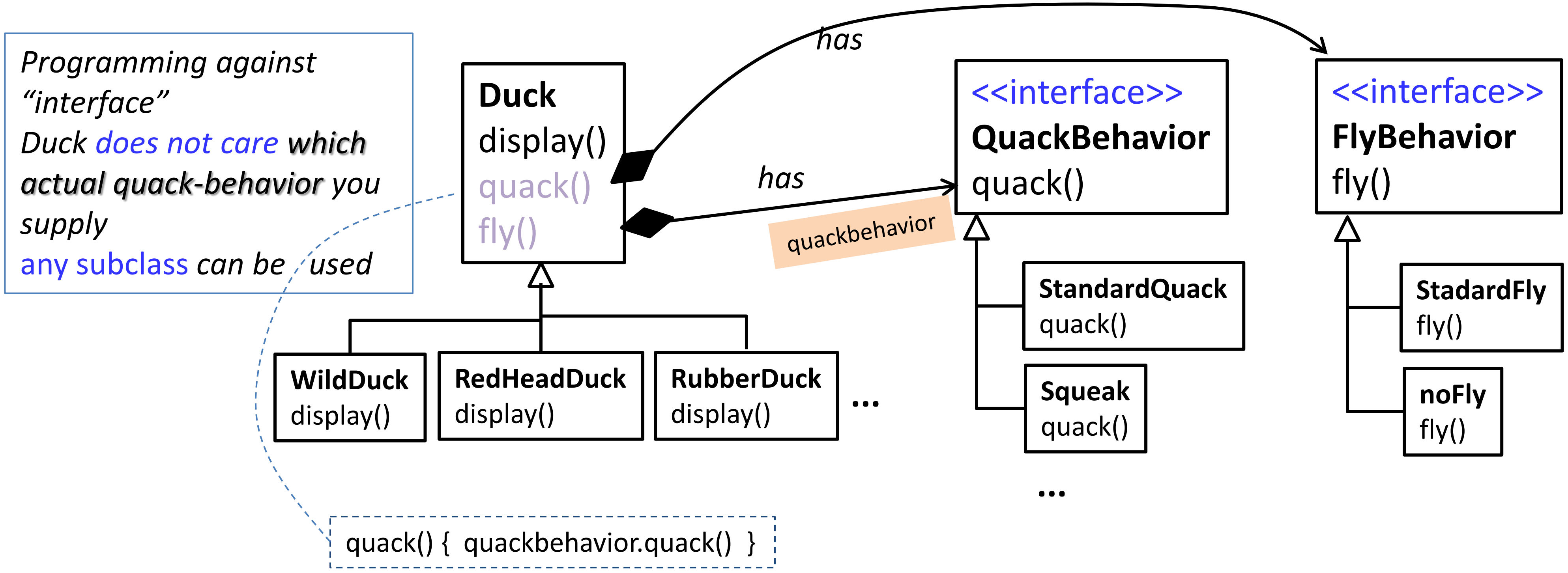
- You can lose code reuse; bad for maintenance
- Use multiple inheritance instead?
  - Not supported in all OO languages.

- If WildDuck and RedHeadDuck **fly the same way**, **quack the same way**, which now has to be duplicated in WildDuck and RedHeadDuck.
- **For possible** extensions **or** changes of functions:
  - **Adding** more **behavior** , eat() → you have to duplicate implementation in Wild and RedHead
  - **Changing behavior** of fly → you have to change 2 times, in WildDuck and RedHeadDuck

A Number of "Design Principles" are Proposed

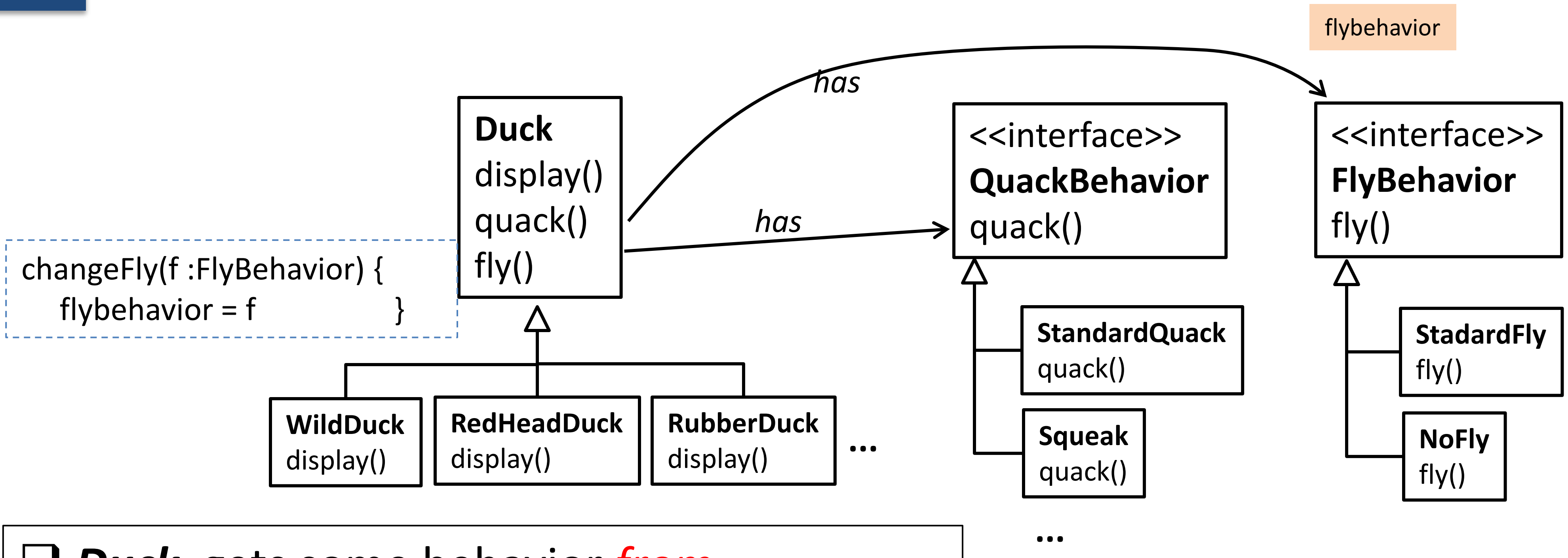
- Separate and encapsulate varying aspects from constant ones
- Program against "interface" rather than implementation
  - Relying on the signature of a **superclass** allow you the flexibility to replace its instances with those of subclasses
  - Suitable to extensibility of subclasses
- Consider using "composition" over inheritance as an option

Separating and Encapsulating quack and fly methods



Losing static checking: It is no longer possible to statically guarantee that every WildDuck will be able to quack, even if we make it so in the implementation.

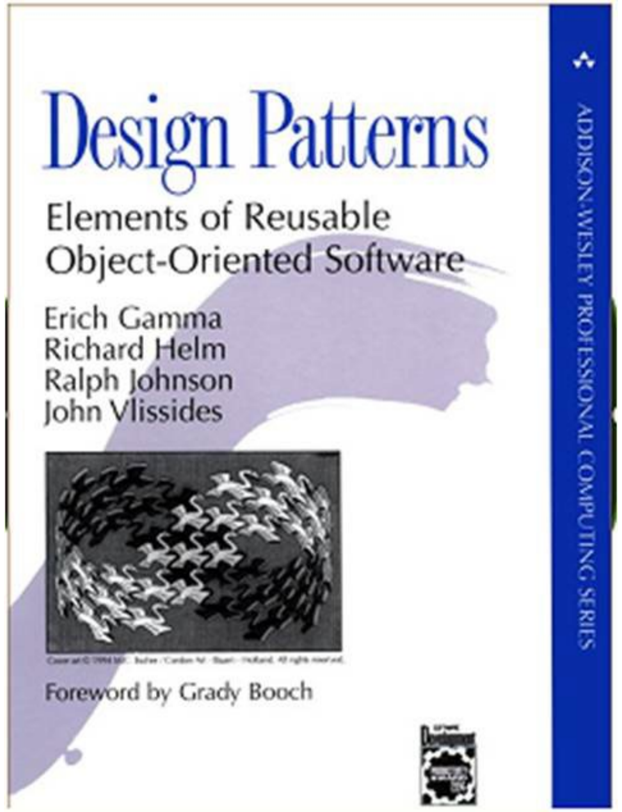
Composition instead of Inheritence
for quack() and fly() methods

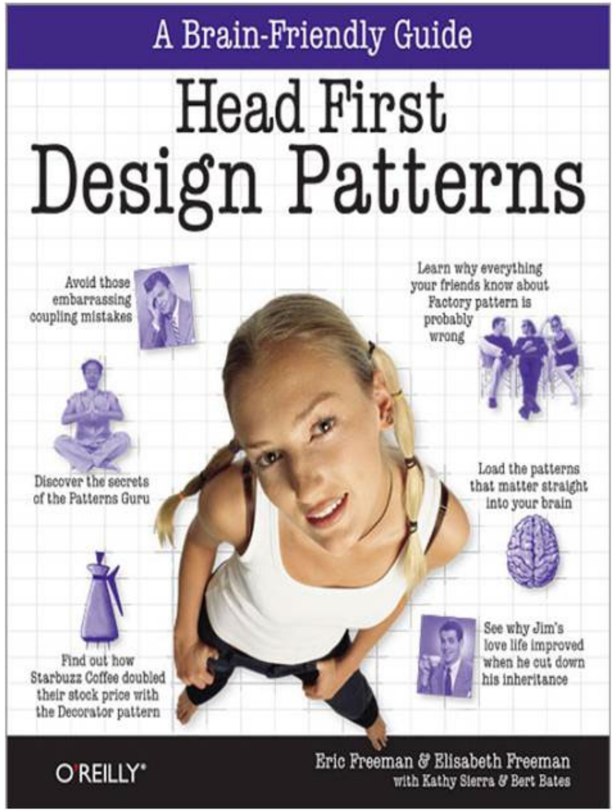


- ❑ ***Duck*** gets some behavior *from "composition" rather than inheritance.*
- ❑ *Composition has the advantage of "**can be changed dynamically**"*
  - ❖ *But,you lose some static checking*

Losing static checking: it is no longer possible to statically guarantee that every Wild duck will be able to quack, even if we make it so in the implementation.

Books





1995, "Gang of 4"
Catalog of 23 patterns

2004
Eric Freeman & Elisabeth Freeman
Much better explanation, good reviews

A Simple Example: Weather “plugins”

d : WeatherDisplay

get data

s : Weather Station

w : WeatherDataProvider

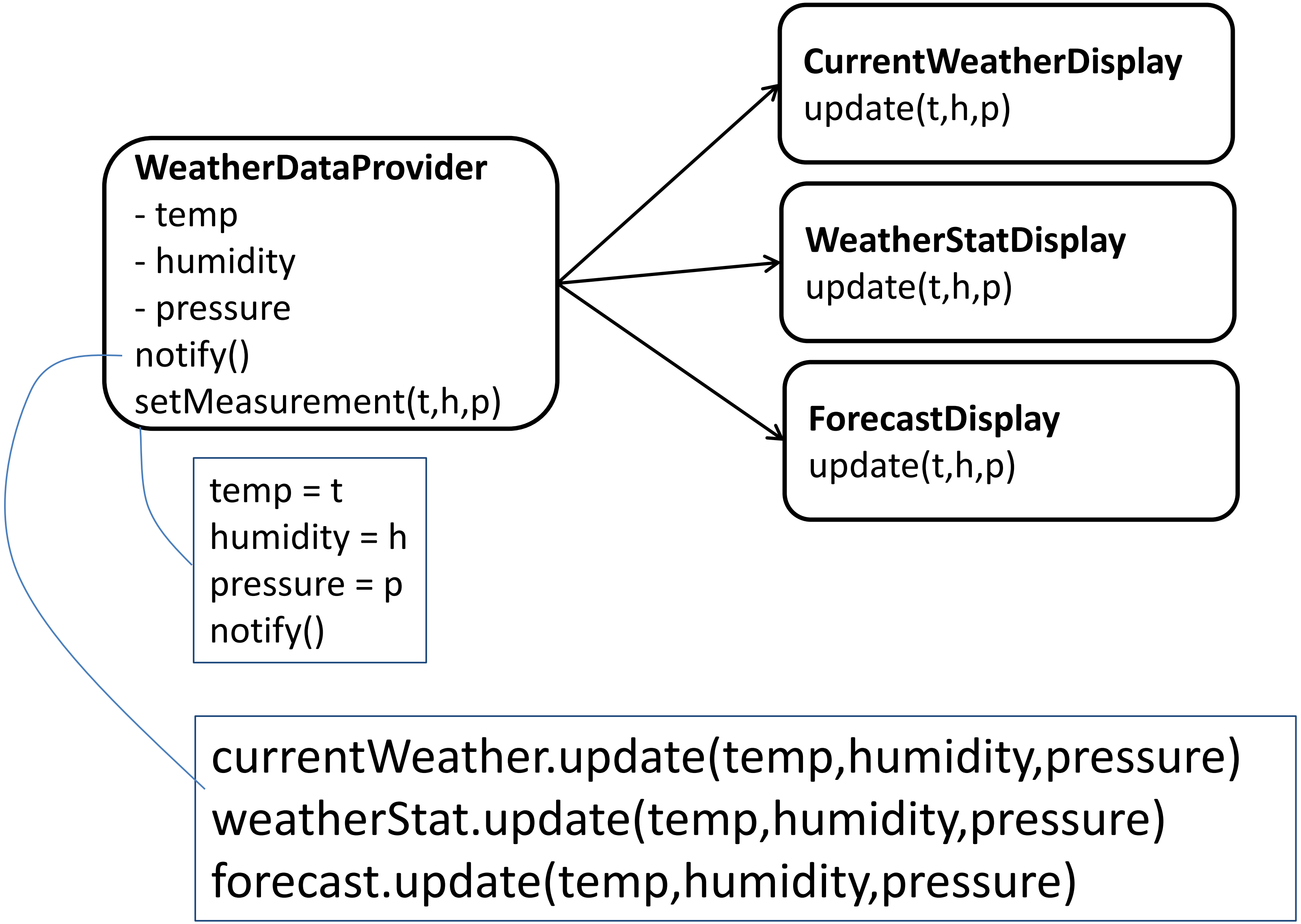
provides live data on temp, humidity, pressure.

Three kinds of displays:

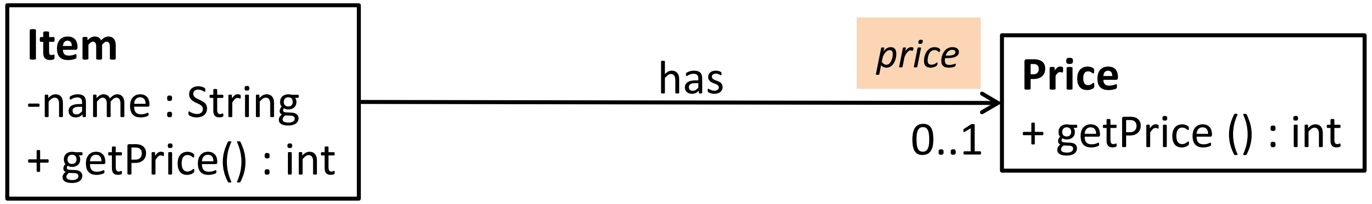
- current weather
- weather statistics
- prediction

Have to be regularly updated.

Design of example weather «plugin»



Converting UML classes to Java

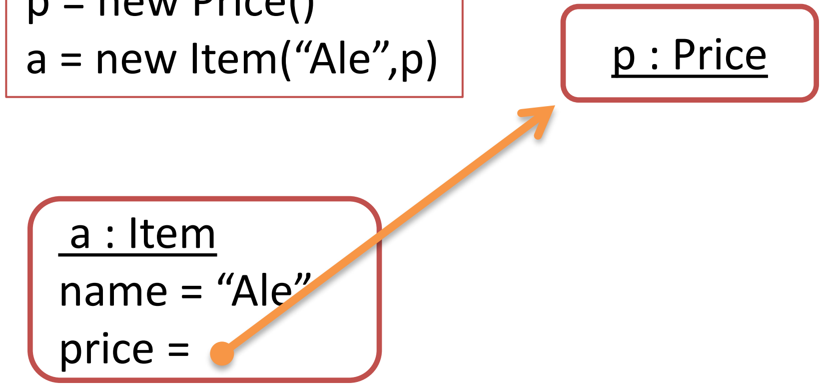



```
class Item {
    private name : String
    private  price : Price
    public getPrice () …
}
```

```
Item(n,p) {
    name = n
    price = p
}
```

```
class Price {
    public getPrice () …
}
```

```
p = new Price()
a = new Item("Ale",p)
```

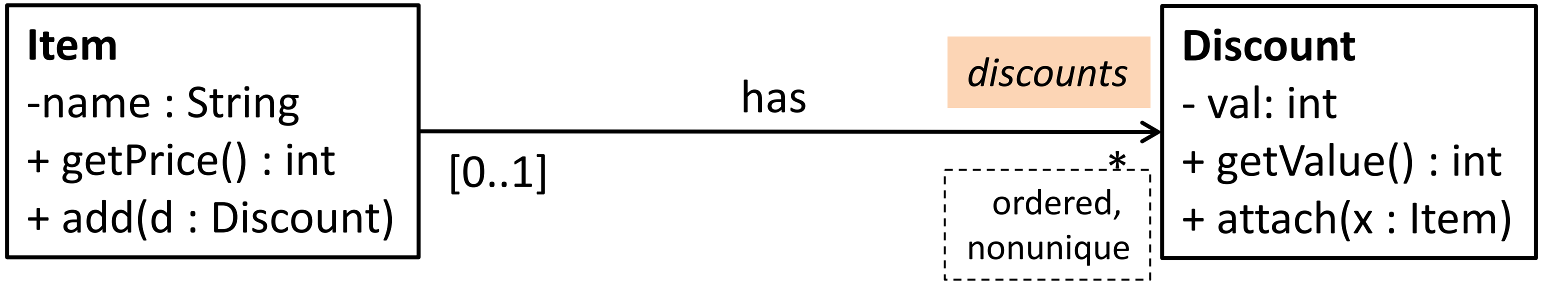


Map *association to attribute*

- How to map:
  - association name
  - multiplicity
  - navigation direction
  - aggregation & composition
  - inheritance

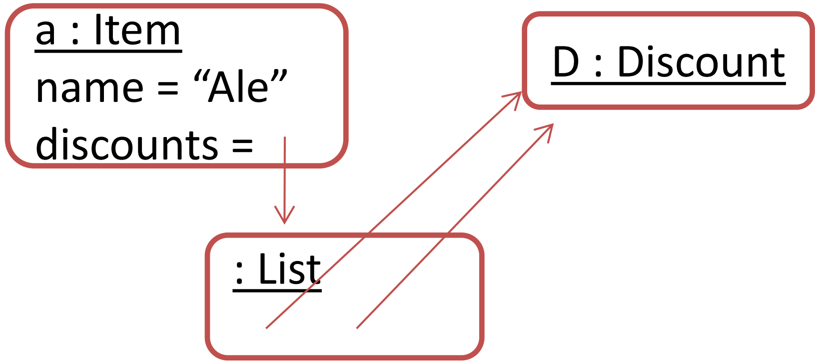
Java has no direct support for association name

Converting UML classes to Java




```
class Item {
    private name : String
    private discounts : List<Discount>
    public Item(…) …
    public getPrice () …
    public add(d:Discount)  …
}
```

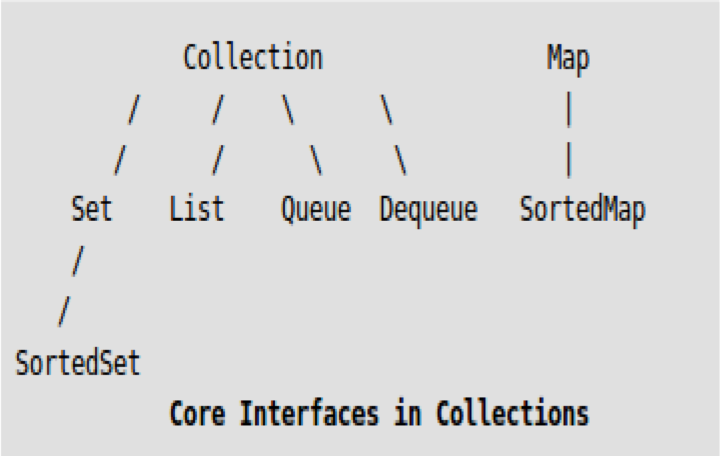
```
a = new Item("Ale")
a.add(D)
a.add(D)
```



List Interface in Java with Examples

List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

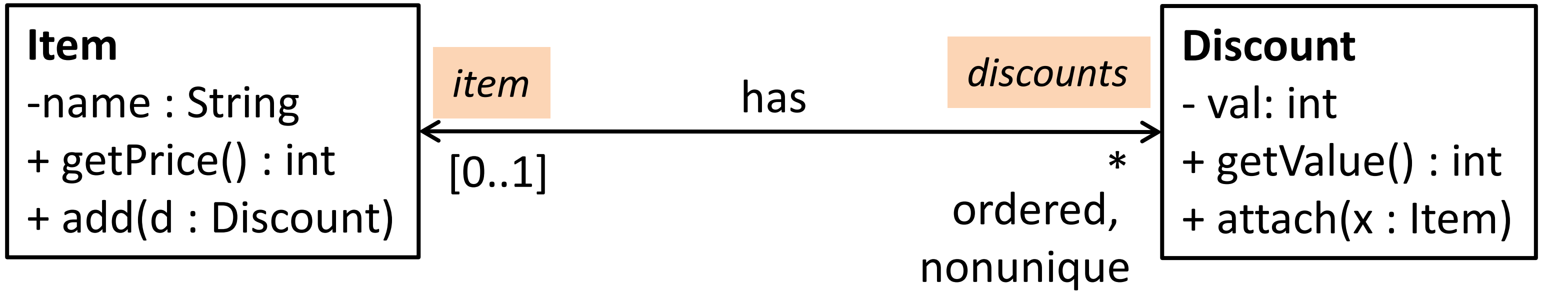
```
// Obj is the type of object to be stored in List.
List<Obj> list = new List<Obj> ();

// Creating a list
        List<Integer> l1 = new ArrayList<Integer>();
        l1.add(0, 1);  // adds 1 at 0 index
        l1.add(1, 2);  // adds 2 at 1 index
        System.out.println(l1);  // [1, 2]
```



Generic List Object:

```
// Obj is the type of object to be stored in List.
List<Obj> list = new List<Obj> ();
List<Object> listAnything = new ArrayList<Object>();
```

Converting UML classes to Java



Java has no direct support for bidirectional navigation.

```
class Item {
   private name : String
   private discounts : List<Discount>
   public Item(…) …
   public getPrice () …
   public add(d:Discount)  …
}
```

```
class Discount {
   private val : int
   private item : Item
   public Discount(…) …
   public getValue() { return val }
   public attach(x:Item)  …
}
```

attach(x) { item = x ; x.add(this) ; }

Singleton Pattern

GoF (Gang of Four) Definition

Ensure a class only has one instance, and provide a global point of access to it.

Principles of Singleton Pattern

- A class cannot have multiple instances.
- Once created, the next time onward, you use only the existing instance.
- This approach helps you restrict unnecessary object creations in a centralized system.
- The approach also promotes easy maintenance.

Real-World Example Representing Singleton Pattern

- Assume that you are a member of a sports team, and your team is participating in a tournament.
- Your team needs to play against multiple opponents throughout the tournament.
- Before each of these matches, the captains of the two sides must do a coin toss.
- If your team does not have a captain, you need to elect someone as a captain.
- Prior to each game and each coin toss, you may not repeat the process of electing a captain if you already nominated a person as a captain for the team.
- From every team member's perspective, there should be only one captain of the team.

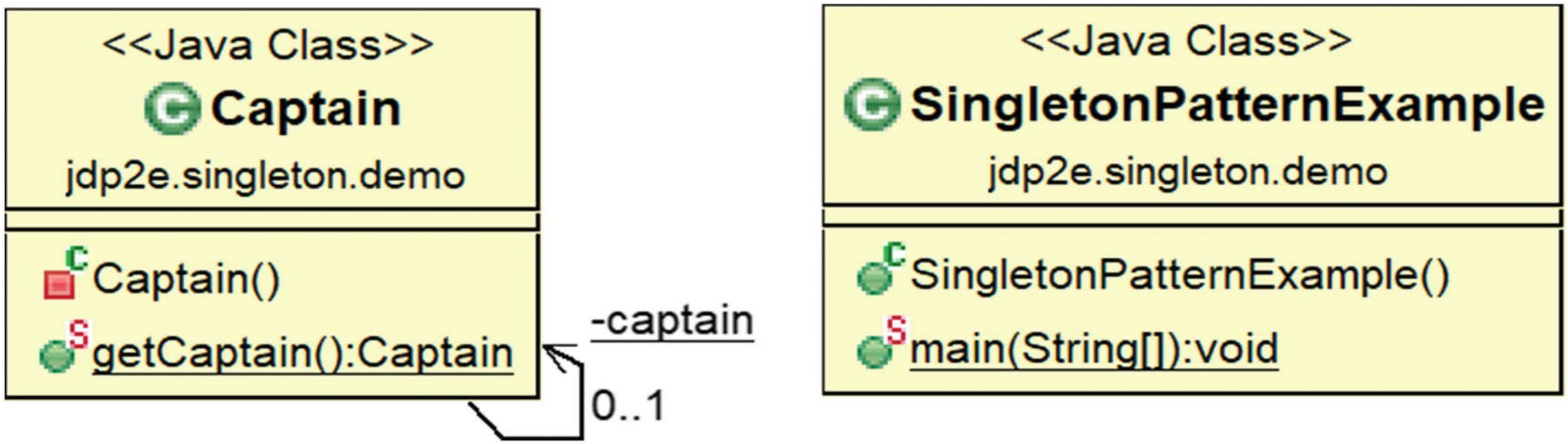
Computer-World Example

- In some specific software systems, you may prefer to use only one file system for the centralized management of resources.
- Skeleton pattern can implement a caching mechanism.

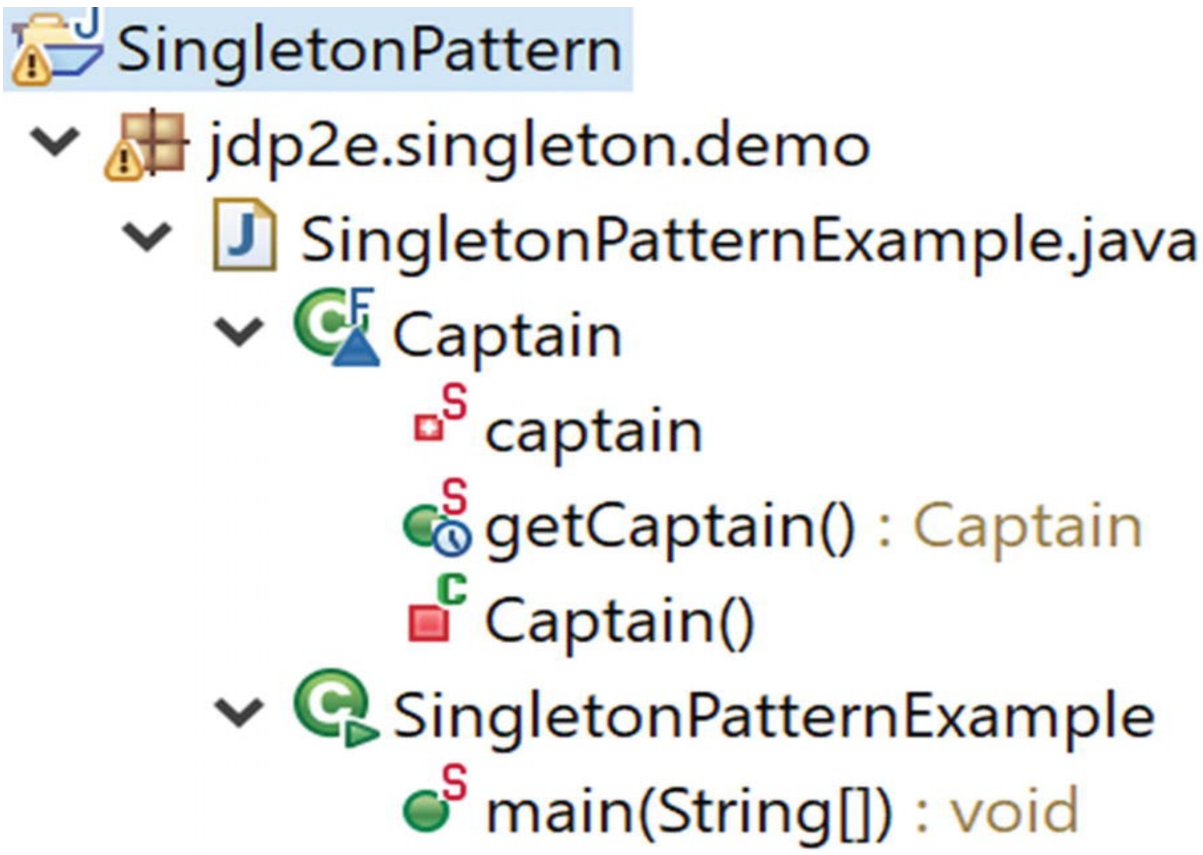
Illustration of Singleton Pattern

- The constructor is private to prevent the use of a "new" operator.
- You'll create an instance of the class, if you did not create any such instance earlier; otherwise, you'll simply reuse the existing one.
- To take care of thread safety, we use the "synchronized" keyword.

Class Diagram of Singleton Pattern



Package Explorer View of Singleton Pattern



The Properties of Singleton Pattern

- The constructor is private, so you cannot instantiate the Singleton class(Captain) outside.
- It helps us to refer the only instance that can exist in the system
- You restrict the additional object creation of the Captain class.
- The private constructor also ensures that the Captain class cannot be extended.
  - So, subclasses cannot misuse the concept.
- “Synchronized” keyword is used.
  - Multiple threads cannot involve in the instantiation process at the same time.
  - We force each thread to wait its turn to get the method, so thread- safety is ensured.
  - Synchronization is a costly operation and once the instance is created, it is an additional overhead

```java
package jdp2e.singleton.demo;
final class Captain{
      private static Captain captain;
  //We make the constructor private to prevent the use of "new"
     private Captain() {    }
  public static synchronized Captain getCaptain()
        {       // Lazy initialization   ??
                        if (captain == null)       {
                                captain = new Captain();
     System.out.println("New captain is elected for your team.");       }
    else     {
  System.out.print     ("You already have a captain for your team.");
  System.out.println  ("Send him for the toss.");       }
                  return captain;    }}
```

Java static method

- Static methods in Java can be called without creating an object of class
- Java allows procedural programming through its static methods (e.g. the main() method).
- A static method belongs to a class independent of any of the class's instances.
- It would be possible to implement an entire program through static methods, which call each other procedurally, but that is not OOP.

4 Mart Dersinin İnteraktif Çalışması

Tartışılacak Video Sunumu:

Workflows Refoctaring Martin Fowler

https://www.youtube.com/watch?v=vqEg37e4Mkw

Videoyu dinlemeniz ve not almanız istenmektedir.

Ders sırasında videoya dönüş için yapabilmek için alacağınız notlarda zaman kaydı yapmanız istenmektedir.

Derste öğrencilerden izlenilen video ve dersin kapsamı dahilinde görüşleri tartışılacaktır.