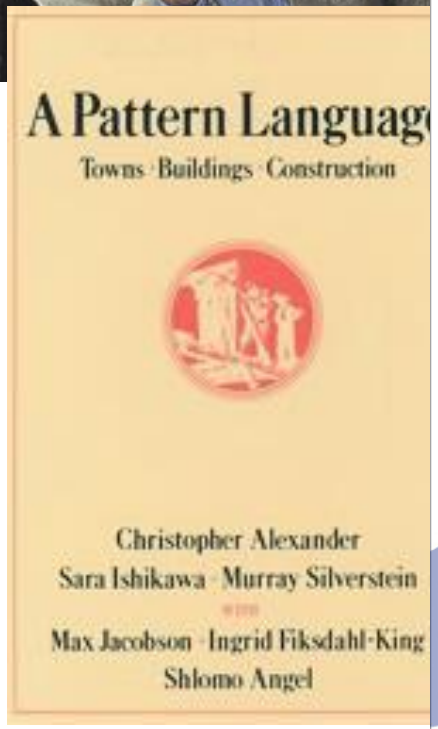
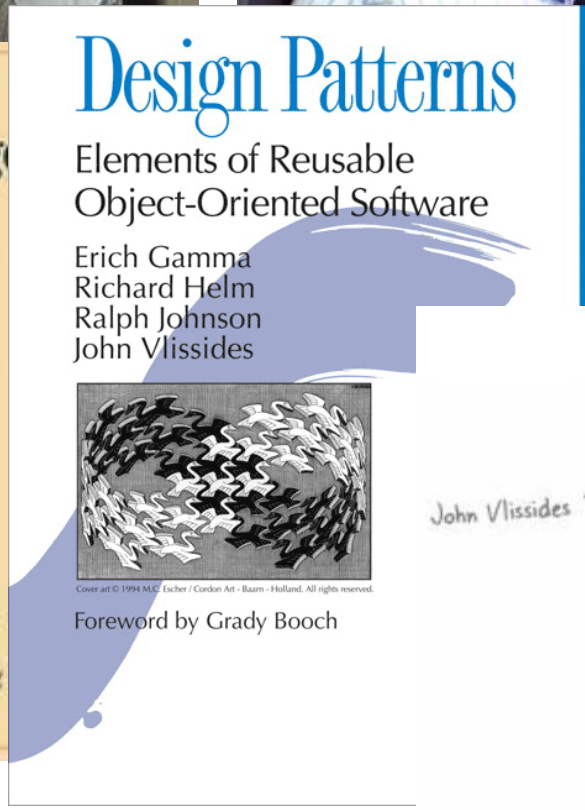
<epam> Delivering Excellence in Software Engineering

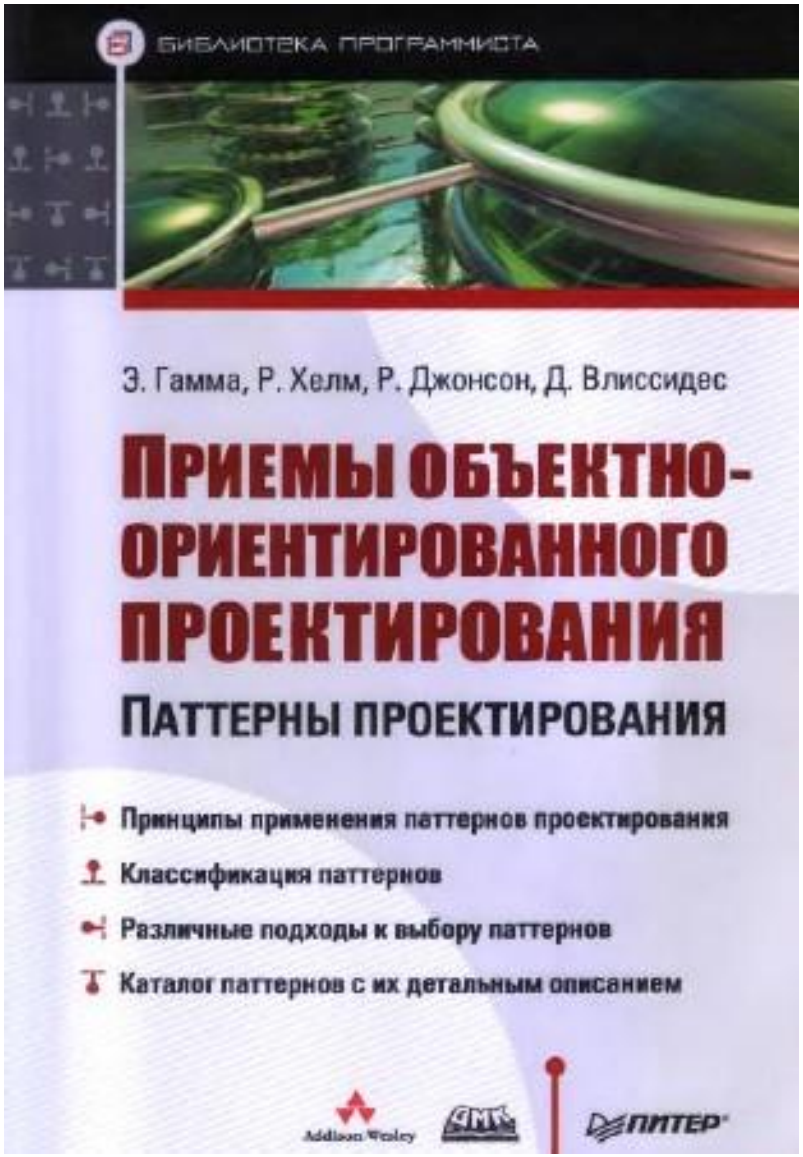
The Strategy Pattern

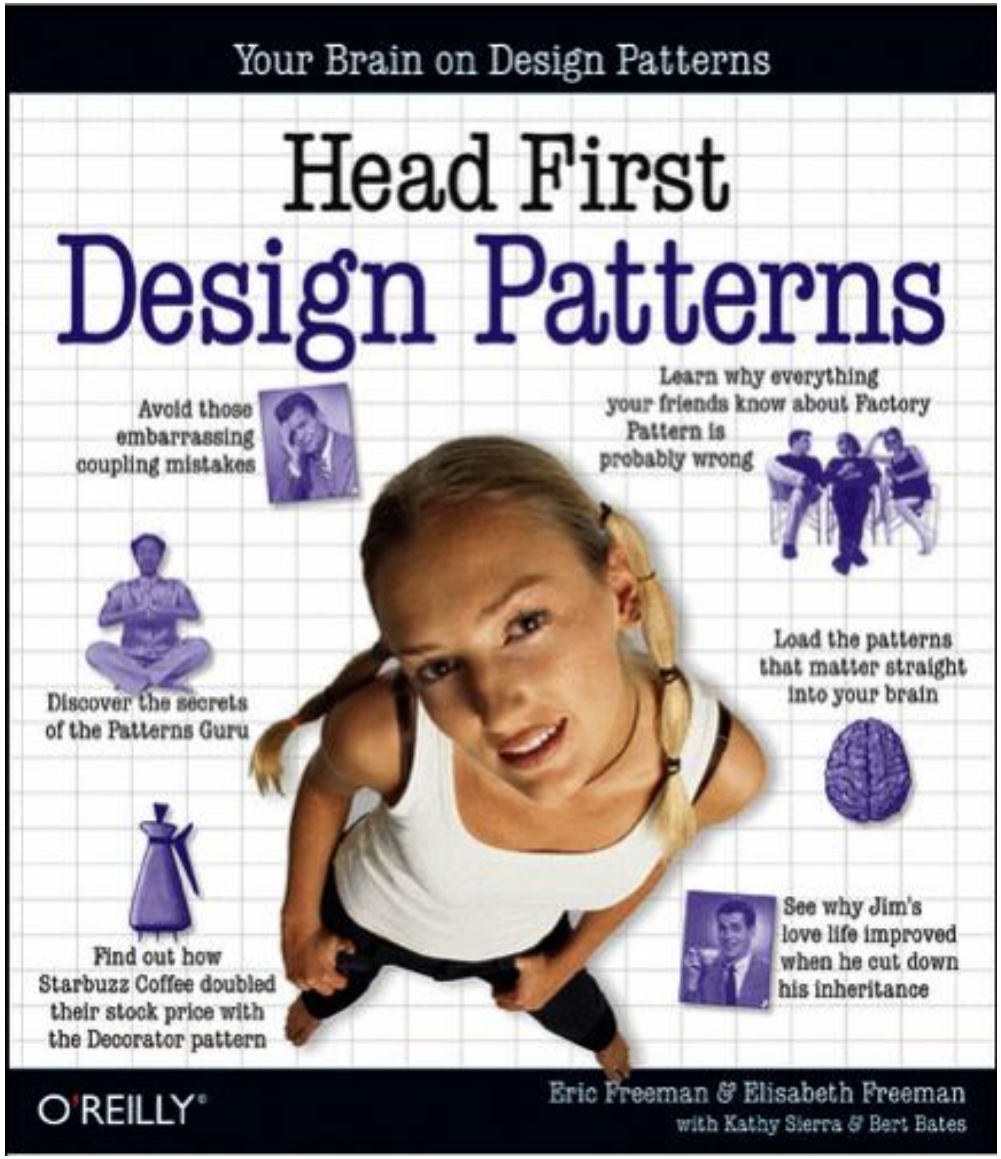
Christopher Alexander & Gang of four





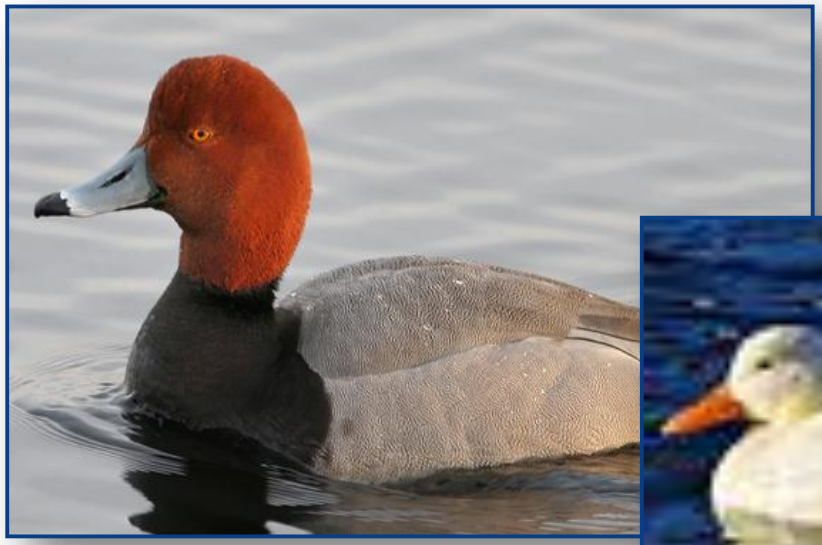
Books

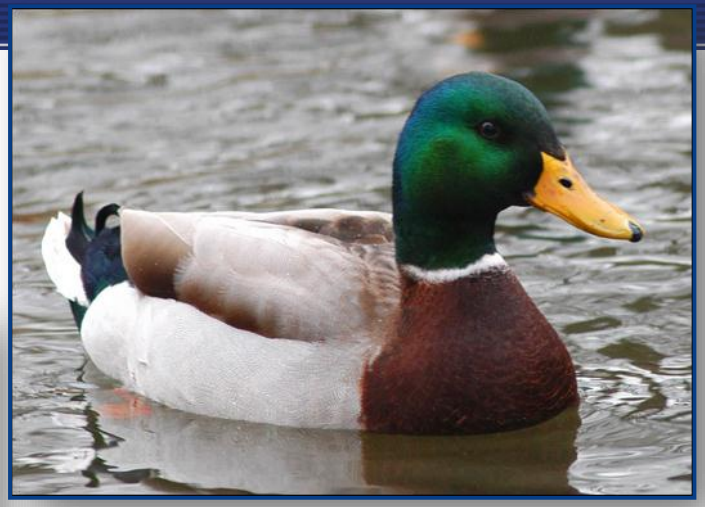




Different behaviors

Classes of ducks





RedheadDuck

+ quack() {…}
+ swim() {…}
+ display();
+ …

WhiteDuck

+ quack() {…}
+ swim() {…}
+ display();
+ …

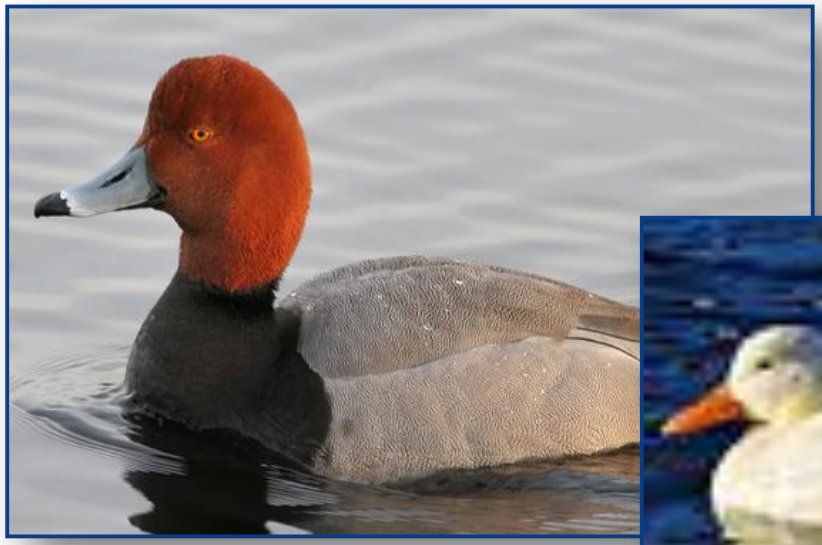
TiredDuck

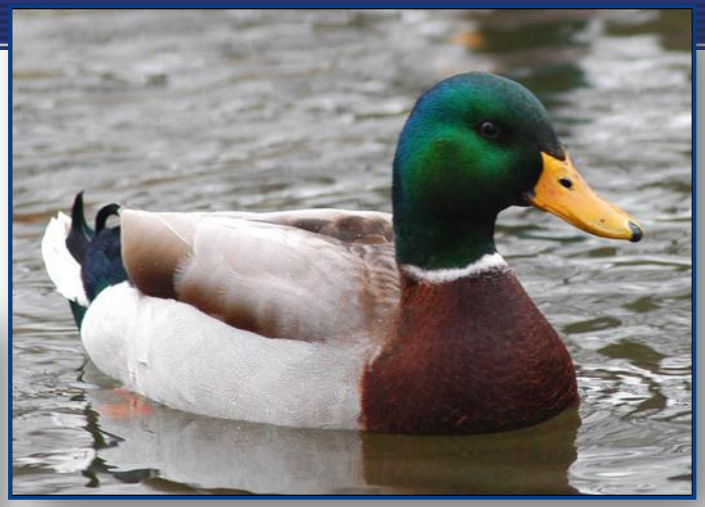
+ quack() {…}
+ swim() {…}
+ display();
+ …

MallardDuck

+ quack() {…}
+ swim() {…}
+ display();
+ …

Classes of ducks





RedheadDuck

+ quack() {…}
+ swim() {…}
+ display();
+ …

WhiteDuck

+ quack() {…}
+ swim() {…}
+ display();
+ …

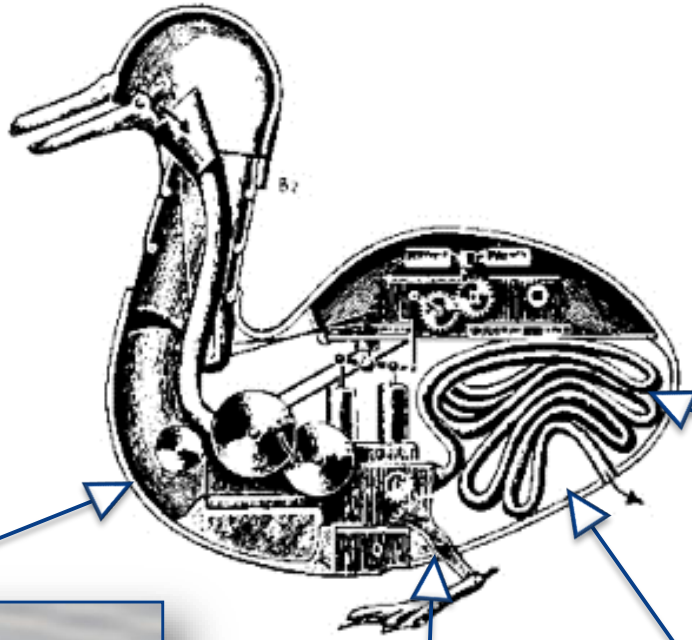
TiredDuck

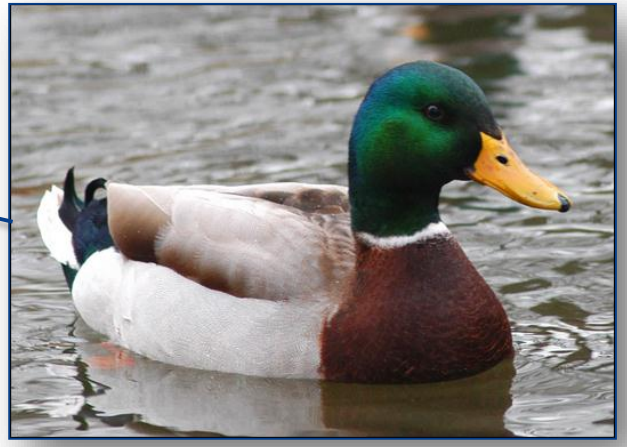
+ quack() {…}
+ swim() {…}
+ display();
+ …

MallardDuck

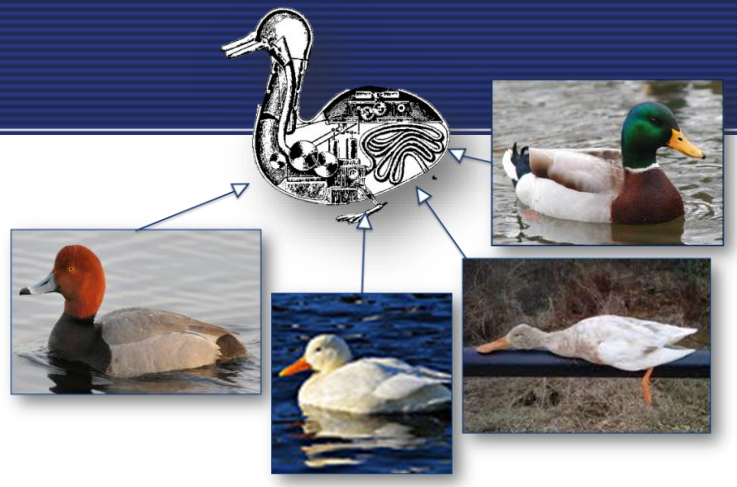
+ quack() {…}
+ swim() {…}
+ display();
+ …

Hierarchy of ducks





Hierarchy of ducks



Duck

+ quack() {…}
+ swim() {…}
+ display();
+ …

RedheadDuck

+ display() {…}
+ …

WhiteDuck

+ display() {…}
+ …

TiredDuck

+ display() {…}
+ …

MallardDuck

+ display() {…}
+ …

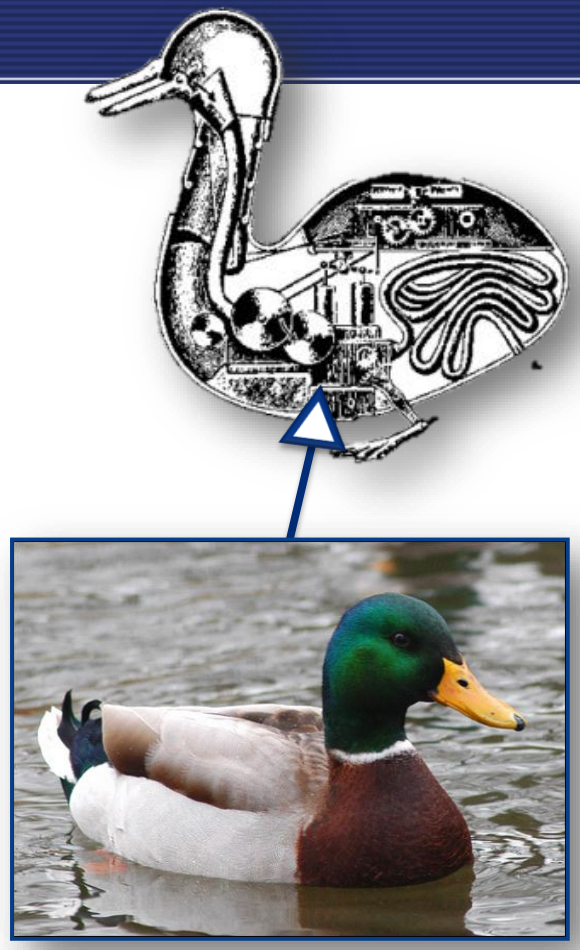
Code of base (super) Duck

```
1   public abstract class Duck {
2
3       public abstract void display();
4
5       public void swim() {
6           // swimming logic implementation
7       }
8
9       public void quack() {
10          // quack logic implementation
11      }
12  }
```

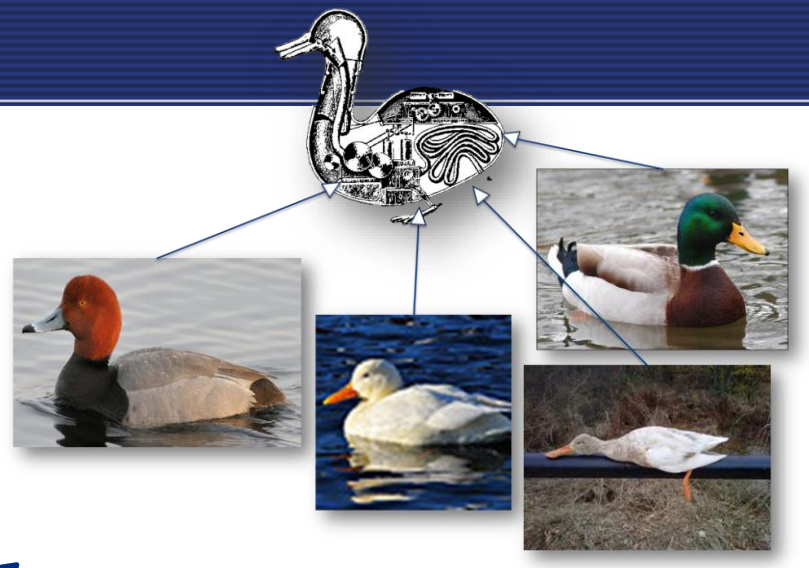
Code of Ducks

```java
public abstract class Duck {

    public abstract void display();

    public void swim() {
        // swimming logic implementation
    }

    public void quack() {
        // quack logic implementation
    }
}
```

```java
public class MallardDuck extends Duck {

    @Override
    public void display() {
        // mallarduck display logic implementation
    }

}
```



Hierarchy of ducks



Duck

+ quack() {…}
+ fly() {…}
+ swim() {…}
+ display();
+ …

RedheadDuck

+ display() {…}
+ …

WhiteDuck

+ display() {…}
+ …

TiredDuck

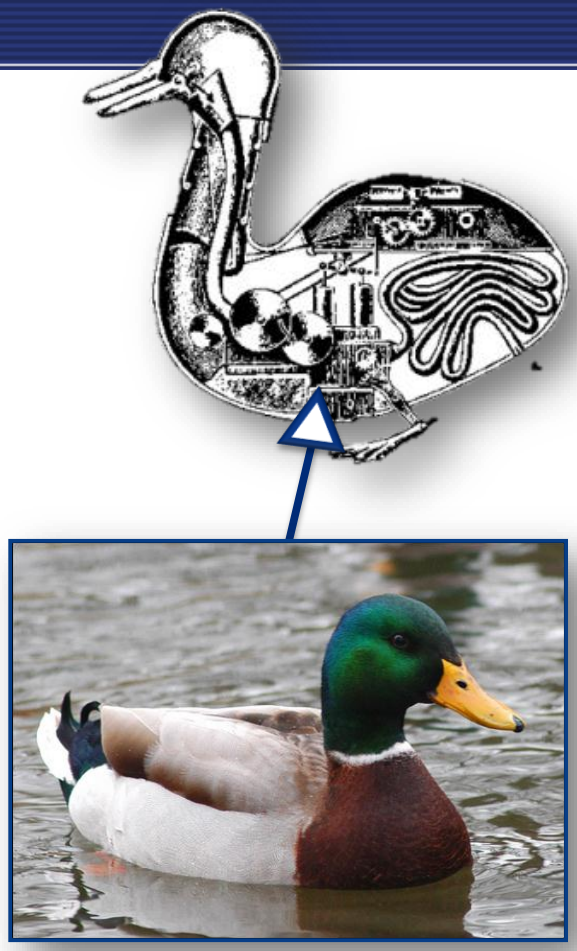
+ display() {…}
+ …

MallardDuck

+ display() {…}
+ …

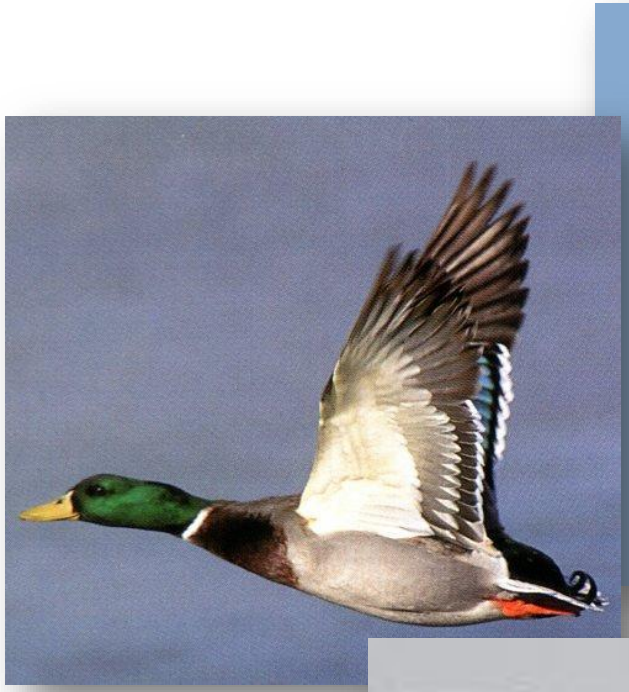
Code of Ducks

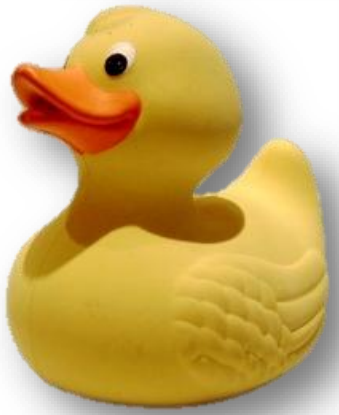
```
1  public abstract class Duck {
2
3      public abstract void display();
4
5      public void swim() {
6          // swimming logic implementation
7      }
8
9      public void quack() {
10         // quack logic implementation
11     }
12
13     public void fly() {
14         // fly logic implementation
15     }
16 }
```

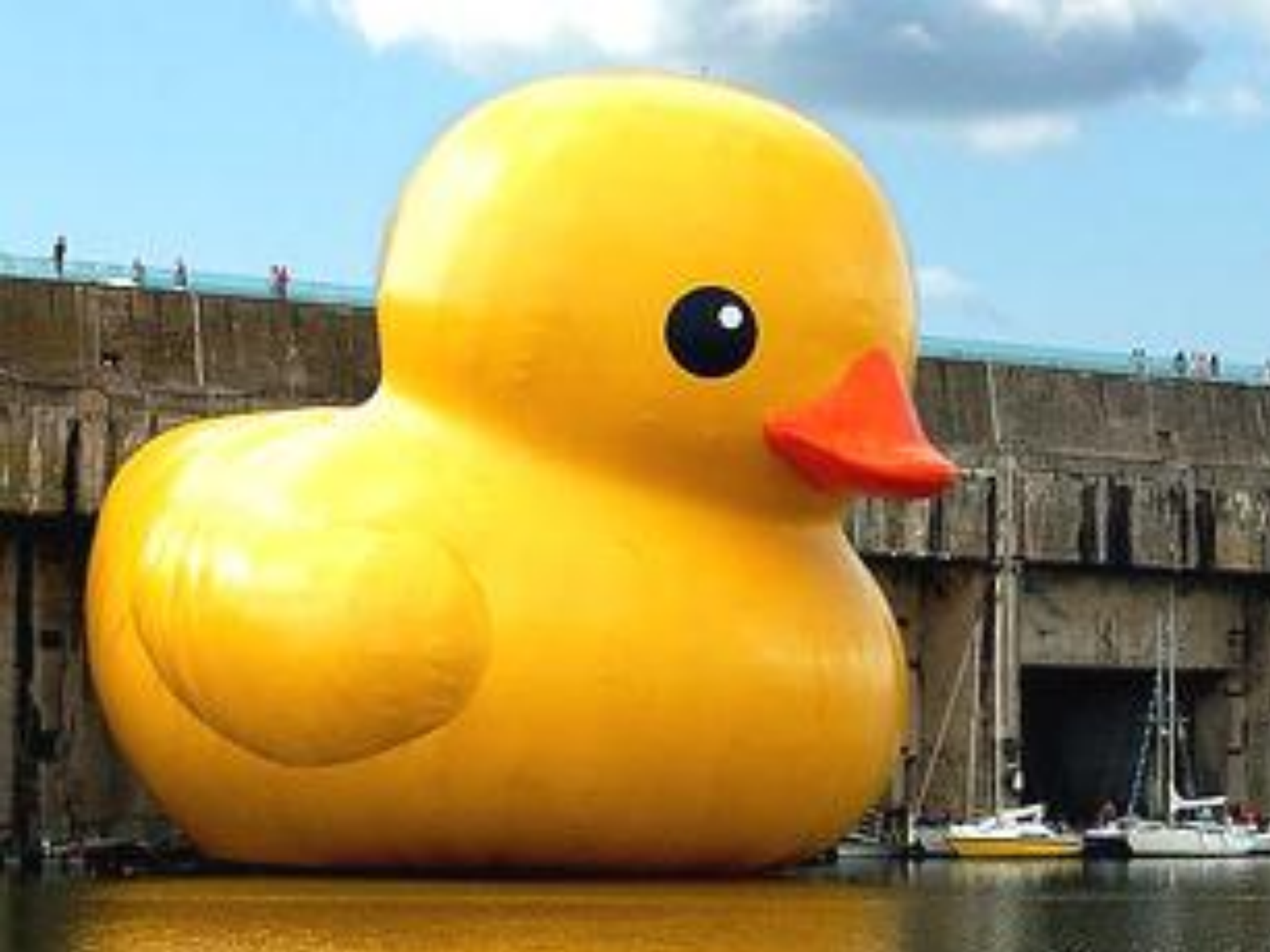


```
1  public class MallardDuck extends Duck {
2
3      @Override
5      public void display() {
6          // mallarduck display logic implementation
7      }
8  }
```

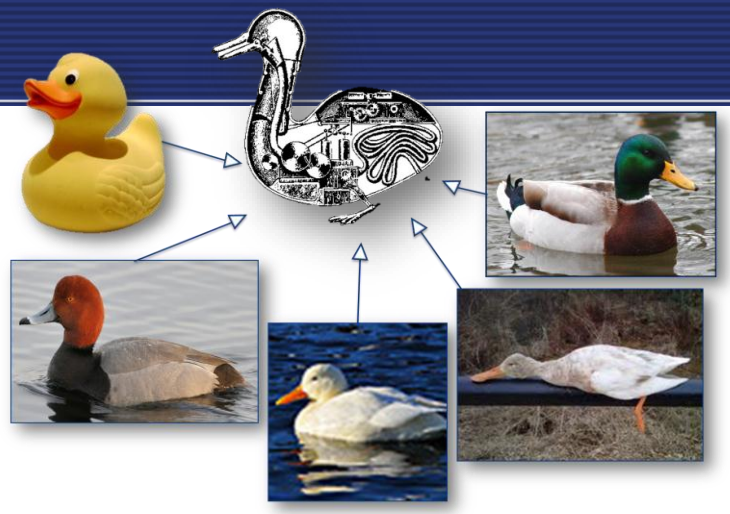
Flying



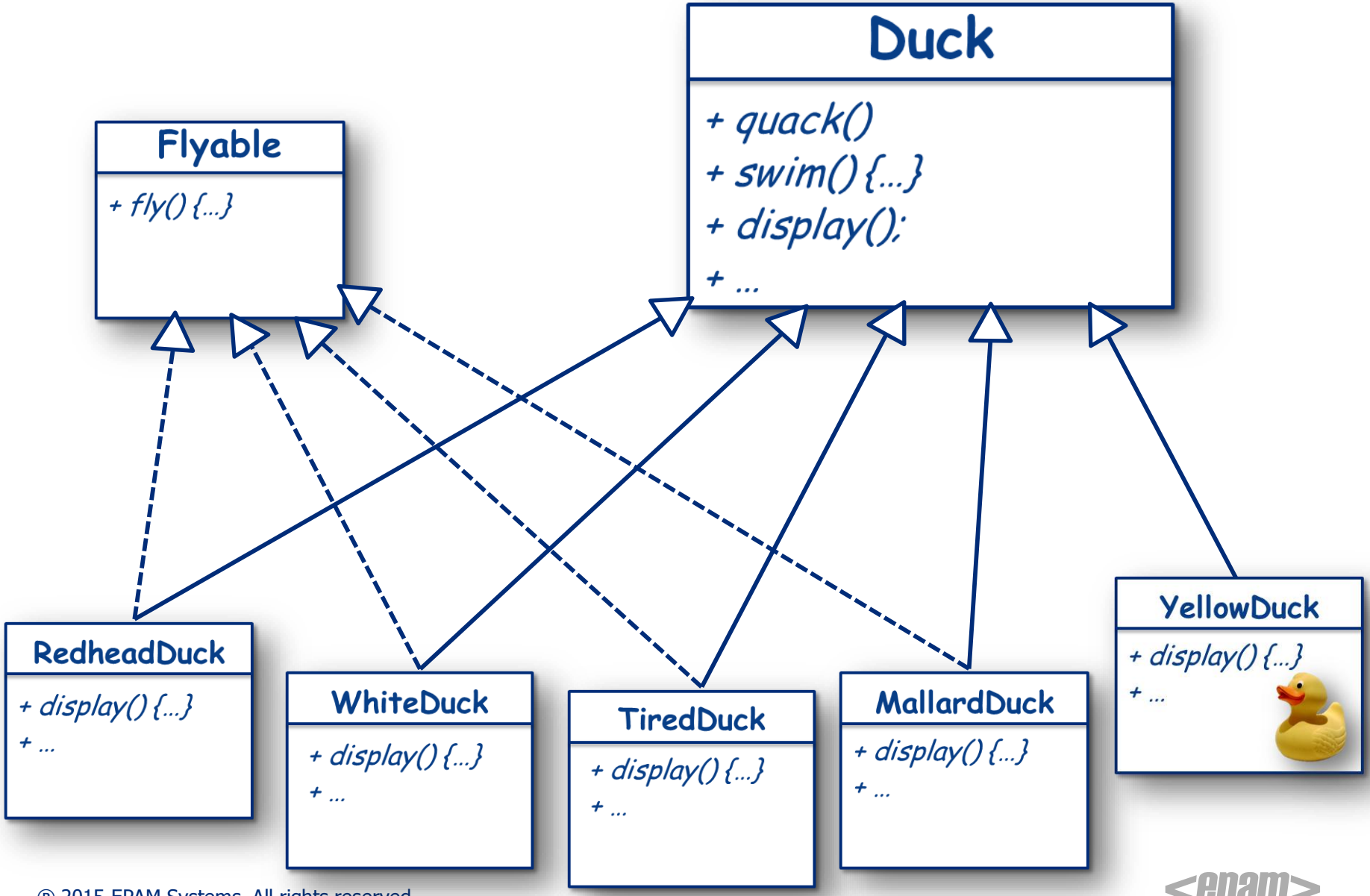




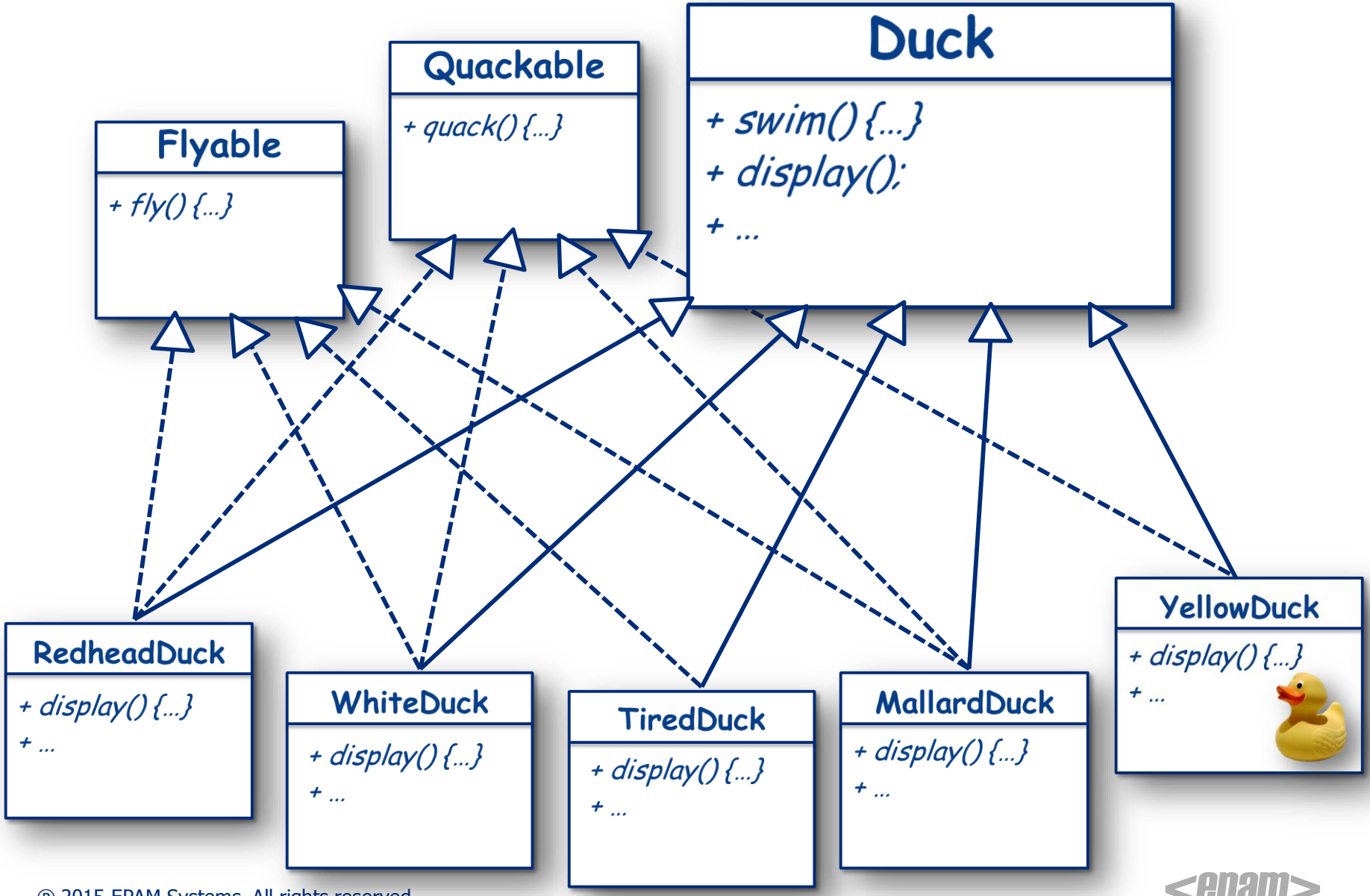
Hierarchy of ducks



What do you think about this design?

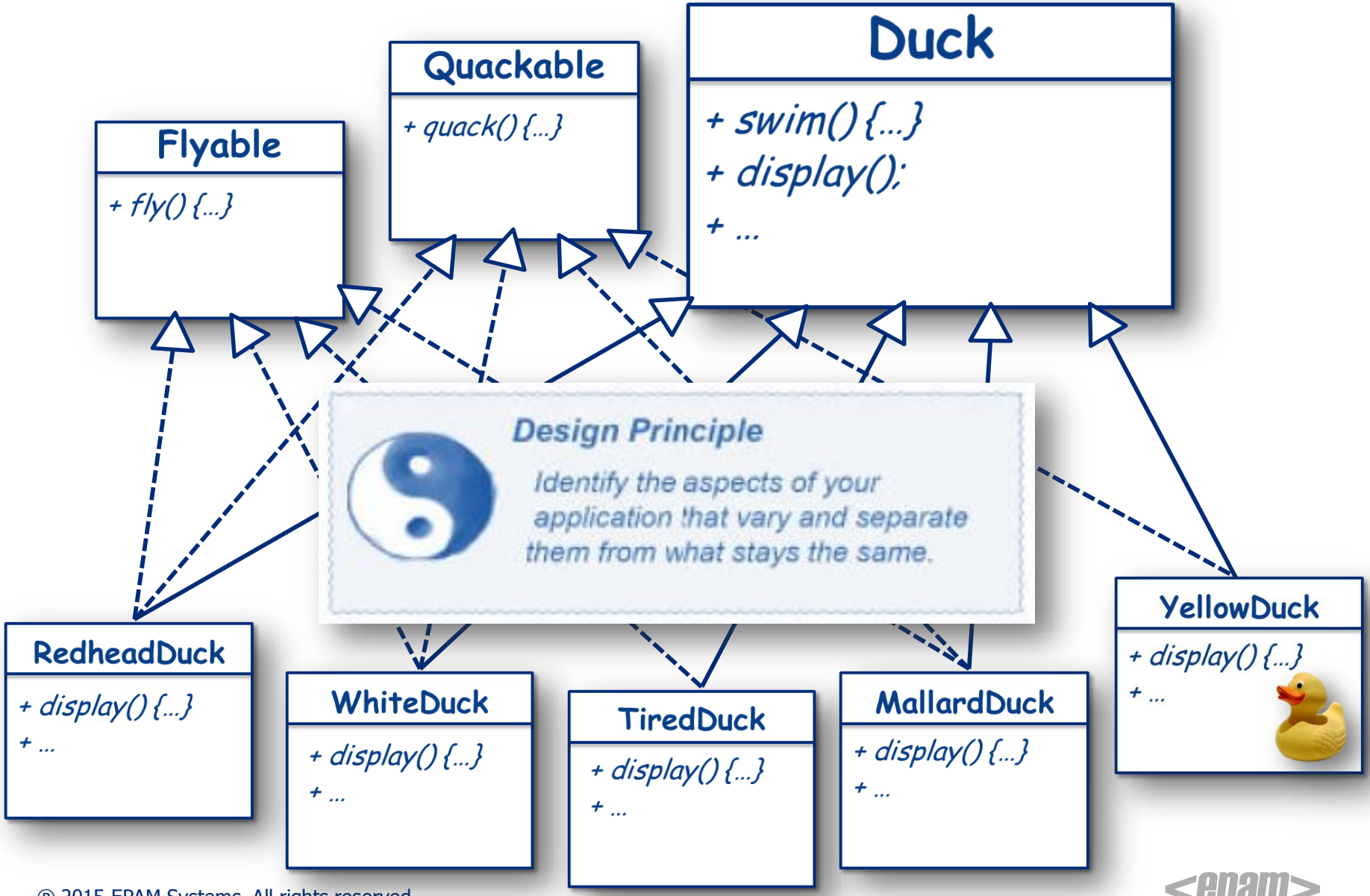


What do you think about this design?

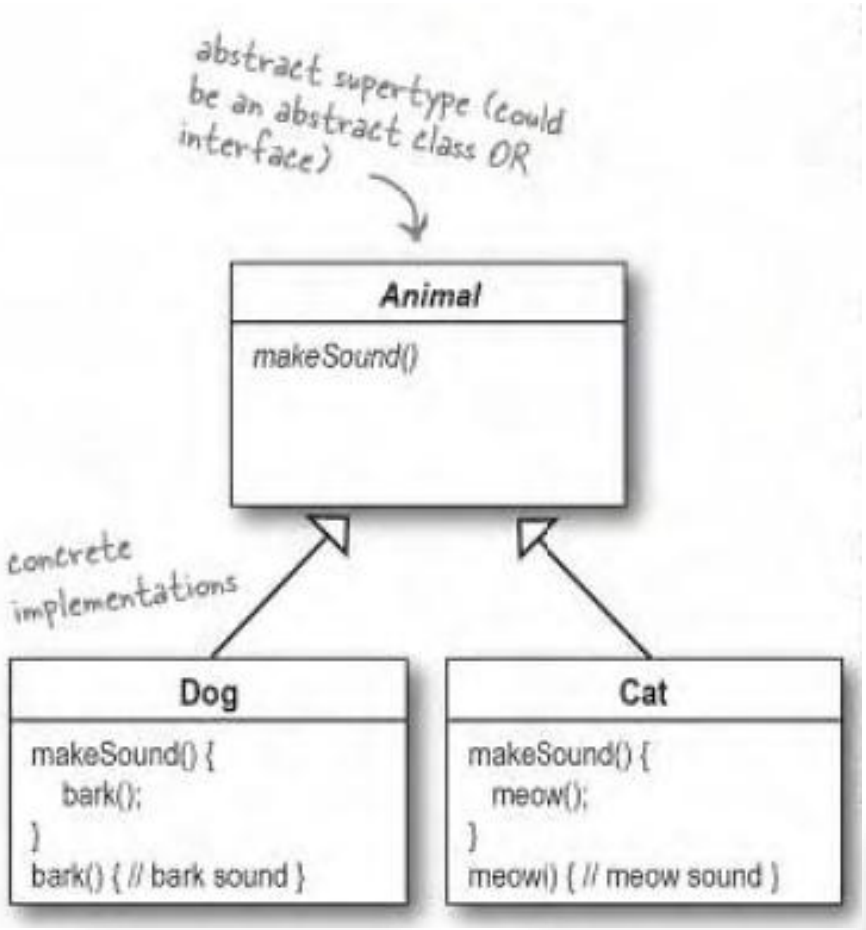


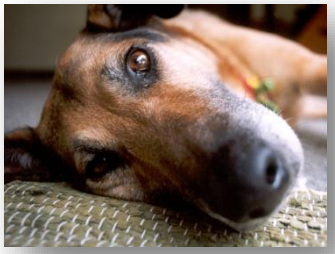
<epam>

What do you think about this design?



Designing the Duck Behavior



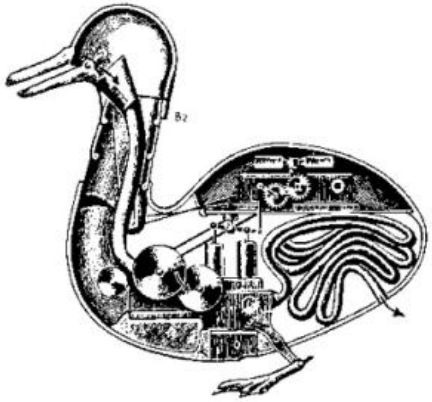


Design Principle

Program to an interface, not an implementation.

Testing the Duck code

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }

    public void setFlyBefavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBefavior(QuackBehavior qb){
        quackBehavior = qb;
    }
}
```



Fly Behavior

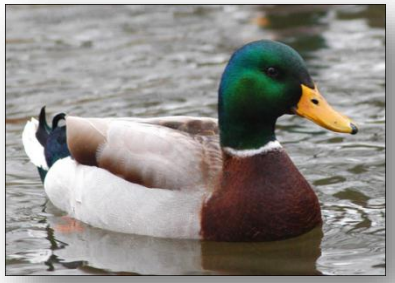
```
public interface FlyBehavior {
    public void fly();
}

public class FlyWithWings implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I'm flying :)");
    }
}

public class FlyNoWay implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I can't fly :(");
    }
}

public class FlyRocketPowered implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I'm flying with a rocket !");
    }
}
```

Quack Behavior

```
public interface QuackBehavior {
    public void quack();
}

public class Quack implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("Quack");
    }
}

public class Squeak implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("Squeak");
    }
}

public class MuteQuack implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("<< Silence >>");
    }
}
```

Testing Subclasses of Duck

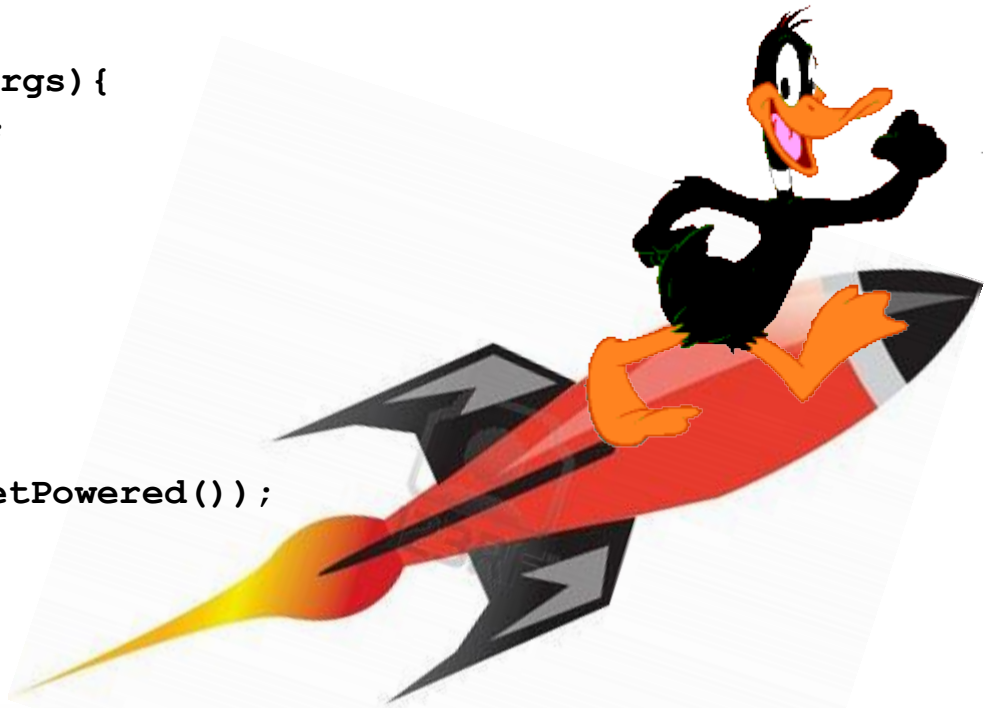
```
public class MallardDuck extends Duck{
    public MallardDuck(){
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
    @Override
    public void display() {
        System.out.println("I'm a real Mallard duck !");
    }
}
```



```
public class DuffyDuck extends Duck{
public DuffyDuck(){
    quackBehavior = new Quack();
    flyBehavior = new FlyNoWay();
}
    @Override
    public void display() {
        System.out.println("I'm a real Duck... Duffy Duck !!!");
    }
}
```

Testing the MiniDuckSimulator

```
public class MiniDuckSimulator {
    public static void main(String ... args){
        Duck mallard = new MallardDuck();
        mallard.performFly();
        mallard.performQuack();

        Duck duffy = new DuffyDuck();
        duffy.display();
        duffy.performFly();

        duffy.setFlyBefavior(new FlyRocketPowered());

        duffy.performFly();
    }
}
```



Console

```
<terminated> PizzaTestDrive [Java Application] C:\Program Files\Java\
I'm flying :)
Quack
I'm a real Duck... Duffy Duck !!!
I can't fly :(
I'm flying with a rocket !
```

Design Principle

Favor composition over inheritance.

Speaking of design pattern

Congratulations on your first pattern!

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Summery

- ✓ знание основ ООП не сделает из Вас хорошего ООП-проектировщика
- ✓ хорошие ООП-архитектуры хорошо расширяются, просты в сопровождении и пригодны для повторного использования
- ✓ паттерны показывают, как строить системы с хорошими качествами ООП-проектирования
- ✓ паттерн содержит проверенный опыт ООП-проектирования
- ✓ паттерны описывают общие решения проблем проектирования и применяются в конкретных приложениях
- ✓ паттерны не придумывают – их находят
- ✓ большинство паттернов и принципов направлено на решение проблем изменения программных архитектур
- ✓ многие паттерны основаны на инкапсуляции переменных аспектов системы
- ✓ паттерны образуют единую номенклатуру, которая повышает эффективность Вашего общения с другими разработчиками

<epam> Delivering Excellence in Software Engineering

The Strategy Pattern

For more information, please contact:
Victor Ivanchenko,
epam trainer
Email: ivanvikvik@gmail.com

EPAM Systems, Inc.
http://www.epam.com