POLITEKNIK STATISTIKA STIS

For Better Official Statistics

DESIGN PATTERNS

(PART I)

Mutaqin

BASIC OO PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Composition

“Over time an application must **grow** and **change** or it will **die**”

(Eric Freeman)

Extensible & Maintainable

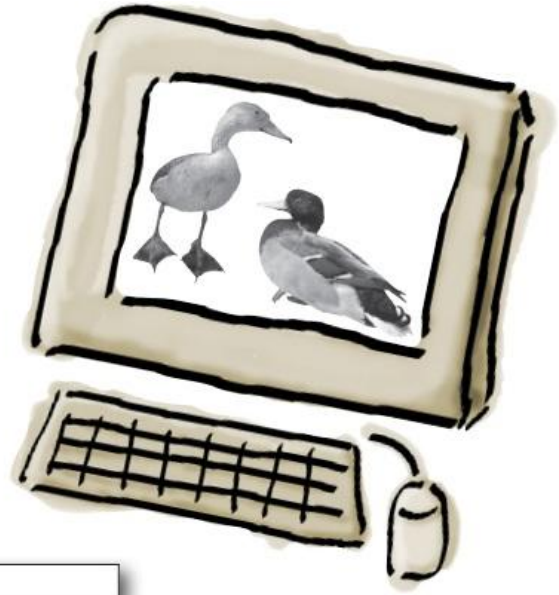
DESIGN PRINCIPLES

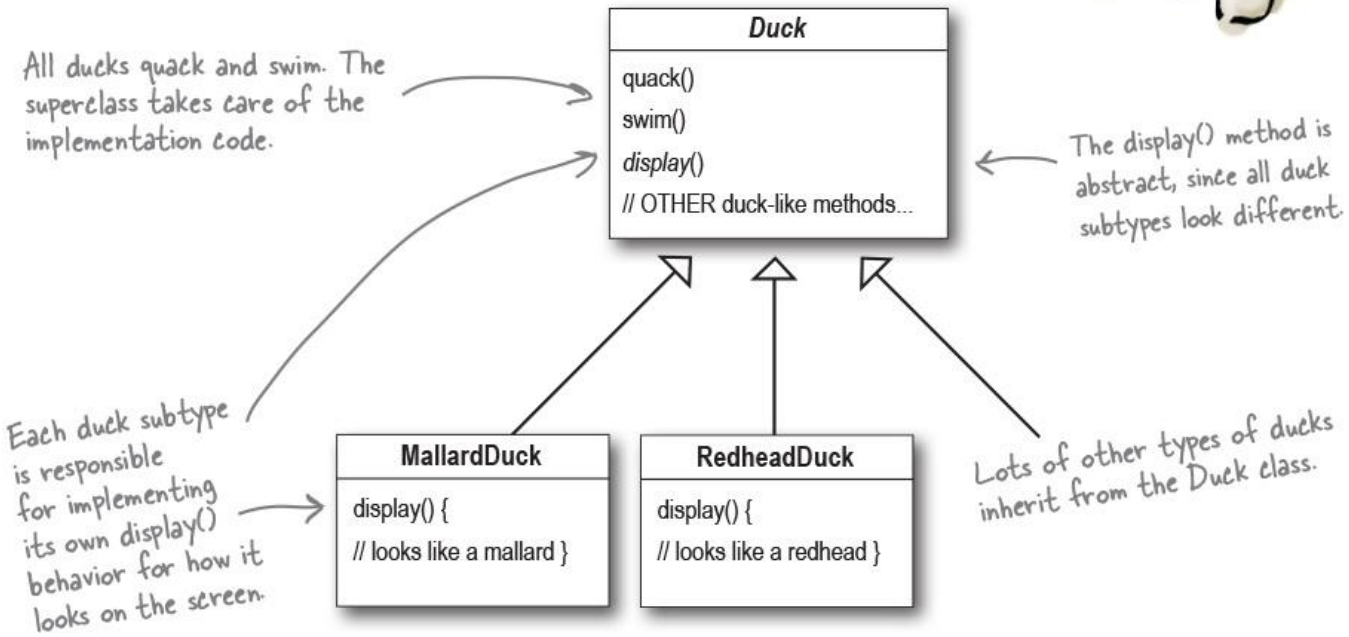
- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface, not an implementation.
- Favor composition over inheritance.
- Minimize the Accessibility of Classes and Members
- Encapsulate what varies
- Software Entities (Classes, Modules, Functions) should be Open for Extension, but Closed for Modification.
- Functions that use references to base classes must be able to use objects of derived subclasses without knowing it.
- Depend On Abstractions. Do not depend on Concrete Classes.

DESIGN PATTERNS

START FROM THE PROBLEM...

Sim UDuck App

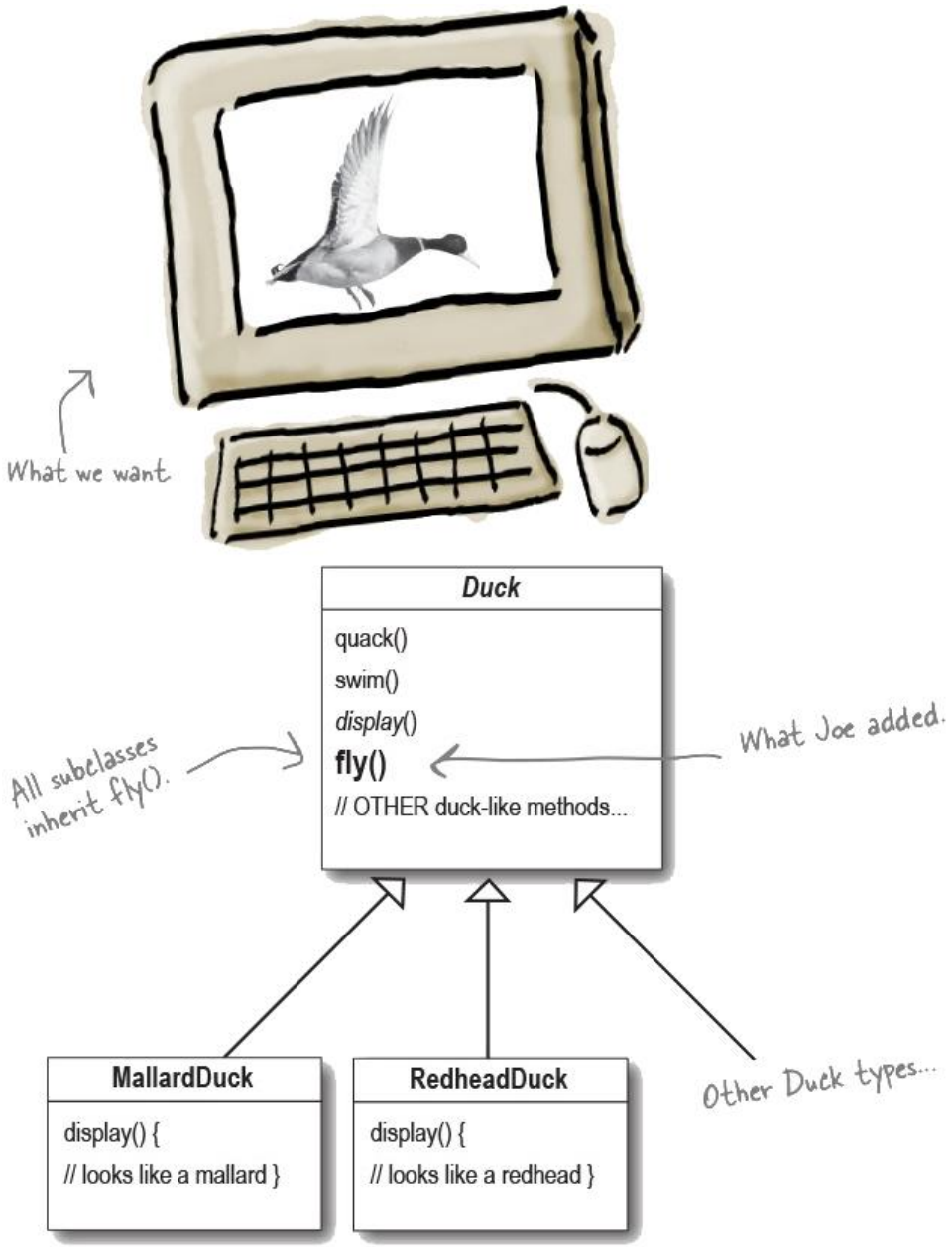




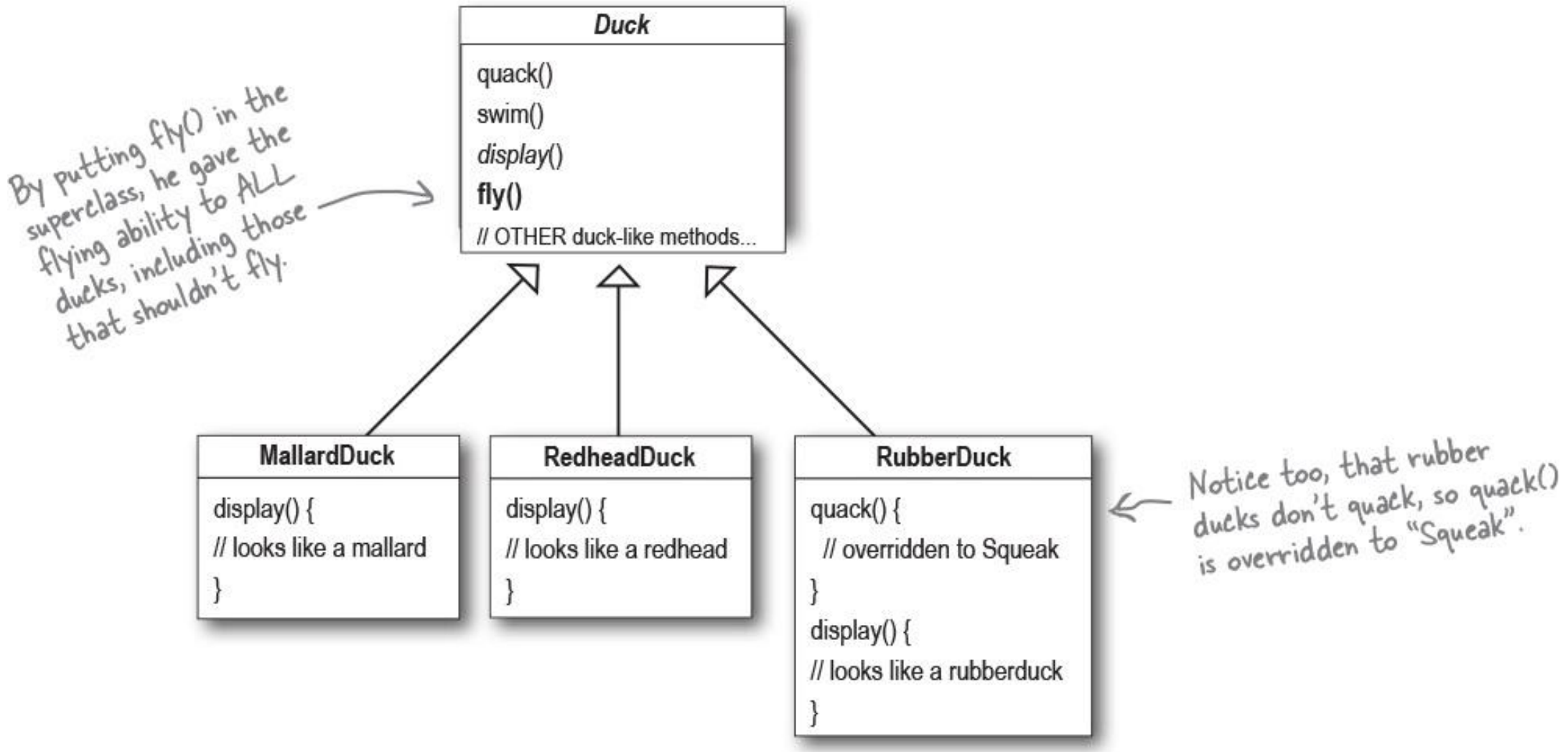
```java
public abstract class Duck {
    public void quack(){
        System.out.println("A duck quack");
    }
    public void swim(){
        System.out.println("A duck swim");
    }
    public abstract void display();
    //many other methods here
```

```java
public class RedheadDuck extends Duck{

    @Override
    public void display() {
        System.out.println("It is a read head duck.");
    }
    //many other methods here
```

```java
public class MallardDuck extends Duck{

    @Override
    public void display() {
        System.out.println("It's a mallard");
    }
    //many other methods here
```


What we want
Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

```java
public abstract class Duck {

    public void quack(){
        System.out.println("A duck quacks");
    }
    public void swim(){
        System.out.println("A duck swims");
    }
    public void fly(){
        System.out.println("A duck flies");
    }
    public abstract void display();
    //many other methods here
```


By putting fly() in the superclass, he gave the flying ability to ALL ducks, including those that shouldn't fly.
Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard
}
RedheadDuck
display() {
// looks like a redhead
}
RubberDuck
quack() {
// overridden to Squeak
}
display() {
// looks like a rubberduck
}
Notice too, that rubber ducks don't quack, so quack() is overridden to "Squeak".

RubberDuck

```
quack()  { // squeak}
display() { // rubber duck }
fly() {
  // override to do nothing
}
```

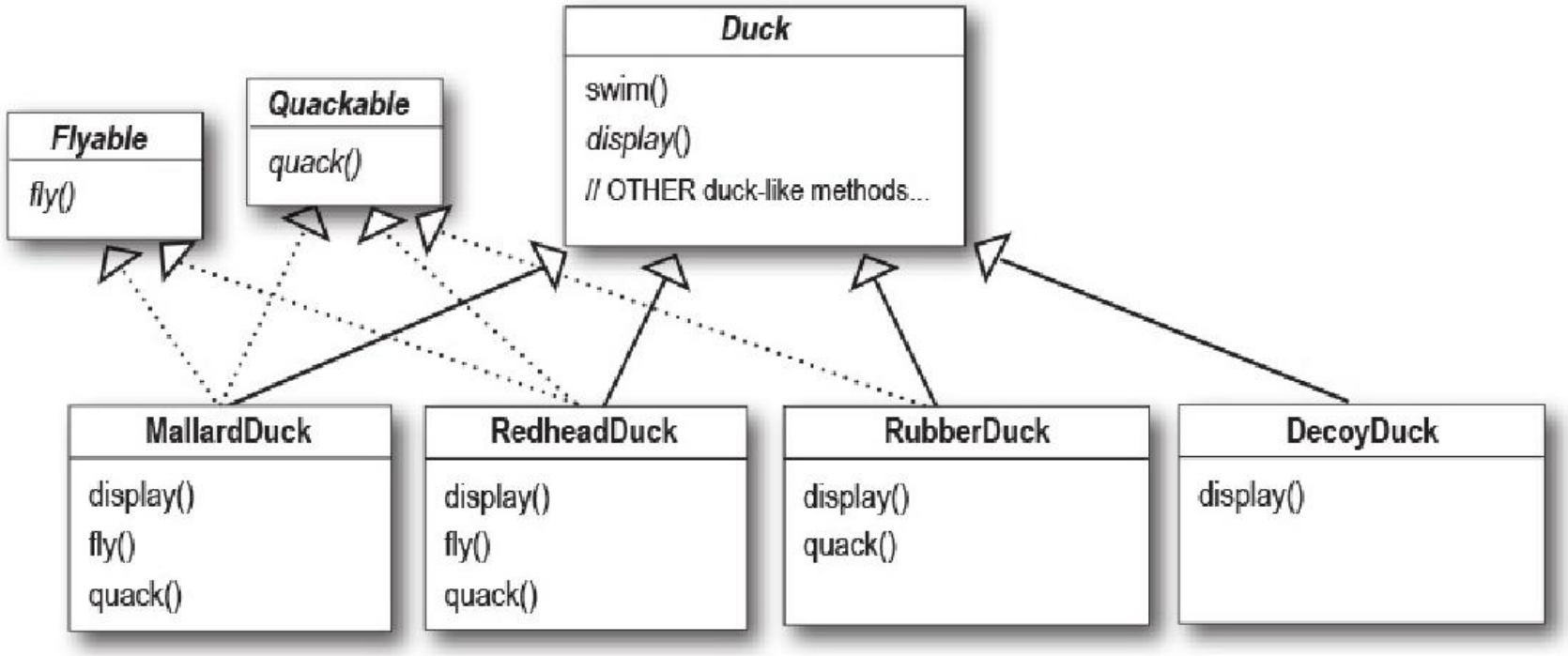
DecoyDuck

```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

```java
public class RubberDuck extends Duck{

    @Override
    public void quack() {
        System.out.println("Rubber duck not quack but squeak");
    }
    @Override
    public void display() {
        System.out.println(" rubber duck");
    }
    @Override
    public void fly() {
        //do nothing
    }
    //many other methods here
```

```java
public class DecoyDuck extends Duck{

    @Override
    public void quack() {
        //do nothing
    }
    @Override
    public void display() {
        System.out.println("it is a decoy duck");
    }
    @Override
    public void fly() {
        //do nothing
    }
```


Duck
swim()
display()
// OTHER duck-like methods...
Flyable
fly()
Quackable
quack()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()

```java
public class MallardDuck extends Duck implements Flyable, Quackable{

    @Override
    public void display() {
        System.out.println("It's a mallard");
    }
    @Override
    public void quack(){
        System.out.println("A duck quacks");
    }
    @Override
    public void fly() {
        System.out.println("A duck flies");
    }
```

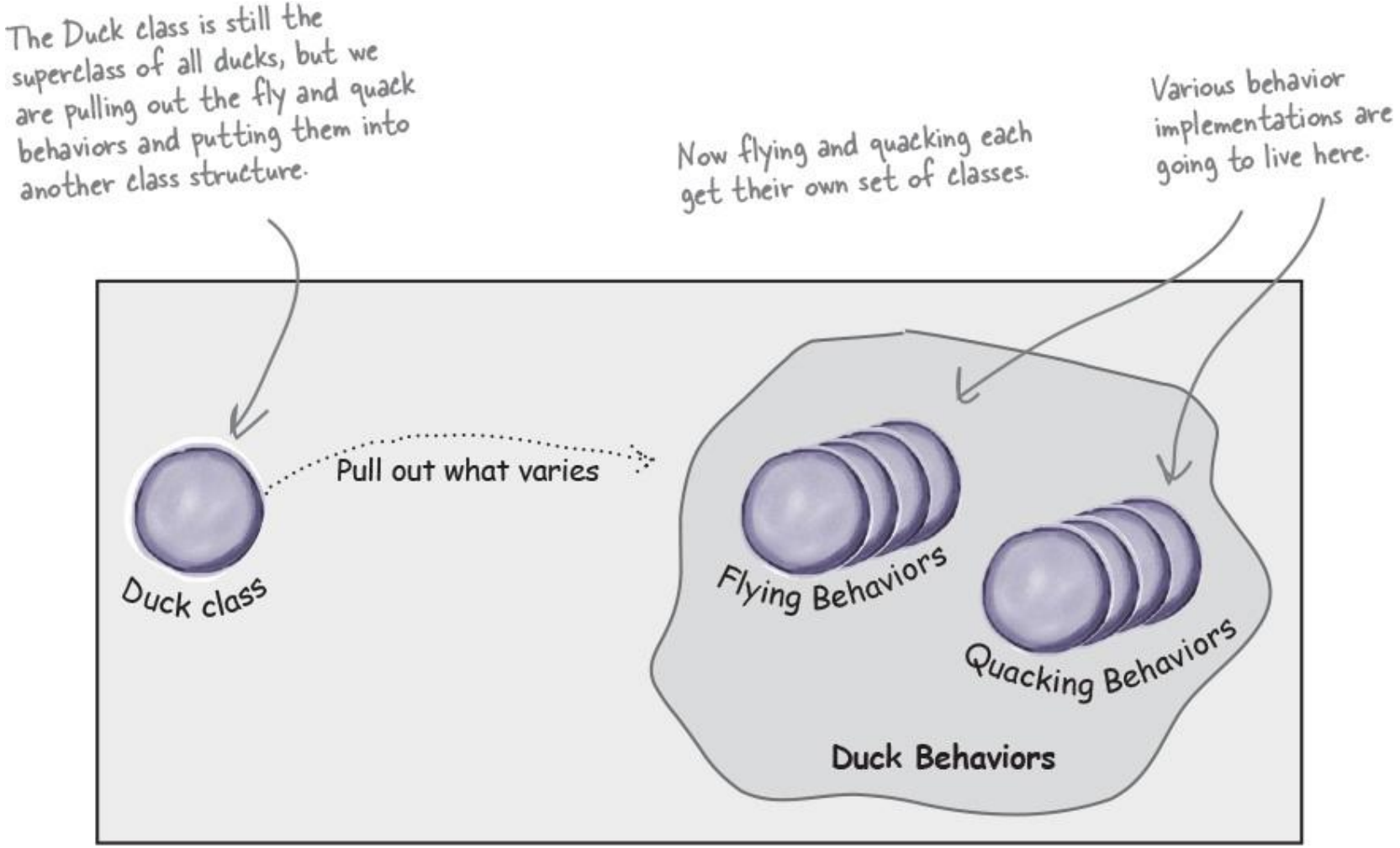
```java
public class RedheadDuck extends Duck implements Flyable, Quackable{

    @Override
    public void display() {
        System.out.println("It is a read head duck.");
    }
    public void quack(){
        System.out.println("A duck quacks");
    }
    @Override
    public void fly() {
        System.out.println("A duck flies");
    }
```

```java
public class RubberDuck extends Duck implements Quackable{

    @Override
    public void quack() {
        System.out.println("Rubber duck not quack but squeak");
    }
    @Override
    public void display() {
        System.out.println(" rubber duck");
    }
```

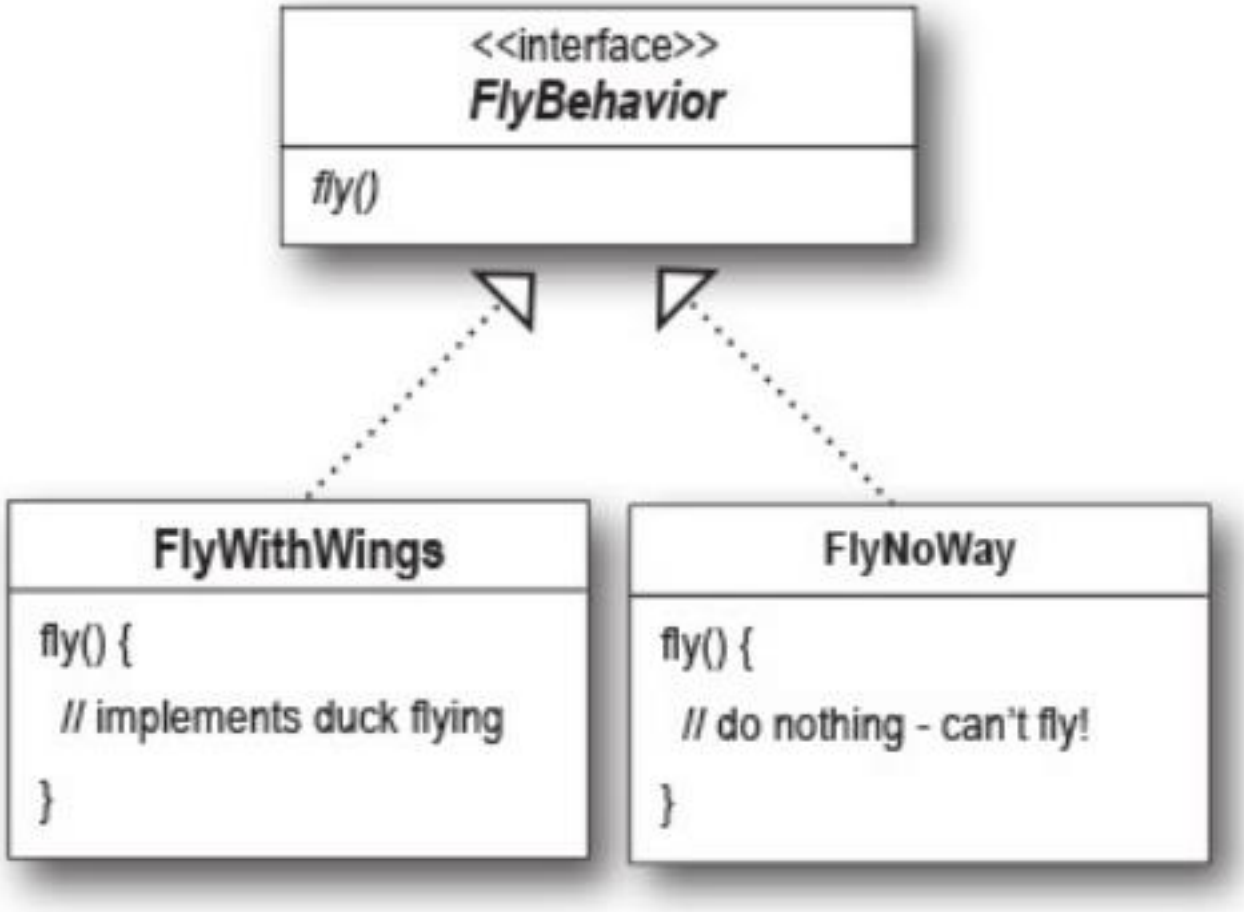
```java
public class DecoyDuck extends Duck{

    @Override
    public void display() {
        System.out.println("it is a decoy duck");
    }
```

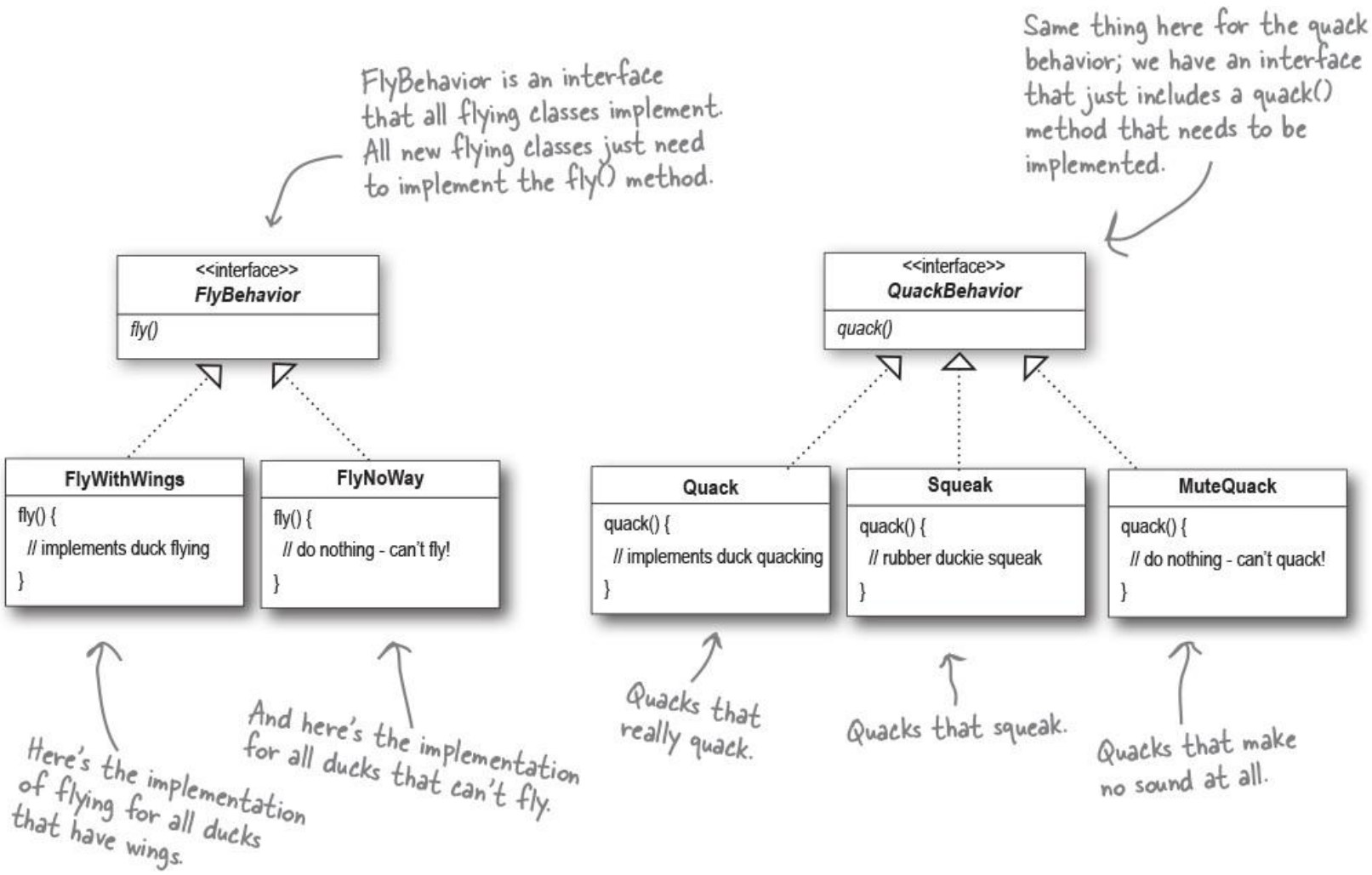
ZEROING IN ON THE PROBLEM...

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



Program to an interface, not an implementation




FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly() method.
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

```java
public interface FlyBehaviour {
    public void fly();
}
```

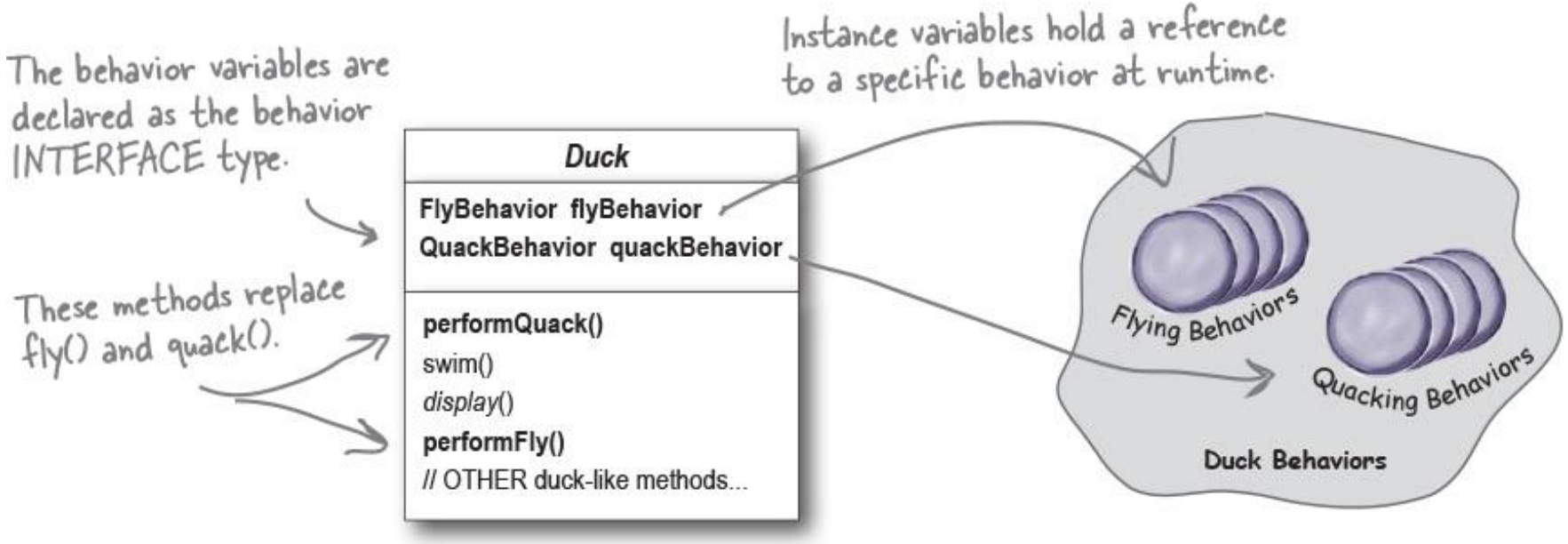
```java
public interface QuackBehaviour {
    public void quack();
}
```

```java
public class FlyWithWings implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I'm flying!!");
    }
}
```

```java
public class Quack implements QuackBehaviour{

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

```java
public class FlyNoWay implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

```java
public class Squeak implements QuackBehaviour{

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


The behavior variables are declared as the behavior INTERFACE type.
These methods replace fly() and quack().
Instance variables hold a reference to a specific behavior at runtime.
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
Flying Behaviors
Quacking Behaviors
Duck Behaviors

```java
public abstract class Duck {
    FlyBehaviour flyBehaviour;
    QuackBehaviour quackBehaviour;

    public abstract void display();

    public void performFly() {
        flyBehaviour.fly();
    }
    public void performQuack() {
        quackBehaviour.quack();
    }
    public void swim() {
        System.out.println("All duck float, even decoys!");
    }
```

```java
public interface FlyBehaviour {
    public void fly();
}
```

```java
public interface QuackBehaviour {
    public void quack();
}
```

```java
public class MallardDuck extends Duck {

    public MallardDuck() {
        flyBehaviour = new FlyWithWings();
        quackBehaviour = new Quack();
    }

    @Override
    public void display() {
        System.out.println("I'm mallard");
    }
}
```

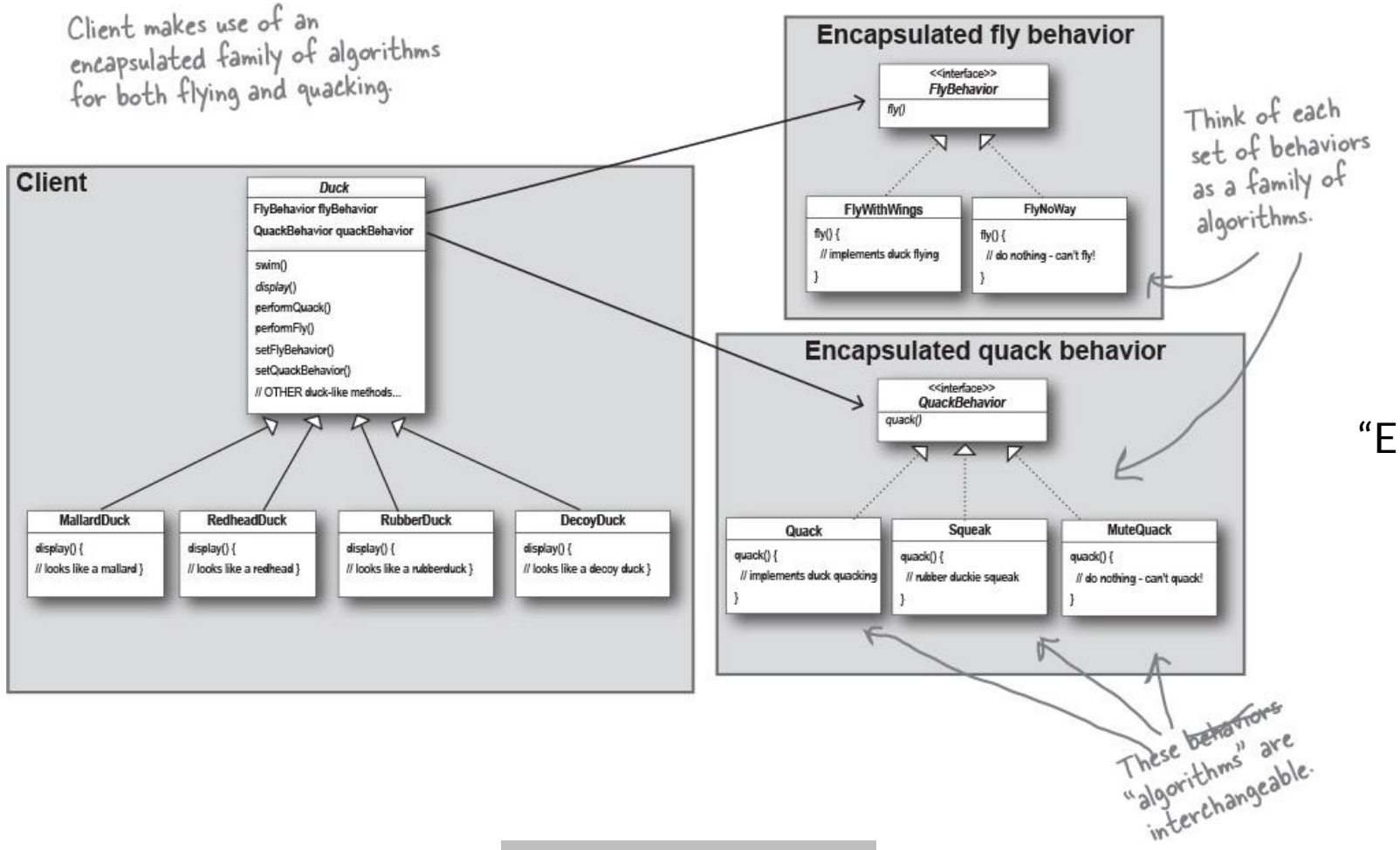
```java
public class RubberDuck extends Duck {

    public RubberDuck() {
        flyBehaviour = new FlyNoWay();
        quackBehaviour = new Squeak();
    }

    @Override
    public void display() {
        System.out.println("I'm rubber duck");
    }
}
```

```java
public class RedheadDuck extends Duck {

    public RedheadDuck() {
        flyBehaviour = new FlyWithWings();
        quackBehaviour = new Quack();
    }

    @Override
    public void display() {
        System.out.println("I'm readhead");
    }
}
```

```java
public class MiniDuckSimulator {

    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.display();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

```
run:
I'm mallard
Quack
I'm flying!!
```

"Program to an interface, not an implementation"



"Favor composition over inheritance"

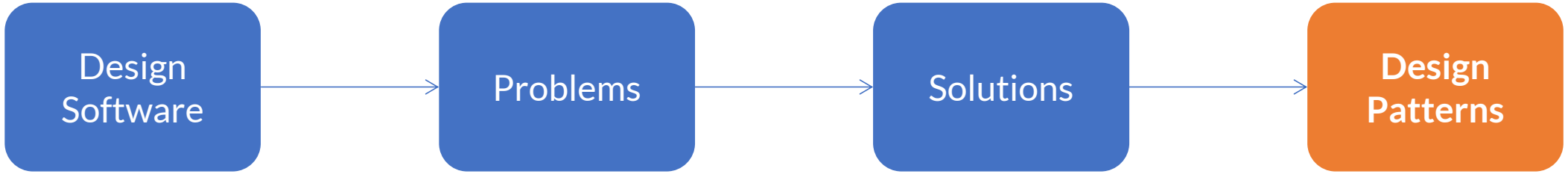
"Encapsulate what varies"

STRATEGY PATTERN

DESIGN PATTERNS

Typical solutions to common problems in software design.

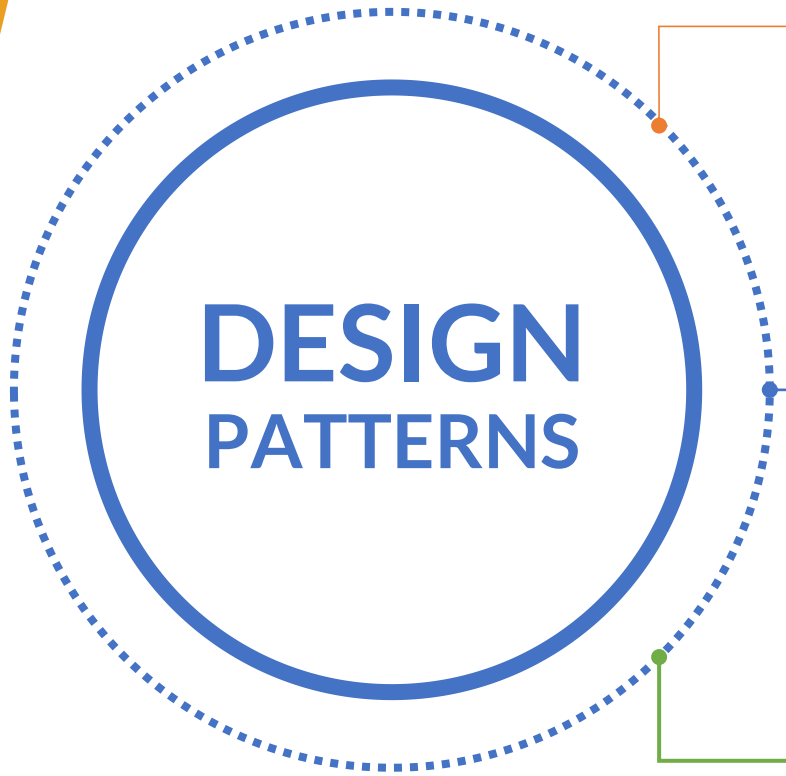
Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.



BENEFIT OF DESIGN PATTERNS

- Desain perangkat lunak menjadi lebih baik.
- Perangkat lunak yang dikembangkan mudah di-*maintenance*.
- Komunikasi dengan tim pengembang menjadi lebih efektif

CLASSIFICATION OF PATTERNS



CREATIONAL PATTERN

provide object creation mechanisms that increase flexibility and reuse of existing code.

STRUCTURAL PATTERN

explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

BEHAVIORAL PATTERN

take care of effective communication and the assignment of responsibilities between objects.

CREATIONAL PATTERNS

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton

STRUCTURAL PATTERNS

1. Adapter
2. Facade
3. Bridge
4. Composite
5. Decorator
6. Flyweight
7. Proxy

BEHAVIORAL PATTERNS

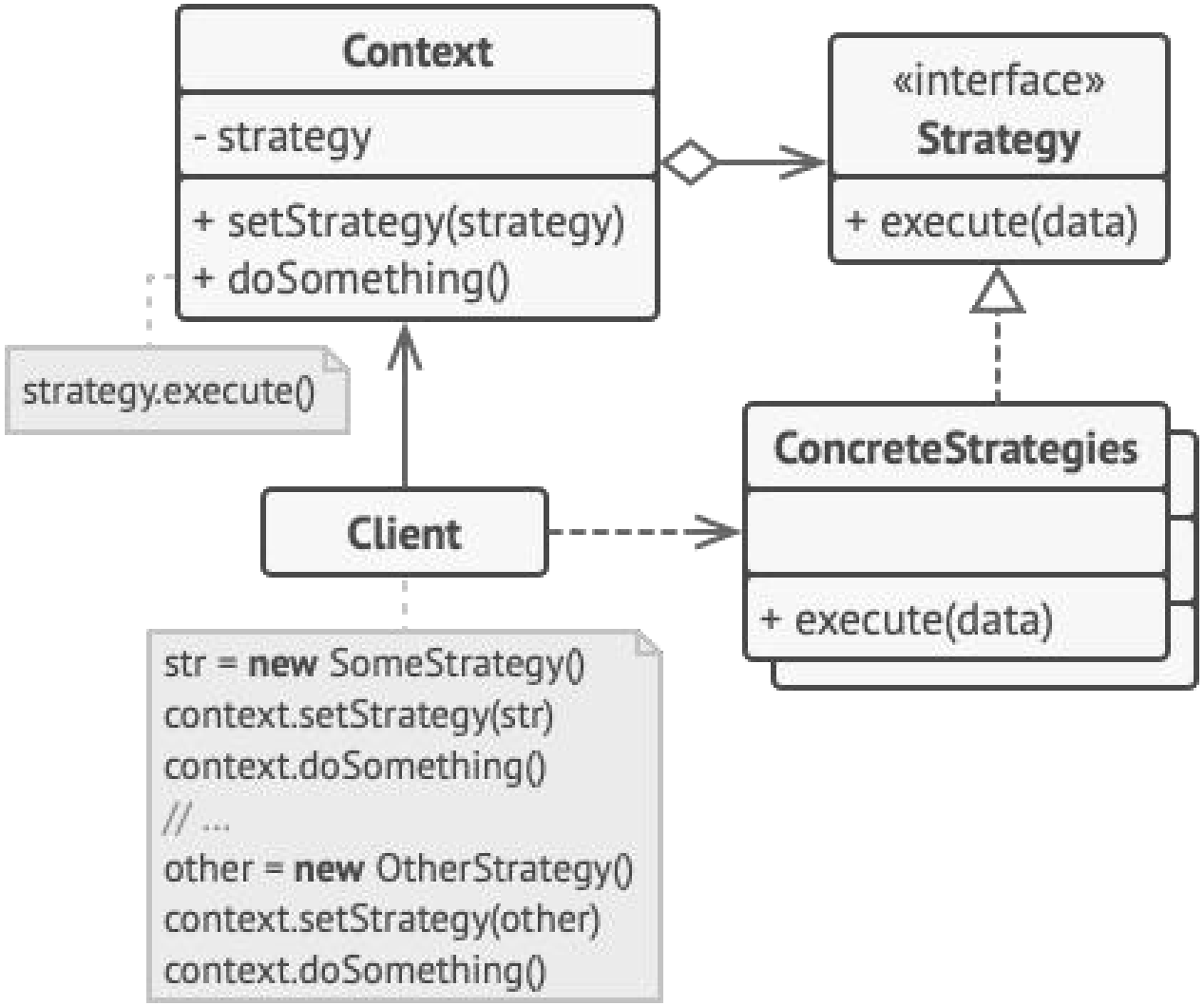
1. Chain of Responsibility
2. Command
3. Iterator
4. Mediator
5. Memento
6. Observer
7. State
8. **Strategy**
9. Template Method
10. Visitor

STRATEGY PATTERN

Strategy Pattern

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

STRUCTURE


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
strategy.execute()
Client
ConcreteStrategies
+ execute(data)
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

Context

```
public abstract class Duck {
    FlyBehaviour flyBehaviour;
    QuackBehaviour quackBehaviour;

    public abstract void display();

    public void performFly() {
        flyBehaviour.fly();
    }
    public void performQuack() {
        quackBehaviour.quack();
    }
    public void swim() {
        System.out.println("All duck float, even decoys!");
    }
```

Client

```
public class MiniDuckSimulator {

    public static void main(String[] args) {
        Duck mallar = new MallardDuck();
        mallar.display();
        mallar.performFly();
        mallar.setFlyBehaviour(new FlyRocketPowered());
        mallar.performFly();
    }
}
```

Strategy

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

Concrete Strategies

```
public class FlyWithWings implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I'm flying!!");
    }
}
public class FlyNoWay implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

APPLICABILITY

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.
- Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.
- Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.

HOW TO IMPLEMENT

- In the context class, identify an algorithm that's prone to frequent changes.
- It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.

Context

```
public abstract class Duck {

    public void quack(){
        System.out.println("A duck quacks");
    }
    public void swim(){
        System.out.println("A duck swims");
    }
    public void fly(){
        System.out.println("A duck flies");
    }
    public abstract void display();
    //many other methods here
```

HOW TO IMPLEMENT

Declare the strategy interface common to all variants of the algorithm.

One by one, extract all algorithms into their own classes. They should all implement the strategy interface.

Interface Strategy

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

Concrete Strategies

```
public class FlyWithWings implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I'm flying!!");
    }
}
public class FlyNoWay implements FlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

HOW TO IMPLEMENT

- In the context class, add a field for storing a reference to a strategy object.
- Provide a setter for replacing values of that field.
- The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.

Context

```
public abstract class Duck {
    FlyBehaviour flyBehaviour;

    public void setFlyBehaviour(FlyBehaviour flyBehaviour) {
        this.flyBehaviour = flyBehaviour;
    }

    public void performFly() {
        flyBehaviour.fly();
    }
```

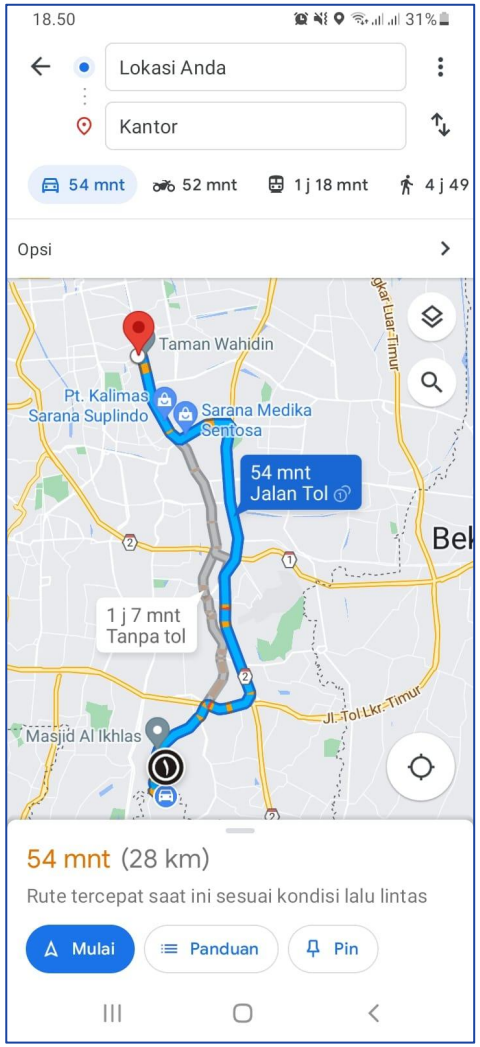
HOW TO IMPLEMENT

- Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

Client

```
public class MiniDuckSimulator {

    public static void main(String[] args) {
        Duck mallar = new MallardDuck();
        mallar.display();
        mallar.performFly();
        mallar.setFlyBehaviour(new FlyRocketPowered());
        mallar.performFly();
    }
}
```

CASE STUDY



- Sebuah aplikasi memiliki fitur untuk menentukan rute dua titik lokasi. Fitur ini memungkinkan pengguna untuk memilih rute terbaik sesuai moda transportasi yang dipilih.
- Buatlah desain kelas dari fitur aplikasi tersebut menggunakan *class diagram* dengan menerapkan *strategry pattern*. Selanjutnya implementasikan dengan bahasa pemrograman Java.

REFERENCES


O'REILLY
Second Edition
Head First
Design Patterns
Building Extensible & Maintainable Object-Oriented Software
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates
A Brain-Friendly Guide


Dive Into
DESIGN PATTERNS
Alexander Shvets

see you next chapter...
DESIGN PATTERNS
PART II