By Linkedin :[Eman A. Hjazi]

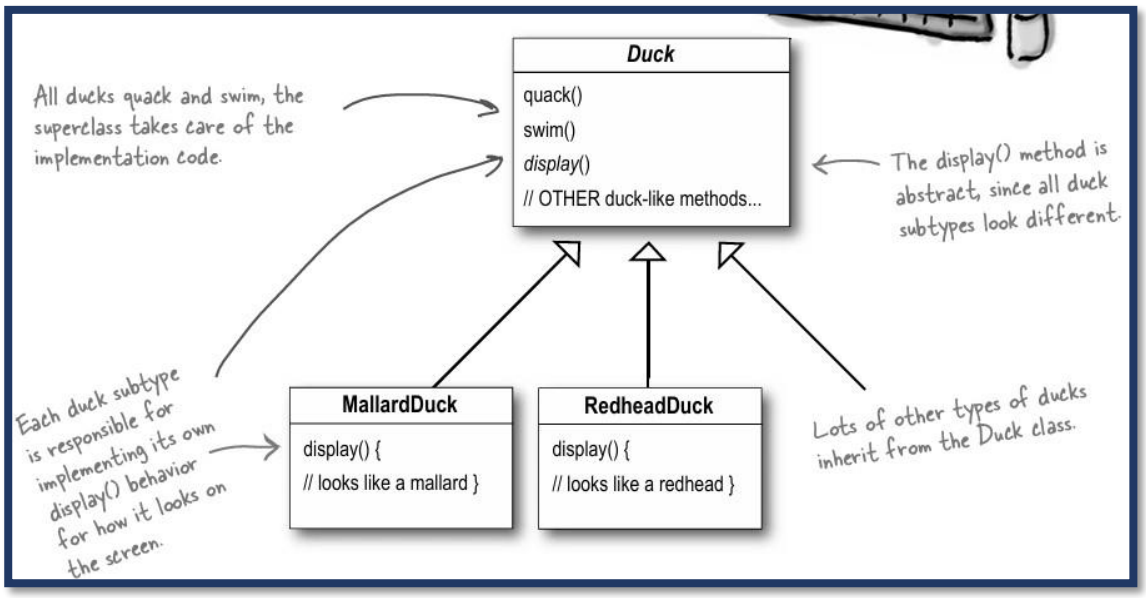
OOP Design Principles

بدنا نشرح مثال من كتاب ***Head First Design Patterns*** ومن خلاله بنوصل للمبادئ بس بالبداية خلينا نعرف انه أي تطبيق بنعمله او كود دائما معرض لتغييرات لانه هذا الشيء ثابت في أي برنامج بنعمله مثلا ممكن ينطلب مني add feature او new requirements الخ وهذا بيؤدي ممكن لتغير كامل في التصميم

In Application development: The Only Constant is Change

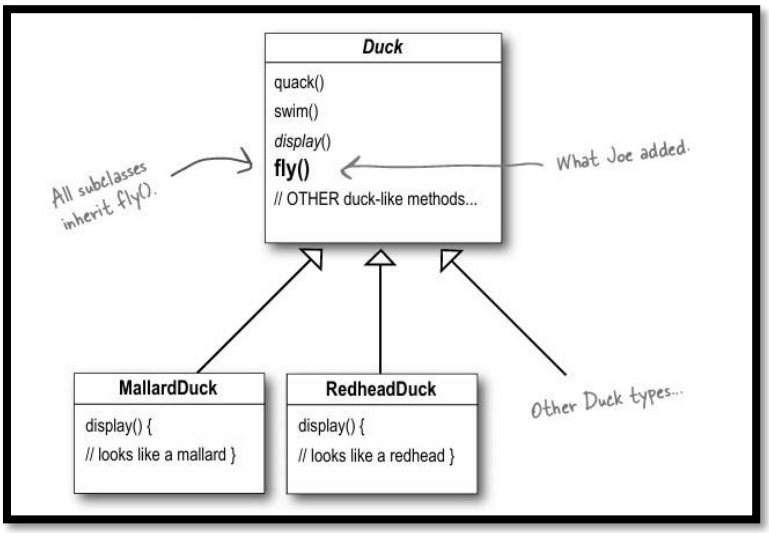
"SimuDuck" app

An application for showing ducks that can swim & make sounds



Requirement Changes

1- A new requirement came in to **add fly feature**



2- This time a new requirement came in to add a **new types of ducks Rubber ducks**

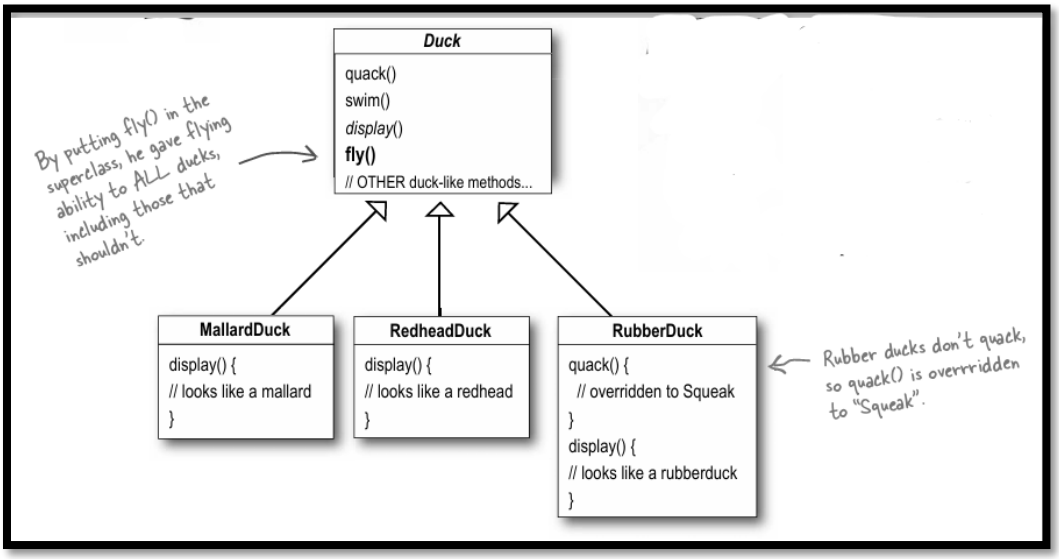
By Linkedin :Eman A. Hjazi

● ***Solution 1:***

○ Create a new class "**RubberDuckClass**" which inherits the Duck superclass

■ Override the quack() method as this type does not make sense (This kind of Duck doesn't quack) **A great use of inheritance for the purpose of code reusability**

What is wrong with this design?

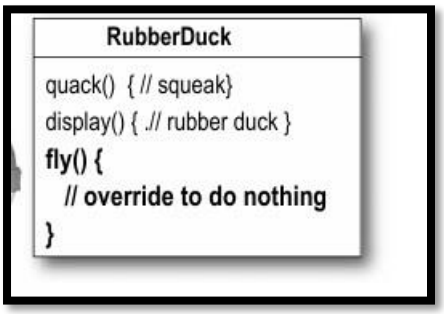


Adding the **fly() method** in the superclass gave the flying feature to all subclasses

RubberDuck في featureاخدت مش لازم تاخده بسبب **inheritance**

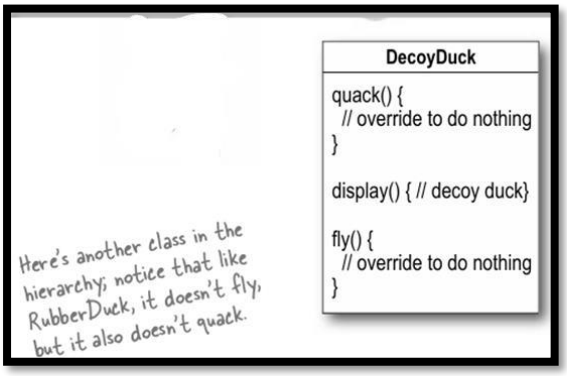
● Solution: Fix to the issue/ **inheriting** an ***inappropriate feature***

○ Override the fly() method in the same way as the quack() was overridden



● The issue now is that fly() and quack() methods needs to be implemented in each subclass Every Subclass needs to override the fly feature fly(),quack() for example is abstract in the superclass, so it has to be overriden in each concrete subclass

By Linkedin :Eman A. Hjazi

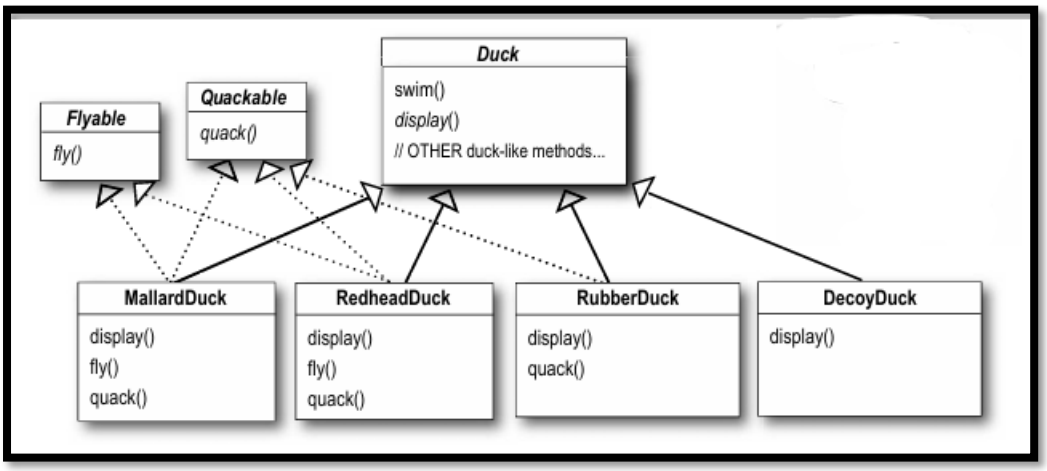


وهيك راحت فائدة inheritance انه نقلل من تكرر الكود فصرت اروح عل كل subclass ونعمل إعادة كتابة الكود لكل دالة

We know that not all of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer

Why this is not a good solution? ***Disadvantages of using inheritance*** ● The design is inflexible

- Inheritance did not solve the problem of changing requirement
- The duck behaviour keeps changing
  - Not all features (properties) are needed by all subclasses
  - The changing features need to be overridden appropriately in each subclass
- ***Solution 2 Using Interfaces:***
- Take changing behaviour **out of the superclass**
- Have an **interface** for each behaviour that is changing
  - An interface for the flying behaviour
  - Another one for the sound behaviour
- Each subclass will implement the ones that it needs



But while having the subclasses implement Flyable and/or Quackable solves part of the problem (no inappropriately flying rubber ducks),

Only some (but not all) of the duck types fly or quack.

By Linkedin :Eman A. Hjazi

Does the interface solve the problem?

- The design is inflexible
- Interfaces do not have implementation
  - Each subclass has its own implementation
  - Changing the behaviour
    - For example, if we want to change the fly() then we need to visit all subclasses and do the changes صعب في التعديل

This destroys the code reuse, makes **code maintenance hard,** and might **introduce bugs**

Can you say, "duplicate code"? If you thought having to override a few methods was bad, how are you gonna feel when you need to make a little change to the flying behavior... in all 48 of the flying Duck subclasses ?!

The one constant in software development is Change

The Application must grow and change or it will die

التغيير معناه في شيء جديد وتحديث عل الكود بشكل دائم وهذا شيء منيح

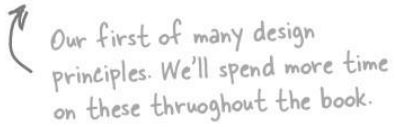
Solution 3 حسب المبادئ

First Design Pattern

Design Principle

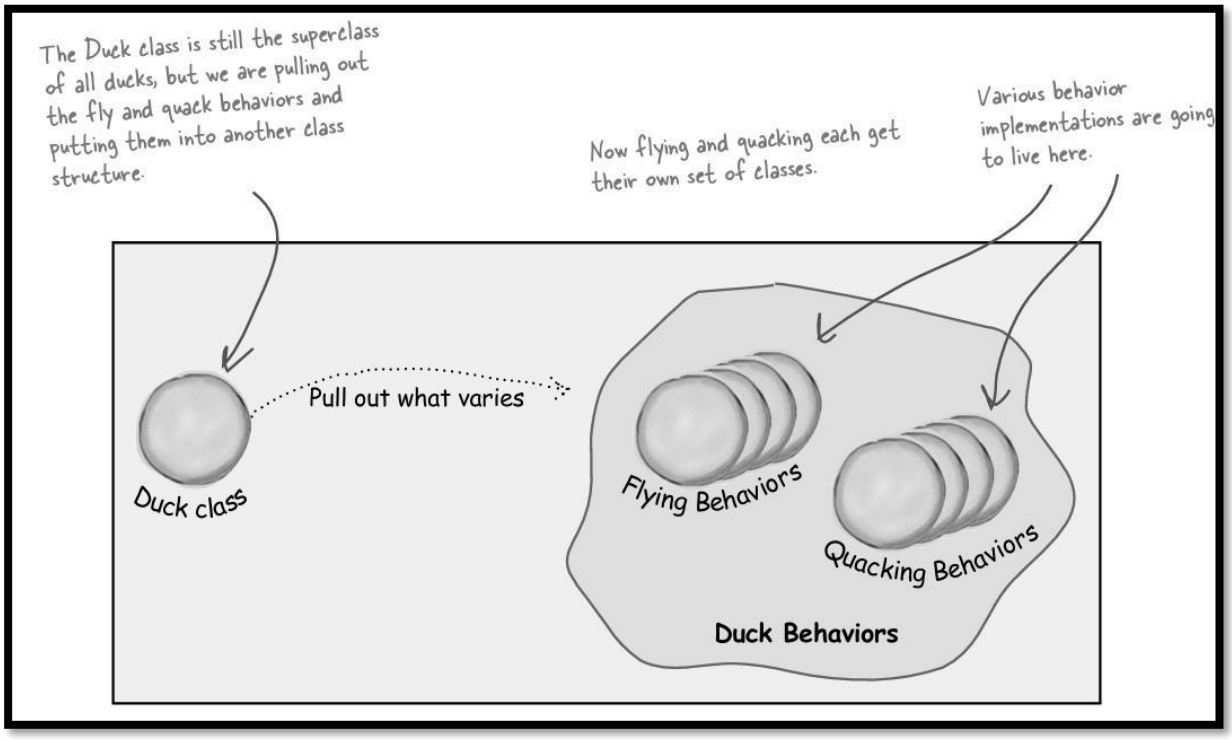
Identify the aspects of your application that vary and separate them from what stays the same.

Take the parts ***that vary and encapsulate*** them, so that later you can alter or extend the parts that vary without affecting those that don't.



فصل الأشياء الي بتتغيير وعملية implementation not in subclass مش زي قبل بس subclass بيحدد أي واحد بدو إياه لكن ما بعمل implementation الي كان بالحل السابق

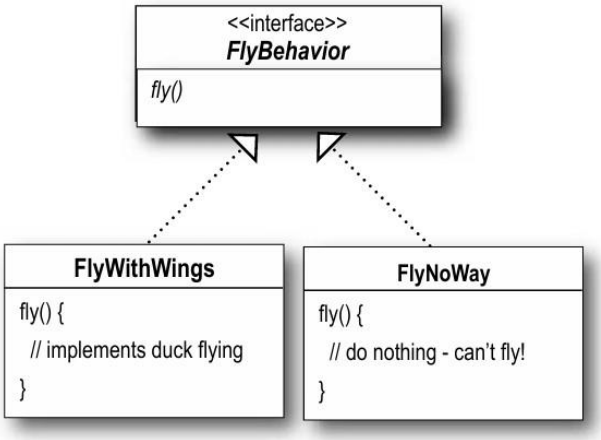
By Linkedin :Eman A. Hjazi



Encapsulate What Varies

The specific concrete behavior coded in the class that implements the FlyBehavior or QuackBehavior

المسؤول عن تحديد ال behavior هو concrete behavior implementation مش subclass ولا superclass



1- عملية الإضافة صارت سهلة عل الكود بنضيف الكلاس وبنخليه يعمل implement ل interface behavior

- modifying/extending them becomes possible without affecting parts that do not change
- Each behaviour will contain various implementation ■ Include setter methods for these behaviours in the Duck class

What did we achieve?

These behaviours can be reused, and a new type of behaviours can be easily added

By Linkedin :Eman A. Hjazi

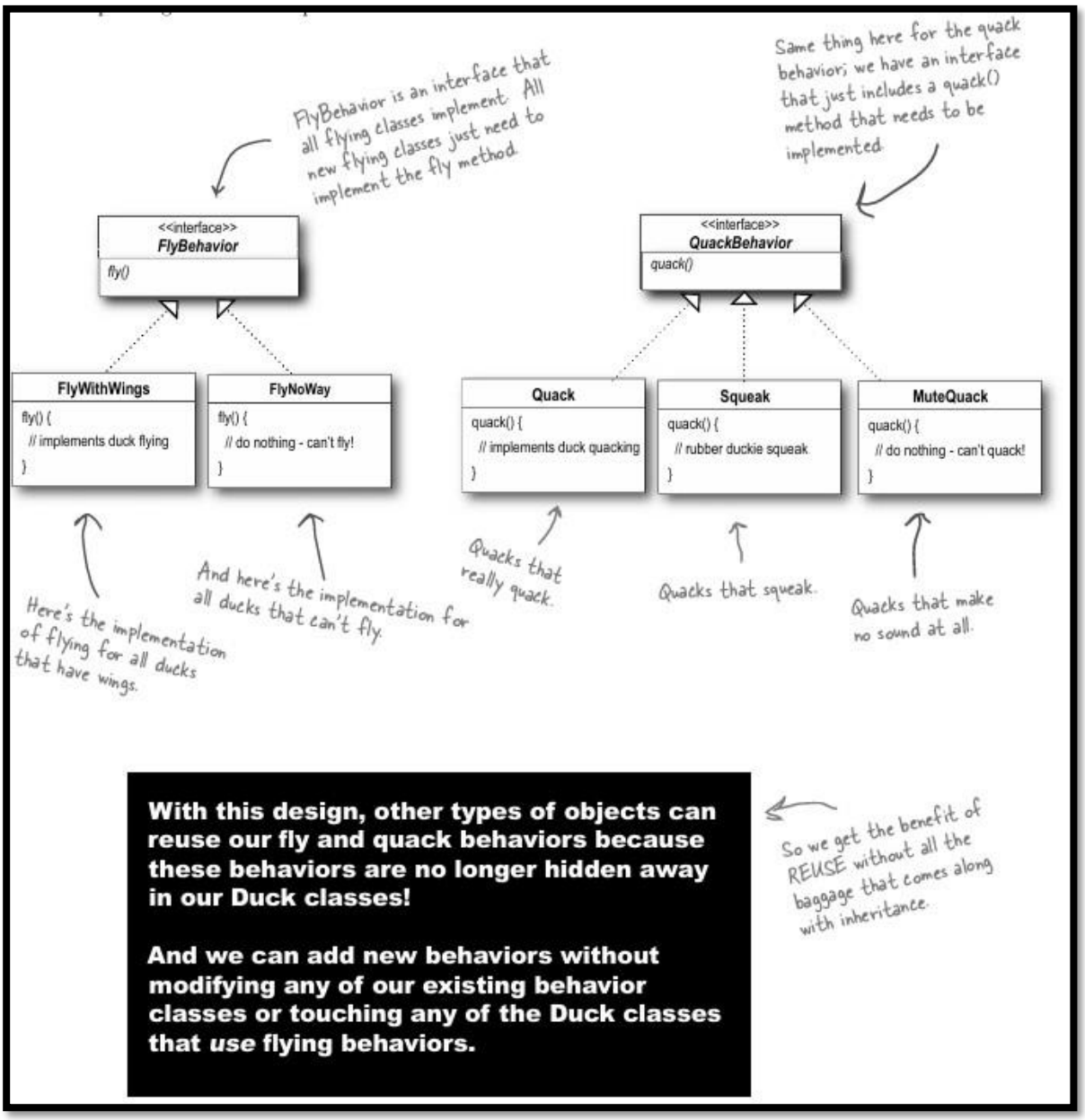
س ابقاً بالحل الأول inheritance كان **concrete implementation in the superclass (Duck class)**

حسب من أي فبتالي subclass بتاخد من superclass فالمشكلة هان ممكن يورث subclass ميزة ما بيستعملها

بالحل الثاني كان using Interface و ال subclass هو الي حينفذ implementation كلاس ماخدة

المشكلة هان كانت بالتعديل لو صار تعديل عل دالة fly معناها بدي اروح اعدل عل كل subclass والانترفيس ما فيه implementation code فبتالي كل subclass فيه الابلمنتيشن الخاص فيه وهذا بصعب عملية الصيانة بالحل هذا الثالث : Encapsulate What Varies

عملية الصيانة صارت اسهل انه لما بدي اعدل بعدل عل concrete behavior فبتالي أي كلاس مستخدميتهن مش حتتأثر هي فقط معتمدة عل concrete behavior

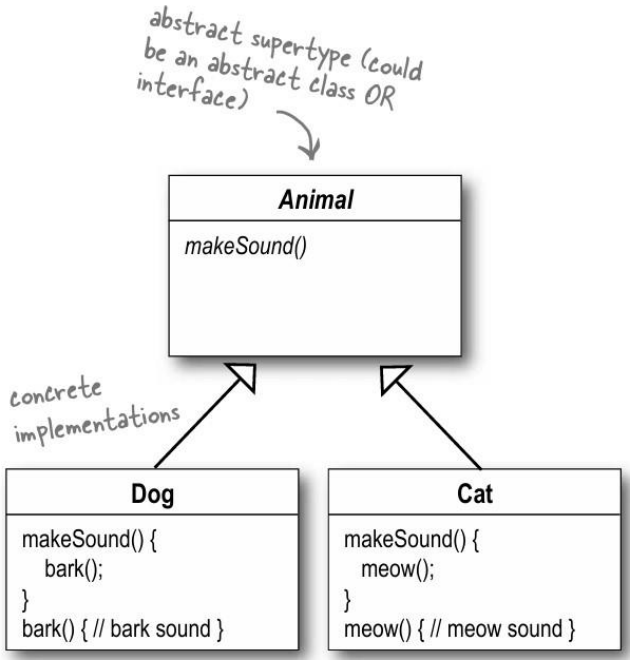


Second design principle

By Linkedin :Eman A. Hjazi

"Program to an interface " really means "Program to a supertype." supertype, usually an abstract class or interface

Imagine an abstract *class Animal*, with two *concrete implementations, Dog and Cat.*



- Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

- Program to interface would be:

```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

الأفضل انه نخلي اليوزر وقت runtime يحدد الاوبجكت الي بدو إياه ففي وقت runtime بقدر اعدل reference مثلا لو دخل حرف c بعمل instance cat اما لو دخل حرف d بعمل instance dog

Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

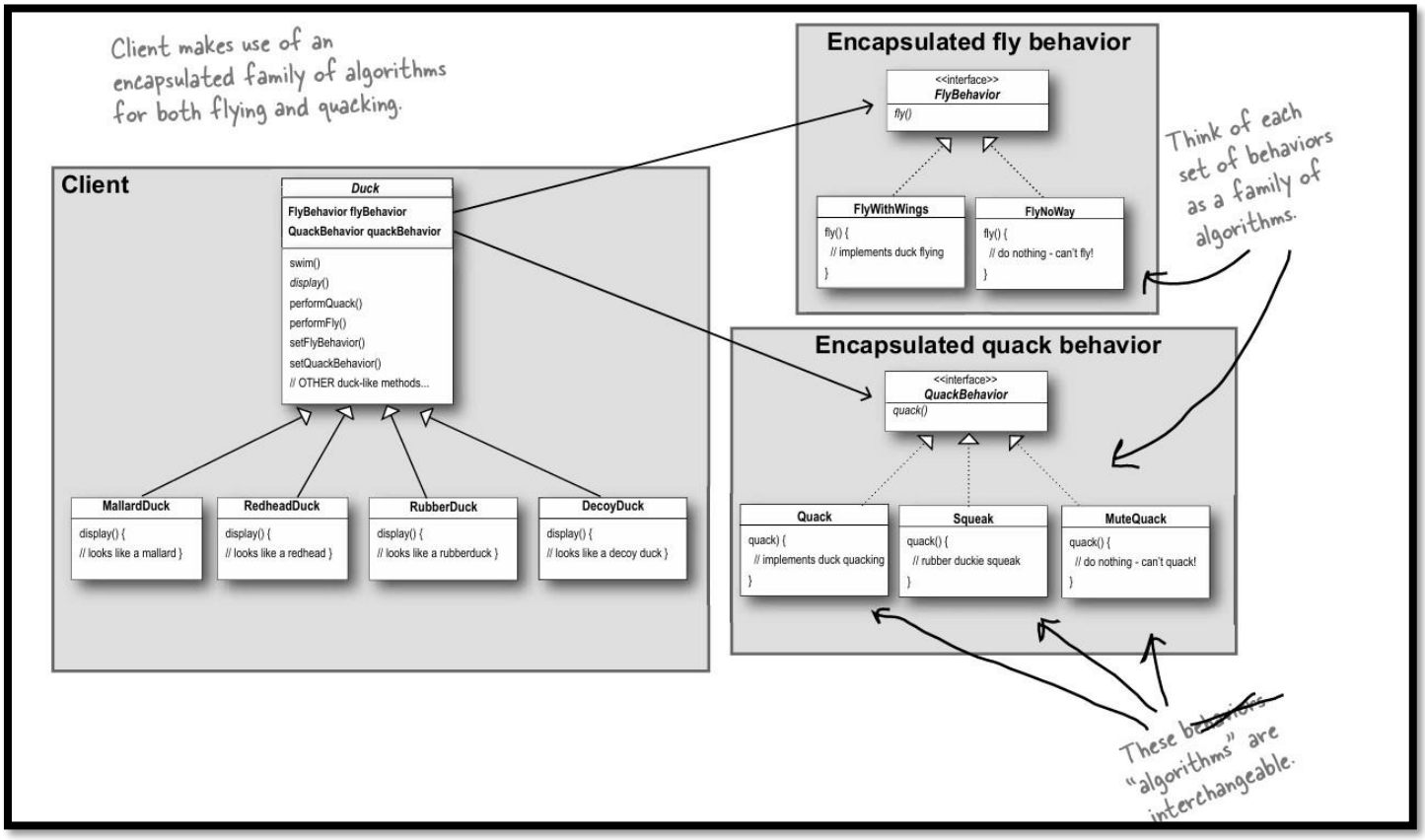
```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().

لكن لو زي الحالات الي قبل ما بقدر

• Behavior variable are declared as INTERFACE TYPE; FlyBehavior and QuackBehavior

By Linkedin :Eman A. Hjazi



setFlyBehavior()& setQuackBehavior() in Duck class can be set at runtime

Delegate Behavior :

بتعتمد عل انترفيس فما بتعرف شو نوع Behavior الي حيطبق

2 Now we implement performQuack():

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

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

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

- The Duck class delegates to the object reference quackBehavior

- Superclass only cares that the object can quack but how that happens is not important here

مش مهمته تعرف كيف بيتم عملية performQuack implementation وفوضه لكلاس QuackBehabior

فهذا المبدء: يسمح لك بالقيام بوظيفة معينة مثلا fly بأكثر من طريقة بحيث تكون قابلة للتبديل أثناء ال Runtime!، بمعنى آخر، فإنك قد تقوم ببناء وظيفة معينة يمكن تطبيقها بأكثر من طريقة، كل طريقة من هذه الطرق يمكن استخدامها محل الأخرى

By Linkedin :Eman A. Hjazi

ودون الحاجة لأي علاقة فيما بينهم سوى interface مشتركة!،

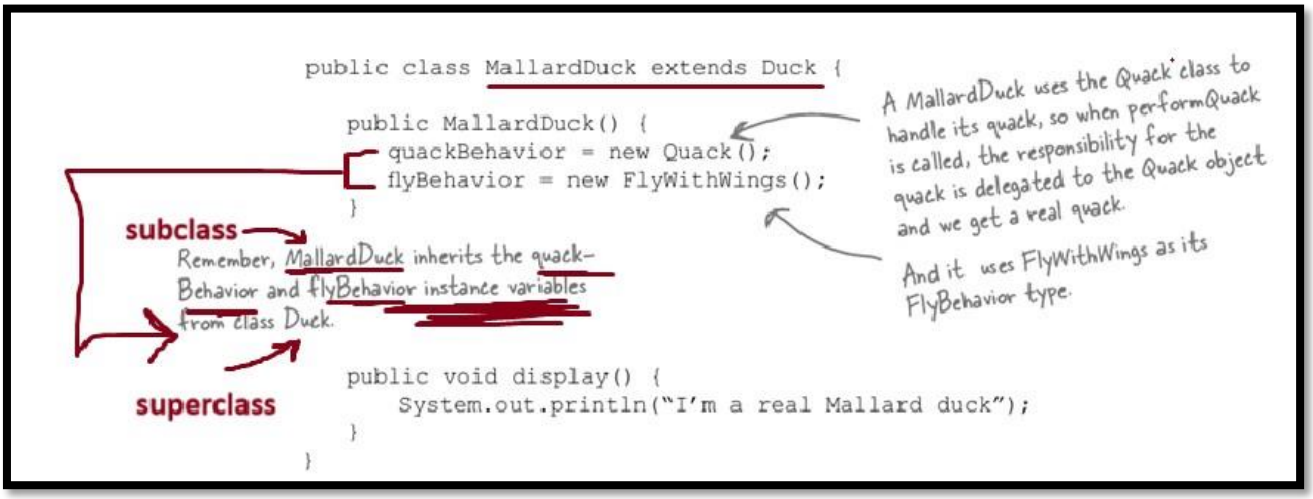
هو مبدأ يعني **تفويض تنفيذ وظيفة أو سلوك إلى كائن آخر** داخل الكلاس بدل ما ينفذه الكلاس نفسه.

ببساطة: بدل ما الكائن "يعمل الحاجة بنفسه"، هو "يطلب من كائن آخر)داخلية أوInjected (يعملها

بالنيابة عنه.

Concrete Subclass

- The subclass inherits the variable from the superclass (quackBehavior and the flyBehavior)



لو غيرنا quack فمفش داعي اغير ب subclassفبعدل بس في الكونكريت ابلمنتيشن

وحددنا نوع البهيفير هان وقت compilation , quackBehavior

Third design pattern : Favor composition over inheritance

HAS-A Relationship

- Each Duck has-a FlyBehaviour & QuackBehaviour - This is called **composition**

هو عبارة انه داخل الكلاس يكون في عندك ريفرنس ل انترفيس او ابستركشن حسب مبدأ الثاني

Instead of inheriting the fly & quack behaviours, composing the two is used

Favored **Has-A relationship** over **Is-A relationship** • favor **composition** over **inheritance**

لا يمكن الاستغناء عل الوراثة بلزمك وفي بعض الباترين بستخدم الاثنين ومش معناه انه الوراثة انه سئ

Inheritance code reuse مثال جيد عل creating systems using composition **gives you a lot more f lexibility**.

By Linkedin :Eman A. Hjazi

It let you **encapsulate a family of algorithms** into their own set of classes, but it also lets you change behavior at runtime (هو وهذا strategy pattern)

“Is inheritance always bad?”

No. Inheritance is fine when:

- You have **true IS-A** relationships
- Behavior is **stable**, not changing
- You’re not overriding half the superclass methods

If you're **fighting** the inheritance tree — it's a sign you need composition.

“Can I use both?”

Absolutely.

A `Duck` may **inherit** from an abstract `Bird`, but **compose** its `FlyBehavior` and `QuackBehavior`.

That’s not just okay — that’s often *ideal*.