**Software Construction And Development**

**“Assignment no :01”**

**Submitted To: Dr. Mukhtiar Bano**

**By: Bushra Ashraf Bhatti**

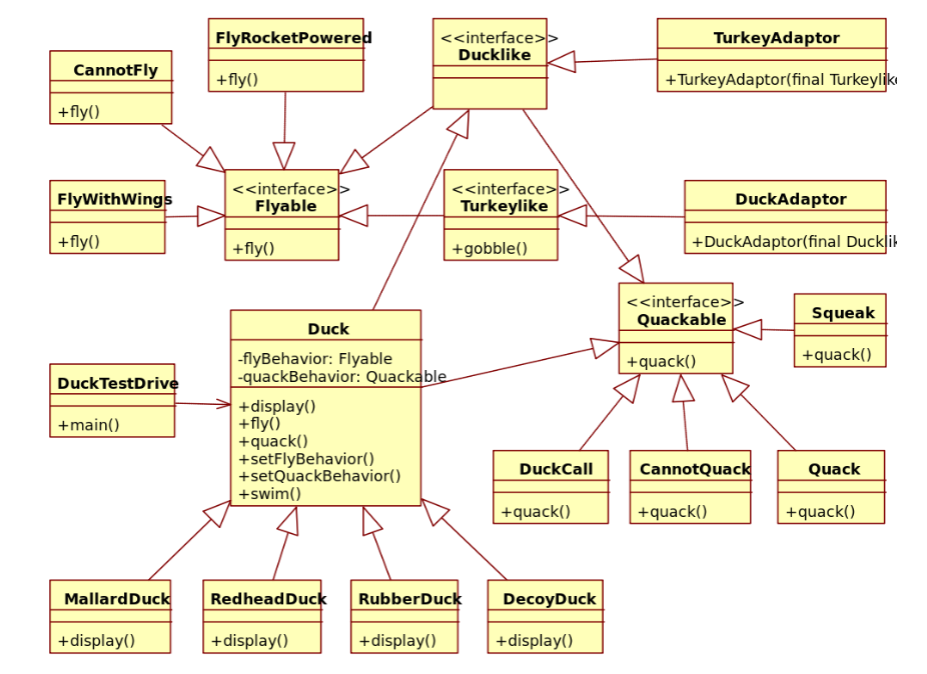
**Reg\_no: 2023-BSE-015**

**Objective**:

Of this assignment is to increase the learning capabilities of the students about Software Design in Object-Oriented Paradigm.

**Question**

Consider the below UML diagram, explain thoroughly each class, subclass, object and their associations with multiplicities.



**Over review of Diagram:** The given UML diagram represents a Duck Simulation System designed using Object-Oriented Programming (OOP) principles*.*  
It demonstrates how abstraction*,* inheritance*,* polymorphism*,* interface implementation, and composition are applied to achieve flexibility and reusability in software design.

## **Interface Classes**

### **Flyable:** Interface defines the general behavior for flying. It declares the method +fly(). Any class that implements this interface must define how the object flies.It is implemented by the classes:

### **FlyWithWings()** – normal flying using wings

* **CannotFly()** – represents no flying ability
* **FlyRocketPowered()** – represents rocket-powered flying

1. **Quackable**: This interface defines the general behavior for quacking. It declares the method **+quack()**. Any class that implements this interface must define its own way of quacking.  
   It is implemented by the classes:

* **Quack()** – normal duck sound
* **Squeak()** – squeaky toy duck sound
* **DuckCall()** – sound like a duck call device
* **CannotQuack()** – represents silence (no sound)

1. **Ducklike**: This interface represents the basic nature or characteristics of a duck.  
   It is implemented by the **Duck** class and provides a generalized structure for duck-like objects.
2. **Turkeylike:** This interface represents the behavior of a turkey. It declares the method **+gobble()**. It is implemented by the **TurkeyAdaptor** class, which adapts a turkey’s behavior to act like a duck.

## **Base (Super) Class**

### **Duck:** The **Duck** class is the main or **parent class** for all types of ducks. It defines the **common attributes and behaviors** shared by all duck types.

### **Attributes:**

* flyBehavior: Flyable
* quackBehavior: Quackable

**Methods:**

* display() – displays duck type
* fly() – performs flying behavior using Flyable interface
* quack() – performs quacking behavior using Quackable interface
* setFlyBehavior() – allows changing flying behavior at runtime
* setQuackBehavior() – allows changing quacking behavior at runtime

**Associations:**

* Each Duck **has one Flyable** behavior (multiplicity: **1**)
* Each Duck **has one Quackable** behavior (multiplicity: **1**)

**Inherited by subclasses:**

* **MallardDuck**
* **RedheadDuck**
* **RubberDuck**
* **DecoyDuck**

So, the **Duck** class acts as the **base (superclass)** for all types of ducks.

## **Derived (Sub) Classes**

These are **child classes** that inherit from the **Duck** base class.  
They reuse its structure but modify or extend specific features.

* **MallardDuck:** Inherits from Duck; overrides display() to show “I am a Mallard Duck.”
* **RedheadDuck:** Inherits from Duck; overrides display() to show “I am a Redhead Duck.”
* **RubberDuck:** Inherits from Duck; overrides display(), uses **Squeak** behavior, and **CannotFly**.
* **DecoyDuck:** Inherits from Duck; overrides display(), does not fly or quack.

Each subclass **is-a Duck** (inheritance relationship).

## **Behavior Classes (Implement Interfaces)**

These are **concrete classes** that implement the behavior interfaces to define how a duck flies or quacks.

### **Fly Behavior Classes (implement Flyable):**

* **FlyWithWings:** Represents natural flying ability.
* **FlyRocketPowered:** Represents flying using rockets.
* **CannotFly:** Represents no flying capability.

### **Quack Behavior Classes (implement Quackable):**

* **Quack:** Produces normal quack sound.
* **Squeak:** Produces a squeak sound (rubber duck).
* **DuckCall:** Mimics a duck call device.
* **CannotQuack:** Represents silence.

These behavior classes are **reusable and interchangeable**, allowing behavior to be changed at runtime.

**Test Class**

### **DuckTestDrive (Driver Class)**

This class is used for testing the system.  
It **creates objects** of Duck subclasses, assigns them behaviors, and calls their methods to show results.

* Demonstrates how different ducks behave.
* Shows that duck behavior can be changed dynamically at runtime.

## **Objects (Instances)**

Objects are **real instances** of classes created at runtime. Examples:

1. mallard = new MallardDuck()
   * flyBehavior = **FlyWithWings**
   * quackBehavior = **Quack**
2. rubber = new RubberDuck()
   * flyBehavior = **CannotFly**
   * quackBehavior = **Squeak**
3. test = new DuckTestDrive()
   * Creates and uses multiple Duck objects for testing.

## **Relationships and Multiplicities**

### **Aggregation (Has-a, Weak Ownership):** Aggregation means one class contains or uses another, but they can exist independently.

* **Duck → Flyable**
  + Multiplicity: **1 (one fly behavior reference per Duck)**
  + Each Duck object has one Flyable behavior.
  + The Flyable object can exist independently and can be shared among multiple ducks.
* **Duck → Quackable**
  + Multiplicity: **1 (one quack behavior reference per Duck)**
  + Each Duck object has one Quackable behavior.
  + The same Quackable implementation can be shared by many ducks.

### **Inheritance (Is-a Relationship)**

* **Duck → MallardDuck, RedheadDuck, RubberDuck, DecoyDuck**
  + This shows **class generalization** (no numeric multiplicity).
  + One base class can have **many derived subclasses**.
  + Each subclass **is-a Duck**.

### **Association (Uses-a Relationship)**

* **DuckTestDrive → Duck**
  + Multiplicity: **0..**\* (the driver can create many Duck objects).
  + The test class uses or interacts with multiple duck instances during execution.

### **Interface Implementation Multiplicities**

* **Flyable Interface → FlyWithWings, CannotFly, FlyRocketPowered:** One-to-Many  
  (One interface can have multiple implementing classes.)
* **Quackable Interface → Quack, Squeak, DuckCall, CannotQuack:** One-to-Many

**Multiplicities**

|  |  |  |
| --- | --- | --- |
| **Relationship** | **Multiplicity** | **Meaning** |
| Duck → Flyable | 1..1 | Each duck has one fly behavior. |
| Duck → Quackable | 1..1 | Each duck has one quack behavior. |
| DuckTestDrive → Duck | 0..\* | Test class can create and use many ducks. |
| TurkeyAdaptor → Turkeylike | 1..1 | Each adapter wraps one turkey. |
| DuckAdaptor → Ducklike | 1..1 | Each adapter wraps one duck. |
| Flyable → Implementing Classes | 1..\* | One interface can have multiple implementations. |
| Quackable → Implementing Classes | 1..\* | One interface can have multiple implementations. |
| Duck (Base) → Subclasses | 1..\* | One base class can have many derived subclasses. |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***