SimUDuck

Group17 Members:

MSCS18002  
MSEE19001  
MSCS20001

https://github.com/risaldar/SimUDuck\_Kivy

# Kivy Framework (kivy.org)

The framework that we use in **Kivy** framework to make the SimUDuck application.

Kivy is a free and open source Python framework for developing mobile apps and other applications with user interface. Kivy is python based.

# Composite Pattern

In the SimUDuck application we have used composite pattern. The purpose of using this pattern is to treat graphical objects alike. We have different types of ducks and different types of layouts (or Coops).

We have a DuckComponentClass abstract class, it is used to implement the Composite Pattern For Duck and Coop Classes.

class DuckComponentClass:

def setFlyBehavior(self, behavior):

def setQuackBehavior(self, behavior):

def performFly(self):

def performQuack(self):

def getColumns(self):

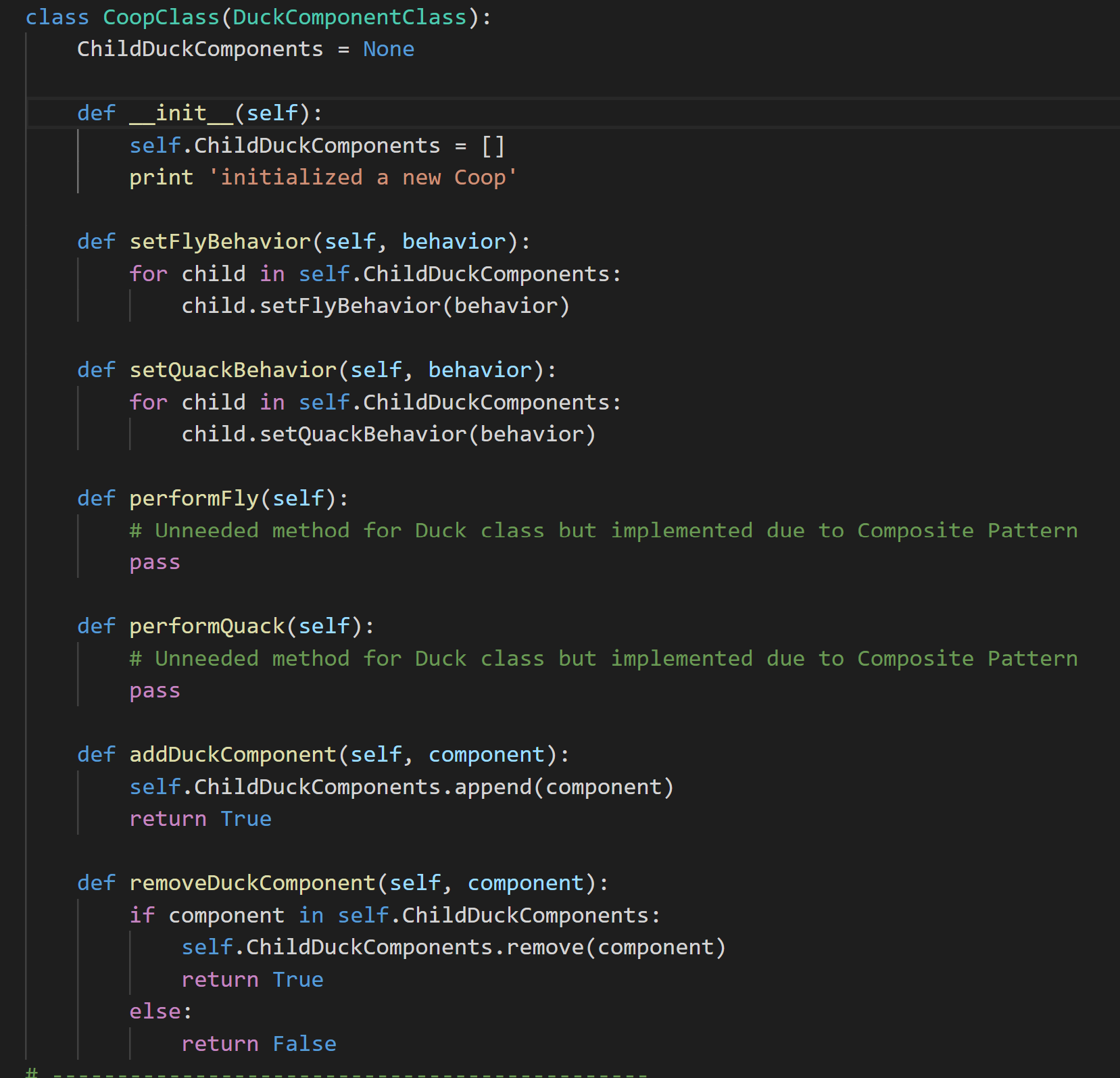
def getRows(self):

def addDuckComponent(self, component):

def removeDuckComponent(self, component):

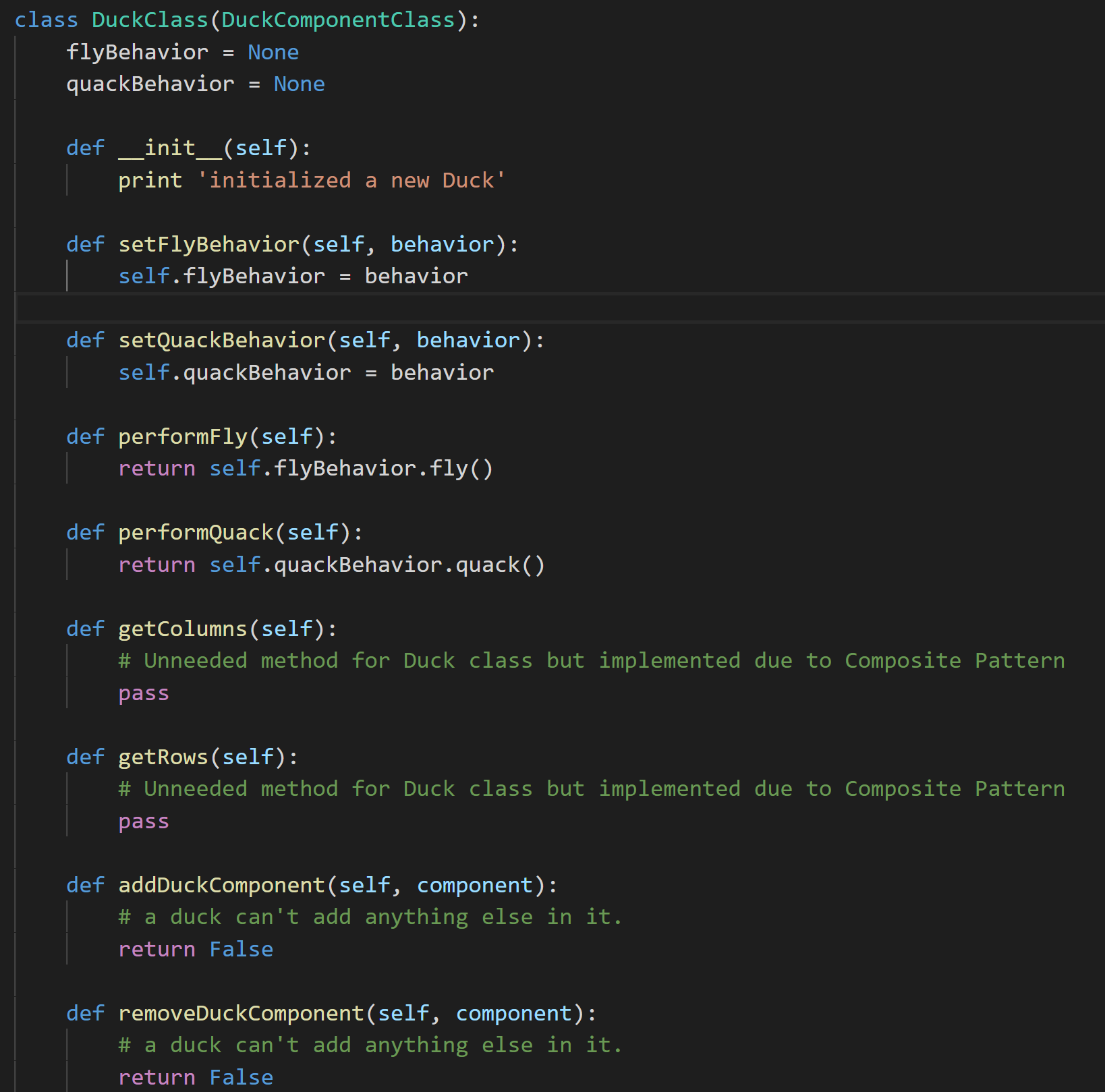
## CoopClass

The CoopClass inherits from the DuckComponentClass. If a new layout needs to be added, it shall be added by inheriting this class.



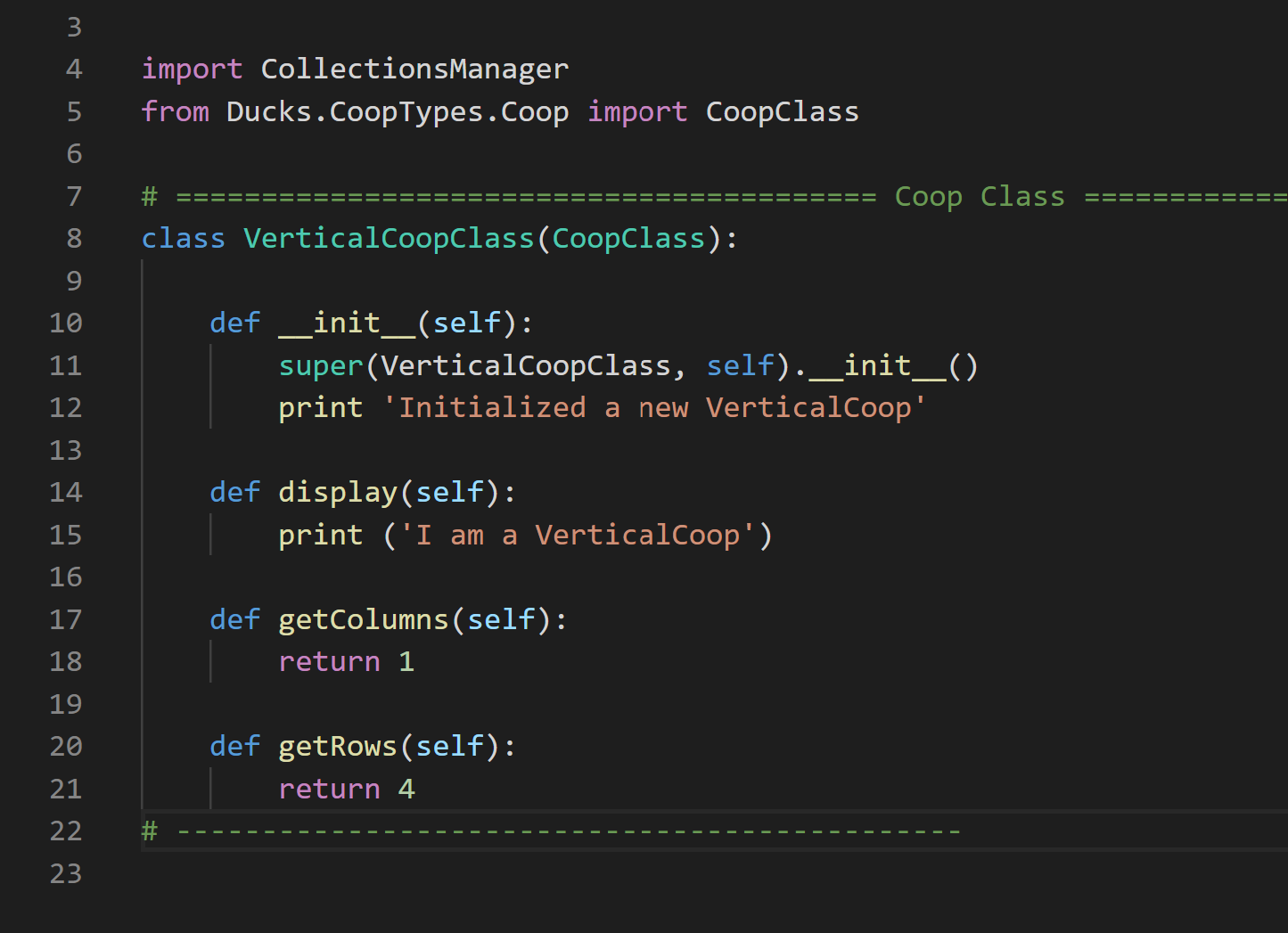
## DuckClass

The DuckClass inherits from the DuckComponentClass. If a new Duck needs to be added, it shall be added by inheriting this class.



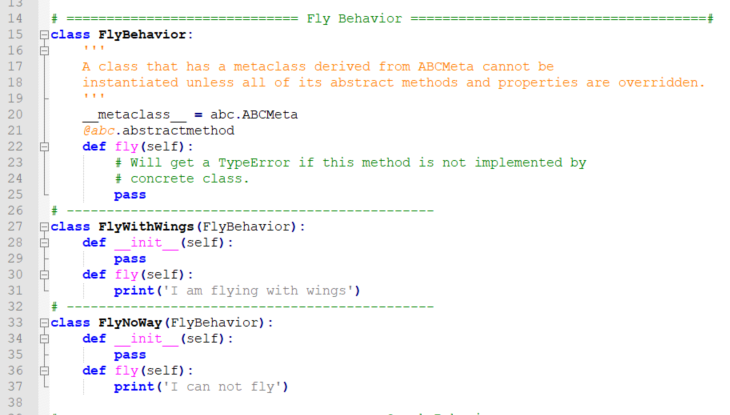
## VerticalCoopClass

The VerticalCoopClass inherits from the CoopClass. It allows vertical components which can be Coops or Ducks. Likewise, we also have HorizontalCoopClass.

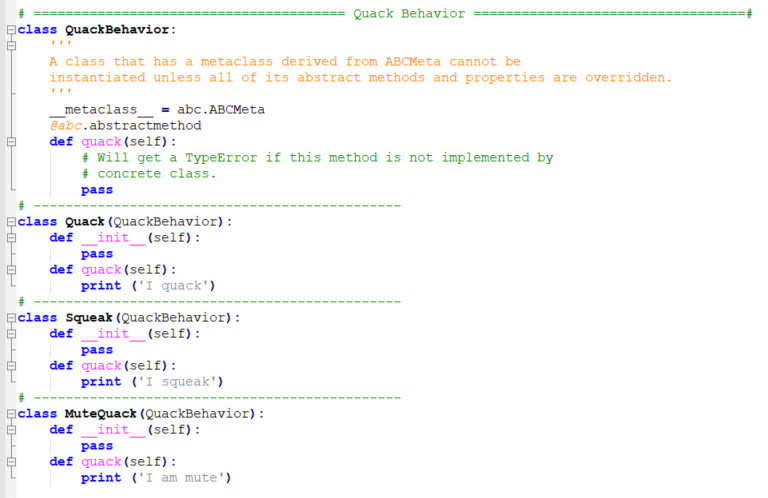


# Strategy Pattern

The SimUDuck application is developed using the strategy pattern with the principle “program to interface, not to implementation”. This is achieved by having *FlyBehavior* and *QuackBehavior* interfaces. For *FlyBehavior* interface the implementation is present in *FlyWithWings*, *FlyNoWay* classes.



Similarly for *QuackBehavior*, the behaviors are implemented in *Quack*, *Squeak* classes.



We have the parent DuckClass and all types of duck inherit from this parent class. The DuckClass has composition of *FlyBehavior* and *QuackBehavior* maintaining a HAS A relationship. This can be seen here:



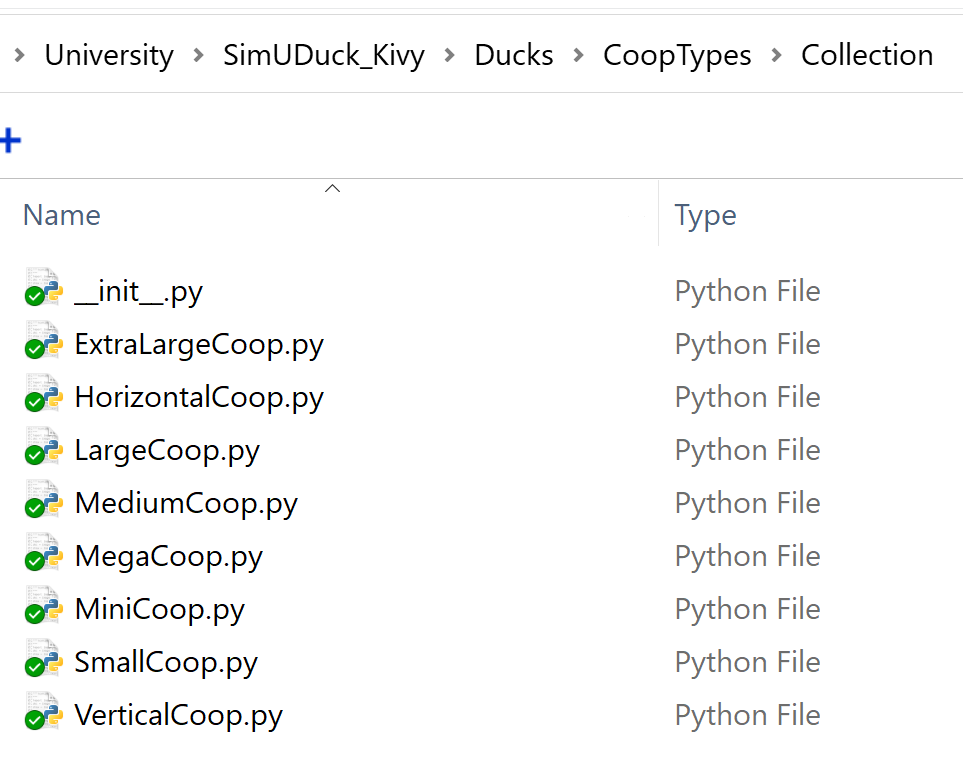
All new duck types inherit from the same parent DuckClass.

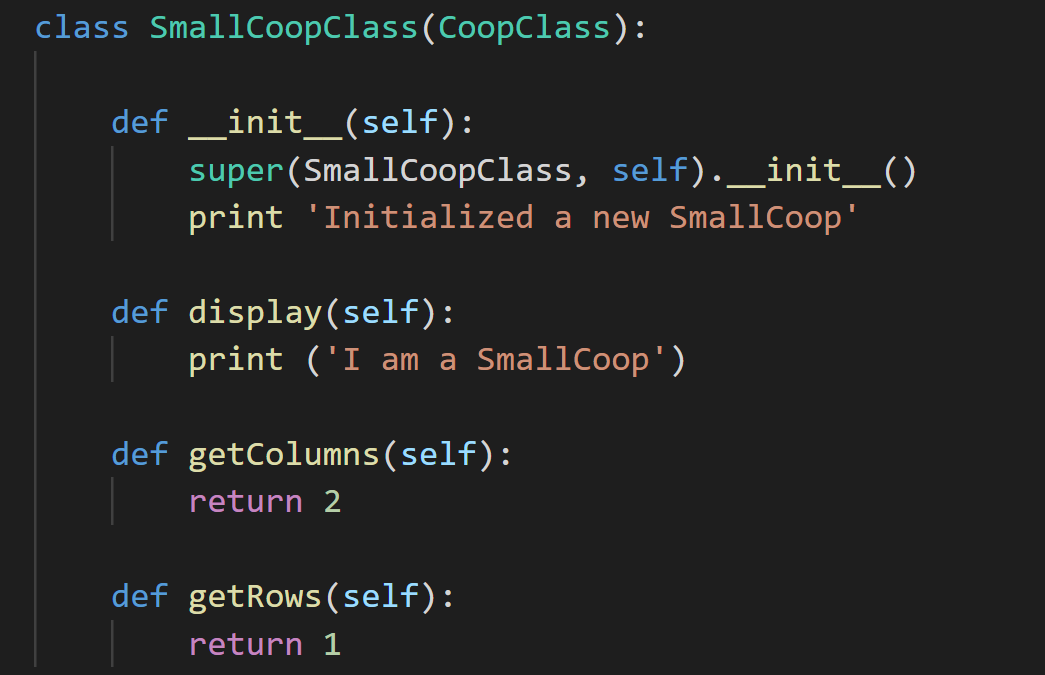
# Extending the application:

## Adding New Layout (and Dynamic Loading)

If a new layout needs to be added, a new .py file should be added in folder CoopTypes\Collection.

The new layout class shall be implemented in it inheriting the CoopClass.





### Adding a new Duck type:

To add a new duck we need to inherit a new class from DuckClass and have the display implementation.

Class NewDuckClass(DuckClass)

def \_\_init\_\_(self):

print ('Initialized a new Duck')

def display(self):

print ('I am a new Duck')

### Adding a new FlyBehavior or QuackBehavior:

To add a new FlyBehavior or QuackBehavior, we add a new class that implements the interface class FlyBehavior and QuackBehavior respectively.

class NewFly(FlyBehavior):

def \_\_init\_\_(self):

pass

def fly(self):

print('I am a new flying’)

class NewQuack(QuackBehavior):

def \_\_init\_\_(self):

pass

def quack(self):

print ('I am a new quack’)

### Dynamically changing Behavior:

Because of strategy pattern, we can also change the behavior of the duck at runtime. That is by using SetFlyBehavior() and SetQuackBehavior() functions.

### Polymorphism

We have also achieved polymorphism because of using inheritance. No matter which type of duck it is, we can simply call the performFly() and performQuack() functions, and the specific behavior of flying and quacking will be displayed.

New\_Mallard = MallardDuckClass()

New\_Rubber = RubberDuckClass()

New\_Mallard.performFly()

New\_Mallard.performQuack()

New\_Rubber.performFly()

New\_Rubber.performQuack()

# Design Principles

### Single Responsibility Principle

* Separate classes deal with different aspects of interface

### Open/Closed Principle

* We can add our own layouts

### Liskov Substitution Principle

* Proper hierarchy is maintained for inheritance

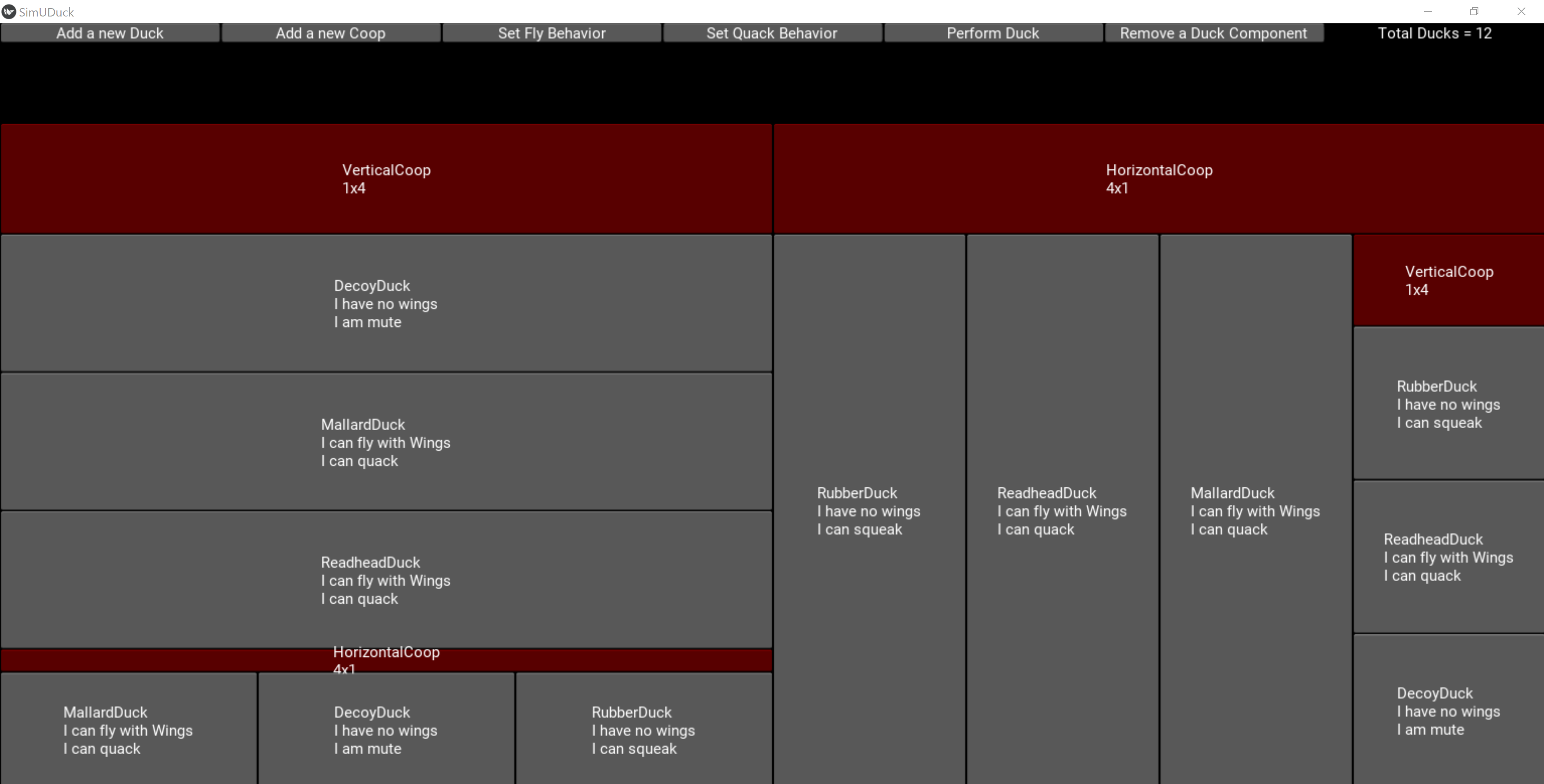
### Interface Segregation Principle

* We have class classes in the framework which perform separate tasks

### Dependency Inversion Principle

* High level components do not depend on low level components in the framework

# Application Screenshots



Old Version:

