Software Architecture & Design SEC3071

Lecture No. 38

Muhammad Shahid
Department of Computer Science
National Textile University

shahid.abdullah@hotmail.com

Last Lecture Review

- Structural Design Patterns
- Decorator Design Pattern
  - Intent
  - Definition
  - Class Diagram
  - Code Implementation
- Decorator – Example
- Decorator – Pros & Cons

Agenda – What will you Learn Today?

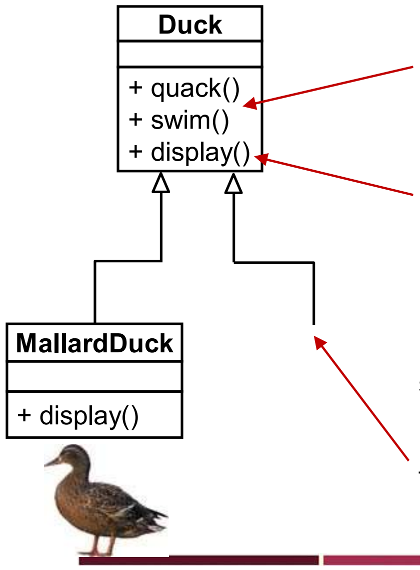
Strategy Design Pattern

Strategy Design Pattern

Let’s Design a Game !!!!!

- Joe works at a company that produces a simulation game called SimUDuck
- He is an OO Programmer and his duty is to implement the necessary functionality for the game
- The game should have the following specifications:
  - A variety of different ducks should be integrated into the game
  - The ducks should swim
  - The ducks should quake

A First Design of a Duck Simulator



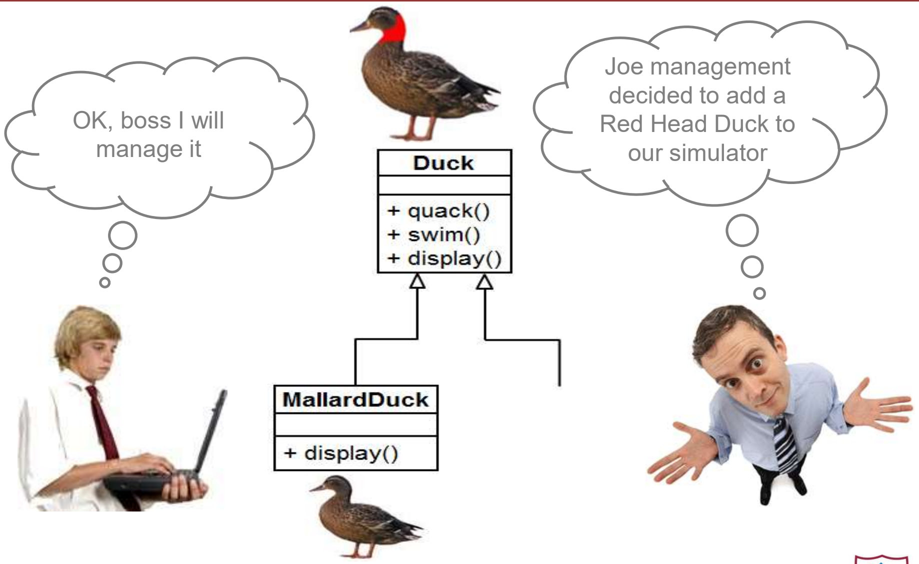
All ducks `quack()` and `swim()`
The base class takes care of the implementation

The `display()` method is abstract since all the duck subtypes look different

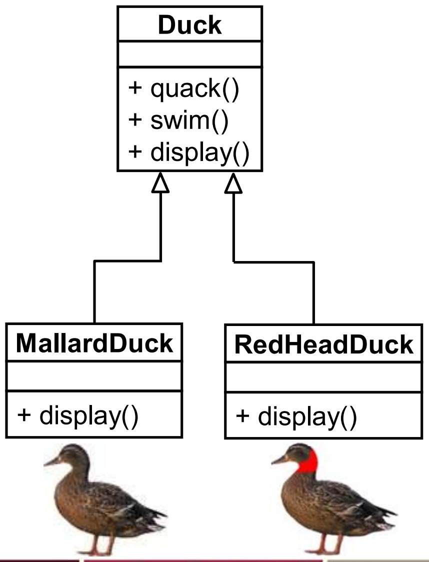
Each duck subtype is responsible for implementing its own `display()` behavior for how it looks on the screen

Lots of other types of ducks inherit from the Duck type

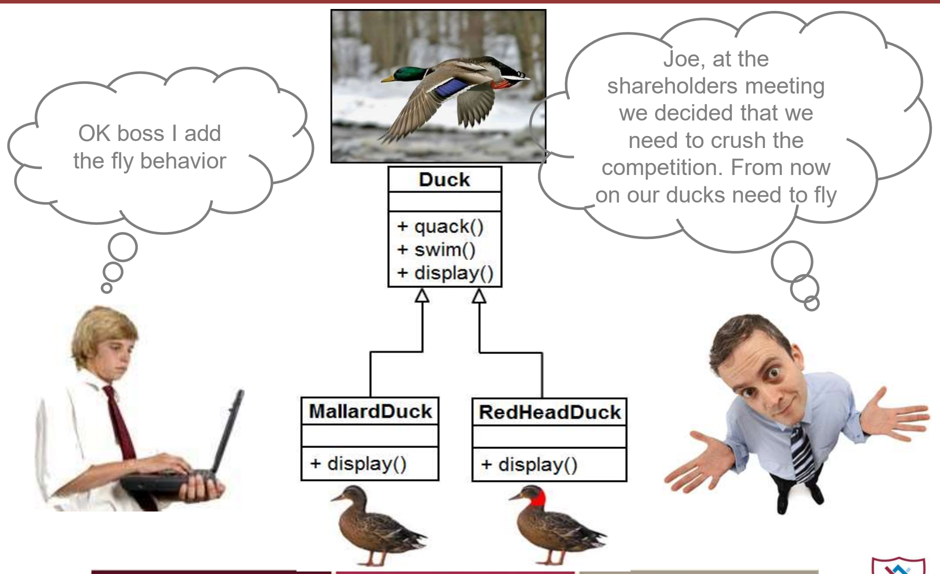
Red Head Duck


OK, boss I will manage it
Joe management decided to add a Red Head Duck to our simulator
Duck
+ quack()
+ swim()
+ display()
MallardDuck
+ display()

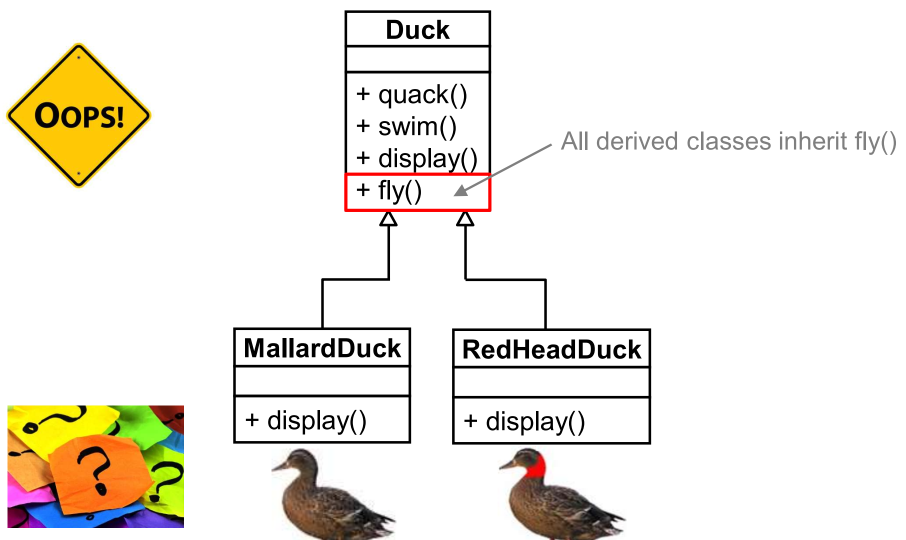
Red Head Duck

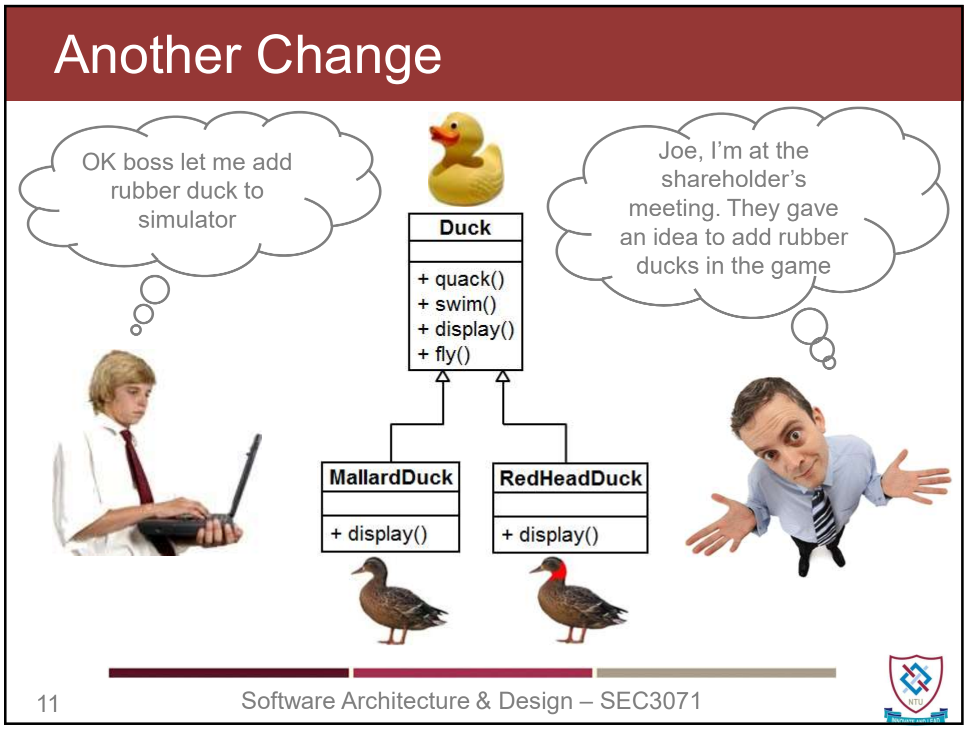


Ducks that Fly


OK boss I add the fly behavior
Joe, at the shareholders meeting we decided that we need to crush the competition. From now on our ducks need to fly
Duck
+ quack()
+ swim()
+ display()
MallardDuck
+ display()
RedHeadDuck
+ display()

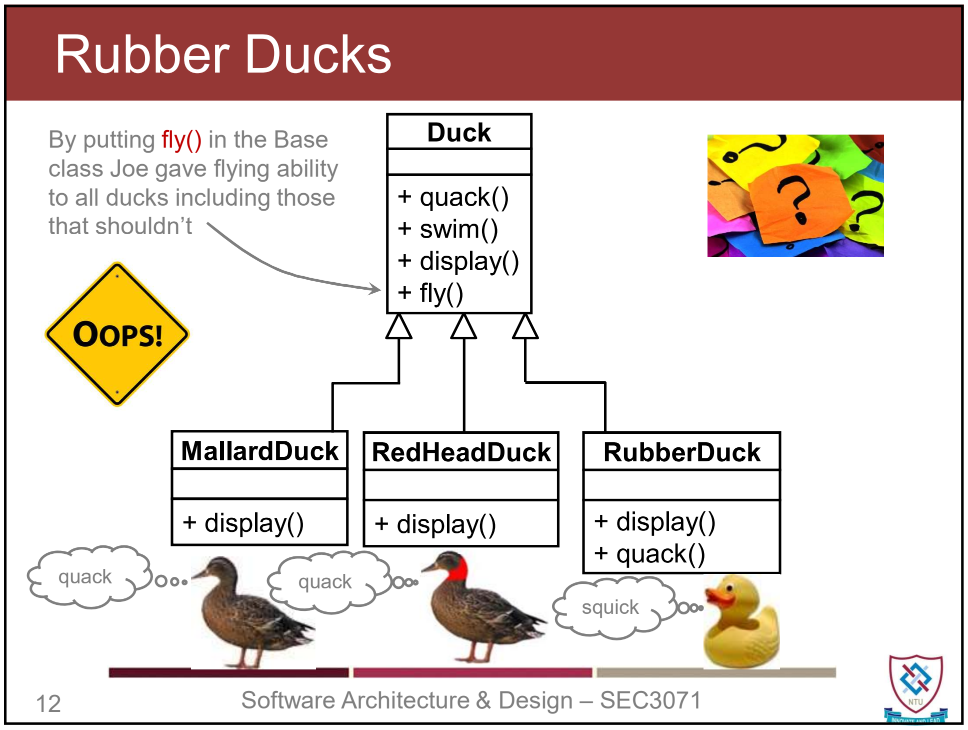
Ducks that Fly





Another Change

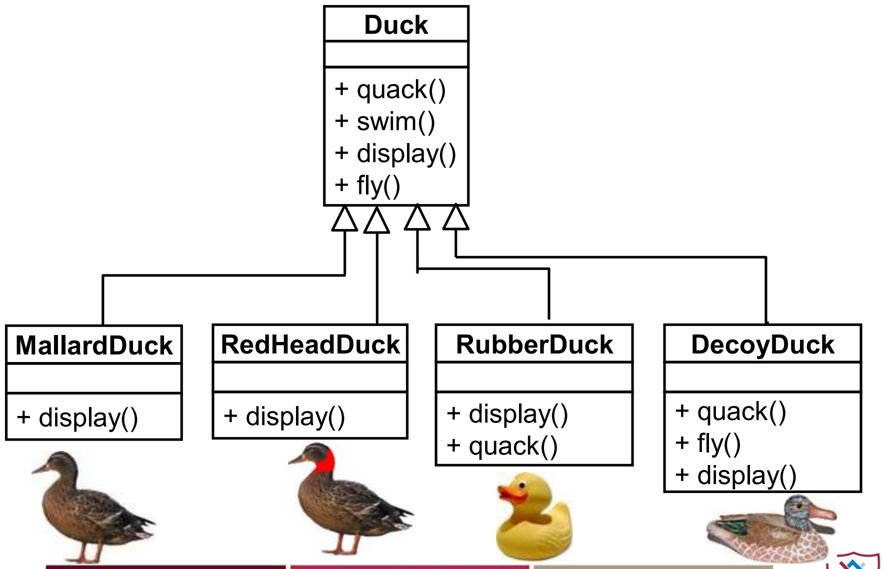
OK boss let me add rubber duck to simulator
Duck
+ quack()
+ swim()
+ display()
+ fly()
MallardDuck
+ display()
RedHeadDuck
+ display()
Joe, I'm at the shareholder's meeting. They gave an idea to add rubber ducks in the game



Rubber Ducks

By putting fly() in the Base class Joe gave flying ability to all ducks including those that shouldn't

Yet Another Duck is Added



Embracing Change

In software projects you can count on one thing that is constant

CHANGE

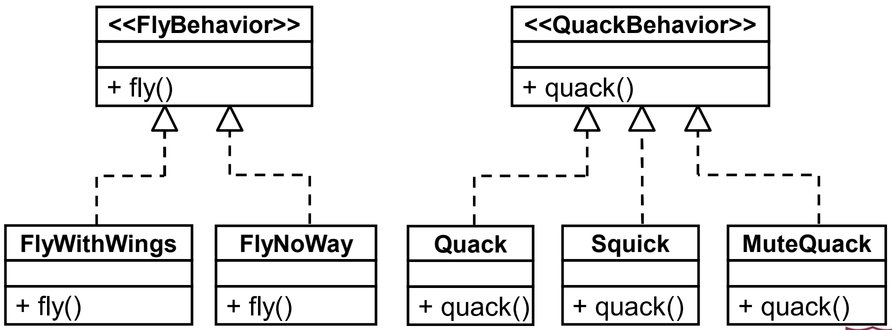
Design Principle

Encapsulate What Varies

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

Embracing Change in Ducks

- `fly()` and `quack()` are the parts that vary
- We create a new set of classes to represent each behavior

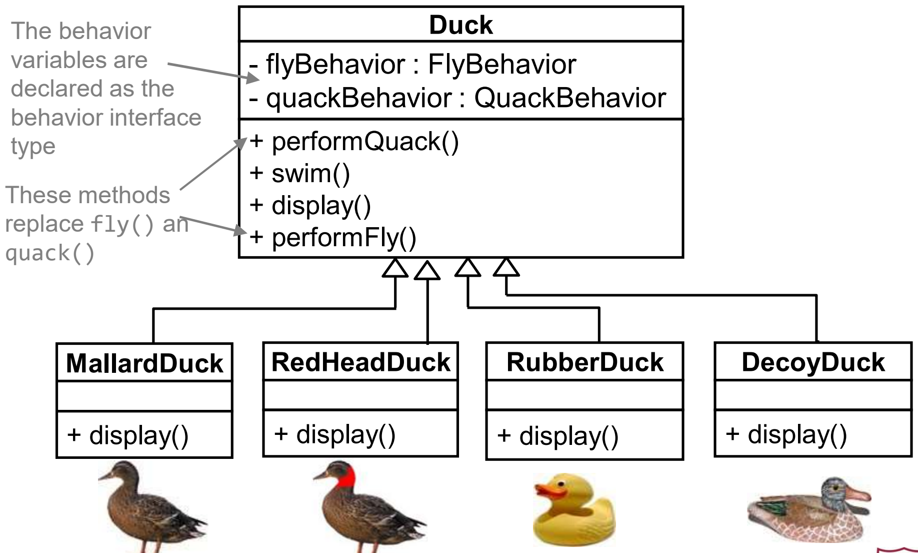


Design Principle

Program to an Interface not to an Implementation

"Abstract the behavior and place that in an interface that the major class will know about."

Yet Another Duck is Added

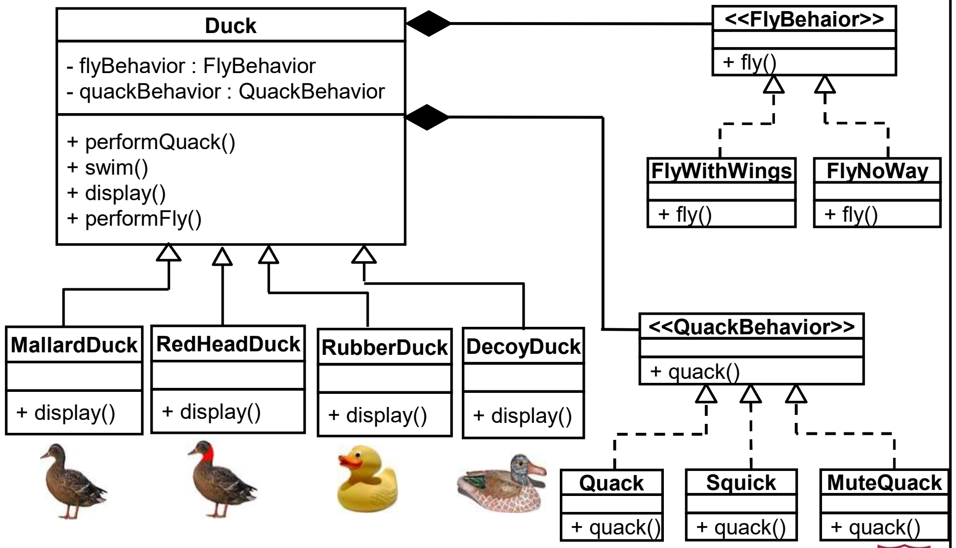

The behavior variables are declared as the behavior interface type
These methods replace fly() an quack()
Duck
- flyBehavior : FlyBehavior
- quackBehavior : QuackBehavior
+ performQuack()
+ swim()
+ display()
+ performFly()
MallardDuck
+ display()
RedHeadDuck
+ display()
RubberDuck
+ display()
DecoyDuck
+ display()

Design Principle

Favor Composition Over Inheritance

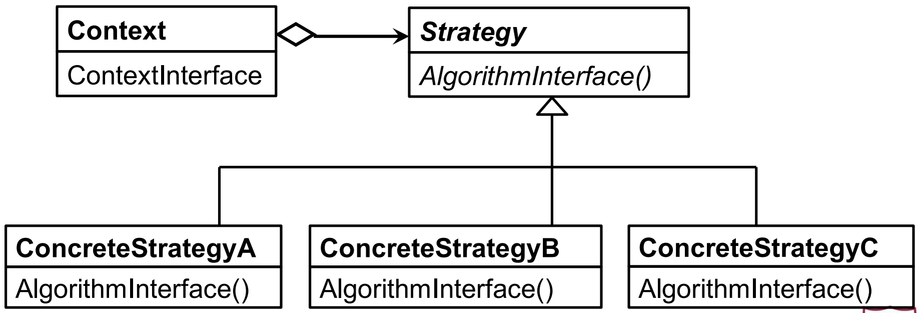
"Object composition allows to modify behavior at runtime and it helps us keep our classes very focused."

The Big Picture



Strategy Design Pattern

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



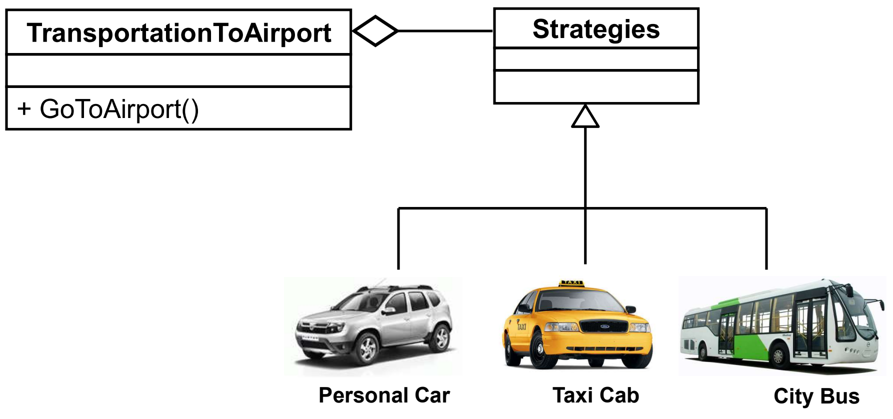
Strategy Pattern - Applicability

- Use the Strategy pattern when:
  - Many related classes <u>differ</u> only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms

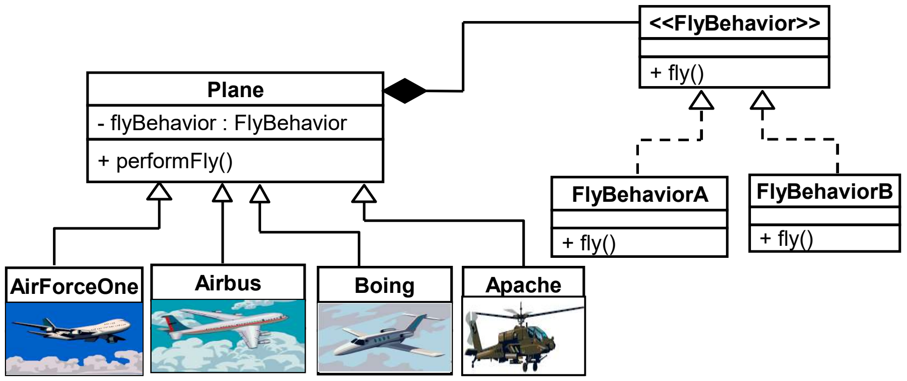
Strategy Pattern - Applicability

- Use the Strategy pattern when:
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

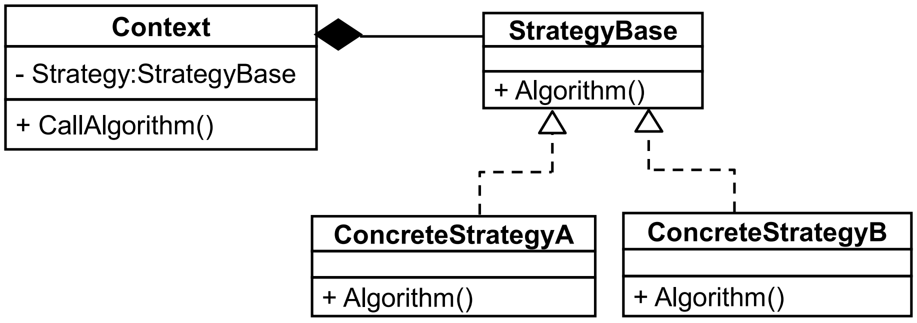
Strategy – Non Software Example



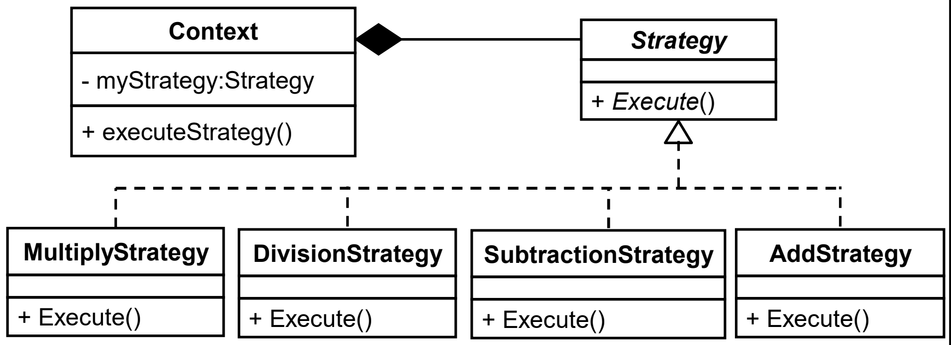
Strategy Design Pattern



Strategy – Implementation



Adding Division and Multiplication



Adding Division and Multiplication

```
class MultiplicationStrategy : Strategy
{
        public override void Execute(int a, int b)
        {
                Console.WriteLine("Multiplication:{0}",a*b);
        }
}
```

```
class DivisionStrategy : Strategy
{
        public override void Execute(int a, int b)
        {
                Console.WriteLine("Division:{0}", a/b);
        }
}
```

Adding Division and Multiplication

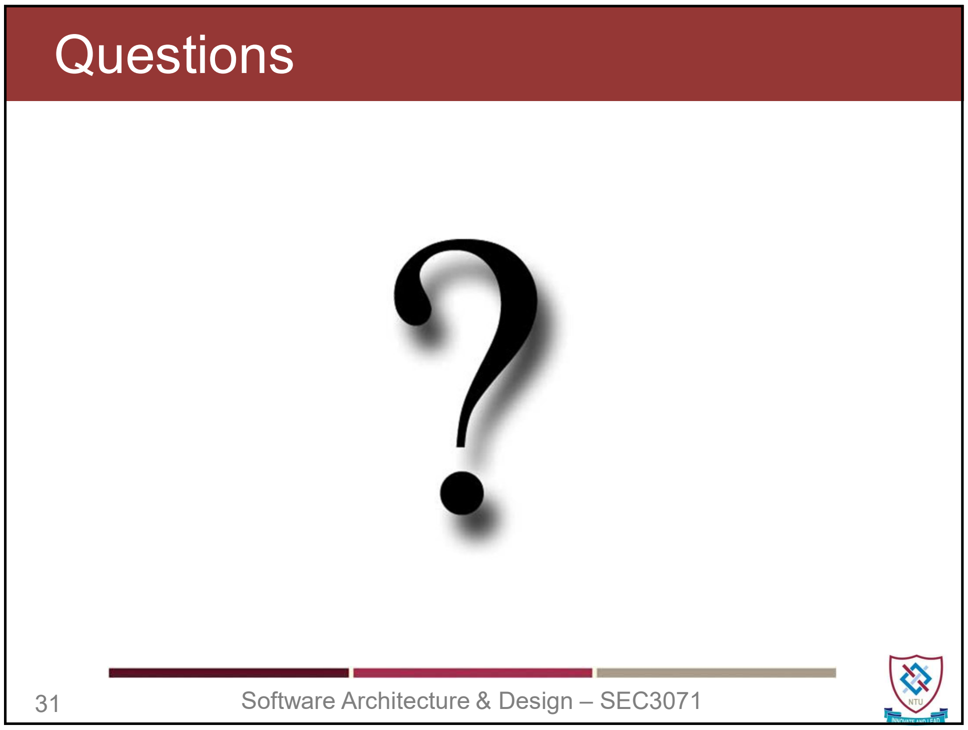
```
static void Main(string[] args)
{
        Context mul = new Context(new MultiplyStrategy());
        mul.executeStrategy(200, 100);

        Context div = new Context(new DivStrategy());
        div.executeStrategy(200, 100);
}
```

```
Multiplication:    20000
Division:          100
```

Recap

- Behavioral Design Patterns
- Design Principle
  – Encapsulate What Varies
  – Program to an Interface not to an Implementation
  – Favor Composition Over Inheritance
- Strategy Design Pattern
  – Applicability
  – Implementation



Questions

?

31 Software Architecture & Design – SEC3071