SOFTWARE DESIGN & ARCHITECTURE

LAB 13

# FACTORY, ABSTRACT & OBSERVER DESIGN PATTERN

# STRATEGY DESIGN PATTERN:

# Strategy is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable inside original context object. The original object, called context, holds a reference to a strategy object and delegates it is executing the behavior.

**Usage examples:** The Strategy pattern is very common in C# code. It’s often used in various frameworks to provide users a way to change the behavior of a class without extending it.

**Identification:** Strategy pattern can be recognized by a method that lets nested object do the actual work, as well as the setter that allows replacing that object with a different one.

## **Conceptual Example**

This example illustrates the structure of the **Strategy** design pattern. It focuses on answering these questions:

* What classes does it consist of?
* What roles do these classes play?
* In what way the elements of the pattern are related?

#### **Program.cs:** Conceptual example

**using** System;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.Strategy.Conceptual

{

// The Context defines the interface of interest to clients.

**class** **Context**

{

// The Context maintains a reference to one of the Strategy objects. The

// Context does not know the concrete class of a strategy. It should

// work with all strategies via the Strategy interface.

**private** IStrategy \_strategy;

**public** Context()

{ }

// Usually, the Context accepts a strategy through the constructor, but

// also provides a setter to change it at runtime.

**public** Context(IStrategy strategy)

{

**this**.\_strategy = strategy;

}

// Usually, the Context allows replacing a Strategy object at runtime.

**public** **void** SetStrategy(IStrategy strategy)

{

**this**.\_strategy = strategy;

}

// The Context delegates some work to the Strategy object instead of

// implementing multiple versions of the algorithm on its own.

**public** **void** DoSomeBusinessLogic()

{

Console.WriteLine("Context: Sorting data using the strategy (not sure how it'll do it)");

**var** **result** = **this**.\_strategy.DoAlgorithm(**new** List<**string**> { "a", "b", "c", "d", "e" });

**string** resultStr = **string**.Empty;

**foreach** (**var** **element** **in** result **as** List<**string**>)

{

resultStr += element + ",";

}

Console.WriteLine(resultStr);

}

}

// The Strategy interface declares operations common to all supported

// versions of some algorithm.

//

// The Context uses this interface to call the algorithm defined by Concrete

// Strategies.

**public** **interface** **IStrategy**

{

**object** DoAlgorithm(**object** data);

}

// Concrete Strategies implement the algorithm while following the base

// Strategy interface. The interface makes them interchangeable in the

// Context.

**class** **ConcreteStrategyA** : IStrategy

{

**public** **object** DoAlgorithm(**object** data)

{

**var** **list** = data **as** List<**string**>;

list.Sort();

**return** list;

}

}

**class** **ConcreteStrategyB** : IStrategy

{

**public** **object** DoAlgorithm(**object** data)

{

**var** **list** = data **as** List<**string**>;

list.Sort();

list.Reverse();

**return** list;

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

// The client code picks a concrete strategy and passes it to the

// context. The client should be aware of the differences between

// strategies in order to make the right choice.

**var** **context** = **new** Context();

Console.WriteLine("Client: Strategy is set to normal sorting.");

context.SetStrategy(**new** ConcreteStrategyA());

context.DoSomeBusinessLogic();

Console.WriteLine();

Console.WriteLine("Client: Strategy is set to reverse sorting.");

context.SetStrategy(**new** ConcreteStrategyB());

context.DoSomeBusinessLogic();

}

}

}

# ADAPTER DESIGN PATTERN:

Adapter is a structural design pattern, which allows incompatible objects to collaborate. The Adapter acts as a wrapper between two objects. It catches calls for one object and transforms them to format and interface recognizable by the second object.

**Usage examples:** The Adapter pattern is common in C# code. It’s very often used in systems based on some legacy code. In such cases, Adapters make legacy code work with modern classes.

**Identification:** Adapter is recognizable by a constructor which takes an instance of a different abstract/interface type. When the adapter receives a call to any of its methods, it translates parameters to the appropriate format and then directs the call to one or several methods of the wrapped object.

## **Conceptual Example**

This example illustrates the structure of the **Adapter** design pattern. It focuses on answering these questions:

* What classes does it consist of?
* What roles do these classes play?
* In what way the elements of the pattern are related?

#### **Program.cs:** Conceptual example

**using** System;

**namespace** **RefactoringGuru**.DesignPatterns.Adapter.Conceptual

{

// The Target defines the domain-specific interface used by the client code.

**public** **interface** **ITarget**

{

**string** GetRequest();

}

// The Adaptee contains some useful behavior, but its interface is

// incompatible with the existing client code. The Adaptee needs some

// adaptation before the client code can use it.

**class** **Adaptee**

{

**public** **string** GetSpecificRequest()

{

**return** "Specific request.";

}

}

// The Adapter makes the Adaptee's interface compatible with the Target's

// interface.

**class** **Adapter** : ITarget

{

**private** **readonly** Adaptee \_adaptee;

**public** Adapter(Adaptee adaptee)

{

**this**.\_adaptee = adaptee;

}

**public** **string** GetRequest()

{

**return** $"This is '{this.\_adaptee.GetSpecificRequest()}'";

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

Adaptee adaptee = **new** Adaptee();

ITarget target = **new** Adapter(adaptee);

Console.WriteLine("Adaptee interface is incompatible with the client.");

Console.WriteLine("But with adapter client can call it's method.");

Console.WriteLine(target.GetRequest());

}

}

}

**Task:** Implement the given codes and identify the working of each Design Pattern respectively.