Data Oriented Programming using Modern C#

Praseed Pai K.T. (praseedp@gmail.com)

Author of ".NET Design Patterns" (Packt Publishing,2018)

Author of "C++ Reactive Programming"(Packt Publishing, 2018)

Author of SLANG Compiler Infrastructure

Four Principles of Data Oriented Programming

- Principle #1: Separate Code from Data
- Principle #2: Represent Data with Generic Data Structures
- Principle #3: Data is Immutable
- Principle #4: Separate Data Schema from Data Representation

Principle #1: Separate Code from Data (First\Program.cs)

```csharp
// Data stored separately in a record (immutable by default)
public record User(int Id, string Name, string Email);

public class UserService
{
    // Code operates on data, but data is defined separately
    public string GetDisplayName(User user)
    {
        return $"{user.Name} ({user.Email})";
    }
}

// Usage
var user = new User(1, "Alice", "alice@example.com");
var service = new UserService();
Console.WriteLine(service.GetDisplayName(user));
```

Principle #2: Represent Data with Generic Data Structures (First\Program.cs)

```
// Using Dictionary instead of a custom class
var product = new Dictionary<string, object>
{
    { "Id", 101 },
    { "Name", "Laptop" },
    { "Price", 1200.50 }
};

// Generic function to print dictionary contents
void PrintData(IDictionary<string, object> data)
{
    foreach (var kvp in data)
    {
        Console.WriteLine($"{kvp.Key}: {kvp.Value}");
    }
}

PrintData(product);
```

Principle #3: Data is Immutable (First\Program.cs)

```
// Schema defined as a record
public record Order(int OrderId, string Customer, decimal Amount);

// Representation: JSON serialization
using System.Text.Json;

var order = new Order(5001, "Bob", 250.75m);

// Immutable: cannot change order fields after creation
string json = JsonSerializer.Serialize(order);
Console.WriteLine(json);

// Deserialize back into schema
var deserializedOrder = JsonSerializer.Deserialize<Order>(json);
Console.WriteLine(deserializedOrder);
```

Principle #4: Separate Data Schema from Data Representation (First\Program.cs)

```
// Schema defined as a record
public record Order(int OrderId, string Customer, decimal Amount);

// Representation: JSON serialization
using System.Text.Json;

var order = new Order(5001, "Bob", 250.75m);

// Immutable: cannot change order fields after creation
string json = JsonSerializer.Serialize(order);
Console.WriteLine(json);

// Deserialize back into schema
var deserializedOrder = JsonSerializer.Deserialize<Order>(json);
Console.WriteLine(deserializedOrder);
```

Three Pillars of OOP and Four types of Polymorphism

- Encapsulation
- Abstraction
- Polymorphism
  - Polymorphism through Inheritance (extension and implementation)
  - Parametric Polymorphism (Generics classes/methods/lambdas)
  - Ad hoc Polymorphism with method overloading
  - Ad hoc Polymorphism with Algebraic Data Types

Algebraic Data Types (Some theory!)

- Product Types (aka Records)
- Sum Types (aka Union types)
- Exponential Types (aka Function Types)
- Ad-hoc Polymorphism using Sum , Product & Exponential Types
- Switch Expressions
- Pattern Matching

Sum Types

•Represent a value that can be **one of several types**.
•Common in functional languages like F#, Rust, and Haskell.
•In C# 10, you emulate this using **abstract records** and **pattern matching**.

```
public abstract record Expr;
public record Constant(double Value) : Expr;
public record Add(Expr Left, Expr Right) : Expr;
```

- Evaluated using **switch expressions** and **type patterns**.

Product Types

- Represent a value that combines multiple fields.
- like a tuple or struct — each instance holds multiple values.

```
public record Person(string Name, int Age);
```

Exponential Types (aka Function Types)

Represent a mapping from one type to another: A → B.

In category theory, this is analogous to exponentiation: B^A.

In C#, this is expressed via delegates or lambdas:

```
Func<int, string> describe = age => $"Age: {age}";
```

How these are combined together?

- C# 14 brings these concepts closer to functional programming:
- Sum types via abstract records + pattern matching
- Product types via concise record syntax
- Exponential types via lambdas, delegates, and Func<>
- Switch expressions unify control flow with pattern matching

Goodbye Composite/Visitor

- Algebraic data types help you to model Types
- Processing is done through Recursive function, Functional switch
- No complicated manouever like Double Dispatch etc
- Enables Adhoc polymorphism

Composite/Visitor version of Evaluator

```
class Constant : Expr {
    public double Value;
    public override double Evaluate() => Value;
}
class Add : Expr {
    public Expr Left, Right;
    public override double Evaluate() => Left.Evaluate() + Right.Evaluate();
}
interface IVisitor {
    double Visit(Constant c);
    double Visit(Add a);
}
abstract class Expr {
    public abstract double Accept(IVisitor visitor);
}
```

Algebraic data type version of Evaluator

```
public abstract record Expr;
public record Constant(double Value) : Expr;
public record Add(Expr Left, Expr Right) : Expr;

double Evaluate(Expr expr) => expr switch
{
    Constant c => c.Value,
    Add a => Evaluate(a.Left) + Evaluate(a.Right),
    _ => throw new InvalidOperationException()
};
```

Examples of ADT and DOP

- Expression Evaluator
- Shape Hierarchy
- Org Hierarchy
- Tree Data Structure

Example One

- Expression Evaluator using Composite
  - ExprComposite\Program.cs
- Expression Evaluator using Visitor
  - ExpressionVisitor\Program.cs
- Expression Evaluator using ADT
  - Expr\Program.cs

Example Two

- Shapes Hierarchy using Composite/Visitor
  - ShapeVisitor*.* (Code)
- Shapes Hierarchy using ADT
  - ShapeADT*.* (Code)

Example Three

- Org Hierarchy using Composite
  - OrgComposite\Program.cs
- Org Hierarchy using Visitor
  - OrgVisitor\Program.cs
- Org Hierarchy using ADT
  - OrgADT\Program.cs

Example Four

- Tree using ADT
  - Tree\Program.cs

Questions?

- If any!