# Code Review

The following is a review of a code snippet of a program that is meant to create build and save a customer invoice

```
class Program
{
    public static void Main(string[] args)
    {
        // create a new invoice
        var invoice = new Invoice
        {
            InvoiceNo = 1,
            Customer = "John Doe",
            IssuedDate = new DateOnly(2023, 4, 1),
            Description = "Website Design",
            Amount = 1000,
            Tax = Amount * CURRENT_TAX_RATE
        };

        invoice.Save();

    }
}
```

# C# General Principles

Standard C# coding conventions - NO

KISS - NO

DRY - NO

YAGNI - NO

Single responsibility principle - YES

Open/closed principle - NO

Liskov substitution principle - NO

Interface segregation principle - NO

Dependency inversion principle - NO

The code sample above seems to follow the general principles except for the single responsibility principle. The code seems to declare, create and save an invoice this goes against the single resonsibility principle, this is apart of the SOLID programming principles within OOP
(Object Orientated Programming).
the principle states that any single object within OOP should be made for one specific function so these should be split in to differnet Classes or Methods.

---

# C# Best Practice Rules

39.Does the code violate any of the following best practice rules?

Select all that apply

Clear and consistent layout/formatting - NO

Use appropriate names for variables, functions, classes, etc. - YES

Use recognised commenting conventions - NO

Distinguish between objects and data structures - NO

---

Although the code seems to be well structured and has good use of indentation the code still violates naming convewntions in C#. The class name Program should be more descriptive of what the class does.

---

# C# Code Smells

Does the code exhibit any of the following code smells?

Select all that apply

Alternative Classes with Different Interfaces - NO

Duplicated Code - NO

Feature Envy - NO

Global Data - NO

Insider Trading - NO

Middle man - NO

Mutable Data - NO

Primitive Obsession - YES

Repeated Switches - NO

Shotgun Surgery - NO

Speculative Generality - NO

Temporary Field - NO

---

The Sample above seems to have the Primitive Obsession code smell becuase in the tax calculation the the `Amount` field is diectly used instead of encapsulating this into a more specialised class or method.

---

# Resolution

To Resolve these issues the code sample was reworked and cleaned up to make it more maintainable and readable. this is how it looks.

```
class Invoice
{
    public int InvoiceNo { get; private set; }
    public string Customer { get; private set; }
    public DateOnly IssuedDate { get; private set; }
    public string Description { get; private set; }
    public decimal Amount { get; private set; }
    public decimal Tax { get; private set; }

    // Private constructor
    private Invoice() { }


    // Method to create a new invoice
        public static Invoice CreateNewInvoice(int invoiceNo, string customer,
DateOnly issuedDate, string description, decimal amount)
    {
        var invoice = new Invoice
        {
            InvoiceNo = invoiceNo,
            Customer = customer,
            IssuedDate = issuedDate,
            Description = description,
            Amount = amount
        };

        return invoice;
    }

    // Calculate tax for the invoice
```

```
    public void CalculateTax(decimal taxRate)
    {
        Tax = Amount * taxRate;
    }

    // save invoice
        public void Save()
    {
        Console.WriteLine("Invoice saved.");
        // Actual saving logic can be implemented here.
    }
```

```
class TaxInvoice
{
    public static void Main(string[] args)
    {
        // Create a new invoice
        var invoice = Invoice.CreateNewInvoice(
            1,
            "John Doe",
            new DateOnly(2023, 4, 1),
            "Website Design",
            1000);

        // Calculate and set the tax for the invoice
        invoice.CalculateTax(CURRENT_TAX_RATE);

        // Save the invoice
        invoice.Save();
    }
}
```

---

It can be seen that the refactored code has sorted the Signle Responsibility Principle
by splitting up the invoice creation and tax calculation by giving them their own methods and classes.
the tax calculations have now been encapsulated so that it does not use the `amount` field directly.
the code looks easier to read now and will be easier to maintain.