VB.NET Programming: Professional

(Integrative-Generative AI Edition)

ET

Contents:

Design Patterns-1
Architecture & Best Practices-93
Unit Testing & Test-Trived Development-
Advanced Database & ORM-280
Deployment & Maintenance
Bibliiglaphy-3400

Author: Student Price Book Center

คำนำ

การพัฒนาแอปพลิเคชันในระดับมืออาชีพต้องอาศัยมากกว่าการเขียนโค้ดให้ถูกต้องเพียงอย่างเดียว นักพัฒนาจำเป็นต้องเข้าใจแนวทางการออกแบบซอฟต์แวร์, สถาปัตยกรรม, เทคนิคการทดสอบ, การจัดการฐานข้อมูลขั้นสูง, และกระบวนการส่งมอบซอฟต์แวร์อย่างเป็นระบบ หนังสือเล่มนี้ **"VB.NET Programming: Professional"** ถูกออกแบบมาเพื่อตอบโจทย์เหล่านี้ โดยเน้นการประยุกต์ใช้ VB.NET ในงานจริงที่ซับซ้อนและต้องการ maintainability สูง

เนื้อหาของหนังสือครอบคลุมหัวข้อสำคัญ 5 บทหลัก ได้แก่ **Design Patterns, Architecture & Best Practices, Unit Testing & Test-Driven Development, Advanced Database & ORM, และ Deployment & Maintenance** โดยแต่ละบทไม่เพียงแต่ให้แนวคิดเชิงทฤษฎี แต่ยังมี **ตัวอย่างบูรณาการและโปรเจกต์จริง** ให้ผู้อ่านสามารถทดลองใช้งานและเข้าใจแนวทางการออกแบบและพัฒนาซอฟต์แวร์ได้อย่างครบวงจร

บทที่ 18 **Design Patterns** มุ่งเน้นแนวทางการแก้ปัญหาการออกแบบซ้ำ ๆ อย่างเป็นระบบ เช่น **Singleton, Factory Method, Repository, Dependency Injection, และ Observer Pattern** พร้อมตัวอย่างบูรณาการตั้งแต่ระบบ Mini E-Commerce จนถึง Full System แบบ Advanced Integration ทำให้ผู้อ่านสามารถสร้างโค้ดที่ยืดหยุ่น, maintainable, และลดความซับซ้อนในการพัฒนาระบบขนาดใหญ่

บทที่ 19 **Architecture & Best Practices** แนะนำแนวทางการออกแบบซอฟต์แวร์ที่มืออาชีพควรทราบ เช่น **Layered Architecture (Presentation, Business, Data Access), SOLID Principles, Clean Code Guidelines, และ Error Logging & Monitoring** โดยมีตัวอย่างบูรณาการ Mini E-Commerce Project ให้เห็นภาพการประยุกต์ใช้จริง ทั้งหมดช่วยให้นักพัฒนาสามารถสร้างโค้ดที่เสถียร, testable, และง่ายต่อการขยายฟีเจอร์ใหม่

บทที่ 20 **Unit Testing & Test-Driven Development (TDD)** ครอบคลุมการเขียน Unit Test ด้วย **MSTest และ NUnit**, การใช้ **Moq** สำหรับ mocking dependencies, การทดสอบ **Async Methods**, และ **Integration Testing** แนวทาง TDD ทำให้นักพัฒนาสามารถออกแบบฟีเจอร์โดยเริ่มจาก test ก่อน ทำให้โค้ดมั่นคง, maintainable, และลดข้อผิดพลาดเมื่อมีการ refactor

บทที่ 21 **Advanced Database & ORM** เจาะลึกเทคนิคการจัดการข้อมูล เช่น การใช้ **Stored Procedures, Transactions, และ Entity Framework (Code First/Database First)** รวมถึงการใช้ **EF Migration** เพื่อปรับปรุง schema อย่างปลอดภัย ตัวอย่างบูรณาการช่วยให้เห็นภาพการทำงานร่วมกันระหว่างโค้ด VB.NET กับฐานข้อมูลอย่างครบวงจร

บทที่ 22 **Deployment & Maintenance** มุ่งเน้นการส่งมอบซอฟต์แวร์อย่างมืออาชีพ ทั้งการสร้าง **Installer (MSI, ClickOnce, Custom)** การ **Deploy บน IIS สำหรับ ASP.NET VB.NET**, การสร้าง **All-in-One Project พร้อม CI/CD Pipeline**, และแนวทางการ maintain ระบบในระยะยาว

ตัวอย่างบูรณาการช่วยให้นักพัฒนาสามารถ deploy และอัปเดตระบบได้อย่างเป็นระบบและมีประสิทธิภาพ

ท้ายที่สุด หนังสือเล่มนี้ช่วยให้นักพัฒนาสามารถยกระดับทักษะ VB.NET ไปสู่ระดับมืออาชีพ สร้างระบบที่ **มีสถาปัตยกรรมดี, โค้ดสะอาด, ทดสอบได้ง่าย, และพร้อมใช้งานจริง** ทั้งสำหรับโปรแกรมเดสก์ท็อป, เว็บแอป, และระบบที่ต้องรองรับการพัฒนาต่อเนื่อง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 18 Design Patterns 1

- Design Patterns
- Design Patterns (เชิงลึกใน VB.NET)
- Singleton Pattern (VB.NET)
- Factory Method Pattern (VB.NET)
- Repository Pattern
- Dependency Injection (DI)
- ตัวอย่างบูรณาการ
- Observer Pattern
- ตัวอย่างบูรณาการ Mini E-Commerce System
- Mini E-Commerce Full System (Advanced Integration)

บทที่ 19 Architecture & Best Practices 93

- Architecture & Best Practices
- Architecture & Best Practices (เชิงลึก)
- รายละเอียดเชิงลึกของ Layered Architecture
- SOLID Principles (เชิงลึก)
- Clean Code Guidelines (VB.NET)
- Error Logging & Monitoring (VB.NET)
- ตัวอย่างบูรณาการ
- โปรเจกต์บูรณาการ Mini E-Commerce

บทที่ 20 Unit Testing & Test-Driven Development 194

- Unit Testing & Test-Driven Development
- Unit Testing & Test-Driven Development (TDD)
- การใช้ MSTest และ NUnit
- Mocking ด้วย Moq
- Testing Async Methods

● Integration Testing
● ตัวอย่างบูรณาการ
บทที่ 21 Advanced Database & ORM 280
● Advanced Database & ORM
● Advanced Database & ORM (เชิงลึก)
● Stored Procedures (SP)
● Transactions (เชิงลึก)
● Entity Framework (EF) – Advanced
● EF Migration (Code First)
● ตัวอย่างบูรณาการ
บทที่ 22 Deployment & Maintenance 346
● Deployment & Maintenance
● Deployment & Maintenance – รายละเอียดเชิงลึก
● รายละเอียดเชิงลึกและตัวอย่างจริงสำหรับการสร้าง Installer
● All-in-One VB.NET Project ที่รวม ทั้ง MSI + ClickOnce + Custom
● รายละเอียดเชิงลึกเกี่ยวกับการ Deploy บน IIS สำหรับ ASP.NET VB.NET
● All-in-One ASP.NET VB.NET Project สำหรับ IIS Deployment
● All-in-One VB.NET / ASP.NET Project พร้อม CI/CD Pipeline
● ตัวอย่างบูรณาการ

บรรณานุกรม 400

บทที่ 18

Design Patterns

(Design Patterns)

เนื้อหา

- Design Patterns
- Design Patterns (เชิงลึกใน VB.NET)
- Singleton Pattern (VB.NET)
- Factory Method Pattern (VB.NET)
- Repository Pattern
- Dependency Injection (DI)
- ตัวอย่างบูรณาการ
- Observer Pattern
- ตัวอย่างบูรณาการ Mini E-Commerce System
- Mini E-Commerce Full System (Advanced Integration)

บทนำ: Design Patterns

การออกแบบซอฟต์แวร์ที่มีคุณภาพไม่เพียงแต่ขึ้นอยู่กับความสามารถในการเขียนโค้ดเท่านั้น แต่ยังขึ้นอยู่กับการใช้ **แนวทางการออกแบบซอฟต์แวร์ที่เป็นมาตรฐาน** เพื่อให้โค้ดมีความยืดหยุ่น ดูแลรักษาได้ง่าย และสามารถขยายฟีเจอร์ใหม่ ๆ ได้โดยไม่ทำลายโครงสร้างเดิม **Design Patterns** จึงเป็นเครื่องมือสำคัญที่ช่วยให้นักพัฒนาสามารถแก้ไขปัญหาการออกแบบซ้ำ ๆ ด้วยวิธีที่พิสูจน์แล้ว

บทนี้เน้น **Design Patterns** ที่นิยมใช้ใน VB.NET และแอปพลิเคชันจริง ได้แก่ **Singleton, Factory Method, Repository, Dependency Injection (DI), และ Observer Pattern** การเข้าใจและประยุกต์ใช้ pattern เหล่านี้จะช่วยให้โค้ดมี modularity, reusability, และ maintainability สูงขึ้น

Singleton Pattern เป็น pattern ที่ใช้ควบคุมการสร้างวัตถุให้มีเพียง instance เดียวในระบบ เหมาะสำหรับจัดการ resource ที่ใช้ร่วมกัน เช่น การเชื่อมต่อฐานข้อมูล หรือ configuration ของแอปพลิเคชัน การใช้ Singleton อย่างถูกต้องช่วยลดข้อผิดพลาดและเพิ่มประสิทธิภาพของระบบ

Factory Method Pattern ช่วยแยกกระบวนการสร้างวัตถุออกจากการใช้งาน ทำให้โปรแกรมสามารถเปลี่ยนชนิดของ object ที่สร้างได้โดยไม่กระทบส่วนอื่นของระบบ Pattern นี้สนับสนุนหลักการ Open/Closed Principle ของ SOLID และทำให้โค้ดยืดหยุ่นต่อการเปลี่ยนแปลง

Repository Pattern เป็นแนวทางในการจัดการการเข้าถึงข้อมูล ทำให้การเชื่อมต่อฐานข้อมูลและการ query ข้อมูลถูกแยกออกจาก logic ของโปรแกรมหลัก ทำให้โค้ดมีความสะอาด ทดสอบง่าย และ maintainable การใช้ Repository Pattern ยังช่วยให้สามารถเปลี่ยนแหล่งข้อมูลได้โดยไม่กระทบส่วนอื่นของโปรแกรม

Dependency Injection (DI) เป็น pattern ที่ช่วยลดการเชื่อมโยงระหว่างคลาส ทำให้โค้ดมีความยืดหยุ่นและทดสอบง่ายขึ้น นักพัฒนาสามารถ inject dependency ผ่าน constructor, property หรือ method แทนการสร้าง instance ภายในคลาสโดยตรง ซึ่งช่วยสนับสนุนหลักการ SOLID และ test-driven development

Observer Pattern เป็น pattern สำหรับจัดการกับ event-driven programming ทำให้วัตถุหนึ่งสามารถแจ้งเตือนผู้สนใจ (observers) เมื่อเกิดเหตุการณ์บางอย่าง การใช้ Observer Pattern ช่วยให้โค้ดยืดหยุ่น รองรับการเปลี่ยนแปลงในอนาคต และง่ายต่อการขยายฟีเจอร์โดยไม่กระทบส่วนอื่นของระบบ

Design Patterns

- Singleton
- Factory Method
- Repository
- Dependency Injection (DI)
- Observer Pattern

บทที่ 18: Design Patterns

1. Singleton Pattern

แนวคิด

- ใช้เมื่อเราต้องการให้ **คลาสมีเพียงอินสแตนซ์เดียวตลอดการทำงานของโปรแกรม**
- ตัวอย่างการใช้งาน: การจัดการ **Configuration, Logging, Database Connection Pool**
- ข้อดี: ลดการใช้ทรัพยากร, ควบคุมการเข้าถึงได้ง่าย
- ข้อเสีย: ถ้าใช้พร่ำเพรื่ออาจทำให้เกิด **Global State** ที่ยากต่อการทดสอบ

โค้ดตัวอย่าง

```
Public Class Logger
    Private Shared _instance As Logger
    Private Shared ReadOnly _lock As New Object()

    Private Sub New()
```

```
        ' ป้องกันการ new จากภายนอก
    End Sub

    Public Shared ReadOnly Property Instance As Logger
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New Logger()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Sub Log(message As String)
        Console.WriteLine($"[{DateTime.Now}] {message}")
    End Sub
End Class
```

การใช้งาน

```
Logger.Instance.Log("เริ่มการทำงานของระบบ")
Logger.Instance.Log("บันทึกข้อมูลสำเร็จ")
```

---

2. Factory Method Pattern

แนวคิด

- ใช้เมื่อเราต้องการให้ **คลาสแม่กำหนดโครงสร้างการสร้างออบเจ็กต์** แต่ปล่อยให้ **คลาสลูกเป็นผู้กำหนดว่าออบเจ็กต์แบบไหนจะถูกสร้าง**
- เหมาะกับระบบที่มี **การเปลี่ยนแปลงประเภทของออบเจ็กต์บ่อย ๆ** เช่น การสร้าง **Shape, Payment Method, Report Exporter**

โค้ดตัวอย่าง

```
' Product
Public MustInherit Class Shape
    Public MustOverride Sub Draw()
End Class
```

```
' Concrete Product
Public Class Circle
    Inherits Shape
    Public Overrides Sub Draw()
        Console.WriteLine("วาดวงกลม")
    End Sub
End Class

Public Class Square
    Inherits Shape
    Public Overrides Sub Draw()
        Console.WriteLine("วาดสี่เหลี่ยม")
    End Sub
End Class

' Factory
Public MustInherit Class ShapeFactory
    Public MustOverride Function CreateShape() As Shape
End Class

' Concrete Factory
Public Class CircleFactory
    Inherits ShapeFactory
    Public Overrides Function CreateShape() As Shape
        Return New Circle()
    End Function
End Class

Public Class SquareFactory
    Inherits ShapeFactory
    Public Overrides Function CreateShape() As Shape
        Return New Square()
    End Function
End Class
```

การใช้งาน

```
Dim factory As ShapeFactory = New CircleFactory()
Dim shape As Shape = factory.CreateShape()
shape.Draw()   ' แสดงผล: วาดวงกลม
```

---

3. Repository Pattern

แนวคิด

- ใช้ในการแยก **Data Access Layer (DAL)** ออกจาก **Business Logic Layer (BLL)**
- Repository ทำหน้าที่เป็น **ตัวกลางระหว่างแอปกับ Database**
- ข้อดี: ทดสอบง่าย, เปลี่ยนฐานข้อมูลโดยไม่กระทบ Business Logic

โค้ดตัวอย่าง

```
' Entity
Public Class Product
    Public Property Id As Integer
    Public Property Name As String
End Class

' Repository Interface
Public Interface IProductRepository
    Function GetAll() As IEnumerable(Of Product)
    Sub Add(product As Product)
End Interface

' Repository Implementation (In-Memory)
Public Class ProductRepository
    Implements IProductRepository

    Private ReadOnly _products As New List(Of Product)()

    Public Function GetAll() As IEnumerable(Of Product) Implements IProductRepository.GetAll
        Return _products
    End Function

    Public Sub Add(product As Product) Implements IProductRepository.Add
```

```
        _products.Add(product)
    End Sub
End Class
```

การใช้งาน

```
Dim repo As IProductRepository = New ProductRepository()
repo.Add(New Product With {.Id = 1, .Name = "Notebook"})
repo.Add(New Product With {.Id = 2, .Name = "Mouse"})

For Each p In repo.GetAll()
    Console.WriteLine($"{p.Id} - {p.Name}")
Next
```

---

4. Dependency Injection (DI)

แนวคิด

- คือการ **ส่งพารามิเตอร์ (Dependency) จากภายนอกเข้ามา** แทนที่จะสร้างเองในคลาส
- ทำให้โค้ด **ยืดหยุ่น, ทดสอบง่าย, เปลี่ยน Implementation ได้**
- ใช้คู่กับ **IoC Container** (เช่น Autofac, Unity, Microsoft.Extensions.DependencyInjection)

โค้ดตัวอย่าง (Manual DI)

```
' Service Interface
Public Interface IMessageService
    Sub Send(message As String)
End Interface

' Concrete Service
Public Class EmailService
    Implements IMessageService
    Public Sub Send(message As String) Implements IMessageService.Send
        Console.WriteLine($"ส่งอีเมล: {message}")
    End Sub
End Class

' Consumer Class
Public Class Notification
    Private ReadOnly _service As IMessageService
```

```
    Public Sub New(service As IMessageService)
        _service = service
    End Sub

    Public Sub Notify(userMessage As String)
        _service.Send(userMessage)
    End Sub
End Class
```

การใช้งาน

```
Dim service As IMessageService = New EmailService()
Dim notify As New Notification(service)
notify.Notify("สมัครสมาชิกสำเร็จ!")
```

---

5. Observer Pattern

แนวคิด

- ใช้ในสถานการณ์ที่มี **ความสัมพันธ์แบบ One-to-Many**
- เมื่อ **Subject เปลี่ยนแปลงสถานะ ⟶ แจ้งเตือน Observer ทุกตัว**
- ตัวอย่าง: ระบบ Event, Stock Price, UI Binding

โค้ดตัวอย่าง

```
' Observer Interface
Public Interface IObserver
    Sub Update(price As Decimal)
End Interface

' Subject
Public Class Stock
    Private ReadOnly observers As New List(Of IObserver)()
    Private _price As Decimal

    Public Sub Attach(observer As IObserver)
        observers.Add(observer)
    End Sub
```

```
    Public Sub Detach(observer As IObserver)
        observers.Remove(observer)
    End Sub

    Public Sub SetPrice(newPrice As Decimal)
        _price = newPrice
        NotifyObservers()
    End Sub

    Private Sub NotifyObservers()
        For Each obs In observers
            obs.Update(_price)
        Next
    End Sub
End Class

' Concrete Observer
Public Class Investor
    Implements IObserver

    Private _name As String
    Public Sub New(name As String)
        _name = name
    End Sub

    Public Sub Update(price As Decimal) Implements IObserver.Update
        Console.WriteLine($"{_name} ได้รับการแจ้งเตือน ราคาหุ้นใหม่: {price}")
    End Sub
End Class
```

การใช้งาน

```
Dim stock As New Stock()
Dim inv1 As New Investor("สมชาย")
Dim inv2 As New Investor("สมหญิง")
```

```
stock.Attach(inv1)
stock.Attach(inv2)

stock.SetPrice(120.5D)
' สมชาย ได้รับการแจ้งเตือน ราคาหุ้นใหม่: 120.5
' สมหญิง ได้รับการแจ้งเตือน ราคาหุ้นใหม่: 120.5
```

สรุป

- **Singleton** ⟶ ใช้เมื่ออยากให้มีเพียงอินสแตนซ์เดียว เช่น Logger
- **Factory Method** ⟶ สร้างออบเจ็กต์ผ่าน Factory ปรับเปลี่ยนประเภทได้ง่าย
- **Repository** ⟶ แยกการเข้าถึงข้อมูลออกจาก Business Logic
- **Dependency Injection (DI)** ⟶ ส่ง dependency จากภายนอก เพื่อความยืดหยุ่นและ testability
- **Observer** ⟶ ใช้กับระบบแจ้งเตือนเมื่อมีการเปลี่ยนแปลง

Design Patterns (เชิงลึกใน VB.NET)

1. Singleton Pattern

Concept

- คลาสมี **อินสแตนซ์เพียงหนึ่งเดียว** และเข้าถึงได้ทั่วทั้งระบบ (Global Access Point)
- ใช้ควบคุม resource ที่มีต้นทุนสูง เช่น Database Connection, Logger, Configuration Manager

Structure

- Private Constructor ⟶ กันไม่ให้ New ได้
- Shared Property/Method ⟶ เป็นจุดเข้าถึง (Instance)
- Thread-Safe ⟶ ป้องกันปัญหา Multi-thread สร้าง instance ซ้ำ

Pros

- ประหยัดหน่วยความจำ (แชร์เพียง object เดียว)
- ควบคุมจุดเข้าถึงเพียงที่เดียว

Cons

- ถ้าใช้เกินความจำเป็นจะกลายเป็น **Global State** ⟶ ยากต่อการทดสอบ (Unit Test)
- ทำให้ Coupling สูง

Real-World Use Case

- Logger

- App Configuration (อ่านจากไฟล์ app.config)
- Connection Pool

---

2. Factory Method Pattern

Concept

- Encapsulate กระบวนการสร้าง object → ลดการพึ่งพา New โดยตรง
- ให้ **Subclass** ตัดสินใจว่า object แบบไหนจะถูกสร้าง
- ช่วยให้โค้ด **Open for Extension, Closed for Modification** (หลักการ OCP ของ SOLID)

Structure

- **Product** (Abstract Class / Interface)
- **ConcreteProduct** (Implementation จริง)
- **Creator (Factory)** → เมธอดสำหรับสร้าง Product
- **ConcreteCreator** → กำหนดการสร้าง Product แต่ละชนิด

Pros

- เพิ่มประเภทของ object ได้โดยไม่ต้องแก้โค้ดหลัก
- ใช้กับระบบที่ต้องรองรับ **หลายรูปแบบ** เช่น Payment (CreditCard/PayPal/QR)

Cons

- อาจซับซ้อนขึ้นถ้า Product มีจำนวนมาก
- ต้องใช้ร่วมกับ **Abstract Factory** หรือ **DI Container** ในระบบใหญ่

Real-World Use Case

- Exporter (PDF, Excel, Word)
- Payment Gateway
- UI Control Factory

---

3. Repository Pattern

Concept

- เป็น Layer ที่ **คั่นกลางระหว่าง Business Logic กับ Data Access (Database, API, File)**
- ลดการซ้ำซ้อนของโค้ด Query
- เปลี่ยน Database ได้โดยไม่กระทบ Layer อื่น

Structure

- **Entity** → ข้อมูลจริง เช่น Product, Customer
- **Repository Interface** → กำหนดสัญญา เช่น GetAll(), Add()
- **Repository Implementation** → เชื่อมต่อ DB (SQL, NoSQL, File, API)

- **Service Layer** ⟶ เรียกใช้ Repository

Pros

- Test ได้ง่าย (ใช้ FakeRepository / MockRepository)
- Decouple ระหว่าง Business Logic และ Database
- สามารถใช้ร่วมกับ **Unit of Work**

Cons

- โค้ดอาจซ้ำซ้อนถ้ามีหลาย Repository
- ถ้า Query ซับซ้อน ⟶ Repository อาจบวมใหญ่

Real-World Use Case

- Product Repository (เชื่อมต่อ SQL/NoSQL)
- User Repository (Login, Register, Profile)
- Logging Repository (เขียน Log ลงไฟล์/DB)

---

4. Dependency Injection (DI)

Concept

- แทนที่ **Class จะสร้าง Dependency ด้วยตนเอง (new)** ⟶ ให้ **Dependency ถูกส่งเข้ามาจากภายนอก**
- ใช้กับหลักการ **Inversion of Control (IoC)**
- ลด Coupling, เพิ่ม Testability

Structure

- **Service Interface** ⟶ กำหนดสัญญา
- **Concrete Implementation** ⟶ การทำงานจริง
- **Consumer Class** ⟶ รับ Service ผ่าน Constructor/Property
- **DI Container** (เช่น Autofac, Unity, .NET Core DI) ⟶ ช่วย resolve dependencies

Pros

- เปลี่ยน Implementation ได้ง่าย (EmailService ↔ SMSService)
- Test ง่ายขึ้น (ใช้ Mock Service)
- โครงสร้าง Clean & Flexible

Cons

- ถ้าไม่ใช้ Container ⟶ ต้อง inject เอง ⟶ โค้ดซับซ้อน
- ถ้าใช้ Container ⟶ อาจยากต่อการ Debug

Real-World Use Case

- Notification System (Email, SMS, Push Notification)

- Payment Service (CreditCard, PayPal)
- Logging Service (Console, File, Database)

---

5. Observer Pattern

Concept

- ความสัมพันธ์ **One-to-Many**
- เมื่อ Subject (Publisher) เปลี่ยน ⟶ แจ้ง Observer (Subscriber) ทุกตัว
- เหมาะกับระบบ **Event-driven**

Structure

- **Subject** ⟶ เก็บ Observer, แจ้งเมื่อ state เปลี่ยน
- **Observer Interface** ⟶ กำหนด Update()
- **ConcreteObserver** ⟶ ตอบสนองต่อการเปลี่ยนแปลง

Pros

- Loose Coupling ⟶ Subject ไม่ต้องรู้ Observer ทำงานยังไง
- เพิ่ม/ลบ Observer ได้ runtime

Cons

- ถ้ามี Observer มากเกินไป ⟶ อาจเกิด Performance Issue
- Debug ยาก (เหตุการณ์ propagate หลายชั้น)

Real-World Use Case

- Stock Price ⟶ แจ้งนักลงทุน
- Chat Application ⟶ แจ้งเมื่อมีข้อความใหม่
- UI Event ⟶ Button Click, Data Binding

☐ **การบูรณาการ (Integration Idea)**

สมมติคุณทำ **ระบบ E-Commerce (VB.NET)**

- **Singleton** ⟶ ใช้เป็น Logger (ทั้งระบบแชร์ object เดียว)
- **Factory Method** ⟶ ใช้สร้าง Payment Method (CreditCard, PayPal, QRCode)
- **Repository** ⟶ ใช้แยก Data Access เช่น ProductRepository, OrderRepository
- **Dependency Injection** ⟶ ส่ง Service เข้าสู่ Business Logic เช่น INotificationService
- **Observer** ⟶ แจ้งเตือน Stock Update ไปยัง Investor หรือ UI Realtime

---

Singleton Pattern (VB.NET)

1. แนวคิด (Concept)

Singleton คือ Design Pattern ที่ควบคุมให้คลาสหนึ่ง ๆ **มีอินสแตนซ์เดียว (Single Instance)** ในโปรแกรมทั้งหมด และต้องสามารถเข้าถึงได้จากทุกที่

➡ จุดสำคัญ:

- คลาส **มีเพียงอินสแตนซ์เดียว**
- ให้ **จุดเข้าถึงกลาง (Global Access Point)**
- ควบคุมการสร้างด้วย **private constructor**

---

2. โครงสร้าง (Structure)

```
+-------------------+
|   Singleton       |
+-------------------+
| - instance        |  ' ตัวแปร Shared เก็บอินสแตนซ์เดียว
| - New() Private   |  ' ป้องกันการสร้างจากภายนอก
+-------------------+
| + Instance        |  ' Property สำหรับเข้าถึงอินสแตนซ์
| + Method()        |  ' เมธอดปกติ
+-------------------+
```

---

3. จุดแข็ง (Pros)

☐ ประหยัดทรัพยากร (เช่น connection, logger)

☐ ควบคุม state ได้จากที่เดียว

☐ ใช้ง่าย เข้าถึงจากทุกที่

4. จุดอ่อน (Cons)

☐ อาจกลายเป็น **Global State** ⟶ โค้ดพึ่งพากันมาก (High Coupling)

☐ ทำ Unit Test ยาก (mock ยาก)

☐ Thread Safety ต้องระวัง (multi-thread)

---

5. ตัวอย่างโค้ด VB.NET (Thread-Safe Singleton)

```
Public Class Logger
    ' เก็บอินสแตนซ์เดียว
    Private Shared _instance As Logger
    Private Shared ReadOnly _lock As New Object()
```

```
    ' ป้องกันการ new จากภายนอก
    Private Sub New()
    End Sub

    ' Property สำหรับเข้าถึงอินสแตนซ์
    Public Shared ReadOnly Property Instance As Logger
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New Logger()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    ' เมธอดปกติ
    Public Sub Log(message As String)
        Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] {message}")
    End Sub
End Class
```

---

6. การใช้งาน (Usage)

```
Module Program
    Sub Main()
        ' เรียกใช้ Logger ผ่าน Instance
        Logger.Instance.Log("เริ่มระบบ")
        Logger.Instance.Log("บันทึกข้อมูลสินค้า")
        Logger.Instance.Log("ระบบปิดการทำงาน")
    End Sub
End Module
```

☐ Output:

```
[10:15:30] เริ่มระบบ
[10:15:30] บันทึกข้อมูลสินค้า
```

[10:15:30] ระบบปิดการทำงาน

---

7. Best Practices

1. **Lazy Initialization** ⟶ สร้าง object ตอนใช้งานจริง (ประหยัด resource)
2. **Thread-Safety** ⟶ ใช้ SyncLock ป้องกัน multi-thread สร้างอินสแตนซ์ซ้ำ
3. **Sealed Class (NotInheritable)** ⟶ กันไม่ให้ subclass สร้าง instance ใหม่
4. **ใช้ Dependency Injection แทน ถ้าโค้ดใหญ่** ⟶ เพราะ test ง่ายกว่า

---

8. การประยุกต์ใช้จริง (Use Cases)

- **Logger System** ⟶ บันทึก log จากทุก module ไปที่เดียว
- **Configuration Manager** ⟶ โหลดค่าคอนฟิกจากไฟล์ app.config
- **Database Connection Pool** ⟶ ควบคุมจำนวน connection ที่เปิด
- **Cache Manager** ⟶ เก็บข้อมูลที่ใช้บ่อย (ลดการ query DB)

ตัวอย่างโปรแกรม VB.NET แบบเต็มไฟล์ เกี่ยวกับ **Singleton Pattern**

☐ ผมจะแบ่งเป็น 2 ส่วน

- **พื้นฐาน 3 โปรแกรม** ⟶ Singleton ในรูปแบบที่ใช้บ่อย
- **แนวประยุกต์ 3 โปรแกรม** ⟶ นำ Singleton ไปใช้ในระบบจำลองจริง

ทุกตัวอย่างจะมี:

☐ โครงสร้างไฟล์โปรแกรม

☐ โค้ดเต็ม VB.NET

☐ คำอธิบายโค้ด

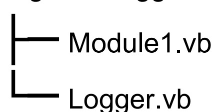
☐ ผลการรัน

☐ ชุดที่ 1: Singleton (พื้นฐาน 3 โปรแกรม)

☐ โปรแกรมที่ 1: Logger Singleton (พื้นฐานที่สุด)

โครงสร้าง

SingletonLogger/



Module1.vb

Logger.vb

Logger.vb

Public NotInheritable Class Logger

```
    Private Shared _instance As Logger
    Private Shared ReadOnly _lock As New Object()

    ' ป้องกันการสร้าง instance จากภายนอก
    Private Sub New()
    End Sub

    Public Shared ReadOnly Property Instance As Logger
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New Logger()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Sub Log(message As String)
        Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] {message}")
    End Sub
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Logger.Instance.Log("เริ่มโปรแกรม")
        Logger.Instance.Log("กำลังประมวลผล...")
        Logger.Instance.Log("จบโปรแกรม")
    End Sub
End Module
```

คำอธิบาย

- Logger เป็น Singleton → มีเพียง 1 instance
- ใช้ SyncLock เพื่อกัน multi-thread สร้างซ้ำ
- ใช้ Logger.Instance.Log() ได้จากทุกที่

ผลการรัน

[14:01:12] เริ่มโปรแกรม

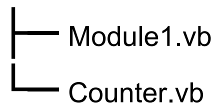
[14:01:12] กำลังประมวลผล...

[14:01:12] จบโปรแกรม

□ **โปรแกรมที่ 2: Counter Singleton (นับจำนวนเรียกใช้งาน)**

โครงสร้าง

SingletonCounter/



Module1.vb

Counter.vb

Counter.vb

```
Public NotInheritable Class Counter
    Private Shared _instance As Counter
    Private Shared ReadOnly _lock As New Object()
    Private _count As Integer = 0

    Private Sub New()
    End Sub

    Public Shared ReadOnly Property Instance As Counter
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New Counter()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Sub Increment()
        _count += 1
    End Sub
```

```
    Public Function GetCount() As Integer
        Return _count
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim c1 = Counter.Instance
        Dim c2 = Counter.Instance

        c1.Increment()
        c2.Increment()
        c1.Increment()

        Console.WriteLine("Count: " & c2.GetCount())
    End Sub
End Module
```

คำอธิบาย

- Counter เป็น Singleton
- ไม่ว่าจะเข้าผ่าน c1 หรือ c2 ⟶ คือ object เดียวกัน
- ค่า count จึงแชร์กัน

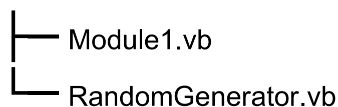
ผลการรัน

Count: 3

☐ โปรแกรมที่ 3: Random Generator Singleton

โครงสร้าง

SingletonRandom/



Module1.vb

RandomGenerator.vb

RandomGenerator.vb

```
Public NotInheritable Class RandomGenerator
    Private Shared _instance As RandomGenerator
    Private Shared ReadOnly _lock As New Object()
    Private ReadOnly _random As Random
```

```
    Private Sub New()
        _random = New Random()
    End Sub

    Public Shared ReadOnly Property Instance As RandomGenerator
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New RandomGenerator()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Function NextNumber(min As Integer, max As Integer) As Integer
        Return _random.Next(min, max)
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        For i = 1 To 5
            Console.WriteLine(RandomGenerator.Instance.NextNumber(1, 100))
        Next
    End Sub
End Module
```

คำอธิบาย

- ใช้ Random เดียวตลอดโปรแกรม
- ถ้าใช้ New Random() หลายครั้งอาจได้เลขซ้ำเพราะ seed ใกล้กัน
- Singleton ทำให้มั่นใจว่า Random ใช้เพียงตัวเดียว

ผลการรัน (ตัวเลขสุ่ม)

42

7

88

23

15

☐ ชุดที่ 2: Singleton (แนวประยุกต์ 3 โปรแกรม)

☐ โปรแกรมที่ 4: Configuration Manager Singleton

โครงสร้าง

```
SingletonConfig/
├── Module1.vb
└── ConfigManager.vb
```

ConfigManager.vb

```
Imports System.IO

Public NotInheritable Class ConfigManager
    Private Shared _instance As ConfigManager
    Private Shared ReadOnly _lock As New Object()
    Private ReadOnly _settings As Dictionary(Of String, String)

    Private Sub New()
        _settings = New Dictionary(Of String, String)()
        ' โหลดไฟล์ config.txt
        For Each line In File.ReadAllLines("config.txt")
            Dim parts = line.Split("="c)
            If parts.Length = 2 Then
                _settings(parts(0).Trim()) = parts(1).Trim()
            End If
        Next
    End Sub

    Public Shared ReadOnly Property Instance As ConfigManager
        Get
            SyncLock _lock
```

```
                If _instance Is Nothing Then
                    _instance = New ConfigManager()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Function GetValue(key As String) As String
        If _settings.ContainsKey(key) Then
            Return _settings(key)
        End If
        Return Nothing
    End Function
End Class
```

config.txt

```
Database=SQLServer
ConnectionString=Server=.;Database=Shop;Trusted_Connection=True;
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim db = ConfigManager.Instance.GetValue("Database")
        Dim conn = ConfigManager.Instance.GetValue("ConnectionString")

        Console.WriteLine("Database: " & db)
        Console.WriteLine("ConnectionString: " & conn)
    End Sub
End Module
```

ผลการรัน

```
Database: SQLServer
ConnectionString: Server=.;Database=Shop;Trusted_Connection=True;
```

☐ โปรแกรมที่ 5: Database Connection Singleton (จำลอง)

โครงสร้าง

```
SingletonDB/
├── Module1.vb
└── DatabaseConnection.vb
```

DatabaseConnection.vb

```
Public NotInheritable Class DatabaseConnection
    Private Shared _instance As DatabaseConnection
    Private Shared ReadOnly _lock As New Object()

    Private Sub New()
        Console.WriteLine("สร้างการเชื่อมต่อ Database...")
    End Sub

    Public Shared ReadOnly Property Instance As DatabaseConnection
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New DatabaseConnection()
                End If
                Return _instance
            End SyncLock
        End Get
    End Property

    Public Sub Query(sql As String)
        Console.WriteLine("Execute SQL: " & sql)
    End Sub
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim db1 = DatabaseConnection.Instance
        db1.Query("SELECT * FROM Products")

        Dim db2 = DatabaseConnection.Instance
```

```
        db2.Query("INSERT INTO Products VALUES ('Mouse')")

        Console.WriteLine(Object.ReferenceEquals(db1, db2)) ' True
    End Sub
End Module
```

ผลการรัน

สร้างการเชื่อมต่อ Database...

Execute SQL: SELECT * FROM Products

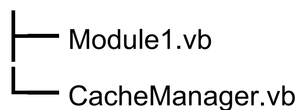
Execute SQL: INSERT INTO Products VALUES ('Mouse')

True

☐ **โปรแกรมที่ 6: Cache Manager Singleton**

โครงสร้าง

SingletonCache/



Module1.vb

CacheManager.vb

CacheManager.vb

```
Public NotInheritable Class CacheManager
    Private Shared _instance As CacheManager
    Private Shared ReadOnly _lock As New Object()
    Private ReadOnly _cache As Dictionary(Of String, String)

    Private Sub New()
        _cache = New Dictionary(Of String, String)()
    End Sub

    Public Shared ReadOnly Property Instance As CacheManager
        Get
            SyncLock _lock
                If _instance Is Nothing Then
                    _instance = New CacheManager()
                End If
                Return _instance
            End SyncLock
```

```
        End Get
    End Property

    Public Sub SetValue(key As String, value As String)
        _cache(key) = value
    End Sub

    Public Function GetValue(key As String) As String
        If _cache.ContainsKey(key) Then
            Return _cache(key)
        End If
        Return Nothing
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        CacheManager.Instance.SetValue("User1", "Somchai")
        CacheManager.Instance.SetValue("User2", "Somsri")

        Console.WriteLine("User1: " & CacheManager.Instance.GetValue("User1"))
        Console.WriteLine("User2: " & CacheManager.Instance.GetValue("User2"))
    End Sub
End Module
```

ผลการรัน

```
User1: Somchai
User2: Somsri
```

☐ **สรุป**

- **พื้นฐาน 3 ตัวอย่าง**
    1. Logger Singleton ⟶ ใช้สำหรับ Logging
    2. Counter Singleton ⟶ แชร์ตัวนับทั่วระบบ
    3. Random Generator Singleton ⟶ สุ่มเลขไม่ซ้ำ seed

- **แนวประยุกต์ 3 ตัวอย่าง**
  4. ConfigManager ⟶ โหลด config จากไฟล์
  5. DatabaseConnection ⟶ จำลองการเชื่อมต่อ DB
  6. CacheManager ⟶ เก็บข้อมูลในหน่วยความจำแบบ Shared

Factory Method Pattern (VB.NET)

1. แนวคิด (Concept)

Factory Method เป็น **Creational Design Pattern** ที่ใช้เพื่อ

☐ กำหนด **interface สำหรับสร้าง object** แต่ให้ **subclass เป็นผู้ตัดสินใจว่าควรสร้าง object แบบไหน**

กล่าวคือ **Factory Method แทนการใช้คำสั่ง New โดยตรง** ⟶ เพื่อให้โค้ดยืดหยุ่น รองรับการเปลี่ยนแปลงหรือการเพิ่มชนิดของ object ได้ง่าย

---

2. โครงสร้าง (Structure)

```
+---------------------+
|   Product (Interface)|
+---------------------+
| +Operation()        |
+---------------------+

+---------------------+
| ConcreteProductA    |
+---------------------+
| +Operation()        |
+---------------------+

+---------------------+
| ConcreteProductB    |
+---------------------+
| +Operation()        |
+---------------------+

+---------------------+
```

```
| Creator (Abstract)  |
+---------------------+
| +FactoryMethod()    | ---> return Product
+---------------------+

+---------------------+
| ConcreteCreatorA    |
+---------------------+
| +FactoryMethod()    | ---> return ConcreteProductA
+---------------------+

+---------------------+
| ConcreteCreatorB    |
+---------------------+
| +FactoryMethod()    | ---> return ConcreteProductB
+---------------------+
```

---

3. จุดแข็ง (Pros)

- ☐ ลดการผูกติดกับ class จริง (decoupling)
- ☐ เพิ่ม Product ใหม่ได้ง่าย โดยเพิ่ม subclass ของ Creator
- ☐ ส่งเสริมหลักการ OOP (Open/Closed Principle)

4. จุดอ่อน (Cons)

- ☐ จำนวน class เยอะขึ้น (เพิ่ม complexity)
- ☐ ไม่เหมาะกับงานเล็ก ๆ ที่ไม่ต้องเปลี่ยน product บ่อย

---

5. ตัวอย่างโค้ด VB.NET (พื้นฐาน)

สมมุติว่าเรามีระบบ **สร้างรายงาน (Report)** ที่มีหลายรูปแบบ เช่น **PDF** และ **Excel**

Product.vb

```
' Product interface
Public Interface IReport
    Sub Generate()
End Interface
```

Concrete Products

```
' PDF Report
```

```
Public Class PdfReport
    Implements IReport

    Public Sub Generate() Implements IReport.Generate
        Console.WriteLine("สร้างรายงาน PDF สำเร็จ")
    End Sub
End Class

' Excel Report
Public Class ExcelReport
    Implements IReport

    Public Sub Generate() Implements IReport.Generate
        Console.WriteLine("สร้างรายงาน Excel สำเร็จ")
    End Sub
End Class
```

Creator (Factory)

```
' Abstract Factory
Public MustInherit Class ReportCreator
    Public MustOverride Function CreateReport() As IReport
End Class

' Concrete Factory for PDF
Public Class PdfReportCreator
    Inherits ReportCreator

    Public Overrides Function CreateReport() As IReport
        Return New PdfReport()
    End Function
End Class

' Concrete Factory for Excel
Public Class ExcelReportCreator
    Inherits ReportCreator
```

```
    Public Overrides Function CreateReport() As IReport
        Return New ExcelReport()
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim pdfFactory As ReportCreator = New PdfReportCreator()
        Dim excelFactory As ReportCreator = New ExcelReportCreator()

        Dim pdf As IReport = pdfFactory.CreateReport()
        pdf.Generate()

        Dim excel As IReport = excelFactory.CreateReport()
        excel.Generate()
    End Sub
End Module
```

---

6. ผลการรัน

สร้างรายงาน PDF สำเร็จ

สร้างรายงาน Excel สำเร็จ

---

7. Best Practices

- ใช้เมื่อ **มีหลาย subclass** ที่อาจถูกเลือกสร้างใน runtime
- ใช้คู่กับ **Dependency Injection (DI)** เพื่อ inject Factory เข้าไปใน Service
- ใช้กับ **Plugin Architecture** → เช่น โปรแกรมรองรับหลาย format, หลาย driver

---

8. การประยุกต์ใช้จริง (Use Cases)

- **Database Provider Factory** → สร้าง connection สำหรับ SQL Server, Oracle, MySQL
- **UI Control Factory** → สร้างปุ่ม/กล่องข้อความตาม OS (Windows, Mac, Linux)
- **Document Exporter** → รองรับการ export เป็น PDF, Excel, Word
- **Payment Gateway** → รองรับการจ่ายผ่าน PayPal, Credit Card, PromptPay

---

- **3 โปรแกรมพื้นฐาน** → ทำความเข้าใจแก่นของ Factory Method
- **3 โปรแกรมแนวประยุกต์** → ใช้ Factory Method ในระบบจริง

ทุกโปรแกรมจะมี:

☐ โครงสร้างไฟล์

☐ โค้ด VB.NET แบบเต็มไฟล์

☐ คำอธิบายโค้ด

☐ ผลการรัน

☐ **ชุดที่ 1: Factory Method (พื้นฐาน 3 โปรแกรม)**

☐ **โปรแกรมที่ 1: สร้างรายงาน (PDF / Excel)**

โครงสร้าง

```
FactoryReport/
├── Module1.vb
├── IReport.vb
├── PdfReport.vb
├── ExcelReport.vb
├── ReportCreator.vb
├── PdfReportCreator.vb
└── ExcelReportCreator.vb
```

IReport.vb

```
Public Interface IReport
    Sub Generate()
End Interface
```

PdfReport.vb

```
Public Class PdfReport
    Implements IReport

    Public Sub Generate() Implements IReport.Generate
        Console.WriteLine("สร้างรายงาน PDF สำเร็จ")
    End Sub
End Class
```

ExcelReport.vb

```
Public Class ExcelReport
    Implements IReport

    Public Sub Generate() Implements IReport.Generate
        Console.WriteLine("สร้างรายงาน Excel สำเร็จ")
    End Sub
End Class
```

ReportCreator.vb

```
Public MustInherit Class ReportCreator
    Public MustOverride Function CreateReport() As IReport
End Class
```

PdfReportCreator.vb

```
Public Class PdfReportCreator
    Inherits ReportCreator

    Public Overrides Function CreateReport() As IReport
        Return New PdfReport()
    End Function
End Class
```

ExcelReportCreator.vb

```
Public Class ExcelReportCreator
    Inherits ReportCreator

    Public Overrides Function CreateReport() As IReport
        Return New ExcelReport()
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim pdfCreator As ReportCreator = New PdfReportCreator()
        Dim excelCreator As ReportCreator = New ExcelReportCreator()
```

```
        Dim pdf = pdfCreator.CreateReport()
        pdf.Generate()

        Dim excel = excelCreator.CreateReport()
        excel.Generate()
    End Sub
End Module
```

ผลการรัน

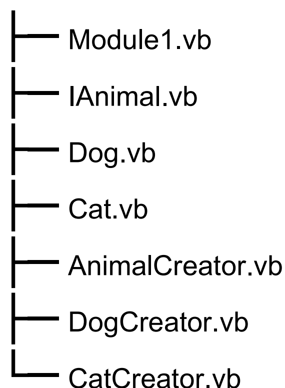
สร้างรายงาน PDF สำเร็จ

สร้างรายงาน Excel สำเร็จ

☐ โปรแกรมที่ 2: สัตว์ (Dog / Cat)

โครงสร้าง

FactoryAnimal/



```
├── Module1.vb
├── IAnimal.vb
├── Dog.vb
├── Cat.vb
├── AnimalCreator.vb
├── DogCreator.vb
└── CatCreator.vb
```

IAnimal.vb

```
Public Interface IAnimal
    Sub Speak()
End Interface
```

Dog.vb

```
Public Class Dog
    Implements IAnimal

    Public Sub Speak() Implements IAnimal.Speak
        Console.WriteLine("Woof! (หมาเห่า)")
    End Sub
End Class
```

Cat.vb

```
Public Class Cat
    Implements IAnimal

    Public Sub Speak() Implements IAnimal.Speak
        Console.WriteLine("Meow! (แมวร้อง)")
    End Sub
End Class
```

AnimalCreator.vb

```
Public MustInherit Class AnimalCreator
    Public MustOverride Function CreateAnimal() As IAnimal
End Class
```

DogCreator.vb

```
Public Class DogCreator
    Inherits AnimalCreator

    Public Overrides Function CreateAnimal() As IAnimal
        Return New Dog()
    End Function
End Class
```

CatCreator.vb

```
Public Class CatCreator
    Inherits AnimalCreator

    Public Overrides Function CreateAnimal() As IAnimal
        Return New Cat()
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim dogCreator As AnimalCreator = New DogCreator()
        Dim catCreator As AnimalCreator = New CatCreator()

        Dim dog = dogCreator.CreateAnimal()
```

```
        dog.Speak()

        Dim cat = catCreator.CreateAnimal()
        cat.Speak()
    End Sub
End Module
```

ผลการรัน

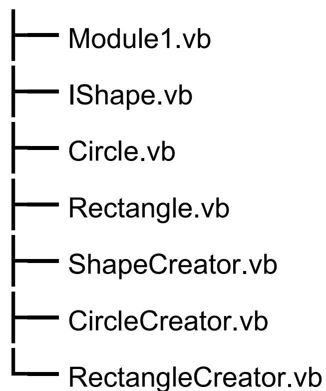
Woof! (หมาเห่า)

Meow! (แมวร้อง)

☐ **โปรแกรมที่ 3: Shape (Circle / Rectangle)**

โครงสร้าง

FactoryShape/



IShape.vb

```
Public Interface IShape
    Sub Draw()
End Interface
```

Circle.vb

```
Public Class Circle
    Implements IShape

    Public Sub Draw() Implements IShape.Draw
        Console.WriteLine("วาดวงกลม")
    End Sub
End Class
```

Rectangle.vb

```
Public Class Rectangle
```

```
    Implements IShape

    Public Sub Draw() Implements IShape.Draw
        Console.WriteLine("วาดสี่เหลี่ยม")
    End Sub
End Class
```

ShapeCreator.vb

```
Public MustInherit Class ShapeCreator
    Public MustOverride Function CreateShape() As IShape
End Class
```

CircleCreator.vb

```
Public Class CircleCreator
    Inherits ShapeCreator

    Public Overrides Function CreateShape() As IShape
        Return New Circle()
    End Function
End Class
```

RectangleCreator.vb

```
Public Class RectangleCreator
    Inherits ShapeCreator

    Public Overrides Function CreateShape() As IShape
        Return New Rectangle()
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim circleCreator As ShapeCreator = New CircleCreator()
        Dim rectCreator As ShapeCreator = New RectangleCreator()

        circleCreator.CreateShape().Draw()
        rectCreator.CreateShape().Draw()
```

```
    End Sub
End Module
```

ผลการรัน

วาดวงกลม

วาดสี่เหลี่ยม

☐ **ชุดที่ 2: Factory Method (แนวประยุกต์ 3 โปรแกรม)**

☐ **โปรแกรมที่ 4: Payment Gateway (CreditCard / PayPal)**

โครงสร้าง

```
FactoryPayment/
 ├── Module1.vb
 ├── IPayment.vb
 ├── CreditCardPayment.vb
 ├── PayPalPayment.vb
 ├── PaymentCreator.vb
 ├── CreditCardCreator.vb
 └── PayPalCreator.vb
```

IPayment.vb

```
Public Interface IPayment
    Sub Pay(amount As Decimal)
End Interface
```

CreditCardPayment.vb

```
Public Class CreditCardPayment
    Implements IPayment

    Public Sub Pay(amount As Decimal) Implements IPayment.Pay
        Console.WriteLine($"ชำระเงิน {amount:C} ผ่านบัตรเครดิต")
    End Sub
End Class
```

PayPalPayment.vb

```
Public Class PayPalPayment
    Implements IPayment
```

```
    Public Sub Pay(amount As Decimal) Implements IPayment.Pay
        Console.WriteLine($"ชำระเงิน {amount:C} ผ่าน PayPal")
    End Sub
End Class
```

PaymentCreator.vb

```
Public MustInherit Class PaymentCreator
    Public MustOverride Function CreatePayment() As IPayment
End Class
```

CreditCardCreator.vb

```
Public Class CreditCardCreator
    Inherits PaymentCreator

    Public Overrides Function CreatePayment() As IPayment
        Return New CreditCardPayment()
    End Function
End Class
```

PayPalCreator.vb

```
Public Class PayPalCreator
    Inherits PaymentCreator

    Public Overrides Function CreatePayment() As IPayment
        Return New PayPalPayment()
    End Function
End Class
```

Module1.vb

```
Module Module1
    Sub Main()
        Dim ccCreator As PaymentCreator = New CreditCardCreator()
        Dim ppCreator As PaymentCreator = New PayPalCreator()

        ccCreator.CreatePayment().Pay(500)
        ppCreator.CreatePayment().Pay(1000)
    End Sub
```