ICT365

Software Development Frameworks

Dr Afaq Shah

Murdoch UNIVERSITY

Introduction to Design Patterns

Murdoch
UNIVERSITY

In this Topic

UML class diagrams

Introduction to Design patterns

Façade

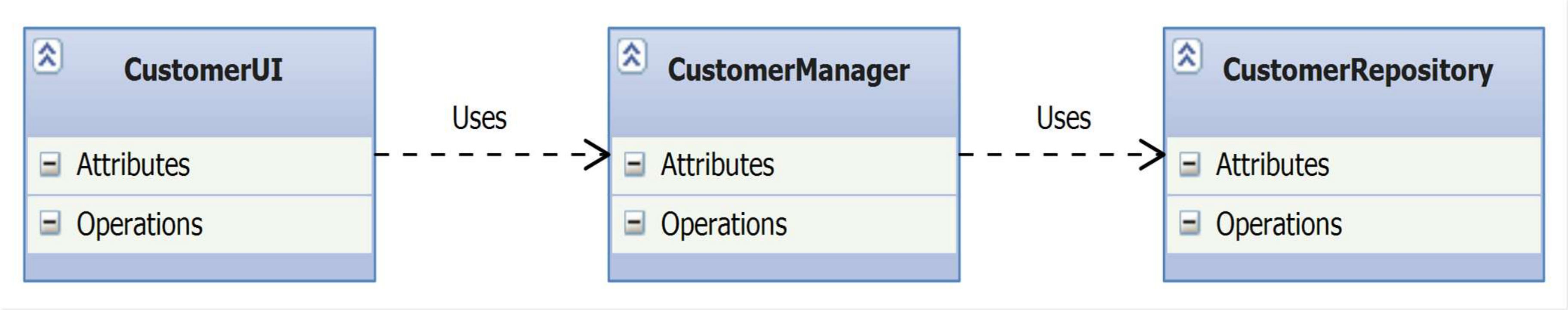
Adapter

UML Diagram - Class Relationships

- Dependency
- Association
- Aggregation
- Composition
- Generalization/Inheritance

Acknowledgements: Slides taken from - “Design Patterns (GoF) in .NET” by Aniruddha Chakrabarti

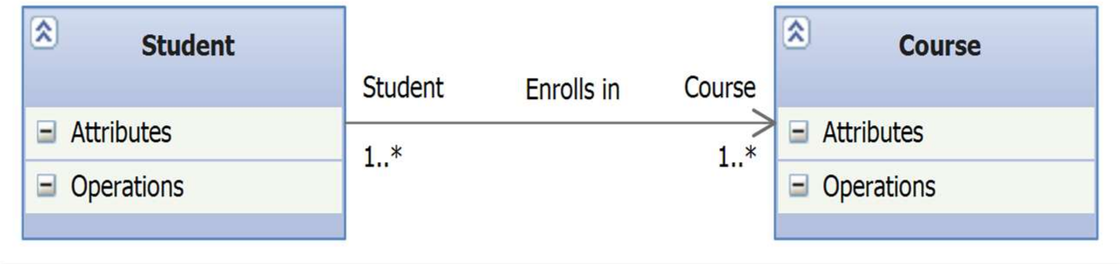
Dependency



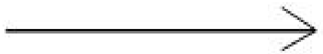
Dependency - Weaker form of relationship which indicates that one class depends on another because it uses it at some point of time.

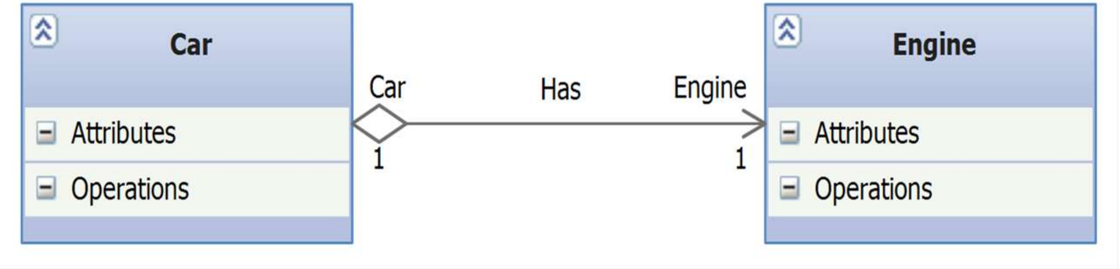
Dependency exists if a class is a parameter variable or local variable of a method of another class.

Association, Aggregation and Composition

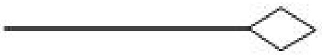


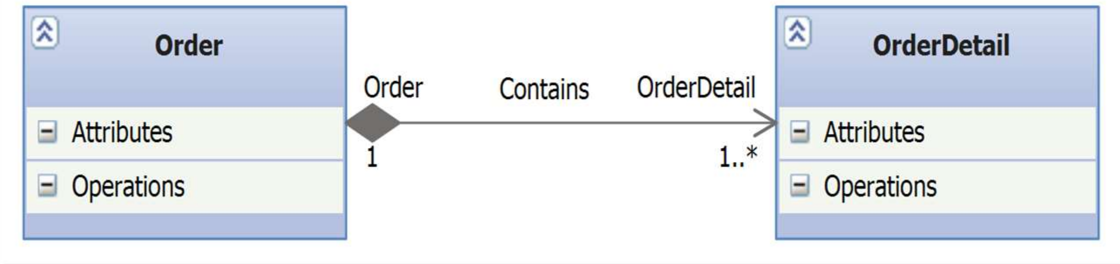
Association – Loose form of relationship (Student can enroll in multiple Course, and A Course can have multiple Student)



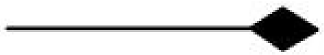


Aggregation - Whole part relationship. Part can exist without Whole.
(Engine can exist even if Car is destroyed, the same Engine could be used in a different Car)

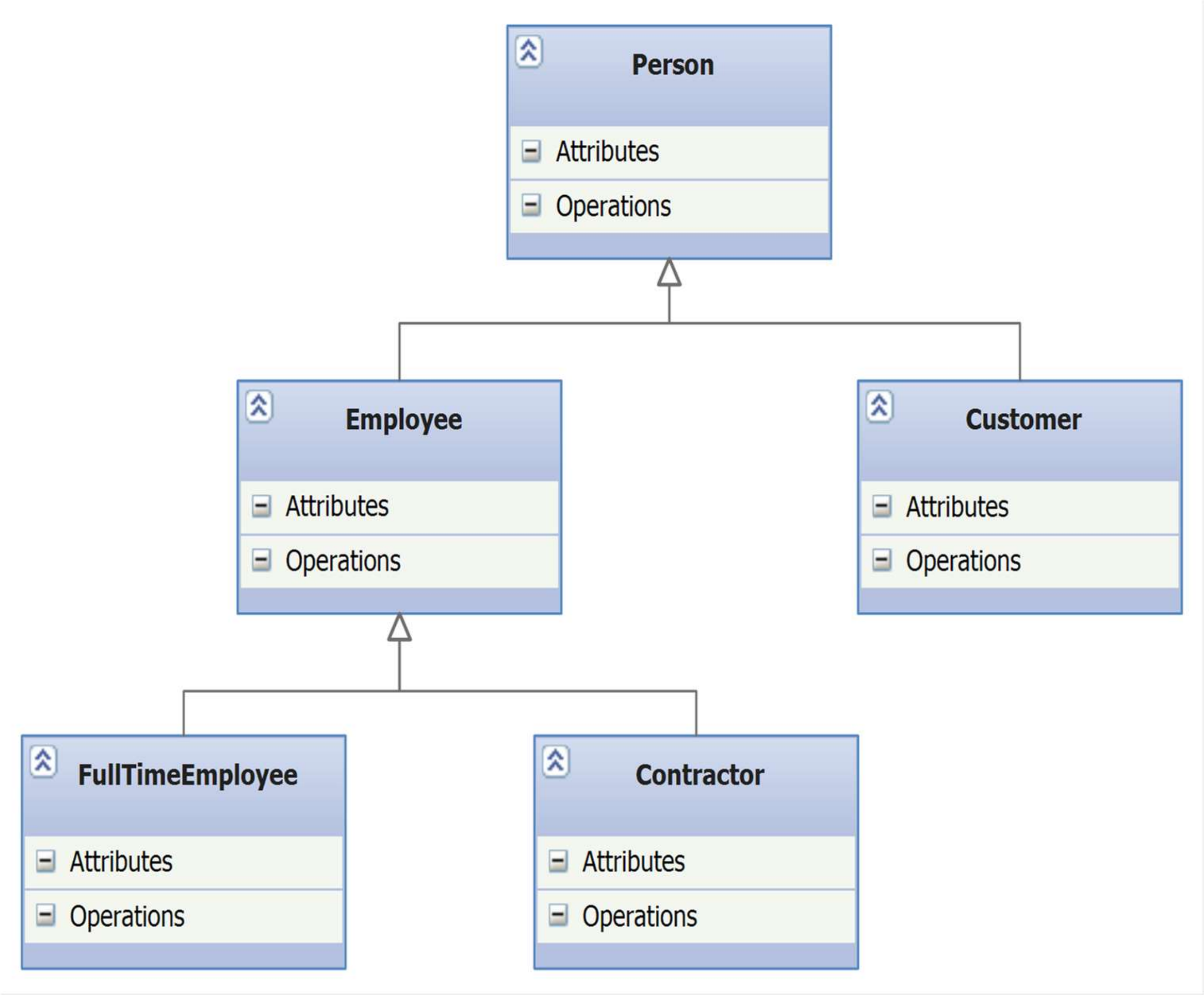




Composition – Stronger form of whole part relationship. Part can not exist without Whole.
(OrderDetail can not exist if Order is deleted. If Order is deleted, OrderDetail also gets deleted)

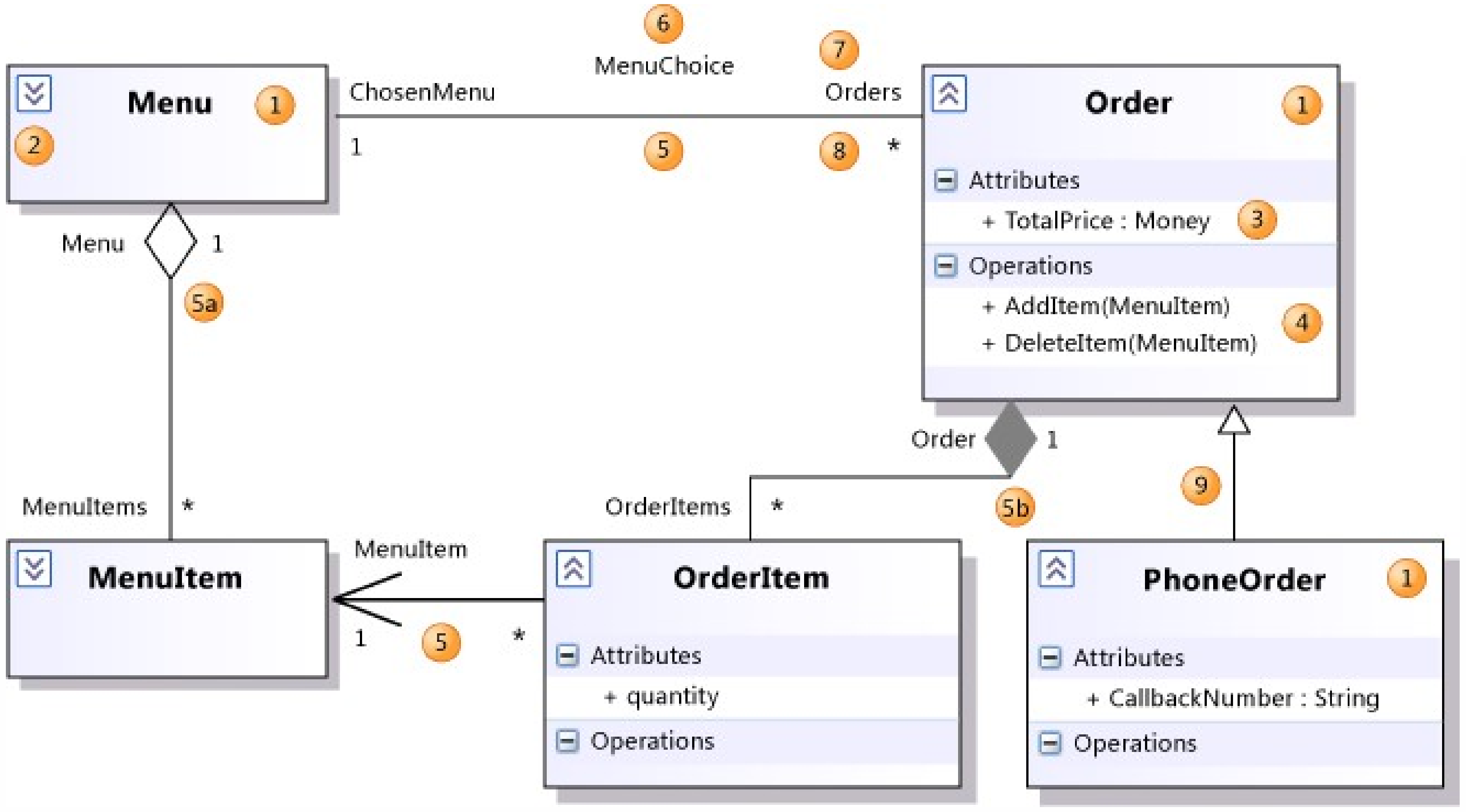


Generalization / Inheritance



UML Class Diagrams: Reference

- https://msdn.microsoft.com/en-us/library/dd409437.aspx



What are Design Patterns

- A *pattern* is a solution to a standard problem
- General reusable solution to a commonly occurring problem in software design.
- Extension of OOP and OOAD.
- Description or template for how to solve a problem that can be used in many different situations.
- Mostly documented with the following sections

Intent

Motivation (Forces)

Structure

Participants

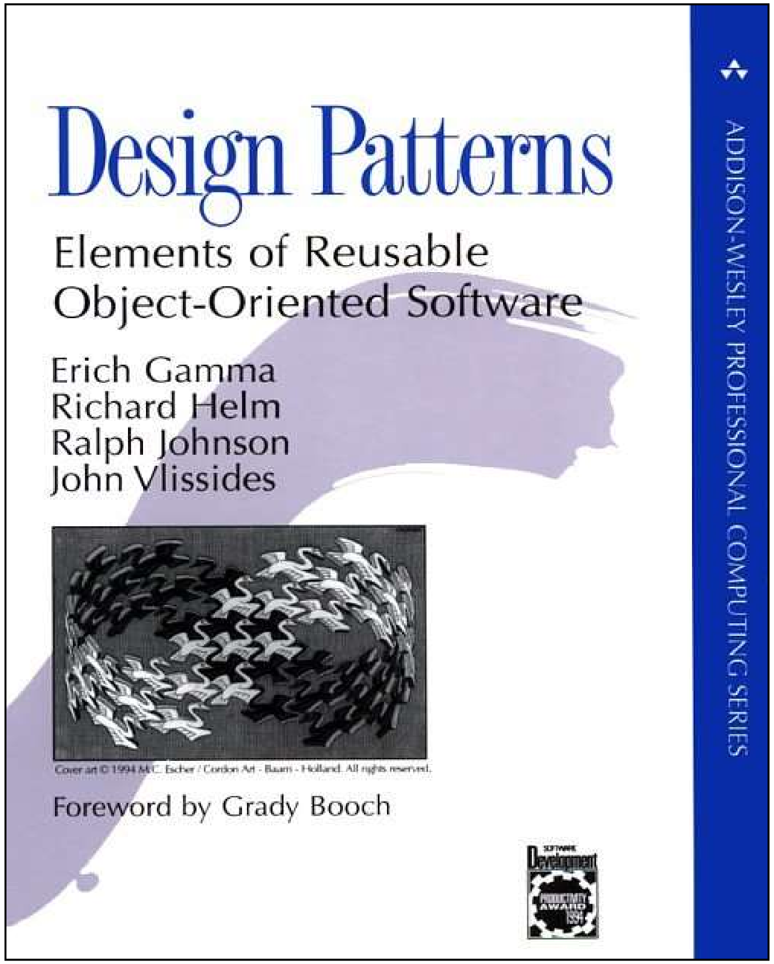
Implementation

Known Uses

Related Patterns

History of Design Patterns

- Patterns originated as an architectural concept by Christopher Alexander - 1977
- Kent Beck and Ward Cunningham applied patterns to programming and presented their results at OOPSLA conference - 1987
- Gained popularity after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published by "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides) – 1994
- First *Pattern Languages of Programming* Conference was held – 1994
- Following year, the *Portland Pattern Repository* was set up for documentation of design patterns.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns

- Design Patterns provide standardized and efficient solutions to software design and programming problems.
- However, you have to take care to select the right pattern for the right problem.
- You may also create your own custom Design Patterns.
  - Whenever you come up with a certain solution that is reusable in a vast majority of your projects.
- Design Patterns are divided into 3 categories :
  - Creational Patterns,
  - Structural Patterns and
  - Behavioral Patterns.

This and following slides come from:
https://csharpdesignpatterns.codeplex.com/wikipage?title=Adapter%20Pattern&referringTitle=Home

List of Design Patterns

- **Creational** Patterns

  Singleton

  Abstract Factory

  Builder

  Factory Method

  Prototype

- **Structural** Patterns

  Adapter

  Bridge

  Composite

  Decorator

  Façade

  Flyweight

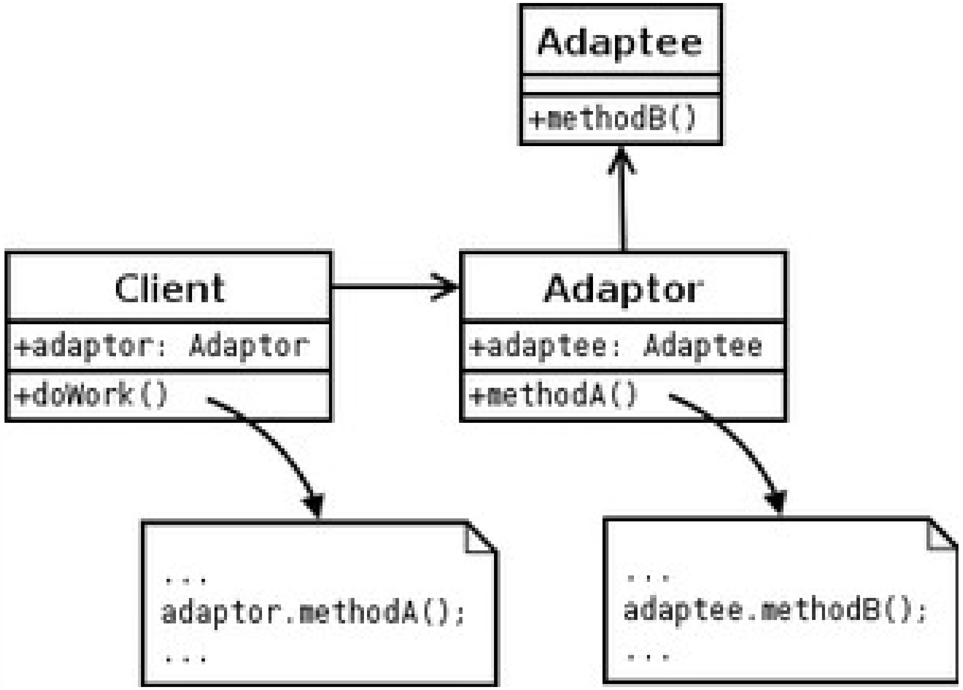
  Proxy

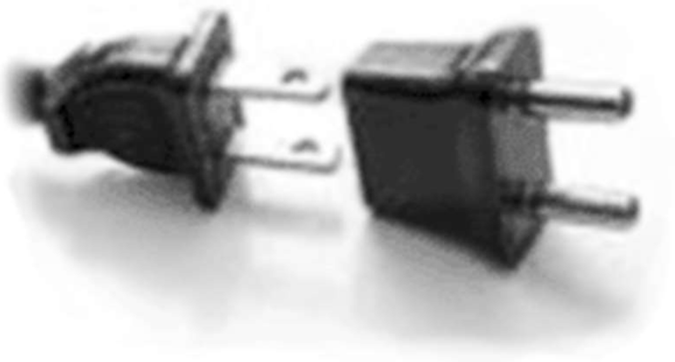
- **Behavioral** Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Adapter Design Pattern

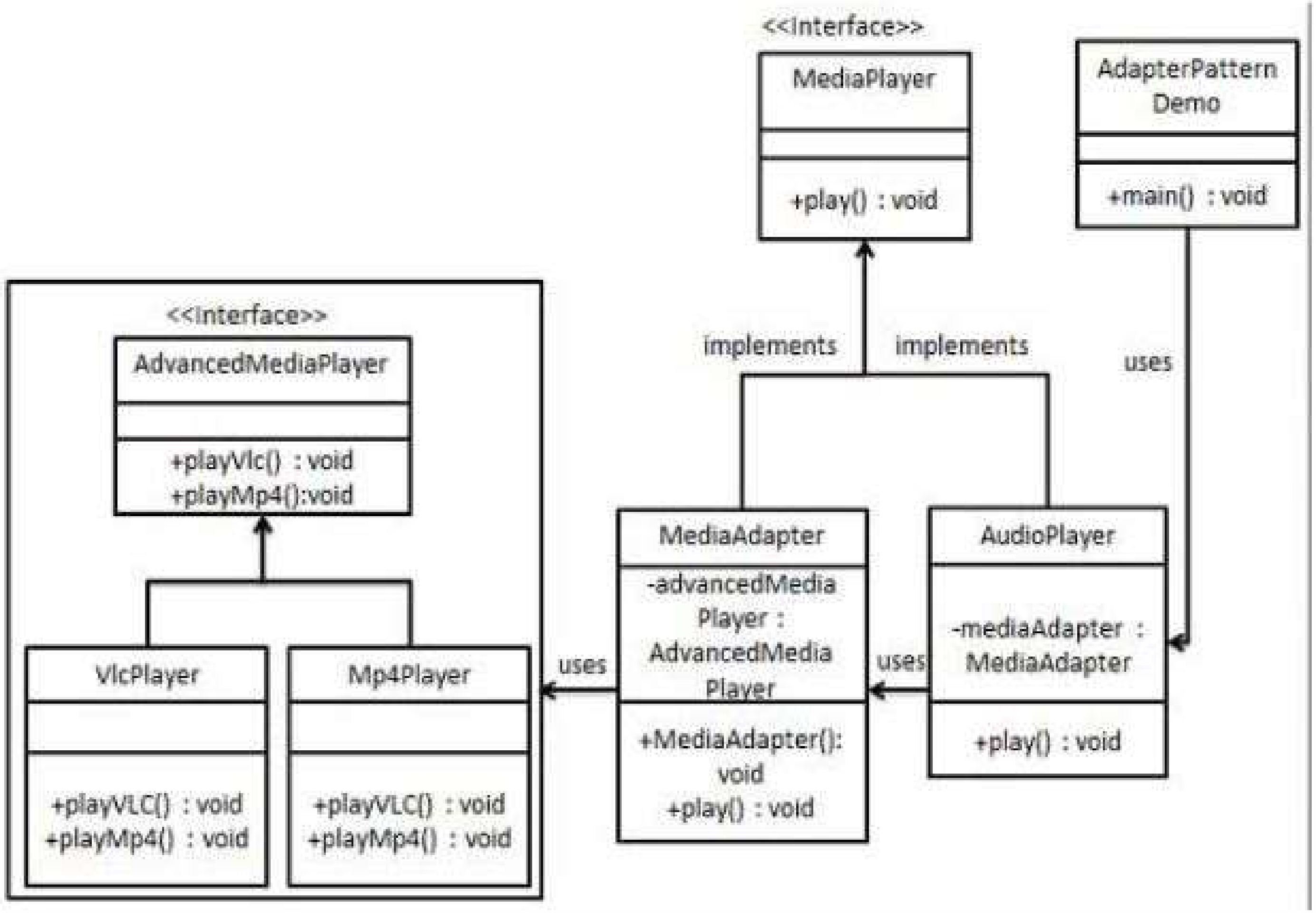
- Adapter pattern (also called wrapper pattern or wrapper) translates one interface for a class into a compatible interface.
- Allows classes (with incompatible interfaces) to work together
- The adapter translates calls to its interface into calls to original interface.
- Responsible for transforming data.
- Often used while working with existing API/code base

Adapter Class Diagram





Adapter Pattern



Pattern Name: Adapter

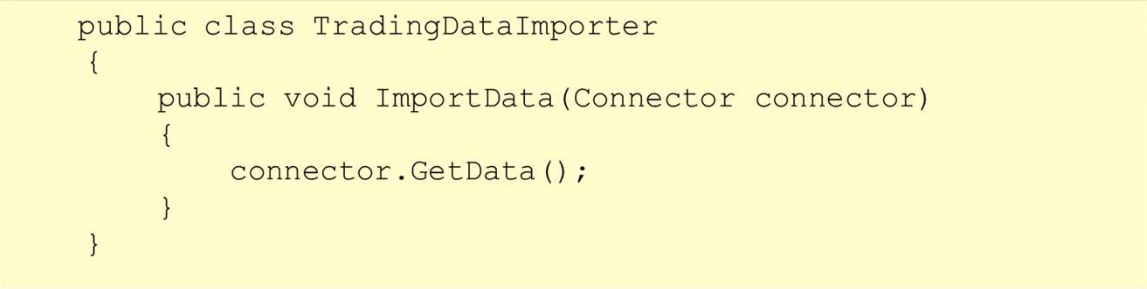
Murdoch UNIVERSITY

Short Description:

- Match interfaces of classes with different interfaces

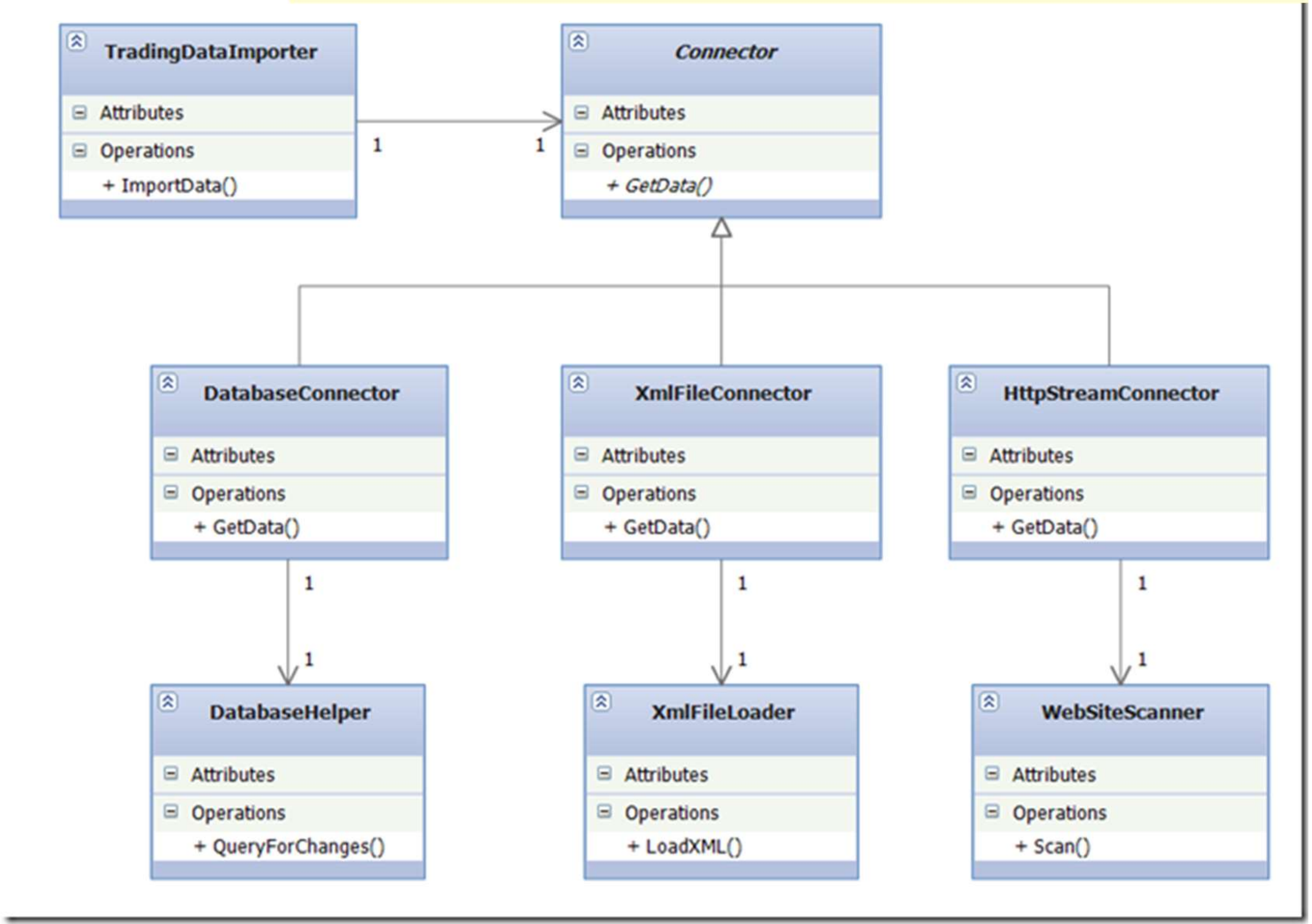
Usage:

- Often used and easy to implement, useful if classes need to work together that have incompatible existing interfaces.



```
public class TradingDataImporter
 {
      public void ImportData(Connector connector)
      {
           connector.GetData();
      }
 }
```

cd Adapter



Explanation

- The TradingDataImporter class acts as a client using classes with an existing Connector interface.

```
public class TradingDataImporter
 {
     public void ImportData(Connector connector)
     {
         connector.GetData();
     }
 }
```

- The abstract Adapter class defines the interface that the client class knows and that it can work with.
- The concrete Adapter classes convert the interface of the incompatible classes into an interface the client expects. They make different existing interfaces work together.

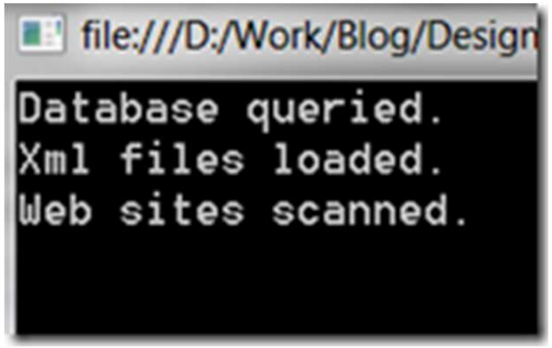
```
public abstract class Connector
{
    public abstract void GetData();
}

public class DatabaseConnector : Connector
{
    public override void GetData()
    {
        var databaseHelper = new DatabaseHelper();
        databaseHelper.QueryForChanges();
    }
}

public class XmlFileConnector : Connector
{
    public override void GetData()
    {
        var xmlfileLoader = new XmlFileLoader();
        xmlfileLoader.LoadXML();
    }
}

public class HttpStreamConnector : Connector
{
    public override void GetData()
    {
        var websiteScanner = new WebSiteScanner();
        websiteScanner.Scan();
    }
}
```

```
public class DatabaseHelper
{
    public void QueryForChanges()
    {
        Console.WriteLine("Database queried.");
    }
}

public class WebSiteScanner
{
    public void Scan()
    {
        Console.WriteLine("Web sites scanned.");
    }
}

public class XmlFileLoader
{
    public void LoadXML()
    {
        Console.WriteLine("Xml files loaded.");
    }
}
```

- Here are some examples of different adaptee classes that implement different interfaces. However, the client expects a generic interface that they currently don’t provide. That is why they get wrapped by the concrete adapter classes to make them compatible with the client.

```
    public static void Adapter()
    {
        var tradingdataImporter = new TradingDataImporter();

        Connector databaseConnector =
new DatabaseConnector();
        tradingdataImporter.ImportData(databaseConnector);

        Connector xmlfileConnector = new XmlFileConnector();
        tradingdataImporter.ImportData(xmlfileConnector);

        Connector httpstreamConnector =
new HttpStreamConnector();
        tradingdataImporter.ImportData(httpstreamConnector);


        Console.ReadKey();
    }
```

- Correct classes are instantiated during runtime:



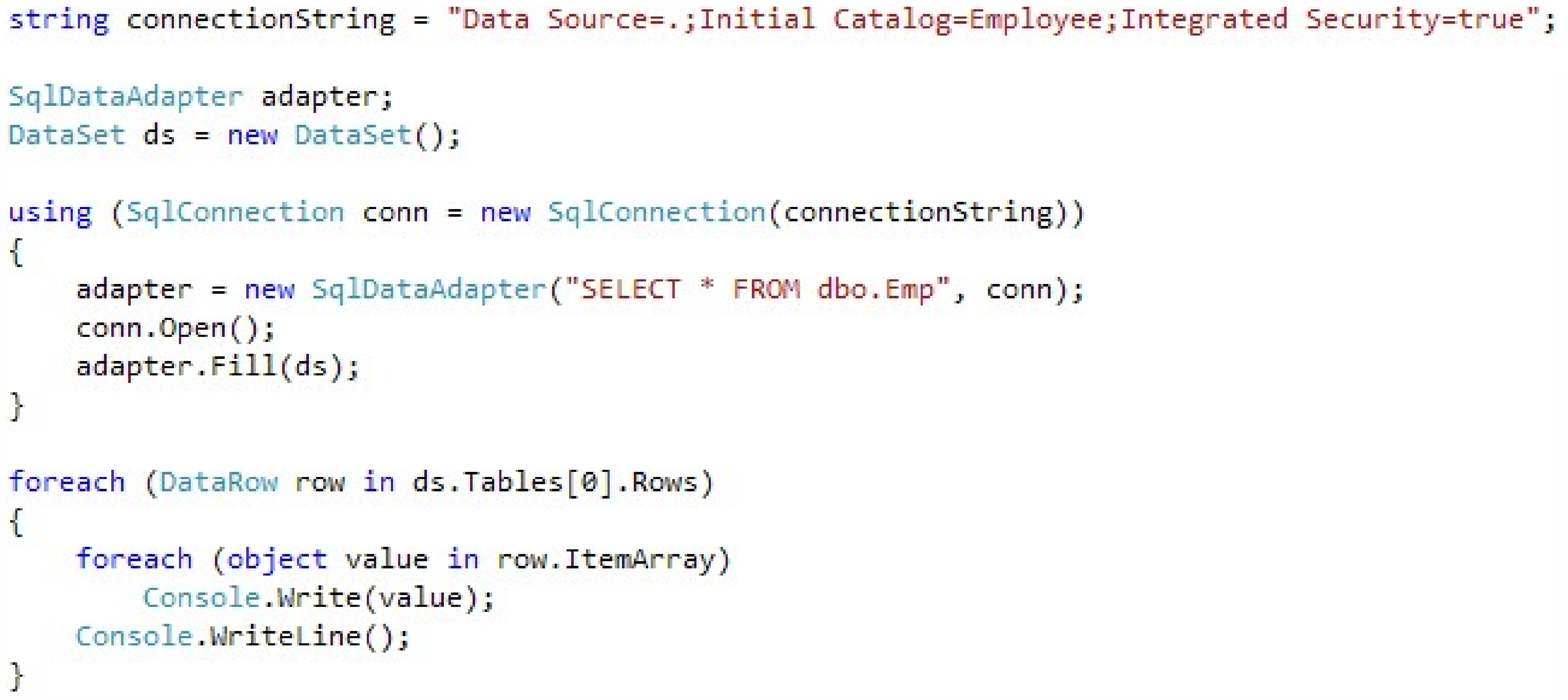
```
Database queried.
Xml files loaded.
Web sites scanned.
```

Adapter Pattern in ADO.NET

Data Adapters adapt data from different source (SQL Server, Oracle, ODBC, OLE DB) to dataset which is data-source unaware

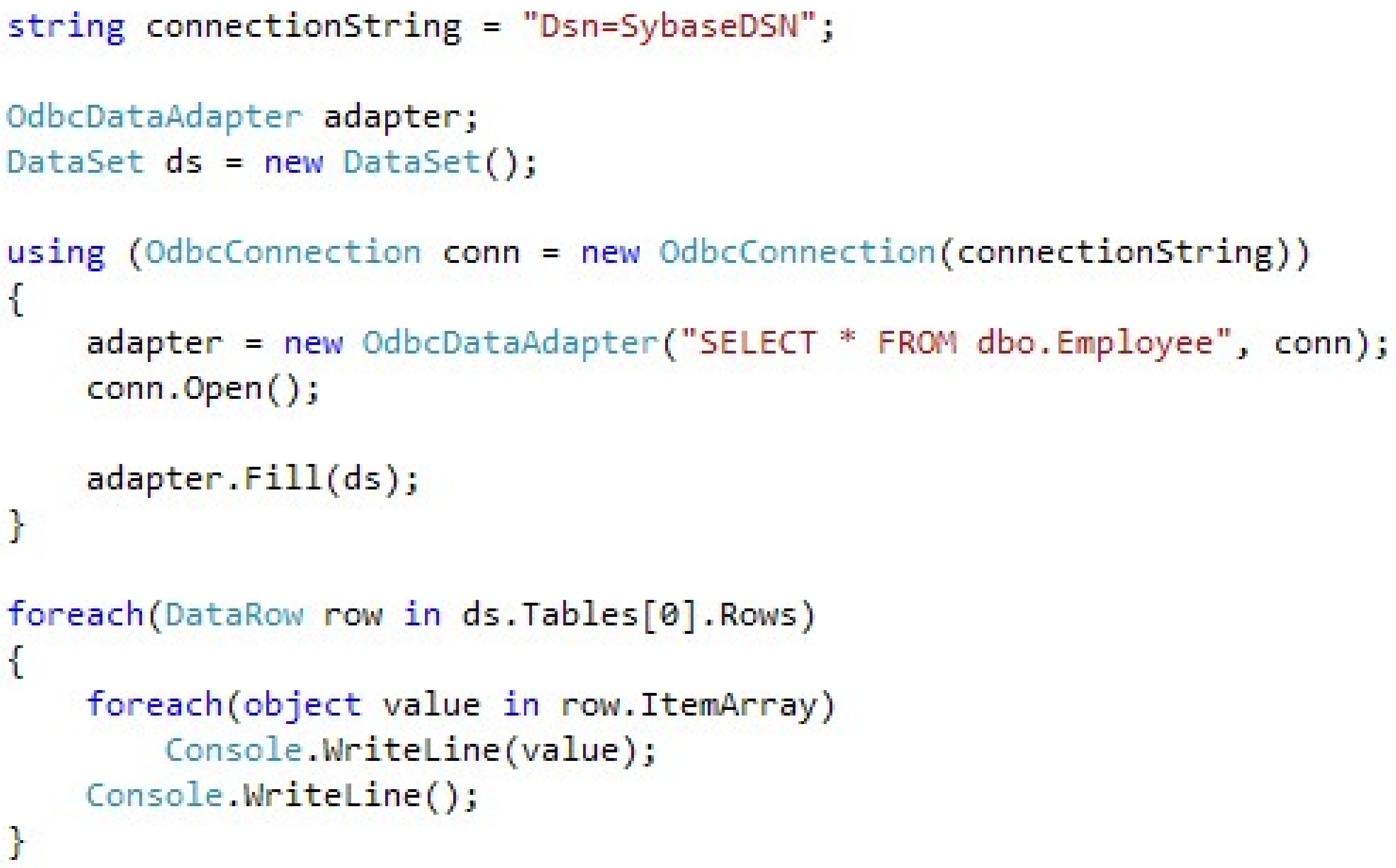
Different Data Adapter classes are used

- SqlDataAdapter
- OdbcDataAdapter
- OleDbDataAdapter



```
string connectionString = "Data Source=.;Initial Catalog=Employee;Integrated Security=true";

SqlDataAdapter adapter;
DataSet ds = new DataSet();

using (SqlConnection conn = new SqlConnection(connectionString))
{
    adapter = new SqlDataAdapter("SELECT * FROM dbo.Emp", conn);
    conn.Open();
    adapter.Fill(ds);
}

foreach (DataRow row in ds.Tables[0].Rows)
{
    foreach (object value in row.ItemArray)
        Console.Write(value);
    Console.WriteLine();
}
```

Adapter Pattern in ADO.NET – Cont'd



```
string connectionString = "Dsn=SybaseDSN";

OdbcDataAdapter adapter;
DataSet ds = new DataSet();

using (OdbcConnection conn = new OdbcConnection(connectionString))
{
    adapter = new OdbcDataAdapter("SELECT * FROM dbo.Employee", conn);
    conn.Open();

    adapter.Fill(ds);
}

foreach(DataRow row in ds.Tables[0].Rows)
{
    foreach(object value in row.ItemArray)
        Console.WriteLine(value);
    Console.WriteLine();
}
```

Singleton Pattern

- Used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object.
- Useful when exactly one object is needed to coordinate actions across the system.
- Common Uses:

Abstract Factory, Builder and Prototype patterns can use Singletons in their implementation.

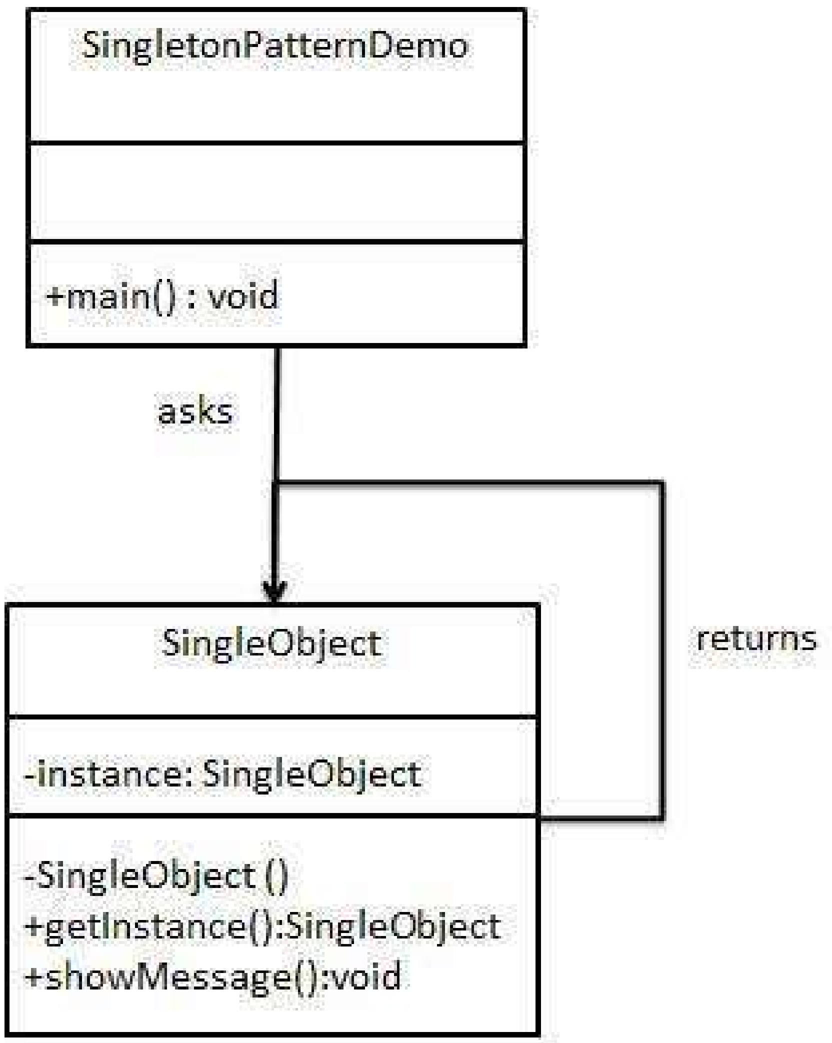
Facade objects are often Singletons because only one Facade object is required.

Singletons are often preferred to global variables because:

They don't pollute the global name space (or, in languages with namespaces, their containing namespace) with unnecessary variables.

They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

Singleton Pattern



SingletonPatternDemo

+main() : void

asks

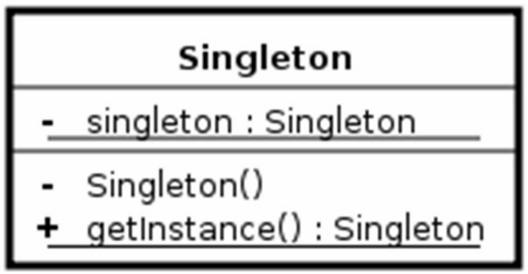
returns

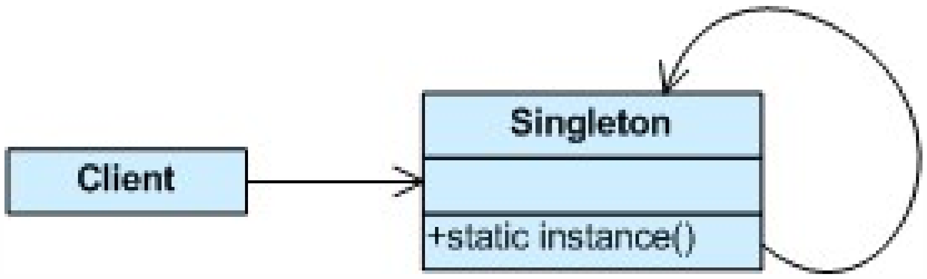
SingleObject

-instance: SingleObject

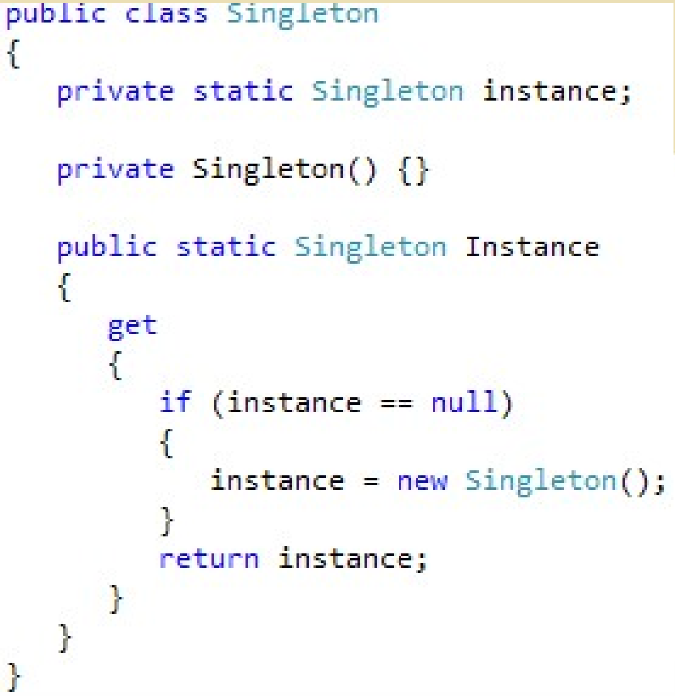
-SingleObject ()
+getInstance():SingleObject
+showMessage():void

Singleton Class Diagram

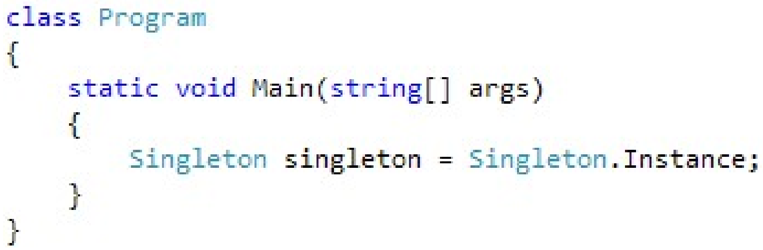




Implement Singleton in .NET (GoF way)



```
public class Singleton
{
   private static Singleton instance;

   private Singleton() {}

   public static Singleton Instance
   {
      get
      {
         if (instance == null)
         {
            instance = new Singleton();
         }
         return instance;
      }
   }
}
```



```
class Program
{
   static void Main(string[] args)
   {
      Singleton singleton = Singleton.Instance;
   }
}
```

- Advantages:

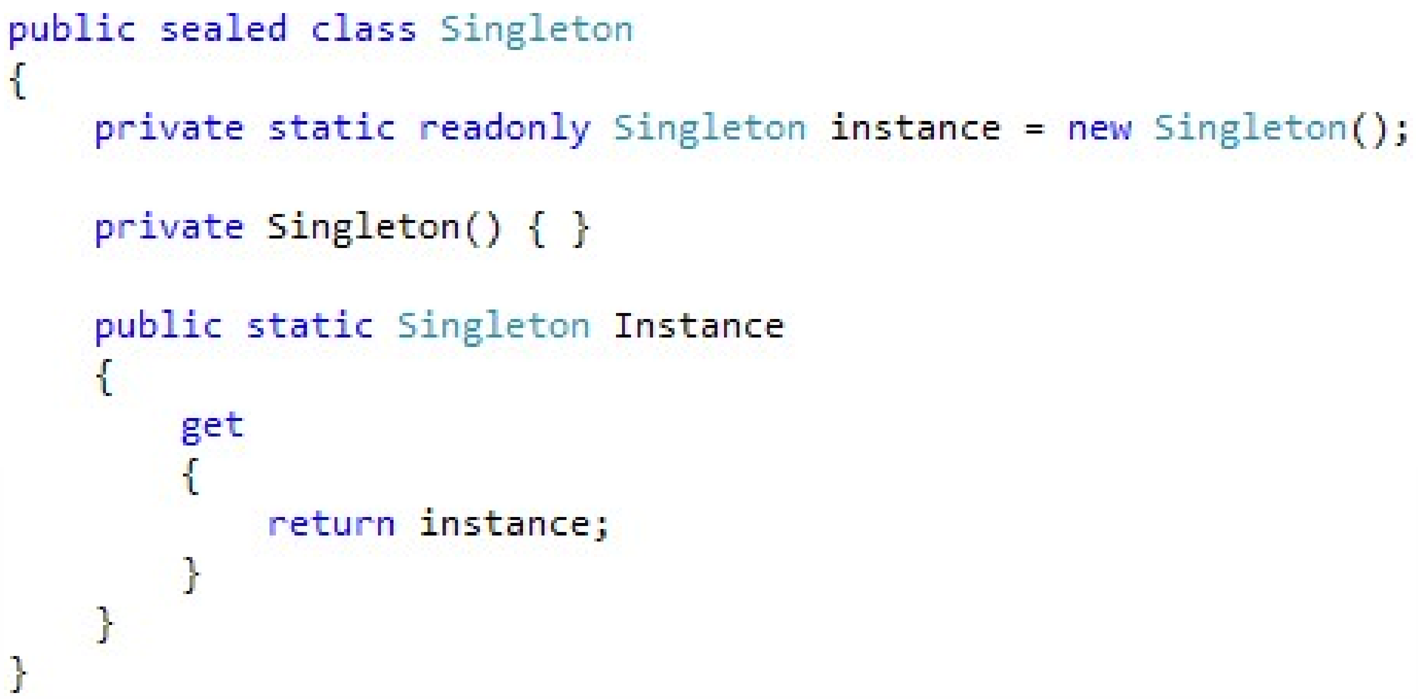
Because the instance is created inside the Instance property method, the class can exercise additional functionality.

The instantiation is not performed until an object asks for an instance; this approach is referred to as lazy instantiation. Lazy instantiation avoids instantiating unnecessary singletons when the application starts.

- Disadvantages:

Not safe for multithreaded environments. If separate threads of execution enter the Instance property method at the same time, more that one instance of the Singleton object may be created.

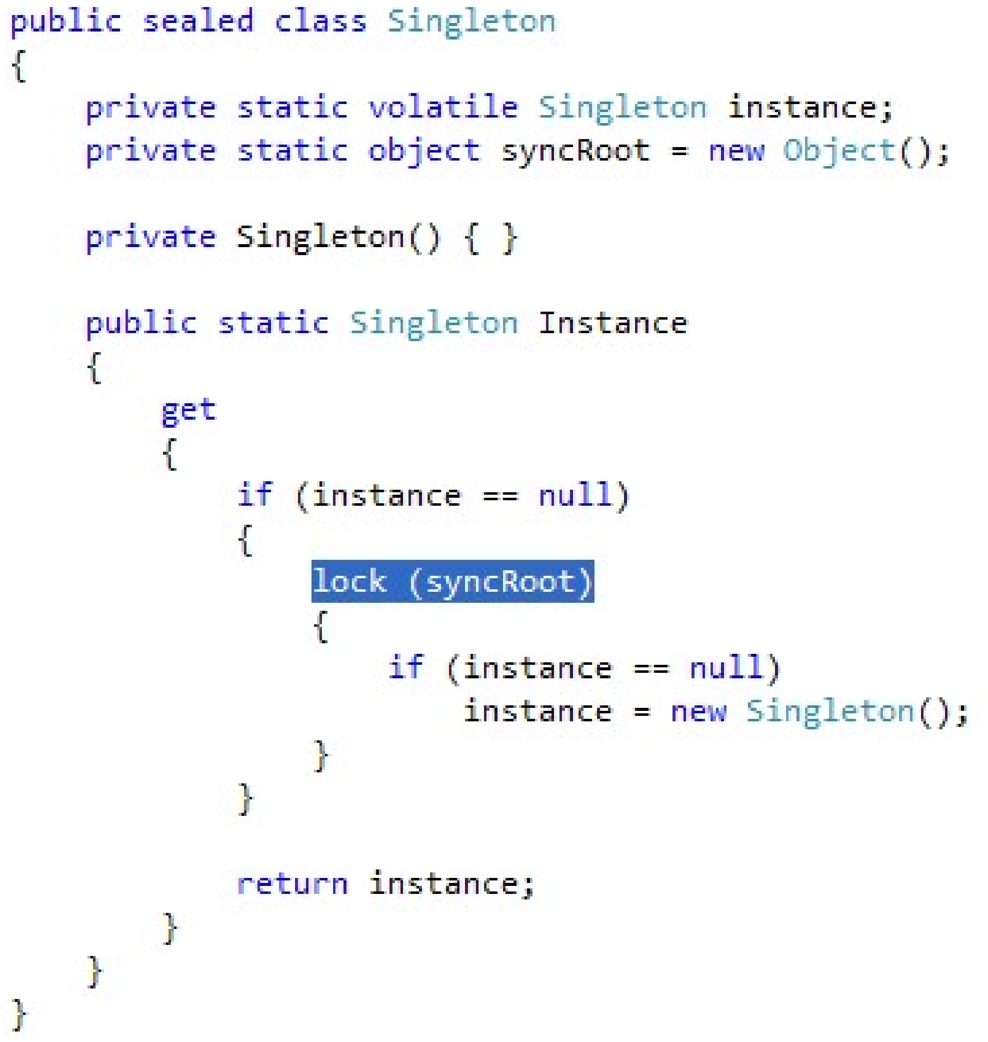
Thread Safe Singleton in .NET (using Static)



```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }
}
```

- In this strategy, the instance is created the first time any member of the class is referenced.
- In addition, the variable is marked **readonly.**

Multithreaded Singleton in .NET



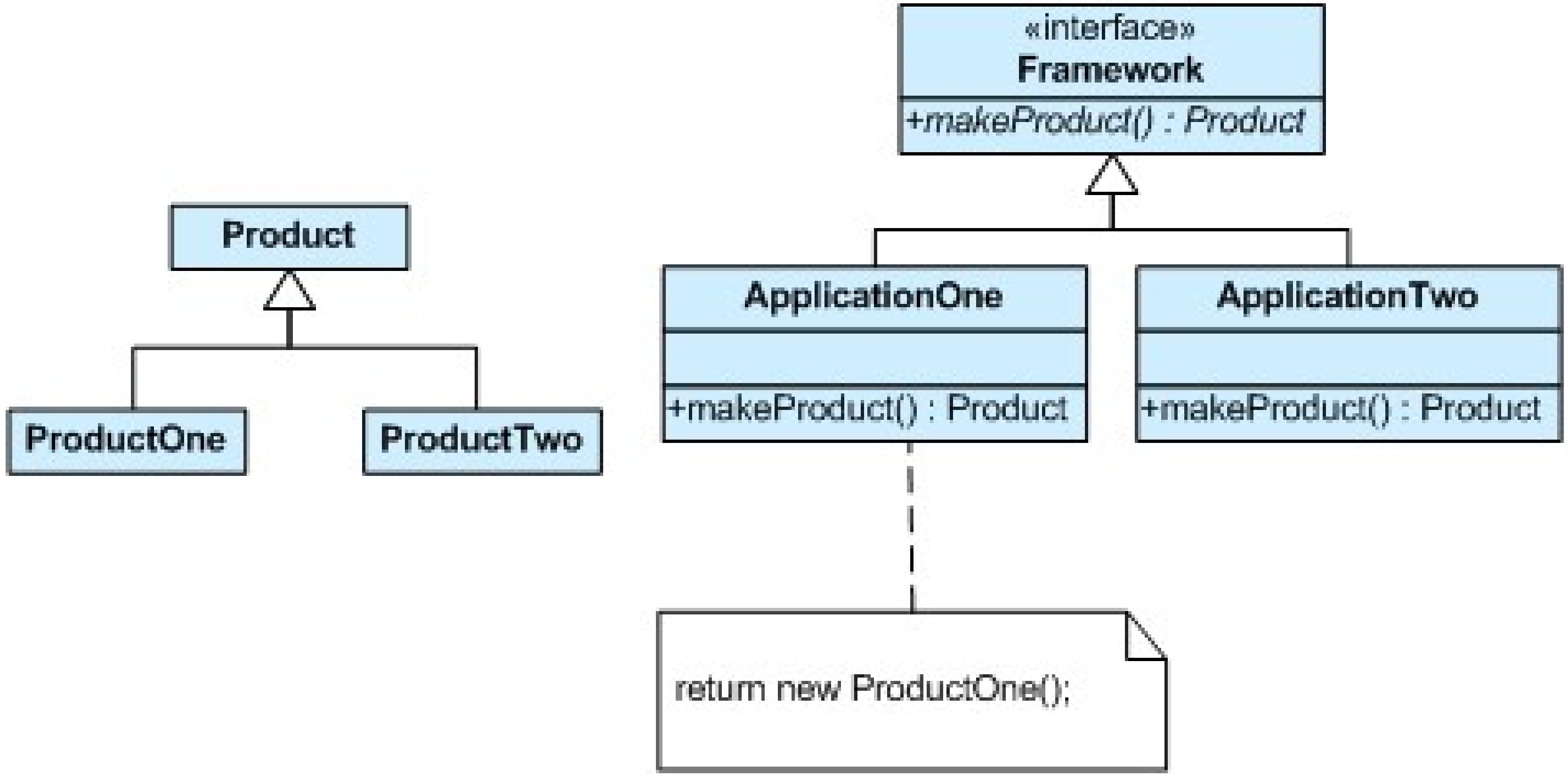
```
public sealed class Singleton
{
    private static volatile Singleton instance;
    private static object syncRoot = new Object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncRoot)
                {
                    if (instance == null)
                        instance = new Singleton();
                }
            }

            return instance;
        }
    }
}
```

- In some cases we cannot rely on the CLR to ensure thread safety.
- **Double-Check Locking** idiom.
- Variable is declared **volatile**.
- Uses a **syncRoot** instance to lock on.

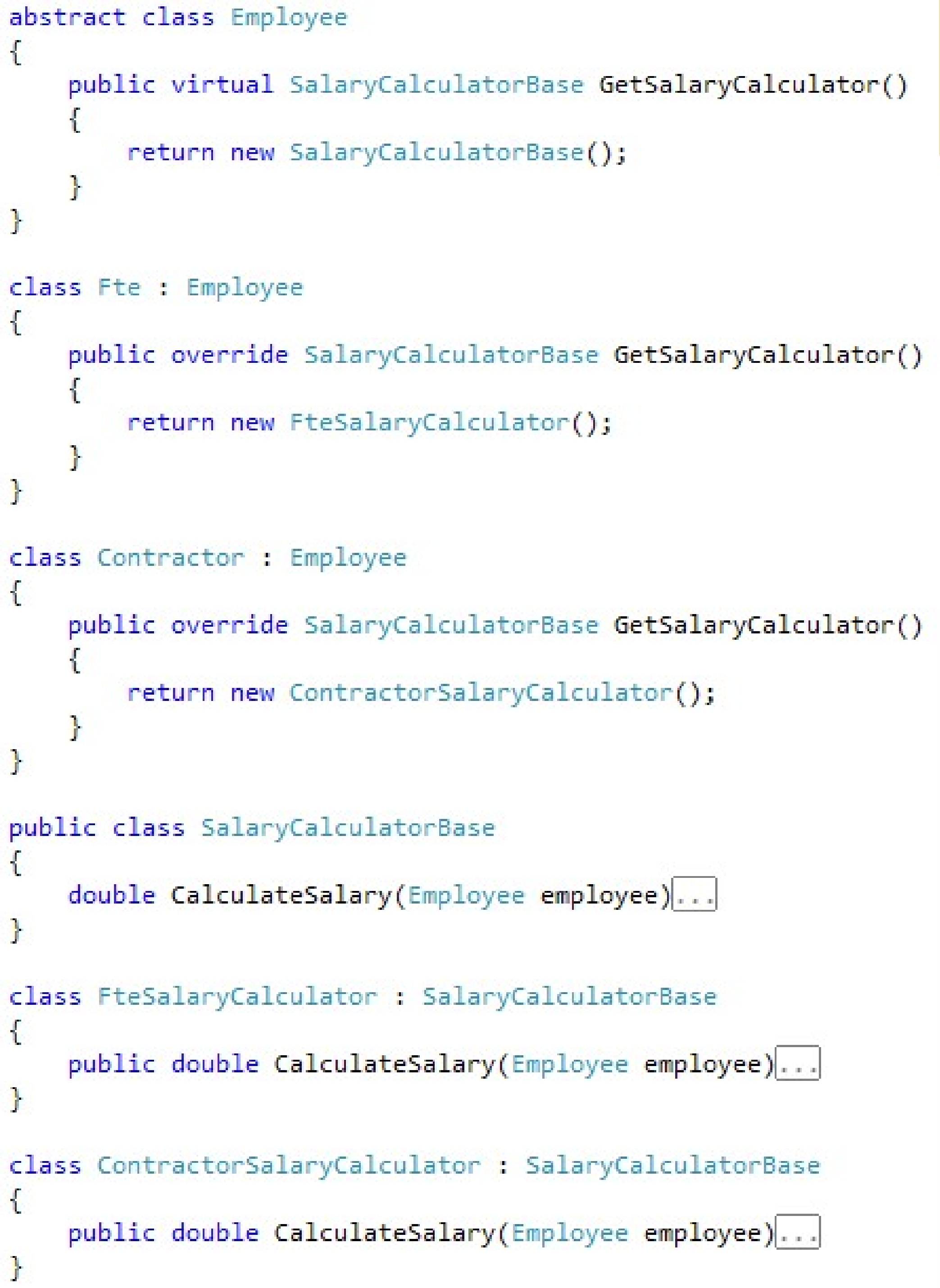
- Double-check locking approach solves thread concurrency problems while avoiding an exclusive lock in every call to the **Instance** property method. Also allows you to delay instantiation until the object is first accessed.

Factory Method

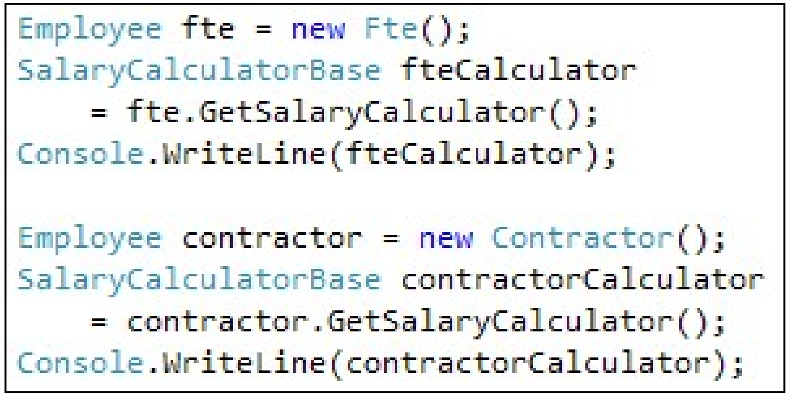
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.



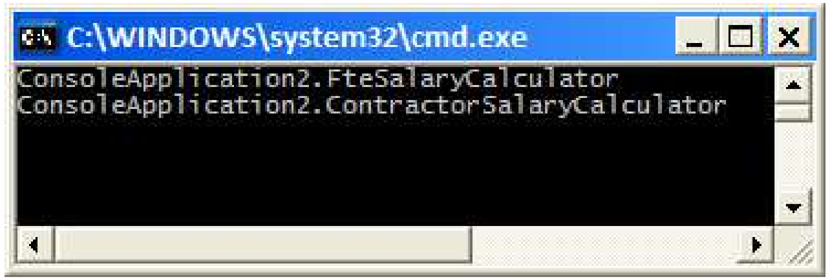
Factory Method in .NET



```
abstract class Employee
{
    public virtual SalaryCalculatorBase GetSalaryCalculator()
    {
        return new SalaryCalculatorBase();
    }
}

class Fte : Employee
{
    public override SalaryCalculatorBase GetSalaryCalculator()
    {
        return new FteSalaryCalculator();
    }
}

class Contractor : Employee
{
    public override SalaryCalculatorBase GetSalaryCalculator()
    {
        return new ContractorSalaryCalculator();
    }
}

public class SalaryCalculatorBase
{
    double CalculateSalary(Employee employee)...
}

class FteSalaryCalculator : SalaryCalculatorBase
{
    public double CalculateSalary(Employee employee)...
}

class ContractorSalaryCalculator : SalaryCalculatorBase
{
    public double CalculateSalary(Employee employee)...
}
```



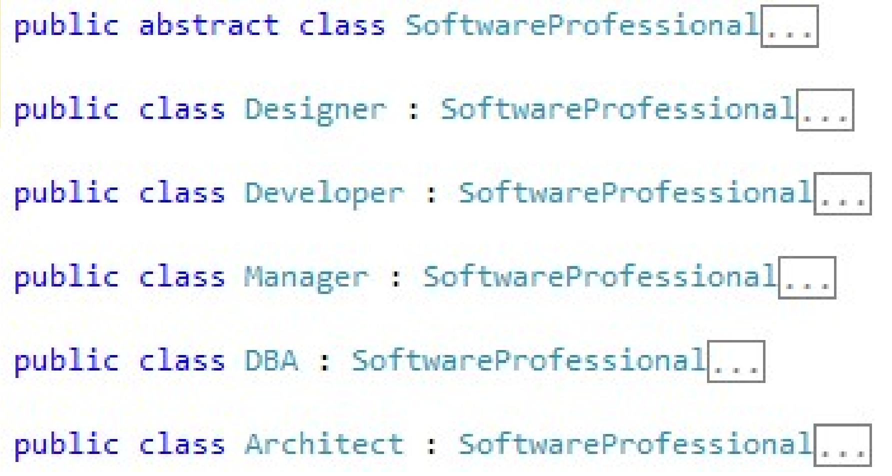
```
Employee fte = new Fte();
SalaryCalculatorBase fteCalculator
    = fte.GetSalaryCalculator();
Console.WriteLine(fteCalculator);

Employee contractor = new Contractor();
SalaryCalculatorBase contractorCalculator
    = contractor.GetSalaryCalculator();
Console.WriteLine(contractorCalculator);
```



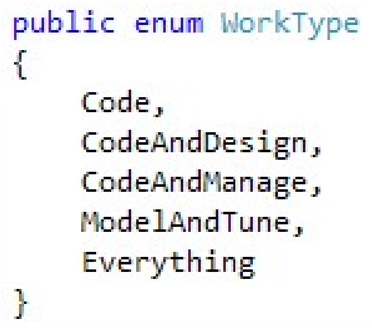
```
ConsoleApplication2.FteSalaryCalculator
ConsoleApplication2.ContractorSalaryCalculator
```

Abstract Factory Pattern

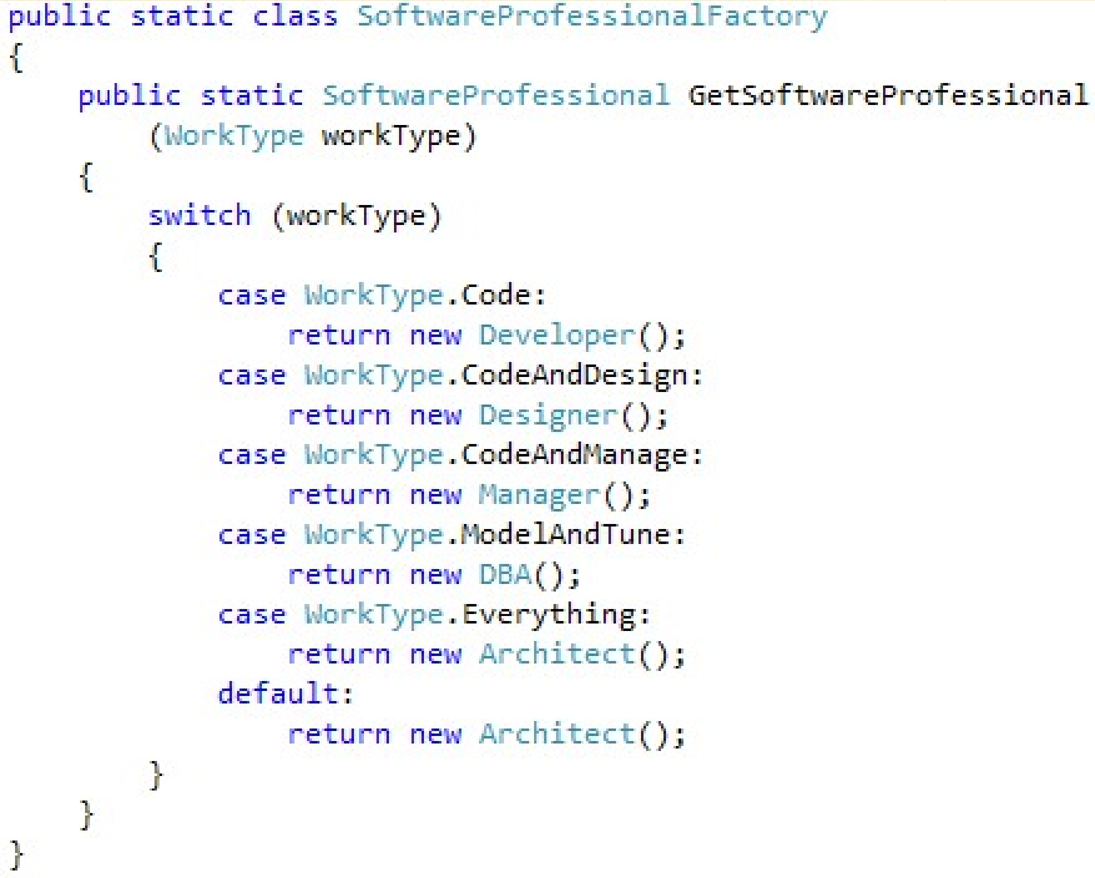
- Provides a way to encapsulate a group of individual factories that have a common theme.
- The client software creates a concrete implementation of the abstract factory.
- Client does not know (or care) which concrete objects it gets from each of these internal factories



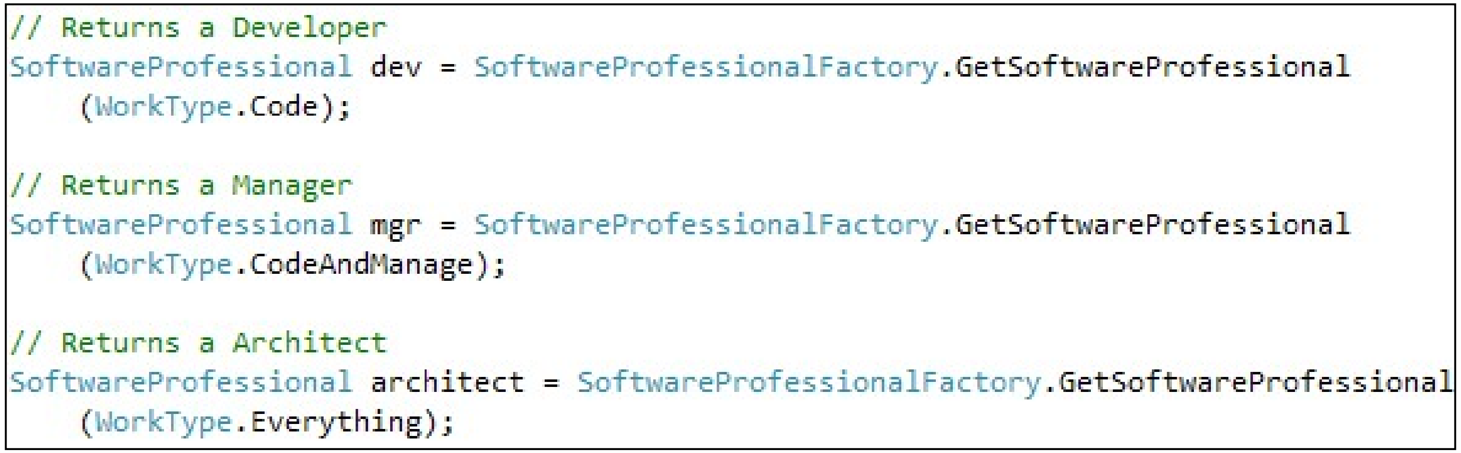
```
public abstract class SoftwareProfessional...

public class Designer : SoftwareProfessional...

public class Developer : SoftwareProfessional...

public class Manager : SoftwareProfessional...

public class DBA : SoftwareProfessional...

public class Architect : SoftwareProfessional...
```



```
public enum WorkType
{
    Code,
    CodeAndDesign,
    CodeAndManage,
    ModelAndTune,
    Everything
}
```



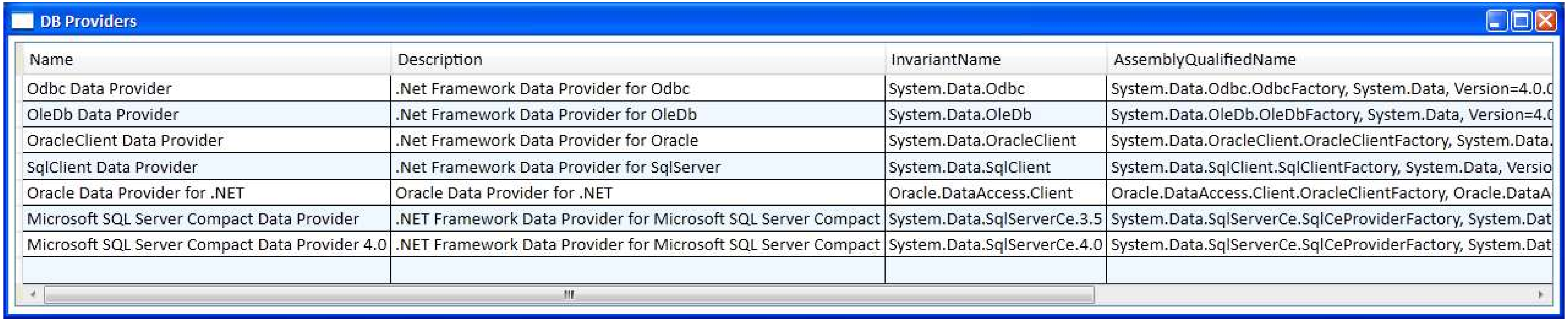
```
public static class SoftwareProfessionalFactory
{
    public static SoftwareProfessional GetSoftwareProfessional
        (WorkType workType)
    {
        switch (workType)
        {
            case WorkType.Code:
                return new Developer();
            case WorkType.CodeAndDesign:
                return new Designer();
            case WorkType.CodeAndManage:
                return new Manager();
            case WorkType.ModelAndTune:
                return new DBA();
            case WorkType.Everything:
                return new Architect();
            default:
                return new Architect();
        }
    }
}
```



```
// Returns a Developer
SoftwareProfessional dev = SoftwareProfessionalFactory.GetSoftwareProfessional
    (WorkType.Code);

// Returns a Manager
SoftwareProfessional mgr = SoftwareProfessionalFactory.GetSoftwareProfessional
    (WorkType.CodeAndManage);

// Returns a Architect
SoftwareProfessional architect = SoftwareProfessionalFactory.GetSoftwareProfessional
    (WorkType.Everything);
```

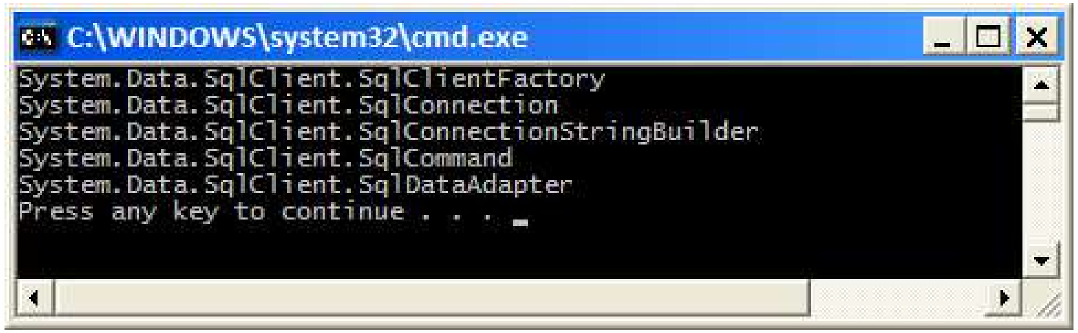
Factory in .NET: DbProviderFactory

```csharp
private void Form1_Load(object sender, EventArgs e)
{
    dgvDBProviders.DataSource = DbProviderFactories.GetFactoryClasses();
}
```



Name	Description	InvariantName	AssemblyQualifiedName
Odbc Data Provider	.Net Framework Data Provider for Odbc	System.Data.Odbc	System.Data.Odbc.OdbcFactory, System.Data, Version=4.0.0
OleDb Data Provider	.Net Framework Data Provider for OleDb	System.Data.OleDb	System.Data.OleDb.OleDbFactory, System.Data, Version=4.0
OracleClient Data Provider	.Net Framework Data Provider for Oracle	System.Data.OracleClient	System.Data.OracleClient.OracleClientFactory, System.Data.
SqlClient Data Provider	.Net Framework Data Provider for SqlServer	System.Data.SqlClient	System.Data.SqlClient.SqlClientFactory, System.Data, Versio
Oracle Data Provider for .NET	Oracle Data Provider for .NET	Oracle.DataAccess.Client	Oracle.DataAccess.Client.OracleClientFactory, Oracle.DataA
Microsoft SQL Server Compact Data Provider	.NET Framework Data Provider for Microsoft SQL Server Compact	System.Data.SqlServerCe.3.5	System.Data.SqlServerCe.SqlCeProviderFactory, System.Dat
Microsoft SQL Server Compact Data Provider 4.0	.NET Framework Data Provider for Microsoft SQL Server Compact	System.Data.SqlServerCe.4.0	System.Data.SqlServerCe.SqlCeProviderFactory, System.Dat

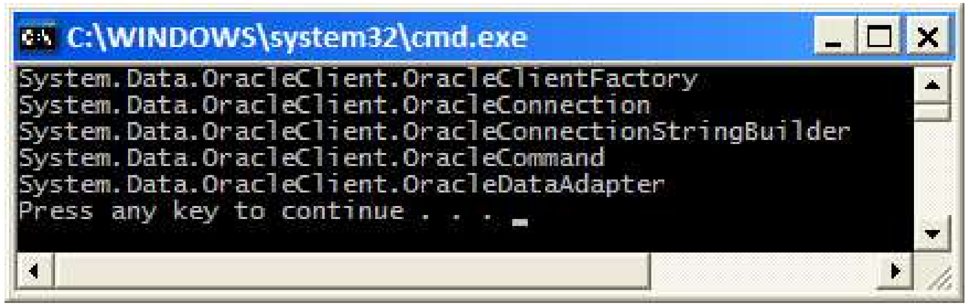
```csharp
SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
builder.DataSource = ".";
builder.IntegratedSecurity = true;
builder.InitialCatalog = "Employee";

DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.SqlClient");
Console.WriteLine(factory.GetType());

DbConnection conn = factory.CreateConnection();
Console.WriteLine(conn.GetType());

DbConnectionStringBuilder connBuilder = factory.CreateConnectionStringBuilder();
Console.WriteLine(connBuilder.GetType());

DbCommand command = factory.CreateCommand();
Console.WriteLine(command.GetType());

DataAdapter adapter = factory.CreateDataAdapter();
Console.WriteLine(adapter.GetType());
```



```
System.Data.SqlClient.SqlClientFactory
System.Data.SqlClient.SqlConnection
System.Data.SqlClient.SqlConnectionStringBuilder
System.Data.SqlClient.SqlCommand
System.Data.SqlClient.SqlDataAdapter
Press any key to continue . . . _
```

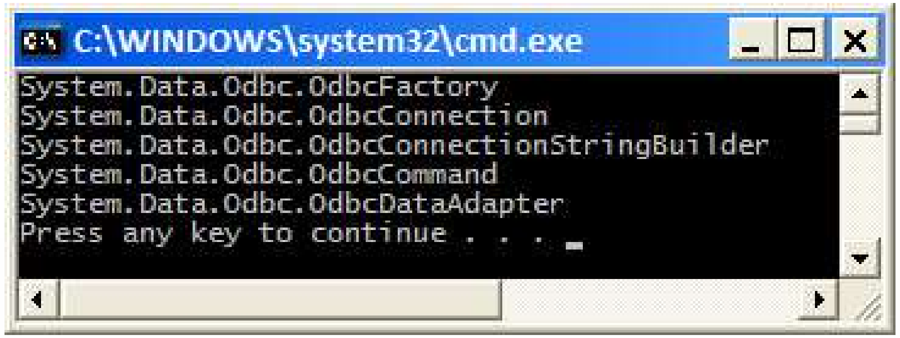
Factory in .NET: DbProviderFactory

```
DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.OracleClient");
Console.WriteLine(factory.GetType());

DbConnection conn = factory.CreateConnection();
Console.WriteLine(conn.GetType());

DbConnectionStringBuilder connBuilder = factory.CreateConnectionStringBuilder();
Console.WriteLine(connBuilder.GetType());

DbCommand command = factory.CreateCommand();
Console.WriteLine(command.GetType());

DataAdapter adapter = factory.CreateDataAdapter();
Console.WriteLine(adapter.GetType());
```



```
System.Data.OracleClient.OracleClientFactory
System.Data.OracleClient.OracleConnection
System.Data.OracleClient.OracleConnectionStringBuilder
System.Data.OracleClient.OracleCommand
System.Data.OracleClient.OracleDataAdapter
Press any key to continue . . .
```

```
DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.Odbc");
Console.WriteLine(factory.GetType());

DbConnection conn = factory.CreateConnection();
Console.WriteLine(conn.GetType());

DbConnectionStringBuilder connBuilder = factory.CreateConnectionStringBuilder();
Console.WriteLine(connBuilder.GetType());

DbCommand command = factory.CreateCommand();
Console.WriteLine(command.GetType());

DataAdapter adapter = factory.CreateDataAdapter();
Console.WriteLine(adapter.GetType());
```

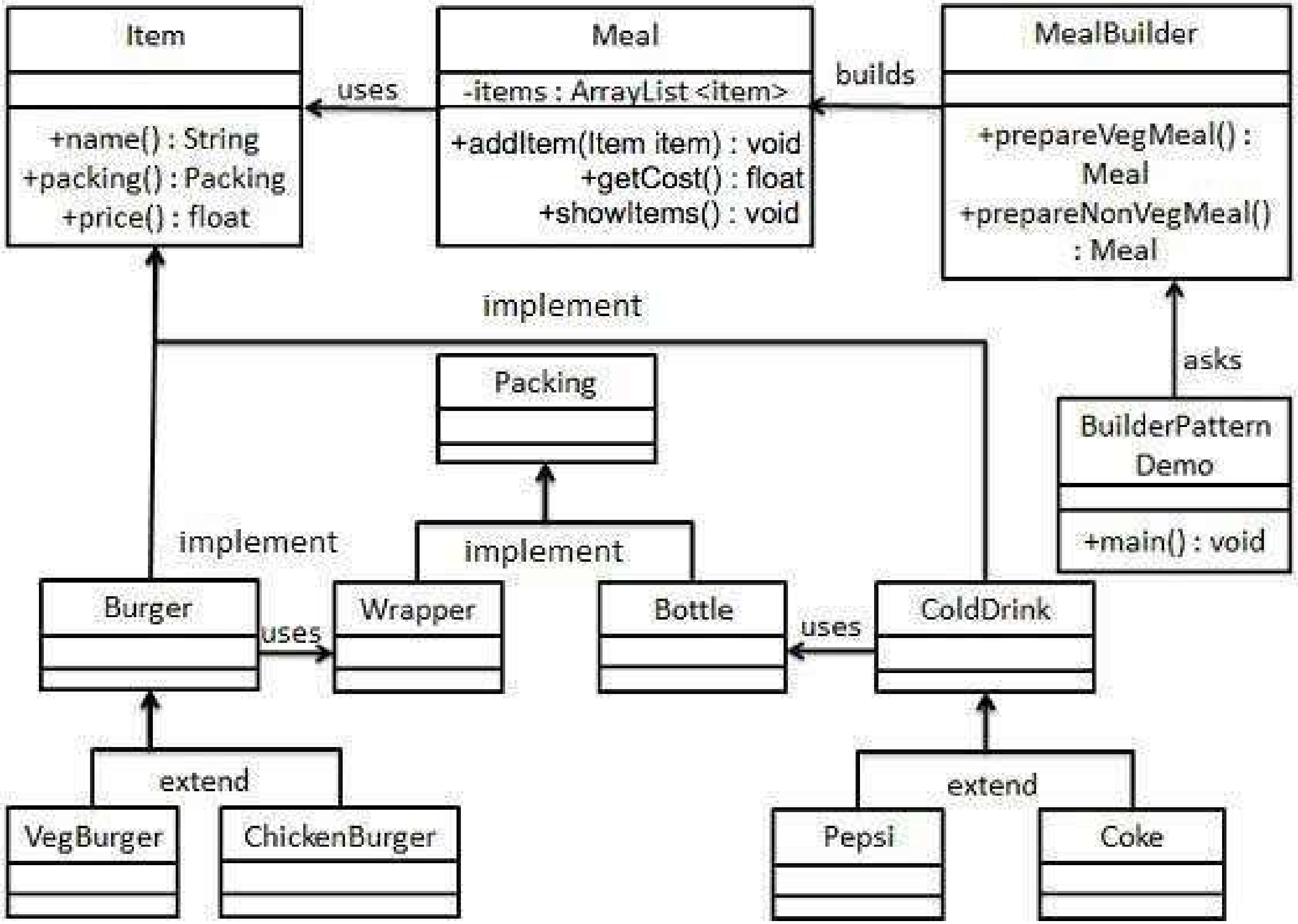


```
System.Data.Odbc.OdbcFactory
System.Data.Odbc.OdbcConnection
System.Data.Odbc.OdbcConnectionStringBuilder
System.Data.Odbc.OdbcCommand
System.Data.Odbc.OdbcDataAdapter
Press any key to continue . . .
```

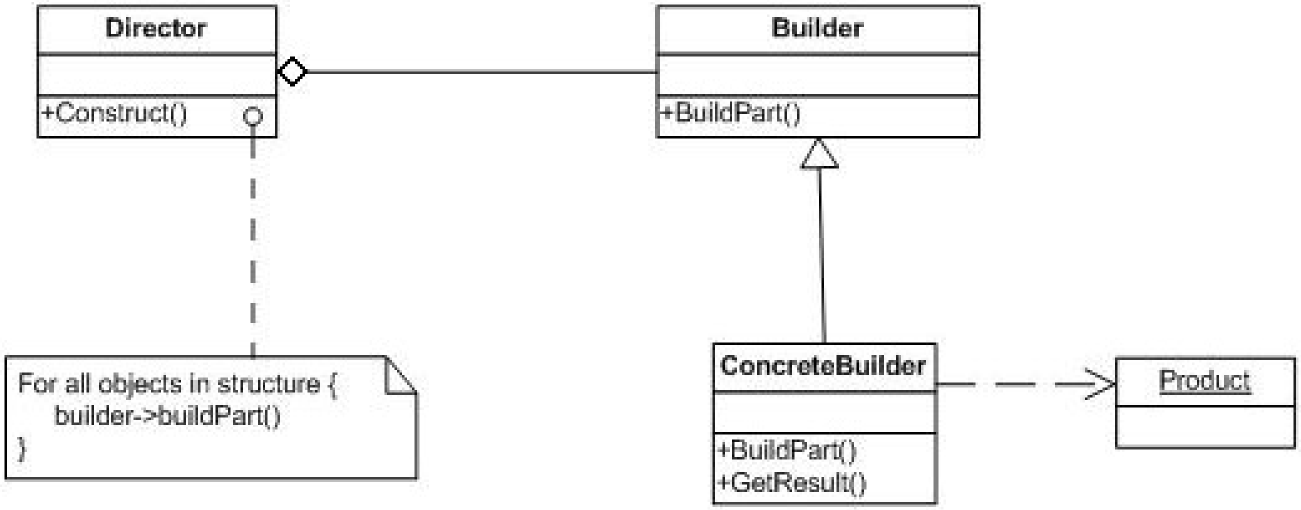
Builder Design Pattern

- Builder focuses on constructing a complex object step by step.
- Builder often builds a Composite.
- Often, designs start out using Factory Method and evolve toward Abstract Factory, Prototype, or Builder.
- Sometimes creational patterns are complementary.

Builder Design Pattern



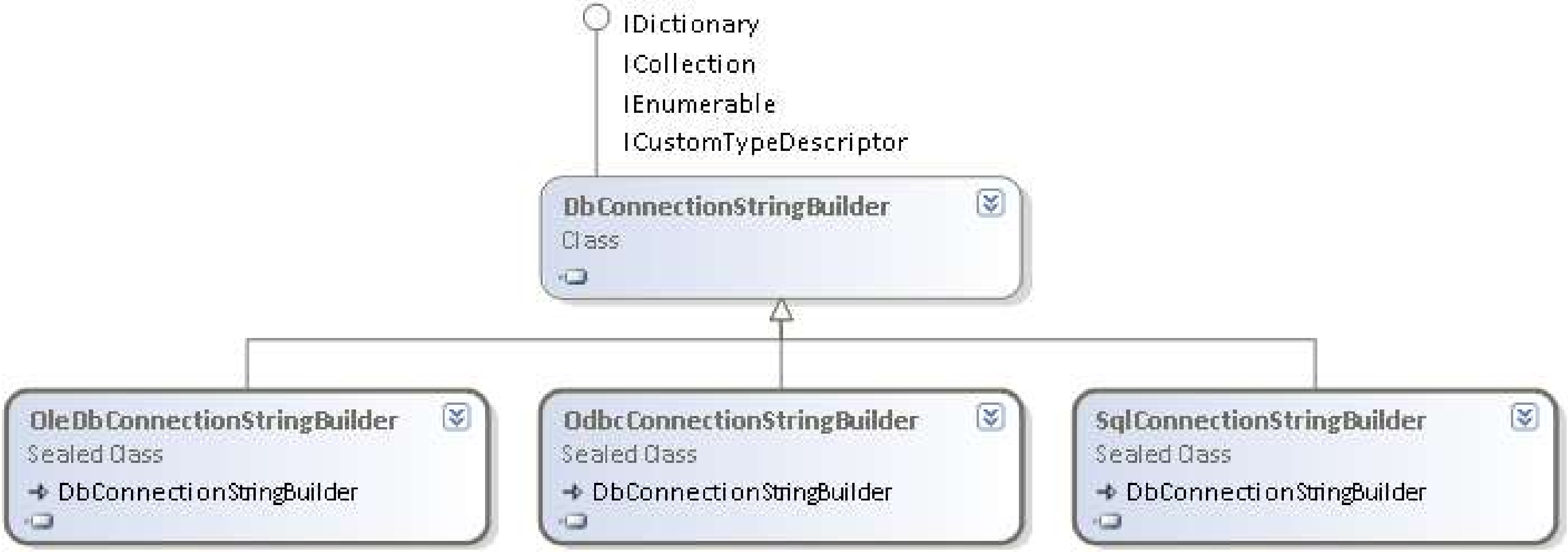
Builder Pattern



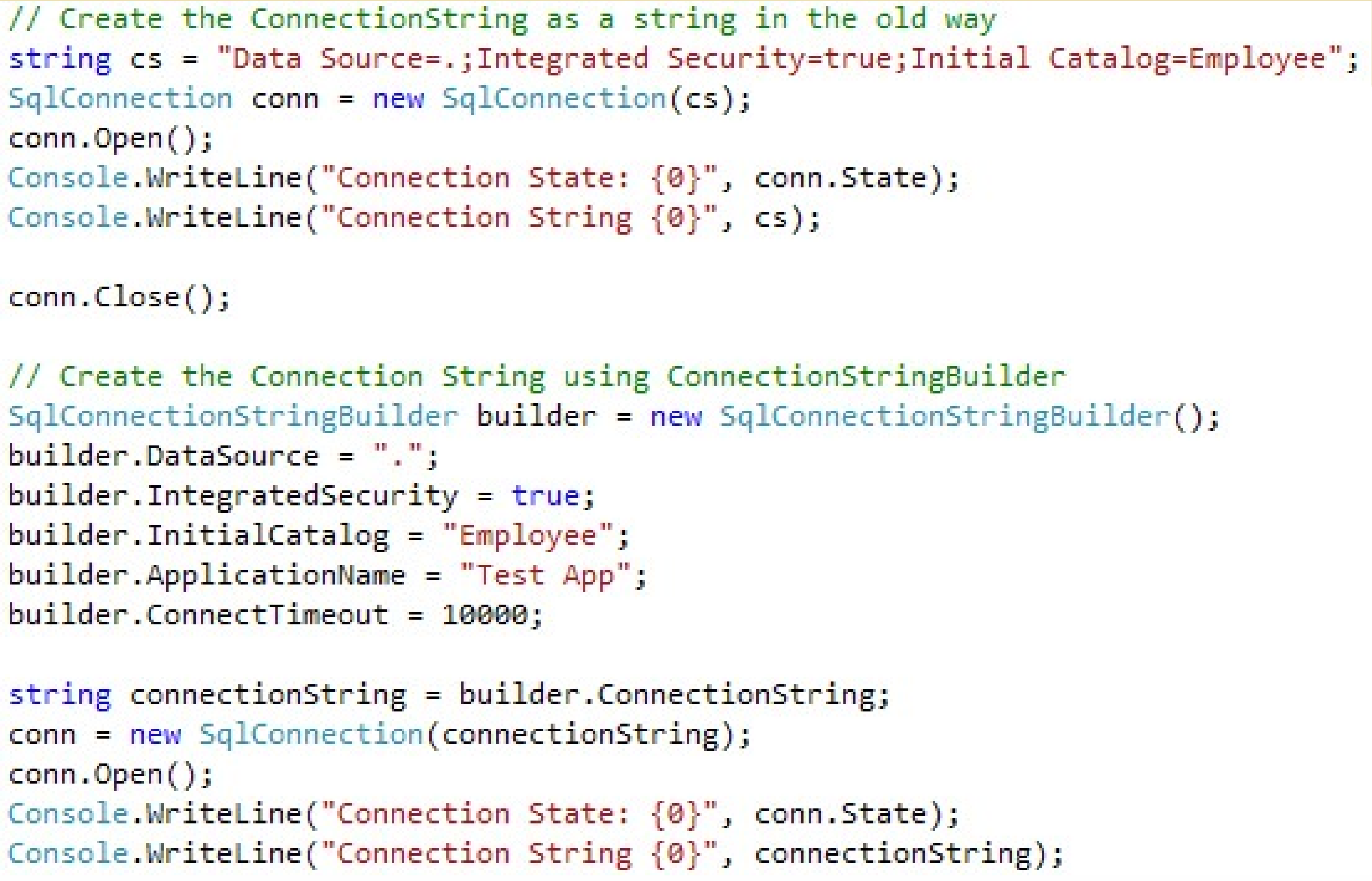
- Separate construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.
- Difference Between Builder and Factory

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects - simple or complex.

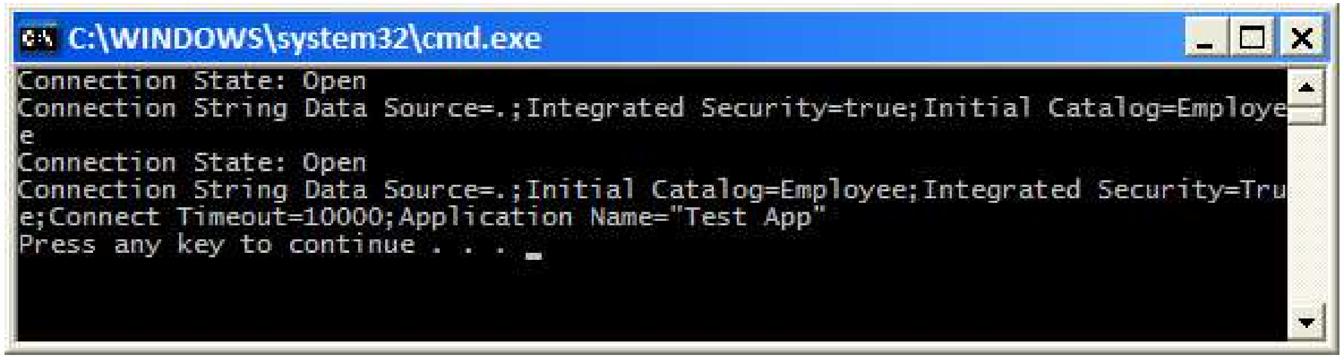
Builder Pattern in .NET BCL


IDictionary
ICollection
IEnumerable
ICustomTypeDescriptor
DbConnectionStringBuilder
Class
OleDbConnectionStringBuilder
Sealed Class
DbConnectionStringBuilder
OdbcConnectionStringBuilder
Sealed Class
DbConnectionStringBuilder
SqlConnectionStringBuilder
Sealed Class
DbConnectionStringBuilder

Builder: SqlConnectionStringBuilder in .NET



```
// Create the ConnectionString as a string in the old way
string cs = "Data Source=.;Integrated Security=true;Initial Catalog=Employee";
SqlConnection conn = new SqlConnection(cs);
conn.Open();
Console.WriteLine("Connection State: {0}", conn.State);
Console.WriteLine("Connection String {0}", cs);

conn.Close();

// Create the Connection String using ConnectionStringBuilder
SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
builder.DataSource = ".";
builder.IntegratedSecurity = true;
builder.InitialCatalog = "Employee";
builder.ApplicationName = "Test App";
builder.ConnectTimeout = 10000;

string connectionString = builder.ConnectionString;
conn = new SqlConnection(connectionString);
conn.Open();
Console.WriteLine("Connection State: {0}", conn.State);
Console.WriteLine("Connection String {0}", connectionString);
```

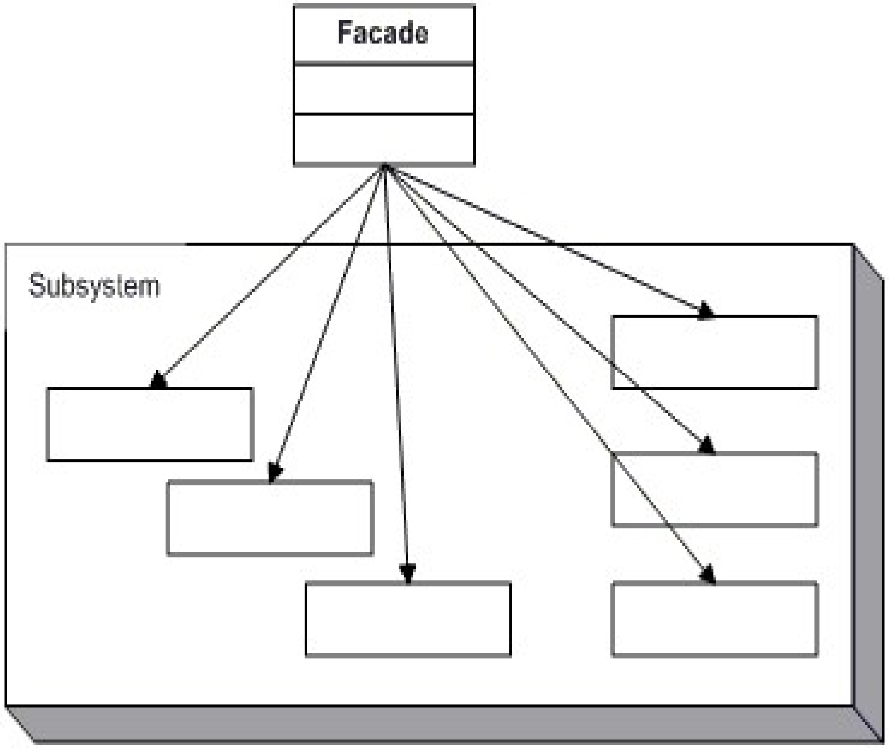


```
Connection State: Open
Connection String Data Source=.;Integrated Security=true;Initial Catalog=Employe
e
Connection State: Open
Connection String Data Source=.;Initial Catalog=Employee;Integrated Security=Tru
e;Connect Timeout=10000;Application Name="Test App"
Press any key to continue . . . _
```

Façade

- Facade or Façade is generally one side of the exterior of a building, especially the front, but also sometimes the sides and rear.
- The word comes from the French language, literally meaning "frontage" or "face".

Façade Pattern



- A facade is an object that provides a simplified interface to a larger body of code, such as a class library.
- A facade can:

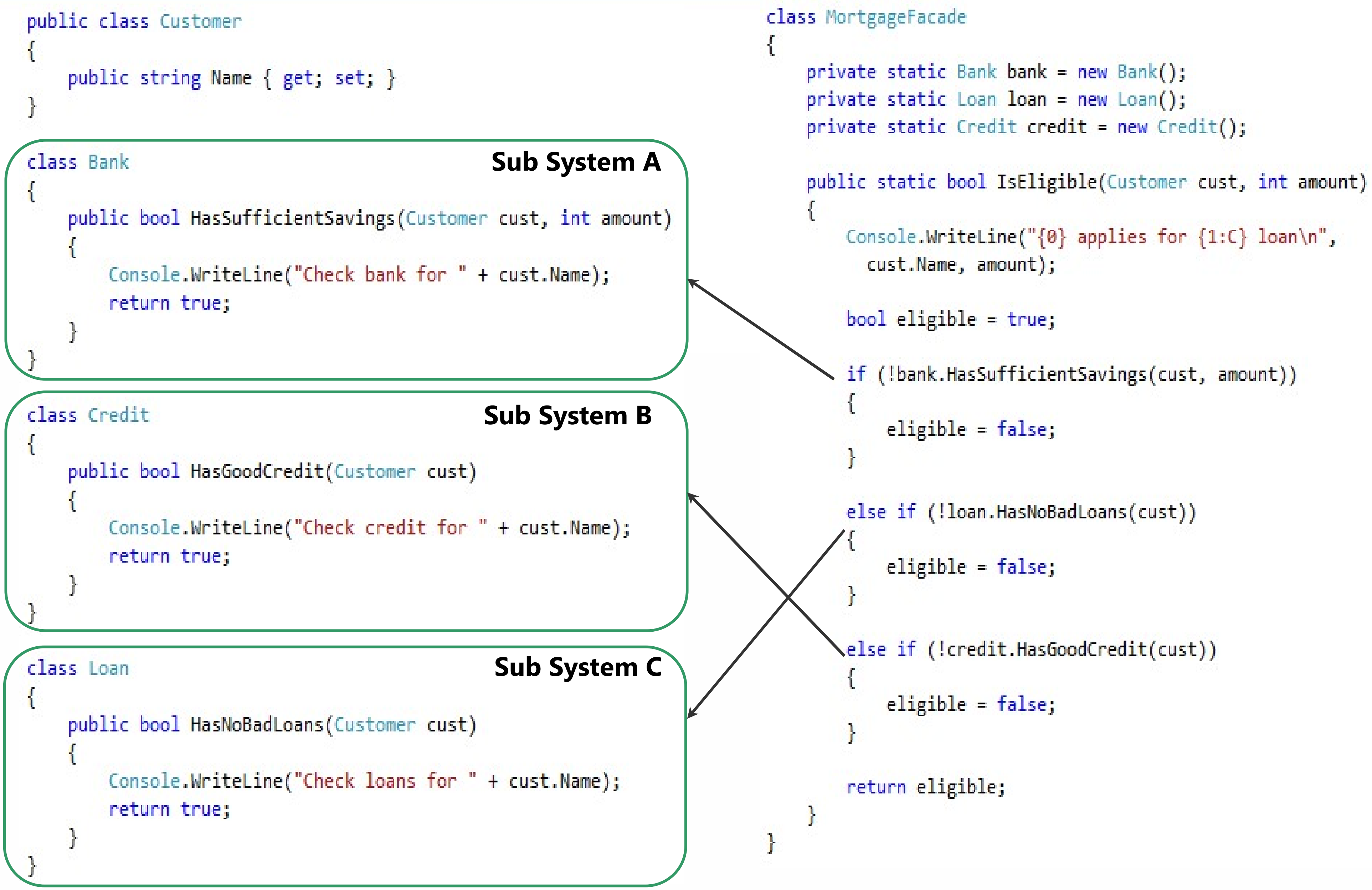
Make a software library easier to use, understand and test

Make code that uses the library more readable

Reduce dependencies of outside code on the inner workings of a library

Wrap a poorly-designed collection of APIs with a single well-designed API

Implementing Façade in .NET



```
public class Customer
{
    public string Name { get; set; }
}
```

Sub System A

```
class Bank
{
    public bool HasSufficientSavings(Customer cust, int amount)
    {
        Console.WriteLine("Check bank for " + cust.Name);
        return true;
    }
}
```

Sub System B

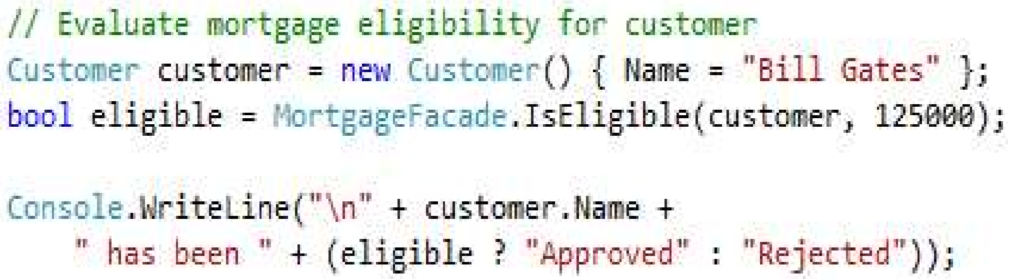
```
class Credit
{
    public bool HasGoodCredit(Customer cust)
    {
        Console.WriteLine("Check credit for " + cust.Name);
        return true;
    }
}
```

Sub System C

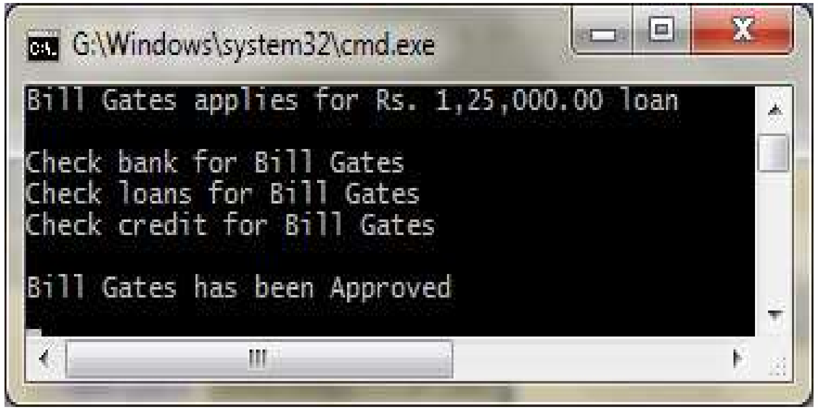
```
class Loan
{
    public bool HasNoBadLoans(Customer cust)
    {
        Console.WriteLine("Check loans for " + cust.Name);
        return true;
    }
}
```

```
class MortgageFacade
{
    private static Bank bank = new Bank();
    private static Loan loan = new Loan();
    private static Credit credit = new Credit();

    public static bool IsEligible(Customer cust, int amount)
    {
        Console.WriteLine("{0} applies for {1:C} loan\n",
          cust.Name, amount);

        bool eligible = true;

        if (!bank.HasSufficientSavings(cust, amount))
        {
            eligible = false;
        }

        else if (!loan.HasNoBadLoans(cust))
        {
            eligible = false;
        }

        else if (!credit.HasGoodCredit(cust))
        {
            eligible = false;
        }

        return eligible;
    }
}
```

Implementing Façade in .NET (Cont'd)



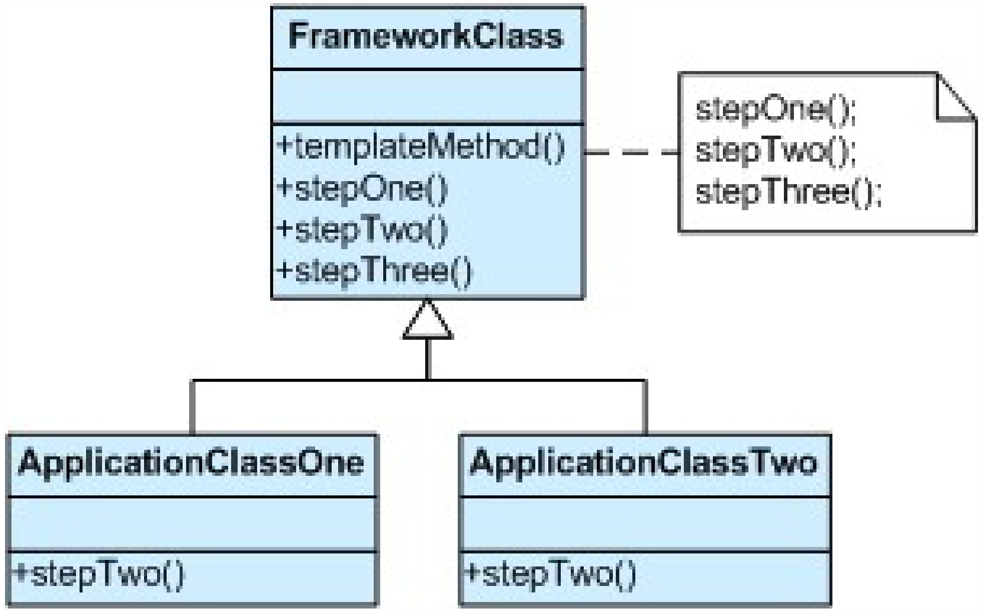
```
// Evaluate mortgage eligibility for customer
Customer customer = new Customer() { Name = "Bill Gates" };
bool eligible = MortgageFacade.IsEligible(customer, 125000);

Console.WriteLine("\n" + customer.Name +
    " has been " + (eligible ? "Approved" : "Rejected"));
```



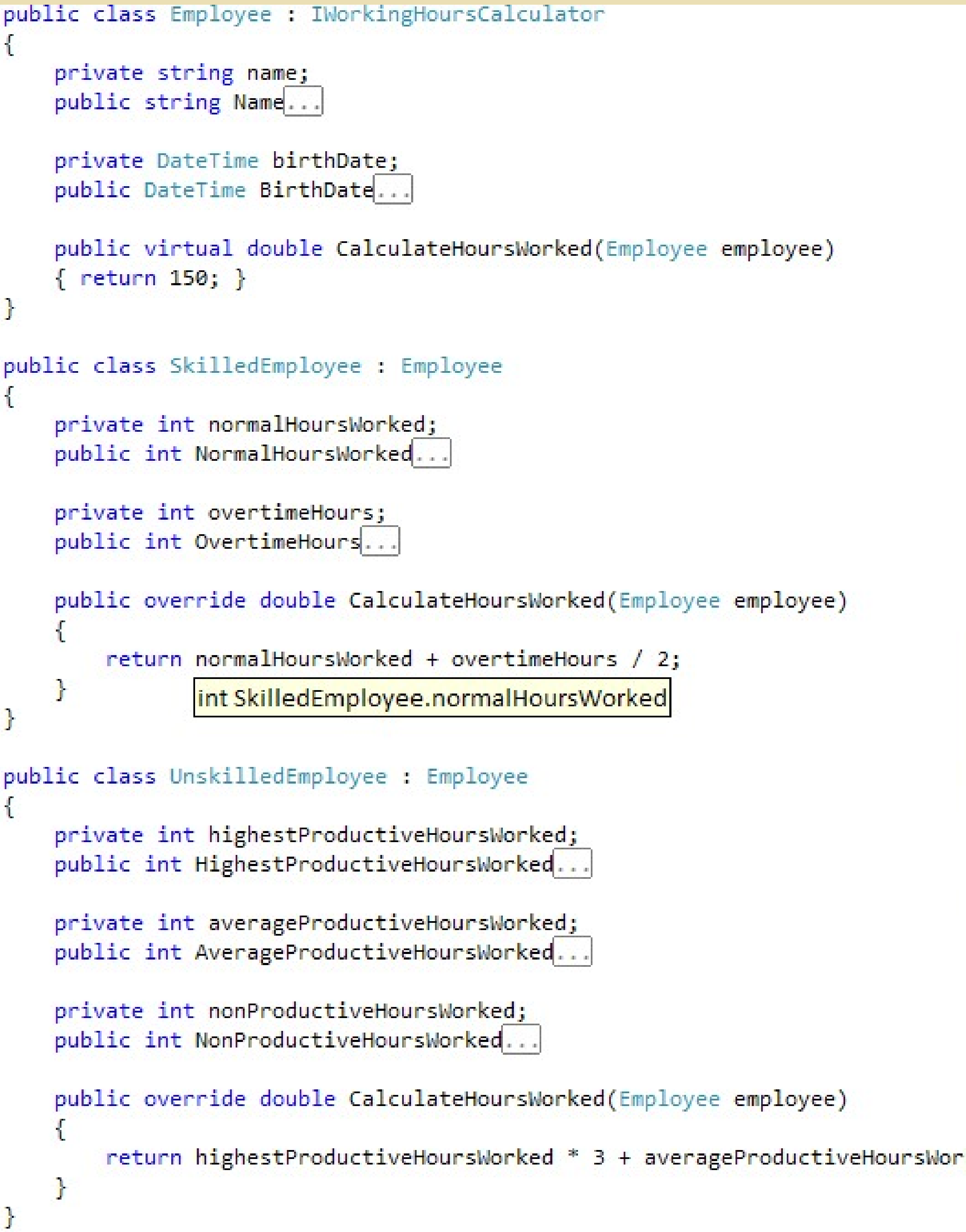
```
Bill Gates applies for Rs. 1,25,000.00 loan

Check bank for Bill Gates
Check loans for Bill Gates
Check credit for Bill Gates

Bill Gates has been Approved
```

Template Method Pattern

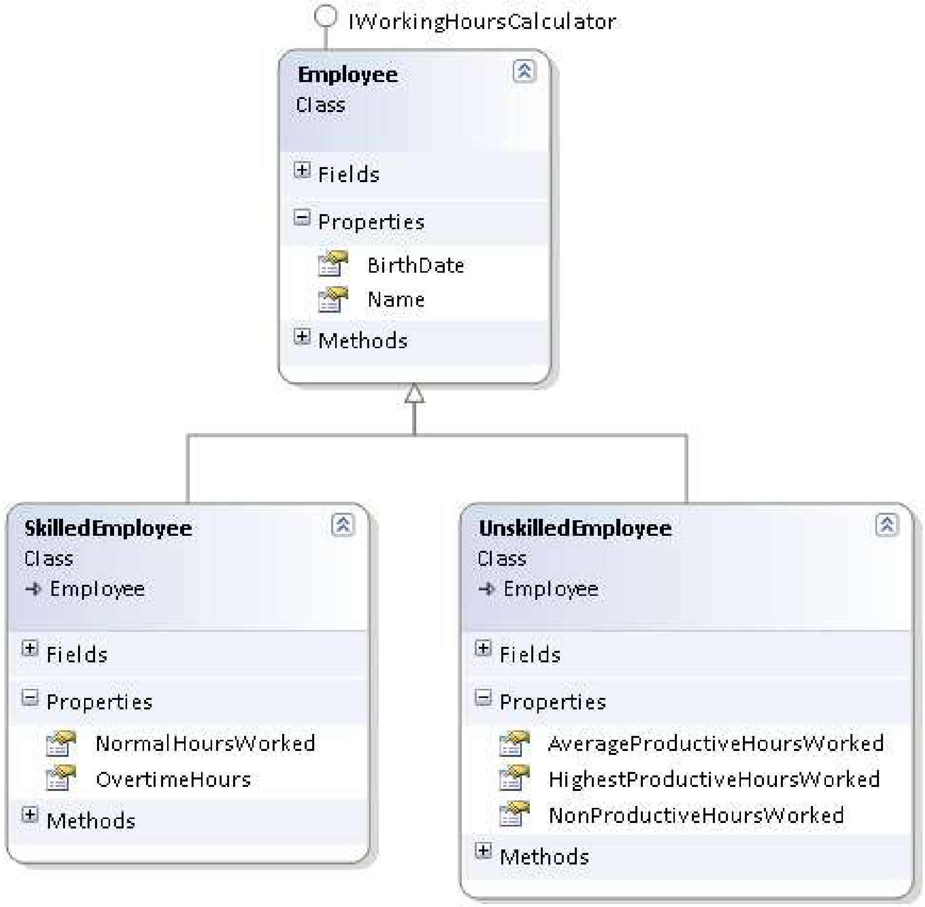
- Define the skeleton of an algorithm in an operation
- Template Method lets subclasses redefine certain steps of an algorithm.
- Base class declares algorithm ‘placeholders’, and derived classes implement the placeholders.



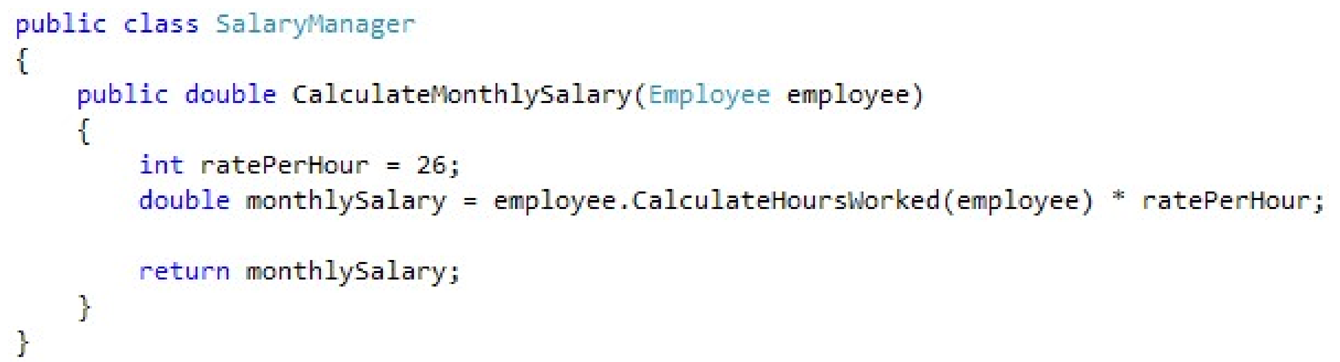
Implement Template Method Pattern in .NET



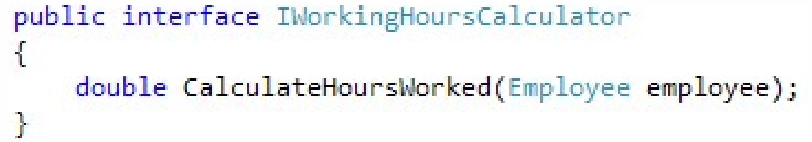
```
public class Employee : IWorkingHoursCalculator
{
    private string name;
    public string Name...

    private DateTime birthDate;
    public DateTime BirthDate...

    public virtual double CalculateHoursWorked(Employee employee)
    { return 150; }
}

public class SkilledEmployee : Employee
{
    private int normalHoursWorked;
    public int NormalHoursWorked...

    private int overtimeHours;
    public int OvertimeHours...

    public override double CalculateHoursWorked(Employee employee)
    {
        return normalHoursWorked + overtimeHours / 2;
    }       int SkilledEmployee.normalHoursWorked
}

public class UnskilledEmployee : Employee
{
    private int highestProductiveHoursWorked;
    public int HighestProductiveHoursWorked...

    private int averageProductiveHoursWorked;
    public int AverageProductiveHoursWorked...

    private int nonProductiveHoursWorked;
    public int NonProductiveHoursWorked...

    public override double CalculateHoursWorked(Employee employee)
    {
        return highestProductiveHoursWorked * 3 + averageProductiveHoursWorked + nonProductiveHoursWorked / 4;
    }
}
```



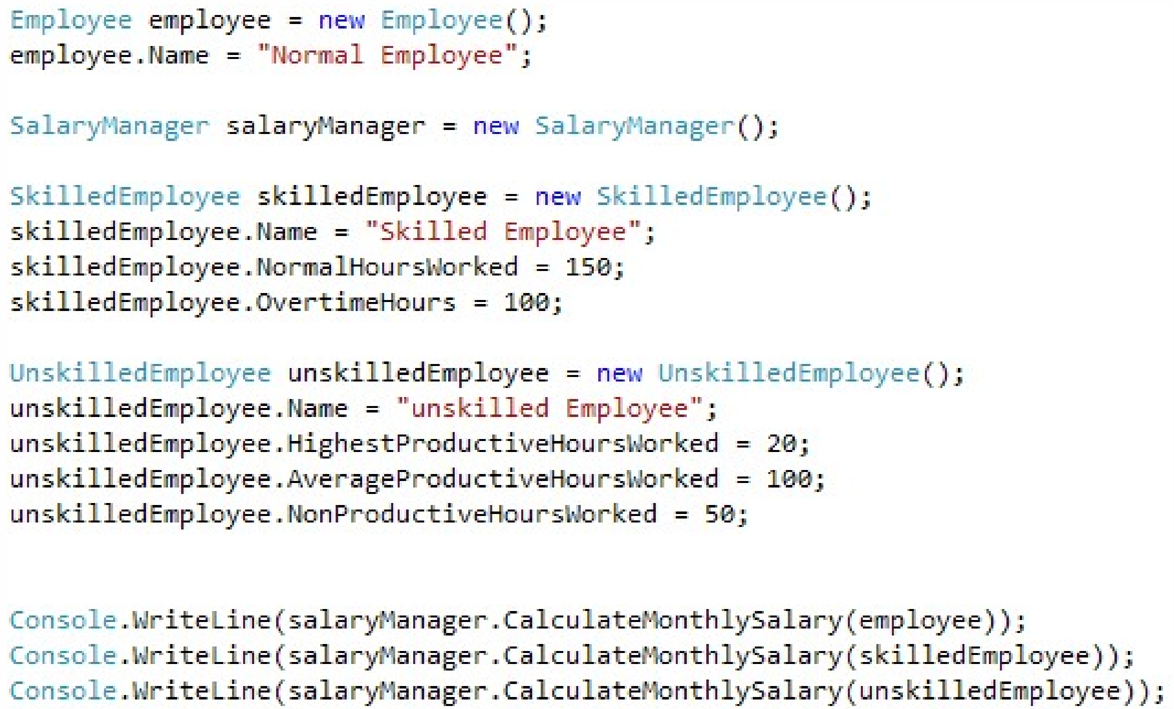
Implement Template Method Pattern in .NET (Cont'd)



```
public class SalaryManager
{
    public double CalculateMonthlySalary(Employee employee)
    {
        int ratePerHour = 26;
        double monthlySalary = employee.CalculateHoursWorked(employee) * ratePerHour;

        return monthlySalary;
    }
}
```

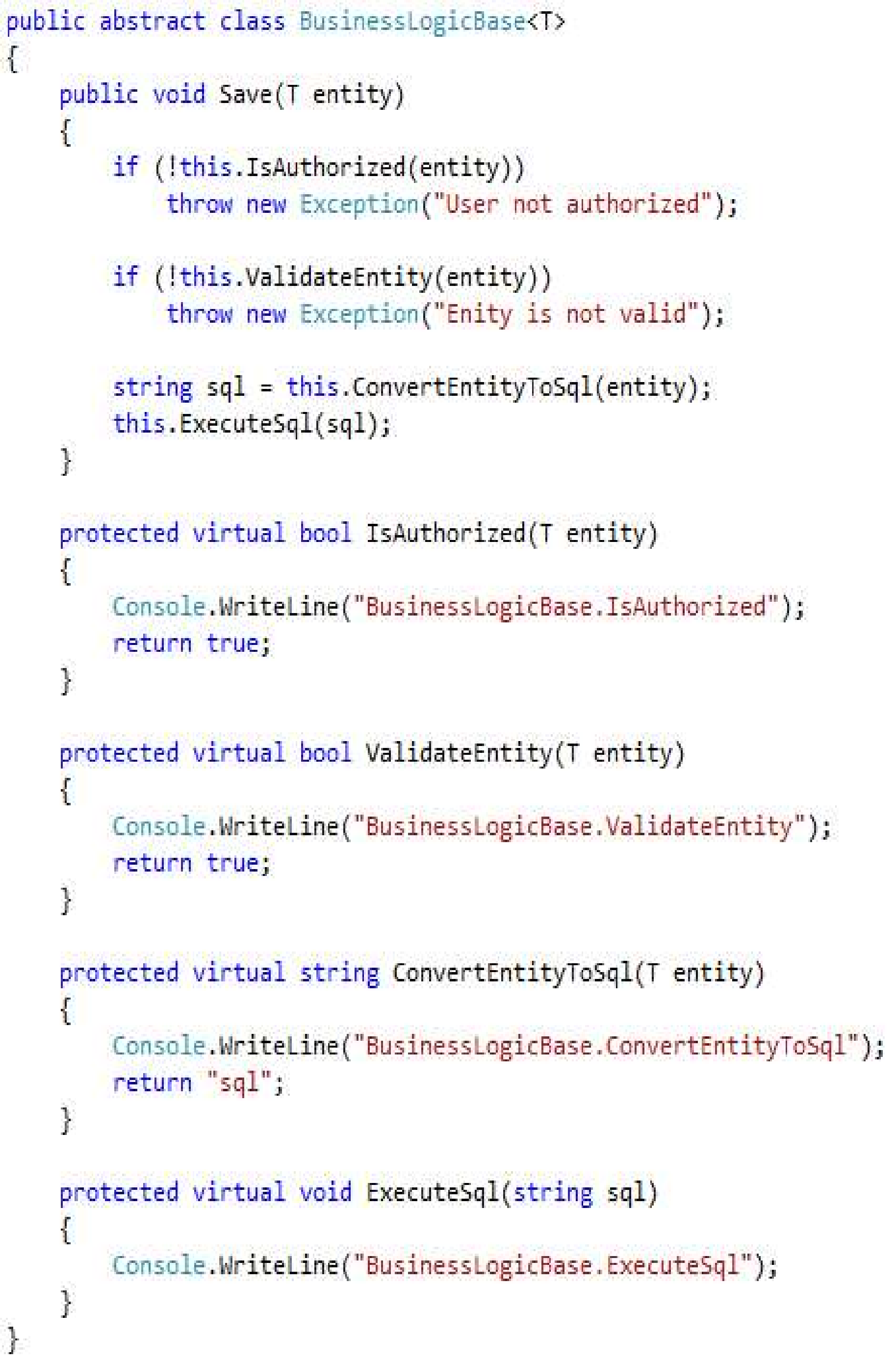


```
public interface IWorkingHoursCalculator
{
    double CalculateHoursWorked(Employee employee);
}
```



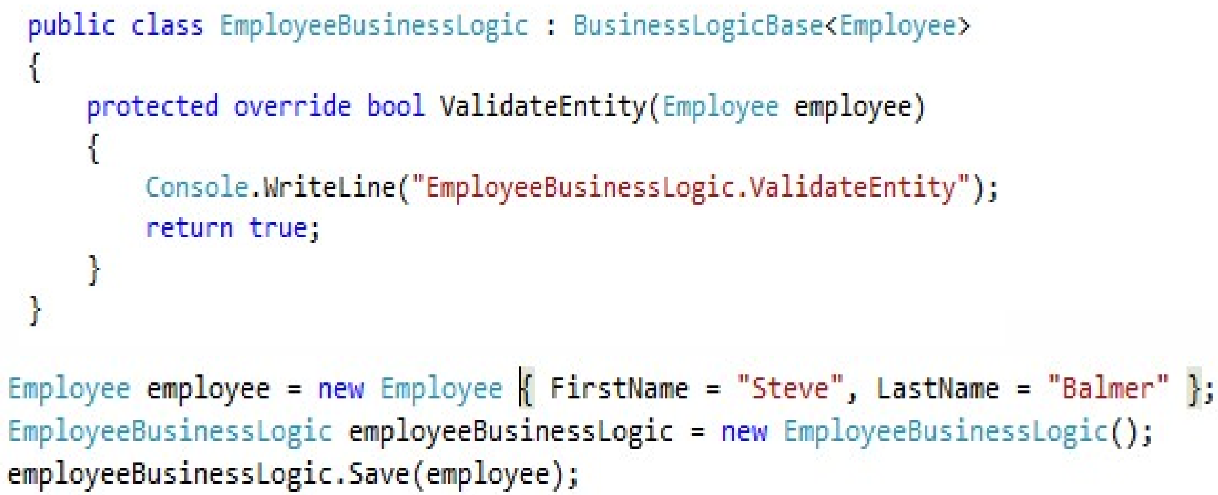
```
Employee employee = new Employee();
employee.Name = "Normal Employee";

SalaryManager salaryManager = new SalaryManager();

SkilledEmployee skilledEmployee = new SkilledEmployee();
skilledEmployee.Name = "Skilled Employee";
skilledEmployee.NormalHoursWorked = 150;
skilledEmployee.OvertimeHours = 100;

UnskilledEmployee unskilledEmployee = new UnskilledEmployee();
unskilledEmployee.Name = "unskilled Employee";
unskilledEmployee.HighestProductiveHoursWorked = 20;
unskilledEmployee.AverageProductiveHoursWorked = 100;
unskilledEmployee.NonProductiveHoursWorked = 50;


Console.WriteLine(salaryManager.CalculateMonthlySalary(employee));
Console.WriteLine(salaryManager.CalculateMonthlySalary(skilledEmployee));
Console.WriteLine(salaryManager.CalculateMonthlySalary(unskilledEmployee));
```

Template Method Pattern – Example 2

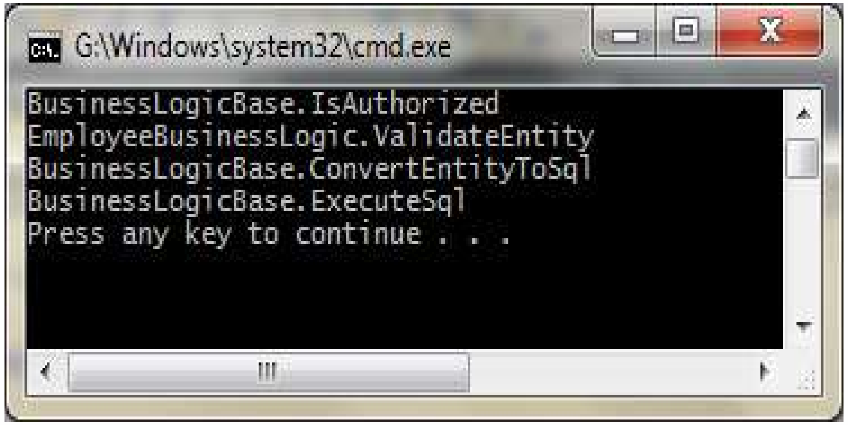


```
public abstract class BusinessLogicBase<T>
{
    public void Save(T entity)
    {
        if (!this.IsAuthorized(entity))
            throw new Exception("User not authorized");

        if (!this.ValidateEntity(entity))
            throw new Exception("Enity is not valid");

        string sql = this.ConvertEntityToSql(entity);
        this.ExecuteSql(sql);
    }

    protected virtual bool IsAuthorized(T entity)
    {
        Console.WriteLine("BusinessLogicBase.IsAuthorized");
        return true;
    }

    protected virtual bool ValidateEntity(T entity)
    {
        Console.WriteLine("BusinessLogicBase.ValidateEntity");
        return true;
    }

    protected virtual string ConvertEntityToSql(T entity)
    {
        Console.WriteLine("BusinessLogicBase.ConvertEntityToSql");
        return "sql";
    }

    protected virtual void ExecuteSql(string sql)
    {
        Console.WriteLine("BusinessLogicBase.ExecuteSql");
    }
}
```

Template Method Pattern – Example 2

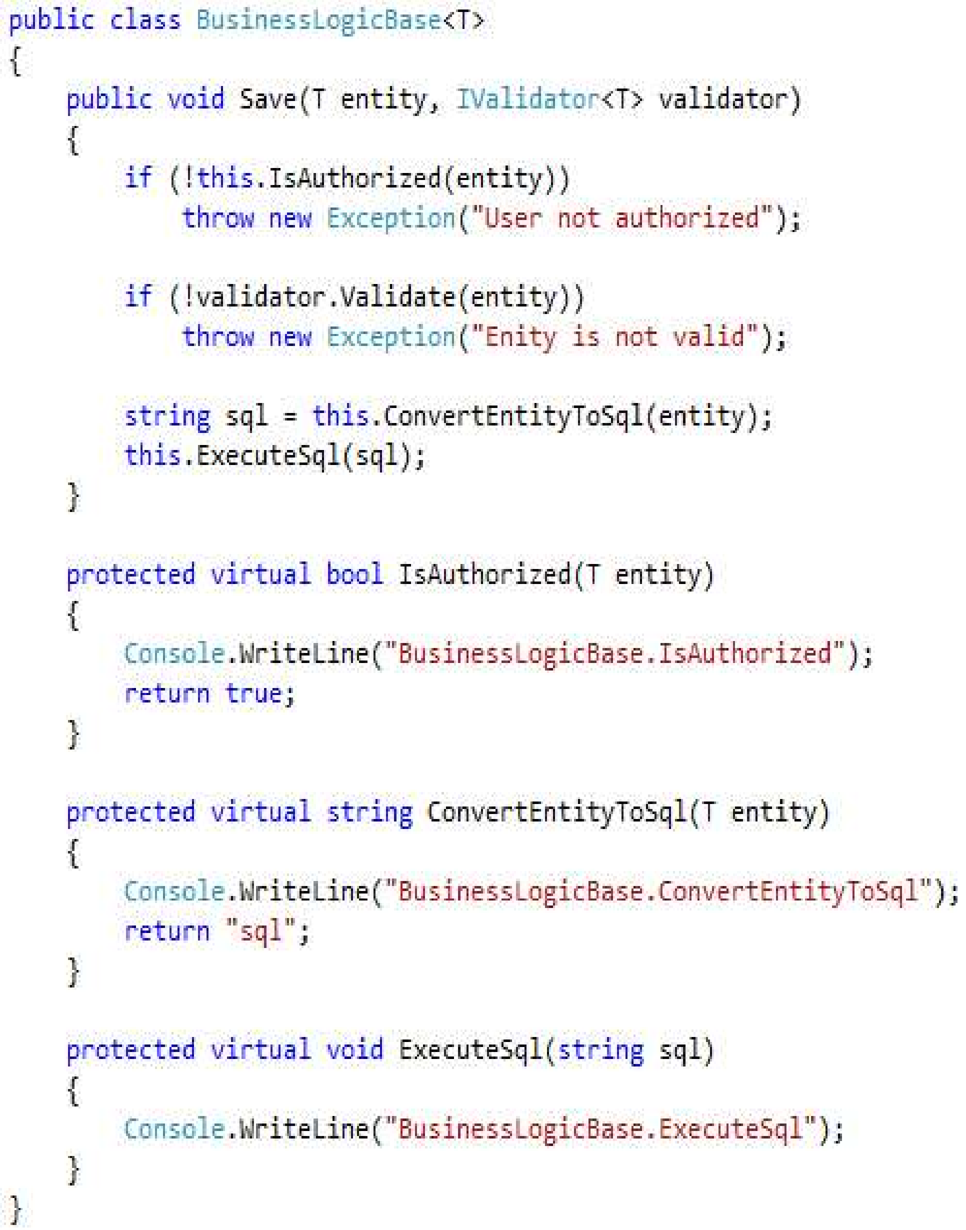


```
public class EmployeeBusinessLogic : BusinessLogicBase<Employee>
{
    protected override bool ValidateEntity(Employee employee)
    {
        Console.WriteLine("EmployeeBusinessLogic.ValidateEntity");
        return true;
    }
}

Employee employee = new Employee { FirstName = "Steve", LastName = "Balmer" };
EmployeeBusinessLogic employeeBusinessLogic = new EmployeeBusinessLogic();
employeeBusinessLogic.Save(employee);
```

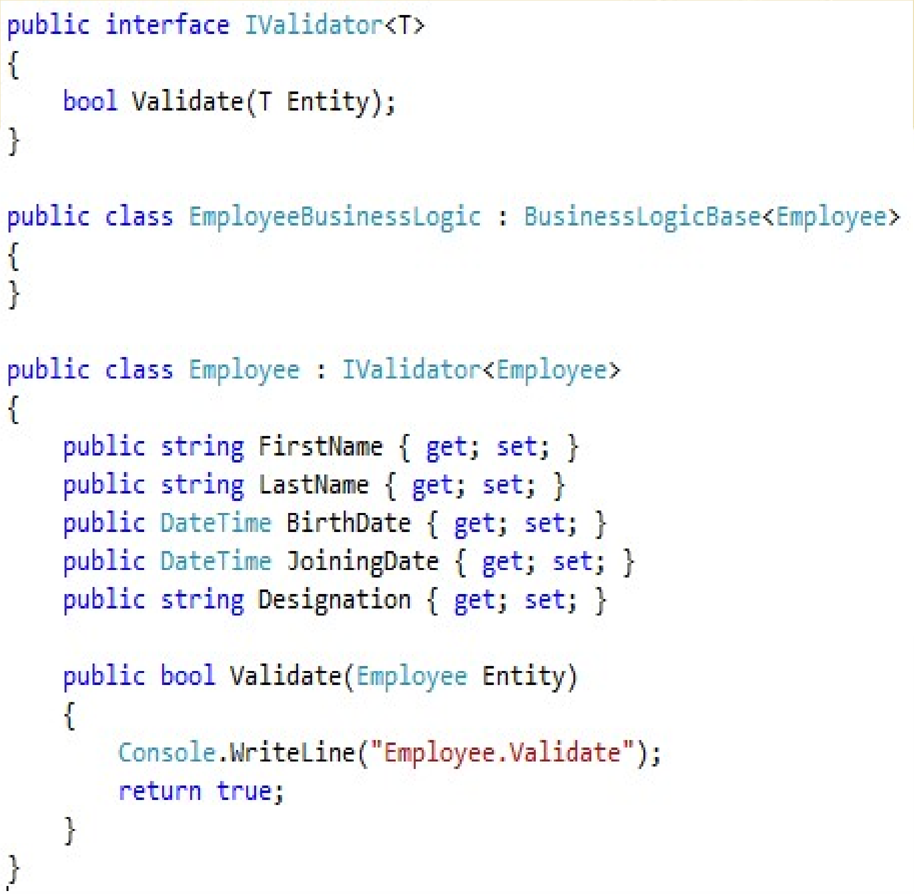


```
BusinessLogicBase.IsAuthorized
EmployeeBusinessLogic.ValidateEntity
BusinessLogicBase.ConvertEntityToSql
BusinessLogicBase.ExecuteSql
Press any key to continue . . .
```

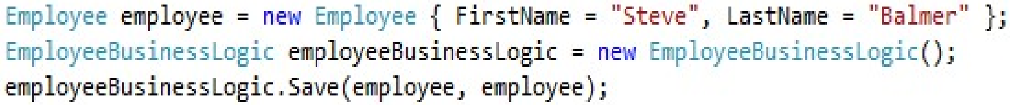
Alternate Template Method Implementation (using Interface)



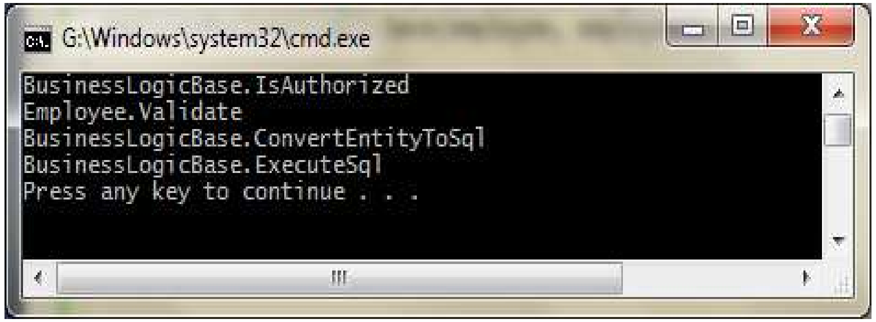
```
public class BusinessLogicBase<T>
{
    public void Save(T entity, IValidator<T> validator)
    {
        if (!this.IsAuthorized(entity))
            throw new Exception("User not authorized");

        if (!validator.Validate(entity))
            throw new Exception("Enity is not valid");

        string sql = this.ConvertEntityToSql(entity);
        this.ExecuteSql(sql);
    }

    protected virtual bool IsAuthorized(T entity)
    {
        Console.WriteLine("BusinessLogicBase.IsAuthorized");
        return true;
    }

    protected virtual string ConvertEntityToSql(T entity)
    {
        Console.WriteLine("BusinessLogicBase.ConvertEntityToSql");
        return "sql";
    }

    protected virtual void ExecuteSql(string sql)
    {
        Console.WriteLine("BusinessLogicBase.ExecuteSql");
    }
}
```



```
public interface IValidator<T>
{
    bool Validate(T Entity);
}

public class EmployeeBusinessLogic : BusinessLogicBase<Employee>
{
}

public class Employee : IValidator<Employee>
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public DateTime JoiningDate { get; set; }
    public string Designation { get; set; }

    public bool Validate(Employee Entity)
    {
        Console.WriteLine("Employee.Validate");
        return true;
    }
}
```



```
Employee employee = new Employee { FirstName = "Steve", LastName = "Balmer" };
EmployeeBusinessLogic employeeBusinessLogic = new EmployeeBusinessLogic();
employeeBusinessLogic.Save(employee, employee);
```

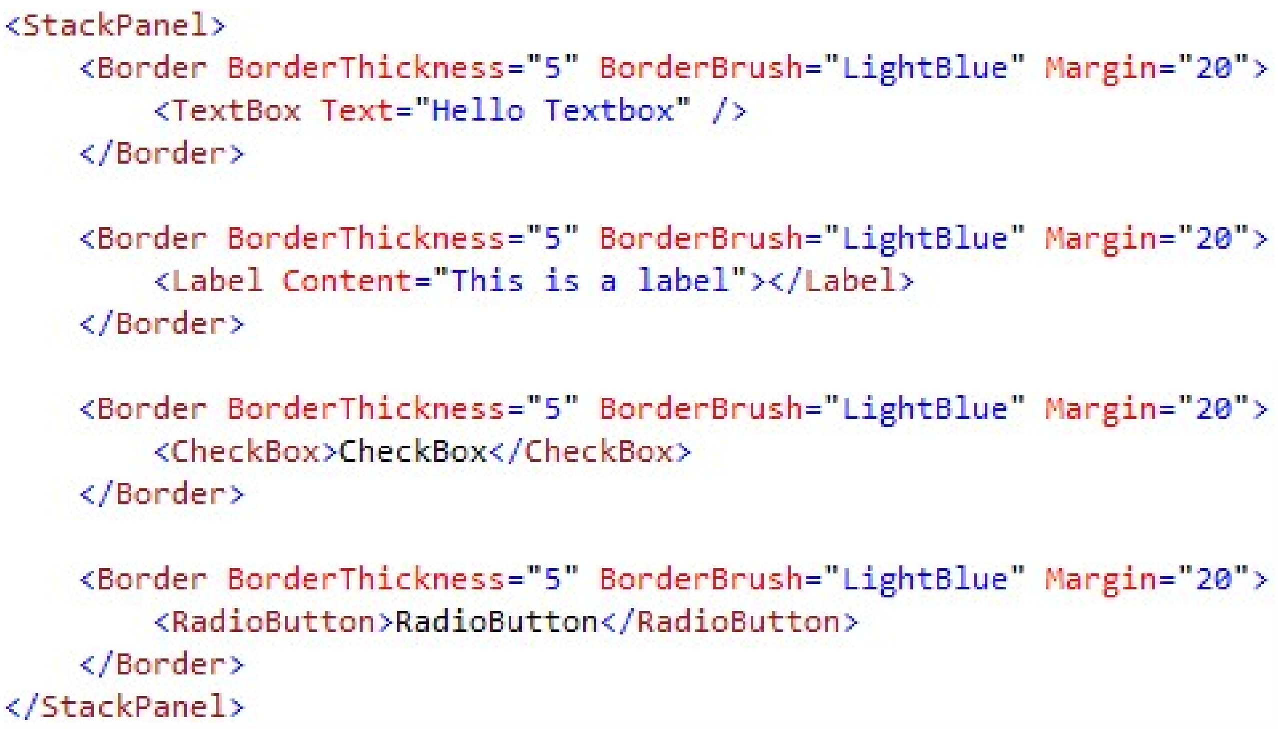


```
BusinessLogicBase.IsAuthorized
Employee.Validate
BusinessLogicBase.ConvertEntityToSql
BusinessLogicBase.ExecuteSql
Press any key to continue . . .
```

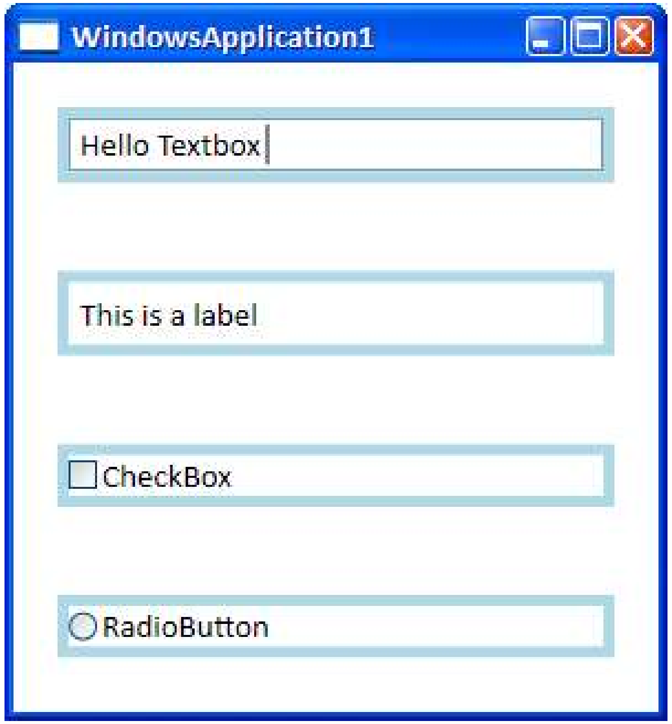
Decorator Pattern

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box ☺
- **You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.**

Decorator in WPF



```
<StackPanel>
    <Border BorderThickness="5" BorderBrush="LightBlue" Margin="20">
        <TextBox Text="Hello Textbox" />
    </Border>

    <Border BorderThickness="5" BorderBrush="LightBlue" Margin="20">
        <Label Content="This is a label"></Label>
    </Border>

    <Border BorderThickness="5" BorderBrush="LightBlue" Margin="20">
        <CheckBox>CheckBox</CheckBox>
    </Border>

    <Border BorderThickness="5" BorderBrush="LightBlue" Margin="20">
        <RadioButton>RadioButton</RadioButton>
    </Border>
</StackPanel>
```



WindowsApplication1

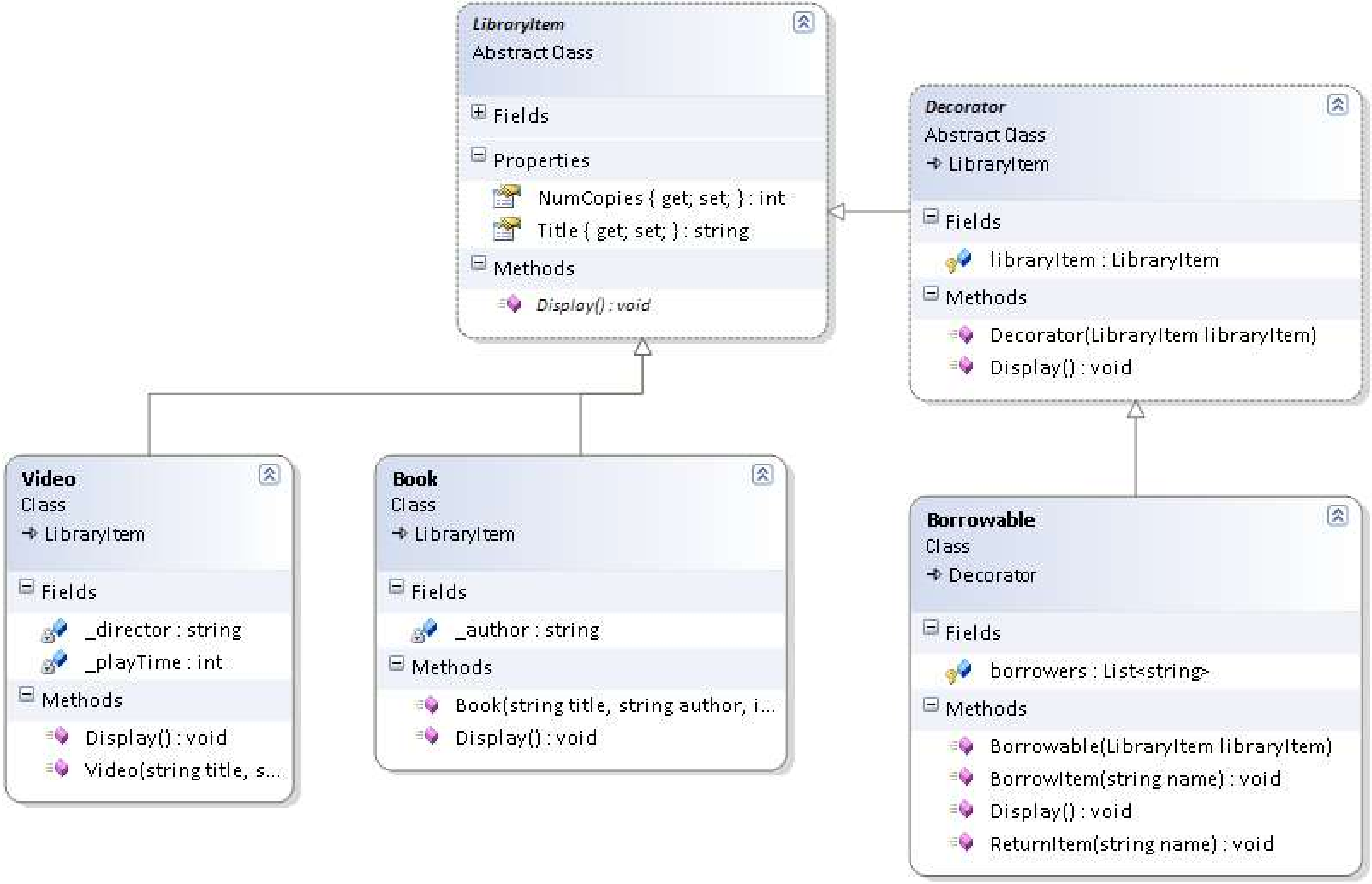
Hello Textbox

This is a label

□ CheckBox

○ RadioButton

Implementing Decorator in .NET



LibraryItem
Abstract Class

Fields

Properties

NumCopies { get; set; } : int

Title { get; set; } : string

Methods

Display() : void

Decorator
Abstract Class
→ LibraryItem

Fields

libraryItem : LibraryItem

Methods

Decorator(LibraryItem libraryItem)

Display() : void

Video
Class
→ LibraryItem

Fields

_director : string

_playTime : int

Methods

Display() : void

Video(string title, s...

Book
Class
→ LibraryItem

Fields

_author : string

Methods

Book(string title, string author, i...

Display() : void

Borrowable
Class
→ Decorator

Fields

borrowers : List<string>

Methods

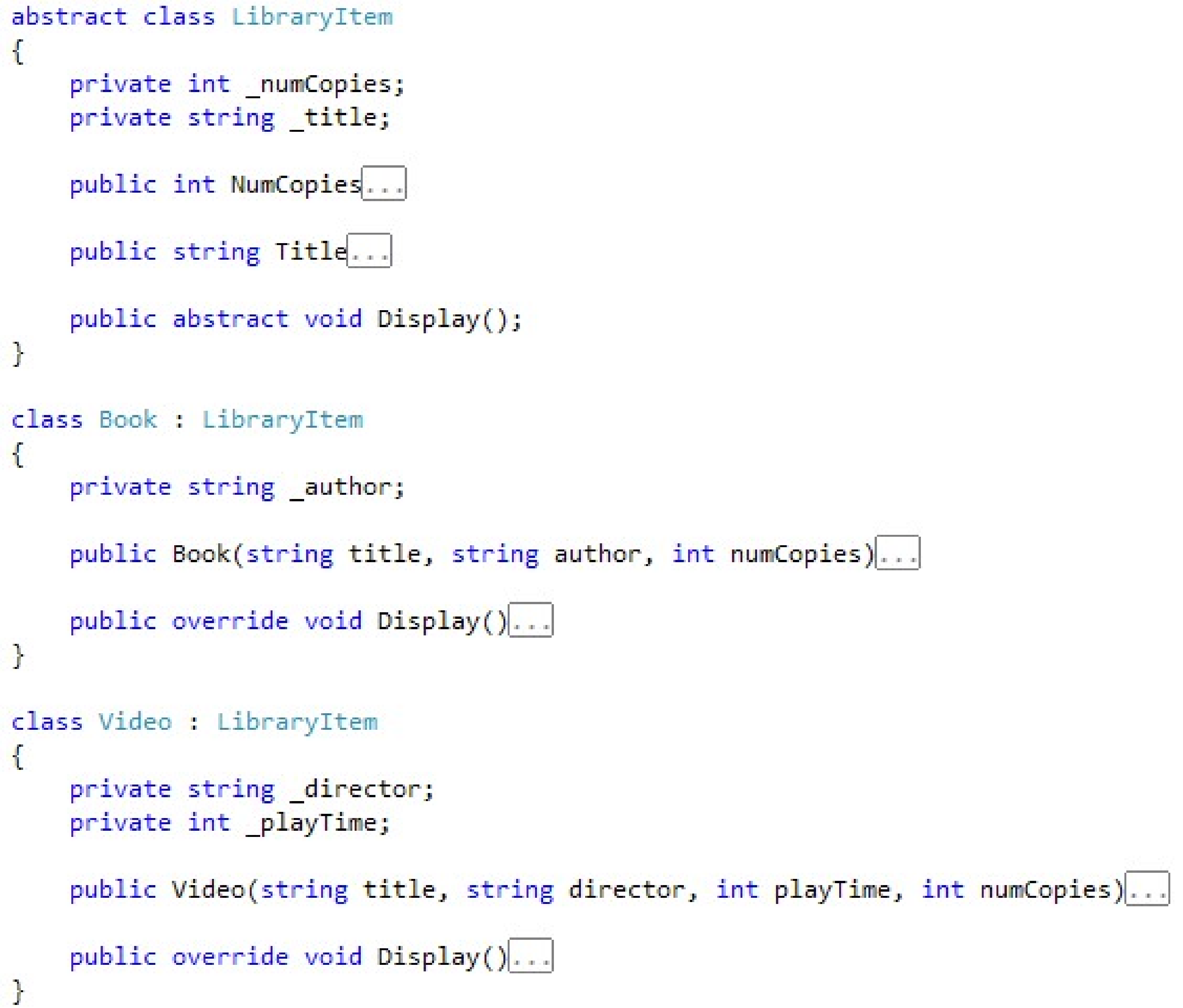
Borrowable(LibraryItem libraryItem)

BorrowItem(string name) : void

Display() : void

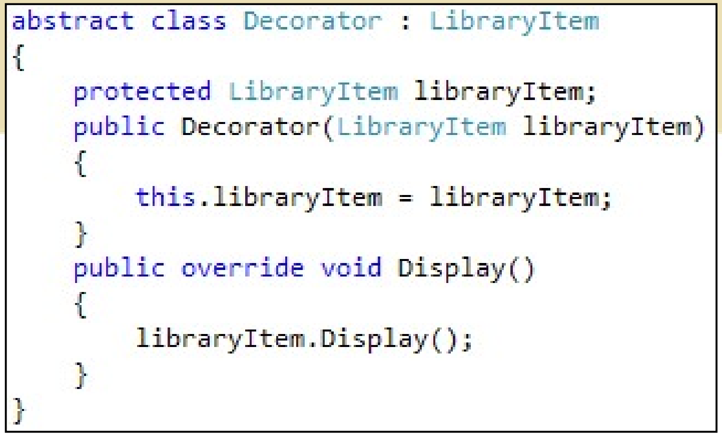
ReturnItem(string name) : void

Implementing Decorator in .NET (Cont’d)

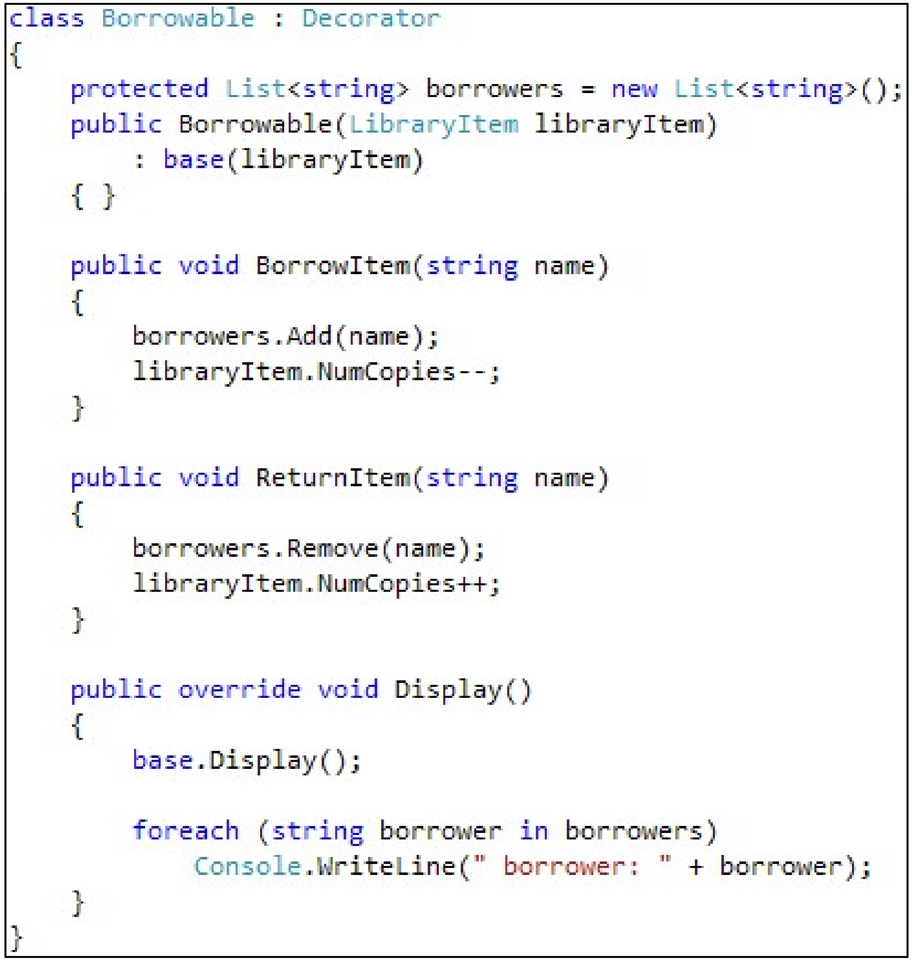


```
abstract class LibraryItem
{
    private int _numCopies;
    private string _title;

    public int NumCopies...

    public string Title...

    public abstract void Display();
}

class Book : LibraryItem
{
    private string _author;

    public Book(string title, string author, int numCopies)...

    public override void Display()...
}

class Video : LibraryItem
{
    private string _director;
    private int _playTime;

    public Video(string title, string director, int playTime, int numCopies)...

    public override void Display()...
}
```

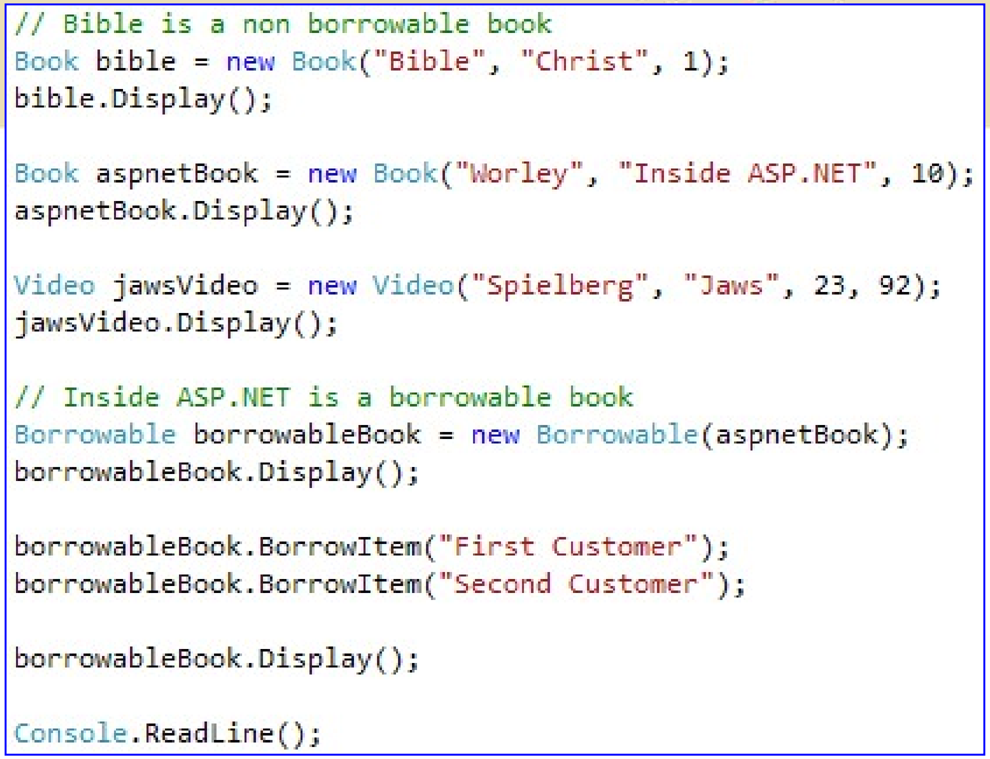
Implementing Decorator in .NET (Cont'd)



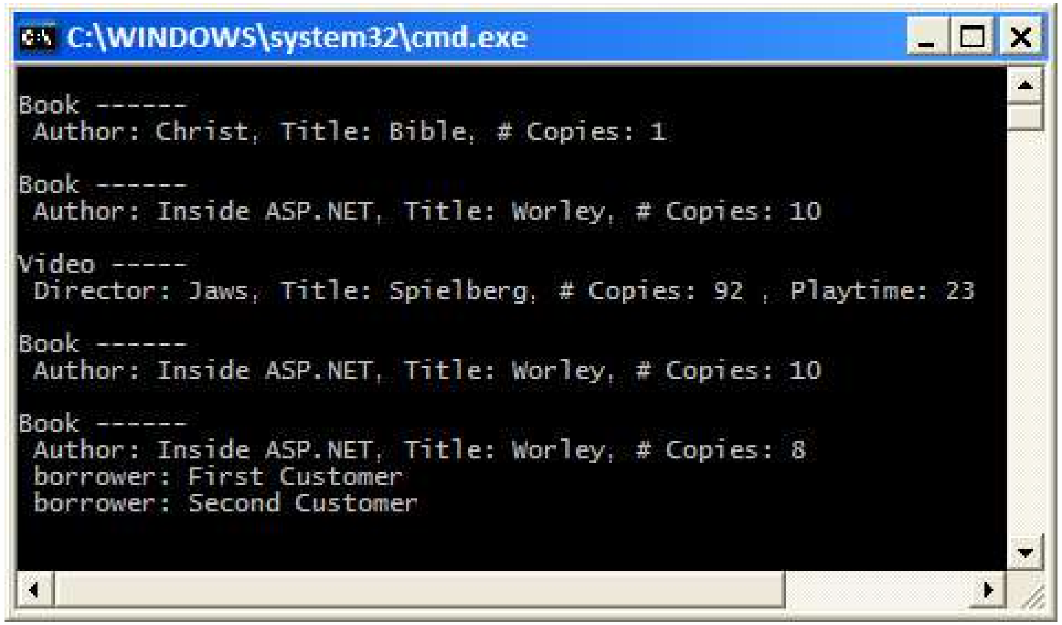
```
abstract class Decorator : LibraryItem
{
    protected LibraryItem libraryItem;
    public Decorator(LibraryItem libraryItem)
    {
        this.libraryItem = libraryItem;
    }
    public override void Display()
    {
        libraryItem.Display();
    }
}
```



```
class Borrowable : Decorator
{
    protected List<string> borrowers = new List<string>();
    public Borrowable(LibraryItem libraryItem)
        : base(libraryItem)
    { }

    public void BorrowItem(string name)
    {
        borrowers.Add(name);
        libraryItem.NumCopies--;
    }

    public void ReturnItem(string name)
    {
        borrowers.Remove(name);
        libraryItem.NumCopies++;
    }

    public override void Display()
    {
        base.Display();

        foreach (string borrower in borrowers)
            Console.WriteLine(" borrower: " + borrower);
    }
}
```



```
// Bible is a non borrowable book
Book bible = new Book("Bible", "Christ", 1);
bible.Display();

Book aspnetBook = new Book("Worley", "Inside ASP.NET", 10);
aspnetBook.Display();

Video jawsVideo = new Video("Spielberg", "Jaws", 23, 92);
jawsVideo.Display();

// Inside ASP.NET is a borrowable book
Borrowable borrowableBook = new Borrowable(aspnetBook);
borrowableBook.Display();

borrowableBook.BorrowItem("First Customer");
borrowableBook.BorrowItem("Second Customer");

borrowableBook.Display();

Console.ReadLine();
```



```
Book ------
 Author: Christ, Title: Bible, # Copies: 1

Book ------
 Author: Inside ASP.NET, Title: Worley, # Copies: 10

Video -----
 Director: Jaws, Title: Spielberg, # Copies: 92 , Playtime: 23

Book ------
 Author: Inside ASP.NET, Title: Worley, # Copies: 10

Book ------
 Author: Inside ASP.NET, Title: Worley, # Copies: 8
 borrower: First Customer
 borrower: Second Customer
```

- **UML Class Diagrams: Reference**
- https://msdn.microsoft.com/en-us/library/dd409437.aspx
- **Designing and Viewing Classes and Types**
- https://msdn.microsoft.com/en-us/library/ab7aty24.aspx